Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms

Anne Greenbaum[1]   Timothy P. Chartier

September 27, 2007

[1]These are course notes for Math 464/5/6, given Autumn 2006 through Spring 2007 at the University of Washington by Anne Greenbaum. Professor Greenbaum and Professor Chartier retain the copyright to these notes. They do not give anyone permission to copy computer files related to them. Send email to greenbau@math.washington.edu.

Contents

1 Basic Operations with MATLAB **6**
1.1 Launching MATLAB 7
1.2 Vectors 7
1.3 Getting help 10
1.4 Matrices 11
1.5 Creating and running `.m` files 12
1.6 Comments 12
1.7 Plotting 13
1.8 Creating your own functions 15
1.9 Printing 15
1.10 More loops and conditionals 17
1.11 Clearing variables 18
1.12 Logging your session 18
1.13 More advanced commands 19

2 Solution of a Single Nonlinear Equation in One Unknown **27**
2.1 Bisection 29
2.2 Taylor's Theorem 33
2.3 Newton's Method 35
2.4 Quasi-Newton Methods 41
2.4.1 Avoiding Derivatives 41
2.4.2 Constant Slope Method 42
2.4.3 Secant Method 42
2.5 Analysis of Fixed Point Methods 45
2.6 Fractals, Julia Sets and Mandelbrot Sets* 50

3 Floating Point Arithmetic **60**
3.1 Costly Disasters Caused by Rounding Errors 60
3.2 Binary Representation and Base 2 Arithmetic 63
3.3 Floating Point Representation 64
3.4 IEEE Floating Point Arithmetic 66
3.5 Rounding 68

3.6 Correctly Rounded Floating Point Operations . . . 69
3.7 Exceptions . . . 70

4 Conditioning of Problems; Stability of Algorithms 74
4.1 Conditioning of Problems . . . 74
4.2 Stability of Algorithms . . . 75

5 Solving Linear Systems and Least Squares Problems 80
5.1 Review of Matrix Algebra . . . 81
5.2 Gaussian Elimination . . . 81
5.2.1 Operation Counts . . . 85
5.2.2 LU Factorization . . . 87
5.2.3 Pivoting . . . 88
5.2.4 Banded Matrices and Matrices for which Pivoting is Not Required . . . 91
5.2.5 Implementation Considerations for High Performance* . . . 93
5.3 Other Methods for Solving $Ax = b$ . . . 96
5.4 Conditioning of Linear Systems . . . 98
5.4.1 Norms . . . 99
5.4.2 Sensitivity of Solutions of Linear Systems . . . 102
5.5 Stability of Gaussian Elimination with Partial Pivoting . . . 105
5.6 Least Squares Problems . . . 106
5.6.1 The Normal Equations . . . 107
5.6.2 QR Decomposition . . . 108
5.6.3 Fitting Polynomials to Data . . . 110

6 Polynomial and Piecewise Polynomial Interpolation 117
6.1 The Vandermonde System . . . 117
6.2 The Lagrange Form of the Interpolation Polynomial . . . 117
6.3 The Newton Form of the Interpolation Polynomial . . . 119
6.3.1 Divided Differences . . . 121
6.4 The Error in Polynomial Interpolation . . . 123
6.5 Piecewise Polynomial Interpolation . . . 126
6.5.1 Piecewise Cubic Hermite Interpolation . . . 128
6.5.2 Cubic Spline Interpolation . . . 129

7 Numerical Differentiation and Richardson Extrapolation 134
7.1 Numerical Differentiation . . . 134
7.2 Richardson Extrapolation . . . 139

8 Numerical Integration 144
8.1 Newton-Cotes Formulas . . . 144
8.2 Formulas Based on Piecewise Polynomial Interpolation . . . 148
8.3 Gaussian Quadrature . . . 150

8.3.1 Orthogonal Polynomials . . . 151
8.4 Romberg Integration . . . 155
8.5 Periodic Functions and the Euler-Maclaurin Formula . . . 156
8.6 Singularities . . . 160

9 Numerical Solution of the Initial Value Problem for Ordinary Differential Equations 164
9.1 Existence and Uniqueness of Solutions . . . 165
9.2 One-Step Methods . . . 168
9.2.1 Euler's Method . . . 168
9.2.2 Higher Order Methods Based on Taylor Series . . . 172
9.2.3 Midpoint Method . . . 173
9.2.4 Methods Based on Quadrature Formulas . . . 174
9.2.5 Classical Fourth-Order Runge-Kutta and Runge-Kutta-Fehlberg Methods. . . 175
9.2.6 Analysis of One-Step Methods . . . 176
9.2.7 Practical Implementation Considerations . . . 179
9.2.8 Systems of Equations . . . 180
9.3 Multistep Methods . . . 181
9.3.1 Adams-Bashforth and Adams-Moulton Methods . . . 181
9.3.2 General m-Step Methods . . . 183
9.3.3 Linear Difference Equations . . . 186
9.3.4 The Dahlquist Equivalence Theorem . . . 189
9.4 Stiff Equations . . . 189
9.4.1 Absolute Stability . . . 191
9.4.2 Backward Differentiation Formulae (BDF Methods) . . . 195
9.4.3 Implicit Runge-Kutta (IRK) Methods . . . 196
9.5 Solving Systems of Nonlinear Equations in Implicit Methods . . . 196
9.5.1 Fixed Point Iteration . . . 197
9.5.2 Newton's Method . . . 198

10 More Numerical Linear Algebra: Eigenvalues, Singular Values, and Iterative Methods for Solving Linear Systems 205
10.1 Eigenvalue Problems . . . 205
10.1.1 The Power Method for Computing the Largest Eigenpair . . . 212
10.1.2 Inverse Iteration . . . 215
10.1.3 Rayleigh Quotient Iteration . . . 216
10.1.4 The QR Algorithm . . . 217
10.1.5 Google's PageRank . . . 219

11 Numerical Solution of Two-Point Boundary Value Problems 221
11.1 An Application: Steady-State Temperature Distribution . . . 221
11.2 Finite Difference Methods . . . 222
11.2.1 Accuracy . . . 224

11.2.2 More General Equations and Boundary Conditions 229
11.3 Finite Element Methods 233
11.3.1 Accuracy 238

12 Numerical Solution of Partial Differential Equations 243
12.1 Elliptic Equations 244
12.1.1 Finite Difference Methods 245
12.1.2 Finite Element Methods 249
12.2 Parabolic Equations 250
12.2.1 Semidiscretization and the Method of Lines 251
12.2.2 Discretization in Time 251
12.3 Separation of Variables 259
12.3.1 Separation of Variables for Difference Equations 262
12.4 Hyperbolic Equations 263
12.4.1 Characteristics 263
12.4.2 Systems of Hyperbolic Equations 265
12.4.3 Boundary Conditions 266
12.4.4 Finite Difference Methods 266
12.5 Fast Methods for Poisson's Equation 270
12.5.1 The Fast Fourier Transform 272
12.6 Multigrid Methods 275

Chapter 1

Basic Operations with MATLAB

This book is concerned with the understanding of algorithms for problems of continuous mathematics. Part of this understanding includes the ability to implement such algorithms. To avoid distracting implementation details, however, we would like to accomplish this implementation in the simplest way possible, even if it is not necessarily the very most efficient. One system in which algorithm implementation is especially easy is called MATLAB [19] (short for MATrix LABoratory).

While a helpful academic and instructive tool, MATLAB is used in industry and government, as well. For instance, systems engineers at the NASA Jet Propulsion Laboratory used MATLAB to understand system behavior before launching the Mars Exploration Rover (MER) spacecraft into space.

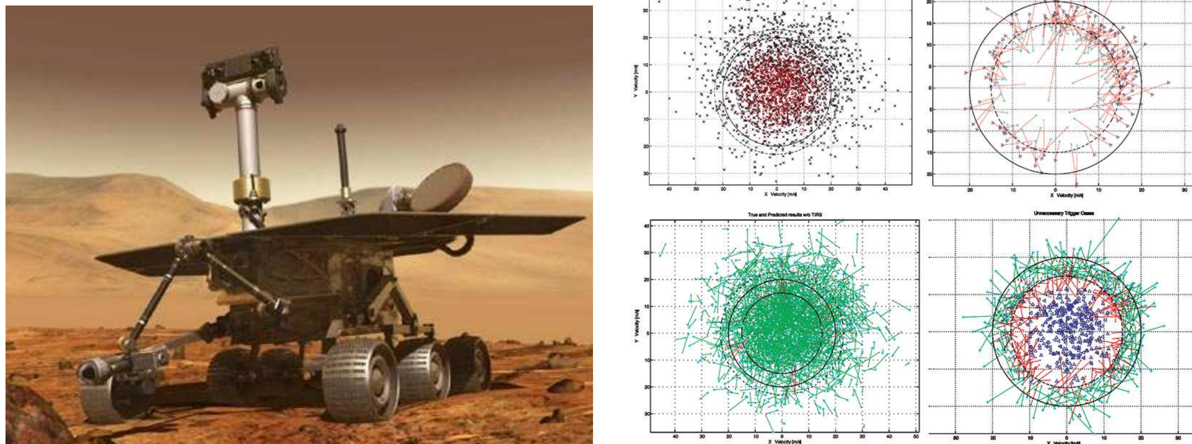


Figure 1.1: (Left) An artist's concept of the Mars rover. (Right) Custom statistical MATLAB visualizations that were used to predict how the onboard systems would respond under various atmospheric conditions during descent to the Mars surface.

This chapter contains a short description of basic MATLAB commands, and more commands

are described in programs throughout the book. For further information about using MATLAB, we recommend [11].

1.1 Launching MATLAB

MATLAB is a high level programming language that is especially well-suited to linear algebra type computations, but it can be used for almost any numerical problem. Following are some of the basic features of MATLAB that you will need to carry out the programming exercises in this book. Depending on what system you are using, you will start MATLAB either by double clicking on a MATLAB icon or by typing 'matlab' or by some similar means. When MATLAB is ready for input from you it will give a prompt such as `>>`.

You can use MATLAB like a calculator. For instance, if you type at the prompt:

```
>> 1+2*3
```

then MATLAB returns with the answer:

```
ans =

     7
```

Since you did not give a name to your result, MATLAB stores the result in a variable called 'ans'. You can do further arithmetic using the result in 'ans':

```
>> ans/4
```

and MATLAB will return with the result:

```
ans =

    1.7500
```

1.2 Vectors

MATLAB can store row or column vectors. The following commands:

```
>>  v = [1; 2; 3; 4]

v =

     1
     2
     3
     4
```

```
>> w = [5, 6, 7, 8]

w =

     5     6     7     8
```

create a column vector **v** of length 4 and a row vector **w** of length 4. In general, when defining a matrix or vector, semicolons are used to separate rows, while commas or spaces are used to separate the entries within a row. You can refer to an entry of a vector by giving its index:

```
>> v(2)

ans =

     2

>> w(3)

ans =

     7
```

MATLAB can add two vectors of the same dimension, but it cannot add **v** and **w** because **v** is 4×1 and **w** is 1×4. If you try to do this, MATLAB will give an error message:

```
>> v+w
??? Error using ==> +
Matrix dimensions must agree.
```

The transpose of **w** is denoted **w'**:

```
>> w'

ans =

     5
     6
     7
     8
```

You can add **v** and **w'** using ordinary vector addition:

```
>> v + w'

ans =
```

```
     6
     8
    10
    12
```

Suppose you wish to compute the sum of the entries in **v**. One way to do this is the following:

```
>> v(1) + v(2) + v(3) + v(4)

ans =

    10
```

Another way is to use a **for** loop:

```
>> sumv = 0;
>> for i=1:4, sumv = sumv + v(i); end;
>> sumv

sumv =

    10
```

This code initializes the variable **sumv** to 0. It then loops through each value $i = 1, 2, 3, 4$ and replaces the current value of **sumv** with that value plus **v(i)**. The line with the `for` statement actually contains three separate MATLAB commands. It could have been written in the form:

```
for i=1:4
  sumv = sumv + v(i);
end
```

MATLAB allows one line to contain multiple commands, provided they are separated by commas or semicolons. Hence in the one-line version of the **for** loop, we had to put a comma (or a semicolon) after the statement `for i=1:4`. This could have been included in the three-line version as well, but it is not necessary. Note also that in the three-line version, we have *indented* the statement(s) inside the **for** loop. This is not necessary, but it is a good programming practice. It makes it easy to see which statements are inside and which are outside the **for** loop. Note that the statement `sumv = 0` is followed by a semicolon, as is the statement `sumv = sumv + v(i)` inside the **for** loop. Following a statement by a semicolon suppresses printing of the result. Had we not put the semicolon at the end of the first statement, MATLAB would have printed out the result: **sumv = 0**. Had we not put a semicolon after the statement **sumv = sumv + v(i)**, then each time through the **for** loop, MATLAB would have printed out the current value of **sumv**. In a loop of length 4, this might be acceptable; in a loop of length 4 million, it probably would not be! To see the value of **sumv** at the end, we simply type **sumv** without a semicolon and MATLAB prints out its value.

1.3 Getting help

Actually, the entries of a vector are most easily summed using a builtin MATLAB function called **sum**. If you are unsure of how to use a MATLAB function or command, you can always type: **help** followed by the command name, and MATLAB will provide an explanation of how the command works:

```
>> help sum

 SUM Sum of elements.
    For vectors, SUM(X) is the sum of the elements of X. For
    matrices, SUM(X) is a row vector with the sum over each
    column. For N-D arrays, SUM(X) operates along the first
    non-singleton dimension.

    SUM(X,DIM) sums along the dimension DIM.

    Example: If X = [0 1 2
                     3 4 5]

    then sum(X,1) is [3 5 7] and sum(X,2) is [ 3
                                              12];

    See also PROD, CUMSUM, DIFF.
```

In general, you can type **help** in MATLAB and receive a summary of the classes of commands for which help is available. If you are not sure of the command name for which you are looking, there are two other helpful commands. First, typing **helpdesk** displays the help browser, which is a very *helpful* tool. Second, if you are interested in commands related to summing, you can type **lookfor sum**. With this command, MATLAB searches for the specified keyword "sum" in all help entries. This command results in the following:

```
>> lookfor sum
TRACE  Sum of diagonal elements.
CUMSUM Cumulative sum of elements.
SUM Sum of elements.
SUMMER Shades of green and yellow colormap.
UIRESUME Resume execution of blocked M-file.
UIWAIT Block execution and wait for resume.
RESUME Resumes paused playback.
RESUME Resumes paused recording.
```

It may take some searching to find precisely the topic and/or command that you are looking for.

1.4 Matrices

MATLAB also works with matrices:

```
>> A = [1, 2, 3; 4, 5, 6; 7, 8, 0]

A =

     1     2     3
     4     5     6
     7     8     0

>> b = [0; 1; 2]

b =

     0
     1
     2
```

There are many builtin functions for solving matrix problems. For example, to solve the linear system $Ax = b$, type `A\b`:

```
>> x = A\b

x =

    0.6667
   -0.3333
    0.0000
```

Note that the solution is printed out to only four decimal places. It is actually stored to about sixteen decimal places. (See Chapter **??**.) To see more decimal places, you can type:

```
>> format long
>> x

x =

    0.66666666666667
   -0.33333333333333
    0.00000000000000
```

Other options include `format short e` and `format long e` to display numbers using scientific notation.

You can check this answer by typing `b - A*x`. The notation `A*x` denotes standard matrix-vector multiplication, and standard vector subtraction is used when you subtract the result from **b**. This should give a vector of 0's, if **x** solves the system exactly. Since the machine carries only about 16 decimal digits, we do not expect it to be exactly zero, but, as we will see later, it should be just a moderate size multiple of 10^{-16}:

```
>> format short
>> b - A*x

ans =

   1.0e-15 *

   -0.0740
   -0.2220
         0
```

This is a good result!

1.5 Creating and running `.m` files

Typing MATLAB commands at the keyboard is fine if you are doing a computation once and will never need to make modifications and run it again. Once you `exit` MATLAB, however, all of the commands that you typed may be lost. To save the MATLAB commands that you type so that they can be executed again, you must enter them into a file called *filename*`.m`. Then, in MATLAB, if you type *filename*, it will run the commands from that file. The `.m` file can be produced using any text editor, such as the one that comes up as part of the MATLAB window. Once you save this file, it will be available for future use.

1.6 Comments

Adding documentation in your MATLAB code allows you and others to maintain your code for future use. Many a programmer has coded what appears to be a crystal clear implementation of an algorithm and later returned to be lost in the listing of commands. Comments can help to alleviate this problem. Adding comments to Matlab code is easy. Simply adding a % makes the remaining portion of that line a comment. In this way, you can have an entire line be a comment as in:

```
% Solve Ax=b
```

or you can append a comment after a MATLAB command, as in:

```
x = A\b;  % This solves the linear system Ax=b and stores the result in x.
```

The text following the % is simply a comment for the programmer and is ignored by MATLAB during runtime.

1.7 Plotting

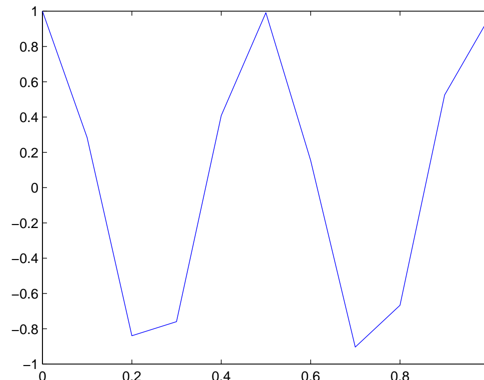
Tables and figures are usually more helpful than long strings of numerical results. Suppose you are interested in viewing a plot of $\cos(50x)$ for $0 \leq x \leq 1$. You can create two vectors, one consisting of x-values and the other consisting of corresponding $y = \cos(50x)$ values, and use the MATLAB `plot` command. To plot the values of $\cos(50x)$ at $x = 0,\ 0.1,\ 0.2,\ \ldots, 1$, type

```
>> x = 0:0.1:1;    % Form the (row) vector of x values.
>> y = cos(50*x); % Evaluate cos(50*x) at each of the x values.
>> plot(x,y)       % Plot the result.
```

Note that the statement `x = 0:0.1:1` behaves just like the `for` loop:

```
>> for i=1:11, x(i) = 0.1*(i-1); end;
```

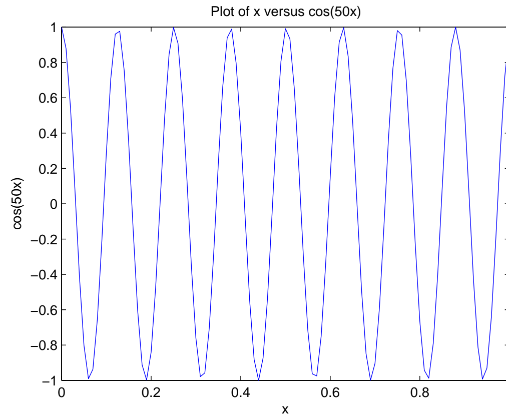
Note also that, unless otherwise specified, each of these statements produces a *row* vector. In order to produce a column vector, one could replace `x(i)` in the above **for** loop by `x(i,1)`, or one could replace the statement in the original code by `x = [0:0.1:1]';`. Clearly, this small number of evaluation points will not produce very high resolution, as is seen in the plot produced by MATLAB depicted below.



The plot command will graph the points $(x(1), y(1)), (x(2), y(2)), \ldots$, which also implies that x and y must be vectors of the same length. We did not use enough points to achieve a high resolution. Therefore, we will change x to include more points and include a title and labels for the axes in the plot with the commands below:

```
>> x = 0:0.01:1;      % Create a vector of 101 x values.
>> plot(x,cos(50*x)) % Plot x vs cos(50*x), without (explicitly) storing cos(50*x).
>> title('Plot of x versus cos(50x)')
>> ylabel('cos(50x)')
>> xlabel('x')
```

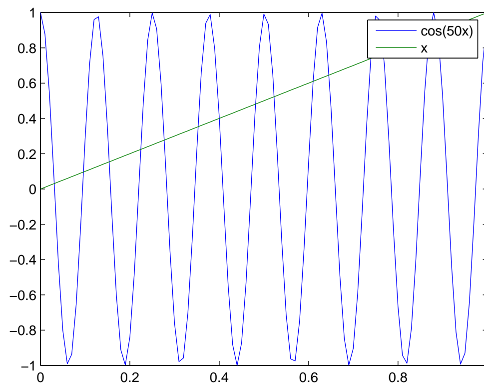
The plot resulting from these commands is:



It is also possible to plot more than one function on the same graph. To plot the two functions $f(x) = \cos(x)$ and $g(x) = x$ on the same graph, type:

```
>> plot(x,cos(50*x),x,x)
```

The result is shown below.



Notice the small boxed legend on the plot. This was added to the plot with the following command:

```
legend('cos(50x)','x')
```

Another way to plot two functions on the same graph is to first plot one, then type `hold on`, and then plot the other. If you do not type `hold on`, then the second plot will replace the first, but this command tells MATLAB to keep plots on the screen after they are created. To go back to the default of removing old plots before new ones are added, type `hold off`. Again, type `help plot` for more information on plotting or type `doc plot` which posts the Helpdesk documentation on the `plot` command. Other useful commands are `axis` and `plot3`. For a bit of fun, type the following commands:

```
>> x = 0:0.001:10;
>> comet(x,cos(3*x))
```

For more information on options available for plots, type the commands `hndlgraf`, `hndlaxis` and `ardemo`.

1.8 Creating your own functions

You can create your own functions to use in MATLAB. There are two options for this. If your function is simple (e.g., $f(x) = x^2 + 2x$), then you may enter it using the command `inline`:

```
>> f = inline('x.^2 + 2*x')

f =

     Inline function:
     f(x) = x.^2 + 2*x
```

Note the `.^2` notation. The expression `x^2` produces the square of `x` if `x` is a scalar, but it gives an error message if `x` is a vector, since standard vector multiplication is defined only if the inner dimensions of the vectors are the same (i.e., the first vector is 1 by n and the second is n by 1 or the first is n by 1 and the second is 1 by m). The operation `.^` applied to a vector, however, squares each entry individually. Since we may wish to evaluate the function at each entry of a vector of x values, we must use the `.^` operation. To evaluate f at the integers between 0 and 5, type:

```
>> f([0:5])

ans =

     0     3     8    15    24    35
```

If the function is more complicated, you may create a file whose name ends in `.m` which tells MATLAB how to compute the function. Type `help function` to see the format of the function file. Our function here could be computed using the following file (called `f.m`):

```
function [output] = f(x)
output = x.^2 + 2*x;
```

This function is called from MATLAB in the same way as above, i.e. `f(x)`, where `x` can be a scalar or vector.

1.9 Printing

While graphical output can be an important visualization tool, numerical results are often presented in tables. There are several ways to print output to the screen in MATLAB. First, to simply display a variable's contents, use the command `display`.

```
>> x = 0:.5:2;
>> display(x)
x =
         0    0.5000    1.0000    1.5000    2.0000
```

In many cases, the extra carriage return imposed by the `display` command clutters printed results. Therefore, another helpful command is `disp`, which is similar to the `display` command.

```
>> disp(x)
         0    0.5000    1.0000    1.5000    2.0000
```

You may still wish to have the variable name printed. Concatenating an array of text for output accomplishes this purpose.

```
>> disp(['x = ',num2str(x)])
x = 0         0.5           1         1.5           2
```

For more information, type `help num2str`.

Tables can be created such as the following:

```
>> disp('    Score 1   Score 2   Score 3'), disp(rand(5,3))
    Score 1   Score 2   Score 3
    0.4514    0.3840    0.6085
    0.0439    0.6831    0.0158
    0.0272    0.0928    0.0164
    0.3127    0.0353    0.1901
    0.0129    0.6124    0.5869
```

You may find the `fprintf` command easier to use for tables of results. For instance, consider the simple loop:

```
>> fprintf('   x          sqrt(x)\n=====================\n')
for i=1:5, fprintf('%f     %f\n',i,sqrt(i)), end
   x          sqrt(x)
=====================
1.000000     1.000000
2.000000     1.414214
3.000000     1.732051
4.000000     2.000000
5.000000     2.236068
```

The `fprintf` command takes format specifiers and variables to be printed in those formats. The `%f` format indicates that a number will be printed in fixed point format in that location of the line, and the `\n` forces a carriage return after the two quantities `i` and `sqrt(i)` are printed. You can specify the total field width and the number of places to be printed after the decimal point by replacing `%f` by, say, `%8.4f` to indicate that the entire number is to be printed in 8 spaces, with 4 places printed after the decimal point. You can send your output to a file instead of the screen by typing:

```
fid = fopen('sqrt.txt','w');
fprintf(fid,'   x      sqrt(x)\n=================\n');
for i=1:5, fprintf(fid,'%4.0f     %8.4f\n',i,sqrt(i)); end
```

which prints the following table in a file called `sqrt.txt`:

```
   x      sqrt(x)
=================
   1       1.0000
   2       1.4142
   3       1.7321
   4       2.0000
   5       2.2361
```

Again, for more information, refer to MATLAB documentation.

1.10 More loops and conditionals

We have already seen how `for` loops can be used in MATLAB to execute a set of commands a given number of times. Suppose, instead, that one wishes to execute the commands until some condition is satisfied. For example, one might approximate a root of a given function `f(x)` by first plotting the function on a coarse scale where one can see the approximate root, then plotting appropriate sections on finer and finer scales until one can identify the root to the precision needed. This can be accomplished with the following MATLAB code:

```
xmin = input(' Enter initial xmin: ');
xmax = input(' Enter initial xmax: ');
tol = input(' Enter tolerance: ')
while xmax-xmin > tol,
  x = [xmin:(xmax-xmin)/100:xmax];
  y = f(x);
  plot(x,y)
  xmin = input(' Enter new value for xmin: ');
  xmax = input(' Enter new value for xmax: ');
end;
```

The user looks at each plot to determine a value **xmin** that is just left of the root and a value **xmax** that is just right of the root. The next plot then contains only this section, so that closer values **xmin** and **xmax** can be determined. In the next chapter we discuss more efficient ways of finding a root of **f(x)**.

Another important statement is the conditional `if` statement. In the above code segment, one might wish to let the user know if the code happens to find a point at which the absolute value of **f** is less than some other tolerance, say, **delta**. This could be accomplished by inserting the following lines after the statement `y = f(x);`:

```
[ymin,index] = min(abs(y));  % This finds the minimum absolute value of y
                             % and its index.
if ymin < delta,
  fprintf(' f( %f ) = %f\n', x(index), y(index))  % This prints the x and y
                                                  % values at this index.
end;
```

The `if` statement may also contain an `else` clause. For example, to additionally write a message when **ymin** is greater than or equal to **delta**, one could modify the above **if** statement to say:

```
if ymin < delta,
  fprintf(' f( %f ) = %f\n', x(index), y(index))  % This prints the x and y
                                                  % values at this index.
else
  fprintf(' No points found where |f(x)| < %f\n', delta)
end;
```

1.11 Clearing variables

You may clear a particular variable by typing

```
>> clear x
```

or all variables with

```
>> clear all
```

This is important when you want to be sure that all variable names have been erased from memory.

1.12 Logging your session

You can keep a record of your MATLAB session by typing

```
    >> diary('hw1.txt')
    ... some other commands ...
    >> diary off
```

This command records all subsequent commands that you type and all responses that MATLAB returns in a file named `hw1.txt`. You will want to name the file by replacing `hw1.txt` with a more descriptive name related to your work. Note, however, that you **cannot** then run the file from MATLAB; this is simply a device for recording what happened during your keyboard session.

Note, if you execute an M-file in a session logged with the `diary` command, you may want to type "`echo on`" before executing the M-file. In this way, the commands in the M-file are echoed along with MATLAB's response. Otherwise, the diary file will contain only the responses, not the commands.

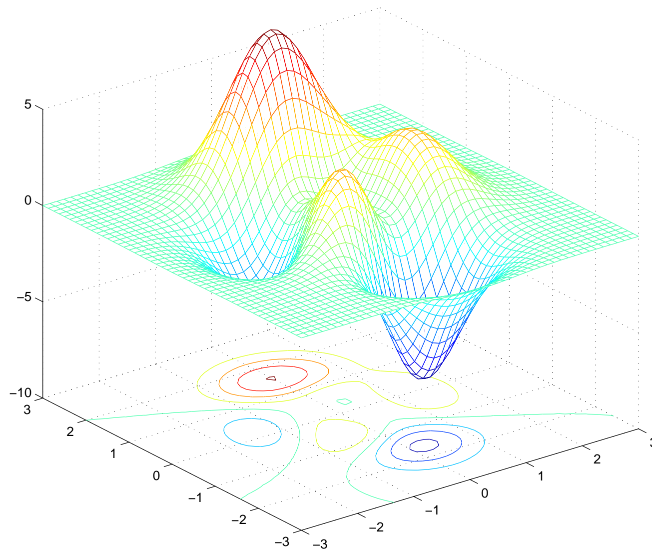
1.13 More advanced commands

It is perhaps apparent from the reference at the beginning of this chapter to the work on the Mars Exploration Rover, that MATLAB has a large number of commands. As we close this chapter, let us demonstrate some of the graphical capabilities through an example.

The commands

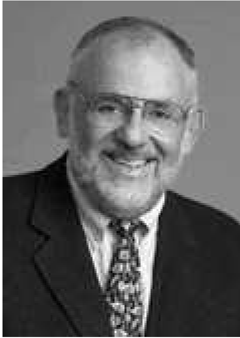
```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
meshc(X,Y,Z);
axis([-3 3 -3 3 -10 5])
```

produce the plot:



In order to understand this plot, search MATLAB documentation for the commands `meshgrid`, `peaks`, `meschc`, and `axis`. While this chapter will get you started using MATLAB, effective use of the MATLAB documentation will be the key to proceeding to more complicated programs.

MATLAB'S ORIGINS



<u>Cleve Moler</u> – is chairman and chief scientist at *The MathWorks*. In the mid to late 1970s, he was one of the authors of LINPACK and EISPACK, Fortran libraries for numerical computing. To give easy access to these libraries to his students at the University of New Mexico, Dr. Moler invented MATLAB. In 1984, he co-founded The MathWorks with Jack Little to commercialize the MATLAB program.
Cleve Moler received his bachelor's degree from Caltech in 1961, and a Ph.D. from Stanford University. Moler was a professor of math and computer science for almost 20 years at the University of Michigan, Stanford University and the University of New Mexico. Before joining The MathWorks full time in 1989, he also worked for Intel Hypercube and Ardent.
Dr. Moler was elected to the National Academy of Engineering in 1997. He has received honorary degrees from Linköping University, Sweden, the University of Waterloo and the Technical University of Denmark.

Exercises

1. Run the examples in this chapter using MATLAB to be sure that you see the same results.

2. With the following matrices and vectors:

$$A = \begin{pmatrix} 10 & -3 \\ 4 & 2 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ -1 & 2 \end{pmatrix}, \quad v = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad w = \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

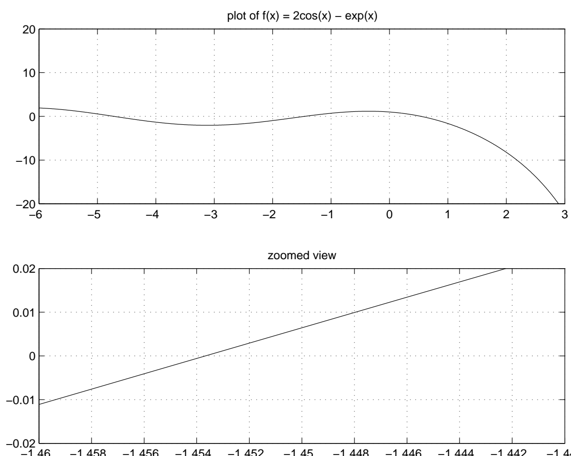
   compute the following **both** by hand and in MATLAB.

   (a) $v^T w$
   (b) vw^T
   (c) Av
   (d) $A^T v$
   (e) AB
   (f) BA
   (g) A^2 $(= AA)$
   (h) The vector y for which $By = w$.
   (i) The vector x for which $Ax = v$.

3. Use MATLAB to produce a single plot displaying the graphs of the functions $\sin(kx)$ across $[0, 2\pi]$, for $k = 1, \ldots, 5$.

4. Use MATLAB to print a table of values x, $\sin(x)$, and $\cos(x)$, for $x = 0, \frac{\pi}{6}, \frac{2\pi}{6}, \ldots, 2\pi$. Label the columns of your table.

5. Download the file `plotfunction1.m` for the web page for this class and execute it. This should produce the two plots shown below. The top plot shows the function $f(x) = 2\cos(x) - e^x$ for $-6 \le x \le 3$, and from this plot it appears that $f(x)$ has three roots in this interval. The bottom plot is a zoomed view near one of these roots, showing that $f(x)$ has a root near $x = -1.454$. Note the different vertical scale on this plot. Note also that when we zoom in on this function it looks nearly *linear* over this short interval. This will be important when we study numerical methods for approximating roots.



   (a) Modify this script so that the bottom plot shows a zoomed view near the leftmost root. Write an estimate of the value of this root to at least 3 decimal places. You may find it useful to first use the zoom feature in MATLAB to see approximately where the root is and then to choose your axis command for the second plot appropriately.

   (b) Edit the script from part (a) to plot the function

$$f(x) = \frac{4x\sin(x) - 3}{2 + x^2}$$

   over the range $0 \le x \le 4$ and also plot a zoomed view near the leftmost root. Write an estimate of the value of the root from the plots that is accurate to 3 decimal places. Once you have defined the vector `x` properly, you will need to use appropriate componentwise multiplication and division to evaluate this expression:

```
y = (4*x.*sin(x) - 3) ./ (2 + x.^2);
```

6. Plot each of the functions below over the range specified. Produce 4 plots on the same page using the `subplot` command.

   (a) $f(x) = |x - 1|$ for $-3 \le x \le 3$. (Use `abs` in MATLAB)

   (b) $f(x) = \sqrt{|x|}$ for $-4 \le x \le 4$. (Use `sqrt` in MATLAB)

   (c) $f(x) = e^{-x^2} = \exp(-x^2)$ for $-4 \le x \le 4$. (Use `exp` in MATLAB)

(d) $f(x) = \dfrac{1}{10x^2+1}$ for $-2 \le x \le 2$.

7. Use MATLAB to plot the circles

$$(x-2)^2 + (y-1)^2 = 2$$
$$(x-2.5)^2 + y^2 = 3.5$$

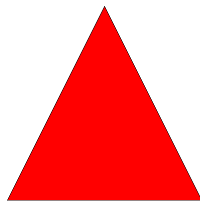
and zoom in on the plot to determine approximately where the circles intersect.

Hint: One way to plot the first circle is the following:

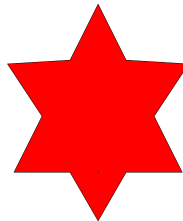
```
theta = linspace(0, 2*pi, 1000);
r = sqrt(2);
x = 2 + r*cos(theta);
y = 1 + r*sin(theta);
plot(x,y)
axis equal   % so the circles look circular!  (Could also use: axis square)
```

Use the command `hold on` after this to keep this circle on the screen while you plot the second circle in a similar manner.

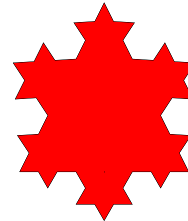
8. In this exercise, you will plot initial stages of a process that creates a fractal known as *Koch's Snowflake*, which is depicted below.



Stage 0



Stage 1



Stage 2

This exercise uses the MATLAB M-file `koch.m`, which you will find on the course's webpage. The M-file contains all the necessary commands to create the fractal, except for the necessary plotting commands. Edit this M-file so that each stage of the fractal is plotted. (Hint – you will need to add a plot command before the for-loop and also in the for-loop.) Add the following commands to keep consistency between plots in the animation:

```
axis([-0.75 0.75 -sqrt(3)/6 1]);
axis equal
```

Note, the `cla` command clears the axes. Finally, add the command `pause(0.5)` in appropriate places to slow the animation. (The `fill` command, as opposed to `plot`, produced the filled fractals depicted above.) We will create fractals using Newton's Method in the next chapter.

9. A magic square is an arrangement of the numbers from 1 to n^2 in an $n \times n$ matrix, where each number occurs exactly once, and the sum of the entries of any row, any column, or any main diagonal is the same. The MATLAB command `magic(n)` creates an $n \times n$ (where $n > 2$) magic square. Create a 5×5 magic square and verify using the `sum` command in MATLAB that the sum of the columns, rows and diagonals are equal. Create a log of your session that records your work. (Hint: To find the sum of the diagonals, read documentation for the `diag` and the `flipud` commands.)

10. More advanced plotting commands can be useful in MATLAB programming.

    (a) In the MATLAB command window type:

    ```
    [X,Y,Z] = peaks(30);
    surf(X,Y,Z);
    ```

    (b) Give this plot the title "3-D shaded surface plot"

    (c) Type `colormap hot` and observe the change in the plot.

