Numerical Methods for Civil and Mechanical Engineers
Class Notes for MATH 344

Todd Young and Martin Mohlenkamp
Department of Mathematics
Ohio University
Athens, OH 45701
young@math.ohiou.edu

May 9, 2008

Preface

These notes were developed by the first author in the process of teaching a course on applied numerical methods for Civil Engineering majors during 2002-2004 and was modified to include Mechanical Engineering in 2005. The materials have been periodically updated since then and underwent a major revision by the second author in 2006-2007.

The main goals of these lectures are to introduce concepts of numerical methods and introduce MATLAB in an Engineering framework. By this we do not mean that every problem is a "real life" engineering application, but more that the engineering way of thinking is emphasized throughout the discussion.

The philosophy of this book was formed over the course of many years. I grew up with a Civil Engineer father and spent a large portion of my youth surveying with him in Kentucky. At the University of Kentucky I took most of the basic Engineering courses while getting a Bachelor's degree in Mathematics. Immediately afterward I completed a M.S. degree in Engineering Mechanics at Kentucky. I later obtained a Ph.D. in Mathematics at Georgia Tech. During my education, I observed that incorporation of computation in coursework had been extremely unfocused and poor. For instance during my college career I had to learn 8 different programming and markup languages on 4 different platforms plus numerous other software applications. There was almost no technical help provided in the courses and I wasted innumerable hours figuring out software on my own. A typical, but useless, inclusion of software has been (and still is in most calculus books) to set up a difficult 'applied' problem and then add the line "write a program to solve" or "use a computer algebra system to solve".

At Ohio University we have tried to take a much more disciplined and focused approach. The Russ College of Engineering and Technology decided that MATLAB should be the primary computational software for undergraduates. At about the same time members of the Mathematics Department proposed an 1804 project to bring MATLAB into the calculus sequence and provide access to the program at nearly all computers on campus, including in the dorm rooms. The stated goal of this project was to make MATLAB the universal language for computation on campus. That project was approved and implemented in the 2001-2002 academic year.

In these lecture notes, instruction on using MATLAB is dispersed through the material on numerical methods. In these lectures details about how to use MATLAB are detailed (but not verbose) and explicit. To teach programming, students are usually given examples of working programs and are asked to make modifications.

The lectures are designed to be used in a computer classroom, but could be used in a lecture format with students doing computer exercises afterward. The lectures are divided into four Parts with a summary provided at the end of each Part. Throughout the text MATLAB commands are preceded by the symbol `>`, which is the prompt in the command window. Programs are surrounded by a box.

Todd Young, May 9, 2008

Contents

Part I

Matlab and Solving Equations

Lecture 1

Vectors, Functions, and Plots in Matlab

Entering vectors

In Matlab, the basic objects are matrices, i.e. arrays of numbers. Vectors can be thought of as special matrices. A row vector is recorded as a $1 \times n$ matrix and a column vector is recorded as a $m \times 1$ matrix. To enter a row vector in Matlab, type the following at the prompt (`>`) in the command window (do not type `>`):

```
> v = [0 1 2 3]
```

and press enter. Matlab will print out the row vector. To enter a column vector type:

```
> u = [0; 1; 2; 3]
```

You can change a row vector into a column vector, and vice versa easily in Matlab using:

```
> w = v'
```

(This is called *transposing* the vector and we call `'` the transpose operator.) There are also useful shortcuts to make vectors such as:

```
> x = -1:.1:1
```

and

```
> y = linspace(0,1,11)
```

In the rest of the book `>` will indicate commands to be entered in the command window.

Plotting Data

Consider the following table, obtained from experiments on the viscosity of a liquid.[1] We can enter

T (C°)	5	20	30	50	55
μ	0.08	0.015	0.009	0.006	0.0055

this data into Matlab with the following commands entered in the command window:

```
> x = [ 5  20  30  50  55 ]
> y = [ 0.08  0.015  0.009  0.006  0.0055]
```

Entering the name of the variable retrieves its current values. For instance:

```
> x
> y
```

We can plot data in the form of vectors using the plot command:

```
> plot(x,y)
```

This will produce a graph with the data points connected by lines. If you would prefer that the data points be represented by symbols you can do so. For instance:

[1]Adapted from Ayyup & McCuen 1996, p.174.

```
> plot(x,y,'*')
> plot(x,y,'o')
> plot(x,y,'.')
```

Data as a Representation of a Function

A major theme in this course is that often we are interested in a certain function $y = f(x)$, but the only information we have about this function is a discrete set of data $\{(x_i, y_i)\}$. Plotting the data, as we did above, can be thought of envisioning the function using just the data. We will find later that we can also do other things with the function, like differentiating and integrating, just using the available data. Numerical methods, the topic of this course, means doing mathematics by computer. Since a computer can only store a finite amount of information, we will almost always be working with a finite, discrete set of values of the function (data), rather than a formula for the function.

Built-in Functions

If we wish to deal with formulas for functions, MATLAB contains a number of built-in functions, including all the usual functions, such as `sin( )`, `exp( )`, etc.. The meaning of most of these is clear. The dependent variable (input) always goes in parentheses in MATLAB. For instance:

```
> sin(pi)
```

should return the value of $\sin\pi$, which is of course 0 and

```
> exp(0)
```

will return e^0 which is 1. More importantly, the built-in functions can operate not only on single numbers but on vectors. For example:

```
> x = linspace(0,2*pi,40)
> y = sin(x)
> plot(x,y)
```

will return a plot of $\sin x$ on the interval $[0, 2\pi]$

Some of the built-in functions in MATLAB include: `cos( )`, `tan( )`, `sinh( )`, `cosh( )`, `log( )` (natural logarithm), `log10( )` (log base 10), `asin( )` (inverse sine), `acos( )`, `atan( )`.

User-Defined Inline Functions

If we wish to deal with a function which is a composition of the built-in function, MATLAB has a couple of ways for the user to define functions. One which we will use a lot is the inline function, which is a way to define a function in the command window. The following is a typical inline function:

```
> f = inline('2*x.^2 - 3*x + 1','x')
```

This produces the function $f(x) = 2x^2 - 3x + 1$. To obtain a single value of this function enter:

```
> f(2.23572)
```

Just as for built-in functions, the function f as we defined it can operate not only on single numbers but on vectors. Try the following:

```
> x = -2:.2:2
> y = f(x)
```

The results can be plotted using the `plot` command, just as for data:

```
> plot(x,y)
```

The reason $f(x)$ works when x is a vector is because we represented x^2 by `x.^2`. The `.` turns the exponent operator `^` into entry-wise exponentiation. This is an example of *vectorization*, i.e. putting several numbers into a vector and treating the vector all at once, rather than one component at a time. The ability to do this is one of the main advantages of MATLAB over usual programming languages such as C, C++, or Fortran.

Also notice that before plotting the function, we in effect converted it into data. Plotting on any machine always requires this step.

Exercises

1.1 Find a table of data in an engineering textbook. Input it as vectors and plot it. Use the insert icon to label the axes and add a title to your graph. Turn in the graph. Indicate what the data is and where it came from.

1.2 Make an inline function $g(x) = x + \cos(x^5)$. Plot it using vectors `x = -5:.1:5;` and `y = g(x);`. What is wrong with this graph? Find a way to make it better. Turn in both printouts.

Lecture 2

Matlab Programs

In Matlab, programs may be written and saved in files with a suffix `.m` called *M-files*. There are two types of M-file programs: *functions* and *scripts*.

Function Programs

Begin by clicking on the new document icon in the top left of the Matlab window (it looks like an empty sheet of paper).

In the document window type the following:

```
function y = myfunc(x)
y = 2*x.^2 - 3*x + 1;
```

Save this file as: `myfunc.m` in your working directory. This file can now be used in the command window just like any predefined Matlab function; in the command window enter:

`> x = -2:.1:2;` .... Produces a vector of x values.
`> y = myfunc(x);` .... Produces a vector of y values.
`> plot(x,y)`

Note that the fact we used `x` and `y` in both the function program and in the command window was just a coincidence. We could just as well have made the function

```
function nonsense = myfunc(inputvector)
nonsense = 2*inputvector.^2 - 3*inputvector + 1;
```

Look back at the program. All function programs are like this one, the essential elements are:
- Begin with the word `function`.
- There are inputs and outputs.
- The outputs, name of the function and the inputs must appear in the first line.
- The body of the program must assign values to the outputs.

Functions can have multiple inputs and/or multiple outputs. Next let's create a function will have 1 input and 3 output variables. Open a new document and type:

```
function [x2 x3 x4]  = mypowers(x)
x2 = x.^2;
x3 = x.^3;
x4 = x.^4;
```

Save this file as `mypowers.m`. In the command window, we can use the results of the program to make graphs:

```
 > x = -1:.1:1
```

```
> [x2 x3 x4] = mypowers(x);
> plot(x,x,'black',x,x2,'blue',x,x3,'green',x,x4,'red')
```

Script Programs

MATLAB uses a second type of program that differs from a function program in several ways, namely:
- There are no inputs and outputs.
- A script program may use and change variables in the current workspace (the variables used by the command window.)

Below is a script program that accomplishes the same thing as the function program plus the commands in the previous section:

```
x2 = x.^2;
x3 = x.^3;
x4 = x.^4;
plot(x,x,'black',x,x2,'blue',x,x3,'green',x,x4,'red')
```

Type this program into a new document and save it as `mygraphs.m`. In the command window enter:

```
> x = -1:.1:1;
> mygraphs
```

.

Note that the program used the variable `x` in its calculations, even though `x` was defined in the command window, not in the program.

Many people use script programs for routine calculations that would require typing more than one command in the command window. They do this because correcting mistakes is easier in a program than in the command window.

Program Comments

For programs that have more than a couple of lines it is important to include comments. Comments allow other people to know what your program does and they also remind yourself what your program does if you set it aside and come back to it later. It is best to include comments not only at the top of a program, but also with each section. In MATLAB anything that comes in a line after a `%` is a comment. For a script program it is often helpful to include the name of the program at the beginning. For example:

```
% mygraphs
% plots the graphs of x, x^2, x^3, and x^4
% on the interval [-1,1]

% fix the domain and evaluation points
x = -1:.1:1;

% calculate powers
x2 = x.^2;
x3 = x.^3;
x4 = x.^4;

% plot each of the graphs
plot(x,x,'black',x,x2,'blue',x,x3,'green',x,x4,'red')
```

Exercises

2.1 Write a function program for the function $x^2e^{-x^2}$, using entry-wise operations (such as `.*` and `.^`). Include adequate comments in the program. Plot the function on $[-5, 5]$. Turn in printouts of the program and the graph. (The graph of the function e^{-x^2} is the bell-curve that is used in probability and statistics.)

2.2 Write a script program that graphs the functions $\sin x$, $\sin 2x$, $\sin 3x$, $\sin 4x$, $\sin 5x$ and $\sin 6x$ on the interval $[0, 2\pi]$ on one plot. (π is `pi` in MATLAB.) Include comments in the program. Turn in the program and the graph.

Lecture 3

Newton's Method and Loops

Solving equations numerically

For the next few lectures we will focus on the problem of solving an equation:

$$f(x) = 0. \tag{3.1}$$

As you learned in calculus, the final step in many optimization problems is to solve an equation of this form where f is the derivative of a function, F, that you want to maximize or minimize. In real engineering problems the function you wish to optimize can come from a large variety of sources, including formulas, solutions of differential equations, experiments, or simulations.

Newton iterations

We will denote an actual solution of equation (3.1) by x^*. There are three methods which you may have discussed in Calculus: the bisection method, the secant method and Newton's method. All three depend on beginning close (in some sense) to an actual solution x^*.

Recall Newton's method. You should know that the basis for Newton's method is approximation of a function by it linearization at a point, i.e.

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0). \tag{3.2}$$

Since we wish to find x so that $f(x) = 0$, set the left hand side ($f(x)$) of this approximation equal to 0 and solve for x to obtain:

$$x \approx x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{3.3}$$

We begin the method with the initial guess x_0, which we hope is fairly close to x^*. Then we define a sequence of points $\{x_0, x_1, x_2, x_3, \ldots\}$ from the formula:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \tag{3.4}$$

which comes from (3.3). If $f(x)$ is reasonably well-behaved near x^* and x_0 is close enough to x^*, then it is a fact that the sequence will converge to x^* and will do it very quickly.

The loop: `for ... end`

In order to do Newton's method, we need to repeat the calculation in (3.4) a number of times. This is accomplished in a program using a *loop*, which means a section of a program which is repeated.

The simplest way to accomplish this is to count the number of times through. In MATLAB, a `for ... end` statement makes a loop as in the following simple function program:

```
function S = mysum(n)
% gives the sum of the first n integers
S = 0;              % start at zero
% The loop:
for i = 1:n         % do n times
    S = S + i;      % add the current integer
end                 % end of the loop
```

Call this function in the command window as:
```
 > mysum(100)
```
The result will be the sum of the first 100 integers. All `for ... end` loops have the same format, it begins with `for`, followed by an index (`i`) and a range of numbers (`1:n`). Then come the commands that are to be repeated. Last comes the `end` command.

Loops are one of the main ways that computers are made to do calculations that humans cannot. Any calculation that involves a repeated process is easily done by a loop.

Now let's do a program that does n steps (iterations) of Newton's method. We will need to input the function, its derivative, the initial guess, and the number of steps. The output will be the final value of x, i.e. x_n. If we are only interested in the final approximation, not the intermediate steps, which is usually the case in the real world, then we can use a single variable `x` in the program and change it at each step:

```
function x = mynewton(f,f1,x0,n)
% Solves f(x) = 0 by doing n steps of Newton's method starting at x0.
% f must be a function and f1 must be its derivative.
x = x0;                     % set x equal to the initial guess x0
for i = 1:n                 % Do n times
    x = x - f(x)/f1(x)      % Newton's formula
end
```

In the command window define an inline function: $f(x) = x^3 - 5$ i.e.
```
 > f = inline('x^3 - 5')
```
and define $f1$ to be its derivative, i.e.
```
 > f1 = inline('3*x^2').
```
Then run `mynewton` on this function. Change to `format long`. By trial and error, what is the lowest value of `n` for which the program converges (stops changing). By simple algebra, the true root of this function is $\sqrt[3]{5}$. How close is the program's answer to the true value?

Convergence

Newton's method converges rapidly when $f'(x^*)$ is nonzero and finite, and x_0 is close enough to x^* that the linear approximation (3.2) is valid. Let us take a look at what can go wrong.

For $f(x) = x^{1/3}$ we have $x^* = 0$ but $f'(x^*) = \infty$. If you try
```
 > f = inline('x^(1/3)')
 > f1 = inline('(1/3)*x^(-2/3)')
 > x = mynewton(f,f1,0.1,10)
```
then x explodes.

For $f(x) = x^2$ we have $x^* = 0$ but $f'(x^*) = 0$. If you try

```
 > f = inline('x^2')
 > f1 = inline('2*x')
 > x = mynewton(f,f1,1,10)
```

then x does converge to 0, but not that rapidly.

If x_0 is not close enough to x^* that the linear approximation (3.2) is valid, then the iteration (3.4) gives some x_1 that may or may not be any better than x_0. If we keep iterating, then either

- x_n will eventually get close to x^* and the method will then converge (rapidly), or
- the iterations will not approach x^*.

Exercises

3.1 Enter: `format long`. Use `mynewton` on the function $f(x) = x^5 - 7$, with $x_0 = 2$. By trial and error, what is the lowest value of `n` for which the program converges (stops changing). How close is the answer to the true value? Plug the program's answer into $f(x)$; is the value zero?

3.2 Suppose a ball with a coefficient of elasticity of .9 is dropped from a height of 2 meters onto a hard surface. Write a script program to calculate the distance traveled by the ball after `n` bounces. By trial and error approximate how large `n` must be so that total distance stops changing. Turn in the program and a brief summary of the results.

3.3 For $f(x) = x^3 - 4$, perform 3 iterations of Newton's method with starting point $x_0 = 2$. (On paper, but use a calculator.) Calculate the solution ($x^* = 4^{1/3}$) on a calculator and find the errors and percentage errors of x_0, x_1, x_2 and x_3. Put the results in a table.

Lecture 4

Controlling Error and Conditional Statements

Measuring error

If we are trying to find a numerical solution of an equation $f(x) = 0$, then there are a few different ways we can measure the error of our approximation. The most direct way to measure the error would be as:

$$\{\text{Error at step } n\} = e_n = x_n - x^*$$

where x_n is the n-th approximation and x^* is the true value. However, we usually do not know the value of x^*, or we wouldn't be trying to approximate it. This makes it impossible to know the error directly, and so we must be more clever.

For Newton's method we have the following principle: **At each step the number of significant digits roughly doubles.** While this is an important statement about the error (since it means Newton's method converges really quickly), it is somewhat hard to use in a program.

Rather than measure how close x_n is to x^*, in this and many other situations it is much more practical to measure how close the equation is to being satisfied, in other words, how close $f(x_n)$ is to 0. We will use the quantity $r_n = f(x_n) - 0$, called the *residual*, in many different situations. Most of the time we only care about the size of r_n, so we look at $|r_n| = |f(x_n)|$.

The `if ... end` statement

If we have a certain tolerance for $|r_n| = |f(x_n)|$, then we can incorporate that into our Newton method program using an `if ... end` statement:

```
function x = mynewton(f,f1,x0,n,tol)
% Solves f(x) = 0 by doing n steps of Newton's method starting at x0.
% f must be a function and f1 must be its derivative.
x = x0;                  % set x equal to the initial guess x0
for i = 1:n              % Do n times
    x = x - f(x)./f1(x); % Newton's formula
end
r = abs(f(x));
if r > tol
    warning('The desired accuracy was not attained')
end
```

In this program `if` checks if `abs(y) > tol` is true or not. If it is true then it does everything between there and `end`. If not true, then it skips ahead to `end`.

In the command window define the function

```
> f = inline('x^3-5','x')
```

and its derivative

```
> f1 = inline('3*x^2','x').
```

Then use the program with $n = 5$ and $tol = .01$. Next, change tol to 10^{-10} and repeat.

The loop: `while ... end`

While the previous program will tell us if it worked or not, we still have to input `n`, the number of steps to take. Even for a well-behaved problem, if we make `n` too small then the tolerance will not be attained and we will have to go back and increase it, or, if we make `n` too big, then the program will take more steps than necessary.

One way to control the number of steps taken is to iterate until the residual $|r_n| = |f(x)| = |y|$ is small enough. In MATLAB this is easily accomplished with a `while ... end` loop.

```
function x = mynewtontol(f,f1,x0,tol)
% Solves f(x) = 0 by doing Newton's method starting at x0.
% f must be a function and f1 must be its derivative.
x = x0;                   % set x equal to the initial guess x0
y = f(x);
while abs(y) > tol      % Do until the tolerence is reached.
    x = x - y/f1(x)     % Newton's formula
    y = f(x);
end
```

The statement `while ... end` is a loop, similar to `for ... end`, but instead of going through the loop a fixed number of times it keeps going as long as the statement `abs(y) > tol` is true.

One obvious drawback of the program is that `abs(y)` might never get smaller than `tol`. If this happens, the program would continue to run over and over until we stop it. Try this by setting the tolerance to a really small number:

```
> tol = 10^(-100)
```

then run the program again for $f(x) = x^3 - 5$.

You can use `Ctrl-c` to stop the program when it's stuck.

Exercises

4.1 In Calculus we learn that a geometric series has an exact sum:

$$\sum_{i=0}^{\infty} r^i = \frac{1}{1-r}$$

provided that $|r| < 1$. For instance, if $r = .5$ then the sum is exactly 2. Below is a script program that lacks one line as written. Put in the missing command and then use the program to verify the result above.

How many steps does it take? How close is the answer to 2? Change `r = .5` to `r=.999`. How many iterations does it take? Is the answer accurate?

```
format long
r = .5;
Snew = 0;
Sold = -1;
i = 0;
while Snew > Sold   % is the sum still changing?
   Sold = Snew;
   Snew = Snew + r^i;
   i=i+1;
Snew                % prints the final value.
i                   % prints the # of iterations.
```

Lecture 5

The Bisection Method and Locating Roots

Bisecting and the `if ... else ... end` statement

Recall the bisection method. Suppose that $c = f(a) < 0$ and $d = f(b) > 0$. If f is continuous, then obviously it must be zero at some x^* between a and b. The bisection method then consists of looking half way between a and b for the zero of f, i.e. let $x = (a+b)/2$ and evaluate $y = f(x)$. Unless this is zero, then from the signs of c, d and y we can decide which new interval to subdivide. In particular, if c and y have the same sign, then $[x, b]$ should be the new interval, but if c and y have different signs, then $[a, x]$ should be the new interval. (See Figure 5.1.)

Deciding to do different things in different situations in a program is called *flow control*. The most common way to do this is the `if ... else ... end` statement which is an extension of the `if ... end` statement we have used already.

Bounding the Error

One good thing about the bisection method, that we don't have with Newton's method, is that we always know that the actual solution x^* is inside the current interval $[a, b]$, since $f(a)$ and $f(b)$ have different signs. This allows us to be sure about what the maximum error can be. Precisely, the error is always less than half of the length of the current interval $[a, b]$, i.e.

$$\{\text{Absolute Error}\} = |x - x^*| < (b-a)/2,$$

where x is the center point between the current a and b.

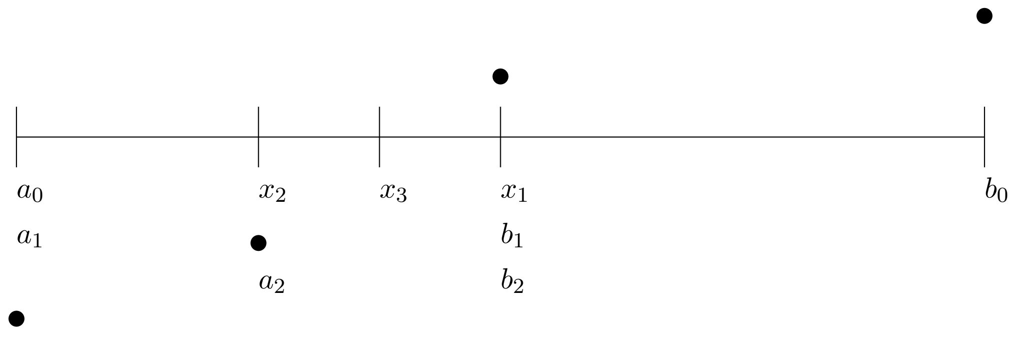

Figure 5.1: The bisection method.

The following function program (also available on the webpage) does n iterations of the bisection method and returns not only the final value, but also the maximum possible error:

```
function [x e] = mybisect(f,a,b,n)
% function [x e] = mybisect(f,a,b,n)
% Does n iterations of the bisection method for a function f
% Inputs: f -- an inline function
%         a,b -- left and right edges of the interval
%         n -- the number of bisections to do.
% Outputs: x -- the estimated solution of f(x) = 0
%          e -- an upper bound on the error
format long
c = f(a); d = f(b);
if c*d > 0.0
    error('Function has same sign at both endpoints.')
end
disp('           x                  y')
for i = 1:n
    x = (a + b)/2;
    y = f(x);
    disp([    x     y])
    if y == 0.0     % solved the equation exactly
        e = 0;
        break      % jumps out of the for loop
    end
    if c*y < 0
        b=x;
    else
        a=x;
    end
end
e = (b-a)/2;
```

Another important aspect of bisection is that it always works. We saw that Newton's method can fail to converge to x^* if x_0 is not close enough to x^*. In contrast, the current interval $[a, b]$ in bisection will always get decreased by a factor of 2 at each step and so it will always eventually shrink down as small as you want it.

Locating a root

The bisection method and Newton's method are both used to obtain closer and closer approximations of a solution, but both require starting places. The bisection method requires two points a and b that have a root between them, and Newton's method requires one point x_0 which is reasonably close to a root. How do you come up with these starting points? It depends. If you are solving an equation once, then the best thing to do first is to just graph it. From an accurate graph you can see approximately where the graph crosses zero.

There are other situations where you are not just solving an equation once, but have to solve the same equation many times, but with different coefficients. This happens often when you are developing software for a specific application. In this situation the first thing you want to take advantage of is the natural domain of the problem, i.e. on what interval is a solution physically reasonable. If that is known, then it is easy to get close to the root by simply checking the sign of the function at a fixed number of points inside the interval. Whenever the sign changes from one point to the next, there is a root between those points. The following program will look for the roots of a function f on a specified interval $[a_0, b_0]$.

```
function [a,b] = myrootfind(f,a0,b0)
% function [a,b] = myrootfind(f,a0,b0)
% Looks for subintervals where the function changes sign
% Inputs: f -- an inline function
%        a0 -- the left edge of the domain
%        b0 -- the right edge of the domain
% Outputs: a -- an array, giving the left edges of subintervals
%              on which f changes sign
%          b -- an array, giving the right edges of the subintervals
n = 1001;   % number of test points to use
a = [];     % start empty array
b = [];
x = linspace(a0,b0,n);
y = f(x);
for i = 1:(n-1)
    if y(i)*y(i+1) < 0  % The sign changed, record it
        a = [a x(i)];
        b = [b x(i+1)];
    end
end
if a == []
    warning('no roots were found')
end
```

The final situation is writing a program that will look for roots with no given information. This is a difficult problem and one that is not often encountered in engineering applications.

Once a root has been located on an interval $[a, b]$, these a and b can serve as the beginning points for the bisection and secant methods (see the next section). For Newton's method one would want to choose x_0 between a and b. One obvious choice would be to let x_0 be the bisector of a and b, i.e. $x_0 = (a + b)/2$. An even better choice would be to use the secant method to choose x_0.

Exercises

5.1 Modify `mybisect` to solve until the error is bounded by a given tolerance. How should error be measured? Run your program on the function $f(x) = 2x^3 + 3x - 1$ with starting interval $[0, 1]$ and a tolerance of 10^{-8}. How many steps does the program use to achieve this tolerance? (You can count the step by adding 1 to a counting variable `i` in the loop of the program.) Turn in your program and a brief summary of the results.