    (d) Print the resulting plot with the given title.

11. Computer graphics make extensive use of matrix operations. For example, rotating an object is a simple matrix vector operation. In two dimensions, a curve can be rotated through an angle θ about the origin by multiplying every point that lies on the curve by the rotation matrix:

$$R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}.$$

As an example, let us rotate the square with vertices $[1, -.5]$, $[2, -.5]$, $[2, .5]$, and $[1, .5]$ through the angle $\theta = \pi/4$ about the origin. In MATLAB, type the following code to generate the original and rotated squares plotted one on top of the other.

```
% Create matrix whose first column contains coordinates of the first vertex,
% whose second column contains coordinates of the second vertex, etc.
U = [1 2 2 1; -.5 -.5 .5 .5];

% Create a red unit square
% Note U(1,:) creates a vector of the first row of U
fill(U(1,:),U(2,:),'r')

% Use axis equal to make square look square, then plot between -4 and 4
% so that rotated squares can be seen.  Retain current plot so that
% subsequent graphing commands add to the existing graph
axis('equal')
axis([-4 4 -4 4])
hold on
```

```
% Create rotation matrix.
theta = pi/4;
R = [cos(theta) -sin(theta); sin(theta) cos(theta)];

% Rotate each of the vertices and color rotated square blue.
U = R*U;
fill(U(1,:), U(2,:),'b');
```

Note that the `fill` command in MATLAB plots a filled polygon determined by two vectors of vertices.

(a) Continue rotating the square by applying the rotation matrix `R` to `U` over and over again and plotting the new squares in different colors. You should find that after applying `R` eight times, the new square lands on top of the original.

(b) Adapt this code to plot a triangle with vertices $(5, 0), (6, 2)$ and $(4, 1)$. Plot the triangles resulting from rotating this triangle $\pi/2, \pi$, and $3\pi/2$ radians about the origin. Plot all four triangles on the same set of axes.

(c) Rotating by θ radians and then again rotating by $-\theta$ radians leaves the figure unchanged. This corresponds to multiplying first by $R(\theta)$ and then by $R(-\theta)$. Using MATLAB, verify that these matrices are inverses of one another (that is, their product is the identity) for $\theta = \pi/3$ and $\pi/4$.

(d) Using the trigonometric identities $\cos(\theta) = \cos(-\theta)$ and $-\sin(\theta) = \sin(-\theta)$, prove that $R(\theta)$ and $R(-\theta)$ are inverses of one another for any θ.

12. In Figure 1.2, we see a well–known model in computer graphics known as the "Stanford Bunny." This triangulation is formed using 3851 triangles with 1889 vertices. Suppose V is a matrix with three columns where row i contains the x, y and z coordinate of the ith vertex in the model. The image can be rotated by θ radians about the y-axis by multiplying V by R_y where

$$R_y = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix}.$$

Download the file `bunny.zip` from the web page for this course. Unzip the file and run `bunny.m` in MATLAB. This will plot the triangulation of the Stanford Bunny.

(a) Edit the code to rotate the image by $\pi/6$ radians about the y-axis.

(b) The matrix containing the vertices is an 1889×3 matrix. How many (scalar) multiplications are performed when you use matrix multiplication (with R_y) to rotate the image by $\pi/6$ radians as you just did? Keep this in mind as you watch how fast MATLAB performs the calculations and plots the results.

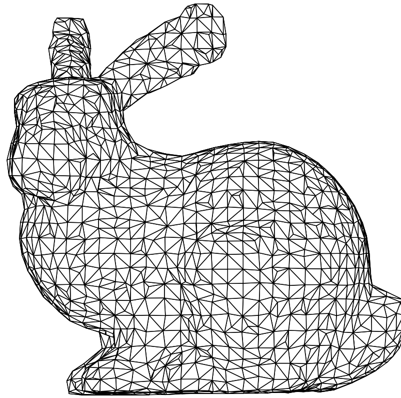


Figure 1.2: Triangulations are used in computer graphics to create models of objects in a computer. Here we see the triangulation of the well–known "Stanford Bunny" – Need to attain permission. Taken from http://graphics.stanford.edu/data/3Dscanrep/

(c) Further edit the code to continuously rotate the image by $\pi/24$ radians until the image has rotated one full rotation about the y axis. Before drawing a new rendering of the rotated image, you will want to clear the current plot and possibly include a short pause. These two steps can be performed with the following code:

```
cla;       % clear current plot
pause(0.5) % pause for 0.5 seconds
```

You may want to look for the "Rotate 3D" button in the Figure containing the triangulation. You can rotate the image using your mouse. Of special note, look at the bottom of the bunny and notice how you can look inside the model, as if it were a porcelain figurine.

13. Type `helpdesk` and find documentation on the `movie` command or alternatively type `doc movie`. At the bottom of the documentation on this command, you will find the following code:

```
Z = peaks;
surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
for j = 1:20
    surf(sin(2*pi*j/20)*Z,Z)
    F(j) = getframe;
end movie(F,5)
```

Cut and paste this code into the MATLAB command window and describe the results. Proficiency in MATLAB programming can increase dramatically through effective use of the available documentation.

Chapter 2

Solution of a Single Nonlinear Equation in One Unknown

Consider the problem facing an artillery officer during battle. As the sun rises, he sees the position of the advancing enemy army. Overnight their artillery has dug into a position d meters away.

The officer wishes to aim his cannon to hit the entrenched enemy.

He knows that the cannonball will leave the muzzle of the gun at an initial speed of v_0 meters per second. Neglecting air resistance, the only force acting on the cannonball after it leaves the cannon is gravity, and the height $y(t)$ of the cannonball at any time $t > 0$ satisfies the differential equation:

$$y''(t) = -g \approx -9.2 \text{ m/sec}^2.$$

To determine the height uniquely, one can specify the initial height and vertical velocity of the cannonball:

$$y(0) = 0 \text{ m}, \quad y'(0) = v_0 \sin\theta \text{ m/sec},$$

where θ is the angle at which the cannon is aimed. Note that the initial value $y(0) = 0$ is an approximation; the tip of the cannon muzzle surely isn't at ground level, but it is not far off.

One can solve this differential equation by integrating:

$$y'(t) = -gt + c_1,$$

and since $y'(0) = v_0 \sin\theta$, the constant c_1 must be $v_0 \sin\theta$. Integrating again gives

$$y(t) = -\frac{1}{2}gt^2 + v_0 \sin\theta \; t + c_2,$$

where $c_2 = 0$ since $y(0) = 0$.

Now that we have a formula for the height at any time $t > 0$, we can determine the amount of time before the cannonball hits the ground:

$$0 = -\frac{1}{2}gt^2 + v_0 \sin\theta \; t \quad \Rightarrow \quad t = 0 \;\text{ or }\; t = \frac{2v_0 \sin\theta}{g}.$$

Finally, neglecting air resistance, the cannonball travels at a constant horizontal velocity, $v_0 \cos\theta$, so that it hits the ground a distance

$$\frac{2v_0^2 \sin\theta \; \cos\theta}{g} \text{ m}$$

away from where it was fired. The problem thus becomes one of solving a nonlinear equation,

$$\frac{2v_0^2 \sin\theta \; \cos\theta}{g} = d, \tag{2.1}$$

for the angle θ at which to aim the cannon.

While one could solve this problem numerically, perhaps using an available software package for solving nonlinear equations, there are several things that one should take into account before returning to the cannoneer with a definitive value for θ.

1. The model developed above is only an approximation. Consequently, even if one returns with the exact solution (θ) to (2.1), the fired cannonball may end up missing the target. A strong tailwind might cause the officer's shot to sail over the approaching enemy. Likewise, the amount of gunpowder may vary between shots, which corresponds to v_0 in (2.1) changing for each shot. Such factors may make adjustments to the model necessary.

2. Equation (2.1) may not have a solution. The maximum value of $\sin\theta \; \cos\theta$ occurs at $\theta = \pi/4$, where $\sin\theta \; \cos\theta = \frac{1}{2}$. If $d > \frac{v_0^2}{g}$, then the problem has no solution. In this case, the enemy is simply out of range.

3. If a solution does exist, it may not be unique. If θ solves problem (2.1), then so does $\theta \pm 2k\pi$, $k = 1, 2, \ldots$. Also $\frac{\pi}{2} - \theta$ solves the problem. Either θ or $\frac{\pi}{2} - \theta$ might be preferable as a solution, depending on what lies in front of the cannon!

4. Actually this problem can be solved analytically. Using the trigonometric identity $2\sin\theta\cos\theta = \sin(2\theta)$, one finds
$$\theta = \frac{1}{2}\arcsin\frac{dg}{v_0^2}.$$

In light of item 1 above, however, it may be prudent to go ahead and write a code to handle the problem numerically, since more realistic models may require approximate solution by numerical methods. By a good *approximate* solution, we mean one that is acceptably close for the application under consideration. A computed angle that results in the cannonball landing within a foot of its target is probably acceptable in this situation, but if the equation arose from an application in laser surgery considerably more accuracy would likely be required.

5. To solve a nonlinear equation numerically, we usually start by writing it in the form $f(\theta) = 0$. Equation (2.1) can be written as

$$f(\theta) \equiv \frac{2v_0^2 \sin\theta \ \cos\theta}{g} - d = 0.$$

This function is simple enough that it can be differentiated analytically. We will see in the following sections that such functions can then be solved using Newton's method. If we made the model more realistic, however, this might not be the case. For example, if we included the effect of air resistance in our model, then we might be unable to solve the differential equation for $y(t)$ analytically. We could approximate its solution numerically, for a given value of θ. In this case, the problem could be solved using bisection, the secant method, or quasi-Newton methods, to be described in this chapter.

The point of this discussion is that if one blindly tries to solve a physical problem using a numerical software package, the results may be less than satisfactory. Consideration of the model being used, whether or not the problem has a solution and, if so, how many solutions and whether one solution is to be preferred to another, are important preprocessing steps. Likewise, consideration of the level of accuracy required in the result will be of importance.

2.1 Bisection

Faced with an oncoming army, the artillery officer might use the following procedure to solve the problem: He would guess the angle at which he should aim the cannon, fire it, and see where the cannonball lands. If it fails to hit the target, he would adjust the angle and try again. If he finds that at one angle θ_1 the cannonball goes too far and at another θ_2 it lands short, then he might try some angle between θ_1 and θ_2, say, their average $\frac{\theta_1+\theta_2}{2}$. This simple procedure is the method of *bisection*, and it can be used to solve any equation $f(x) = 0$, provided f is a continuous function of x and we have starting values x_1 and x_2 where f has different signs.

The basis for the bisection method is the Intermediate Value Theorem:

Theorem 2.1.1. *If f is continuous on $[a, b]$ and y lies between $f(a)$ and $f(b)$, then there is a point $x \in [a, b]$ where $f(x) = y$.*

If one of $f(a)$ and $f(b)$ is positive and the other negative, then there is a solution to $f(x) = 0$ in $[a, b]$. In the bisection algorithm, we start with an interval $[a, b]$ where $\mathrm{sign}(f(a)) \neq \mathrm{sign}(f(b))$. We then compute the midpoint $\frac{a+b}{2}$ and evaluate $f\left(\frac{a+b}{2}\right)$. We then replace the endpoint (a or b) where f has the same sign as $f\left(\frac{a+b}{2}\right)$ by $\frac{a+b}{2}$, and repeat.

BISECTING A PHONE BOOK

Have a friend choose a name at random from your local phone book. Bet your friend that in less than thirty yes or no questions, you will be able to guess the name that was chosen. This task is simple. Place about half of the pages in your left hand (or right if you prefer) and ask your friend, "Is the name in this half of the phone book?" In one question, you have eliminated approximately half of the names. Holding half of the non-eliminated pages in your hand, ask the question again. Repeat this process until you have discovered the page which contains the name. Continue to ask questions which halve the number of possible names until you have discovered the name.
Like the Bisection Algorithm, you use the speed of exponential decay to your advantage. Since $(1/2)^{30}$ is about 9.3×10^{-10} and the population of the United States is approximately 280 million, you can see that any city phone book will suffice (and should leave you a comfortable margin for error).

Following is a simple MATLAB code that implements the bisection algorithm:

```
%  This routine uses bisection to find a zero of a user-supplied
%  continuous function f.  The user must supply two points a and b
%  such that f(a) and f(b) have different signs.  The user also
%  supplies a convergence tolerance delta.

fa = f(a);
if fa==0, root = a; break; end;  % Check to see if a is a root.
fb = f(b);
if fb==0, root = b; break; end;  % Check to see if b is a root.

if sign(fa)==sign(fb),           % Check for an error in input.
  display('Error: f(a) and f(b) have same sign.')
  break
end;

while abs(b-a) > 2*delta,        % Iterate until interval width <= 2*delta.
  c = (a+b)/2;                   % Bisect interval.
  fc = f(c);                     % Evaluate f at midpoint.
  if fc==0, root = c; break; end; % Check to see if c is a root.

  if sign(fc)==sign(fa),         % Replace a by c, fa by fc.
    a = c;
```

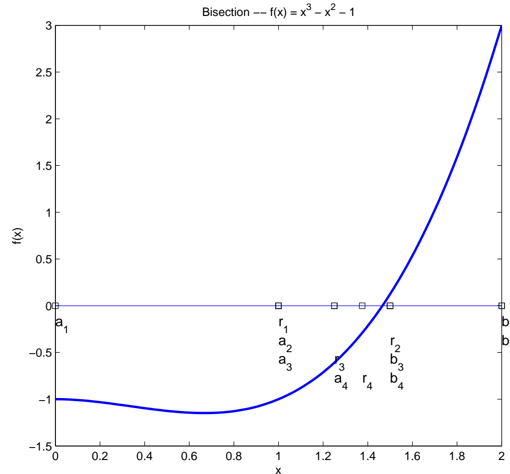
```
    fa = fc;
  else                              % Replace b by c, fb by fc.
    b = c;
    fb = fc;
  end;
end;

root = (a+b)/2;                     % Take midpoint as approximation to root.
```

There are several things to notice about this code. First, it checks to see if the user has made an error in the input and supplied two points where the sign of `f` is the same. When writing code for others to use, it is always good to check for errors in input. Of course, typing in such things is time-consuming for the programmer, so when we write codes for ourselves, we sometimes omit these things. (And sometimes live to regret it!) Next, it checks to see if it has happened upon an "exact" root, that is, a point where f evaluates to exactly zero. If this happens, it returns with that point as the computed root. This is not necessary; it could continue to bisect the interval until the interval width reaches `2*delta`. The MATLAB `sign` function returns 1 if the argument is positive, -1 if it is negative, and 0 if it is 0, so if `fc` were 0, the code would end up replacing `b` by `c`.

Example 1. Use bisection to find a root of $f(x) = x^3 - x^2 - 1$ in the interval $[a, b]$, where $a = 0$ and $b = 2$.

n	a_n	b_n	r_n	$f(r_n)$
1	0	2	1	-1
2	1	2	1.5	0.125
3	1	1.5	1.25	-0.609
4	1.25	1.5	1.375	-0.291
5	1.375	1.5	1.4375	-0.096
6	1.4375	1.5	1.46875	0.011
7	1.4375	1.46875	1.453125	-0.043
8	1.453125	1.46875	1.4609375	-0.016

A graphical depiction of these results is given below.



Rate of Convergence What can be said about the *rate* of convergence of the bisection algorithm? Since the interval size is reduced by a factor of 2 at each step, the interval size after k steps is $|b-a|/2^k$, which converges to zero as $k \to \infty$. To obtain an interval of size 2δ we need

$$\frac{|b-a|}{2^k} \leq 2\delta \;\Leftrightarrow\; 2^{k+1} \geq \frac{|b-a|}{\delta} \;\Leftrightarrow\; k \geq \log_2\left(\frac{|b-a|}{\delta}\right) - 1.$$

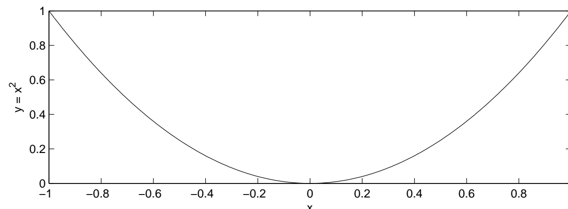
If the approximate root is taken to be the midpoint of this interval, then it differs from an actual root by at most δ.

Example 2. If $a = 1$ and $b = 2$ and we want to guarantee an error less than or equal to 10^{-4}, how many iterations do we need to take?

$$\frac{|b-a|}{2^k} \leq 2 \cdot 10^{-4} \quad\Leftrightarrow\quad 2^{k+1} \geq \frac{|b-a|}{10^{-4}} \quad\Leftrightarrow\quad k > \log_2\left(\frac{|b-a|}{10^{-4}}\right) - 1.$$

Since $b - a = 1$, then $k \geq 12.29$ iterations are required to guarantee an error less than or equal to 10^{-4}. After 13 iterations, your answer could have an error much less than 10^{-4} but will not have an error greater than 10^{-4}.

Since each step reduces the error (i.e., half the bracket size) by a constant factor (i.e., a factor of 2), the bisection algorithm is said to converge *linearly*. The most difficult part of using the bisection algorithm may be in finding an initial interval where the sign of f is different at the endpoints. Once this is found the algorithm is guaranteed to converge. Note that the bisection algorithm cannot be used for a problem like $x^2 = 0$, pictured below, since the sign of the function does not change.



2.2 Taylor's Theorem

Probably the *most-used* theorem in numerical analysis is Taylor's theorem, especially Taylor's theorem with remainder. It is worth reviewing this important result.

Brook Taylor (1685-1731)

Theorem 2.2.1 (Taylor's theorem with remainder:). *Let f, f' , ..., $f^{(n)}$ be continuous on $[a, b]$ and let $f^{(n+1)}(x)$ exist for all $x \in (a, b)$. Then there is a number $\xi \in (a, b)$ such that*

$$f(b) = f(a) + (b-a)f'(a) + \frac{(b-a)^2}{2!}f''(a) + \ldots + \frac{(b-a)^n}{n!}f^{(n)}(a) + \frac{(b-a)^{n+1}}{(n+1)!}f^{(n+1)}(\xi). \tag{2.2}$$

Formula (2.2) can also be written using the summation notation:

$$f(b) = \sum_{j=0}^{n} \frac{(b-a)^j}{j!}f^{(j)}(a) + \frac{(b-a)^{n+1}}{(n+1)!}f^{(n+1)}(\xi).$$

The remainder term in the Taylor series expansion of $f(x)$ about the point a,

$$R_n(x) \equiv f(x) - \sum_{j=0}^{n} \frac{(x-a)^j}{j!}f^{(j)}(a) = \frac{(x-a)^{n+1}}{(n+1)!}f^{(n+1)}(\xi),$$

is sometimes written as $O((x-a)^{n+1})$, since there is a finite constant C (namely, $f^{(n+1)}(a)/(n+1)!$) such that

$$\lim_{x\to a} \frac{R_n(x)}{(x-a)^{n+1}} = C.$$

Sometimes the $O(\cdot)$ notation is used to mean, in addition, that there is no higher power $j > n+1$ such that $\lim_{x\to a} R_n(x)/(x-a)^j$ is finite. This will be the case if the constant C above is nonzero.

If f is infinitely differentiable, then the Taylor series expansion of $f(x)$ about the point a is

$$\sum_{j=0}^{\infty} \frac{(x-a)^j}{j!} f^{(j)}(a).$$

Depending on the properties of f, the Taylor series may converge to f everywhere or throughout some interval about a or only at the point a itself. For very well-behaved functions like e^x (which we will also denote as $\exp(x)$), $\sin(x)$ and $\cos(x)$, the Taylor series converges everywhere. The Taylor series about $x = 0$ occurs often and is called a Maclaurin series.

The Taylor series contains an infinite number of terms. If we are interested only in the behavior of the function in a neighborhood around $x = a$, then only the first few terms of the Taylor series may be needed:

$$f(x) \approx P_n(x) = \sum_{j=0}^{n} \frac{(x-a)^j}{j!} f^{(j)}(a).$$

The *Taylor polynomial* $P_n(x)$ may serve as an approximation to the function.

Example 3. Find the Taylor series of $e^x = \exp(x)$ about 1. First, find the value of the function and its derivatives at $x = 1$:

$$\begin{aligned} f(1) &= \exp(1) \\ f'(1) &= \exp(1) \\ f''(1) &= \exp(1) \\ &\vdots \end{aligned}$$

Next, construct the series using these values and $a = 1$:

$$\begin{aligned} \exp(x) &= \underbrace{f(1)}_{=\exp(1)} + (x-1)\underbrace{f'(1)}_{=\exp(1)} + \frac{(x-1)^2}{2!}\underbrace{f''(1)}_{=\exp(1)} + \frac{(x-1)^3}{3!}\underbrace{f^{(3)}(1)}_{=\exp(1)} + \ldots \\ &= \exp(1)\left[1 + (x-1) + \frac{(x-1)^2}{2!} + \frac{(x-1)^3}{3!} + \ldots\right] \\ &= \exp(1)\sum_{k=0}^{\infty} \frac{(x-1)^k}{k!} \end{aligned}$$

Example 4. Find the Maclaurin series for $\cos(x)$. First, find the value of the function $f(x) = \cos(x)$ and its derivatives at $x = 0$:

$$\begin{array}{rclcl} f(0) & = & \cos(0) & = & 1 \\ f'(0) & = & -\sin(0) & = & 0 \\ f''(0) & = & -\cos(0) & = & -1 \end{array} \qquad \begin{array}{rclcl} f^{(3)}(0) & = & \sin(0) & = & 0 \\ f^{(4)}(0) & = & \cos(0) & = & 1 \\ & \vdots & & & \end{array}$$

Next, construct the series using these values and $a = 0$:

$$\begin{aligned} \cos(x) &= \underbrace{f(0)}_{=1} + x \underbrace{f'(0)}_{=0} + \frac{x^2}{2!} \underbrace{f''(0)}_{=-1} + \frac{x^3}{3!} \underbrace{f^{(3)}(0)}_{=0} + \frac{x^4}{4!} \underbrace{f^{(4)}(0)}_{=1} + \dots \\ &= 1 - \frac{x^2}{2} + \frac{x^4}{4!} + \dots = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!} \end{aligned}$$

Example 5. Find the fourth order *Taylor polynomial* $P_4(x)$ of $\cos(x)$ about $x = 0$. Using the work from the previous exercise, we find

$$P_4(x) = 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4$$

In the following section, Taylor's Theorem will be used to derive a numerical method. At the end of the chapter, exercise 18 discusses how John Machin, a contemporary of Brook Taylor, used Taylor's Theorem to find 100 decimal places of π.

2.3 Newton's Method

Isaac Newton, a founder of Calculus, described a method, which became known as *Newton's method* in his *Method of Fluxions and Infinite Series.* [6, 16]

Isaac Newton (1643-1727)

We start with a geometric derivation of Newton's method. Suppose we wish to solve $f(x) = 0$ and we are given an initial guess x_0 for the solution. We evaluate $f(x_0)$ and then construct the tangent line to f at x_0 and determine where it hits the x-axis. If f were linear, this would be the root of f. In general, it might give a better approximation to a root than x_0 does, as illustrated in Figures 2.1 (a) and (b). The slope of this tangent line is $f'(x_0)$, and it goes through the point $(x_0, f(x_0))$, so its equation is $y - f(x_0) = f'(x_0)(x - x_0)$. It hits the x-axis ($y = 0$) at $x = x_0 - f(x_0)/f'(x_0)$, assuming $f'(x_0) \neq 0$. Letting x_1 be the point where this tangent line hits the x-axis, this process can be repeated by constructing the tangent line to f at x_1, determining where it hits the x-axis and calling that point x_2, etc. This process is known as *Newton's method:*

Newton's Method: (For solving $f(x) = 0$)

Given an initial guess x_0, for $k = 0, 1, 2, \ldots,$

Set $x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)}$.

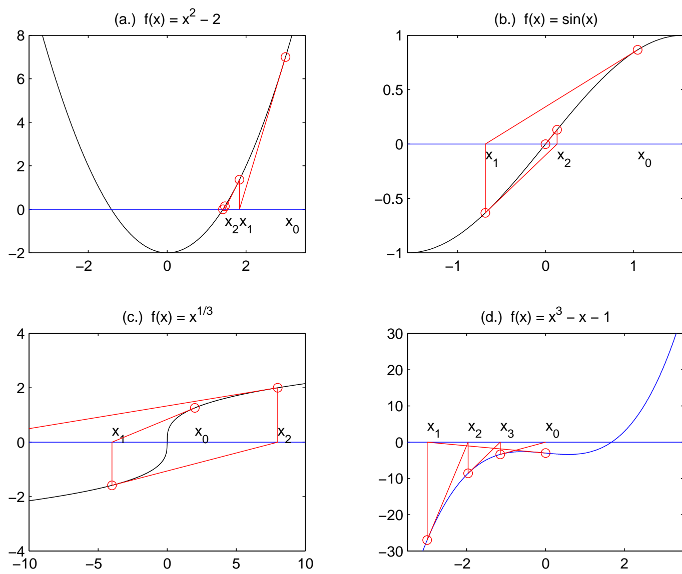


Figure 2.1: Newton's Method

As a historical note, Newton explained his method in 1669 using numerical examples and did not use the geometric motivation given here involving the approximation of a curve with its tangent line. His work also did not develop the recurrence relation given above, which was developed by the English mathematician Joseph Raphson in 1690. It is for this reason that the method is often called the *Newton-Raphson Method*.

Unfortunately, Newton's method does not always converge. Figures 2.1 (c) and (d) illustrate cases in which Newton's method fails. In the example of (c), Newton's method diverges, while in (d) it cycles between points, never approaching the actual zero of the function.

One might combine Newton's method with bisection, to obtain a more reliable algorithm. Once an interval is determined where the function changes sign, we know by the Intermediate Value Theorem that that interval contains a root. Hence if the next Newton step would land outside that interval, then do not accept it but do a bisection step instead. With this modification, the method is guaranteed to converge *provided* it finds an interval where the function has different signs at the endpoints. This combination would solve example (c) in Figure 2.1 because once x_0 and x_1 were determined, the next Newton iterate x_2 would be rejected and replaced by $(x_0 + x_1)/2$. Example (d) in Figure 2.1 would still fail because the algorithm does not locate any pair of points where the function has opposite signs.

Newton's method can also be derived from Taylor's theorem. According to that theorem,

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2} f''(\xi),$$

for some ξ between x_0 and x. If $f(x_*) = 0$, then

$$0 = f(x_0) + (x_* - x_0)f'(x_0) + \frac{(x_* - x_0)^2}{2} f''(\xi), \tag{2.3}$$

and if x_0 is close to the root x_*, then the last term involving $(x_* - x_0)^2$ can be expected to be small in comparison to the others. Neglecting this last term, and defining x_1, instead of x_*, to be the point that satisfies the equation when this term is omitted, we have:

$$0 = f(x_0) + (x_1 - x_0)f'(x_0),$$

or,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Repeating this process for $k = 0, 1, 2, \ldots$ gives

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \tag{2.4}$$

This leads to the following theorem about the convergence of Newton's method.

Theorem 2.3.1. *If $f \in C^2$, if x_0 is* sufficiently close *to a root x_* of f, and if $f'(x_*) \neq 0$, then Newton's method converges to x_*, and* ultimately *the convergence rate is* quadratic*; that is, there exists a constant $C_* = |f''(x_*)/(2f'(x_*))|$ such that*

$$\lim_{k \to \infty} \frac{|x_{k+1} - x_*|}{|x_k - x_*|^2} = C_*. \tag{2.5}$$

Before proving this theorem, we first make a few observations. The quadratic convergence rate described in (2.5) means that for k large enough, convergence will be very rapid. If C is any constant greater than C_*, it follows from (2.5) that there exists K such that for all $k \geq K$,

$$|x_{k+1} - x_*| \leq C|x_k - x_*|^2.$$

If, say, $C = 1$ and $|x_K - x_*| = 0.1$, then we will have $|x_{K+1} - x_*| \leq 10^{-2}$, $|x_{K+2} - x_*| \leq 10^{-4}$, $|x_{K+3} - x_*| \leq 10^{-8}$, etc. There are some important limitations, however. First, the theorem requires that the initial guess x_0 be *sufficiently close* to the desired root x_*, but it does not spell out exactly what *sufficiently close* means nor does it provide a way to check whether a given initial guess is sufficiently close to a desired root. The proof will make it clearer why it is difficult to say just how close is close enough. Second, it shows that the *ultimate* rate of convergence is quadratic, but it does not say how many steps of the iteration might be required before this quadratic convergence rate is achieved. This too will depend on how close the initial guess is to the solution.

Proof of Theorem. It follows from equation (2.3), with x_0 replaced by x_k, that

$$x_* = x_k - \frac{f(x_k)}{f'(x_k)} - \frac{(x_* - x_k)^2}{2} \frac{f''(\xi_k)}{f'(x_k)},$$

for some ξ_k between x_k and x_*. Subtracting this from equation (2.4) for x_{k+1} gives

$$x_{k+1} - x_* = \frac{f''(\xi_k)}{2f'(x_k)} \, (x_k - x_*)^2. \tag{2.6}$$

Now since f'' is continuous and $f'(x_*) \neq 0$, if $C_* = |f''(x_*)/(2f'(x_*))|$, then for any $C > C_*$, there is an interval about x_* in which $|f''(x)/(2f'(x))| \leq C$. If, for some K, x_K lies in this interval, and if also $|x_K - x_*| < 1/C$ (which is sure to hold, even for $K = 0$, if x_0 is sufficiently close to x_*), then (2.6) implies that $|x_{K+1} - x_*| \leq C|x_K - x_*|^2 < |x_K - x_*|$, so that x_{K+1} also lies in this interval and satisfies $|x_{K+1} - x_*| < 1/C$. Proceeding by induction, we find that all iterates x_k, $k \geq K$, lie in this interval and so, by (2.6), satisfy

$$\begin{aligned}
|x_{k+1} - x_*| &\leq C|x_k - x_*|^2 \\
&\leq (C|x_k - x_*|) \cdot |x_k - x_*| \\
&\leq (C|x_k - x_*|)(C|x_{k-1} - x_*|) \cdot |x_{k-1} - x_*| \\
&\vdots \\
&\leq (C|x_k - x_*|) \ldots (C|x_K - x_*|) \cdot |x_K - x_*| \\
&\leq (C|x_K - x_*|)^{k+1-K} |x_K - x_*|.
\end{aligned}$$

Since $C|x_K - x_*| < 1$, it follows that $(C|x_K - x_*|)^{k+1-K} \to 0$ as $k \to \infty$, and this implies that $x_k \to x_*$ as $k \to \infty$. Finally, since x_k and hence ξ_k in (2.6) converge to x_*, it follows from (2.6) that

$$\frac{|x_{k+1} - x_*|}{|x_k - x_*|^2} = \left| \frac{f''(\xi_k)}{2f'(x_k)} \right| \to C_*.$$

□

As noted previously, the hypothesis of x_0 being "sufficiently close" to a root x_* (where "sufficiently close" can be taken to mean that x_0 lies in a neighborhood of x_* where $|f''/(2f')|$ is less than or equal to some constant C and $|x_0 - x_*| < 1/C$) is usually impossible to check since one does not know the root x_*. There are a number of other theorems giving different sufficient conditions on the initial guess to guarantee convergence of Newton's method, but as with this theorem, the conditions are usually difficult or impossible to check, or they may be far more restrictive than necessary, so we do not include these results here. Early mathematicians looking at conditions under which Newton's method converges include Mourraille in 1768 and later Lagrange. In 1818, Fourier looked at the question of the rate of convergence and in 1821 and again in 1829 Cauchy addressed this question. For more information about the historical development of the method, see [6].

Examples:

1. Compute $\sqrt{2}$ using Newton's method; i.e., solve $x^2 - 2 = 0$.

   Since $f(x) = x^2 - 2$, we have $f'(x) = 2x$, and Newton's method becomes

   $$x_{k+1} = x_k - \frac{x_k^2 - 2}{2x_k}, \quad k = 0, 1, \ldots .$$

   Starting with $x_0 = 2$, and letting e_k denote the error $x_k - \sqrt{2}$, we find:

$x_0 = 2$	$e_0 = 0.59$
$x_1 = 1.5$	$e_1 = 0.086$
$x_2 = 1.4167$	$e_2 = .0025$
$x_3 = 1.4142157$	$e_3 = 2.1e-6 \approx .35e_2^2$

   The constant $|f''(x_*)/(2f'(x_*))|$ for this problem is $1/(2\sqrt{2}) \approx 0.3536$. Note that for this problem Newton's method fails if $x_0 = 0$. For $x_0 > 0$ it converges to $\sqrt{2}$ and for $x_0 < 0$ it converges to $-\sqrt{2}$.

2. Suppose $f'(x_*) = 0$. Newton's method may still converge, but only linearly. Consider the problem $f(x) \equiv x^2 = 0$. Since $f'(x) = 2x$, Newton's method becomes

   $$x_{k+1} = x_k - \frac{x_k^2}{2x_k} = \frac{1}{2}x_k.$$

   Starting, for example, with $x_0 = 1$, we find that $e_0 = 1$, $x_1 = e_1 = \frac{1}{2}$, $x_2 = e_2 = \frac{1}{4}$, …, $x_k = e_k = \frac{1}{2^k}$. Instead of squaring the error in the previous step, each step reduces the error by a factor of 2.

3. Consider the problem $f(x) \equiv x^3 = 0$. Since $f'(x) = 3x^2$, Newton's method becomes

   $$x_{k+1} = x_k - \frac{x_k^3}{3x_k^2} = \frac{2}{3}x_k.$$

   Clearly the error (the difference between x_k and the true root $x_* = 0$) is multiplied by the factor $\frac{2}{3}$ at each step.

4. Suppose $f(x) \equiv x^j$. What is the error reduction factor for Newton's method?

To understand example 2, recall formula (2.6) from the proof of Theorem 2.1:

$$x_{k+1} - x_* = \frac{f''(\xi_k)}{2f'(x_k)}(x_k - x_*)^2.$$

Since $f'(x_*) = 0$ in example 2, the term multiplying $(x_k - x_*)^2$ becomes larger and larger as x_k approaches x_*, so we cannot expect quadratic convergence. However, if we expand $f'(x_k)$ about x_* using a Taylor series, we find

$$f'(x_k) = f'(x_*) + (x_k - x_*)f''(\eta_k) = (x_k - x_*)f''(\eta_k),$$

for some η_k between x_* and x_k. Substituting this expression for $f'(x_k)$ above gives

$$x_{k+1} - x_* = \frac{f''(\xi_k)}{2f''(\eta_k)}(x_k - x_*).$$

If $f''(x_*) \neq 0$, then this establishes linear convergence (under the assumption that x_0 is sufficiently close to x_*) with the convergence factor approaching $\frac{1}{2}$. By retaining more terms in the Taylor series expansion of $f'(x_k)$ about x_*, this same approach can be used to explain examples 3 and 4.

Theorem 2.3.2. *If $f \in C^{p+1}$ for $p \geq 1$, if x_0 is sufficiently close to a root x_* of f, and if $f'(x_*) = \ldots = f^{(p)}(x_*) = 0$ but $f^{(p+1)}(x_*) \neq 0$, then Newton's method converges linearly to x_*, with the error ultimately being reduced by about the factor $p/(p+1)$ at each step; that is,*

$$\lim_{k \to \infty} \frac{|x_{k+1} - x_*|}{|x_k - x_*|} = \frac{p}{p+1}.$$

Proof. Taking formula (2.4) and subtracting x_* from each side gives

$$x_{k+1} - x_* = x_k - x_* - \frac{f(x_k)}{f'(x_k)}. \tag{2.7}$$

Now expanding $f(x_k)$ and $f'(x_k)$ about x_* gives

$$f(x_k) = \frac{(x_k - x_*)^{p+1}}{(p+1)!} f^{(p+1)}(\xi_k),$$

$$f'(x_k) = \frac{(x_k - x_*)^p}{p!} f^{(p+1)}(\eta_k),$$

for some points ξ_k and η_k between x_k and x_*, since the first $p+1$ terms in the first Taylor series expansion and the first p terms in the second are zero. Making these substitutions in (2.7) gives

$$x_{k+1} - x_* = (x_k - x_*)\left(1 - \frac{1}{p+1}\frac{f^{(p+1)}(\xi_k)}{f^{(p+1)}(\eta_k)}\right). \tag{2.8}$$

Since $f^{(p+1)}(x_*) \neq 0$ and $f^{(p+1)}$ is continuous, for any $\epsilon > 0$, there is an interval about x_* in which the factor $1 - (1/(p+1))f^{(p+1)}(\xi)/f^{(p+1)}(\eta)$ is within ϵ of its value at $\xi = \eta = x_*$ (namely, $1 - 1/(p+1) = p/(p+1)$), whenever ξ and η lie in this interval. In particular, for ϵ sufficiently small, this factor will be bounded above by a constant $C < 1$. It follows from (2.8) that if x_k lies in this interval, then $|x_{k+1} - x_*| \leq C|x_k - x_*|$, so x_{k+1} also lies in this interval, and, if x_0 lies in this interval then, by induction, $|x_{k+1} - x_*| \leq C^{k+1}|x_0 - x_*|$, so $x_k \to x_*$ as $k \to \infty$. The limiting value of the error reduction factor is $p/(p+1)$. □

As another example, consider the functions $f_1(x) = \sin(x)$ and $f_2(x) = \sin^2(x)$, both of which have $x_* = \pi$ as a root. Note that $f_1'(x) = \cos(x) \neq 0$ at $x_* = \pi$, while $f_2'(x) = 2\sin(x)\cos(x) = 0$ at $x_* = \pi$. Note, however, that $f_2''(\pi) \neq 0$. Therefore, we expect quadratic and linear convergence (with a convergence factor of about 1/2) for Newton's method applied to $f_1(x)$ and $f_2(x)$, respectively. This is seen graphically in Figure 2.2. For f_1, the root $x_* = \pi$ is said to have multiplicity 1 or to be a simple root, while for f_2 it is a root of multiplicity 2. Newton's method converges only linearly for a root of multiplicity greater than 1.