5.2 Perform 3 iterations of the bisection method on the function $f(x) = x^3 - 4$, with starting interval $[1, 3]$. (On paper, but use a calculator.) Calculate the errors of x_1, x_2, and x_3. Compare the errors with those in exercise 3.3.

Lecture 6

Secant Methods*

In this lecture we introduce two additional methods to find numerical solutions of the equation $f(x) = 0$. Both of these methods are based on approximating the function by secant lines just as Newton's method was based on approximating the function by tangent lines.

The Secant Method

The secant method requires two initial points x_0 and x_1 which are both reasonably close to the solution x^*. Preferably the signs of $y_0 = f(x_0)$ and $y_1 = f(x_1)$ should be different. Once x_0 and x_1 are determined the method proceeds by the following formula:

$$x_{i+1} = x_i - \frac{x_i - x_{i-1}}{y_i - y_{i-1}} y_i \tag{6.1}$$

The *Regula Falsi* Method

The *Regula Falsi* method is somewhat a combination of the secant method and bisection method. The idea is to use secant lines to approximate $f(x)$, but choose how to update using the sign of $f(x_n)$, but to update based on the sign at the newest point.

As for the bisection, begin with a and b for which $f(a)$ and $f(b)$ have different signs. Then let:

$$x = b - \frac{b - a}{f(b) - f(a)} f(b).$$

Next check the sign of $f(x)$. If it is the same as the sign of $f(a)$ then x becomes the new a. Otherwise let $b = x$.

Convergence

If we can begin with a good choice x_0, then Newton's method will converge to x^* rapidly. The secant method is a little slower than Newton's method and the *Regula Falsi* method is slightly slower than that. Both however are still much faster than the bisection method.

figure yet to be drawn, alas

Figure 6.1: The secant method.

If we do not have a good starting point or interval, then the secant method, just like Newton's method can fail altogether. The *Regula Falsi* method, just like the bisection method always works because it keeps the solution inside a definite interval.

Simulations and Experiments

Although Newton's method converges faster than any other method, there are contexts when it is not convenient, or even impossible. One obvious situation is when it is difficult to calculate a formula for $f'(x)$ even though one knows the formula for $f(x)$. This is often the case when $f(x)$ is not defined explicitly, but implicitly. There are other situations, which are very common in engineering, where even a formula for $f(x)$ is not known. This happens when $f(x)$ is the result of experiment or simulation rather than a formula. In such situations, the secant method is usually the best choice.

Exercises

6.1 Write a program `mysecant` based on `mybisect`. Use it on $f(x) = x^3 - 4$. Compare the results with those of Exercises 3.3 and 5.3.

Lecture 7

Symbolic Computations

The focus of this course is on numerical computations, i.e. calculations, usually approximations, with floating point numbers. However, Matlab can also do *symbolic* computations which means exact calculations using symbols as in Algebra or Calculus. You should have done some symbolic Matlab computations in your Calculus courses and in this chapter we review what you should already know.

Defining functions and basic operations

Before doing any symbolic computation, one must declare the variables used to be symbolic:

```
 > syms x y
```

A function is defined by simply typing the formula:

```
 > f = cos(x) + 3*x^2
```

Note that coefficients must be multiplied using `*`. To find specific values, you must use the command `subs`:

```
 > subs(f,pi)
```

This command states for *substitute*, it substitutes π for x in the formula for f.

If we define another function:

```
 > g = exp(-y^2)
```

then we can compose the functions:

```
 > h = compose(g,f)
```

i.e. $h(x) = g(f(x))$. Since f and g are functions of different variables, their product must be a function of two variables:

```
 > k = f*g
 > subs(k,[x,y],[0,1])
```

We can do simple calculus operations, like differentiation:

```
 > f1 = diff(f)
```

indefinite integrals (antiderivatives):

```
 > F = int(f)
```

and definite integrals:

```
 > int(f,0,2*pi)
```

To change a symbolic answer into a numerical answer, use the `double` command which stands for *double precision*, (not times 2):

```
 > double(ans)
```

Note that some antiderivatives cannot be found in terms of elementary functions, for some of these it can be expressed in terms of special functions:

```
 > G = int(g)
```

and for others Matlab does the best it can:

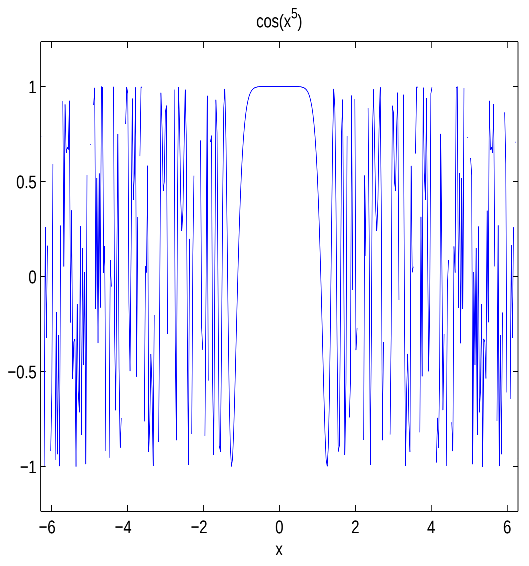

Figure 7.1: Graph of $\cos(x^5)$ produced by the ezplot command. It is wrong because $\cos u$ should oscillate smoothly between -1 and 1. The problem with the plot is that $\cos(x^5)$ oscillates extremely rapidly, and the plot did not consider enough points.

```
> int(h)
```

For definite integrals that cannot be evaluated exactly, Matlab delivers an error:

```
> int(h,0,1)
```

We will see later that even functions that don't have an antiderivative can be integrated numerically. You can change the last answer to a numerical answer using:

```
> double(ans)
```

Plotting a symbolic function can be done as follows:

```
> ezplot(f)
```

or the domain can be specified:

```
> ezplot(g,-10,10)
> ezplot(g,-2,2)
```

To plot a symbolic function of two variables use:

```
> ezsurf(k)
```

It is important to keep in mind that even though we have defined our variables to be symbolic variables, plotting can only plot a finite set of points. For intance:

```
> ezplot(cos(x^5))
```

will produce a plot that is clearly wrong, because it does not plot enough points.

Other useful symbolic operations

Matlab allows you to do simple algebra. For instance:

```
> poly = (x - 3)^5
> polyex = expand(poly)
> polysi = simple(polyex)
```

To find the symbolic solutions of an equation, $f(x) = 0$, use:

```
> solve(f)
> solve(g)
> solve(polyex)
```

Another useful property of symbolic functions is that you can substitute numerical vectors for the variables:

```
> X = 2:0.1:4;
> Y = subs(polyex,X);
> plot(X,Y)
```

Exercises

7.1 Starting from `mynewton` write a program `mysymnewton` that takes as its input a symbolic function $f(x)$, x_0 and n. Let the program take the symbolic derivative $f'(x)$, and then use `subs` to proceed with Newton's method. Test it on $f(x) = x^3 - 4$ starting with $x_0 = 2$. Turn in the program and a brief summary of the results.

Review of Part I

Methods and Formulas

Solving equations numerically:
$f(x) = 0$ — an equation we wish to solve.
x^* — a true solution.
x_0 — starting approximation.
x_n — approximation after n steps.
$e_n = x_n - x^*$ — error of n-th step.
$r_n = y_n = f(x_n)$ — residual at step n. Often $|r_n|$ is sufficient.

Newton's method:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Bisection method:
$f(a)$ and $f(b)$ must have different signs.
$x = (a + b)/2$
Choose $a = x$ or $b = x$, depending on signs.
x^* is always inside $[a, b]$.
$e < (b - a)/2$, current maximum error.

Secant method*:

$$x_{i+1} = x_i - \frac{x_i - x_{i-1}}{y_i - y_{i-1}} y_i$$

Regula Falsi*:

$$x = b - \frac{b - a}{f(b) - f(a)} f(b)$$

Choose $a = x$ or $b = x$, depending on signs.

Convergence:
Bisection is very slow.
Newton is very fast.
Secant methods are intermediate in speed.
Bisection and Regula Falsi never fail to converge.
Newton and Secant can fail if x_0 is not close to x^*.

Locating roots:
Use knowledge of the problem to begin with a reasonable domain.
Systematically search for sign changes of $f(x)$.
Choose x_0 between sign changes using bisection or secant.

Usage:
For Newton's method one must have formulas for $f(x)$ and $f'(x)$.

Secant methods are better for experiments and simulations.

Matlab

Commands:

`> v = [0 1 2 3]`	Make a row vector.
`> u = [0; 1; 2; 3]`	Make a column vector.
`> w = v'`	Transpose: row vector $\leftrightarrow$ column vector
`> x = linspace(0,1,11)`	Make an evenly spaced vector of length 11.
`> x = -1:.1:1`	Make an evenly spaced vector, with increments 0.1.
`> y = x.^2`	Square all entries.
`> plot(x,y)`	plot y vs. x.
`> f = inline('2*x.^2 - 3*x + 1','x')`	Make a function.
`> y = f(x)`	A function can act on a vector.
`> plot(x,y,'*','red')`	A plot with options.
`> Control-c`	Stops a computation.

Program structures:

`for ... end`

Example:

```
for i=1:20
    S = S + i;
end
```

`if ... end`

Example:

```
if y == 0
    disp('An exact solution has been found')
end
```

`while ... end`

Example:

```
while i <= 20
    S = S + i;
    i = i + 1;
end
```

`if ... else ... end`

Example:

```
if c*y>0
    a = x;
else
    b = x;
end
```

Function Programs:

- Begin with the word `function`.
- There are inputs and outputs.
- The outputs, name of the function and the inputs must appear in the first line.
  i.e. `function x = mynewton(f,x0,n)` - The body of the program must assign values to the outputs.

- internal variables are not visible outside the function.

Script Programs:
- There are no inputs and outputs.
- A script program may use and change variables in the current workspace.

Symbolic:

```
> syms x y
> f = 2*x^2 - sqrt(3*x)
> subs(f,sym(pi))
> double(ans)
> g = log(abs(y)) .................................. MATLAB uses log for natural logarithm.
> h(x) = compose(g,f)
> k(x,y) = f*g
> ezplot(f)
> ezplot(g,-10,10)
> ezsurf(k)
```

```
> f1 = diff(f,'x')
> F = int(f,'x') ......................................... indefinite integral (antiderivative)
> int(f,0,2*pi) ........................................................ definite integral
```

```
> poly = x*(x - 3)*(x-2)*(x-1)*(x+1)
> polyex = expand(poly)
> polysi = simple(polyex)
```

```
> solve(f)
> solve(g)
> solve(polyex)
```

Part II
Linear Algebra

Lecture 8

Matrices and Matrix Operations in Matlab

Matrix operations

Recall how to multiply a matrix A times a vector $\mathbf{v}$:

$$A\mathbf{v} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} -1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \cdot (-1) + 2 \cdot 2 \\ 3 \cdot (-1) + 4 \cdot 2 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}.$$

This is a special case of matrix multiplication. To multiply two matrices, A and B you proceed as follows:

$$AB = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} -1 & -2 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} -1+4 & -2+2 \\ -3+8 & -6+4 \end{pmatrix} = \begin{pmatrix} 3 & 0 \\ 5 & -2 \end{pmatrix}.$$

Here both A and B are 2×2 matrices. Matrices can be multiplied together in this way provided that the number of columns of A match the number of rows of B. We always list the size of a matrix by rows, then columns, so a 3×5 matrix would have 3 rows and 5 columns. So, if A is $m \times n$ and B is $p \times q$, then we can multiply AB if and only if $n = p$. A column vector can be thought of as a $p \times 1$ matrix and a row vector as a $1 \times q$ matrix. Unless otherwise specified we will assume a vector $\mathbf{v}$ to be a column vector and so $A\mathbf{v}$ makes sense as long as the number of columns of A matches the number of entries in $\mathbf{v}$.

Enter a matrix into Matlab with the following syntax:
```
 >  A = [ 1  3 -2 5 ;  -1  -1 5 4 ; 0 1 -9  0]
```
Also enter a vector $\mathbf{u}$:
```
 > u = [ 1  2  3  4]'
```
To multiply a matrix times a vector $A\mathbf{u}$ use `*`:
```
 > A*u
```
Since A is 3 by 4 and $\mathbf{u}$ is 4 by 1 this multiplication is valid and the result is a 3 by 1 vector.

Now enter another matrix B using:
```
 > B = [3 2 1; 7 6 5; 4 3 2]
```
You can multiply B times A:
```
 > B*A
```
but A times B is not defined and
```
 > A*B
```
will result in an error message.

You can multiply a matrix by a scalar:
```
 > 2*A
```
Adding matrices $A + A$ will give the same result:
```
 > A + A
```

You can even add a number to a matrix:

```
> A + 3
```
This should add 3 to every entry of A.

Component-wise operations

Just as for vectors, adding a '`.`' before '`*`', '`/`', or '`^`' produces entry-wise multiplication, division and exponentiation. If you enter:

```
> B*B
```
the result will be BB, i.e. matrix multiplication of B times itself. But, if you enter:

```
> B.*B
```
the entries of the resulting matrix will contain the squares of the same entries of B. Similarly if you want B multiplied by itself 3 times then enter:

```
> B^3
```
but, if you want to cube all the entries of B then enter:

```
> B.^3
```
Note that `B*B` and `B^3` only make sense if B is square, but `B.*B` and `B.^3` make sense for any size matrix.

The identity matrix and the inverse of a matrix

The $n \times n$ *identity matrix* is a square matrix with ones on the diagonal and zeros everywhere else. It is called the identity because it plays the same role that 1 plays in multiplication, i.e.

$$AI = A, \qquad IA = A, \qquad I\mathbf{v} = \mathbf{v}$$

for any matrix A or vector $\mathbf{v}$ where the sizes match. An identity matrix in MATLAB is produced by the command:

```
> I = eye(3)
```

A square matrix A can have an *inverse* which is denoted by A^{-1}. The definition of the inverse is that:

$$AA^{-1} = I \qquad \text{and} \qquad A^{-1}A = I.$$

In theory an inverse is very important, because if you have an equation:

$$A\mathbf{x} = \mathbf{b}$$

where A and $\mathbf{b}$ are known and $\mathbf{x}$ is unknown (as we will see, such problems are very common and important) then the theoretical solution is:

$$\mathbf{x} = A^{-1}\mathbf{b}.$$

We will see later that this is not a practical way to solve an equation, and A^{-1} is only important for the purpose of derivations.

In MATLAB we can calculate a matrix's inverse very conveniently:

```
> C = randn(5,5)
> inv(C)
```

However, not all square matrices have inverses:

```
> D = ones(5,5)
> inv(D)
```

The "Norm" of a matrix

For a vector, the "norm" means the same thing as the length. Another way to think of it is how far the vector is from being the zero vector. We want to measure a matrix in much the same way and the *norm* is such a quantity. The usual definition of the norm of a matrix is the following:

Definition 1 *Suppose A is a $m \times n$ matrix. The norm of A is*

$$|A| \equiv \max_{|\mathbf{v}|=1} |A\mathbf{v}|.$$

The maximum in the definition is taken over all vectors with length 1 (unit vectors), so the definition means the largest factor that the matrix stretches (or shrinks) a unit vector. This definition seems cumbersome at first, but it turns out to be the best one. For example, with this definition we have the following inequality for any vector $\mathbf{v}$:

$$|A\mathbf{v}| \leq |A||\mathbf{v}|.$$

In Matlab the norm of a matrix is obtained by the command:

```
> norm(A)
```

For instance the norm of an identity matrix is 1:

```
> norm(eye(100))
```

and the norm of a zero matrix is 0:

```
> norm(zeros(50,50))
```

For a matrix the norm defined above and calculated by Matlab is not the square root of the sum of the square of its entries. That quantity is called the *Froebenius norm*, which is also sometimes useful, but we will not need it.

Some other useful commands

Try out the following:

`> C = rand(5,5)`	random matrix with uniform distribution in $[0, 1]$.
`> size(C)`	gives the dimensions $(m \times n)$ of C.
`> det(C)`	the determinant of the matrix.
`> max(C)`	the maximum of each column.
`> min(C)`	the minimum in each column.
`> sum(C)`	sums each column.
`> mean(C)`	the average of each column.
`> diag(C)`	just the diagonal elements.
`> C'`	tranpose the matrix.

In addition to `ones`, `eye`, `zeros`, `rand` and `randn`, Matlab has several other commands that automatically produce special matrices:

```
> hilb(6)
> pascal(5)
```

Exercises

8.1 Enter the matrix M by

```
>   M = [1,3,-1,6;2,4,0,-1;0,-2,3,-1;-1,2,-5,1]
>   det(M)
>   inv(M)
```

and also the matrix

$$\mathsf{N} = \begin{bmatrix} -1 & -3 & 3 \\ 2 & -1 & 6 \\ 1 & 4 & -1 \\ 2 & -1 & 2 \end{bmatrix}.$$

Multiply M and N using `M * N`. Can the order of multiplication be switched? Why or why not? Try it to see how MATLAB reacts.

8.2 By hand, calculate $A\mathbf{v}$, AB, and BA for:

$$A = \begin{pmatrix} 2 & 4 & -1 \\ -2 & 1 & 9 \\ -1 & -1 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 0 & -1 & -1 \\ 1 & 0 & 2 \\ -1 & -2 & 0 \end{pmatrix} \quad \mathbf{v} = \begin{pmatrix} 3 \\ 1 \\ -1 \end{pmatrix}$$

Check the results using MATLAB. Think about how fast computers are. Turn in your hand work.

8.3 Write a MATLAB function program that makes a $n \times n$ random matrix (normally distributed, `A = randn(n,n)`), calculates its inverse (`B = inv(A)`), multiplies the two back together, and calculates the `error` equal to the norm of the difference of the result from the $n \times n$ identity matrix (`eye(n)`). Starting with `n = 10` increase `n` by factors of 2 until the program stops working, recording the results of each trial. What happens to `error` as `n` gets big? Turn in a printout of the program and a very brief report on the results of your experiments with it, including a plot of error vs. `n` (wouldn't this be a great time for a log plot?).

Lecture 9

Introduction to Linear Systems

How linear systems occur

Linear systems of equations naturally occur in many places in engineering, such as structural analysis, dynamics and electric circuits. Computers have made it possible to quickly and accurately solve larger and larger systems of equations. Not only has this allowed engineers to handle more and more complex problems where linear systems naturally occur, but has also prompted engineers to use linear systems to solve problems where they do not naturally occur such as thermodynamics, internal stress-strain analysis, fluids and chemical processes. It has become standard practice in many areas to analyze a problem by transforming it into a linear systems of equations and then solving those equation by computer. In this way, computers have made linear systems of equations the most frequently used tool in modern engineering.

In Figure 9.1 we show a truss. In statics you learned to write equations for each node of the truss. This set of equations is an example of linear system.

You could solve this system by hand if you had enough time and patience. You would do it by systematically eliminating variables and substituting. Obviously, it would be a lot better to put the equations on a computer and let the computer solve it. In the next few lectures we will learn how to use a computer effectively to solve linear systems. The first key to dealing with linear systems is to realize that they are equivalent to matrices, which contain numbers, not variables.

As we discuss various aspects of matrices, we wish to keep in mind that the matrices that come up in engineering systems are *really large*. It is not unusual in real engineering to use matrices whose dimensions are in the thousands! It is frequently the case that a method that is fine for a 2×2 or 3×3 matrix is totally inappropriate for a 2000×2000 matrix. We thus want to emphasize methods that work for large matrices.

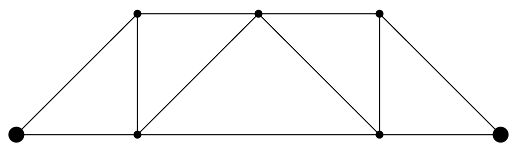

Figure 9.1: A truss.

Linear systems are equivalent to matrix equations

The system of linear equations,

$$\begin{aligned} x_1 - 2x_2 + 3x_3 &= 4 \\ 2x_1 - 5x_2 + 12x_3 &= 15 \\ 2x_2 - 10x_3 &= -10, \end{aligned}$$

is equivalent to the matrix equation,

$$\begin{pmatrix} 1 & -2 & 3 \\ 2 & -5 & 12 \\ 0 & 2 & -10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 15 \\ -10 \end{pmatrix},$$

which is equivalent to the *augmented matrix*,

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 2 & -5 & 12 & 15 \\ 0 & 2 & -10 & -10 \end{array}\right).$$

The advantage of the augmented matrix, is that it contains only numbers, not variables. The reason this is better is because computers are much better in dealing with numbers than variables. To solve this system, the main steps are called *Gaussian elimination* and *back substitution.*

Triangular matrices and back substitution

Consider a linear system whose augmented matrix happens to be:

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 0 & -1 & 6 & 7 \\ 0 & 0 & 2 & 4 \end{array}\right). \tag{9.1}$$

Recall that each row represents an equation and each column a variable. The last row represents the equation $2x_3 = 4$. The equation is easily solved, i.e. $x_3 = 2$. The second row represents the equation $-x_2 + 6x_3 = 7$, but since we know $x_3 = 2$, this simplifies to: $-x_2 + 12 = 7$. This is easily solved, giving $x_2 = 5$. Finally, since we know x_2 and x_3, the first row simplifies to: $x_1 - 10 + 6 = 4$. Thus we have $x_1 = 8$ and so we know the whole solution vector: $\mathbf{x} = \langle 8, 5, 2\rangle$. The process we just did is called *back substitution*, which is both efficient and easily programmed. The property that made it possible to solve the system so easily is that A in this case is *upper triangular.* In the next section we show an efficient way to transform an augmented matrix into an upper triangular matrix.

Gaussian Elimination

Consider the matrix:

$$A = \left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 2 & -5 & 12 & 15 \\ 0 & 2 & -10 & -10 \end{array}\right).$$

The first step of Gaussian elimination is to get rid of the 2 in the (2,1) position by subtracting 2 times the first row from the second row, i.e. (new 2nd = old 2nd - (2) 1st). We can do this because it is essentially the same as adding equations, which is a valid algebraic operation. This leads to:

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 0 & -1 & 6 & 7 \\ 0 & 2 & -10 & -10 \end{array}\right).$$

There is already a zero in the lower left corner, so we don't need to eliminate anything there. To eliminate the third row, second column, we need to subtract -2 times the second row from the third row, (new 3rd = old 3rd - (-2) 2nd):

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

This is now just exactly the matrix in equation (9.1), which we can now solve by back substitution.

Matlab's matrix solve command

In Matlab the standard way to solve a system $A\mathbf{x} = \mathbf{b}$ is by the command:

```
> x = A\b.
```

This command carries out Gaussian elimination and back substitution. We can do the above computations as follows:

```
> A = [1 -2 3 ; 2 -5 12 ; 0 2 -10]
> b = [4 15 -10]'
> x = A\b
```

Next, use the Matlab commands above to solve $A\mathbf{x} = \mathbf{b}$ when the augmented matrix for the system is:

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{array}\right).$$

Check the result by entering:

```
> A*x - b
```

You will see that the resulting answer satisfies the equation exactly. Next try solving using the inverse of A:

```
> x = inv(A)*b
```

This answer can be seen to be inaccurate by checking:

```
> A*x - b
```

Thus we see one of the reasons why the inverse is never used for actual computations, only for theory.

Exercises

9.1 Find the solutions of the following linear systems by hand using the augmented matrix, Gaussian elimination and back substitution. Check your solutions using Matlab.

(a)

$$\begin{aligned} x_1 + x_2 &= 2 \\ 4x_1 + 5x_2 &= 10 \end{aligned}$$

(b)

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= -1 \\ 4x_1 + 7x_2 + 14x_3 &= 3 \\ x_1 + 4x_2 + 4x_3 &= 1 \end{aligned}$$

Lecture 10

Some Facts About Linear Systems

Some inconvenient truths

In the last lecture we learned how to solve a linear system using Matlab. Input the following:

```
 > A = ones(4,4)
 > b = randn(4,1)
 > x = A\b
```

As you will find, there is no solution to the equation $A\mathbf{x} = \mathbf{b}$. This unfortunate circumstance is mostly the fault of the matrix, A, which is too simple, its columns (and rows) are all the same. Now try:

```
 > b = ones(4,1)
 > x = A\b
```

This gives an answer, even though it is accompanied by a warning. So the system $A\mathbf{x} = \mathbf{b}$ does have a solution. Still unfortunately, that is not the only solution. Try:

```
 > x = [ 0 1 0 0]'
 > A*x
```

We see that this x is also a solution. Next try: `> x = [ -4  5  2.27 -2.27]'`

```
 > A*x
```

This x is a solution! It is not hard to see that there are endless possibilities for solutions of this equation.

Basic theory

The most basic theoretical fact about linear systems is:

Theorem 1 *A linear system $A\mathbf{x} = \mathbf{b}$ may have 0, 1, or infinitely many solutions.*

Obviously, in most engineering applications we would want to have exactly one solution. The following two theorems show that having one and only one solution is a property of A.

Theorem 2 *Suppose A is a square ($n \times n$) matrix. The following are all equivalent:*

1. *The equation $A\mathbf{x} = \mathbf{b}$ has exactly one solution for any $\mathbf{b}$.*
2. *$det(A) \neq 0$.*
3. *A has an inverse.*
4. *The only solution of $A\mathbf{x} = \mathbf{0}$ is $\mathbf{x} = \mathbf{0}$.*
5. *The columns of A are linearly independent (as vectors).*
6. *The rows of A are linearly independent.*

If A has these properties then it is called **non-singular**.

On the other hand, a matrix that does not have these properties is called **singular**.