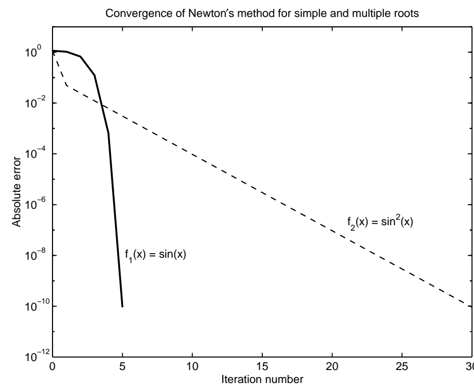


Figure 2.2: Convergence of Newton's method for a function with root of multiplicity 1 versus a function with root of multiplicity 2.

2.4 Quasi-Newton Methods

2.4.1 Avoiding Derivatives

Newton's method has the *very* nice property of quadratic convergence, when sufficiently close to a root. This means that one can hone in on a root from a nearby point very quickly, much faster than with the bisection method, for example. The price is that one must evaluate both f and its derivative at each step. For the simple examples presented here, this is not difficult. In many problems, however, the function f is quite complicated. It might not be one that can be written down analytically. Instead, one might have to run a program to evaluate $f(x)$. In such cases, differentiating f analytically is difficult to impossible, and, even if a formula can be found, it may

be very expensive to evaluate. For such problems, one would like to avoid computing derivatives or, at least, compute as few of them as possible, while maintaining something close to quadratic convergence. Iterations of the form:

$$x_{k+1} = x_k - \frac{f(x_k)}{g_k}, \quad \text{where } g_k \approx f'(x_k), \tag{2.9}$$

are sometimes called *quasi-Newton* methods.

2.4.2 Constant Slope Method

One idea is to evaluate f' once at x_0 and then to set g_k in (2.9) equal to $f'(x_0)$ for all k. If the slope of f does not change much as one iterates, then one might expect this method to mimic the behavior of Newton's method. This method is called the *constant slope method*:

$$x_{k+1} = x_k - \frac{f(x_k)}{g}, \quad \text{where } g = f'(x_0). \tag{2.10}$$

To analyze this iteration, we once again use Taylor's theorem. Expanding $f(x_k)$ about the root x_*, we find

$$f(x_k) = (x_k - x_*)f'(x_*) + O((x_k - x_*)^2),$$

using the $O(\cdot)$ notation, instead of writing out the remainder term explicitly. Subtracting x_* from each side of (2.10) and again letting $e_k \equiv x_k - x_*$ denote the error in x_k, we have

$$e_{k+1} = e_k - \frac{f(x_k)}{g} = e_k \left(1 - \frac{f'(x_*)}{g}\right) + O(e_k^2).$$

If $|1 - f'(x_*)/f'(x_0)| < 1$, then for x_0 sufficiently close to x_* (close enough so that the $O(e_k^2)$ term above is negligible), the method converges and convergence is *linear*. In general, we cannot expect better than linear convergence from the constant slope method.

A variation on this idea is to update the derivative occasionally. Instead of taking g_k in (2.9) to be $f'(x_0)$ for all k, one might monitor the convergence of the iteration and, when it seems to be slowing down, compute a new derivative $f'(x_k)$. This requires more derivative evaluations than the constant slope method, but it may converge in fewer iterations. Choosing between methods for solving nonlinear equations usually involves a tradeoff between the cost of an iteration and the number of iterations required to reach a desired level of accuracy.

2.4.3 Secant Method

The *secant method* is defined by taking g_k in (2.9) to be

$$g_k = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}. \tag{2.11}$$

Note that in the limit as x_{k-1} approaches x_k, g_k approaches $f'(x_k)$, so that g_k can be expected to be a reasonable approximation to $f'(x_k)$ when x_{k-1} is close to x_k. It is actually the slope of the

line through the points $(x_{k-1}, f(x_{k-1}))$ and $(x_k, f(x_k))$, called a *secant* line to the curve f since it intersects the curve at two points. The secant method is then defined by

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}, \quad k = 1, 2, \ldots. \tag{2.12}$$

To begin the secant method, one needs *two* starting points x_0 and x_1. An illustration of the secant method for finding a root of $f(x) = x^3 - x^2 - 1$ with $x_0 = 0.5$ and $x_1 = 2$ is given in Figure 2.3.

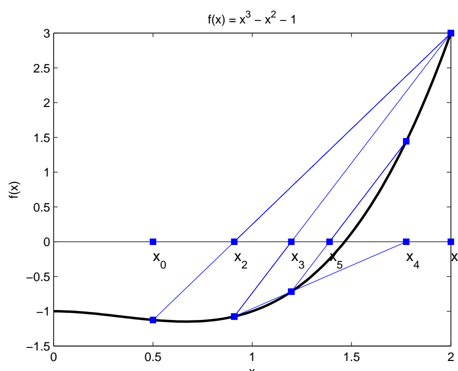


Figure 2.3: Finding a root of $f(x) = x^3 - x^2 - 1$ using $x_0 = 0.5$ and $x_1 = 2$ with the secant method.

Example: Consider again the problem $f(x) \equiv x^2 - 2 = 0$. Taking $x_0 = 1$ and $x_1 = 2$, the secant method generates the following approximations and errors:

$x_2 = 1.3333$	$e_2 = -0.0809$
$x_3 = 1.4$	$e_3 = -0.0142$
$x_4 = 1.4146$	$e_4 = 4.2e - 4$
$x_5 = 1.4142114$	$e_5 = -2.1e - 6$

From the table, it is difficult to determine exactly what the convergence rate of the secant method might be. It seems faster than linear: the ratio $|e_{k+1}/e_k|$ is getting smaller with k. But it seems slower than quadratic: $|e_{k+1}/e_k^2|$ seems to be growing larger with k. Can you guess what the order of convergence is?

It turns out that the order of convergence of the secant method is $\frac{1+\sqrt{5}}{2} \approx 1.62$. I bet you didn't guess that! This result is obtained from the following lemma:

Lemma 2.4.1. *If $f \in C^2$, if x_0 and x_1 are sufficiently close to a root x_* of f, and if $f'(x_*) \neq 0$, then the error e_k in the secant method satisfies*

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k e_{k-1}} = C_*, \tag{2.13}$$

where $C_* = f''(x_*)/(2f'(x_*))$.

Statement (2.13) means that for k large enough

$$e_{k+1} \approx C_* e_k e_{k-1}, \tag{2.14}$$

where the approximate equality can be made as close as we like by choosing k large enough.

Before proving the lemma, let us demonstrate why the stated convergence rate follows from this result. Assume that there exist constants a and α such that, for all k sufficiently large,

$$|e_{k+1}| \approx a|e_k|^\alpha,$$

where again we can make this as close as we like to an actual equality by choosing k large enough. (In a formal proof, this assumption would need justification, but we will just make the assumption in our argument.) Since also $|e_k| \approx a|e_{k-1}|^\alpha$, we can write

$$|e_{k-1}| \approx (|e_k|/a)^{1/\alpha}.$$

Now, assuming that (2.14) holds, substituting these expressions for e_{k-1} and e_{k+1} into (2.14) gives

$$a|e_k|^\alpha \approx C|e_k|(|e_k|/a)^{1/\alpha} = C|e_k|^{1+1/\alpha}a^{-1/\alpha}.$$

Moving the constant terms to one side, this becomes

$$a^{1+1/\alpha}C^{-1} \approx |e_k|^{1+1/\alpha-\alpha}.$$

Since the left-hand side is constant independent of k, the right-hand side must be as well, which means that the exponent of $|e_k|$ must be zero:

$$1 + 1/\alpha - \alpha = 0.$$

Solving for α, we find

$$\alpha = \frac{1 \pm \sqrt{5}}{2}.$$

In order for the method to converge, α must be positive, so the order of convergence must be the positive value $\alpha = (1 + \sqrt{5})/2$.

Proof of Lemma. Subtracting x_* from each side of (2.12) gives

$$e_{k+1} = e_k - \frac{f(x_k)(e_k - e_{k-1})}{f(x_k) - f(x_{k-1})},$$

where we have substituted $e_k - e_{k-1} = (x_k - x_*) - (x_{k-1} - x_*)$ for $x_k - x_{k-1}$. Combining terms, this becomes

$$\begin{aligned} e_{k+1} &= \frac{f(x_k)e_{k-1} - f(x_{k-1})e_k}{f(x_k) - f(x_{k-1})} \\ &= e_k e_{k-1}\left[\frac{f(x_k)/e_k - f(x_{k-1})/e_{k-1}}{x_k - x_{k-1}} \cdot \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}\right], \end{aligned} \tag{2.15}$$

where the last factor inside the brackets in (2.15) converges to $1/f'(x_*)$ as x_k and x_{k-1} approach x_*. The first factor can be written in the form

$$\frac{\frac{f(x_k)-f(x_*)}{x_k-x_*} - \frac{f(x_{k-1})-f(x_*)}{x_{k-1}-x_*}}{x_k - x_{k-1}}.$$

If we define

$$g(x) \equiv \frac{f(x) - f(x_*)}{x - x_*}, \tag{2.16}$$

then the first factor inside the brackets in (2.15) is $(g(x_k) - g(x_{k-1}))/(x_k - x_{k-1})$, which converges to $g'(x_*)$ as x_k and x_{k-1} approach x_*. Differentiating expression (2.16), we find that

$$g'(x) = \frac{(x - x_*)f'(x) - (f(x) - f(x_*))}{(x - x_*)^2},$$

and taking the limit as $x \to x_*$, using L'Hopital's rule, gives

$$\lim_{x \to x_*} g'(x) = \lim_{x \to x_*} \frac{(x - x_*)f''(x)}{2(x - x_*)} = \frac{f''(x_*)}{2}.$$

Thus, *assuming* that e_k and e_{k-1} converge to 0 as $k \to \infty$, it follows from (2.15) that

$$\lim_{k \to \infty} \frac{e_{k+1}}{e_k e_{k-1}} = \frac{1}{2} \frac{f''(x_*)}{f'(x_*)}.$$

Finally, the assumption that x_0 and x_1 are sufficiently close to x_* means that for any $C > \frac{1}{2}|f''(x_*)/f'(x_*)|$, we can assume that x_0 and x_1 lie in an interval about x_* in which $|e_2| \leq C|e_1| \cdot |e_0|$ and that, say, $|e_1| \leq 1/(2C)$ and $|e_0| \leq 1/(2C)$. It follows that $|e_2| \leq \frac{1}{2} \min\{|e_1|, |e_0|\} \leq 1/(4C)$, so x_2 also lies in this interval. By induction it follows that all iterates x_k lie in this interval and that $|e_k| \leq \frac{1}{2}|e_{k-1}| \leq \ldots \leq \frac{1}{2^k}|e_0|$. Hence e_k and e_{k-1} converge to 0 as $k \to \infty$. □

2.5 Analysis of Fixed Point Methods

Sometimes, instead of writing a nonlinear equation in the form $f(x) = 0$, one writes it in the form of a *fixed point problem*: $x = \varphi(x)$; that is, the problem is to find a point x that remains fixed under the mapping φ. A natural approach to solving such a problem is to start with an initial guess x_0, and then iterate according to:

$$x_{k+1} = \varphi(x_k). \tag{2.17}$$

Both Newton's method and the constant slope method can be thought of in this way. For Newton's method, since $x_{k+1} = x_k - f(x_k)/f'(x_k)$, the function φ whose fixed point is being sought is $\varphi(x) = x - f(x)/f'(x)$. Note that $\varphi(x) = x$ if and only if $f(x) = 0$ (assuming that $f'(x) \neq 0$). For the constant slope method, since $x_{k+1} = x_k - f(x_k)/g$, we have $\varphi(x) = x - f(x)/g$, where $g = f(x_0)$. Note that for this function as well, $\varphi(x) = x$ if and only if $f(x) = 0$. The secant

method, $x_{k+1} = x_k - f(x_k)(x_k - x_{k-1})/(f(x_k) - f(x_{k-1}))$ does not fit this general pattern since the right-hand side is a function of *both* x_k and x_{k-1}.

Figure 2.4 illustrates cases where fixed point iteration converges and diverges. In particular, the iteration can produce monotone convergence, oscillatory convergence, periodic behavior, or chaotic/quasi-periodic behavior.

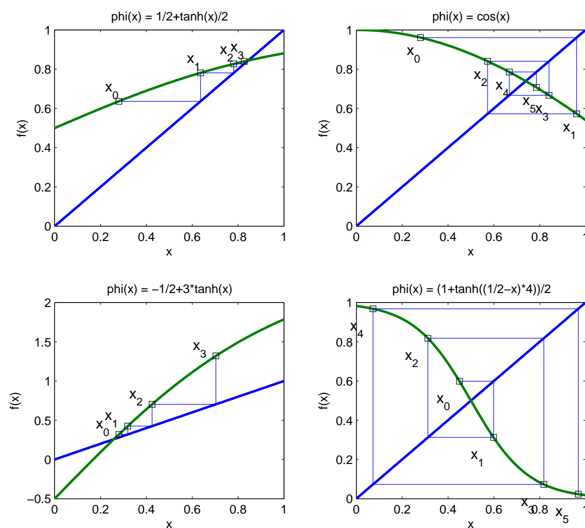
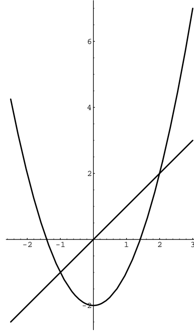


Figure 2.4: Fixed Point Iteration. The iteration may display monotonic convergence (upper left), oscillatory convergence (upper right), monotonic divergence (lower left), or oscillatory divergence (lower right).

Geometrically, fixed points lie at the intersection of the line $y = x$ and the curve $y = \varphi(x)$, as seen in Figure 2.4. Mathematically, such points occur when the output of a function is equal to its input: $\varphi(x) = x$. Fixed points correspond to equilibria of dynamical systems; that is, to points where the solution neither grows nor decays.

Example: Find the fixed points of $\varphi(x) = x^2 - 6$.

First, plot $y = x^2 - 6$ and $y = x$ on the same axes to get an idea of where the intersection points are.



To determine the fixed points mathematically, solve the quadratic equation $x^2 - 6 = x$, or, $x^2 - x - 6 = (x-3)(x+2) = 0$. The solutions are $x = 3$ and $x = -2$.

What determines whether a fixed point iteration will converge? The following theorem gives a *sufficient* condition for convergence.

Theorem 2.5.1. *Assume that $\varphi \in C^1$ and $|\varphi'(x)| < 1$ in some interval about a fixed point x_*. If x_0 is in this interval then the fixed point iteration (2.17) converges to x_*.*

Proof. Expanding $\varphi(x_k)$ in a Taylor series about x_* gives

$$x_{k+1} = \varphi(x_k) = \varphi(x_*) + (x_k - x_*)\varphi'(\xi_k) = x_* + (x_k - x_*)\varphi'(\xi_k),$$

for some ξ_k between x_k and x_*. Subtracting x_* from each side, this becomes

$$e_{k+1} = e_k \varphi'(\xi_k),$$

and taking absolute values on each side gives

$$|e_{k+1}| \leq |\varphi'(\xi_k)| \; |e_k|.$$

Thus if $|\varphi'(x)| < 1$ for all x in an interval about x_*, and if x_0 is in this interval, then future iterates remain in this interval and $|e_k|$ decreases at each step. The error reduction factor approaches $|\varphi'(x_*)|$:

$$\lim_{k \to \infty} \frac{|e_{k+1}|}{|e_k|} = |\varphi'(x_*)|.$$

□

Example: As an example, let us consider three fixed point iteration schemes that can be derived from the equation $x^3 + 6x^2 - 8 = 0$, which has a unique solution in the interval $[1, 2]$. The functions whose fixed points we seek are:

1. $\varphi_1(x) = x^3 + 6x^2 + x - 8$,

2. $\varphi_2(x) = \sqrt{\dfrac{8}{x+6}}$, and

3. $\varphi_3(x) = \sqrt{\dfrac{8-x^3}{6}}$.

Simple algebra shows that each of these functions has a fixed point in the interval $[1,2]$ where $f(x) = x^3 + 6x^2 - 8$ has a root; for example, $\varphi_2(x) = x$ if x is in the interval and $x^2 = 8/(x+6)$, or, $x^3 + 6x^2 - 8 = 0$.

Table 2.1 shows the behavior of fixed point iteration applied to each of these functions with initial guess $x = 1.5$. Run the MATLAB code `fixedpoint.m` to replicate this table or experiment with different initial guesses.

n	$\varphi_1(x) = x$	$\varphi_2(x) = x$	$\varphi_3(x) = x$
0	1.5	1.5	1.5
1	10.375	1.032795559	0.8779711461
2	1764.990234	1.066549452	1.104779810
3	5516973759	1.063999177	1.052898680
4	1.679×10^{29}	1.064191225	1.067142690
5	4.734×10^{87}	1.064176759	1.063386479
6	1.061×10^{263}	1.064177849	1.064388114
7		1.064177767	1.064121800
8		1.064177773	1.064192663
9		1.064177772	1.064173811
10			1.064178826
11			1.064177492
12			1.064177847
13			1.064177753
14			1.064177778
15			1.064177771
16			1.064177773
17			1.064177772

Table 2.1: Fixed point iteration applied to three different functions derived from the equation $x^3 + 6x^2 - 8 = 0$.

These results are explained by the previous theorem:

1. First consider $\varphi_1(x) = x^3 + 6x^2 + x - 8$, a function for which fixed-point iteration quickly diverged. Note that $\varphi_1'(x) = 3x^2 + 12x + 1 > 1$ for all $x \in [1,2]$, so that one must expect divergence.

2. Next consider $\varphi_2(x) = \sqrt{\dfrac{8}{x+6}}$. In this case, $|\varphi_2'(x)| = \sqrt{2}/(x+6)^{3/2} < 1$ for all $x \in [1,2]$. Since $x_0 = 1.5 \in [1,2]$, convergence is guaranteed.

3. Proving the convergence of fixed point iteration for $\varphi_3(x) = x$ is left as an exercise for the reader.

Actually, $\varphi'(x_*)$ need not exist in order to have convergence. The iteration (2.17) will converge to a fixed point x_* if φ is a *contraction*; that is, if there exists a constant $L < 1$ such that for all x and y

$$|\varphi(x) - \varphi(y)| \leq L|x - y|. \tag{2.18}$$

Theorem 2.5.2. *If φ is a contraction, then it has a unique fixed point x_* and the iteration $x_{k+1} = \varphi(x_k)$ converges to x_* from any x_0.*

Proof. We will show that the sequence $\{x_k\}_{k=0}^{\infty}$ is a Cauchy sequence and hence converges. To see this, let j and k be positive integers with $k > j$ and use the triangle inequality to write

$$|x_k - x_j| \leq |x_k - x_{k-1}| + |x_{k-1} - x_{k-2}| + \ldots + |x_{j+1} - x_j|. \tag{2.19}$$

Each difference $|x_m - x_{m-1}|$ can be written as $|\varphi(x_{m-1}) - \varphi(x_{m-2})|$, which is bounded by $L|x_{m-1} - x_{m-2}|$, where $L < 1$ is the constant in (2.18). Repeating this argument for $|x_{m-1} - x_{m-2}|$, we find that $|x_m - x_{m-1}| \leq L^2|x_{m-2} - x_{m-3}|$, and continuing in this way gives $|x_m - x_{m-1}| \leq L^{m-1}|x_1 - x_0|$. Making these substitutions in (2.19) we find

$$|x_k - x_j| \leq (L^{k-1} + L^{k-2} + \ldots + L^j)|x_1 - x_0| = L^j \frac{1 - L^{k-j}}{1 - L}|x_1 - x_0|.$$

If k and j are both greater than or equal to some positive integer N, we will have

$$|x_k - x_j| \leq L^N \frac{1}{1 - L}|x_1 - x_0|,$$

and this quantity goes to 0 as $N \to \infty$. This shows that the sequence x_k, $k = 0, 1, \ldots$ is a Cauchy sequence and hence converges.

To show that x_k converges to a fixed point of φ, we first note that the fact that φ is a contraction implies that it is continuous. Hence $\varphi(\lim_{k\to\infty} x_k) = \lim_{k\to\infty} \varphi(x_k)$. Letting x_* denote $\lim_{k\to\infty} x_k$, we have

$$\varphi(x_*) \equiv \varphi(\lim_{k\to\infty} x_k) = \lim_{k\to\infty} \varphi(x_k) = \lim_{k\to\infty} x_{k+1} = x_*,$$

so that the limit of the sequence x_k is indeed a fixed point. Finally, if y_* is also a fixed point, then since φ is a contraction, we must have

$$|x_* - y_*| = |\varphi(x_*) - \varphi(y_*)| \leq L|x_* - y_*|,$$

where $L < 1$. This can hold only if $y_* = x_*$, so the fixed point is unique. □

JULIA OF THE JULIA SET

GASTON JULIA (1893–1978) – was only 25 years of age when he published his 199 page, *Mémoire sur l'itération des fonctions rationelles* [?]. The publication led to fame in the 1920's but was virtually forgotten until Benoit Mandelbrot's study of fractals. The leather strap seen covering a portion of Julia's face was due to a severe injury in World War I that led to the loss of his nose.

2.6 Fractals, Julia Sets and Mandelbrot Sets*

Fixed point iteration and Newton's method can be used for problems defined in the complex plane as well. Consider, for example, the problem of finding a fixed point of $\varphi(z) \equiv z^2$. It is easy to predict the behavior of the fixed point iteration $z_{k+1} = z_k^2$. If $|z_0| < 1$, then the sequence z_k converges to the fixed point 0. If $|z_0| > 1$, then the iterates grow in modulus and the method diverges. If $|z_0| = 1$, then $|z_k| = 1$ for all k. If $z_0 = 1$ or $z_0 = -1$, the sequence quickly settles down to the fixed point 1. On the other hand, if, say, $z_0 = e^{2\pi i/3}$, then $z_1 = e^{4\pi i/3}$ and $z_2 = e^{8\pi i/3} = e^{2\pi i/3} = z_0$, so the cycle repeats. It can be shown that if $z_0 = e^{2\pi i\alpha}$ where α is irrational then the sequence of points z_k never repeats but becomes dense on the unit circle. The sequence of points z_0, $z_1 = \varphi(z_0)$, $z_2 = \varphi(\varphi(z_0))$, ... is called the *orbit* of z_0 under φ.

If $\varphi(z)$ is a polynomial function, then the set of points z_0 for which the orbit remains bounded is called the *filled Julia set* for φ, and its boundary is called the *Julia set*. Thus the filled Julia set for z^2 is the closed unit disk, while the Julia set is the unit circle. These sets are named after the French mathematician Gaston Julia. In 1918, he and Pierre Fatou investigated the behavior of these sets, which can be *far* more interesting than the simple example presented above. Their work received renewed attention in the 1980's when the advent of computer graphics made numerical experimentation and visualization of the results easy and fun.

Figure 2.5 displays the filled Julia set for the function $\varphi(z) = z^2 - 1.25$. The boundary of this set is a *fractal*, meaning that it has dimension neither one nor two but some fraction in between. The figure is also *self-similar*, meaning that if one repeatedly zooms in on a small subset of the picture, the pattern still looks like the original.

The plot in Figure 2.5 was obtained by starting with many different points z_0 throughout a rectangular region containing the two fixed points of φ, and running the fixed point iteration until one of three things occurred: Either $|z_k| \geq 2$, in which case we conclude that the orbit is unbounded and z_0 is not in the set; or z_k comes within 10^{-6} of a fixed point and stays within that distance for 5 iterations, in which case we conclude that the iterates are converging to the fixed point and z_0 is in the set; or the number of iterations k reaches 100 before either of the previous two conditions

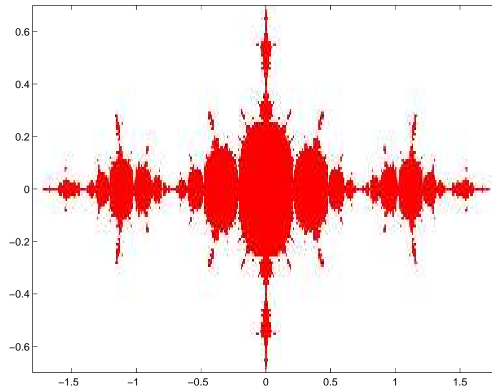


Figure 2.5: Filled Julia set for $\varphi(z) = z^2 - 1.25$.

occur, in which case we conclude that the sequence z_k is not converging to a fixed point but does remain bounded and hence z_0 is again in the set. Note that since $|z_{k+1}| \geq |z_k|^2 - 1.25$, if $|z_k| \geq 2$, then $|z_{k+1}| \geq 4 - 1.25 = 2.75$, $|z_{k+2}| \geq 2.75^2 - 1.25 \approx 6.3$, etc., so that the sequence really is unbounded. The code to produce this plot is given in Figure 2.6.

While this small bit of analysis enables us to determine a sufficient condition for the orbit to be unbounded, we really have only an educated guess as to the points for which the orbit is bounded. It is possible that the iterates remain less than 2 in modulus for the first 100 steps but then grow unboundedly, or that they appear to be converging to one of the fixed points but then start to diverge. Without going through further analysis, one can gain confidence in the computation by varying some of the parameters. Instead of assuming that the sequence is bounded if the iterates remain less than 2 in absolute value for 100 iterations, one could try 200 iterations. To improve the efficiency of the code, one might also try a lower value, say, 50 iterations. If reasonable choices of parameters yield the same results, then that suggests that the computed sets may be correct, while if different parameter choices yield different sets then one must conclude that the numerical results are suspect. Other ways to test the computation include comparing with other results in the literature and comparing the computed results with what is known analytically about these sets. This comparison of numerical results and theory is often important in both pure and applied mathematics. One uses the theory to check numerical results, and in cases where theory is lacking, one uses numerical results to suggest what theory might be true.

If one investigates Julia sets for all functions of the form $\varphi(z) = z^2 + c$, where c is a complex constant, one finds that some of these sets are *connected* (one can move between any two points in the set without leaving the set), while others are not. Julia and Fatou simultaneously found a simple criterion for the Julia set associated with $\varphi(z)$ to be connected: It is connected if and only if 0 is in the filled Julia set. In 1982, Benoit Mandelbrot used computer graphics to study the question of which values of c give rise to Julia sets that are connected, by testing different values of c throughout the complex plane and running fixed point iteration as above, with initial value $z_0 = 0$. The astonishing answer was the *Mandelbrot set*, depicted in Figure 2.7. The black points

```
phi = inline('z^2 - 1.25');   % Define the function whose fixed points we seek.
fixpt1 = (1 + sqrt(6))/2;     % These are the fixed points.
fixpt2 = (1 - sqrt(6))/2;

colormap([1 0 0; 1 1 1]);     % Points numbered 1 (inside) will be colored red;
                              %   those numbered 2 (outside) will be colored white.
M = 2*ones(141,361);          % Initialize array of point colors to 2 (white).

for j=1:141,                  % Try initial values with imaginary parts between
  y = -.7 + (j-1)*.01;        %   -0.7 and 0.7
  for i=1:361,                % and with real parts between
    x = -1.8 + (i-1)*.01;     %   -1.8 and 1.8.
    z = x + 1i*y;             % 1i is the MATLAB symbol for sqrt(-1).
    zk = z;
    iflag1 = 0;               % iflag1 and iflag2 count the number of iterations
    iflag2 = 0;               %   when a root is within 1.e-6 of a fixed point;
    kount = 0;                % kount is the total number of iterations.

    while kount < 100 & abs(zk) < 2 & iflag1 < 5 & iflag2 < 5,
      kount = kount+1;
      zk = phi(zk);           % This is the fixed point iteration.
      err1 = abs(zk-fixpt1);  % Test for convergence to fixpt1.
      if err1 < 1.e-6,
         iflag1 = iflag1 + 1;
      else
         iflag1 = 0;
      end;
      err2 = abs(zk-fixpt2);  % Test for convergence to fixpt2.
      if err2 < 1.e-6,
        iflag2 = iflag2 + 1;
      else
        iflag2 = 0;
      end;
    end;
    if iflag1 >= 5 | iflag2 >= 5 | kount >= 100,   % If orbit is bounded, set this
      M(j,i) = 1;                                  %   point color to 1 (red).
    end;
  end;
end;

image([-1.8 1.8],[-.7 .7],M),  % This plots the results.
axis xy                        % If you don't do this, vertical axis is inverted.
```

Figure 2.6: MATLAB code to compute the filled Julia set for $\varphi(z) = z^2 - 1.25$

are the values c for which the Julia set is connected, while the shading of the other points indicates the rate at which the fixed point iteration applied to $z^2 + c$ with $z_0 = 0$ diverges.

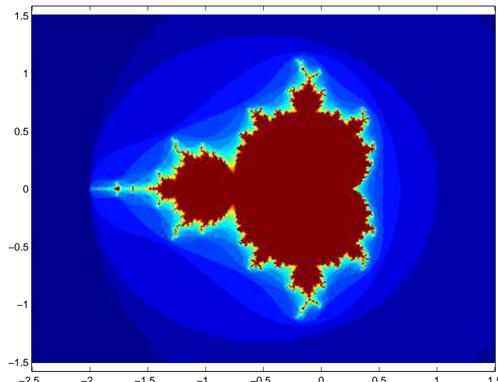


Figure 2.7: Mandelbrot Set

Newton's method can also be applied to problems in the complex plane, and plots of points from which it is or is not convergent are just as beautiful and interesting as the ones obtained from fixed point iteration. Figure 2.8 shows the behavior of Newton's method when applied to the problem $f(z) \equiv z^3 + 1 = 0$. This equation has three roots: $z = -1$, $z = e^{\pi i/3}$, and $z = e^{-\pi i/3}$. Initial points from which Newton's method converges to the first root are colored green, those from which it converges to the second root are colored red, and those from which it converges to the third root are colored blue. The boundary of this image is also a fractal and one can zoom in on different pieces of the figure to see a pattern like that of the original.

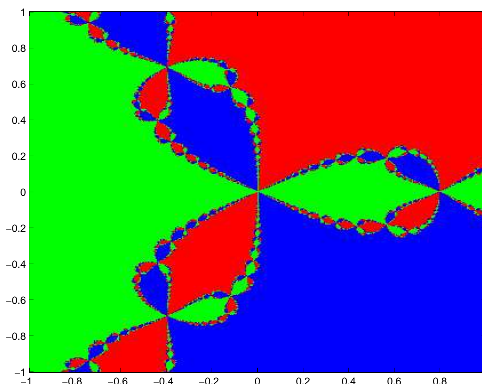


Figure 2.8: Convergence of Newton's method for $f(z) = z^3 + 1$. Green initial points converge to -1, red initial points converge to $e^{\pi i/3}$, and blue initial points converge to $e^{-\pi i/3}$

Once again, we should stress that one cannot determine numerically whether Newton's method

is or is not convergent. The above figure was obtained by trying many different initial guesses z_0, iterating until 10 consecutive Newton iterates were within $1.e-6$ of one of the roots and then coloring that initial guess according to which root was approximated. It is possible that 10 Newton iterates could be within $1.e-6$ of one root while later iterates would converge to a different root or diverge. Further analysis can verify the figure, but that is beyond the scope of this book. For an excellent elementary discussion of Julia sets and Newton's method in the complex plane, see [14].

Exercises

1. Use MATLAB to plot the function $f(x) = (5 - x)\exp(x) - 5$, for x between 0 and 5. (This function is associated with the *Wien radiation law*, which gives a method to estimate the surface temperature of a star.)

   (a) Write a bisection routine or use routine "bisect" available from:

   `http://www.math.washington.edu/~greenbau/Math_464/bisect.m`

   to find a root of $f(x)$ in the interval $[4, 5]$, accurate to six decimal places. At each step, print out the endpoints of the smallest interval known to contain a root. Without running the code, answer the following: How many steps would be required to reduce the size of this interval to 10^{-12}? Explain your answer.

   (b) Write a routine to use Newton's method or use routine "newton" available from:

   `http://www.math.washington.edu/~greenbau/Math_464/newton.m`

   to find a root of $f(x)$, using initial guess $x_0 = 5$. Print out your approximate solution x_k and the value of $f(x_k)$ at each step and run until $|f(x_k)| \leq 10^{-8}$. Without running the code, answer the following: About how many steps would be required to compute the solution to 16 decimal places (assuming that your machine carried enough decimal places to do this)? Explain your answer.

   (c) Take your routine for doing Newton's method and modify it to run the secant method. Repeat the run of part (b), using, say, $x_0 = 4$ and $x_1 = 5$, and again predict (without running the code) how many steps would be required to compute the solution to 16 decimal places (assuming that your machine carried enough decimal places to do this) using the secant method. Explain your answer.

2. Newton's method can be used to compute reciprocals, without division. To compute $1/R$, let $f(x) = x^{-1} - R$ so that $f(x) = 0$ when $x = 1/R$. Write down the Newton iteration for this problem and compute (by hand or with a calculator) the first few Newton iterates for approximating $1/3$, starting with $x_0 = 0.5$, and not using any division. What happens if you start with $x_0 = 1$? For positive R, use the theory of fixed point iteration to determine an interval about $1/R$ from which Newton's method will converge to $1/R$.

3. Use Newton's method to approximate $\sqrt{2}$ to 6 decimal places.

4. Write down the first few iterates of the secant method for solving $x^2 - 3 = 0$, starting with $x_0 = 0$ and $x_1 = 1$.

5. Consider the function $h(x) = x^4/4 - 3x$. In this problem, we will see how to use Newton's method to find the *minimum* of the function $h(x)$.

   (a) Derive a function f that has a root at the point where h achieves its minimum. Write down the formula for Newton's method applied to f.

(b) Take one step (by hand) with Newton's method starting with a guess of $x_0 = 1$.

(c) Take two steps (by hand) towards the root of f (i.e., the minimum of h) using bisection with $a = 0$, $b = 4$.

6. In finding a root with Newton's method, an initial guess of $x_0 = 4$ with $f(x_0) = 1$ leads to $x_1 = 3$. What is the derivative of f at x_0?

7. In using the secant method to find a root, $x_0 = 2$, $x_1 = -1$ and $x_2 = -2$ with $f(x_1) = 4$ and $f(x_2) = 3$. What is $f(x_0)$?

8. Can the bisection method be used to find the roots of the function $f(x) = \sin(x) + 1$? Why or why not?

9. The function
$$f(x) = \frac{x^2 - 2x + 1}{x^2 - x - 2}$$
has exactly one zero in the interval $[0, 3]$, at $x = 1$. Using $a = 0$ and $b = 3$, run the bisection method on $f(x)$ with a stopping tolerance of `delta = 1e-3`. Explain why it does not appear to converge to the root.

   Use MATLAB to plot this function over the interval in a way that makes it clear what's going on. **Hint:** You may want to use the plot command over two different x-intervals separately in order to show the behavior properly.

10. Write out the Newton iteration formula for each of the functions below. For each function and given starting value, write a MATLAB script that takes 5 iterations of Newton's method and prints out the iterates with format `%25.15e` so you see all digits.

    (a) $f(x) = \sin(x), \quad x_0 = 3$

    (b) $f(x) = x^3 - x^2 - 2x, \quad x_0 = 3$

    (c) $f(x) = 1 + 0.99x - x, \quad x_0 = 1$. In exact arithmetic, how many iterations does Newton's method need to find a root of this function?

11. Let function $\varphi(x) = (x^2 + 4)/5$.

    (a) Find the fixed point(s) of $\varphi(x)$?

    (b) Would the fixed point iteration, $x_{k+1} = \varphi(x_k)$, converge to a fixed point in the interval $[0, 2]$ for all initial guesses $x_0 \in [0, 2]$ with $x_0 \neq x_*$?

12. Consider the equation $a = y - \epsilon \sin y$, where $0 < \epsilon < 1$ is given and $a \in [0, \pi]$ is given. Write this in the form of a fixed point problem for the unknown solution y and show that it has a unique solution.

13. If you enter a number into a handheld calculator and repeatedly press the cosine button, what number (approximately) will eventually appear? Provide a proof. [Note: Set your calculator to interpret numbers as radians rather than degrees; otherwise you will get a different answer.]

14. The function $\varphi(x) = \frac{1}{2}\left(-x^2 + x + 2\right)$ has a fixed point at $x = 1$. Starting with $x_0 = 0.5$, we use $x_{n+1} = \varphi(x_n)$ to obtain the sequence $\{x_n\} = \{0.5, 1.1250, 0.9297, 1.0327, 0.9831, \ldots\}$. Describe the behavior of the sequence. (If it converges, how does it converge? If it diverges, how does it diverge?)

15. Steffensen's method for solving $f(x) = 0$ is defined by

$$x_{k+1} = x_k - \frac{f(x_k)}{g_k},$$

where

$$g_k = \frac{f(x_k + f(x_k)) - f(x_k)}{f(x_k)}.$$

Show that this is quadratically convergent, under suitable hypotheses.

16. Returning to the problem facing an artillery officer during battle, he needs to determine the angle θ at which to aim the cannon. Again, the desired θ is a solution to the nonlinear equation:

$$\frac{2v_0^2 \sin\theta \ \cos\theta}{g} = d. \tag{2.20}$$

(a) Show that

$$\theta = \frac{1}{2} \arcsin \frac{dg}{v_0^2}$$

solves equation (2.20), assuming that v_0 and d are fixed.

(b) Use Newton's Method, to find θ (to within two decimal places when $v_0 = 126$ km/hour, $d = 1200$ m and $g = 9.8$ m/s^2. As an initial guess use $\theta = \pi/4$.

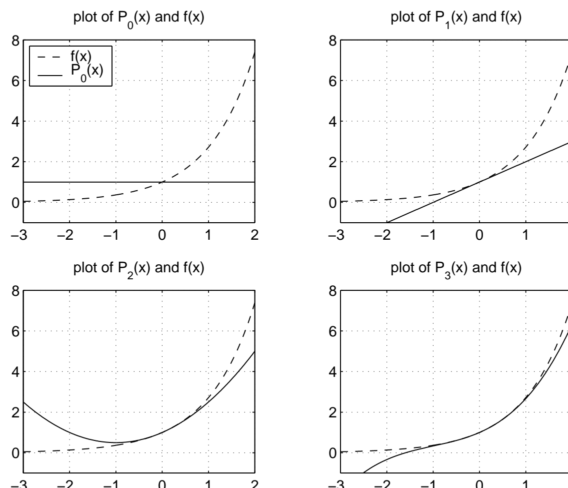
17. Recall that the Taylor series approximation to a function $f(x)$ expanded about a point x_0 is

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2!}f''(x_0)(x - x_0)^2 + \frac{1}{3!}f'''(x_0)(x - x_0)^3 + \cdots$$

From this we can define a sequence of polynomials of increasing degree that approximate the function near the point x_0. The polynomial of degree n is

$$P_n(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{1}{2}f''(x_0)(x-x_0)^2 + \frac{1}{6}f'''(x_0)(x-x_0)^3 + \cdots + \frac{1}{n!}f^{(n)}(x_0)(x-x_0)^n,$$

where $f^{(n)}(x_0)$ means the nth derivative of f evaluated at x_0.



The figure shows the function $f(x) = e^x$ and the first four Taylor series approximations about the point $x_0 = 0$. Since all derivatives of $f(x) = e^x$ are again just e^x, the nth order approximation about a general point x_0 is

$$P_n(x) = e^{x_0} + e^{x_0}(x - x_0) + \frac{1}{2}e^{x_0}(x - x_0)^2 + \frac{1}{6}e^{x_0}(x - x_0)^3 + \cdots + \frac{1}{n!}e^{x_0}(x - x_0)^n.$$

For $x_0 = 0$, we find

$$\begin{aligned} P_0(x) &= 1, \\ P_1(x) &= 1 + x, \\ P_2(x) &= 1 + x + \frac{1}{2}x^2, \\ P_3(x) &= 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3, \end{aligned}$$

and these are the functions plotted in the figure, along with $f(x)$.

Produce a similar set of four plots for the Taylor series approximations to $f(x) = e^{1-x^2}$ about the point $x_0 = 1$. You can start with the m-file (`plotTaylor.m`) used to create the figure above, which can be found on the web page for this class. Choose the `x` values and `axis` parameters so that the plots are over a reasonable range of values to exhibit the functions well. The polynomials should give good approximations to the function near the point $x_0 = 1$.

18. Finding the area of 96–sided polygons that inscribe and circumscribe a circle of diameter 1, Greek mathematician Archimedes (287–212 BC) determined that $\frac{223}{71} < \pi < \frac{22}{7}$. Almost two thousand years later, John Machin (1680–1752) exploited a recent discovery of his contemporary Brook Taylor, the Taylor series. He also employed the following trigonometric identity that he discovered:

$$\pi = 16 \arctan\left(\frac{1}{5}\right) - 4 \arctan\left(\frac{1}{239}\right). \tag{2.21}$$

(a) Find the Maclaurin series for $\arctan(x)$.

(b) Write down $P_n(x)$, the n–th order Taylor polynomial of $\arctan(x)$ centered at $x = 0$.

(c) Approximate π by using $P_n(x)$ and (2.21). More specifically, use the approximation:

$$\pi \approx T_n = 16P_n\left(\frac{1}{5}\right) - 4P_n\left(\frac{1}{239}\right).$$

Using MATLAB, find T_n and the absolute error $|T_n - \pi|$ for $n = 1, 3, 5, 7$, and 9. How many decimal places of accuracy did you find for $n = 9$?

For centuries, Machin's formula served as a primary tool for π–hunters. As late as 1973, one million digits of π were computed on a computer by Guilloud and Bouyer using a version of Machin's formula.

Chapter 3

Floating Point Arithmetic

Numerical analysts use computers and consequently computer arithmetic. Most computers store numbers in *binary* (base 2) format and since there is limited space in a computer word, not all numbers can be represented exactly; they must be *rounded* to fit the word size. This means that arithmetic operations are not performed exactly. Although the error made in any one operation is usually negligible (of relative size about 10^{-16} using double precision), a poorly designed algorithm may magnify this error to the point that it destroys all accuracy in the computed solution. For this reason it is important to understand the effects of rounding errors on computed results. We begin this chapter with two stories that demonstrate the potential cost of overlooking these effects.

3.1 Costly Disasters Caused by Rounding Errors

The Intel Pentium Flaw [1, 9, 15]. In the summer of 1994, Intel anticipated the commercial success of its new Pentium chip. The new chip was twice as fast at division as previous Intel chips running at the same clock rate. Concurrently, Professor Thomas R. Nicely, a mathematician at Lynchburg College in Virginia, was computing the sum of the reciprocals of prime numbers using a computer with the new Pentium chip. The computational and theoretical results differed significantly. However, results run on a computer using an older 486 CPU calculated correct results. In time, Nicely tracked the error to the Intel chip. Having contacted Intel and received little response to his initial queries, Nicely posted a general notice on the Internet asking for others to confirm his findings. The posting (dated October 30, 1994) with subject line *Bug in the Pentium FPU* [1] began:

> It appears that there is a bug in the floating point unit (numeric coprocessor) of many, and perhaps all, Pentium processors.

This email began a furor of activity, so much so that only weeks later on December 13, IBM halted shipment of their Pentium machines, and in late December, Intel agreed to replace all flawed Pentium chips upon request. The company put aside a reserve of $420 million to cover costs, a major investment for a flaw. With a flurry of internet activity between November 29 and December

11, Intel had become a laughingstock on the Internet joke circuit, but it wasn't funny to Intel. On Friday, December 16 Intel stock closed at \$59.50, down \$3.25 for the week.

Intel Pentium Chip

What type of error could the chip make in its arithmetic? The *New York Times* printed the following example of the Pentium bug: Let $A = 4,195,835.0$ and $B = 3,145,727.0$, and consider the quantity

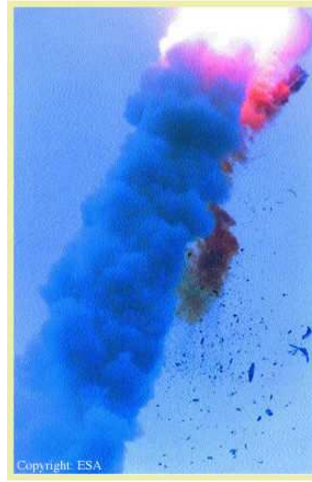
$$A - (A/B) * B.$$

In exact arithmetic, of course, this would be 0, but the Pentium computed 256, because the quotient A/B was accurate to only about 5 decimal places. Is this *close enough*? For many applications it probably is, but we will see in later sections that we need to be able to count on computers to do better than this.

While such an example can make one wonder how Intel missed such an error, it should be noted that subsequent analysis confirmed the subtlety of the mistake. Alan Edelman, professor of Mathematics at Massachusetts Institute Technology, writes in his article of 1997 published in *SIAM Review* [9]:

> We also wish to emphasize that, despite the jokes, the bug is far more subtle than many people realize....The bug in the Pentium was an easy mistake to make, and a difficult one to catch.

Ariane 5 Disaster [2, 10]. The Ariane 5, a giant rocket capable of setting a pair of three-ton satellites into orbit with each launch, took 10 years and 7 billion dollars for the European Space Agency to build. It's maiden launch was met with eager anticipation as the rocket was intended to propel Europe far into the lead of the commercial space business.



(a) (b)

The Ariane-5 rocket (left) lifting off and (right) self destructing after a numerical error on June 4, 1996.

On June 4, 1996, the unmanned rocket took off cleanly but veered off course and exploded in just under 40 seconds after liftoff. Why? To answer this question, we must step back into the programming of the onboard computers.

During the design of an earlier rocket, programmers decided to implement an additional "feature" that would leave the horizontal positioning function (designed for positioning the rocket on the ground) running after the countdown had started, anticipating the possibility of a delayed countdown. Since the expected deviation while on the ground was minimal, only a small amount of memory (16 bits) was allocated to the storage of this information. After the launch, however, the horizontal deviation was large enough that the number could not be stored correctly with 16-bits, resulting in an exception error. This error instructed the primary unit to shutdown. Then, all functions were transferred to a backup unit, created for redundancy in case the primary unit shutdown. As such, the backup system contained the same bug and shut itself down. Suddenly, the rocket was veering off course causing damage between the solid rocket boosters and the main body of the rocket. Detecting the mechanical failure, the master control systems triggered a self-destruct cycle, as had been programmed in the event of serious mechanical failure in flight. Suddenly, the rocket and its expensive payloads were scattered over about 12 square kilometers east of the launch pad. Millions of dollars would have been saved if the data had simply been saved in a larger variable rather than the 16-bit memory location allocated in the program.

As seen in these examples, it is important for numerical analysts to understand the impact of rounding errors on their calculations. This chapter covers the basics of computer arithmetic and the IEEE standard. Later we will see more about how to apply this knowledge to the analysis of algorithms. For an excellent and very readable book on computer arithmetic and the IEEE standard, see [17].

3.2 Binary Representation and Base 2 Arithmetic

Most computers today use *binary* or *base 2* arithmetic. In base 10, a natural number is represented by a sequence of digits from 0 to 9, with the right-most digit representing 1's (or 10^0's), the next representing 10's (or 10^1's), the next representing 100's (or 10^2's), etc. In base 2, the digits are 0 and 1, and the right-most digit represents 1's (or 2^0's), the next represents 2's (or 2^1's), the next 4's (or 2^2's), etc. Thus, for example, the decimal number 10 is written in base 2 as 1010_2: one $2^3 = 8$, zero 2^2, one $2^1 = 2$, and zero 2^0. The decimal number 27 is 11011_2: one $2^4 = 16$, one $2^3 = 8$, zero 2^2, one $2^1 = 2$, and one $2^0 = 1$. The binary representation of a positive integer is determined by first finding the highest power of 2 that is less than or equal to the number; in the case of 27, this is 2^4, so a 1 goes in the fifth position from the right of the number: 1____. One then subtracts 2^4 from 27 to find that the remainder is 11. Since 2^3 is less than 11, a 1 goes in the next position to the right: 11___. Subtracting 2^3 from 11 leaves 3, which is less than 2^2, so a 0 goes in the next position: 110__. Since 2^1 is less than 3, a 1 goes in the next position, and since $3 - 2^1 = 1$, another 1 goes in the right-most position to give $27 = 11011_2$.

Binary arithmetic is carried out in a similar way to decimal arithmetic, except that when adding binary numbers one must remember that $1 + 1$ is 10_2. To add the two numbers 10 and 27, we align their binary digits and do the addition as below. The top row shows the digits that are *carried* from one column to the next.

```
11 1
  1010
+11011
 -----
100101
```

You can check that 100101_2 is equal to 37. Subtraction is similar, with *borrowing* from the next column being necessary when subtracting 1 from 0. Multiplication and division follow similar patterns.

Just as we represent rational numbers using decimal expansions, we can also represent them using *binary* expansions. The digits to the right of the decimal point in base 10 represent 10^{-1}'s (tenths), 10^{-2}'s (hundredths), etc., while those to the right of the binary point in base 2 represent 2^{-1}'s (halves) 2^{-2}'s (fourths), etc. For example, the fraction 11/5 is 5.5 in base 10, while it is 101.1_2 in base 2: one 2^2, one 2^0, and one 2^{-1}. Not all rational numbers can be represented with *finite* decimal expansions. The number 1/3, for example, is $.33\overline{3}$, with the bar over the 3 meaning that this digit is repeated infinitely many times. The same is true for binary expansions, although the numbers that require an infinite binary expansion may be different from the ones that require an infinite decimal expansion. For example, the number $1/10 = 0.1$ in base 10 has the repeating binary expansion: $0.000\overline{1100}_2$. To see this, one can do binary long division in a similar way to base 10 long division:

```
        .0001100
       ----------
1010 /1.00000000
        1010
        ----
         1100
         1010
         ----
           10000
```

Irrational numbers such as $\pi \approx 3.141592654$ can only be approximated by decimal expansions, and the same holds for binary expansions.

3.3 Floating Point Representation

A computer word consists of a certain number of bits, which can be either on (to represent 1) or off (to represent 0). Some early computers used *fixed point* representation, where one bit is used to denote the sign of a number, a certain number of the remaining bits are used to store the part of the binary number to the left of the binary point, and the remaining bits are used to store the part to the right of the binary point. The difficulty with this system is that it can store numbers only in a very limited range. If, say, 16 bits are used to store the part of the number to the left of the binary point, then the left-most bit represents 2^{15}, and numbers greater than or equal to 2^{16} cannot be stored. Similarly, if, say, 15 bits are used to store the part of the number to the right of the binary point, then the right-most bit represents 2^{-15} and no positive number smaller than 2^{-15} can be stored.

A more flexible system is *floating point* representation, which is based on scientific notation. Here a number is written in the form $\pm m \times 2^E$, where $1 \leq m < 2$. Thus, the number $10 = 1010_2$ would be written as $1.010_2 \times 2^3$, while $\frac{1}{10} = 0.000\overline{1100}_2$ would be written as $1.100\overline{1100}_2 \times 2^{-4}$. The computer word consists of three fields: one for the sign, one for the exponent E, and one for the significand m. A *single precision* word consists of 32 bits: 1 bit for the sign (0 for $+$, 1 for $-$), 8 bits for the exponent, and 23 bits for the significand. Thus the number $10 = 1.010_2 \times 2^3$ would be stored in the form

0	E=3	1.010...0

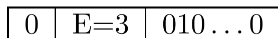
while $5.5 = 1.011_2 \times 2^2$ would be stored in the form

0	E=2	1.0110...0

[We will explain later how the exponent is actually stored.] A number that can be stored exactly using this scheme is called a *floating point number*. The number $1/10 = 1.100\overline{1100}_2 \times 2^{-4}$ is not a floating point number since it must be *rounded* in order to be stored.

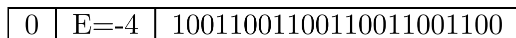
Before describing rounding, let us consider one slight improvement on this basic storage scheme. Recall that when we write a binary number in the form $m \times 2^E$, where $1 \leq m < 2$, the first bit to

the left of the binary point is always 1. Hence there is no need to store it. If the significand is of the form $b_0.b_1 \dots b_{23}$, then instead of storing $b_0, b_1, \dots, b_{22}$, we can keep one extra place by storing $b_1, \dots, b_{23}$, knowing that b_0 is 1. This is called *hidden bit* representation. Using this scheme, the number $10 = 1.010_2 \times 2^3$ is stored as



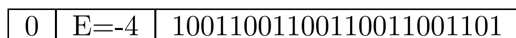
0	E=3	010...0

The number $1/10 = 1.\overbrace{10011001100110011001100}^{23 \text{ bits}}\overline{1100}_2 \times 2^{-4}$ is approximated by either



0	E=-4	10011001100110011001100

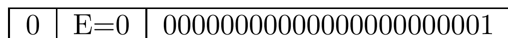
or



0	E=-4	10011001100110011001101

depending on the rounding mode being used. (See Section 3.5.) There is one difficulty with this scheme, and that is how to represent the number 0. Since 0 cannot be written in the form $1.b_1 b_2 \dots \times 2^E$, we will need a special way to represent 0.

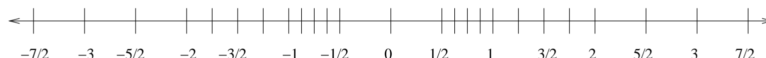
The gap between 1 and the next larger floating point number is called the *machine precision* and is often denoted by ϵ. (In MATLAB, this number is called `eps`.) [Note: Some sources define the machine precision to be $\frac{1}{2}$ times this number.] In single precision, the next floating point number after 1 is $1 + 2^{-23}$, which is stored as:



0	E=0	00000000000000000000001

Thus, for single precision, we have $\epsilon = 2^{-23} \approx 1.2e - 07$. The default precision in MATLAB is not single but *double precision*, where a word consists of 64 bits: 1 for the sign, 11 for the exponent, and 52 for the significand. Hence in double precision the next larger floating point number after 1 is $1 + 2^{-52}$, so that the machine precision is $2^{-52} \approx 2.2e - 16$.

Since single and double precision words contain a large number of bits, it is difficult to get a feel for what numbers can and cannot be represented. It is instructive instead to consider a toy system in which only significands of the form $1.b_1 b_2$ can be represented and only exponents 0, 1, and -1 can be stored. This system is described in [17]. What are all the numbers that can be represented in this system? Since $1.00_2 = 1$, we can represent $1 \times 2^0 = 1$, $1 \times 2^1 = 2$, and $1 \times 2^{-1} = \frac{1}{2}$. Since $1.01_2 = \frac{5}{4}$, we can store the numbers $\frac{5}{4}$, $\frac{5}{2}$, and $\frac{5}{8}$. Since $1.10_2 = \frac{3}{2}$, we can store the numbers $\frac{3}{2}$, 3, and $\frac{3}{4}$. Finally, $1.11_2 = \frac{7}{4}$ gives us the numbers $\frac{7}{4}$, $\frac{7}{2}$, and $\frac{7}{8}$. These numbers are plotted below.



There are several things to note about this toy system. First, the machine precision ϵ is .25, since the number just right of 1 is $\frac{5}{4}$. Second, in general, the gaps between representable numbers become larger as we move away from the origin. This is acceptable since the *relative gaps*, the difference between two consecutive numbers divided by, say, their average, remains of reasonable

size. Note, however, that the gap between 0 and the smallest positive number is *much* larger than the gap between the smallest and next smallest positive number. This is the case with single and double precision floating point numbers as well. The smallest positive (normalized) floating point number in any such system is 1.0×2^{-E}, where $-E$ is the smallest representable exponent; the next smallest number is $(1+\epsilon) \times 2^{-E}$, where ϵ is the machine precision, so the gap between these two numbers is ϵ times the gap between 0 and the first positive number. This gap can be filled in using *subnormal* numbers. Subnormal numbers have less precision than normalized floating point numbers and will be described in the next section.

3.4 IEEE Floating Point Arithmetic

In the 1960's and 1970's, each computer manufacturer developed its own floating point system, resulting in inconsistent program behavior across machines. Most computers used binary arithmetic, but the IBM 360/70 series used hexadecimal (base 16), and Hewlett-Packard calculators used decimal arithmetic. In the early 1980's, due largely to the efforts of W. Kahan, computer manufacturers adopted a standard: the IEEE (Institute for Electrical and Electronics Engineers) standard. This standard required:

- Consistent representation of floating point numbers across machines.
- Correctly rounded arithmetic.
- Consistent and sensible treatment of exceptional situations such as divide by 0.

In order to comply with the IEEE standard, computers represent numbers in the way described in the previous section. A special representation is needed for 0 (since it cannot be represented with the standard hidden bit format), and also for $\pm\infty$ (the result of dividing a nonzero number by 0), and also for NaN (Not a Number; e.g. 0/0). This is done with special bits in the exponent field, which slightly reduces the range of possible exponents. Special bits in the exponent field are also used to signal subnormal numbers.

There are 3 standard precisions. As noted previously, a **single precision** word consists of 32 bits, with 1 bit for the sign, 8 for the exponent, and 23 for the significand. A **double precision** word consists of 64 bits, with 1 bit for the sign, 11 for the exponent, and 52 for the significand. An **extended precision** word consists of 80 bits, with 1 bit for the sign, 15 for the exponent, and 64 for the significand. (Note, however, that numbers stored in extended precision do not use hidden bit storage.)

Table 3.1 shows the floating point numbers, subnormal numbers, and exceptional situations that can be represented using IEEE single precision.

It can be seen from the table that the smallest positive normalized floating point number that can be stored is $1.0_2 \times 2^{-126} \approx 1.2 \times 10^{-38}$, while the largest is $1.1\ldots1_2 \times 2^{127} \approx 3.4 \times 10^{38}$. The exponent field for normalized floating point numbers represents the actual exponent plus 127, so,

If exponent field is:	Then number is:	Type of number:
00000000	$\pm(0.b_1 \ldots b_{23})_2 \times 2^{-126}$	0 or subnormal
$00000001 = 1_{10}$ $00000010 = 2_{10}$ $\vdots$ $01111111 = 127_{10}$ $\vdots$ $11111110 = 254_{10}$	$\pm(1.b_1 \ldots b_{23})_2 \times 2^{-126}$ $\pm(1.b_1 \ldots b_{23})_2 \times 2^{-125}$ $\vdots$ $\pm(1.b_1 \ldots b_{23})_2 \times 2^{0}$ $\vdots$ $\pm(1.b_1 \ldots b_{23})_2 \times 2^{127}$	Normalized numbers. Exponent field is: actual exponent + 127
11111111	$\pm\infty$ if $b_1 = \ldots = b_{23} = 0$, NaN otherwise	Exceptions

Table 3.1: IEEE Single Precision

using the 8 available exponent bits (and setting aside two possible bit configurations for special situations) we can represent exponents between -126 and $+127$. The two special exponent field bit patterns are all 0's and all 1's. An exponent field consisting of all 0's signals either 0 or a subnormal number. Note that subnormal numbers have a 0 in front of the binary point instead of a 1 and are always multiplied by 2^{-126}. Thus the number $1.1_2 \times 2^{-128} = .011_2 \times 2^{-126}$ would be represented with an exponent field string of all 0's and a significand field $0110\ldots0$. Subnormal numbers have less precision than normalized floating point numbers since the significand is shifted right, causing fewer of its bits to be stored. The smallest positive subnormal number that can be represented has twenty two 0's followed by a 1 in its significand field, and its value is $2^{-23} \times 2^{-126} = 2^{-149}$. The number 0 is represented by an exponent field consisting of all 0's and a significand field of all 0's. An exponent field consisting of all 1's signals an exception. If all bits in the significand are 0, then it is $\pm\infty$. Otherwise it represents NaN.

William Kahan – is an eminent mathematician, numerical analyst and computer scientist who has made important contributions in the study of accurate and efficient methods of solving numerical problems on a computer with finite precision.
Among his many contributions, Kahan was the primary architect behind the IEEE 754 standard for floating-point computation. Kahan has received many recognitions including the Turing Award in 1989, and with the honor of being named an ACM Fellow in 1994.
Currently, Kahan is professor of mathematics, computer science, and electrical engineering at the University of California, Berkeley, and continues his contributions to the ongoing revision of IEEE 754.

3.5 Rounding

There are four rounding modes in the IEEE standard. If x is a real number that cannot be stored exactly, then it is replaced by a nearby floating point number according to one of the following rules:

- *Round down.* Round(x) is the largest floating point number that is less than or equal to x.
- *Round up.* Round(x) is the smallest floating point number that is greater than or equal to x.
- *Round towards* 0. Round(x) is either round-down(x) or round-up(x), whichever lies between 0 and x. Thus if x is positive then round(x) = round-down(x), while if x is negative then round(x) = round-up(x).
- *Round to nearest.* Round(x) is either round-down(x) or round-up(x), whichever is closer. In case of a tie, it is the one whose least significant (rightmost) bit is 0.

The default is *round to nearest.*

The number $\frac{1}{10} = 1.100\overline{1100}_2 \times 2^{-4}$ is replaced by

0	01111011	10011001100110011001100

using *round down* or *round towards* 0, while it becomes

0	01111011	10011001100110011001101

using *round up* or *round to nearest.* [Note also the exponent field which is the binary representation of 123, or, 127 plus the exponent -4.]

The *absolute rounding error* associated with a number x is defined as $|\text{round}(x) - x|$. In single precision, if $x = \pm(1.b_1 \ldots b_{23}b_{24} \ldots)_2 \times 2^E$, where E is within the range of representable exponents (-126 to 127), then the absolute rounding error associated with x is less than $2^{-23} \times 2^E$ for any rounding mode; the worst rounding errors occur if, for example, $b_{24} = b_{25} = \ldots = b_n = 1$ for some large number n, and round towards 0 is used. For round to nearest, the absolute rounding error is less than or equal to $2^{-24} \times 2^E$, with the worst case being attained if, say, $b_{24} = 1$ and $b_{25} = \ldots = 0$; in this case, if $b_{23} = 0$, then x would be replaced by $1.b_1 \ldots b_{23} \times 2^E$, while if $b_{23} = 1$, then x would be replaced by this number plus $2^{-23} \times 2^E$. Note that 2^{-23} is machine ϵ for single precision. In double and extended precision we have the analogous result that the absolute rounding error is less than $\epsilon \times 2^E$ for any rounding mode, and less than or equal to $\frac{\epsilon}{2} \times 2^E$ for round to nearest.

Usually one is interested not in the absolute rounding error but in the *relative rounding error*, defined as $|\text{round}(x) - x|/|x|$. Since we have seen that $|\text{round}(x) - x| < \epsilon \times 2^E$ when x is of the form $\pm m \times 2^E$, $1 \le m < 2$, it follows that the relative rounding error is less than $\epsilon \times 2^E/(m \times 2^E) \le \epsilon$. For round to nearest, the relative rounding error is less than or equal to $\frac{\epsilon}{2}$. This means that for any real number x (in the range of numbers that can be represented by normalized floating point numbers), we can write

$$\text{round}(x) = x(1 + \delta), \quad \text{where } |\delta| < \epsilon \ \ (\text{or} \le \tfrac{\epsilon}{2} \text{ for round to nearest}).$$

The IEEE standard requires that **the result of an operation (addition, subtraction, multiplication, or division) on two floating point numbers must be the correctly rounded value of the exact result.** For numerical analysts, this is the most important statement in this chapter. It means that if a and b are floating point numbers and $\oplus$, $\ominus$, $\otimes$, and $\oslash$ represent floating point addition, subtraction, multiplication, and division, then we will have

$$\begin{aligned} a \oplus b &= \text{round}(a+b) = (a+b)(1+\delta_1) \\ a \ominus b &= \text{round}(a-b) = (a-b)(1+\delta_2) \\ a \otimes b &= \text{round}(ab) = (ab)(1+\delta_3) \\ a \oslash b &= \text{round}(a/b) = (a/b)(1+\delta_4), \end{aligned}$$

where $|\delta_i| < \epsilon$ (or $\leq \epsilon/2$ for round to nearest), $i = 1, \ldots, 4$. This is important in the analysis of many algorithms.

3.6 Correctly Rounded Floating Point Operations

While the idea of correctly rounded floating point operations sounds quite natural and reasonable (Why would anyone implement incorrectly rounded floating point operations?!), it turns out that it is not so easy to accomplish. Here we will give just a flavor of some of the implementation details.

First consider floating point addition and subtraction. Let $a = m \times 2^E$, $1 \leq m < 2$, and $b = p \times 2^F$, $1 \leq p < 2$, be two positive floating point numbers. If $E = F$, then $a + b = (m+p) \times 2^E$. This result may need further normalization if $m + p \geq 2$. Following are two examples, where we retain three digits after the binary point.

$$1.100_2 \times 2^1 + 1.000_2 \times 2^1 = 10.100_2 \times 2^1 \longrightarrow 1.010_2 \times 2^2,$$

$$1.101_2 \times 2^0 + 1.000_2 \times 2^0 = 10.101_2 \times 2^0 \longrightarrow 1.010_2 \times 2^1.$$

The second example required rounding, and we used round to nearest.

Next suppose that $E \neq F$. In order to add the numbers we must first shift one of the numbers to align the significands, as below:

$$1.100_2 \times 2^1 + 1.100_2 \times 2^{-1} = 1.100_2 \times 2^1 + .011_2 \times 2^1 = 1.111_2 \times 2^1$$

As another example, consider adding the two single precision numbers 1 and $1.11_2 \times 2^{-23}$. This is illustrated below, with the part of the number after the 23rd bit shown on the right of the vertical line:

$$\begin{array}{r|cc} 1.00000000000000000000000 & & \times 2^0 \\ +.00000000000000000000001 & 11 & \times 2^0 \\ \hline 1.00000000000000000000001 & 11 & \times 2^0 \end{array}$$

Using round to nearest, the result becomes $1.0\ldots010_2$.

Now let us consider subtracting the single precision number $1.1\ldots1_2 \times 2^{-1}$ from 1:

1.00000000000000000000000		$\times 2^0$
−.11111111111111111111111	1	$\times 2^0$
0.00000000000000000000000	1	$\times 2^0$

The result is $1.0_2 \times 2^{-24}$, a perfectly good floating point number, so the IEEE standard requires that we compute this number exactly. In order to do this a *guard bit* is needed to keep track of the 1 to the right of the register after the second number is shifted. Cray computers used to get this wrong because they had no guard bit.

It turns out that correctly rounded arithmetic can be achieved using 2 guard bits and a sticky bit to flag some tricky cases. Following is an example of a tricky case. Suppose we wish to subtract $1.0\ldots01_2 \times 2^{-25}$ from 1:

1.00000000000000000000000		$\times 2^0$
−.00000000000000000000000	0100000000000000000000001	$\times 2^0$
0.11111111111111111111111	1011111111111111111111111	$\times 2^0$

Renormalizing and using round to nearest, the result is $1.1\ldots1_2 \times 2^{-1}$. This is the correctly rounded value of the true difference, $1 - 2^{-25} - 2^{-48}$. With only 2 guard bits, however, or with any number less than 25, the computed result is:

1.00000000000000000000000		$\times 2^0$
−.00000000000000000000000	01	$\times 2^0$
0.11111111111111111111111	11	$\times 2^0$

which, using round to nearest, gives the incorrect answer $1.0_2 \times 2^0$. A sticky bit is needed to flag problems of this sort. In practice, floating point operations are often carried out using the 80-bit extended precision registers, in order to minimize the number of special cases that must be flagged.

Floating point multiplication is relatively easy compared to addition and subtraction. The product $(m \times 2^E) \times (p \times 2^F)$ is $(m \times p) \times 2^{E+F}$. This result may need to be renormalized, but no shifting of the factors is required.

3.7 Exceptions

Usually when one divides by 0 in a code, it is due to a programming error, but there are occasions when one would actually like to do this and to work with the result as if it were the appropriate mathematical quantity, namely either $\pm\infty$ if the numerator is nonzero or NaN (not a number) if the numerator is 0. In the past, when a division by 0 occurred, computers would either stop with an error message or continue by setting the result to the largest floating point number. The latter had the unfortunate consequence that two mistakes could cancel each other out: $1/0 - 2/0 = 0$. Now $1/0$ would be set to ∞, $2/0$ would be set to ∞, and the difference, $\infty - \infty$, would be NaN. The quantities $\pm\infty$ and NaN obey the standard mathematical rules; for example, $\infty \times 0 =$ NaN, $0/0 =$ NaN, $\infty + a = \infty$ and $a - \infty = -\infty$ for a a floating point number.

Overflow is another type of exceptional situation. This occurs when the true result of an operation is greater than the largest floating point number ($1.1\ldots1_2 \times 2^{127} \approx 10^{38}$ for single

precision). How this is handled depends on the rounding mode. Using round up or round to nearest, the result is set to ∞; using round down or round towards 0, it is set to the largest floating point number. *Underflow* occurs when the true result is less than the smallest floating point number. The result is stored as a subnormal number if it is in the range of the subnormal numbers, and otherwise it is set to 0.

Exercises

1. Write down the IEEE single precision representation for the following decimal numbers:

   (a) 1.5, using round up.

   (b) 5.1, using round to nearest.

   (c) -5.1, using round towards zero.

   (d) -5.1, using round down.

2. What is the gap between 2 and the next larger single precision number?

3. How many different normalized single precision numbers are there? Express your answer using powers of 2.

4. Give an algorithm to compare two single precision floating point numbers a and b to determine whether $a < b$, $a = b$, or $a > b$, by comparing each of their bits from left to right, stopping as soon as a differing bit is encountered.

5. Consider IEEE single precision floating point arithmetic, using round to nearest. Let a, b, and c be normalized single precision floating point numbers, and let $\oplus$, $\ominus$, $\otimes$, and $\oslash$ denote correctly rounded floating point addition, subtraction, multiplication, and division.

   (a) Is it necessarily true that $a \oplus b = b \oplus a$? Explain why or give an example where this does not hold.

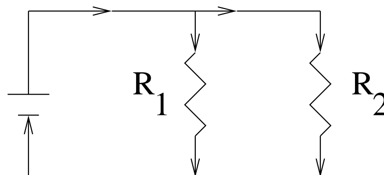
   (b) Is it necessarily true that $(a \oplus b) \oplus c = a \oplus (b \oplus c)$? Explain why or give an example where this does not hold.

   (c) Determine the maximum possible relative error in the computation $(a \otimes b) \oslash c$, assuming that $c \neq 0$. [You may omit terms of order $O(\epsilon^2)$ and higher.] Suppose $c = 0$. What are the possible values that $(a \otimes b) \oslash c$ could be assigned?

6. Let a and b be two positive floating point numbers with the same exponent. Explain why the computed difference $a \ominus b$ is always exact using IEEE arithmetic.

7. The total resistance of an electrical circuit with two resistors connected in parallel, with resistances R_1 and R_2 respectively, is given by the formula

$$T = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}.$$



If $R_1 = R_2 = R$, then the total resistance is $R/2$ since half of the current flows through each resistor. On the other hand, if R_2 is much less than R_1, then most of the current will flow through R_2 but a small amount will still flow through R_1 so the total resistance will be slightly less than R_2. What if $R_2 = 0$? Then T should be 0 since if $R_1 \neq 0$, then all of the current will flow through R_2, and if $R_1 = 0$ then there is no resistance anywhere in the circuit. If the above formula is implemented using IEEE arithmetic, will the correct answer be obtained when $R_2 = 0$? Explain your answer.

Chapter 4

Conditioning of Problems; Stability of Algorithms

4.1 Conditioning of Problems

The *conditioning* of a problem measures how sensitive the answer is to small changes in the input. (Note that this is *independent* of the algorithm used to compute the answer; it is an intrinsic property of the problem.)

Let f be a scalar-valued function of a scalar argument x, and suppose that $\hat{x}$ is close to x. (For example, $\hat{x}$ might be equal to round(x).) How close is $y = f(x)$ to $\hat{y} = f(\hat{x})$? We can ask this question in an *absolute* sense: If

$$|\hat{y} - y| \approx C(x)|\hat{x} - x|,$$

then $C(x)$ is called the *absolute condition number* of the function f at the point x. We also can ask the question in a *relative* sense: If

$$\left|\frac{\hat{y} - y}{y}\right| \approx \kappa(x)\left|\frac{\hat{x} - x}{x}\right|,$$

then $\kappa(x)$ is called the *relative condition number* of f at x.