Theorem 3 *Suppose A is a square matrix. The following are all equivalent:*

1. *The equation $A\mathbf{x} = \mathbf{b}$ has 0 or ∞ many solutions depending on $\mathbf{b}$.*
2. $\det(A) = 0$.
3. *A does not have an inverse.*
4. *The equation $A\mathbf{x} = \mathbf{0}$ has solutions other than $\mathbf{x} = \mathbf{0}$.*
5. *The columns of A are linearly dependent as vectors.*
6. *The rows of A are linearly dependent.*

To see how the two theorems work, define two matrices (type in `A1` then scroll up and modify to make `A2`) :

$$\mathtt{A1} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \qquad \mathtt{A2} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{pmatrix},$$

and two vectors:

$$\mathtt{b1} = \begin{pmatrix} 0 \\ 3 \\ 6 \end{pmatrix}, \qquad \mathtt{b2} = \begin{pmatrix} 1 \\ 3 \\ 6 \end{pmatrix}.$$

First calculate the determinants of the matrices:

```
> det(A1)
> det(A2)
```

Then attempt to find the inverses:

```
> inv(A1)
> inv(A2)
```

Which matrix is singular and which is non-singular? Finally, attempt to solve all the possible equations $A\mathbf{x} = \mathbf{b}$:

```
> x = A1\b1
> x = A1\b2
> x = A2\b1
> x = A2\b2
```

As you can see, equations involving the non-singular matrix have one and only one solution, but equation involving a singular matrix are more complicated.

The residual vector

Recall that the residual for an approximate solution x of an equation $f(x) = 0$ is defined as $r = f(x)$. It is a measure of how close the equation is to being satisfied. For a linear system of equations we define the residual of an approximate solution, $\mathbf{x}$ by

$$\mathbf{r} = A\mathbf{x} - \mathbf{b}. \tag{10.1}$$

Notice that $\mathbf{r}$ is a vector. Its size (norm) is an indication of how close we have come to solving $A\mathbf{x} = \mathbf{b}$.

Exercises

10.1 By hand, find all the solutions (if any) of the following linear system using the augmented matrix and Gaussian elimination:

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 4 \\ 4x_1 + 5x_2 + 6x_3 &= 10 \\ 7x_1 + 8x_2 + 9x_3 &= 14, \end{aligned}$$

10.2 Write a function program whose input is a number `n` which makes a random $n \times n$ matrix A and a random vector $\mathbf{b}$, solves the linear system $A\mathbf{x} = \mathbf{b}$ and calculates the norm of the residual $\mathbf{r} = A\mathbf{x} - \mathbf{b}$ and outputs that number as the error e. Make a log-log plot (see `help loglog`) of e vs. n for $n = 5, 10, 50, 100, 500, 1000, \ldots$. Turn in the plot and the program.

Lecture 11

Accuracy, Condition Numbers and Pivoting

In this lecture we will discuss two separate issues of accuracy in solving linear systems. The first, *pivoting*, is a method that ensures that Gaussian elimination proceeds as accurately as possible. The second, *condition number*, is a measure of how bad a matrix is. We will see that if a matrix has a bad condition number, the solutions are unstable with respect to small changes in data.

The effect of rounding

All computers store numbers as finite strings of binary floating point digits. This limits numbers to a fixed number of significant digits and implies that after even the most basic calculations, rounding must happen.

Consider the following exaggerated example. Suppose that our computer can only store 2 significant digits and it is asked to do Gaussian elimination on:

$$\left(\begin{array}{cc|c} .001 & 1 & 3 \\ 1 & 2 & 5 \end{array}\right).$$

Doing the elimination exactly would produce:

$$\left(\begin{array}{cc|c} .001 & 1 & 3 \\ 0 & -998 & -2995 \end{array}\right),$$

but rounding to 2 digits, our computer would store this as:

$$\left(\begin{array}{cc|c} .001 & 1 & 3 \\ 0 & -1000 & -3000 \end{array}\right).$$

Backsolving this reduced system gives:

$$x_1 = 0, \qquad x_2 = 3.$$

This seems fine until you realize that backsolving the unrounded system gives:

$$x_1 = -1, \qquad x_2 = 3.001.$$

Row Pivoting

A way to fix the problem is to use pivoting, which means to switch rows of the matrix. Since switching rows of the augmented matrix just corresponds to switching the order of the equations, no harm is done:

$$\left(\begin{array}{cc|c} 1 & 2 & 5 \\ .001 & 1 & 3 \end{array}\right).$$

Exact elimination would produce:

$$\left(\begin{array}{cc|c} 1 & 2 & 5 \\ 0 & .998 & 2.995 \end{array}\right).$$

Storing this result with only 2 significant digits gives:

$$\left(\begin{array}{cc|c} 1 & 2 & 5 \\ 0 & 1 & 3 \end{array}\right).$$

Now backsolving produces:

$$x_1 = -1, \qquad x_2 = 3,$$

which is exactly right.

The reason this worked is because 1 is bigger than 0.001. To pivot we switch rows so that the largest entry in a column is the one used to eliminate the others. In bigger matrices, after each column is completed, compare the diagonal element of the next column with all the entries below it. Switch it (and the entire row) with the one with greatest absolute value. For example in the following matrix, the first column is finished and before doing the second column, pivoting should occur since $|-2| > |1|$:

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 0 & 1 & 6 & 7 \\ 0 & -2 & -10 & -10 \end{array}\right).$$

Pivoting the 2nd and 3rd rows would produce:

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 4 \\ 0 & -2 & -10 & -10 \\ 0 & 1 & 6 & 7 \end{array}\right).$$

Condition number

In some systems, problems occur even without rounding. Consider the following augmented matrices:

$$\left(\begin{array}{cc|c} 1 & 1/2 & 3/2 \\ 1/2 & 1/3 & 1 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{cc|c} 1 & 1/2 & 3/2 \\ 1/2 & 1/3 & 5/6 \end{array}\right).$$

Here we have the same A, but two different input vectors:

$$\mathbf{b}_1 = (3/2, 1)' \quad \text{and} \quad \mathbf{b}_2 = (3/2, 5/6)'$$

which are pretty close to one another. You would expect then that the solutions $\mathbf{x}_1$ and $\mathbf{x}_2$ would also be close. Notice that this matrix does not need pivoting. Eliminating exactly we get:

$$\left(\begin{array}{cc|c} 1 & 1/2 & 3/2 \\ 0 & 1/12 & 1/4 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{cc|c} 1 & 1/2 & 3/2 \\ 0 & 1/12 & 1/12 \end{array}\right).$$

Now solving we find:

$$\mathbf{x}_1 = (0, 3)' \quad \text{and} \quad \mathbf{x}_2 = (1, 1)'$$

which are *not close at all* despite the fact that we did the calculations exactly. This poses a new problem: some matrices are very sensitive to small changes in input data. The extent of this sensitivity is measured by the **condition number**. The definition of condition number is: consider all small changes δA and $\delta\mathbf{b}$ in A and $\mathbf{b}$ and the resulting change, $\delta\mathbf{x}$, in the solution $\mathbf{x}$. Then:

$$\mathrm{cond}(A) \equiv \max\left(\frac{|\delta\mathbf{x}|/|\mathbf{x}|}{\frac{|\delta A|}{|A|}+\frac{|\delta\mathbf{b}|}{|\mathbf{b}|}}\right) = \max\left(\frac{\text{Relative error of output}}{\text{Relative error of inputs}}\right).$$

Put another way, changes in the input data get multiplied by the condition number to produce changes in the outputs. Thus a high condition number is bad. It implies that small errors in the input can cause large errors in the output.

In MATLAB enter:
```
> H = hilb(2)
```
which should result in the matrix above. MATLAB produces the condition number of a matrix with the command:
```
> cond(H)
```
Thus for this matrix small errors in the input can get magnified by 19 in the output. Next try the matrix:
```
> A = [ 1.2969 0.8648 ; .2161  .1441]
> cond(A)
```
For this matrix small errors in the input can get magnified by 2.5×10^8 in the output! This is obviously not very good for engineering where all measurements, constants and inputs are approximate.

Is there a solution to the problem of bad condition numbers? Usually, bad condition numbers in engineering contexts result from poor design. So, the engineering solution to bad conditioning is **redesign**.

Finally, find the determinant of the matrix A above:
```
> det(A)
```
which will be small. If $det(A) = 0$ then the matrix is singular, which is bad because it implies there will not be a unique solution. The case here, $\det(A) \approx 0$, is also bad, because it means the matrix is almost singular. Although $\det(A) \approx 0$ generally indicates that the condition number will be large, they are actually independent things. To see this, find the determinant and condition number of the matrix `[1e-10,0;0,1e-10]` and the matrix `[1e+10,0;0,1e-10]`.

Exercises

11.1 Find the determinant and inverse of:

$$\mathsf{A} = \begin{bmatrix} 1.2969 & .8648 \\ .2161 & .1441 \end{bmatrix}.$$

Let B be the matrix obtained from A by rounding off to three decimal places ($1.2969 \mapsto 1.297$). Find the determinant and inverse of B. How do A^{-1} and B^{-1} differ? Explain how this happened.

Set `b1 = [1.2969; 0.2161]` and do `x = A\b1` . Repeat the process but with a vector `b2` obtained from `b1` by rounding off to three decimal places. Explain exactly what happened. Why was the first answer so simple? Why do the two answers differ by so much?

11.2 Try

```
> B = [sin(sym(1)) sin(sym(2)); sin(sym(3)) sin(sym(4))]
> c = [1; 2]
> x = B\c
> pretty(x)
```

Next input the matrix:

$$\mathsf{Cs} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

symbolically as above. Create a numerical version via `Cn = double(Cs)` and define the two vectors `d1 = [4; 8]` and `d2 = [1; 1]`. Solve the systems `Cs*x = d1`, `Cn*x = d1`, `Cs*x = d2`, and `Cn*x = d2`. Explain the results. Does the symbolic or non-symbolic way give more information?

Lecture 12

LU Factorization

In many applications where linear systems appear, one needs to solve $A\mathbf{x} = \mathbf{b}$ for many different vectors $\mathbf{b}$. For instance, a structure must be tested under several different loads, not just one. As in the example of a truss, the loading in such a problem is usually represented by the vector $\mathbf{b}$. Gaussian elimination with pivoting is the most efficient and accurate way to solve a linear system. Most of the work in this method is spent on the matrix A itself. If we need to solve several different systems with the same A, and A is big, then we would like to avoid repeating the steps of Gaussian elimination on A for every different $\mathbf{b}$. This can be accomplished by the *LU decomposition*, which in effect records the steps of Gaussian elimination.

LU decomposition

The main idea of the LU decomposition is to record the steps used in Gaussian elimination on A in the places where the zero is produced. Consider the matrix:

$$A = \begin{pmatrix} 1 & -2 & 3 \\ 2 & -5 & 12 \\ 0 & 2 & -10 \end{pmatrix}.$$

The first step of Gaussian elimination is to subtract 2 times the first row from the second row. In order to record what we have done, we will put the multiplier, 2, into the place it was used to make a zero, i.e. the second row, first column. In order to make it clear that it is a record of the step and not an element of A, we will put it in parentheses. This leads to:

$$\begin{pmatrix} 1 & -2 & 3 \\ (2) & -1 & 6 \\ 0 & 2 & -10 \end{pmatrix}.$$

There is already a zero in the lower left corner, so we don't need to eliminate anything there. We record this fact with a (0). To eliminate the third row, second column, we need to subtract -2 times the second row from the third row. Recording the -2 in the spot it was used we have:

$$\begin{pmatrix} 1 & -2 & 3 \\ (2) & -1 & 6 \\ (0) & (-2) & 2 \end{pmatrix}.$$

Let U be the upper triangular matrix produced, and let L be the lower triangular matrix with the records and ones on the diagonal, i.e.:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix},$$

then we have the following mysterious coincidence:

$$LU = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix} \begin{pmatrix} 1 & -2 & 3 \\ 0 & -1 & 6 \\ 0 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & -2 & 3 \\ 2 & -5 & 12 \\ 0 & 2 & -10 \end{pmatrix} = A.$$

Thus we see that A is actually the product of L and U. Here L is lower triangular and U is upper triangular. When a matrix can be written as a product of simpler matrices, we call that a *decomposition* of A and this one we call the LU decomposition.

Using LU to solve equations

If we also include pivoting, then an LU decomposition for A consists of three matrices P, L and U such that:

$$PA = LU. \tag{12.1}$$

The pivot matrix P is the identity matrix, with the same rows switched as the rows of A are switched in the pivoting. For instance:

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

would be the pivot matrix if the second and third rows of A are switched by pivoting. MATLAB will produce an LU decomposition with pivoting for a matrix A with the following command:

```
> [L U P] = lu(A)
```

where `P` is the pivot matrix. To use this information to solve $A\mathbf{x} = \mathbf{b}$ we first pivot both sides by multiplying by the pivot matrix:

$$PA\mathbf{x} = P\mathbf{b} \equiv \mathbf{b}'.$$

Substituting LU for PA we get:

$$LU\mathbf{x} = \mathbf{b}'.$$

Then we need only to solve two back substitution problems:

$$L\mathbf{y} = \mathbf{b}'$$

and

$$U\mathbf{x} = \mathbf{y}.$$

In MATLAB this would work as follows:

```
> A = rand(5,5)
> [L U P] = lu(A)
> b = rand(5,1)
> bp = P*b
> y = L\bp
> x = U\y
> rnorm = norm(A*x - b)     ............................................. Check the result.
```

We can then solve for any other $\mathbf{b}$ without redoing the LU step. Repeat the sequence for a new right hand side: `c = randn(5,1)`; you can start at the third line. While this may not seem like a big savings, it would be if A were a large matrix from an actual application.