To determine an expression for $C(x)$, note that

$$\hat{y} - y = f(\hat{x}) - f(x) = \frac{f(\hat{x}) - f(x)}{\hat{x} - x} \cdot (\hat{x} - x),$$

and for $\hat{x}$ very close to x, $(f(\hat{x}) - f(x))/(\hat{x} - x) \approx f'(x)$. Therefore $C(x) = |f'(x)|$. To determine the relative condition number $\kappa(x)$, note that

$$\frac{\hat{y} - y}{y} = \frac{f(\hat{x}) - f(x)}{\hat{x} - x} \cdot \frac{\hat{x} - x}{x} \cdot \frac{x}{f(x)}.$$

Again we use the approximation $(f(\hat{x}) - f(x))/(\hat{x} - x) \approx f'(x)$ to determine

$$\kappa(x) = \left|\frac{xf'(x)}{f(x)}\right|.$$

Examples:

- Let $f(x) = 2x$. Then $f'(x) = 2$, so that $C(x) = 2$ and $\kappa(x) = (2x)/(2x) = 1$. This problem is *well-conditioned.*
- Let $f(x) = \sqrt{x}$. Then $f'(x) = \frac{1}{2}x^{-1/2}$, so that $C(x) = \frac{1}{2}x^{-1/2}$ and $\kappa(x) = 1/2$. This problem is well-conditioned in a relative sense. In an absolute sense, it is well-conditioned if x is not too close to 0, but if, say, $x = 10^{-16}$, then $C(x) = .5 \times 10^8$, so a small absolute change in x (say, changing x from 10^{-16} to 0) results in an absolute change in $\sqrt{x}$ of about 10^8 times the change in x (i.e., from 10^{-8} to 0).
- Let $f(x) = \sin(x)$. Then $f'(x) = \cos(x)$, so that $C(x) = |\cos(x)| \leq 1$ and $\kappa(x) = |x\cot(x)|$. The relative condition number for this function is large if x is near $\pm\pi$, $\pm 2\pi$, etc., and it also is large if $|x|$ is very large and $|\cot(x)|$ is not extremely small. As $x \to 0$, we find $\kappa(x) \to \lim_{x\to 0} |\frac{x\cos x}{\sin x}| = \lim_{x \to 0} |\frac{\cos x - x \sin x}{\cos x}| = 1$.

We will later consider vector-valued functions of a vector of arguments; e.g., finding the solution to a linear system $Ay = b$. We will need to define *norms* in order to measure differences in the input vector $\|b - \hat{b}\|/\|b\|$ and in the output vector $\|y - \hat{y}\|/\|y\|$.

4.2 Stability of Algorithms

Suppose we have a well-conditioned problem and an algorithm for solving this problem. Will our algorithm give the answer to the expected number of places when implemented in floating point arithmetic? To answer this question one may do a *rounding error analysis.*

Example: *Computing sums:* If x and y are two real numbers and they are rounded to floating point numbers and their sum is computed on a machine with unit roundoff ϵ, then

$$\text{fl}(x+y) \equiv \text{round}(x) \oplus \text{round}(y) = (x(1+\delta_1) + y(1+\delta_2))(1+\delta_3), \qquad |\delta_i| \leq \epsilon \quad (\epsilon/2 \text{ for round to nearest}), \tag{4.1}$$

where $\text{fl}(\cdot)$ denotes the floating point result.

Forward error analysis: Here one asks the question: How much does the computed value differ from the exact solution? Multiplying the terms in (4.1), we find

$$\text{fl}(x+y) = x + y + x(\delta_1 + \delta_3 + \delta_1\delta_3) + y(\delta_2 + \delta_3 + \delta_2\delta_3).$$

Again one can ask about the *absolute error*:

$$|\text{fl}(x+y) - (x+y)| \leq (|x| + |y|)(2\epsilon + \epsilon^2),$$

or the *relative error*:

$$\left|\frac{\text{fl}(x+y) - (x+y)}{x+y}\right| \leq \frac{(|x|+|y|)(2\epsilon+\epsilon^2)}{|x+y|}.$$

Note that if $y \approx -x$, then the relative error can be large! The difficulty is due to the initial rounding of the real numbers x and y, not the rounding of their sum. (In fact, their sum may be computed exactly; see exercise 6 in Chapter 3.) In the case where $y \approx -x$, this would be considered an ill-conditioned problem, however, since small changes in x and y can make a large relative change in their sum; the algorithm for adding the two numbers does as well as one could hope.

Backward error analysis: Here one tries to show that the computed value is the exact solution to a nearby problem. If the given problem is ill-conditioned (i.e., if a small change in the input data makes a large change in the solution), then probably the best one can hope for is to compute the exact solution of a problem with slightly different input data. For the problem of summing the two numbers x and y, we have from (4.1)

$$\mathrm{fl}(x+y) = x(1+\delta_1)(1+\delta_3) + y(1+\delta_2)(1+\delta_3),$$

so the computed value is the exact sum of two numbers that differ from x and y by relative amounts no greater than $2\epsilon + \epsilon^2$. The algorithm for adding two numbers is *backward stable.*

Example: Compute $\exp(x)$ using the Taylor series expansion

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots.$$

The following MATLAB code can be used to compute $\exp(x)$:

```
oldsum = 0;
newsum = 1;
term = 1;
n = 0;
while newsum ~= oldsum,  % Iterate until next term is negligible
   n = n + 1;
   term = term * x/n;    % x^n/n! = (x^{n-1}/(n-1)!) * x/n
   oldsum = newsum;
   newsum = newsum + term;
end;
```

This code adds terms in the Taylor series until the next term is so small that adding it to the current sum makes no change in the floating point number that is stored. The code works fine for $x > 0$. The computed result for $x = -20$, however, is $5.6219e-9$; the correct result is $2.0612e-9$. The size of the terms in the series increases to 4.3×10^7 (for $n = 20$) before starting to decrease. This results in double precision rounding errors of size about $4.3 \times 10^7 \times 10^{-16} = 4.3 \times 10^{-9}$, which lead to completely wrong results when the true answer is on the order of 10^{-9}.

Note that the *problem* of computing $\exp(x)$ for $x = -20$ is well-conditioned in both the absolute and relative sense: $C(x)|_{x=-20} = \frac{d}{dx}\exp(x)|_{x=-20} = \exp(-20) << 1$ and $\kappa(x)|_{x=-20} = |x\exp(x)/\exp(x)|_{x=-20} = 20$. Hence the difficulty here is with the *algorithm*; it is, in a sense, *unstable.*

Exercise: How can you modify the above code to produce accurate results when x is negative?

Example: *Numerical differentiation:* Using Taylor's theorem with remainder, one can approximate the derivative of a function $f(x)$:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(\xi), \quad \xi \in [x, x+h].$$

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

The term $-\frac{h}{2} f''(\xi)$ is referred to as the *truncation error* or *discretization error* when the approximation $(f(x+h) - f(x))/h$ is used for $f'(x)$. The truncation error in this case is of order h, denoted $O(h)$, and the approximation is said to be *first order accurate*.

Suppose one computes this finite difference quotient numerically. In the best possible case, one may be able to store x and $x+h$ exactly, and suppose that the only errors made in evaluating $f(x)$ and $f(x+h)$ come from rounding the results at the end. Then, ignoring any other rounding errors in subtraction or division, one computes:

$$\frac{f(x+h)(1+\delta_1) - f(x)(1+\delta_2)}{h} = \frac{f(x+h) - f(x)}{h} + \frac{\delta_1 f(x+h) - \delta_2 f(x)}{h}.$$

Since $|\delta_1|$ and $|\delta_2|$ are less than ϵ, the absolute error due to roundoff is less than or equal to about $2\epsilon |f(x)|/h$, for small h. Note that the truncation error is proportional to h, while the rounding error is proportional to $1/h$. Decreasing h lowers the truncation error but increases the error due to roundoff.

Suppose $f(x) = \sin(x)$ and $x = \pi/4$. Then $f'(x) = \cos(x)$ and $f''(x) = -\sin(x)$, so the truncation error is about $\sqrt{2}h/4$, while the rounding error is about $\sqrt{2}\epsilon/h$. The most accurate approximation is obtained when the two errors are approximately equal:

$$\frac{\sqrt{2}h}{4} = \frac{\sqrt{2}\epsilon}{h} \;\Rightarrow\; h = 2\sqrt{\epsilon}.$$

In this case, the error is on the order of the square root of the machine precision, far less than what one might have hoped for!

Once again, the fault is with the *algorithm*, not the problem of determining $\frac{d}{dx} \sin(x)|_{x=\pi/4} = \cos(x)|_{x=\pi/4} = \sqrt{2}/2$. This problem is well-conditioned since if the input argument $x = \pi/4$ is changed slightly then the answer, $\cos(x)$ changes only slightly; quantitatively, the absolute condition number is $C(x)|_{x=\pi/4} = |-\sin x|_{x=\pi/4} = \sqrt{2}/2$, and the relative condition number is $\kappa(x)|_{x=\pi/4} = |-x \sin x / \cos x|_{x=\pi/4} = \pi/4$. When the problem of evaluating the derivative is replaced by one of evaluating the finite difference quotient, then the conditioning becomes bad. Later we will see how to use higher order accurate approximations to the derivative to obtain somewhat better results, but we will not achieve the full machine precision using finite difference quotients.

Exercises

1. What are the absolute and relative condition numbers of the following functions? Where are they large?

   (a) $(x-1)^\alpha$
   (b) $1/(1+x^{-1})$
   (c) $\ln x$
   (d) $\log_{10} x$
   (e) $x^{-1}e^x$
   (f) $\arcsin x$

2. In evaluating the finite difference quotient $(f(x+h)-f(x))/h$, which is supposed to approximate $f'(x)$, suppose that $x=1$ and $h=2^{-24}$. What would be the computed result using IEEE single precision? Explain your answer.

3. Use MATLAB or your calculator to compute $\tan(x)$, for $x=\frac{\pi}{4}+2\pi\times 10^j$, $j=0,1,2,\ldots,20$. In MATLAB, use "format long e" to print out all of the digits in the answers. What is the relative condition number of this problem (considering x as the input)? Suppose the only error that you make in computing the argument $x=\frac{\pi}{4}+(2\pi)\times 10^j$ occurs because of rounding π to 16 decimal places; i.e., assume that the addition and multiplications and division are done exactly. By what absolute amount will your computed argument differ from the exact one? Use this to explain your results.

4. **Compound interest.** Suppose a_0 dollars are deposited in a bank that pays 5% interest per year, compounded quarterly. After one quarter the value of the account is

$$a_0 \times (1+(.05)/4)$$

   dollars. At the end of the second quarter, the bank pays interest not only on the original amount a_0, but also on the interest earned in the first quarter; thus the value of the investment at the end of the second quarter is

$$[a_0 \times (1+(.05)/4)] \times (1+(.05)/4) = a_0 \times (1+(.05)/4)^2$$

   dollars. At the end of the third quarter the bank pays interest on this amount, so that the account is now worth $a_0 \times (1+(.05)/4)^3$ dollars, and at the end of the whole year the investment is finally worth

$$a_0 \times (1+(.05)/4)^4$$

   dollars. In general, if a_0 dollars are deposited at annual interest rate x, compounded n times per year, then the account value after one year is

$$a_0 \times \mathcal{I}_n(x), \qquad \text{where} \quad \mathcal{I}_n(x) = \left(1+\frac{x}{n}\right)^n.$$

   This is the *compound interest formula*. It is well-known that for fixed x, $\lim_{n\to\infty}\mathcal{I}_n(x) = \exp(x)$.

   (a) Determine the relative condition number $\kappa_{\mathcal{I}_n}(x)$ for the problem of evaluating $\mathcal{I}_n(x)$. For $x=.05$, would you say that this problem is well-conditioned or ill-conditioned?

(b) Use MATLAB to compute $\mathcal{I}_n(x)$ for $x = .05$ and for $n = 1,\ 10,\ 10^2,\ ...,\ 10^{15}$. Use "format long e" so that you can see if your results are converging to $\exp(x)$, as one might expect. Turn in a table with your results and a listing of the MATLAB command(s) you used to compute these results.

(c) Try to explain the results of part (b). In particular, for $n = 10^{15}$, you probably computed 1 as your answer. Explain why. To see what is happening for other values of n, consider the problem of computing z^n when n is large. What is the relative condition number of this problem? If you make an error of about 10^{-16} in computing $z = (1 + x/n)$, about what size error would you expect in z^n?

(d) Can you think of a better way than the method you used in part (b) to accurately compute $\mathcal{I}_n(x)$ for $x = .05$ and for large values of n? Demonstrate your new method in MATLAB or explain why it should give more accurate results.

Chapter 5

Solving Linear Systems and Least Squares Problems

This is one of the longest chapters in this book. That may be surprising considering the fact that most people learn to solve linear systems $Ax = b$ in a linear algebra class. How much more can be said about it?!! The answer is quite a lot; this simple sounding problem really lies at the heart of a great many problems in scientific computing, and it has been the focus of a great deal of effort in the numerical analysis community. Whenever one deals with systems of nonlinear equations or optimization problems, ordinary differential equations, or partial differential equations, (and practically every physical phenomenon that can be modeled is modeled by some combination of these), the algorithms that one uses, at their core, almost always solve systems of linear algebraic equations. For this reason it is important that these solvers be *accurate* and *efficient*. This chapter deals with questions of accuracy and efficiency.

At the beginning of the computer age mathematicians believed that linear systems of size greater than about 40 by 40 would never be solvable by computers — not because it would take too long, but because rounding errors would accumulate and destroy all accuracy. An early paper in 1943 by Hotelling [12] arrived at the conclusion that errors in Gaussian elimination could grow exponentially with the size of the matrix, making the algorithm all but useless for large linear systems. A later analysis in 1947 by Goldstine and von Neumann [20] showed that the algorithm could solve positive definite systems accurately, and so they suggested replacing a general linear system $Ax = b$ by $A^T Ax = A^T b$; we will see in this chapter that this is **not** a good thing to do. Despite these predictions of failure, by the early 1950's engineers were successfully using computers to solve linear systems of larger size, 100 by 100 and greater. It was not until 1961, when Wilkinson applied the idea of *backward error analysis* to the solution of linear systems that the algorithm gained a solid theoretical footing and the earlier pessimism was disspelled [21].

5.1 Review of Matrix Algebra

Let A be an m by n matrix:

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \ldots & a_{mn} \end{pmatrix},$$

and let b be an n-vector:

$$b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}.$$

The product Ab is the m-vector defined by:

$$Ab = \begin{pmatrix} a_{11}b_1 + \ldots + a_{1n}b_n \\ \vdots \\ a_{m1}b_1 + \ldots + a_{mn}b_n \end{pmatrix}.$$

There are other ways to think about the matrix-vector product that often prove useful. The ith entry of Ab is the inner product of the ith row of A with the vector b; that is, if $\vec{a}_{i,:}$ denotes the ith row of A, $(a_{i1}, \ldots, a_{in})$, and if $\langle \cdot, \cdot \rangle$ denotes the inner product, then

$$Ab = \begin{pmatrix} \langle \vec{a}_{1,:}, b \rangle \\ \vdots \\ \langle \vec{a}_{m,:}, b \rangle \end{pmatrix}.$$

The product Ab also can be thought of as a linear combination of the columns of A. If $\vec{a}_{:,j}$ denotes the jth column of A, $(a_{1j}, \ldots, a_{mj})^T$, then

$$Ab = \vec{a}_{:,1}b_1 + \ldots + \vec{a}_{:,n}b_n.$$

If C is an n by p matrix, then the product AC is defined and has dimensions m by p. The (i,j)-entry of AC is: $(AC)_{ij} = \sum_{k=1}^{n} a_{ik}c_{kj}$. The jth column of AC is the product of A with the jth column of C, namely, $\vec{a}_{:,1}c_{1j} + \ldots + \vec{a}_{:,n}c_{nj}$. The ith row of AC is the product of the ith row of A with C, namely, $a_{i1}\vec{c}_{1,:} + \ldots + a_{in}\vec{c}_{n,:}$.

5.2 Gaussian Elimination

Suppose A is an n by n nonsingular matrix. Then for any n-vector b, the linear system $Ax = b$ has a unique solution x, which can be determined by *Gaussian elimination*. Consider the following 3 by 3 linear system:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}.$$

Recall that this is shorthand notation for the system of linear equations:

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 1 \\ 4x_1 + 5x_2 + 6x_3 &= 0 \\ 7x_1 + 8x_2 &= 2 \end{aligned}$$

To solve this linear system using Gaussian elimination, we can append the right-hand side vector to the matrix and then add multiples of one row to another to try to eliminate entries below the diagonal. For example, adding -4 times the first row to the second and -7 times the first row to the third, we obtain:

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 0 & 2 \end{array} \right) \longrightarrow \left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -3 & -6 & -4 \\ 0 & -6 & -21 & -5 \end{array} \right).$$

To eliminate the $(3,2)$-entry, add -2 times the second row to the third:

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -3 & -6 & -4 \\ 0 & 0 & -9 & 3 \end{array} \right).$$

The system is now *upper triangular* and can be solved easily by back substitution. Recalling that the above is shorthand for the system of equations:

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 1 \\ -3x_2 - 6x_3 &= -4 \\ -9x_3 &= 3, \end{aligned}$$

we solve the third equation to obtain $x_3 = -1/3$; substituting this into the second gives $-3x_2 - 6 \cdot (-1/3) = -4$, so that $x_2 = 2$; substituting each of these into the first equation gives $x_1 + 2 \cdot 2 + 3 \cdot (-1/3) = 1$, so that $x_1 = -2$.

Another way to think about the process of Gaussian elimination is as *factoring* the matrix A into the product of a lower and an upper triangular matrix. When we added -4 times the first row to the second and -7 times the first row to the third, what we actually did was to multiply the original matrix on the left by a certain lower triangular matrix L_1:

$$L_1 A \equiv \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ -7 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & -6 & -21 \end{pmatrix}.$$

The matrix L_1 is especially easy to invert: just negate the off-diagonal entries and keep everything else the same:

$$L_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 0 & 1 \end{pmatrix}.$$

Similarly, adding -2 times the second row to the third row in the resulting matrix is equivalent to multiplying on the left by another lower triangular matrix L_2:

$$L_2 L_1 A \equiv \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & -6 & -21 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -9 \end{pmatrix}.$$

The inverse of L_2 is also obtained by just negating the off-diagonal entries, and the product $(L_2 L_1)^{-1} = L_1^{-1} L_2^{-1}$ is easy to compute as well: The diagonal entries are 1's and the nonzero off-diagonal entries are those of L_1^{-1} together with those of L_2^{-1}; that is,

$$(L_2 L_1)^{-1} = L_1^{-1} L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{pmatrix}.$$

Letting U denote the upper triangular matrix obtained from A, we have $L_2 L_1 A = U$, so that $A = LU$, where L is the lower triangular matrix $(L_2 L_1)^{-1}$.

If one wishes to solve several linear systems with the same coefficient matrix but different right-hand side vectors, then one can save the lower and upper triangular factors L and U. Then, to solve $Ax \equiv LUx = b$, one first solves the lower triangular system $Ly = b$ (to obtain $y = Ux$) and then the upper triangular system $Ux = y$.

Let us generalize the above to an arbitrary n by n matrix A, and see how a Gaussian elimination routine might be written in MATLAB. The first step is to add multiples of row 1 to each of the other rows in order to eliminate entries below the diagonal in the first column of the matrix. We will apply these same operations to the right-hand side vector b:

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & \ldots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} & b_n \end{array} \right) \longrightarrow \left(\begin{array}{cccc|c} a_{11} & a_{12} & \ldots & a_{1n} & b_1 \\ 0 & a_{22}^{(1)} & \ldots & a_{2n}^{(1)} & b_2^{(1)} \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & a_{n2}^{(1)} & \ldots & a_{nn}^{(1)} & b_n^{(1)} \end{array} \right).$$

The MATLAB code for this step can be written as follows:

```
% Step 1 of Gaussian elimination.
for i=2:n                           % Loop over rows below the diagonal.
  mult = A(i,1)/A(1,1);             % Subtract this multiple of row 1 from
                                    % row i to make A(i,1)=0.
  A(i,:) = A(i,:) - mult*A(1,:); % This is equivalent to the for loop:
                                    %   for k=1:n,
                                    %     A(i,k) = A(i,k) - mult*A(1,k);
                                    %   end;
  b(i) = b(i) - mult*b(1);
end;
```

The next step is to add multiples of row 2 in the modified matrix to rows 3 through n in order to eliminate entries below the diagonal in the second column of the matrix. Again these same operations are applied to b. In general, the jth step will operate on rows $j+1$ through n, adding multiples of row j to these rows in order to eliminate entries below the diagonal in column j:

$$\left(\begin{array}{cccccccc} 0 & \ldots & 0 & a_{jj}^{(j-1)} & a_{j,j+1}^{(j-1)} & \ldots & a_{jn}^{(j-1)} & | & b_j^{(j-1)} \\ 0 & \ldots & 0 & a_{j+1,j}^{(j-1)} & a_{j+1,j+1}^{(j-1)} & \ldots & a_{j+1,n}^{(j-1)} & | & b_{j+1}^{(j-1)} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & \ldots & 0 & a_{nj}^{(j-1)} & a_{n,j+1}^{(j-1)} & \ldots & a_{nn}^{(j-1)} & | & b_n^{(j-1)} \end{array} \right) \longrightarrow$$

$$\left(\begin{array}{cccccccc} 0 & \ldots & 0 & a_{jj}^{(j-1)} & a_{j,j+1}^{(j-1)} & \ldots & a_{jn}^{(j-1)} & | & b_j^{(j-1)} \\ 0 & \ldots & 0 & 0 & a_{j+1,j+1}^{(j)} & \ldots & a_{j+1,n}^{(j)} & | & b_{j+1}^{(j)} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & \ldots & 0 & 0 & a_{n,j+1}^{(j)} & \ldots & a_{nn}^{(j)} & | & b_n^{(j)} \end{array} \right).$$

To accomplish this, the above MATLAB code (slightly modified) must be surrounded by an outer loop over columns $j = 1, \ldots, n-1$. The result is:

```
% Gaussian elimination without pivoting.
for j=1:n-1                         % Loop over columns.
  for i=j+1:n                       % Loop over rows below j.
    mult = A(i,j)/A(j,j);           % Subtract this multiple of row j from
                                    % row i to make A(i,j)=0.
    A(i,:) = A(i,:) - mult*A(j,:); % This does more work than necessary;
                                       % do you see why?
    b(i) = b(i) - mult*b(j);
  end;
end;
```

Recall that at the jth stage of Gaussian elimination, the entries in columns 1 through $j-1$ of rows j through n are zero, so that subtracting `mult` times row j from row i ($i > j$) does not alter these zero entries. Therefore the above code could be made more efficient by replacing the line `A(i,:) = A(i,:) - mult*A(j,:);` by

```
A(i,j:n) = A(i,j:n) - mult*A(j,j:n);
```

This operates only on entries j through n of row i.

The above Gaussian elimination code may fail! The problem is that, in order to use row j to eliminate entries in column j, it must be the case that the (j,j)-entry of the (now modified) matrix is nonzero; i.e., the denominator in the expression for `mult` must be nonzero. To ensure this, we will use a procedure called *partial pivoting* that will be described in Section 5.2.3.

5.2.1 Operation Counts

The timing of a computer code depends on many things, one of which is the amount of arithmetic that the code does. This used to be the primary factor determining timing, but with faster arithmetic operations available on today's computers, other factors come into play such as the frequency of memory references and the amount of work that can be done in parallel. These other factors will be mentioned in Section 5.2.4, but, for now, let us count the number of floating point operations (additions, subtractions, multiplications, and divisions) performed by our Gaussian elimination code. It should be noted that historically, only multiplications and divisions were counted, because these operations were significantly slower than additions and subtractions. Today, on most computers, these operations take about the same amount of time, so we will be concerned with the total number of operations.

In the above Gaussian elimination code, we loop over j going from 1 to $n-1$. Therefore the number of operations performed will be $\sum_{j=1}^{n-1}$ (number of operations performed at stage j). At each stage j, we loop over i going from $j+1$ to n, so the amount of work becomes $\sum_{j=1}^{n-1}\sum_{i=j+1}^{n}$ (number of operations performed on row i at stage j). Now for each row i, we compute `mult`, which requires one division. We then subtract `mult` times row j from row i, which requires n multiplications and n subtractions (using the above code, without the improvement for efficiency). Finally, we subtract `mult` times `b(j)` from `b(i)`, which requires one multiplication and one subtraction. Thus the total number of operations on row i at stage j is $1+2n+2=2n+3$. Summing over i and j now, we find the total number of floating point operations performed by our Gaussian elimination code:

$$\begin{aligned}\sum_{j=1}^{n-1}\sum_{i=j+1}^{n}(2n+3) &= \sum_{j=1}^{n-1}(n-j)(2n+3) \\ &= (2n+3)\sum_{j=1}^{n-1}(n-j) = (2n+3)\sum_{k=1}^{n-1}k \\ &= (2n+3)\frac{n(n-1)}{2} = n^3 + O(n^2).\end{aligned}$$

We are usually interested in the timing of the code when n is large, say $n \geq 1000$, since the code runs very quickly for small n. Therefore, it is customary to determine the highest power of n in the operation count and the constant multiplying that highest power of n and to write the remaining lower order terms using the $O(\cdot)$ notation, since they will be insignificant for large n. Sometimes the above operation count is written simply as $O(n^3)$, to indicate that it depends on the cube of the matrix size, but it is also important to know the constant multiplying n^3, which in this case is 1.

It was noted above that the efficiency of our Gaussian elimination code could be improved by replacing the line `A(i,:) = A(i,:) - mult*A(j,:)` by one that operates only on entries j through n of row i. Since this requires only $2(n-j+1)$ operations instead of $2n$ operations, the work count

becomes

$$\begin{aligned}
\sum_{j=1}^{n-1}\sum_{i=j+1}^{n}[2(n-j)+5] &= \sum_{j=1}^{n-1}[2(n-j)^2+5(n-j)] \\
&= 2\sum_{k=1}^{n-1}k^2+5\sum_{k=1}^{n-1}k. \qquad (5.1)
\end{aligned}$$

Now to compute the operation count, we must use the formula for the sum of the first $n-1$ integers, as well as their squares. These formulas are:

$$\sum_{k=1}^{m}k=\frac{m(m+1)}{2}, \qquad \sum_{k=1}^{m}k^2=\frac{m(m+1)(2m+1)}{6}. \qquad (5.2)$$

Making these substitutions in (5.1) gives the total operation count:

$$2\frac{(n-1)n(2n-1)}{6}+5\frac{(n-1)n}{2}=\frac{2}{3}n^3+O(n^2).$$

The work is still $O(n^3)$, but the constant multiplying n^3 has been reduced from 1 to 2/3.

The first formula in (5.2) is easy to remember: it says that the sum of the first m integers is their average value, $(m+1)/2$, times the number of terms, m. The second formula and formulas for sums of higher powers are less easy to remember, but their highest order terms can be derived as follows. Using rectangles to approximate the area under the curve x^p, for p a positive integer, we find

$$\int_0^m x^p\,dx \le 1^p+2^p+\ldots+m^p \le \int_1^{m+1}x^p\,dx.$$

These inequalities are illustrated below:

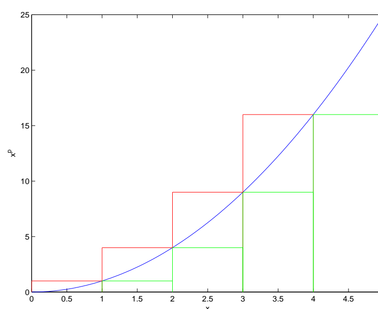


Figure 5.1: Area of red rectangles is an upper bound on $\int_0^4 x^p\,dx$, while area of green rectangles (which is the same as that of the red rectangles) is a lower bound on $\int_1^5 x^p\,dx$.

Since $\int_0^m x^p\,dx = m^{p+1}/(p+1)$ and $\int_1^{m+1} x^p\,dx = ((m+1)^{p+1}-1)/(p+1) = m^{p+1}/(p+1) + O(m^p)$, we have

$$\sum_{k=1}^{m} k^p = \frac{m^{p+1}}{p+1} + O(m^p).$$

5.2.2 *LU* Factorization

We previously interpreted Gaussian elimination as factoring the matrix in the form $A = LU$, where L is lower triangular with 1's on its diagonal (often called a unit lower triangular matrix) and U is upper triangular. It is easy to modify our Gaussian elimination code to save the L and U factors so that they can be used to solve linear systems with the same coefficient matrix but different right-hand side vectors. In fact, this can be done without using any extra storage, because the entries in the strict lower triangle of A (the part of the matrix below the main diagonal) are zeroed out during Gaussian elimination. Therefore this space could be used to store the strict lower triangle of L, and since the diagonal entries of L are always 1, there is no need to store these. The entries of L are simply the multipliers (the variable called `mult` in the previous code) used to eliminate entries in the lower triangle of A. Therefore the Gaussian elimination code can be modified as follows:

```
%  LU factorization without pivoting.
for j=1:n-1                       % Loop over columns.
  for i=j+1:n                     % Loop over rows below j.
    mult = A(i,j)/A(j,j);         % Subtract this multiple of row j from
                                  % row i to make A(i,j)=0.
    A(i,j+1:n) = A(i,j+1:n) - mult*A(j,j+1:n); % This works on columns j+1
                                               % through n.
    A(i,j) = mult;                % Since A(i,j) becomes 0, use the space
                                  % to store L(i,j).
  end;
end;
```

As noted previously, once A has been factored in the form LU, we can solve a linear system $Ax \equiv LUx = b$ by first solving $Ly = b$ and then solving $Ux = y$. To solve $Ly = b$, we solve the first equation for y_1, then substitute this value into the second equation in order to find y_2, etc., as shown below:

$$\begin{pmatrix} \ell_{11} & 0 & \dots & 0 \\ \ell_{21} & \ell_{22} & & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \ell_{n1} & \ell_{2n} & \dots & \ell_{nn} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \Longrightarrow$$

$$
\begin{aligned}
y_1 &= b_1/\ell_{11} \\
y_2 &= (b_2 - \ell_{21} y_1)/\ell_{22} \\
\vdots & \quad \vdots \\
y_i &= (b_i - \sum_{j=1}^{i-1} \ell_{ij} y_j)/\ell_{ii}
\end{aligned}
$$

The procedure simplifies slightly when the diagonal entries of L are all 1's. The following MATLAB function solves $Ly = b$, when L has unit diagonal and the strict lower triangle of L is stored:

```
function y = lsolve(L, b)
%
%  Given a lower triangular matrix L with unit diagonal and a vector b,
%  this routine solves Ly = b and returns the solution y.
%
n = length(b);                  % Determine size of problem.
for i=1:n,                      % Loop over equations.
  y(i) = b(i);                  % Solve for y(i) using
  for j=1:i-1,                  % previously computed y(j),j=1,...,i-1.
    y(i) = y(i) - L(i,j)*y(j);
  end;
end;
```

Since one subtraction and one multiplication are performed inside the innermost loop of this routine, the total operation count is

$$\sum_{i=1}^{n} \sum_{j=1}^{i-1} 2 = \sum_{i=1}^{n} 2(i-1) = 2 \sum_{k=1}^{n-1} k = n(n-1) = n^2 + O(n).$$

Thus, once the matrix has been factored, a triangular solve can be carried out with just $O(n^2)$ operations instead of the $O(n^3)$ required for factorization. This means that if n is large and one saves the LU factors of A, then one can solve with many different right-hand side vectors for almost the same cost as solving with one.

It is left as an exercise to write a MATLAB routine to solve the upper triangular system $Ux = y$ and to count the number of operations performed.

5.2.3 Pivoting

As noted previously, the codes that we have given so far may fail. If, at the jth step of Gaussian elimination, the (j, j)-entry of the modified matrix is zero, then that row cannot be used to eliminate nonzero entries in the jth column. If the (j, j)-entry is zero, then we could use a different row, say row $k > j$, whose jth element is nonzero in order to eliminate other nonzeros in column j. For

convenience, we could interchange rows k and j (remembering also to interchange the right-hand side entries b_k and b_j), so that we maintain the upper triangular form. This process of interchanging rows is sometimes called *pivoting*, and the nonzero entry a_{kj} is called the *pivot element*.

Suppose there are no nonzero entries on or below the main diagonal in column j. Then the matrix must be singular. Do you see why? It would mean that after $j-1$ steps of Gaussian elimination, each of the first j columns of the matrix have nonzeros only in their top $j-1$ positions. Now any j vectors of length $j-1$ must be linearly dependent. This implies that the first j columns of the matrix are linearly dependent and so the matrix is singular. Recall that if A is singular then the linear system $Ax = b$ does not have a unique solution; it has either no solutions (if b is outside the range of A) or infinitely many solutions (if b is in the range of A, in which case the solution set consists of any specific solution plus anything from the null space of A). In this case, it would be reasonable for our Gaussian elimination code to return an error message stating that the matrix is singular.

One possible approach to fixing the Gaussian elimination code, then, would be to modify the code to test if a pivot element (the denominator in `mult`) is zero. If it is, then search through that column (below the main diagonal) to find a nonzero entry. When a nonzero entry is found, interchange that row with the pivot row and then do the elimination. If no nonzero entry is found in the column, then return with an error message.

In exact arithmetic, this strategy would be fine, but in finite precision arithmetic it is not a good idea. Suppose a certain pivot element would be zero in exact arithmetic but because of rounding errors, the computed value is some tiny number, say, the machine precision ϵ. We would use this tiny (and incorrect) number to eliminate entries in other rows by adding huge multiples of the pivot row to these other rows. The results could be completely wrong. Even if there is no error in the pivot entry, adding huge multiples of the pivot row to other rows can result in significant errors, as can be seen with the following linear system:

$$\begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix},$$

The exact solution is very nearly $x_1 = x_2 = 1$. (Precisely, it is $x_2 = 1 - 1/(10^{20} - 1)$, $x_1 = 1 + 1/(10^{20} - 1)$.) Using our Gaussian elimination code with double precision arithmetic, the result after the first stage would be:

$$\left(\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 1 & 1 & 2 \end{array} \right) \longrightarrow \left(\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & -10^{20} & -10^{20} \end{array} \right),$$

since `mult` would be 10^{20}, and $1 - 10^{20}$ and $2 - 10^{20}$ would each be rounded to -10^{20}. Solving the upper triangular system we would then obtain $x_2 = 1$ and $10^{-20} x_1 + 1 = 1$ so $x_1 = 0$.

This wrong answer was easily avoidable! Had we simply interchanged the two rows and used the second row as the pivot row, we would have obtained:

$$\left(\begin{array}{cc|c} 1 & 1 & 2 \\ 10^{-20} & 1 & 1 \end{array} \right) \longrightarrow \left(\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 1 & 1 \end{array} \right),$$

where `mult` is 10^{-20} and the numbers $1 - 10^{-20}$ and $1 - 2 \times 10^{-20}$ were each rounded to 1. The solution to this upper triangular system is: $x_2 = 1$ and $x_1 + 1 = 2$ so that $x_1 = 1$.

To avoid difficulties with tiny pivot elements, a strategy called *partial pivoting* is generally used. Here one searches for the *largest* entry in the column in absolute value and uses that as the pivot element. This ensures that the multipliers of the pivot row used to eliminate in other rows are all less than or equal to 1 in absolute value. As an example, we consider the 3 by 3 problem given at the beginning of this section:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}.$$

Using partial pivoting, we would reduce this problem to "upper triangular" form as follows:

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 0 & 2 \end{array}\right) \overset{\text{pivot}}{\longrightarrow} \left(\begin{array}{ccc|c} 7 & 8 & 0 & 2 \\ 4 & 5 & 6 & 0 \\ 1 & 2 & 3 & 1 \end{array}\right) \longrightarrow \left(\begin{array}{ccc|c} 7 & 8 & 0 & 2 \\ 0 & 3/7 & 6 & -8/7 \\ 0 & 6/7 & 3 & 5/7 \end{array}\right)$$

$$\overset{\text{pivot}}{\longrightarrow} \left(\begin{array}{ccc|c} 7 & 8 & 0 & 2 \\ 0 & 6/7 & 3 & 5/7 \\ 0 & 3/7 & 6 & -8/7 \end{array}\right) \longrightarrow \left(\begin{array}{ccc|c} 7 & 8 & 0 & 2 \\ 0 & 6/7 & 3 & 5/7 \\ 0 & 0 & 9/2 & -3/2 \end{array}\right).$$

Our Gaussian elimination code can be modified to incorporate partial pivoting as follows:

```
% Gaussian elimination with partial pivoting.
for j=1:n-1                        % Loop over columns.

  [pivot,k] = max(abs(A(j:n,j)));  % Find the pivot element in column j.
                                   %  pivot is the largest absolute
                                   %  value of an entry; k+j-1 is its
                                   %  index.
  if pivot==0,                     % If all entries in the column are 0,
    disp(' Matrix is singular.')   % return with an error message.
    break;
  end;
  temp = A(j,:);                   % Otherwise,
  A(j,:) = A(k+j-1,:);             % Interchange rows j and k+j-1.
  A(k+j-1,:) = temp;
  tempb = b(j);
  b(j) = b(k+j-1);
  b(k+j-1) = tempb;

  for i=j+1:n                      % Loop over rows below j.
    mult = A(i,j)/A(j,j);          % Subtract this multiple of row j from
```

```
                                    % row i to make A(i,j)=0.
    A(i,j:n) = A(i,j:n) - mult*A(j,j:n);
    b(i) = b(i) - mult*b(j);
  end;
end;
```

Partial pivoting can also be incorporated into the version of the code that saves the matrix factors. Now, instead of factoring A in the form LU, we factor it in the form PLU, where P is a permutation matrix. A permutation matrix is a matrix that has a 1 in each row and each column and all other elements 0. Exchanging rows can be thought of as applying a permutation matrix on the left. The following theorem guarantees that this factorization exists for any nonsingular matrix A:

Theorem 5.2.1. *Every n by n nonsingular matrix A can be factored in the form $A = PLU$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is an upper triangular matrix.*

We do not want to leave the reader with the impression that to solve a linear system $Ax = b$ in MATLAB one must type in the above code! To solve $Ax = b$, you type `A\b`. The above is (roughly) what MATLAB does when it solves linear systems or computes PLU factorizations.

5.2.4 Banded Matrices and Matrices for which Pivoting is Not Required

For certain matrices, pivoting in Gaussian elimination is not required. For example, it can be shown that for *symmetric positive definite* matrices, Gaussian elimination can be performed stably without pivoting. A matrix A is *symmetric* if $A = A^T$, and a symmetric matrix is *positive definite* if all of its eigenvalues are positive. An example of a symmetric positive definite matrix is

$$\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}, \tag{5.3}$$

whose eigenvalues are 2, $2 + \sqrt{2}$, and $2 - \sqrt{2}$.

Instead of factoring a symmetric positive definite matrix in the form LU, where L has unit diagonal, one can factor it in such a way that L and U have the same diagonal elements. With this scaling it turns out that U is just the transpose of L, and so we have factored A in the form $A = LL^T$. This is called the **Cholesky decomposition**.

Gaussian elimination without pivoting is also known to be stable for *strictly diagonally dominant* matrices. A matrix A is strictly diagonally dominant (or strictly *row* diagonally dominant) if

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, \quad \text{for all } i;$$

that is, the absolute value of each diagonal entry is greater than the sum of the absolute values of all off-diagonal entries in its row.

A number of applications give rise to linear systems that are symmetric positive definite or, occasionally, strictly diagonally dominant, and *banded*; that is, $a_{ij} = 0$, if $|i - j| > m$, where $m << n$ is the half bandwidth. The matrix in (5.3) has half bandwidth $m = 1$. The full bandwidth is $2m + 1 = 3$ and so this matrix is called a *tridiagonal* matrix.

We can save work and storage in the Gaussian elimination algorithm by taking advantage of the band structure. Consider a matrix with half bandwidth m. To perform Gaussian elimination without pivoting, we use the first row to eliminate entries in column 1 of rows 2 through n. But the entries in column 1 of rows $m+2$ through n are already zero, so we need only operate on the m rows with nonzeros in column 1. Moreover, we need not operate on the entire row. Only columns 1 through $m+1$ will be affected by adding a multiple of row 1. Thus, the work at the first stage of Gaussian elimination is reduced from $2(n-1)n$ to $2m(m+1)$, and the bandwidth of the matrix has not increased. This is illustrated below for a matrix with half bandwidth 2:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{pmatrix} \longrightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ \mathbf{0} & \tilde{\mathbf{a}}_{\mathbf{22}} & \tilde{\mathbf{a}}_{\mathbf{23}} & a_{24} & 0 & 0 \\ \mathbf{0} & \tilde{\mathbf{a}}_{\mathbf{32}} & \tilde{\mathbf{a}}_{\mathbf{33}} & a_{34} & a_{35} & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{pmatrix}$$

In general, at stage j of Gaussian elimination, we must eliminate the entry in column j from rows $j+1$ through n. But for a banded matrix, only rows $j+1$ through $\min\{j+m, n\}$ have nonzero entries in column j, and only the entries in columns j through $\min\{j+m, n\}$ of these rows will be affected by adding a multiple of row j. The total work for banded Gaussian elimination without pivoting therefore becomes

$$\sum_{j=1}^{n-1} 2\min\{m, n-j\} \cdot \min\{m+1, n-j+1\} \approx 2m^2 n.$$

For $m << n$, this is a large savings over the $O(n^3)$ operations required for dense Gaussian elimination. Note, however, that while we were able to take advantage of the band structure in Gaussian elimination, we could not take advantage of any zeros within the band. Frequently, matrices arising from partial differential equations are banded with half bandwidth $m \approx \sqrt{n}$ (for 2-D problems), but most of the entries inside the band are also zero. During the process of Gaussian elimination, these entires inside the band *fill in* with nonzeros and so they must be stored and operated on.

Suppose we need to use partial pivoting. Can we take advantage of a band structure using Gaussian elimination with partial pivoting? The answer is yes; the half bandwidth of the U factor is at most *doubled* by using partial pivoting. To see this consider, for example, a tridiagonal matrix

$$\begin{pmatrix} x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 \\ 0 & x & x & x & 0 \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \end{pmatrix},$$

where the x's represent nonzeros. At the first stage, row 1 or 2 will be chosen as the pivot row. If row 1 is chosen, the bandwidth does not increase, but if row 2 is chosen and moved to the top, the half bandwidth increases from 1 to 2, and the zero entry in column 3 of the current row 1 fills in with a nonzero value. The matrix takes the form

$$\begin{pmatrix} x & x & x & 0 & 0 \\ 0 & x & x & 0 & 0 \\ 0 & x & x & x & 0 \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \end{pmatrix}.$$

Now row 2 or 3 can be chosen as the pivot row. If row 2 is chosen then no fill-in occurs, but if row 3 is chosen then when it is moved up the half bandwidth in row 2 becomes 2 as well, and during the second stage of elimination the zero in column 4 of the current row 2 will become nonzero:

$$\begin{pmatrix} x & x & x & 0 & 0 \\ 0 & x & x & x & 0 \\ 0 & 0 & x & x & 0 \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \end{pmatrix}.$$

Continuing in this way, it is not difficult to see that each row interchange causes one additional zero to fill in. If row interchanges are made at every step, the the final matrix U will have the form

$$\begin{pmatrix} x & x & x & 0 & 0 \\ 0 & x & x & x & 0 \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & x \end{pmatrix}.$$

Exercise: Estimate the number of operations required for Gaussian elimination with partial pivoting for a matrix with half bandwidth m.

5.2.5 Implementation Considerations for High Performance*

It was mentioned earlier that *memory references* may be costly on today's supercomputers. Usually there is a memory hierarchy, with larger storage devices such as the main memory or disk being slower to access than smaller storage areas such as local cache and registers. When an arithmetic operation is performed, the operands are fetched from storage, loaded into the registers where the arithmetic is performed, and then the result is stored back into memory. The more work that can be done on a word fetched from memory before storing the result, the better. Also, it is usually more efficient to fetch and store large chunks of data at a time.

Consider the Gaussian elimination code of Subsection 5.2.3. To eliminate the (i, j) entry at stage j, one fetches the pivot row j and the ith row, $i > j$, from memory and then operates on row i and stores the result. This requires $3(n-j+1)$ memory accesses and only about $2(n-j+1)$ arithmetic

operations (multiplication of row j and subtraction from row i). If the time for a memory access is greater than that for an arithmetic operation, then most of the time will be spent in fetching operands and storing results.

On the other hand, consider the following basic linear algebra operations:

1. Matrix-vector multiplication: $y \leftarrow Ax + y$, where A is an n by n matrix and x and y are n-vectors. One must fetch the n^2 entries of A and the $2n$ entries of x and y and store the n entries of the result, for a total of $n^2 + 3n$ memory accesses. The arithmetic to compute the product of matrix A with vector x and add the result to y is about $2n^2$. Thus the ratio of arithmetic operations to memory fetches is about 2.

2. Matrix-matrix multiplication: $C \leftarrow AB + C$, where A, B, and C are n by n matrices. Here one must fetch the $3n^2$ entries of A, B, and C, and one must store the n^2 entries of the result, for a total of $4n^2$ memory accesses. But the work to compute the product AB and add it to C is about n times the work to perform the matrix-vector multiplication above, namely, $2n^3$. Thus the ratio of arithmetic operations to memory fetches is about $n/2$.

If Gaussian elimination could be expressed in terms of matrix-vector or, better, matrix-matrix multiplication, then the ratio of arithmetic operations to memory accesses would be increased, so that more time would be spent in actually doing the work rather than in fetching the data and storing the results. These basic linear algebra operations are implemented in a package called the BLAS (Basic Linear Algebra Subroutines) [5]. Operations, such as the one performed in our Gaussian elimination code, that act on two vectors (e.g., adding a scalar times one row to another), are labeled BLAS1 and are the least efficient operations when memory references are costly. Matrix-vector operations comprise the BLAS2 routines, which are intermediate in their efficiency, and matrix-matrix operations comprise the BLAS3 routines, which are the most efficient.

Block Algorithms

The Gaussian elimination algorithm can be arranged differently. Suppose we divide the matrix A into a p by p array of m by m blocks A_{ij}, where $n = mp$:

$$A = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1p} \\ A_{21} & A_{22} & \dots & A_{2p} \\ \vdots & \vdots & & \vdots \\ A_{p1} & A_{p2} & \dots & A_{pp} \end{pmatrix}. \tag{5.4}$$

Assuming for the moment that pivoting is not required, one can eliminate entires below the diagonal in block column one by computing

$$\begin{pmatrix} A_{22} & \dots & A_{2p} \\ \vdots & & \vdots \\ A_{p2} & \dots & A_{pp} \end{pmatrix} - \begin{pmatrix} A_{21} \\ \vdots \\ A_{p1} \end{pmatrix} A_{11}^{-1}(A_{12}, \dots, A_{1p}).$$

This matrix is called the *Schur complement* of A_{11} in A. Instead of explicitly computing A_{11}^{-1}, one factors it in the form $A_{11} = L_{11}U_{11}$ and then solves triangular linear systems with multiple right-hand sides. This can be implemented as follows. First bring in the first block column and perform Gaussian elimination on that rectangular matrix. This corresponds to a factorization of the form

$$\begin{pmatrix} A_{11} \\ A_{21} \\ \vdots \\ A_{p1} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \\ \vdots \\ L_{p1} \end{pmatrix} U_{11}. \tag{5.5}$$

Here L_{11} and U_{11} are the unit lower triangular and upper triangular factors of A_{11}, while $L_{21}, \ldots, L_{p1}$ are not lower triangular but contain the multipliers used in the elimination of blocks $A_{21}, \ldots, A_{p1}$. This step requires about $2nm$ memory accesses, while the work is about nm^2. To update the rest of the matrix, one then computes:

$$\begin{pmatrix} A_{22} & \ldots & A_{2p} \\ \vdots & & \vdots \\ A_{p2} & \ldots & A_{pp} \end{pmatrix} - \begin{pmatrix} L_{21} \\ \vdots \\ L_{p1} \end{pmatrix} L_{11}^{-1}(A_{12}, \ldots, A_{1p}), \tag{5.6}$$

where $L_{11}^{-1}(A_{12}, \ldots, A_{1p})$ is computed by solving the lower triangular multiple right-hand side system $L_{11}(U_{12}, \ldots, U_{1p}) = (A_{12}, \ldots, A_{1p})$. This is a matrix-matrix (BLAS3) operation. It requires about $2n^2$ memory accesses (assuming that $m << n$) and about $2n^2m$ arithmetic operations. Elimination in successive block columns is handled similarly.

A software package that performs Gaussian elimination, as well as many other linear algebra operations, using block algorithms is called *LAPACK* [3]. It is the software on which MATLAB is based.

Parallelism

A *parallel computer* contains more than one processor, and different processors are capable of working on the same problem simultaneously. Ideally, one might hope that with p processors, an algorithm would run p times as fast. This ideal speedup is seldom achieved, however, even with an algorithm that appears to be highly parallelizable, because of the overhead of communicating between processors. This represents another kind of memory hierarchy. On a *distributed memory* machine, a processor may access data stored in its own memory quickly, but if the data that it needs is on another processor, the time to retrieve it can be much greater.

Gaussian elimination offers many opportunities for parallelism, since elimination in each row (or block row) can be performed independently and in parallel. At a finer level, individual entries in a row could be updated simultaneously. One important reason for using parallel machines, besides improving execution time, is to have more memory. A matrix may be too large to fit in the memory of a single processor, but if it is divided into blocks as in (5.4), then the different blocks can be distributed among the processors. They should be distributed in such a way as to minimize communication costs.

As an example, but not necessarily the most efficient example, of how Gaussian elimination might be parallelized, suppose that we have p processors and each stores one of the block columns in (5.4). Processor 1 works on its block column, performing the operation in (5.5). It then communicates the results, $L_{11}, \ldots, L_{p1}$, to the other processors, which update their block columns simultaneously according to (5.6). Note that with this arrangement of data, each processor needs all of the blocks $L_{11}, \ldots, L_{p1}$ in order to update its block column. When processor 2 has completed its update, it can begin the analogous process to (5.5) for block column 2. However, all processors must wait until processor 2 completes this work, before starting to do the update analogous to (5.6) for the second stage. In this way, many processors operate simultaneously, but they also must wait for others in order to remain in step.

A library known as *ScaLAPACK* [4] (for Scalable LAPACK) implements Gaussian elimination and other linear algebra algorithms for distributed memory parallel machines.

5.3 Other Methods for Solving $Ax = b$

You might wonder why we have concentrated on Gaussian elimination for solving $Ax = b$. You probably have learned some other ways. For example, one might compute A^{-1} and set $x = A^{-1}b$. You also may have been taught Cramer's rule for solving linear systems. This is a useful way to solve small linear systems by hand, but, as we will see, it is completely inappropriate for large linear systems.

We will concentrate here on operation counts. Recall that Gaussian elimination requires about $\frac{2}{3}n^3$ operations to factor a matrix A in the form $A = PLU$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is an upper triangular matrix. To solve a linear system $Ax = b$ requires an additional $2n^2$ operations to solve the triangular systems $Ly = b$ and $Ux = y$.

1. **Compute $\mathbf{A^{-1}}$ and set $\mathbf{x = A^{-1}b}$.** Sometimes people are taught to solve linear systems this way, but it is more work (and more prone to mistakes!) than Gaussian elimination, even if you are solving a small problem by hand. Consider, for example, the linear system

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}.$$

A standard procedure for computing the inverse of A involves appending the identity matrix to A, eliminating entries below the diagonal in A, scaling the diagonal elements to be 1, and then eliminating entries above the diagonal in A. The same operations are performed on the appended matrix, and when the procedure is finished, the appended matrix is A^{-1}. Following is the procedure applied to our matrix:

$$\left(\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 4 & 5 & 6 & 0 & 1 & 0 \\ 7 & 8 & 0 & 0 & 0 & 1 \end{array}\right) \rightarrow \left(\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & -3 & -6 & -4 & 1 & 0 \\ 0 & -6 & -21 & -7 & 0 & 1 \end{array}\right) \rightarrow$$

$$\left(\begin{array}{rrr|rrr} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & -3 & -6 & -4 & 1 & 0 \\ 0 & 0 & -9 & 1 & -2 & 1 \end{array}\right) \to \left(\begin{array}{rrr|rrr} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 1 & 2 & 4/3 & -1/3 & 0 \\ 0 & 0 & 1 & -1/9 & 2/9 & -1/9 \end{array}\right) \to$$

$$\left(\begin{array}{rrr|rrr} 1 & 2 & 0 & 4/3 & -2/3 & 1/3 \\ 0 & 1 & 0 & 14/9 & -7/9 & 2/9 \\ 0 & 0 & 1 & -1/9 & 2/9 & -1/9 \end{array}\right) \to \left(\begin{array}{rrr|rrr} 1 & 0 & 0 & -16/9 & 8/9 & -1/9 \\ 0 & 1 & 0 & 14/9 & -7/9 & 2/9 \\ 0 & 0 & 1 & -1/9 & 2/9 & -1/9 \end{array}\right).$$

We now apply A^{-1} to the right-hand side vector b to obtain:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \frac{1}{9}\begin{pmatrix} -16 & 8 & -1 \\ 14 & -7 & 2 \\ -1 & 2 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} = \frac{1}{9}\begin{pmatrix} -18 \\ 18 \\ -3 \end{pmatrix} = \begin{pmatrix} -2 \\ 2 \\ -1/3 \end{pmatrix}.$$

The procedure we have followed to compute A^{-1} is very much like what one would do in Gaussian elimination to solve **n** linear systems with coefficient matrix A and with right-hand side vectors $e_1, \ldots, e_n$, where e_i has a 1 in position i and zeros elsewhere. Note that we compute the multipliers necessary to reduce A to upper triangular form only once and then we apply these multipliers to each column of the appended identity matrix, that is, to each of the right-hand side vectors $e_1, \ldots, e_n$. We then solve n upper triangular systems with the modified right-hand side vectors. The total work involved is about:

$$\frac{2}{3}n^3 + n \times 2n^2 = \frac{8}{3}n^3,$$

which, for large n, is about 4 times as much work as Gaussian elimination. Additionally, we must compute the product of A^{-1} with b, but this requires only about $2n^2$ operations, which is of lower order than the operation count for computing A^{-1}.

2. **Cramer's Rule.** This is a useful way to solve small linear systems by hand, but, as we will see, it is *completely inappropriate* for large linear systems. To compute the jth component of the solution, replace column j of A by the right-hand side vector b, and compute the ratio of the determinant of this matrix to the determinant of A. For our example problem, expanding determinants by the first row, we find

$$\begin{aligned} \det(A) = \det\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} &= 1 \cdot \det\begin{pmatrix} 5 & 6 \\ 8 & 0 \end{pmatrix} - 2 \cdot \det\begin{pmatrix} 4 & 6 \\ 7 & 0 \end{pmatrix} + 3 \cdot \det\begin{pmatrix} 4 & 5 \\ 7 & 8 \end{pmatrix} \\ &= -48 - 2 \cdot (-42) + 3 \cdot (-3) = 27, \end{aligned}$$

$$\begin{aligned} \det\begin{pmatrix} \mathbf{1} & 2 & 3 \\ \mathbf{0} & 5 & 6 \\ \mathbf{2} & 8 & 0 \end{pmatrix} &= 1 \cdot \det\begin{pmatrix} 5 & 6 \\ 8 & 0 \end{pmatrix} - 2 \cdot \det\begin{pmatrix} 0 & 6 \\ 2 & 0 \end{pmatrix} + 3 \cdot \det\begin{pmatrix} 0 & 5 \\ 2 & 8 \end{pmatrix} \\ &= 1 \cdot (-48) - 2 \cdot (-12) + 3 \cdot (-10) = -54, \end{aligned}$$

$$\det\begin{pmatrix} 1 & \mathbf{1} & 3 \\ 4 & \mathbf{0} & 6 \\ 7 & \mathbf{2} & 0 \end{pmatrix} = 54, \quad \det\begin{pmatrix} 1 & 2 & \mathbf{1} \\ 4 & 5 & \mathbf{0} \\ 7 & 8 & \mathbf{2} \end{pmatrix} = -9,$$

so that

$$x_1 = \frac{-54}{27} = -2, \quad x_2 = \frac{54}{27} = 2, \quad x_3 = \frac{-9}{27} = -\frac{1}{3}.$$

To apply Cramer's rule to an n by n linear system, we need to compute n determinants. Let us estimate the work for evaluating just one determinant using the expansion procedure outlined here. To evaluate a 2 by 2 determinant

$$\det\begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

requires 3 operations: two multiplications and one subtraction. To evaluate a 3 by 3 determinant

$$\det\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = a \cdot \det\begin{pmatrix} e & f \\ h & i \end{pmatrix} - b \cdot \det\begin{pmatrix} d & f \\ g & i \end{pmatrix} + c \cdot \det\begin{pmatrix} d & e \\ g & h \end{pmatrix}$$

requires evaluating three 2 by 2 determinants, multiplying the results by appropriate scalars and adding, for a total of about $3 \times$ (work for 2 by 2 determinant) $+ 5 = 14$ operations. To evaluate a 4 by 4 determinant one must evaluate four 3 by 3 determinants, multiply by appropriate scalars and add the results, thus doing more than a factor of 4 times as much work as for a 3 by 3 determinant. In general, if W_n is the amount of work required to evaluate an n by n determinant, then

$$W_n > n \cdot W_{n-1} > n \cdot (n-1) \cdot W_{n-2} > \ldots > n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 = n!$$

For n as large as 20 or so, a computation of the determinant in this way is out of the question, since $20! \approx 2.4 \times 10^{18}$. On a fast supercomputer that performs, say, 10^9 floating point operations per second, this would require about 76 years!

There are faster ways than the one given here to compute determinants. In fact, one can find the determinant from the LU factorization of a matrix, since $\det(LU) = \det(L) \cdot \det(U)$, and the determinant of a triangular matrix is just the product of its diagonal entries. To do this with Cramer's rule, however, would defeat the purpose, since we were considering it as an alternative to computing an LU factorization of the matrix. For this reason, as well as accuracy concerns, Cramer's rule is **not** used for solving large linear systems on the computer.

5.4 Conditioning of Linear Systems

In Chapter 4, we discussed the absolute and relative condition numbers for the problem of evaluating a scalar-valued function of a scalar argument. The problem of solving $Ax = b$ involves, as input, a matrix A and a right-hand side vector b, while the output is a vector x. To measure the change in output due to a small change in input, we must discuss vector and matrix *norms*.

5.4.1 Norms

Definition. A **norm** for vectors is a function $\|\cdot\|$ satisfying, for all n-vectors v, w:

(i) $\|v\| \geq 0$, with equality if and only if $v = 0$;

(ii) $\|\alpha v\| = |\alpha| \; \|v\|$, for any scalar α;

(iii) $\|v + w\| \leq \|v\| + \|w\|$ (triangle inequality).

The most commonly used vector norm on $\mathbf{R}^n$ is the *2-norm* or *Euclidean norm*:

$$\|v\|_2 \equiv \sqrt{\sum_{i=1}^{n} |v_i|^2}.$$

Other frequently used norms are the ∞-*norm*:

$$\|v\|_\infty \equiv \max_{i=1,\ldots,n} |v_i|,$$

and the *1-norm*:

$$\|v\|_1 \equiv \sum_{i=1}^{n} |v_i|.$$

More generally, it can be shown that for any $p \geq 1$, the *p-norm*, defined by

$$\|v\|_p \equiv \left(\sum_{i=1}^{n} |v_i|^p \right)^{1/p},$$

is a norm. We will usually work with the 1-norm, the 2-norm, or the ∞-norm. Of these, only the 2-norm comes from an inner product; that is,

$$\|v\|_2 = \langle v, v \rangle^{1/2}, \quad \text{where} \;\; \langle x, y \rangle = \sum_{i=1}^{n} x_i y_i.$$

Example. If $v = (1, 2, -3)^T$, then $\|v\|_2 = \sqrt{1^2 + 2^2 + (-3)^2} = \sqrt{14}$, $\|v\|_\infty = \max\{|1|, \; |2|, \; |-3|\} = 3$, and $\|v\|_1 = |1| + |2| + |-3| = 6$. If $w = (4, 5, 6)^T$, then $\langle v, w \rangle = 1 \cdot 4 + 2 \cdot 5 - 3 \cdot 6 = -4$. [You can compute these quantities with MATLAB by entering `v` and `w` and typing `norm(v,1)`, `norm(v,'inf')`, `norm(v,2)` (or simply `norm(v)`), and `w'*v`, respectively.]

Definition. A **matrix norm** is a function $\|\cdot\|$ satisfying, for all m by n matrices A, B:

(i) $\|A\| \geq 0$, with equality if and only if $A = 0$;

(ii) $\|\alpha A\| = |\alpha| \; \|A\|$, for any scalar α;

(iii) $\|A+B\| \leq \|A\| + \|B\|$ (triangle inequality).

Some definitions require, in addition, that a matrix norm be *submultiplicative*; that is, if A is an m by n matrix and C is an n by p matrix, then the product AC must satisfy:

(iv) $\|AC\| \leq \|A\| \cdot \|C\|$.

We will always assume that a matrix norm is submultiplicative.

If $\|\cdot\|$ is a vector norm, the *induced* matrix norm is

$$\max_{\|v\|=1} \|Av\| = \max_{v \neq 0} \frac{\|Av\|}{\|v\|}.$$

It can be shown that the induced norm satisfies all four properties listed above.

The following theorems give expressions for the matrix norms induced by the 1-norm, the 2-norm, and the ∞-norm for vectors.

Theorem 5.4.1. *Let A be an m by n matrix and let $\|A\|_1$ denote the matrix norm induced by the 1-norm for n-vectors. Then $\|A\|_1$ is the maximum absolute column sum:*

$$\|A\|_1 = \max_{j=1,\ldots,n} \sum_{i=1}^{m} |a_{ij}|.$$

Proof. Let $A = (\vec{a}_{:,1}, \ldots \vec{a}_{:,n})$, where $\vec{a}_{:,j} = (a_{1j}, \ldots, a_{mj})^T$ denotes the jth column of A. If v is an n-vector, then $Av = \sum_{j=1}^{n} \vec{a}_{:,j} v_j$ and so

$$\begin{aligned} \|Av\|_1 &= \|\sum_{j=1}^{n} \vec{a}_{:,j} v_j\|_1 \leq \sum_{j=1}^{n} |v_j| \cdot \|\vec{a}_{:,j}\|_1 \quad \text{(by (iii) and (ii) for vector norms)} \\ &\leq \max_j \|\vec{a}_{:,j}\|_1 \cdot \left(\sum_{j=1}^{n} |v_j| \right) = \max_j \|\vec{a}_{:,j}\|_1 \cdot \|v\|_1. \end{aligned}$$

This shows that $\|A\|_1 \leq \max_j \|\vec{a}_{:,j}\|_1$. On the other hand, if the index of a column with maximum 1-norm is J and if $v_J = 1$ and all other entries of v are 0, then $Av = \vec{a}_{:,J}$, so that $\|v\|_1 = 1$ and $\|Av\|_1 = \max_j \|\vec{a}_{:,j}\|_1$. This implies that $\|A\|_1 \geq \max_j \|\vec{a}_{:,j}\|_1$. Combining the two results we obtain equality. □

Theorem 5.4.2. *Let A be an m by n matrix and let $\|A\|_\infty$ denote the matrix norm induced by the ∞-norm for n-vectors. Then $\|A\|_\infty$ is the maximum absolute row sum:*

$$\|A\|_\infty = \max_{i=1,\ldots,m} \sum_{j=1}^{n} |a_{ij}|.$$

Proof. For any n-vector v, we have

$$\begin{aligned}
\|Av\|_\infty &= \max_{i=1,\ldots,m} |(Av)_i| = \max_{i=1,\ldots,m} |\sum_{j=1}^n a_{ij}v_j| \\
&\leq \max_{i=1,\ldots,m} \sum_{j=1}^n |a_{ij}| \cdot |v_j| \\
&\leq \max_{i=1,\ldots,m} (\max_{j=1,\ldots,n} |v_j|) \sum_{j=1}^n |a_{ij}| = \|v\|_\infty \max_{i=1,\ldots,m} \sum_{j=1}^n |a_{ij}|.
\end{aligned}$$

This shows that $\|A\|_\infty \leq \max_{i=1,\ldots,m} \sum_{j=1}^n |a_{ij}|$. On the other hand, if I is the index of a row with maximal absolute row sum and if $|v_j| = 1$ and $a_{Ij}v_j = |a_{Ij}v_j| = |a_{Ij}|$ for all j (i.e., $v_j = 1$ if $a_{Ij} \geq 0$ and $v_j = -1$ if $a_{Ij} < 0$), then we will have equality above, since for each i, $|\sum_{j=1}^n a_{ij}v_j| \leq \sum_{j=1}^n |a_{ij}| \leq \sum_{j=1}^n |a_{Ij}|$, and for $i = I$ this is an equality. This implies that $\|A\|_\infty \geq \max_{i=1,\ldots,m} \sum_{j=1}^n |a_{ij}|$, and combining the two results we obtain equality. □

Theorem 5.4.3. *Let A be an m by n matrix and let $\|A\|_2$ denote the matrix norm induced by the 2-norm for n-vectors. Then $\|A\|_2$ is the square root of the largest eigenvalue of A^TA:*

$$\|A\|_2 = \sqrt{\textit{largest eigenvalue of } A^TA}.$$

[Note: Material needed for this proof will not be covered until the chapter on eigenvalues. If you are not already familiar with the variational characterization of eigenvalues of a symmetric matrix, then you can skip this proof until then.]

Proof. Since $\|Av\|_2^2 = \langle Av, Av\rangle = \langle v, A^TAv\rangle$, the result follows from the variational characterization of eigenvalues of a symmetric matrix: the largest eigenvalue of the symmetric matrix A^TA is $\max_{\|v\|_2=1}\langle v, A^TAv\rangle$. □

Example. Let A be the 3 by 2 matrix

$$A = \begin{pmatrix} 0 & -1 \\ -2 & 3 \\ 1 & 0 \end{pmatrix}.$$

Then $\|A\|_1 = \max\{0+2+1,\ 1+3+0\} = 4$, $\|A\|_\infty = \max\{0+1,\ 2+3,\ 1+0\} = 5$. Forming A^TA, we find

$$A^TA = \begin{pmatrix} 0 & -2 & 1 \\ -1 & 3 & 0 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ -2 & 3 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 5 & -6 \\ -6 & 10 \end{pmatrix},$$

and the eigenvalues of this matrix satisfy

$$\det \begin{pmatrix} 5-\lambda & -6 \\ -6 & 10-\lambda \end{pmatrix} = (5-\lambda)(10-\lambda) - 36 = \lambda^2 - 15\lambda + 14 = (\lambda-1)(\lambda-14) = 0,$$

or, $\lambda_1 = 1$, $\lambda_2 = 14$. Therefore $\|A\|_2 = \sqrt{14} \approx 3.7417$. [You can compute these quantities with MATLAB by entering `A` and typing `norm(A,1)`, `norm(A,'inf')`, and `norm(A,2)` (or just `norm(A)`), respectively.]

5.4.2 Sensitivity of Solutions of Linear Systems

Suppose we start with a nonsingular n by n linear system $Ax = b$ and we change the right-hand side vector b by a small amount. How much does the solution to the linear system change? This question will be important when considering solving linear systems on a computer because, as noted in Chapter 3, the entries of b may not be exactly representable as floating point numbers. But the question can be asked in general, without reference to any particular computing system or algorithm. This will tell us about the *conditioning* of the linear system, as described in Section 4.1.

Suppose, then, that $\hat{b}$ is a vector such that, in a certain norm, $\|b - \hat{b}\|$ is small. Let x denote the solution to the linear system $Ax = b$, and let $\hat{x}$ denote the solution to the linear system $A\hat{x} = \hat{b}$. Subtracting these two equations, we find that $A(x - \hat{x}) = b - \hat{b}$, or, $x - \hat{x} = A^{-1}(b - \hat{b})$. Taking norms on each side gives the inequality

$$\|x - \hat{x}\| \leq \|A^{-1}\| \cdot \|b - \hat{b}\|. \tag{5.7}$$

The factor $\|A^{-1}\|$ can be thought of as an absolute condition number for this problem, corresponding to the absolute condition number described in Section 4.1 for scalar-valued functions of a scalar argument.

As noted in Section 4.1, however, it is usually the relative error rather than the absolute error that is of interest; in this case, we would like to relate $\|x - \hat{x}\|/\|x\|$ to $\|b - \hat{b}\|/\|b\|$. Dividing each side of (5.7) by $\|x\|$, we find

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq \|A^{-1}\| \cdot \frac{\|b - \hat{b}\|}{\|x\|} = \|A^{-1}\| \cdot \frac{\|b - \hat{b}\|}{\|b\|} \cdot \frac{\|b\|}{\|x\|}.$$

Since $\|b\|/\|x\| = \|Ax\|/\|x\| \leq \|A\|$, it follows that

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq \|A^{-1}\| \cdot \|A\| \cdot \frac{\|b - \hat{b}\|}{\|b\|}. \tag{5.8}$$

The number $\|A\| \cdot \|A^{-1}\|$ serves as a sort of relative condition number for the problem of solving $Ax = b$.

Definition. The number $\kappa(A) \equiv \|A\| \cdot \|A^{-1}\|$ is called the *condition number* of the nonsingular matrix A.

Example. Consider the 2 by 2 matrix

$$A = \begin{pmatrix} 1 & -1 \\ 2 & 2 \end{pmatrix},$$

whose inverse is

$$A^{-1} = \frac{1}{4} \begin{pmatrix} 2 & 1 \\ -2 & 1 \end{pmatrix}.$$

The condition number of A in the 1-norm is $\kappa_1(A) \equiv \|A\|_1 \cdot \|A^{-1}\|_1 = 3 \cdot 1 = 3$. The condition number of A in the ∞-norm is $\kappa_\infty(A) \equiv \|A\|_\infty \cdot \|A^{-1}\|_\infty = 4 \cdot (3/4) = 3$. To find the condition number in the 2-norm, we can compute $A^T A$ and $A^{-1^T} A^{-1}$ and find their eigenvalues:

$$A^T A = \begin{pmatrix} 5 & 3 \\ 3 & 5 \end{pmatrix}, \quad A^{-T} A^{-1} = \frac{1}{16} \begin{pmatrix} 8 & 0 \\ 0 & 2 \end{pmatrix}.$$

The eigenvalues of $A^T A$ satisfy $(5-\lambda)^2 - 9 = 0$, so $\lambda_1 = 2$ and $\lambda_2 = 8$. Thus $\|A\|_2 = \sqrt{8} = 2\sqrt{2}$. The eigenvalues of $A^{-T} A^{-1}$ are $1/2$ and $1/8$, so $\|A^{-1}\| = 1/\sqrt{2}$. It follows that $\kappa_2(A) \equiv \|A\|_2 \cdot \|A^{-1}\|_2 = 2$.

Suppose now that we make a small change not only to b but also to A. How can we relate the change in the solution to the changes in both A and b? The following theorem shows that, as long as the change in A is small enough so that the modified matrix is still nonsingular and has an inverse whose norm is close to that of A^{-1}, we again obtain an estimate based on the condition number of A.

Theorem 5.4.4. *Let A be a nonsingular n by n matrix, let b be a given n-vector, and let x satisfy $Ax = b$. Let $A + E$ be another nonsingular n by n matrix and $\hat{b}$ another n-vector, and let $\hat{x}$ satisfy $(A + E)\hat{x} = \hat{b}$. Then*

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq (\|(A + E)^{-1}\| \cdot \|A\|) \left(\frac{\|b - \hat{b}\|}{\|b\|} + \frac{\|E\|}{\|A\|} \right). \tag{5.9}$$

If $\|E\|$ is small enough so that $\|A^{-1}\| \cdot \|E\| < 1$, then

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq \frac{\kappa(A)}{1 - \kappa(A)\|E\|/\|A\|} \cdot \left(\frac{\|b - \hat{b}\|}{\|b\|} + \frac{\|E\|}{\|A\|} \right). \tag{5.10}$$

Proof. Subtracting the two equations $(A + E)\hat{x} = \hat{b}$ and $(A + E)x = b + Ex$, we find that $(A + E)(x - \hat{x}) = b - \hat{b} + Ex$, or, $x - \hat{x} = (A + E)^{-1}(b - \hat{b} + Ex)$. Taking norms on each side we obtain the inequality

$$\|x - \hat{x}\| \leq \|(A + E)^{-1}\| \cdot \left(\|b - \hat{b}\| + \|E\| \ \|x\| \right).$$

Dividing each side by $\|x\|$ gives

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq \|(A + E)^{-1}\| \cdot \left(\frac{\|b - \hat{b}\|}{\|b\|} \cdot \frac{\|b\|}{\|x\|} + \|E\| \right).$$

Since $\|b\|/\|x\| = \|Ax\|/\|x\| \leq \|A\|$, this inequality can be replaced by

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq (\|(A + E)^{-1}\| \cdot \|A\|) \cdot \left(\frac{\|b - \hat{b}\|}{\|b\|} + \frac{\|E\|}{\|A\|} \right).$$

This establishes (5.9).