Exercises

12.1 Solve the systems below by hand using the LU decomposition. Pivot if appropriate. In each of the two problems, check by hand that $LU = PA$ and $A\mathbf{x} = \mathbf{b}$.

(a) $A = \begin{pmatrix} 1 & 4 \\ 3 & 5 \end{pmatrix}$, $\qquad \mathbf{b} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$

(b) $A = \begin{pmatrix} 2 & 4 \\ .5 & 4 \end{pmatrix}$, $\qquad \mathbf{b} = \begin{pmatrix} 0 \\ -3 \end{pmatrix}$

12.2 Write a Matlab function program that solves linear systems using the above LU decomposition with pivoting and also with the built-in solve function (`A\b`). Make the first line be:

```
function [x1, r1, x2, r2] = mysolve(A,b)
```

where `A` is the input matrix and `b` is the right-hand vector. The outputs `x1` and `r1` should be the solution and norm of the residual for the LU method and `x2` and `r2` should be the solution and norm of the residual for the built-in method. Let the second line of the program be:

```
 format long.
```

Test the program on both random matrices (`randn(n,n)`) and Hilbert matrices (`hilb(n)`) with n really big and document the results.

Lecture 13

Nonlinear Systems - Newton's Method

An Example

The LORAN (LOng RAnge Navigation) system calculates the position of a boat at sea using signals from fixed transmitters. From the time differences of the incoming signals, the boat obtains differences of distances to the transmitters. This leads to two equations each representing hyperbolas defined by the differences of distance of two points (foci). An example of such equations from ([2]) are:

$$\frac{x^2}{186^2} - \frac{y^2}{300^2 - 186^2} = 1$$

and

$$\frac{(y-500)^2}{279^2} - \frac{(x-300)^2}{500^2 - 279^2} = 1.$$

Solving two quadratic equations with two unknowns, would require solving a 4 degree polynomial equation. We could do this by hand, but for a navigational system to work well, it must do the calculations automatically and numerically. We note that the Global Positioning System (GPS) works on similar principles and must do similar computations.

Vector Notation

In general, we can usually find solutions to a system of equations when the number of unknowns matches the number of equations. Thus, we wish to find solutions to systems that have the form:

$$\begin{aligned} f_1(x_1, x_2, x_3, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, x_3, \ldots, x_n) &= 0 \\ f_3(x_1, x_2, x_3, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, x_3, \ldots, x_n) &= 0. \end{aligned} \tag{13.1}$$

For convenience we can think of $(x_1, x_2, x_3, \ldots, x_n)$ as a vector $\mathbf{x}$ and $(f_1, f_2, \ldots, f_n)$ as a vector-valued function $\mathbf{f}$. With this notation, we can write the system of equations (13.1) simply as:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0},$$

i.e. we wish to find a vector that makes the vector function equal to the zero vector.

As in Newton's method for one variable, we need to start with an initial guess $\mathbf{x}_0$. In theory, the more variables one has, the harder it is to find a good initial guess. In practice, this must be overcome by using physically reasonable assumptions about the possible values of a solution, i.e. take advantage of engineering knowledge of the problem. Once $\mathbf{x}_0$ is chosen, let

$$\Delta\mathbf{x} = \mathbf{x}_1 - \mathbf{x}_0.$$

Linear Approximation for Vector Functions

In the single variable case, Newton's method was derived by considering the linear approximation of the function f at the initial guess $\mathbf{x}_0$. From Calculus, the following is the linear approximation of $\mathbf{f}$ at $\mathbf{x}_0$, for vectors and vector-valued functions:

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + D\mathbf{f}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0).$$

Here $D\mathbf{f}(\mathbf{x}_0)$ is an $n \times n$ matrix whose entries are the various partial derivative of the components of $\mathbf{f}$. Specifically:

$$D\mathbf{f}(\mathbf{x}_0) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}_0) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}_0) & \frac{\partial f_1}{\partial x_3}(\mathbf{x}_0) & \dots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}_0) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}_0) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}_0) & \frac{\partial f_2}{\partial x_3}(\mathbf{x}_0) & \dots & \frac{\partial f_2}{\partial x_n}(\mathbf{x}_0) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}(\mathbf{x}_0) & \frac{\partial f_n}{\partial x_2}(\mathbf{x}_0) & \frac{\partial f_n}{\partial x_3}(\mathbf{x}_0) & \dots & \frac{\partial f_n}{\partial x_n}(\mathbf{x}_0) \end{pmatrix}.$$

Newton's Method

We wish to find $\mathbf{x}$ that makes $\mathbf{f}$ equal to the zero vectors, so let's choose $\mathbf{x}_1$ so that

$$\mathbf{f}(\mathbf{x}_0) + D\mathbf{f}(\mathbf{x}_0)(\mathbf{x}_1 - \mathbf{x}_0) = \mathbf{0}.$$

Since $D\mathbf{f}(\mathbf{x}_0)$ is a square matrix, we can solve this equation by

$$\mathbf{x}_1 = \mathbf{x}_0 - (D\mathbf{f}(\mathbf{x}_0))^{-1}\mathbf{f}(\mathbf{x}_0),$$

provided that the inverse exists. The formula is the vector equivalent of the Newton's method formula we learned before. However, in practice we never use the inverse of a matrix for computations, so we cannot use this formula directly. Rather, we can do the following. First solve the equation

$$D\mathbf{f}(\mathbf{x}_0)\Delta\mathbf{x} = -\mathbf{f}(\mathbf{x}_0). \tag{13.2}$$

Since $D\mathbf{f}(\mathbf{x}_0)$ is a known matrix and $-\mathbf{f}(\mathbf{x}_0)$ is a known vector, this equation is just a system of linear equations, which can be solved efficiently and accurately. Once we have the solution vector $\Delta\mathbf{x}$, we can obtain our improved estimate $\mathbf{x}_1$ by:

$$\mathbf{x}_1 = \mathbf{x}_0 + \Delta\mathbf{x}.$$

For subsequent steps, we have the following process:

- Solve $D\mathbf{f}(\mathbf{x}_i)\Delta\mathbf{x} = -\mathbf{f}(\mathbf{x}_i)$ for $\Delta\mathbf{x}$.
- Let $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}$

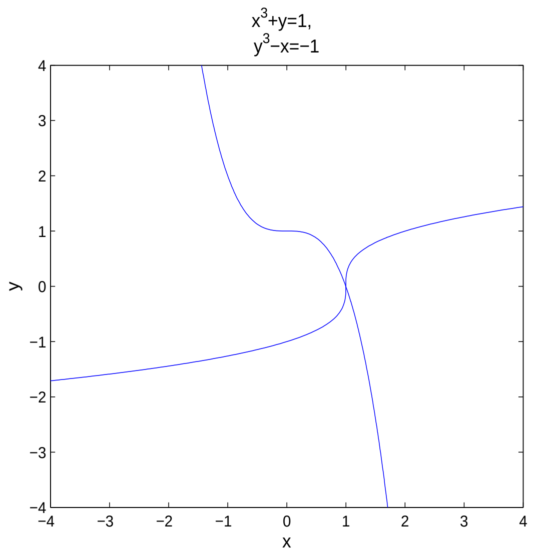

Figure 13.1: Graphs of the equations $x^3 + y = 1$ and $y^3 - x = -1$. There is one and only one intersection; at $(x, y) = (1, 0)$.

An Experiment

We will solve the following set of equations:

$$\begin{aligned} x^3 + y &= 1 \\ y^3 - x &= -1. \end{aligned} \tag{13.3}$$

You can easily check that $(x, y) = (1, 0)$ is a solution of this system. By graphing both of the equations you can also see that $(1, 0)$ is the only solution (Figure 13.1).

We can put these equations into vector form by letting $x_1 = x$, $x_2 = y$ and

$$\begin{aligned} f_1(x_1, x_2) &= x_1^3 + x_2 - 1 \\ f_2(x_1, x_2) &= x_2^3 - x_1 + 1. \end{aligned} \tag{13.4}$$

or

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} x_1^3 + x_2 - 1 \\ x_2^3 - x_1 + 1 \end{pmatrix}. \tag{13.5}$$

Now that we have the equation in the vector form, write the following script program:

```
format long
f = inline('[ x(1)^3+x(2)-1 ; x(2)^3-x(1)+1 ]');
x = [.5;.5]
x = fsolve(f,x)
```

Save this program as `myfsolve.m` and run it. You will see that the internal MATLAB solving command `fsolve` approximates the solution, but only to about 7 demimal places. While that would be close enough for most applications, one would expect that we could do better on such a simple problem.

Next we will implement Newton's method for this problem. Modify your `myfsolve` program to:

```
%mymultnewton
format long
n=8  % set the number of iterations
f = inline('[x(1)^3+x(2)-1 ; x(2)^3-x(1)+1]');
Df = inline('[3*x(1)^2,  1 ; -1,  3*x(2)^2]');
x = [.5;.5]
for i = 1:n
      Dx = -Df(x)\f(x);
      x = x + Dx
      f(x)  % see if f(x) is really zero
end
```

Save and run this program (as `mymultnewton`) and you will see that it finds the root exactly (to machine precision) in only 6 iterations. Why is this simple program able to do better than Matlab's built-in program?

Exercises

13.1 Adapt the `mymultnewton` program to find a solution of the pair of equations in the LORAN example in the lecture. These equations actually have 4 different solutions! By trying different starting vectors, see how many you can find. Think of at least one way that the navigational system could determine which is correct.

Lecture 14

Eigenvalues and Eigenvectors

Suppose that A is a square $(n \times n)$ matrix. We say that a nonzero vector $\mathbf{v}$ is an eigenvector (**ev**) and a number λ is its eigenvalue (**ew**) if

$$A\mathbf{v} = \lambda\mathbf{v}. \tag{14.1}$$

Geometrically this means that $A\mathbf{v}$ is in the same direction as $\mathbf{v}$, since multiplying a vector by a number changes its length, but not its direction.

Matlab has a built-in routine for finding eigenvalues and eigenvectors:

```
 > A = pascal(4,4)
 > [v e] = eig(A)
```

The results are a matrix `v` that contains eigenvectors as columns and a diagonal matrix `e` that contains eigenvalues on the diagonal. We can check this by:

```
 > v1 = v(:,1)
 > A*v1
 > e(1,1)*v1
```

Finding Eigenvalues for 2×2 and 3×3

If A is 2×2 or 3×3 then we can find its eigenvalues and eigenvectors by hand. Notice that Equation (14.1) can be rewritten as:

$$A\mathbf{v} - \lambda\mathbf{v} = \mathbf{0}. \tag{14.2}$$

It would be nice to factor out the $\mathbf{v}$ from the right-hand side of this equation, but we can't because A is a matrix and λ is a number. However, since $I\mathbf{v} = \mathbf{v}$, we can do the following:

$$\begin{aligned} A\mathbf{v} - \lambda\mathbf{v} &= A\mathbf{v} - \lambda I\mathbf{v} \\ &= (A - \lambda I)\mathbf{v} \\ &= \mathbf{0} \end{aligned} \tag{14.3}$$

If $\mathbf{v}$ is nonzero, then by Theorem 3 the matrix $(A - \lambda I)$ must be singular. By the same theorem, we must have:

$$\det(A - \lambda I) = 0. \tag{14.4}$$

This is called the *characteristic equation.*

For a 2×2 matrix, $A - \lambda I$ is calculated as in the following example:

$$\begin{aligned} A - \lambda I &= \begin{pmatrix} 1 & 4 \\ 3 & 5 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 4 \\ 3 & 5 \end{pmatrix} - \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix} \\ &= \begin{pmatrix} 1-\lambda & 4 \\ 3 & 5-\lambda \end{pmatrix}. \end{aligned} \tag{14.5}$$

The determinant of $A - \lambda I$ is then

$$\begin{aligned} \det(A - \lambda I) &= (1-\lambda)(5-\lambda) - 4 \cdot 3 \\ &= -7 - 6\lambda + \lambda^2. \end{aligned} \tag{14.6}$$

The characteristic equation $\det(A - \lambda I) = 0$ is simply a quadratic equation:

$$\lambda^2 - 6\lambda - 7 = 0. \tag{14.7}$$

The roots of this equation are $\lambda_1 = 7$ and $\lambda_2 = -1$. These are the **ew**'s of the matrix A. Now to find the corresponding **ev**'s we return to the equation $(A - \lambda I)\mathbf{v} = \mathbf{0}$. For $\lambda_1 = 7$, the equation for the **ev** $(A - \lambda I)\mathbf{v} = \mathbf{0}$ is equivalent to the augmented matrix

$$\left(\begin{array}{cc|c} -6 & 4 & 0 \\ 3 & -2 & 0 \end{array}\right). \tag{14.8}$$

Notice that the first and second rows of this matrix are multiples of one another. Thus Gaussian elimination would produce all zeros on the bottom row. Thus this equation has infinitely many solutions, i.e. infinitely many **ev**'s. Since only the direction of the **ev** matters, this is okay, we only need to find one of the **ev**'s. Since the second row of the augmented matrix represents the equation

$$3x - 2y = 0,$$

we can let

$$\mathbf{v}_1 = \begin{pmatrix} 2 \\ 3 \end{pmatrix}.$$

This comes from noticing that $(x, y) = (2, 3)$ is a solution of $3x - 2y = 0$.