To establish (5.10) we will use a Neumann series expansion of $(A+E)^{-1}$, which will be discussed in Chapter ?. Writing $(A+E)^{-1}$ in the form $[A(I+A^{-1}E)]^{-1} = (I+A^{-1}E)^{-1}A^{-1}$, it can be shown that when $\|A^{-1}E\| < 1$, then

$$(I+A^{-1}E)^{-1} = I - A^{-1}E + (A^{-1}E)^2 - \dots,$$

just as the formula $1/(1+\alpha) = 1 - \alpha + \alpha^2 - \dots$ holds for scalars α with $|\alpha| < 1$. Thus

$$(A+E)^{-1} = A^{-1} + \left(\sum_{k=1}^{\infty} (-1)^k (A^{-1}E)^k \right) A^{-1},$$

and

$$\|(A+E)^{-1}\| \le \|A^{-1}\| \left(1 + \sum_{k=1}^{\infty} \|A^{-1}E\|^k \right).$$

Summing the geometric series this becomes

$$\|(A+E)^{-1}\| \le \|A^{-1}\| \left(1 + \frac{\|A^{-1}E\|}{1 - \|A^{-1}E\|} \right) \le \|A^{-1}\| \left(1 + \frac{\|A^{-1}\| \, \|E\|}{1 - \|A^{-1}\| \, \|E\|} \right).$$

The right-hand side can be written as

$$\|A^{-1}\| \left(1 + \frac{\kappa(A)\|E\|/\|A\|}{1 - \kappa(A)\|E\|/\|A\|} \right) = \|A^{-1}\| \frac{1}{1 - \kappa(A)\|E\|/\|A\|},$$

and substituting this for $\|(A+E)^{-1}\|$ in (5.9) gives the result (5.10). □

Example. Let $\delta > 0$ be a small number and let

$$A = \begin{pmatrix} 1 & 1+\delta \\ 1-\delta & 1 \end{pmatrix}.$$

Then

$$A^{-1} = \frac{1}{\delta^2} \begin{pmatrix} 1 & -1-\delta \\ -1+\delta & 1 \end{pmatrix}.$$

The condition number of A in the ∞-norm is

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \frac{(2+\delta)^2}{\delta^2}.$$

If $\delta = .01$, then $\kappa_\infty(A) = (201)^2 = 40401$.

This means that if we start with a linear system $Ax = b$ and perturb b and/or A by a tiny amount, the solution may change by as much as 40401 times as much. For instance, consider the linear system

$$\begin{pmatrix} 1 & 1.01 \\ .99 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2.01 \\ 1.99 \end{pmatrix},$$

whose solution is $x = y = 1$. If we replace it by the linear system

$$\begin{pmatrix} 1 & 1.01 \\ .99 & 1 \end{pmatrix} \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \end{pmatrix},$$

so that $\|b - \hat{b}\|_\infty / \|b\|_\infty = .01/2.01$, then the solution to the new system (determined by Gaussian elimination or Cramer's rule or any other algorithm) is $x = -200$, $y = 200$, a relative change in ∞-norm of 201.

5.5 Stability of Gaussian Elimination with Partial Pivoting

Suppose we use some algorithm to solve an ill-conditioned linear system on a computer. We cannot necessarily expect to find anything close to the true solution since the matrix and right-hand side vector may have been rounded as they were stored in the computer. Then, even if all other arithmetic operations were performed exactly, we would be solving a slightly different linear system from the one intended and so the solution might be quite different. At best, we could hope to find the exact solution to a nearby problem. An algorithm that does this is called *backward stable*, as mentioned in Section 4.2. Thus we can ask the question, "Is Gaussian elimination with partial pivoting backward stable?" What about the other algorithms we discussed; i.e., computing A^{-1} and then forming $A^{-1}b$ or solving a linear system by Cramer's rule?

Consider the following example, where we factor an ill-conditioned matrix in the form LU using Gaussian elimination and retaining only two decimal places:

$$\begin{pmatrix} 1 & 1.01 \\ .99 & 1.01 \end{pmatrix} \xrightarrow{L_1} \begin{pmatrix} 1 & 1.01 \\ 0 & .01 \end{pmatrix}.$$

Here $1.01-(.99\times 1.01)$ was computed to be .01, since $.99\times 1.01 = .9999$ was rounded to 1. Looking at the product of the L and U factors we find

$$\begin{pmatrix} 1 & 0 \\ .99 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1.01 \\ 0 & .01 \end{pmatrix} = \begin{pmatrix} 1 & 1.01 \\ .99 & 1.0099 \end{pmatrix}.$$

Thus we have computed the exact LU decomposition of a nearby matrix (one whose entries differ from those of the original by only about the machine precision).

It turns out that **for practically all problems, Gaussian elimination with partial pivoting is backward stable**: it finds the true solution to a nearby problem. There are exceptions, but they are *extremely* rare. An open problem in numerical analysis is to further classify and quantify the "unlikelihood" of these examples where Gaussian elimination with partial pivoting is unstable. The classic example to illustrate the possible instability has a matrix A with 1's on its main diagonal, -1's throughout its strict lower triangle, 1's in its last column, and 0's everywhere else:

$$A = \begin{pmatrix} 1 & & & 1 \\ -1 & 1 & & 1 \\ \vdots & \ddots & \ddots & \vdots \\ -1 & \dots & -1 & 1 \end{pmatrix}$$

Take a matrix of this form of size, say, $n = 70$, and try to solve a linear system with this coefficient matrix in MATLAB. You might not see the instability because the entries are integers; in that case, try perturbing the last column with a small random perturbation, say, `A(:,n) = A(:,n) + 1.e-6*randn(n,1)`. You should find that the matrix is well-conditioned (type `cond(A)` to see its condition number) but that the computed solution to a linear system with this coefficient matrix is far from correct. (You can set up a problem for which you know the true solution by typing `x=randn(n,1); b=A*x`. Then look at `norm(A\b - x)`.) Despite the existence of such rare examples, Gaussian elimination with partial pivoting is the most-used algorithm for solving linear systems. It is what MATLAB uses and it is the accepted procedure in numerical analysis. Another option is *full pivoting*, where one searches for the largest entry in absolute value in the entire matrix or the remaining submatrix and interchanges both rows and columns in order to move this entry into the pivot position.

How can one tell if an algorithm is backward stable? Short of a detailed analysis, one can do some numerical experiments. If an algorithm for solving $Ax = b$ is backward stable, then it should produce an approximate solution $\hat{x}$ for which the *residual* $b - A\hat{x}$ is small, even if the problem is ill-conditioned so that the *error* $x - \hat{x}$ is not small. To see why this is so, let x satisfy $Ax = b$ and suppose that the computed solution $\hat{x}$ satisfies $(A+E)\hat{x} = \hat{b}$. Then $b-A\hat{x} = \hat{b}-A\hat{x}+b-\hat{b} = E\hat{x}+b-\hat{b}$. Taking norms on each side gives the inequality

$$\|b - A\hat{x}\| \leq \|E\| \cdot \|\hat{x}\| + \|b - \hat{b}\|.$$

Dividing each side by $\|A\| \cdot \|\hat{x}\|$ gives

$$\frac{\|b - A\hat{x}\|}{\|A\| \cdot \|\hat{x}\|} \leq \frac{\|E\|}{\|A\|} + \frac{\|b - \hat{b}\|}{\|b\|} \cdot \frac{\|b\|}{\|A\| \cdot \|\hat{x}\|}.$$

For $\|E\|$ and $\|b - \hat{b}\|$ small, the factor $\|b\|/(\|A\| \cdot \|\hat{x}\|)$ will be less than or equal to about 1, and so for a backward stable algorithm, we should expect the quantity on the left-hand side to be on the order of the machine precision ϵ, independent of $\kappa(A)$. Frequently it will be a moderate size constant or, perhaps, a power of the problem size n times ϵ, but these cases are also judged to be backward stable. In the exercises, you will be asked to do some numerical experiments to verify the backward stability of Gaussian elimination with partial pivoting and to test whether solving a linear system by computing A^{-1} or by Cramer's rule is backward stable.

5.6 Least Squares Problems

Consider a system of linear equations with more equations than unknowns. Let A be an $m \times n$ matrix with $m > n$ and b a given m-vector. We seek an n-vector x such that $Ax \approx b$. For example:

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \approx \begin{pmatrix} 2 \\ 0 \\ 4 \end{pmatrix}. \tag{5.11}$$

In most cases we will not be able to obtain exact equality. Instead, we might choose x to minimize the squared residual norm,

$$\|b - Ax\|_2^2 = \sum_{i=1}^{m} (b_i - \sum_{j=1}^{n} a_{ij} x_j)^2. \tag{5.12}$$

This is called a *least squares problem*.

5.6.1 The Normal Equations

One way to determine the value of x that achieves the minimum in (5.12) is to differentiate $\|b - Ax\|^2$ with respect to each component x_k and set these derivatives to 0. Because this is a quadratic functional the point where these partial derivatives are 0 will indeed be the minimum. Differentiating, we find

$$\frac{\partial}{\partial x_k} \left(\|b - Ax\|^2 \right) = \sum_{i=1}^{m} 2(b_i - \sum_{j=1}^{n} a_{ij} x_j)(-a_{ik}),$$

and setting these derivatives to 0 for $k = 1, \ldots, n$ gives

$$\sum_{i=1}^{m} a_{ik} (\sum_{j=1}^{n} a_{ij} x_j) \equiv \sum_{i=1}^{m} a_{ik} (Ax)_i = \sum_{i=1}^{m} a_{ik} b_i.$$

Equivalently, we can write $\sum_{i=1}^{m} (A^T)_{ki} (Ax)_i = \sum_{i=1}^{m} (A^T)_{ki} b_i$, $k = 1, \ldots, n$, or,

$$A^T A x = A^T b. \tag{5.13}$$

The system (5.13) is called the **normal equations**.

A drawback of this approach is that the 2-norm condition number of $A^T A$ is the square of that of A, since by Theorem 5.3.3,

$$\|A^T A\|_2 = \sqrt{\text{largest eigenvalue of } (A^T A)^2} = \text{largest eigenvalue of } (A^T A) = \|A\|_2^2,$$

and similarly $\|(A^T A)^{-1}\|_2 = \|A^{-1}\|_2^2$. As seen in Section 5.4, a large condition number can lead to a loss of accuracy in the computed solution.

To solve example (5.11) using the normal equations, we write

$$\begin{pmatrix} 1 & 1 & 2 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 2 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 0 \\ 4 \end{pmatrix},$$

which becomes

$$\begin{pmatrix} 6 & 2 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 10 \\ 6 \end{pmatrix}.$$

Solving this square linear system by Gaussian elimination, we find

$$\left(\begin{array}{cc|c} 6 & 2 & 10 \\ 2 & 3 & 6 \end{array} \right) \longrightarrow \left(\begin{array}{cc|c} 6 & 2 & 10 \\ 0 & 7/3 & 8/3 \end{array} \right),$$

resulting in

$$x_2 = \frac{8}{7}, \quad x_1 = \frac{9}{7}.$$

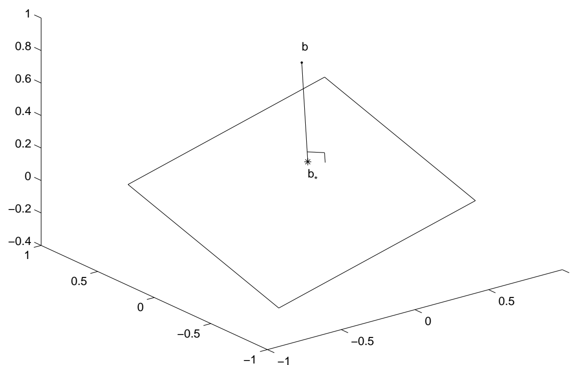
It is always a good idea to "check" your answer by computing the residual to make sure that it looks reasonable:

$$\begin{pmatrix} 2 \\ 0 \\ 4 \end{pmatrix} - \begin{pmatrix} 1 & 1 & 2 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 8/7 \\ 9/7 \end{pmatrix} = \begin{pmatrix} -3/7 \\ 1/7 \\ 3/7 \end{pmatrix}.$$

5.6.2 QR Decomposition

The least squares problem can be approached in a different way. We wish to find a vector x such that $Ax = b_*$, where b_* is the closest vector to b (in the 2-norm) in the range of A. This is equivalent to minimizing $\|b - Ax\|_2$ since, by definition of b_*, $\|b - b_*\|_2 \leq \|b - Ay\|_2$ for all y.

You may recall from linear algebra that the closest vector to a given vector, from a subspace, is the *orthogonal projection* of that vector onto the subspace. This is pictured below.



If $q_1, \ldots, q_k$ form an *orthonormal basis* for the subspace then the orthogonal projection of b onto the subspace is $\sum_{j=1}^{k} \langle b, q_j \rangle q_j$. Assume that the columns of A are linearly independent, so that the range of A (which is the span of its columns) has dimension n. Then the closest vector to b in range(A) is

$$b_* = \sum_{j=1}^{n} \langle b, q_j \rangle q_j,$$

where $q_1, \ldots, q_n$ form an orthonormal basis for range(A). Let Q be the m by n matrix whose columns are the orthonormal vectors $q_1, \ldots, q_n$. Then $Q^T Q = I_{n \times n}$ and the formula for b_* can be written compactly as

$$b_* = Q(Q^T b),$$

since $Q(Q^T b) = \sum_{j=1}^{n} q_j (Q^T b)_j = \sum_{j=1}^{n} q_j (q_j^T b) = \sum_{j=1}^{n} q_j \langle b, q_j \rangle$.

You may also recall from linear algebra that, given a set of linearly independent vectors such as the columns of A, one can construct an orthonormal set that spans the same space using the *Gram-Schmidt* algorithm:

Given a linearly independent set $v_1, v_2, \ldots$, set $q_1 = v_1/\|v_1\|$, and for $j = 2, 3, \ldots$,

Set $\tilde{q}_j = v_j - \sum_{i=1}^{j-1} \langle v_j, q_i \rangle q_i$.

Set $q_j = \tilde{q}_j / \|\tilde{q}_j\|$.

Note that if $v_1, v_2, \ldots, v_n$ are the columns of A, then the Gram-Schmidt algorithm can be thought of as factoring the m by n matrix A in the form $A = QR$, where $Q = (q_1, \ldots, q_n)$ is an m by n matrix with orthonormal columns and R is an n by n upper triangular matrix. To see this, write the equations of the Gram-Schmidt algorithm in the form

$$v_1 = r_{11} q_1, \quad r_{11} = \|v_1\|,$$

$$v_j = r_{jj} q_j + \sum_{i=1}^{j-1} r_{ij} q_i, \quad r_{jj} = \|\tilde{q}_j\|, \quad r_{ij} = \langle v_j, q_i \rangle, \quad j = 2, \ldots, n.$$

These equations can be written using matrices as

$$(v_1, \ldots, v_n) = (q_1, \ldots, q_n) \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & \cdots & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{pmatrix}.$$

Thus if $v_1, \ldots, v_n$ are the columns of A, then we have written A in the form $A = QR$. This is called the *reduced QR factorization* of A. [In the *full QR factorization*, one completes the orthonormal set $q_1, \ldots, q_n$ to an orthonormal basis for $\mathbf{R}^m$: $q_1, \ldots, q_n, q_{n+1}, \ldots, q_m$ so that A is factored in the form $\underbrace{A}_{m \times n} = \underbrace{Q}_{m \times m} \underbrace{\begin{pmatrix} R \\ 0 \end{pmatrix}}_{m \times n}$.]

Having factored A in the form QR, the least squares problem becomes $QRx = b_* = QQ^T b$. We know that this set of equations has a solution since $b_* = QQ^T b$ lies in the range of A. Hence if we multiply each side by Q^T, the resulting equation will have the same solution. Since $Q^T Q = I$, we are left with the upper triangular system $Rx = Q^T b$. Thus the procedure for solving the least squares problem consists of two steps:

1. Compute the reduced QR decomposition of A.

2. Solve the n by n upper triangular system $Rx = Q^T b$.

This is the way MATLAB solves problems with more equations than unknowns, when you type `A\b`. It does not use the Gram-Schmidt algorithm to compute the QR factorization, however. Different methods for computing the QR factorization will be discussed in Chapter ?. You can compute the reduced QR factorization with MATLAB by typing `[Q,R] = qr(A,0)`, while `[Q,R] = qr(A)` gives the full QR factorization.

To see how problem (5.11) can be solved using the QR decomposition, we first use the Gram-Schmidt algorithm to construct a pair of orthonormal vectors that span the same space as the columns of the coefficient matrix:

$$q_1 = \frac{1}{\sqrt{6}} \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}, \quad r_{11} = \sqrt{6},$$

$$\tilde{q}_2 = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} - \frac{(1 \cdot 1 + (-1) \cdot 1 + 1 \cdot 2)}{\sqrt{6}} \cdot \frac{1}{\sqrt{6}} \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix} = \frac{1}{3} \begin{pmatrix} 2 \\ -4 \\ 1 \end{pmatrix}, \quad r_{12} = \frac{2}{\sqrt{6}},$$

$$q_2 = \frac{1}{\sqrt{21}} \begin{pmatrix} 2 \\ -4 \\ 1 \end{pmatrix}, \quad r_{22} = \sqrt{\frac{7}{3}}.$$

We then compute the product of Q^T and the right-hand side vector:

$$\begin{pmatrix} 1/\sqrt{6} & 1/\sqrt{6} & 2/\sqrt{6} \\ 2/\sqrt{21} & -4/\sqrt{21} & 1/\sqrt{21} \end{pmatrix} \begin{pmatrix} 2 \\ 0 \\ 4 \end{pmatrix} = \begin{pmatrix} 10/\sqrt{6} \\ 8/\sqrt{21} \end{pmatrix}.$$

Finally we solve the triangular linear system,

$$\begin{pmatrix} \sqrt{6} & 2/\sqrt{6} \\ 0 & \sqrt{7/3} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 10/\sqrt{6} \\ 8/\sqrt{21} \end{pmatrix},$$

to obtain

$$x_2 = \frac{8}{7}, \quad \sqrt{6} x_1 + \frac{2}{\sqrt{6}} \cdot \frac{8}{7} = \frac{10}{\sqrt{6}} \rightarrow x_1 = \frac{9}{7}.$$

Although the QR decomposition is the preferred method for solving least squares problems on the computer, the arithmetic can become messy when using this algorithm by hand. The normal equations approach is usually easier when solving a small problem by hand.

5.6.3 Fitting Polynomials to Data

Given a set of measured data points (x_i, y_i), $i = 1, \ldots, m$, one sometimes expects a linear relationship of the form $y = c_0 + c_1 x$. Because of errors in the measurements, the points will not fall exactly on a straight line, but they should be close, and one would like to find the straight line that most closely matches the data. That is, one might look for a least squares solution to the overdetermined linear system

$$\begin{aligned} c_0 + c_1 x_1 &\approx y_1 \\ c_0 + c_1 x_2 &\approx y_2 \\ &\vdots \\ c_0 + c_1 x_m &\approx y_m, \end{aligned}$$

by choosing c_0 and c_1 to minimize

$$\sum_{i=1}^{m}(y_i - (c_0 + c_1 x_i))^2.$$

Writing the above equations with matrices, we have

$$\begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_m \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \approx \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix},$$

which can be solved by solving the normal equations or by QR decomposition, as explained in the previous two subsections.

Suppose we wish to fit a polynomial of degree $n-1 < m$ to the measured data points; that is we wish to find coefficients $c_0, c_1, \ldots, c_{n-1}$ such that

$$y_i \approx c_0 + c_1 x_i + c_2 x_i^2 + \ldots + c_{n-1} x_i^{n-1}, \quad i = 1, \ldots, m,$$

or, more precisely, such that $\sum_{i=1}^{m}(y_i - \sum_{j=0}^{n-1} c_j x_i^j)^2$ is minimal. Again writing the system using matrices, we have

$$\underbrace{\begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_m & x_m^2 & \ldots & x_m^{n-1} \end{pmatrix}}_{m \times n} \underbrace{\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{pmatrix}}_{n \times 1} \approx \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ \vdots \\ y_m \end{pmatrix}}_{m \times 1}.$$

The m by n system can again be solved using either the normal equations or the QR decomposition of the coefficient matrix.

Example. Consider the following table of data:

x	y
1	2
2	3
3	6

To find the straight line $y = c_0 + c_1 x$ that best fits the data, we must solve, in a least squares sense, the overdetermined system

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \approx \begin{pmatrix} 2 \\ 3 \\ 6 \end{pmatrix}.$$

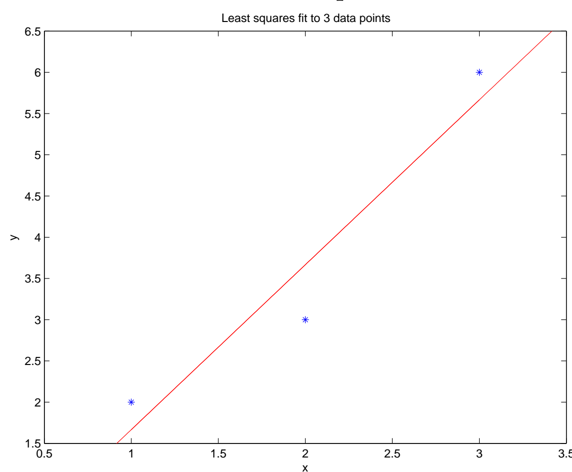
Using the normal equations approach, this becomes

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 6 \end{pmatrix},$$

or,

$$\begin{pmatrix} 3 & 6 \\ 6 & 14 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 11 \\ 26 \end{pmatrix}.$$

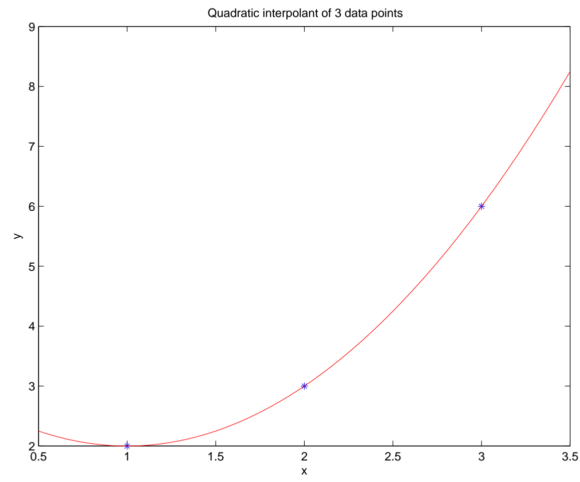
Solving this 2 by 2 system, we find $c_0 = -1/3$ and $c_1 = 2$, so the straight line of best fit has the equation $y = -1/3 + 2x$. The data and this line are plotted below.



Suppose we fit a quadratic polynomial $y = c_0 + c_1 x + c_2 x^2$ to the above data points. We will see in the next section that there is a unique quadratic that *exactly* fits the three data points. This means that when we solve the least squares problem, the residual will be zero. The equations for c_0, c_1, and c_2 are

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 6 \end{pmatrix}.$$

There is no need to form the normal equations here; we can solve this square linear system directly to obtain: $c_0 = 3$, $c_1 = -2$, $c_2 = 1$; i.e., $y = 3 - 2x + x^2$. This quadratic is plotted along with the data points below.


Quadratic interpolant of 3 data points
9
8
7
6
5
4
3
2
y
0.5
1
1.5
2
2.5
3
3.5
x

Exercises

1. Write the following matrix in the form LU, where L is a unit lower triangular matrix and U is an upper triangular matrix:
$$\begin{pmatrix} 4 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & 4 \end{pmatrix}$$
Write the same matrix in the form LL^T, where L is lower triangular.

2. Write a function `usolve`, analogous to function `lsolve` in section 5.2.2, to solve an upper triangular system $Ux = y$.

3. Add partial pivoting to the LU factorization code of section 5.2.2. Use a vector, say, `piv` to keep track of row interchanges. For example, you might initialize `piv` to be `[1:n]`, and then if row `i` is interchanged with row `j`, you would interchange `piv(i)` and `piv(j)`. Now use the vector `piv` along with the L and U factors to solve a linear system $Ax = b$. To do this, you will first need to perform the same interchanges of entries of b that were performed on the rows of A. Then you can use function `lsolve` in section 5.2.2 to solve $Ly = b$, followed by function `usolve` from the previous exercise to solve $Ux = y$. You can check your answer by computing `norm(b-A*x)` (which should be on the order of the machine precision).

4. How many operations (additions, subtractions, multiplications, and divisions) are required to:

   (a) Compute the sum of two n-vectors?

   (b) Compute the product of an m by n matrix with an n-vector?

   (c) Solve an n by n *upper triangular* linear system $Ux = y$?

5. Compute the 2-norm, the 1-norm, and the ∞-norm of
$$v = \begin{pmatrix} 4 \\ 5 \\ -6 \end{pmatrix}.$$

6. Compute the 1-norm and the ∞-norm of
$$A = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}.$$
Also compute the condition number $\kappa(A) \equiv \|A\| \cdot \|A^{-1}\|$ in the 1-norm and the ∞-norm.

7. Show that for all n-vectors v:

   (a) $\|v\|_\infty \leq \|v\|_2 \leq \sqrt{n}\|v\|_\infty$.

   (b) $\|v\|_2 \leq \|v\|_1$.

(c) $\|v\|_1 \leq n\|v\|_\infty$.

For each of the above inequalities, identify a nonzero vector v for which equality holds.

8. Prove that for any nonsingular 2 by 2 matrix, the ∞-norm condition number and the 1-norm condition number are equal. [Hint: Use the fact that the inverse of a 2 by 2 matrix is one over the determinant times the adjoint.]

9. It is shown in the text (inequality (5.8)) that if $Ax = b$ and $A\hat{x} = \hat{b}$, then

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq \kappa(A) \frac{\|b - \hat{b}\|}{\|b\|}.$$

Go through the proof of this inequality and show that for certain nonzero vectors b and $\hat{b}$ (with $\hat{b} \neq b$), equality will hold.

10. Write a routine to generate an n by n matrix with a given 2-norm condition number,

```
function A = matgen(n, condno)
```

by generating two random orthogonal matrices U and V and a diagonal matrix Σ with $\sigma_{ii} = \texttt{condno}^{-(i-1)/(n-1)}$, and setting $A = U\Sigma V^T$. (Note that the largest diagonal entry of Σ is 1 and the smallest is $\texttt{condno}^{-1}$, so the ratio is `condno`.) You can generate a random orthogonal matrix in MATLAB by first generating a random matrix, `Mat = randn(n,n)`, and then computing its QR decomposition, `[Q,R] = qr(Mat)`. The matrix `Q` is then a random orthogonal matrix. You can check the condition number of the matrix you generate by using the function `cond` in MATLAB. Turn in a listing of your code.

For *condno* $= (1,\ 10^4,\ 10^8,\ 10^{12},\ 10^{16})$, use your routine to generate a random matrix A with condition number *condno*. Also generate a random vector `xtrue` of length n, and compute the product `b = A*xtrue`.

Solve $Ax = b$ using Gaussian elimination with partial pivoting. This can be done in MATLAB by typing `x = A\b`. Determine the 2-norm of the relative error in your computed solution, $\|\texttt{x}-\texttt{xtrue}\|/\|\texttt{xtrue}\|$. Explain how this is related to the condition number of A. Compute the 2-norm of the relative residual, $\|b - Ax\|/(\|A\|\|x\|)$. Does the algorithm for solving $Ax = b$ appear to be *backward stable*; that is, is the computed solution the exact solution to a nearby problem?

Solve $Ax = b$ by inverting A: `Ainv = inv(A)`, and then multiplying b by A^{-1}: `x = Ainv*b`. Again look at relative errors and residuals. Does this algorithm appear to be backward stable?

Finally, solve $Ax = b$ using Cramer's rule. Use MATLAB function `det` to compute determinants. (Don't worry — it does not use the $n!$ algorithm discussed in section 5.3.) Again look at relative errors and residuals and determine whether this algorithm is backward stable.

Turn in a table showing the relative errors and residuals for each of the three algorithms and each of the condition numbers tested, along with a brief explanation of the results.

11. Find the polynomial of degree 10 that best fits the function $b(t) = \cos(4t)$ at 50 equally spaced points t between 0 and 1. Set up the matrix A and right-hand side vector b, and determine the polynomial coefficients in two different ways:

    (a) By using the MATLAB command `x = A\b` (which uses a QR factorization).

    (b) By solving the normal equations $A^T Ax = A^T b$. This can be done in MATLAB by typing `x = (A'*A)\(A'*b)`.

    Print the results to 16 digits (using `format long e`) and comment on the differences you see. (Note: You can compute the condition number of A or of $A^T A$ using MATLAB function `cond`.)

Chapter 6

Polynomial and Piecewise Polynomial Interpolation

6.1 The Vandermonde System

Given a set of $n+1$ data points, $(x_1, y_1), (x_2, y_2), \ldots, (x_{n+1}, y_{n+1})$, one can find a polynomial of degree at most n that exactly fits these points. One way to do this was demonstrated in the previous section. Since we want

$$y_i = \sum_{j=0}^{n} c_j x_i^j, \quad i = 1, \ldots, n+1, \tag{6.1}$$

the coefficients $c_0, \ldots, c_n$ must satisfy

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^n \\ 1 & x_2 & x_2^2 & \ldots & x_2^n \\ 1 & x_3 & x_3^2 & \ldots & x_3^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n+1} & x_{n+1}^2 & \ldots & x_{n+1}^n \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n+1} \end{pmatrix}. \tag{6.2}$$

The matrix here is called a **Vandermonde matrix**. It can be shown to be nonsingular, provided $x_1, \ldots, x_{n+1}$ are distinct, but, unfortunately, it can be very *ill-conditioned*. To solve the linear system using Gaussian elimination (or QR-factorization) requires $O(n^3)$ operations. Because of the ill-conditioning and the large amount of work required, polynomial interpolation problems (unlike least squares problems, where the degree of the polynomial is less than the number of data points minus one), usually are not solved in this way. In the next two sections we give two alternatives.

6.2 The Lagrange Form of the Interpolation Polynomial

One can simply write down the polynomial p of degree n that satisfies $p(x_i) = y_i$, $i = 1, \ldots, n+1$. [Notation: When we refer to a polynomial of degree n, we will always mean one whose highest

power is *less than or equal to* n; to distinguish this from a polynomial whose highest power is exactly n we will refer to the latter as a polynomial of *exact degree* n.] To do this we will first write down a polynomial $\varphi_i(x)$ that is 1 at x_i and 0 at all of the other nodes:

$$\varphi_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}.$$

Note that if $x = x_i$, then each factor in the product is 1, while if $x = x_j$ for some $j \neq i$, then one of the factors is 0. Note also that $\varphi_i(x)$ is of the desired degree since the product involves n factors: all but one of $j = 1, \ldots, n+1$. A polynomial of degree n that has value y_i at x_i and value 0 at all other nodes is $y_i \varphi_i(x)$. It follows that a polynomial of degree n that interpolates the points $(x_1, y_1), \ldots, (x_{n+1}, y_{n+1})$ is

$$p(x) = \sum_{i=1}^{n+1} y_i \varphi_i(x) = \sum_{i=1}^{n+1} y_i \left(\prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \right). \tag{6.3}$$

This is called the **Lagrange form** of the interpolation polynomial.

Example. Consider the example of section 5.6.3, where we fit a quadratic to the three data points $(1, 2)$, $(2, 3)$, and $(3, 6)$. The Lagrange form of the interpolation polynomial is:

$$p(x) = 2 \cdot \frac{(x-2)(x-3)}{(1-2)(1-3)} + 3 \cdot \frac{(x-1)(x-3)}{(2-1)(2-3)} + 6 \cdot \frac{(x-1)(x-2)}{(3-1)(3-2)}.$$

You can check that this is equal to $x^2 - 2x + 3$, the same polynomial computed from the Vandermonde system in section 5.6.3.

Since we simply wrote down the Lagrange form of the interpolation polynomial, there is no computer time involved in determining it, but to evaluate formula (6.3) at a point requires $O(n^2)$ operations since there are $n+1$ terms in the sum (requiring n additions) and each term is a product of $n+1$ factors (requiring n multiplications). In contrast, if one knows p in the form (6.1), then the value of p at a point x can be determined using *Horner's rule* with just $O(n)$ operations: Start by setting $y = c_n$. Then replace y by $yx + c_{n-1}$ so that $y = c_n x + c_{n-1}$. Replace this value by $yx + c_{n-2}$ so that $y = c_n x^2 + c_{n-1} x + c_{n-2}$. Continuing for n steps, we obtain $y = \sum_{j=0}^{n} c_j x^j$, using about $2n$ operations. In the next section we will represent the interpolation polynomial in a form that requires somewhat more work to determine than (6.3) but which requires less work to evaluate.

It was stated in the previous section that the Vandermonde matrix can be shown to be nonsingular. In fact, the above derivation of the Lagrange interpolation polynomial can be used to show this. Since the polynomial in (6.3) clearly goes through the points $(x_1, y_1), \ldots, (x_{n+1}, y_{n+1})$, we have shown that the linear system (6.2) has a solution for any right-hand side vector $(y_1, \ldots, y_{n+1})^T$. This implies that the Vandermonde matrix is nonsingular and this, in turn, implies that the solution is unique. We have thus proved that as long as the nodes $x_1, \ldots, x_{n+1}$ are distinct, for any

values $y_1, \ldots, y_{n+1}$, there is a unique polynomial p of degree n such that $p(x_i) = y_i$, $i = 1, \ldots, n+1$. Uniqueness also can be established by using the fact that if a polynomial of degree n vanishes at $n+1$ distinct points, then it must be identically zero. Thus, if there were two nth degree polynomials p and q such that $p(x_i) = q(x_i) = y_i$, $i = 1, \ldots, n+1$, then their difference $p - q$ would be an nth degree polynomial satisfying $(p-q)(x_i) = 0$, $i = 1, \ldots, n+1$, and would therefore be identically zero.

6.3 The Newton Form of the Interpolation Polynomial

Consider the set of polynomials

$$1,\ x - x_1,\ (x-x_1)(x-x_2),\ \ldots, \prod_{j=1}^{n}(x - x_j).$$

These polynomials form a *basis* for the set of all polynomials of degree n; that is, they are linearly independent and any polynomial of degree n can be written as a linear combination of these. The **Newton form** of the interpolation polynomial is

$$p(x) = a_0 + a_1(x-x_1) + a_2(x-x_1)(x-x_2) + \ldots + a_n \prod_{j=1}^{n}(x-x_j), \tag{6.4}$$

where $a_0, a_1, \ldots, a_n$ are chosen so that $p(x_i) = y_i$, $i = 1, \ldots, n+1$.

Before seeing how to determine the coefficients, $a_0, \ldots, a_n$, let us estimate the work to evaluate this polynomial at a point x. We can use a procedure somewhat like Horner's rule:

Start by setting $y = a_n$. For $i = n, n-1, \ldots, 1$, replace y by $y(x - x_i) + a_{i-1}$.

Note that after the first time through the *for* statement, $y = a_n(x - x_n) + a_{n-1}$, after the second time $y = a_n(x-x_n)(x-x_{n-1}) + a_{n-1}(x - x_{n-1}) + a_{n-2}$, etc., so that at the end y contains the value of $p(x)$. The work required is about $3n$ operations.

To determine the coefficients in (6.4), we simply apply the conditions $p(x_i) = y_i$ one by one, in order:

$$\begin{aligned}
p(x_1) &= a_0 = y_1 \\
p(x_2) &= a_0 + a_1(x_2 - x_1) = y_2 \Rightarrow a_1 = \frac{y_2 - a_0}{x_2 - x_1} \\
p(x_3) &= a_0 + a_1(x_3 - x_1) + a_2(x_3-x_1)(x_3-x_2) = y_3 \Rightarrow \\
& \quad a_2 = \frac{y_3 - a_0 - a_1(x_3 - x_1)}{(x_3-x_1)(x_3-x_2)} \\
&\vdots
\end{aligned}$$

What we are actually doing here is solving a lower triangular linear system for the coefficients $a_0, \ldots, a_n$:

$$\begin{pmatrix} 1 & & & & \\ 1 & x_2 - x_1 & & & \\ 1 & x_3 - x_1 & (x_3 - x_1)(x_3 - x_2) & & \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & x_{n+1} - x_1 & (x_{n+1} - x_1)(x_{n+1} - x_2) & \ldots & \prod_{j=1}^{n}(x_{n+1} - x_j) \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n+1} \end{pmatrix}. \tag{6.5}$$

We know that the work to solve a lower triangular system is $O(n^2)$. Thus the Newton form of the interpolant requires less work to determine than the form (6.1) coming from the Vandermonde system (6.2), and it requires less work to evaluate than the Lagrange form (6.3).

Another advantage of the Newton form over that of Vandermonde or Lagrange is that if you add a new point (x_{n+2}, y_{n+2}), the previously computed coefficients do not change. This amounts to adding one more row at the bottom of the lower triangular matrix, along with a new unknown a_{n+1} and a new right-hand side value y_{n+2}. The previously computed coefficients are then substituted into the last equation to determine a_{n+1}.

Example. Considering the same example as in section 6.2, with $x_1 = 1$, $x_2 = 2$, and $x_3 = 3$, we write the Newton form of the interpolation polynomial as

$$p(x) = a_0 + a_1(x - 1) + a_2(x - 1)(x - 2).$$

Since $y_1 = 2$, $y_2 = 3$, and $y_3 = 6$, we determine the coefficients, a_0, a_1, and a_2, by solving the lower triangular system:

$$\begin{pmatrix} 1 & & \\ 1 & 1 & \\ 1 & 2 & 2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 6 \end{pmatrix},$$

giving $a_0 = 2$, $a_1 = 1$, and $a_2 = 1$. You should check that this is the same polynomial, $x^2 - 2x + 3$, computed in section 6.2.