For $\lambda_2 = -1$, $(A - \lambda I)\mathbf{v} = \mathbf{0}$ is equivalent to the augmented matrix:

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

Once again the first and second rows of this matrix are multiples of one another. For simplicity we can let

$$\mathbf{v}_2 = \begin{pmatrix} -2 \\ 1 \end{pmatrix}.$$

One can always check an **ev** and **ew** by multiplying:

$$\begin{aligned} A\mathbf{v}_1 &= \begin{pmatrix} 1 & 4 \\ 3 & 5 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 14 \\ 21 \end{pmatrix} = 7 \begin{pmatrix} 2 \\ 3 \end{pmatrix} = 7\mathbf{v}_1 \quad \text{and} \\ A\mathbf{v}_2 &= \begin{pmatrix} 1 & 4 \\ 3 & 5 \end{pmatrix} \begin{pmatrix} -2 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \end{pmatrix} = -1 \begin{pmatrix} -2 \\ 1 \end{pmatrix} = -1\mathbf{v}_2\,. \end{aligned}$$

For a 3×3 matrix we could complete the same process. The $\det(A - \lambda I) = 0$ would be a cubic polynomial and we would expect to usually get 3 roots, which are the **ew**'s.

Larger Matrices

For a $n \times n$ matrix with $n \geq 4$ this process is too long and cumbersome to complete by hand. Further, this process is not well suited even to implementation on a computer program since it involves determinants and solving a n-degree polynomial. For $n \geq 4$ we need more ingenious methods. These methods rely on the geometric meaning of **ev**'s and **ew**'s rather than solving algebraic equations. We will overview these methods in Lecture 16.

Complex Eigenvalues

It turns out that the eigenvalues of some matrices are complex numbers, even when the matrix only contains real numbers. When this happens the complex **ew**'s must occur in conjugate pairs, i.e.:

$$\lambda_{1,2} = \alpha \pm i\beta.$$

The corresponding **ev**'s must also come in conjugate pairs:

$$\mathbf{w} = \mathbf{u} \pm i\mathbf{v}.$$

In applications, the imaginary part of the **ew**, β, often is related to the frequency of an oscillation. This is because of Euler's formula

$$e^{\alpha+i\beta} = e^{\alpha}(\cos\beta + i\sin\beta).$$

Certain kinds of matrices that arise in applications can only have real **ew**'s and **ev**'s. The most common such type of matrix is the symmetric matrix. A matrix is symmetric if it is equal to its own transpose, i.e. it is symmetric across the diagonal. For example,

$$\begin{pmatrix} 1 & 3 \\ 3 & -5 \end{pmatrix}$$

is symmetric and so we know beforehand that it's **ew**'s will be real, not complex.

Exercises

14.1 Find the eigenvalues and eigenvectors of the following matrix by hand:

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

14.2 Find the eigenvalues and eigenvectors of the following matrix by hand:

$$B = \begin{pmatrix} 1 & -2 \\ 2 & 1 \end{pmatrix}.$$

Can you guess the **ew**'s of the matrix

$$C = \begin{pmatrix} a & -b \\ b & a \end{pmatrix}?$$

Lecture 15

An Application of Eigenvectors: Vibrational Modes

One application of **ew**'s and **ev**'s is in the analysis of vibration problems. A simple nontrivial vibration problem is the motion of two objects with equal masses m attached to each other and fixed outer walls by equal springs with spring constants k, as shown in Figure 15.1.

Let x_1 denote the displacement of the first mass and x_2 the displacement of the second, and note the displacement of the walls is zero. Each mass experiences forces from the adjacent springs proportional to the stretch or compression of the spring. Ignoring any friction, Newton's law of motion $ma = F$, leads to:

$$\begin{array}{llllll} m\ddot{x}_1 & = -k(x_1 - 0) & +k(x_2 - x_1) & & = -2kx_1 + kx_2 \\ m\ddot{x}_2 & = & -k(x_2 - x_1) & +k(0 - x_2) & = kx_1 - 2kx_2 \end{array} . \tag{15.1}$$

Dividing both sides by m we can write these equations in matrix form:

$$\ddot{\mathbf{x}} = -A\mathbf{x}, \tag{15.2}$$

where

$$A = \frac{k}{m}B = \frac{k}{m}\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}. \tag{15.3}$$

For this type of equation, the general solution is:

$$\mathbf{x}(t) = c_1\mathbf{v}_1 \sin\left(\sqrt{\frac{k}{m}\lambda_1}\; t + \phi_1\right) + c_2\mathbf{v}_2 \sin\left(\sqrt{\frac{k}{m}\lambda_2}\; t + \phi_2\right) \tag{15.4}$$

where λ_1 and λ_2 are **ew**'s of B with corresponding **ev**'s $\mathbf{v}_1$ and $\mathbf{v}_2$. We can interpret the **ew**'s as the squares of the frequencies of oscillation. We can find the **ew**'s and **ev**'s of B using Matlab:

```
 > B = [2 -1 ; -1 2]
 > [v e] = eig(B)
```

This should produce a matrix `v` whose columns are the **ev**'s of B and a diagonal matrix e whose entries are the **ew**'s of B. In the first eigenvector, $\mathbf{v}_1$, the two entries are equal. This represents the mode of oscillation where the two masses move in sync with each other. The second **ev**, $\mathbf{v}_2$,

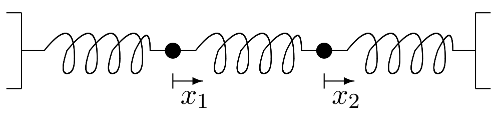

Figure 15.1: Two equal masses attached to each other and fixed walls by equal springs.

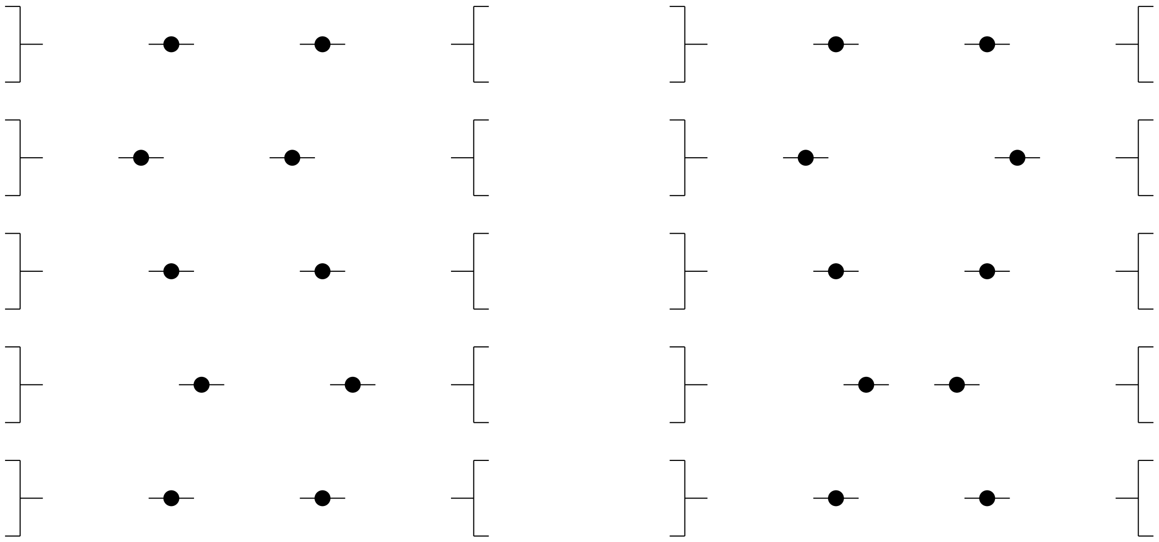

Figure 15.2: Two vibrational modes of a simple oscillating system. In the left mode the weights move together and in the right mode they move opposite. Note that the two modes actually move at different speeds.

has the same entries but opposite signs. This represents the mode where the two masses oscillate in anti-synchronization. Notice that the frequency for anti-sync motion is 3 times that of synchronous motion.

Which of the two modes is the most dangerous for a structure or machine? It is the one with the *lowest frequency* because that mode can have the largest displacement. Sometimes this mode is called the *fundamental mode*.

To get the frequencies for the matrix $A = k/mB$, notice that if $\mathbf{v}_i$ is one of the **ev**'s for B then:

$$A\mathbf{v}_i = \frac{k}{m}B\mathbf{v}_i = \frac{k}{m}\lambda_i\mathbf{v}_i.$$

Thus we can conclude that A has the same **ev**'s as B, but the **ew**'s are multiplied by the factor k/m. Thus the two frequencies are:

$$\frac{k}{m} \quad \text{and} \quad \frac{3k}{m}.$$

We can do the same for three equal masses. The corresponding matrix B would be:

$$B = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}.$$

Find the **ev**'s and **ew**'s as above. There are three different modes. Interpret them from the **ev**'s.

Exercises

15.1 Find the frequencies and modes for 4 equal masses with equal springs.

15.2 Find the frequencies and modes for non-identical masses with equal springs in the following cases. How does this effect the modes?

(a) Two masses with $m_1 = 1$ and $m_2 = 2$.

(b) Three masses with $m_1 = 1$, $m_2 = 2$ and $m_3 = 3$.

Lecture 16

Numerical Methods for Eigenvalues

As mentioned above, the **ew**'s and **ev**'s of an $n \times n$ matrix where $n \geq 4$ must be found numerically instead of by hand. The numerical methods that are used in practice depend on the geometric meaning of **ew**'s and **ev**'s which is equation (14.1). The essence of all these methods is captured in the Power method, which we now introduce.

The Power Method

In the command window of MATLAB enter the following:

```
> A = hilb(5)
> x = ones(5,1)
> x = A*x
> el = max(x)
> x = x/el
```

Compare the new value of `x` with the original. Repeat the last three lines (you can use the scroll up button). Compare the newest value of `x` with the previous one and the original. Notice that there is less change between the second two. Repeat the last three commands over and over until the values stop changing. You have completed what is known as the *Power Method*. Now try the command:

```
> [v e] = eig(A)
```

The last entry in `e` should be the final `el` we computed. The last column in `v` is `x/norm(x)`. Below we explain why our commands gave this eigenvalue and eigenvector.

For illustration consider a 2×2 matrix whose **ew**'s are $1/3$ and 2 and whose corresponding **ev**'s are $\mathbf{v}_1$ and $\mathbf{v}_2$. Let $\mathbf{x}_0$ be any vector which is a combination of $\mathbf{v}_1$ and $\mathbf{v}_2$, e.g.,

$$\mathbf{x}_0 = \mathbf{v}_1 + \mathbf{v}_2.$$

Now let $\mathbf{x}_1$ be A times $\mathbf{x}_0$. It follows from (14.1) that

$$\begin{aligned} \mathbf{x}_1 &= A\mathbf{v}_1 + A\mathbf{v}_2 \\ &= \frac{1}{3}\mathbf{v}_1 + 2\mathbf{v}_2. \end{aligned} \tag{16.1}$$

Thus the $\mathbf{v}_1$ part is shrunk while the $\mathbf{v}_2$ is stretched. If we repeat this process k times then:

$$\begin{aligned} \mathbf{x}_k &= A\mathbf{x}_{k-1} \\ &= A^k\mathbf{x}_0 \\ &= \left(\frac{1}{3}\right)^k \mathbf{v}_1 + 2^k\mathbf{v}_2. \end{aligned} \tag{16.2}$$

Clearly, $\mathbf{x}_k$ grows in the direction of $\mathbf{v}_2$ and shrinks in the direction of $\mathbf{v}_1$. This is the principle of the Power Method, vectors multiplied by A are stretched most in the direction of the **ev** whose **ew** has the largest absolute value.

The **ew** with the largest absolute value is called the *dominant* **ew**. In many applications this quantity will necessarily be positive for physical reasons. When this is the case, the MATLAB code above will work since `max(v)` will be the element with the largest absolute value. In applications where the dominant **ew** may be negative, the program must use flow control to determine the correct number.

Summarizing the Power Method:
- Repeatedly multiply **x** by A and divide by the element with the largest absolute value.
- The element of largest absolute value converges to largest absolute **ew**.
- The vector converges to the corresponding **ev**.

Note that this logic only works when the eigenvalue largest in magnitude is real. If the matrix and starting vector are real then the power method can never give a result with an imaginary part. Eigenvalues with imaginary part mean the matrix has a rotational component, so the eigenvector would not settle down either.

The Inverse Power Method

In the application of vibration analysis, the mode (**ev**) with the lowest frequency (**ew**) is the most dangerous for the machine or structure. The Power Method gives us instead the largest **ew**, which is the least important frequency. In this section we introduce a method, the *Inverse Power Method* which produces exactly what is needed.

The following facts are at the heart of the Inverse Power Method:
- *If λ is an **ew** of A then $1/\lambda$ is an **ew** for A^{-1}.*
- *The **ev**'s for A and A^{-1} are the same.*

Thus if we apply the Power Method to A^{-1} we will obtain the largest absolute **ew** of A^{-1}, which is exactly the reciprocal of the smallest absolute **ew** of A. We will also obtain the corresponding **ev**, which is an **ev** for both A^{-1} and A. Recall that in the application of vibration mode analysis, the smallest **ew** and its **ev** correspond exactly to the frequency and mode that we are most interested in, i.e. the one that can do the most damage.