Suppose we add another data point $(x_4, y_4) = (5, 7)$. Then the polynomial that interpolates these 4 points is of degree 3 and can be written in the form

$$q(x) = b_0 + b_1(x - 1) + b_2(x - 1)(x - 2) + b_3(x - 1)(x - 2)(x - 3),$$

where b_0, b_1, b_2, and b_3 satisfy

$$\begin{pmatrix} 1 & & & \\ 1 & 1 & & \\ 1 & 2 & 2 & \\ 1 & 4 & 12 & 24 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 6 \\ 7 \end{pmatrix},$$

Since the first three equations involving b_0, b_1, and b_2 are the same as those for a_0, a_1, and a_2 above, it follows that $b_i = a_i$, $i = 0, 1, 2$. The equation for b_3 becomes $a_0 + 4a_1 + 12a_2 + 24b_3 = 7$, so that $b_3 = -11/24$.

Returning to the quadratic interpolant, it should be noted that the form of the Newton polynomial (though not the polynomial itself) depends on the ordering of the data points. Had we labeled the data points as $(x_1, y_1) = (3, 6)$, $(x_2, y_2) = (1, 2)$, and $(x_3, y_3) = (2, 3)$, then we would have written the interpolation polynomial in the form

$$p(x) = \alpha_0 + \alpha_1(x - 3) + \alpha_2(x - 3)(x - 1),$$

and we would have determined α_0, α_1, and α_2 from

$$\begin{aligned} p(3) &= \alpha_0 = 6 \\ p(1) &= \alpha_0 - 2\alpha_1 = 2 \Rightarrow \alpha_1 = 2 \\ p(2) &= \alpha_0 - \alpha_1 - \alpha_2 = 3 \Rightarrow \alpha_2 = 1. \end{aligned}$$

Again you should check that this is the same polynomial $x^2 - 2x + 3$.

6.3.1 Divided Differences

A disadvantage of setting up the triangular system (6.5) is that some of the entries may overflow or underflow. If the nodes x_j are far apart, say, the distance between successive nodes is greater than 1, then the product of distances from node $n+1$ to each of the others is likely to be huge and might result in an overflow. On the other hand, if the nodes are close together, say all are contained in an interval of width less than 1, then the product of the distances from node $n+1$ to each of the others is likely to be tiny and might underflow. For these reasons a different algorithm, which also requires $O(n^2)$ operations, is often used to determine the coefficients $a_0, \ldots, a_n$.

At this point, it is convenient to change notation slightly. Assume that the ordinates y_i are the values of some function f at x_i: $y_i \equiv f(x_i)$. We will sometimes abbreviate $f(x_i)$ as simply f_i.

Given a collection of nodes $x_{i_1}, x_{i_2}, \ldots, x_{i_{k+1}}$, from among the nodes $x_1, \ldots, x_{n+1}$, **define** $f[x_{i_1}, x_{i_2}, \ldots, x_{i_{k+1}}]$ (called a **kth order divided difference**) to be the coefficient of x^k in the polynomial of degree k that interpolates $(x_{i_1}, f_{i_1}), (x_{i_2}, f_{i_2}), \ldots, (x_{i_{k+1}}, f_{i_{k+1}})$. In the Newton form of the interpolant,

$$\alpha_0 + \alpha_1(x - x_{i_1}) + \alpha_2(x - x_{i_2})(x - x_{i_1}) + \ldots + \alpha_k \prod_{j=1}^{k}(x - x_{i_j}),$$

this is the coefficient α_k of $\prod_{j=1}^{k}(x - x_{i_j})$. Note that for higher degree Newton interpolants, interpolating $(x_{i_1}, f_{i_1}), \ldots, (x_{i_{k+1}}, f_{i_{k+1}}), (x_{i_{k+2}}, f_{i_{k+2}}), \ldots$, this is again the coefficient of the $(k+1)$st term, $\prod_{j=1}^{k}(x - x_{i_j})$, since these coefficients do not change when more data points are added. Thus $f[x_1, \ldots, x_{k+1}]$ is the coefficient a_k in (6.4).

With this definition we find

$$f[x_j] = f_j,$$

since $p_j(x) \equiv f_j$ is the polynomial of degree 0 that interpolates (x_j, f_j). For any two nodes x_i and x_j, $j \neq i$, we have

$$f[x_i, x_j] = \frac{f_j - f_i}{x_j - x_i} = \frac{f[x_j] - f[x_i]}{x_j - x_i},$$

since $p_{i,j}(x) \equiv f_i + \frac{f_j - f_i}{x_j - x_i}(x - x_i)$ is the polynomial of degree 1 that interpolates (x_i, f_i) and (x_j, f_j). Proceeding in this way, one finds that

$$f[x_i, x_j, x_k] = \frac{f[x_j, x_k] - f[x_i, x_j]}{x_k - x_i},$$

and more generally:

Theorem 6.3.1. *The kth order divided difference $f[x_1, x_2, \ldots, x_{k+1}]$ satisfies*

$$f[x_1, x_2, \ldots, x_{k+1}] = \frac{f[x_2, \ldots, x_{k+1}] - f[x_1, \ldots, x_k]}{x_{k+1} - x_1}. \tag{6.6}$$

Before proving this result, let us see how it can be used to compute the coefficients of the Newton interpolant recursively. We can form a table whose columns are divided differences of order 0, 1, 2, etc.

$f[x_1] = f_1$			
$f[x_2] = f_2$	$f[x_1, x_2]$		
$f[x_3] = f_3$	$f[x_2, x_3]$	$f[x_1, x_2, x_3]$	
$f[x_4] = f_4$	$f[x_3, x_4]$	$f[x_2, x_3, x_4]$	$f[x_1, x_2, x_3, x_4]$

The diagonal entries of this table are the coefficients of the Newton interpolant (6.4). We know the first column of the table. Using Theorem 6.3.1, we can compute the entries of each successive column using the entry to the left and the one just above it. If d_{ij} is the (i, j) entry of the table, then

$$d_{ij} = \frac{d_{i,j-1} - d_{i-1,j-1}}{x_i - x_{i-j+1}}.$$

Proof of Theorem. Let p be the polynomial of degree k interpolating $(x_1, f_1), \ldots, (x_{k+1}, f_{k+1})$. Let q be the polynomial of degree $k-1$ that interpolates $(x_1, f_1), \ldots, (x_k, f_k)$, and let r be the polynomial of degree $k-1$ that interpolates $(x_2, f_2), \ldots, (x_{k+1}, f_{k+1})$. Then p, q, and r have leading coefficients $f[x_1, \ldots, x_{k+1}]$, $f[x_1, \ldots, x_k]$, and $f[x_2, \ldots, x_{k+1}]$, respectively.

We claim that:

$$p(x) = q(x) + \frac{x - x_1}{x_{k+1} - x_1}(r(x) - q(x)). \tag{6.7}$$

To prove this, we will show that this equation holds at each point x_i, $i = 1, \ldots, k+1$. Since $p(x)$ is a polynomial of degree k and the right-hand side is a polynomial of degree k, showing that these two polynomials agree at $k+1$ points will imply that they are equal everywhere. For $x = x_1$, we have $p(x_1) = q(x_1) = f_1$, and the second term on the right-hand side of (6.7) is zero, so we have equality. For $i = 2, \ldots, k$, $p(x_i) = q(x_i) = r(x_i) = f_i$, and so we again have equality in (6.7) for $x = x_i$. Finally, for $x = x_{k+1}$, we have $p(x_{k+1}) = r(x_{k+1}) = f_{k+1}$, and the right-hand side of (6.7) is $q(x_{k+1}) + r(x_{k+1}) - q(x_{k+1}) = f_{k+1}$, so we again have equality. This establishes (6.7), and from

this it follows that the coefficient of x^k in $p(x)$ is $1/(x_{k+1} - x_1)$ times the coefficient of x^{k-1} in $r(x) - q(x)$; i.e.,

$$f[x_1, \ldots, x_{k+1}] = \frac{f[x_2, \ldots, x_{k+1}] - f[x_1, \ldots, x_k]}{x_{k+1} - x_1}.$$

□

Example. Looking again at the example from the previous sections, with $(x_1, y_1) = (1, 2)$, $(x_2, y_2) = (2, 3)$, and $(x_3, y_3) = (3, 6)$, we compute the following table of divided differences:

$f[x_1] = 2$		
$f[x_2] = 3$	$f[x_1, x_2] = \frac{3-2}{2-1} = 1$	
$f[x_3] = 6$	$f[x_2, x_3] = \frac{6-3}{3-2} = 3$	$f[x_1, x_2, x_3] = \frac{3-1}{3-1} = 1$

Reading off the diagonal entries, we see that the coefficients in the Newton form of the interpolation polynomial are $a_0 = 2$, $a_1 = 1$, and $a_2 = 1$.

6.4 The Error in Polynomial Interpolation

Suppose that we interpolate a smooth function f at $n+1$ points using a polynomial p of degree n. How large is the difference between $f(x)$ and $p(x)$ at points x other than the interpolation points (where the difference is 0)? If we use higher and higher degree polynomials, interpolating f at more and more points, will the polynomials eventually approximate f well at all points throughout some interval? These questions will be addressed in this section. The answers may surprise you.

Theorem 6.4.1. *Assume that $f \in C^{n+1}[a, b]$ and that $x_1, \ldots, x_{n+1}$ are in $[a, b]$. Let $p(x)$ be the polynomial of degree n that interpolates f at $x_1, \ldots, x_{n+1}$. Then for any point x in $[a, b]$,*

$$f(x) - p(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) \prod_{j=1}^{n+1} (x - x_j), \tag{6.8}$$

for some point ξ_x in $[a, b]$.

Proof. The result clearly holds if x is one of the nodes $x_1, \ldots, x_{n+1}$, so fix $x \in [a, b]$ to be some point other than these. Let q be the polynomial that interpolates f at $x_1, \ldots, x_{n+1}$ and x. Then

$$q(t) = p(t) + \lambda \prod_{j=1}^{n+1} (t - x_j), \quad \text{where } \lambda = \frac{f(x) - p(x)}{\prod_{j=1}^{n+1} (x - x_j)}. \tag{6.9}$$

Define $\phi(t) \equiv f(t) - q(t)$. Since $\phi(t)$ vanishes at the $n+2$ points $x_1, \ldots, x_{n+1}$ and x, Rolle's theorem implies that $\phi'(t)$ vanishes at $n+1$ points between successive pairs. Applying Rolle's theorem again,

we find that ϕ'' vanishes at n points, and continuing, that $\phi^{(n+1)}$ must vanish at at least one point ξ_x in the interval $[a, b]$. Hence $0 = \phi^{(n+1)}(\xi_x) = f^{(n+1)}(\xi_x) - q^{(n+1)}(\xi_x) = f^{(n+1)}(\xi_x) - \lambda(n+1)!$, so that

$$f^{(n+1)}(\xi_x) = \frac{f(x) - p(x)}{\prod_{j=1}^{n+1}(x - x_j)} \cdot (n+1)!,$$

and the result (6.8) follows. □

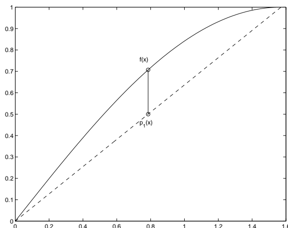
The error in polynomial interpolation can also be expressed in terms of *divided differences*. The divided difference formulas of the previous subsection may have reminded you somewhat of *derivatives*. That is because divided differences are approximations to derivatives; the nth order divided difference approximates $(1/n!)$ times the nth derivative. Consider again the polynomial q in (6.9) that interpolates f at the points $x_1, \ldots, x_{n+1}$ and x. The coefficient λ in that formula (the coefficient of the highest power of t) is, by definition, the divided difference $f[x_1, \ldots, x_{n+1}, x]$. Since $q(x) = f(x)$, it follows that

$$f(x) = p(x) + f[x_1, \ldots, x_{n+1}, x] \cdot \prod_{j=1}^{n+1}(x - x_j). \tag{6.10}$$

Comparing this with (6.8), we see that

$$f[x_1, \ldots, x_{n+1}, x] = \frac{1}{(n+1)!} f^{(n+1)}(\xi_x). \tag{6.11}$$

Example. Let $f(x) = \sin x$, and let p_1 be the first degree polynomial that interpolates f at 0 and $\pi/2$. Then $p_1(x) = (2/\pi)x$. The two functions are plotted below on the interval $[0, \pi/2]$.



Since $|f''(x)| = |\sin x| \leq 1$, it follows from Theorem 6.4.1 that for any point x,

$$|f(x) - p_1(x)| \leq \frac{1}{2!}|(x - 0)(x - \pi/2)|.$$

For $x \in [0, \pi/2]$, this is maximal when $x = \pi/4$, and then

$$|f(x) - p_1(x)| \leq \frac{1}{2}(\pi/4)^2.$$

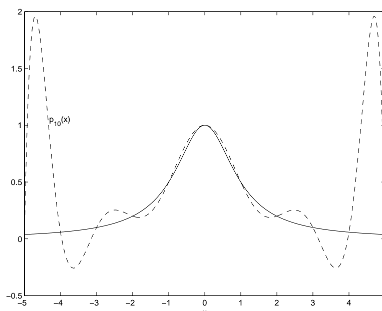
The actual error is $|\sin(\pi/4) - (2/\pi)(\pi/4)| = (\sqrt{2} - 1)/2 \approx .207$, and we obtain a reasonably good approximation within the interval $[0, \pi/2]$.

Suppose, however, that we use $p_1(x)$ to *extrapolate* to the value of f at π. The error in this case will be 2, since $\sin\pi = 0$ while $p_1(\pi) = 2$. Theorem 6.4.1 guarantees only that

$$|f(\pi) - p_1(\pi)| \le \frac{1}{2}|(\pi - 0)(\pi - \pi/2)| = \pi^2/4.$$

Suppose we add more and more interpolation points between 0 and $\pi/2$. Will we ever be able to extrapolate to $x = \pi$ and obtain a reasonable approximation to $f(\pi)$? In this case, the answer is yes, since all derivatives of $f(x) = \sin x$ are bounded in absolute value by 1. As the number of interpolation points $n+1$ increases, the factor $1/(n+1)!$ in (6.8) decreases. The factor $|\prod_{j=1}^{n+1}(x - x_j)|$ increases if $x = \pi$ and each $x_j \in [0, \pi/2]$, but it increases less rapidly than $(n+1)!$: $\lim_{n\to\infty} \pi^{n+1}/(n+1)! = 0$.

Runge Example. Consider the function $f(x) = \frac{1}{1+x^2}$ on the interval $I = [-5, 5]$. Suppose we interpolate f at more and more equally spaced points throughout the interval I. Will the sequence of interpolants p_n converge to f uniformly in I? The answer is **NO**. The function f and its 10th degree interpolant p_{10} are plotted below.



As the degree n increases, the oscillations near the ends of the interval become larger and larger. This does not contradict Theorem 6.4.1, because in this case the quantity $|f^{(n+1)}(\xi_x)\prod_{j=1}^{n+1}(x - x_j)|$ in (6.8) grows even faster with n than $(n+1)!$ near $x = \pm 5$.

For examples like the Runge example, one can do somewhat better by clustering the interpolation points near the endpoints of the interval rather than having them equally spaced. Sometimes the *Chebyshev interpolation points*

$$x_i = \cos\left(\frac{2(n-i)+1}{2n+2} \cdot \pi\right), \quad i = 0, 1, \ldots, n,$$

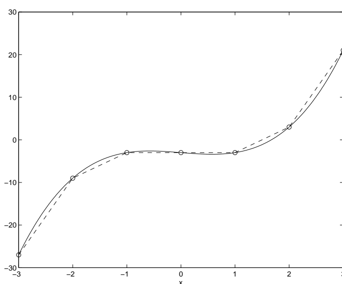
(or scaled and translated versions of these points) are used for interpolation. But for some continuous functions the Chebyshev interpolant diverges as well. For this reason, high degree polynomial interpolation is not recommended. It is usually better to use low degree piecewise polynomial interpolation or a least squares fit.

6.5 Piecewise Polynomial Interpolation

Another way to approximate a function f on an interval $[a, b]$ is to divide the interval into n subintervals, each of length $h \equiv (b-a)/n$, and to use a low degree polynomial to approximate f on each subinterval. For example, if the endpoints of these subintervals are $a \equiv x_0, x_1, \ldots, x_{n-1}, x_n \equiv b$, and $\ell(x)$ is the *piecewise linear* interpolant of f, then

$$\ell(x) = f(x_{i-1})\frac{x - x_i}{x_{i-1} - x_i} + f(x_i)\frac{x - x_{i-1}}{x_i - x_{i-1}}, \quad \text{in} \;\; [x_{i-1}, x_i].$$

[Note that we have written down the Lagrange form of $\ell(x)$.] A function and its piecewise linear interpolant are pictured below.



There are many reasons why one might wish to approximate a function f with piecewise polynomials. For example, suppose one wants to compute $\int_a^b f(x)\,dx$. It might be impossible to integrate f analytically, but it is easy to integrate polynomials. Using the piecewise linear interpolant of f, one would obtain the approximation

$$\begin{aligned}
\int_a^b f(x)\,dx &\approx \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} [f_{i-1}\frac{x - x_i}{x_{i-1} - x_i} + f_i\frac{x - x_{i-1}}{x_i - x_{i-1}}]\,dx \\
&= \sum_{i=1}^{n} [f_{i-1}\frac{h}{2} + f_i\frac{h}{2}] \\
&= \frac{h}{2}[f_0 + 2f_1 + \ldots + 2f_{n-1} + f_n],
\end{aligned}$$

where we have again used the abbreviation $f_j \equiv f(x_j)$. You may recognize this as the *trapezoid rule*. Numerical integration methods will be discussed in Chapter 8.

Another application is interpolation in a table (although this is becoming less and less prevalent with the availability of calculators). For instance, suppose you know that $\sin(\pi/6) = 1/2$ and $\sin(\pi/4) = \sqrt{2}/2$. How can you estimate, say, $\sin(\pi/5)$? One possibility would be to use the linear interpolant of $\sin(x)$ on the interval $[\pi/6, \pi/4]$. This is

$$\ell(x) = \frac{1}{2} \cdot \frac{x - \pi/4}{\pi/6 - \pi/4} + \frac{\sqrt{2}}{2} \cdot \frac{x - \pi/6}{\pi/4 - \pi/6}.$$

Hence $\ell(\pi/5) = (1/2)\cdot(3/5) + (\sqrt{2}/2)\cdot(2/5) = (3 + 2\sqrt{2})/10 \approx .58$.

What can be said about the *error* in the piecewise linear interpolant? We know from Theorem 6.4.1 that in the subinterval $[x_{i-1}, x_i]$,

$$f(x) - \ell(x) = \frac{f''(\xi_x)}{2!}(x - x_{i-1})(x - x_i),$$

for some $\xi_x \in [x_{i-1}, x_i]$. If $|f''(x)| \leq M$ for all $x \in [x_{i-1}, x_i]$, then

$$|f(x) - \ell(x)| \leq \frac{M}{2} \max_{x \in [x_{i-1}, x_i]} |(x - x_{i-1})(x - x_i)| = \frac{M}{2}\left(\frac{h}{2}\right)^2 = \frac{Mh^2}{8},$$

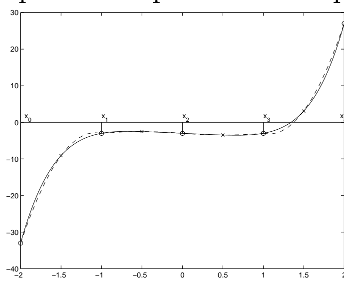
and if $|f''(x)| \leq M$ for all $x \in [a, b]$, then this same estimate will hold for each subinterval. Hence

$$|f(x) - \ell(x)| \leq \frac{Mh^2}{8} \quad \forall x \in [a, b].$$

To make the error less than some desired tolerance δ one can choose h so that $Mh^2/8 < \delta$; i.e., $h < \sqrt{8\delta/M}$. Thus the piecewise linear interpolant always converges to the function $f \in C^2[a, b]$ as the subinterval size h goes to 0.

For convenience, we used a fixed mesh spacing $h \equiv (b-a)/n = x_i - x_{i-1}$, $i = 1, \ldots, n$. To obtain an accurate approximation with fewer subintervals, one might put more of the nodes x_i in regions where f is highly nonlinear and fewer nodes in regions where the function can be well-approximated by a straight line. This can be done adaptively. One might start with fairly large subintervals, in each subinterval $[x_{i-1}, x_i]$ test the difference between $f((x_i + x_{i-1})/2)$ and $\ell((x_i + x_{i-1})/2)$, and if this difference is greater than δ, then replace that subinterval by two subintervals, each of width $(x_i - x_{i-1})/2$. This adaptive strategy is not foolproof; it may happen that f and ℓ are in close agreement at the midpoint of the subinterval but not at other points, and then the subinterval will not be divided as it should be. This strategy often works well, however, and gives a good piecewise linear approximation to f at a lower cost than using many equally sized subintervals.

Another way to increase the accuracy of the interpolant is to use higher degree piecewise polynomials; for example, quadratics that interpolate f at the midpoints as well as the endpoints of the subintervals. A function and its piecewise quadratic interpolant are pictured below.



We know from Theorem 6.4.1 that the error in the quadratic interpolant $q(x)$ in interval $[x_{i-1}, x_i]$ is

$$f(x) - q(x) = \frac{f'''(\xi_x)}{3!}(x - x_{i-1})(x - \frac{x_i + x_{i-1}}{2})(x - x_i),$$

for some point $\xi_x \in [x_{i-1}, x_i]$. If $|f'''(x)/3!|$ is bounded by a constant M for all $x \in [a, b]$, then the error in the quadratic interpolant decreases as $O(h^3)$, since the factor $|(x-x_{i-1})(x-\frac{x_i+x_{i-1}}{2})(x-x_i)|$ is less than h^3 for any $x \in [x_{i-1}, x_i]$.

This piecewise quadratic interpolant, like the piecewise linear interpolant, is *continuous*, but its derivative, in general, is *not* continuous. Another idea might be to require: $q(x_i) = f(x_i)$ *and* $q'(x_i) = f'(x_i)$, $i = 0, 1, \ldots, n$. But if we try to achieve this with a piecewise quadratic q, then within each subinterval $[x_{i-1}, x_i]$, q' must be the linear interpolant of f':

$$q'(x) = f'(x_{i-1}) \frac{x - x_i}{x_{i-1} - x_i} + f'(x_i) \frac{x - x_{i-1}}{x_i - x_{i-1}}, \quad x \in [x_{i-1}, x_i].$$

Integrating q', we find that $q(x) = \int_{x_{i-1}}^{x} q'(t)\, dt + C$, for some constant C. In order that $q(x_{i-1})$ be equal to $f(x_{i-1})$, we must have $C = f(x_{i-1})$. But now that C is fixed, there is no way to force $q(x_i) = f(x_i)$. Thus the goal of matching function values and first derivatives cannot be achieved with piecewise quadratics.

6.5.1 Piecewise Cubic Hermite Interpolation

Suppose instead that we use piecewise cubics p and try to match f and f' at the nodes. A quadratic p' that matches f' at x_{i-1} and x_i can be written in the form

$$p'(x) = f'(x_{i-1}) \frac{x - x_i}{x_{i-1} - x_i} + f'(x_i) \frac{x - x_{i-1}}{x_i - x_{i-1}} + \alpha(x - x_{i-1})(x - x_i),$$

for $x \in [x_{i-1}, x_i]$. Here we have introduced an additional parameter α that can be varied in order to match function values. Integrating, we find

$$p(x) = -\frac{f'(x_{i-1})}{h} \int_{x_{i-1}}^{x} (t - x_i)\, dt + \frac{f'(x_i)}{h} \int_{x_{i-1}}^{x} (t - x_{i-1})\, dt + \alpha \int_{x_{i-1}}^{x} (t - x_{i-1})(t - x_i)\, dt + C.$$

The condition $p(x_{i-1}) = f(x_{i-1})$ implies that $C = f(x_{i-1})$. Making this substitution and performing the necessary integrations then gives

$$p(x) = -\frac{f'(x_{i-1})}{h} [\frac{(x - x_i)^2}{2} - \frac{h^2}{2}] + \frac{f'(x_i)}{h} \frac{(x - x_{i-1})^2}{2} + \alpha(x - x_{i-1})^2 (\frac{x - x_{i-1}}{3} - \frac{h}{2}) + f(x_{i-1}). \tag{6.12}$$

The condition $p(x_i) = f(x_i)$ then implies that

$$p(x_i) = f'(x_{i-1}) \frac{h}{2} + f'(x_i) \frac{h}{2} - \alpha \frac{h^3}{6} + f(x_{i-1}) = f(x_i) \Longrightarrow$$

$$\alpha = \frac{3}{h^2} (f'(x_{i-1}) + f'(x_i)) + \frac{6}{h^3} (f(x_{i-1}) - f(x_i)). \tag{6.13}$$

Example. Let $f(x) = x^4$ on $[0, 2]$. Find the piecewise cubic Hermite interpolant of f using the two subintervals $[0, 1]$ and $[1, 2]$.

For this problem $x_0 = 0$, $x_1 = 1$, $x_2 = 2$, and $h = 1$. Substituting these values and the values of f and its derivative into (6.12) and (6.13), we find that

$$p(x) = \begin{cases} 2x^3 - x^2 & \text{in } [0,1] \\ 6x^3 - 13x^2 + 12x - 4 & \text{in } [1,2] \end{cases}$$

You should check that p and p' match f and f' at the nodes 0, 1, and 2.

6.5.2 Cubic Spline Interpolation

The piecewise cubic Hermite interpolant has one continuous derivative. If we give up the requirement that the derivative of our interpolant match that of f at the nodes, then we can obtain an even smoother piecewise cubic interpolant. A **cubic spline** interpolant s is a piecewise cubic that interpolates f at the nodes (also called *knots*) $x_0, \ldots, x_n$ and has *two* continuous derivatives.

We will not be able to write down a formula for $s(x)$ in $[x_{i-1}, x_i]$ that involves only values of f and its derivatives in this subinterval. Instead, we will have to solve a global linear system, involving the values of f at all of the nodes, to determine the formula for $s(x)$ in all of the subintervals simultaneously. Before doing this, let us count the number of equations and unknowns (also called *degrees of freedom*). A cubic polynomial is determined by 4 parameters; since s is cubic in each subinterval and there are n subintervals, the number of unknowns is $4n$. The conditions $s(x_i) = f_i$, $i = 0, 1, \ldots, n$ provide $2n$ equations, since the cubic in each of the n subintervals must match f at 2 points. The conditions that s' and s'' be continuous at $x_1, \ldots, x_{n-1}$ provide an additional $2n - 2$ equations, for a total of $4n - 2$ equations in $4n$ unknowns. This leaves 2 degrees of freedom. These can be used to enforce various conditions on the endpoints of the spline, resulting in different types of cubic splines.

To derive the equations of a cubic spline interpolant, let us start by setting $z_i = s''(x_i)$, $i = 1, \ldots, n-1$. Suppose now that the values z_i were known. Since s'' is linear within each subinterval, it must satisfy

$$s''(x) = z_{i-1} \frac{x - x_i}{x_{i-1} - x_i} + z_i \frac{x - x_{i-1}}{x_i - x_{i-1}} \quad \text{in } [x_{i-1}, x_i].$$

We will simplify the notation somewhat by assuming a uniform mesh spacing, $x_i - x_{i-1} = h$ for all i; this is not necessary but it does make the presentation simpler. With this assumption, then, and letting s_i denote the restriction of s to subinterval i, we can write

$$s_i''(x) = \frac{1}{h} z_{i-1}(x_i - x) + \frac{1}{h} z_i (x - x_{i-1}). \tag{6.14}$$

Integrating, we find

$$s_i'(x) = -\frac{1}{h} z_{i-1} \frac{(x_i - x)^2}{2} + \frac{1}{h} z_i \frac{(x - x_{i-1})^2}{2} + C_i, \tag{6.15}$$

and

$$s_i(x) = \frac{1}{h} z_{i-1} \frac{(x_i - x)^3}{6} + \frac{1}{h} z_i \frac{(x - x_{i-1})^3}{6} + C_i(x - x_{i-1}) + D_i, \tag{6.16}$$

for some constants C_i and D_i.

Now we begin applying the necessary conditions to s_i. From (6.16), the condition $s_i(x_{i-1}) = f_{i-1}$ implies that

$$\frac{h^2}{6} z_{i-1} + D_i = f_{i-1}, \quad \text{or} \quad D_i = f_{i-1} - \frac{h^2}{6} z_{i-1}. \tag{6.17}$$

The condition $s_i(x_i) = f_i$ implies, with (6.16) and (6.17), that

$$\frac{h^2}{6} z_i + C_i h + f_{i-1} - \frac{h^2}{6} z_{i-1} = f_i,$$

or,

$$C_i = \frac{1}{h}\left[f_i - f_{i-1} + \frac{h^2}{6}(z_{i-1} - z_i) \right]. \tag{6.18}$$

Making these substitutions in (6.16) gives

$$s_i(x) = \frac{1}{h} z_{i-1} \frac{(x_i - x)^3}{6} + \frac{1}{h} z_i \frac{(x - x_{i-1})^3}{6} + \frac{1}{h}\left[f_i - f_{i-1} + \frac{h^2}{6}(z_{i-1} - z_i) \right](x - x_{i-1}) + f_{i-1} - \frac{h^2}{6} z_{i-1}. \tag{6.19}$$

Once the parameters $z_1, \ldots, z_{n-1}$ are known, one can use this formula to evaluate s anywhere.

We will use the continuity of s' to determine $z_1, \ldots, z_{n-1}$. The condition $s_i'(x_i) = s_{i+1}'(x_i)$, together with (6.15) and (6.18), implies that

$$\frac{h}{2} z_i + \frac{1}{h}(f_i - f_{i-1}) + \frac{h}{6}(z_{i-1} - z_i) = -\frac{h}{2} z_i + \frac{1}{h}(f_{i+1} - f_i) + \frac{h}{6}(z_i - z_{i+1}), \quad i = 1, \ldots, n-1.$$

Moving the unknown z_j's to the left-hand side and the known f_j's to the right-hand side, this becomes

$$\frac{2h}{3} z_i + \frac{h}{6} z_{i-1} + \frac{h}{6} z_{i+1} = -\frac{2}{h} f_i + \frac{1}{h} f_{i-1} + \frac{1}{h} f_{i+1}, \quad i = 1, \ldots, n-1.$$

This is a symmetric tridiagonal system of linear equations for the values $z_1, \ldots, z_{n-1}$:

$$\begin{pmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_{n-2} \\ & & \beta_{n-2} & \alpha_{n-1} \end{pmatrix} \begin{pmatrix} z_1 \\ \vdots \\ \vdots \\ z_{n-1} \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_{n-1} \end{pmatrix}, \tag{6.20}$$

where

$$\alpha_i = \frac{2h}{3}, \quad \beta_i = \frac{h}{6} \quad \forall i,$$

$$\begin{aligned} b_i &= \frac{1}{h}(f_{i+1} - 2f_i + f_{i-1}), \quad i = 2, \ldots, n-2, \\ b_1 &= \frac{1}{h}(f_2 - 2f_1 + f_0) - \frac{h}{6} z_0, \\ b_{n-1} &= \frac{1}{h}(f_n - 2f_{n-1} + f_{n-2}) - \frac{h}{6} z_n. \end{aligned}$$

We are free to choose z_0 and z_n, and different choices give rise to different kinds of splines. The choice $z_0 = z_n = 0$ gives what is called the *natural cubic spline*. This can be shown to be the C^2 piecewise cubic interpolant of f whose curvature is (approximately) minimal. The *complete spline* is obtained by choosing z_0 and z_1 so that the first derivative of the spline matches that of f at the endpoints (or takes on some given values at the endpoints). The *not-a-knot spline* is obtained by choosing z_0 and z_n to ensure third derivative continuity at both x_1 and x_{n-1}. This choice may be appropriate if no endpoint derivative information is available.

Example. Let $f(x) = x^4$ on $[0, 2]$. Find the natural cubic spline interpolant of f using the two subintervals $[0, 1]$ and $[1, 2]$.

Since this problem has only two subintervals ($n = 2$), the tridiagonal system (6.20) consists of just one equation in one unknown. Thus $\alpha_1 = 2/3$ and $b_1 = f_2 - 2f_1 + f_0 = 14$, so $z_1 = 21$. Using this value, together with $z_0 = z_2 = 0$, in (6.19) we can find s_1 and s_2:

$$\begin{aligned} s_1(x) &= \frac{7}{2}x^3 - \frac{5}{2}x \\ s_2(x) &= -\frac{7}{2}x^3 + 21x^2 - \frac{47}{2}x + 7. \end{aligned}$$

You should check that all of the conditions:

$$s_1(0) = f(0), \quad s_1(1) = s_2(1) = f(1), \quad s_2(2) = f(2),$$

$$s_1'(1) = s_2'(1), \quad s_1''(1) = s_2''(1),$$

and

$$s_1''(0) = s_2''(2) = 0,$$

are satisfied.

Exercises

1. Following is census data showing the population of the U.S. between 1900 and 2000:

years after 1900	population in millions
0	76.0
20	105.7
40	131.7
60	179.3
80	226.5
100	281.4

   (a) Plot population versus years after 1900 using MATLAB by entering the years after 1900 into a vector x and the populations into a vector y and giving the command `plot(x,y,'o')`. Find the fifth degree polynomial that passes through each data point, and determine its value in the year 2020. Plot this polynomial on the same graph as the population data (but with the x-axis now extending from 0 through 120). Do you think that this a reasonable way to estimate the population of the U.S. in future years? [To see a demonstration of other methods of extrapolation, type `census` in MATLAB.]

   Hand in: Your plot and the value that you estimated for the population in the year 2020 along with your explanation as to why this is or is not a good way to estimate the population.

   (b) Write down the *Lagrange form* of the second degree polynomial that interpolates the population in the years 1900, 1920, and 1940.

   (c) Determine the coefficients of the *Newton form* of the interpolants of degrees 0, 1, and 2, that interpolate the first one, two, and three data points, respectively. Verify that the second degree polynomial that you construct here is identical to that in part (b).

2. The secant method for finding a root of a function $f(x)$ fits a first degree polynomial (i.e., a straight line) through the points $(x_{k-1}, f(x_{k-1}))$ and $(x_k, f(x_k))$ and takes the root of this polynomial as the next approximation x_{k+1}. Another root finding algorithm, called *Muller's method*, fits a quadratic through the three points, $(x_{k-2}, f(x_{k-2}))$, $(x_{k-1}, f(x_{k-1}))$, and $(x_k, f(x_k))$, and takes the root of this quadratic that is closest to x_k as the next approximation x_{k+1}. Write down a formula for this quadratic. Suppose $f(x) = x^3 - 2$, $x_0 = 0$, $x_1 = 1$, and $x_2 = 2$. Find x_3.

3. Use MATLAB to fit a polynomial of degree 12 to the Runge function

$$f(x) = \frac{1}{1 + x^2},$$

   interpolating the function at 13 equally spaced points between -5 and 5. (You can set the points with the command `x = [-5:5/6:5];`.) You may use MATLAB routine `polyfit` to

do this computation or you may use your own routine. (To find out how to use `polyfit` type `help polyfit` in MATLAB.) Plot the function and your 12th degree interpolant on the same graph. Turn in your plot and a listing of your code.

4. Write down a divided difference table and the Newton form of the interpolating polynomial for the following set of data:

x	1	3/2	0	2
$f(x)$	2	6	0	14

5. Let $f(x) = x^2(x-1)^2(x-2)^2(x-3)^2$. What is the piecewise cubic Hermite interpolant of f on the grid $x_0 = 0$, $x_1 = 1$, $x_2 = 2$, $x_3 = 3$? Let $g(x) = ax^3 + bx^2 + cx + d$ for some parameters a, b, c, and d. Write down the piecewise cubic Hermite interpolant of g on the same grid. [Note: You should not need to do any arithmetic for either of these problems.]

6. Determine all the values of a, b, c, d, e, and f for which the following function is a cubic spline:

$$s(x) = \begin{cases} ax^2 + b(x-1)^3 & x \in (-\infty, 1] \\ cx^2 + d & x \in [1, 2] \\ ex^2 + f(x-2)^3 & x \in [2, \infty) \end{cases}$$

7. Use MATLAB's `spline` function to find the not-a-knot cubic spline interpolant of $f(x) = e^{-2x}\sin(10\pi x)$, using $n = 20$ subintervals of equal size between $x_0 = 0$ and $x_21 = 1$. [To learn how to use the spline routine, type `help spline` in MATLAB.] Plot the function along with its cubic spline interpolant, marking the knots on the curves.

8. Let f be a given function satisfying $f(0) = 1$, $f(1) = 2$, and $f(2) = 0$. A *quadratic spline interpolant* $r(x)$ is defined as a piecewise quadratic that interpolates f at the nodes ($x_0 = 0$, $x_1 = 1$, and $x_2 = 2$) and whose first derivative is continuous throughout the interval. Find the quadratic spline interpolant of f that also satisfies $r'(0) = 0$. [Hint: Start from the left subinterval.]