Here as always, we do not really want to calculate the inverse of A directly if we can help it. Fortunately, multiplying $\mathbf{x}_i$ by A^{-1} to get $\mathbf{x}_{i+1}$ is equivalent to solving the system:

$$A\mathbf{x}_{i+1} = \mathbf{x}_i \, ,$$

which can be done efficiently and accurately. Since iterating this process involves solving a linear system with the same A but many different right hand sides, it is a perfect time to use the LU decomposition to save computations. The following function program does n steps of the Inverse Power Method.

```
function [v e] = myipm(A,n)
% performs n steps of the inverse power method on A
% Inputs:
%  A -- a square matrix
%  n -- the number of iterations to do
% Outputs:
%  e -- the smallest absolute eigenvalue of A and
%  v -- the corresponding eigenvector.

% LU decomposition of A with pivoting
[L U P] = lu(A);

% make an initial vector with ones, normalized
m = size(A,1);
v = ones(m,1)/sqrt(m);

% main loop of Inverse Power Method
for i = 1:n
    pv = P*v;
    y = U\pv;
    v = L\y;
    M = max(v);
    m = min(v);
    % use the biggest in absolute value
    if abs(M) >= abs(m)
       el = M;
    else
       el = m;
    end
    v = v/el;
end
e = 1/el;
```

Exercises

16.1 Write a function program to perform the Power Method on the input matrix A. Use a `while` loop to make it continue until the values stop changing (up to some `tol`). Put a counter in the loop to count the number of steps. Try your program on some random symmetric matrices of various sizes (Hint: use `A=randn(n)` then `A=A+A'`). Is there a relationship between the size and the number of steps needed or between the size and the **ew**? Turn in a printout of your program and a brief report on the trials.

Lecture 17

The QR Method*

The Power Method and Inverse Power Method each give us only one **ew**–**ev** pair. While both of these methods can be modified to give more **ew**'s and **ev**'s, there is a better method for obtaining all the **ew**'s called the *QR method*. This is the basis of all modern **ew** software, including MATLAB, so we summarize it briefly here.

The QR method uses the fact that any square matrix has a *QR decomposition*. That is, for any A there are matrices Q and R such the $A = QR$ where Q has the property

$$Q^{-1} = Q'$$

and R is upper triangular. A matrix Q with the property that its transpose equals its inverse is called an *orthogonal* matrix, because its column vectors are mutually orthogonal.

The QR method consists of iterating following steps:
- decompose A in QR.
- multiply Q and R together in reverse order to form a new A.
The diagonal of R will converge to the eigenvalues.

There is a built-in QR decomposition in MATLAB which is called with the command: `[Q R] = qr(A)`. Thus the following program implements QR method until it converges:

```
function E = myqrmethod(A)
[m n] = size(A);
if m ~= n
   warning('The input matrix is not square.')
   return
end
E = zeros(m,1);
change = 1;
while change > 0
    Eold = E;
    [Q R] = qr(A);
    E = diag(R);
    change = norm(E - Eold);
    A = R*Q;
end
```

As you can see the main steps of the program are very simple. The really hard calculations are contained in the built-in command `qr(A)`.

Exercises

17.1 Modify `myqrmethod` to stop after a maximum number of iterations. Use the modified program, with the maximum iterations set to 1000, on the matrix `A = hilb(n)` with `n` equal to 10, 50, and 200. Use the norm to compare the results to the **ew**'s obtained from MATLAB's built-in program `eig`. Turn in a printout of your program and a brief report on the experiment.

Lecture 18

Iterative solution of linear systems*

Review of Part II

Methods and Formulas

Basic Matrix Theory:
Identity matrix: $AI = A$, $IA = A$, and $I\mathbf{v} = \mathbf{v}$
Inverse matrix: $AA^{-1} = I$ and $A^{-1}A = I$
Norm of a matrix: $|A| \equiv \max_{|\mathbf{v}|=1} |A\mathbf{v}|$
A matrix may be singular or nonsingular.

Solving Process:
Gaussian Elimination
Row Pivoting
Back Substitution

Condition number:

$$\operatorname{cond}(A) \equiv \max \left(\frac{|\delta \mathbf{x}|/|\mathbf{x}|}{\frac{|\delta A|}{|A|} + \frac{|\delta \mathbf{b}|}{|\mathbf{b}|}} \right) = \max \left(\frac{\text{Relative error of output}}{\text{Relative error of inputs}} \right).$$

A big condition number is bad; in engineering it usually results from poor design.

LU factorization:

$$PA = LU.$$

Solving steps:
Multiply by P: $\mathbf{b}' = P\mathbf{b}$
Backsolve: $L\mathbf{y} = \mathbf{b}'$
Backsolve: $U\mathbf{x} = \mathbf{y}$

Eigenvalues and eigenvectors:
A nonzero vector $\mathbf{v}$ is an eigenvector (**ev**) and a number λ is its eigenvalue (**ew**) if

$$A\mathbf{v} = \lambda\mathbf{v}.$$

Characteristic equation: $\det(A - \lambda I) = 0$
Equation of the eigenvector: $(A - \lambda I)\mathbf{v} = \mathbf{0}$

Complex ew's:
Occur in conjugate pairs: $\lambda_{1,2} = \alpha \pm i\beta$
and **ev**'s must also come in conjugate pairs: $\mathbf{w} = \mathbf{u} \pm i\mathbf{v}$.

Vibrational modes:
Eigenvalues are frequencies squared. Eigenvectors are modes.

Power Method:
- Repeatedly multiply **x** by A and divide by the element with the largest absolute value.
- The element of largest absolute value converges to largest absolute **ew**.
- The vector converges to the corresponding **ev**.
- Convergence assured for a real symmetric matrix, but not for an arbitrary matrix, which may not have real eigenvalues at all.

Inverse Power Method:
- Apply power method to A^{-1}.
- Use solving rather than the inverse.
- If λ is an **ew** of A then $1/\lambda$ is an **ew** for A^{-1}.
- The **ev**'s for A and A^{-1} are the same.

QR method:
- Decompose A in QR.
- Multiply Q and R together in reverse order to form a new A.
- Repeat
- The diagonal of R will converge to the **ew**'s of A.

Matlab

Matrix arithmetic:

`>  A = [ 1  3 -2 5 ;  -1  -1 5 4 ; 0 1 -9  0]`	Manually enter a matrix.
`> u = [ 1  2  3  4]'`	
`> A*u`	
`> B = [3 2 1; 7 6 5; 4 3 2]`	
`> B*A`	multiply B times A.
`> 2*A`	multiply a matrix by a scalar.
`> A + A`	add matrices.
`> A + 3`	add 3 to every entry of a matrix.
`> B.*B`	component-wise multiplication.
`> B.^3`	component-wise exponentiation.

Special matrices:

`> I = eye(3)`	identity matrix
`> D = ones(5,5)`	
`> O = zeros(10,10)`	
`> C = rand(5,5)`	random matrix with uniform distribution in $[0,1]$.
`> C = randn(5,5)`	random matrix with normal distribution.
`> hilb(6)`	
`> pascal(5)`	

General matrix commands:

`> size(C)`	gives the dimensions ($m \times n$) of A.
`> norm(C)`	gives the norm of the matrix.
`> det(C)`	the determinant of the matrix.
`> max(C)`	the maximum of each row.
`> min(C)`	the minimum in each row.
`> sum(C)`	sums each row.
`> mean(C)`	the average of each row.
`> diag(C)`	just the diagonal elements.

`> inv(C)` .... inverse of the matrix.
`> C'` .... transpose of the matrix.

Matrix decompositions:

```
> [L U P] = lu(C)
> [Q R] = qr(C)
```
`> [U S V] = svd(C)` .... singular value decomposition (important, but we did not use it).

Part III

Functions and Data

Lecture 19

Polynomial and Spline Interpolation

A Chemical Reaction

In a chemical reaction the concentration level y of the product at time t was measured every half hour. The following results were found:

t	0	.5	1.0	1.5	2.0
y	0	.19	.26	.29	.31

We can input this data into MATLAB as:

```
> t1 = 0:.5:2
> y1 = [ 0 .19 .26 .29 .31 ]
```

and plot the data with:

```
> plot(t1,y1)
```

MATLAB automatically connects the data with line segments. This is the simplest form of *interpolation*, meaning fitting a graph (of a function) between data points. What if we want a smoother graph? Try:

```
> plot(t1,y1,'*')
```

which will produce just asterisks at the data points. Next click on `Tools`, then click on the `Basic Fitting` option. This should produce a small window with several fitting options. Begin clicking them one at a time, clicking them off before clicking the next. Which ones produce a good-looking fit? You should note that the spline, the shape-preserving interpolant and the 4th degree polynomial produce very good curves. The others do not. We will discuss polynomial interpolation and spline interpolation in this lecture.

Polynomial Fitting

The most general degree n polynomial is:

$$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0.$$

If we have exactly $n+1$ data points, that is enough to exactly determine the $n+1$ coefficients of $p_n(x)$ (as long as the data does not have repeated x values). If there are more data points, that would give us an overdetermined system (more equations than unknowns) and if there is less data the system would be undetermined.

Conversely, if we have n data points, then an $n-1$ degree polynomial has exactly enough coefficients to fit the data. This is the case in the example above; there are 5 data points so there is exactly one 4th degree polynomial that fits the data.

When we tried to use a 5th or higher degree polynomial MATLAB returned a warning that the polynomial is not unique since "degree >= number of data points".

When we tried a lower degree polynomial, we did not get a warning, but we did notice that the resulting curve does not hit all the data points. If there was an overdetermined system how did MATLAB come up with a pretty good approximation (for cubic)? The answer is the Least Squares Method, which we will study later.

Predicting the future?

Suppose we want to use the data to extrapolate into the future. Set the plot to the 4th degree polynomial. Then click the `Edit` button and select the `Axes Properties` option. A box should appear that allows you to adjust the domain of the x axes. Change the upper limit of the x-axis from 2 to 4. Based on the 4th degree polynomial, what will the chemical reaction do in the future? Is this reasonable?

Next change from 4th degree polynomial to spline interpolant. According to the spline, what will the chemical reaction do in the future? Try the shape-preserving interpolant also.

From our (limited) knowledge of chemical reactions, what should be the behavior as time goes on? It should reach a limiting value (chemical equilibrium). Could we use the data to predict this equilibrium value? Yes, we could and it is done all the time in many different contexts, but to do so we need to know that there is an equilibrium to predict. This requires that we understand the chemistry of the problem. Thus we have the following principle: To *extrapolate* beyond the data, one must have some knowledge of the process.

More data

Generally one would think that more data is better. Input the following data vectors:

```
> t2 = [ 0  .1  .4  .5  .6  1.0  1.4 1.5 1.6 1.9 2.0]
> y2 = [ 0  .06 .17 .19 .21 .26  .29 .29 .30 .31 .31]
```

There are 11 data points, so a 10-th degree polynomial will fit the data. However, this does not give a good graph. Thus: **A polynomial fit is better for small data sets.**

Finally, we note that each data point (x_i, y_i) gives an equation:

$$p_n(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n = y_i.$$

The unknowns are the coefficients $a_0, \ldots, a_n$. This equation is linear in these unknowns, so determining a polynomial fit requires solving a linear system of equations. This makes polynomial fits quick to calculate since *solving linear systems is what computers do best.*

A challenging data set

Input the following data set:

```
> x = -4:1:5
> y = [ 0 0 0 1 1 1 0 0 0 0]
```

and plot it:

```
> plot(x,y,'*')
```

There are 10 data points, so there is a unique 9 degree polynomial that fits the data. Under `Tools` and `Basic Fitting` select the 9th degree polynomial fit. How does it look? De-select the 9th degree polynomial and select the spline interpolant. This should produce a much more satisfactory graph and the shape-preserving spline should be even better. In this section we learn what a spline is.

The idea of a spline

The general idea of a spline is this: on each interval between data points, represent the graph with a simple function. The simplest spline is something very familiar to you; it is obtained by connecting the data with lines. Since linear is the most simple function of all, linear interpolation is the simplest form of spline. The next simplest function is quadratic. If we put a quadratic function on each interval then we should be able to make the graph a lot smoother. If we were really careful then we should be able to make the curve smooth at the data points themselves by matching up the derivatives. This can be done and the result is called a quadratic spline. Using cubic functions or 4th degree functions should be smoother still. So, where should we stop? There is an almost universal consensus that *cubic* is the optimal degree for splines and so we focus the rest of the lecture on cubic splines.

Cubic spline

Again, the basic idea of the cubic spline is that we represent the function by a different cubic function on each interval between data points. That is, if there are n data points, then the spline $S(x)$ is the function:

$$S(x) = \begin{cases} C_1(x), & x_0 \le x \le x_1 \\ C_i(x), & x_{i-1} \le x \le x_i \\ C_n(x), & x_{n-1} \le x \le x_n \end{cases} \tag{19.1}$$

where each C_i is a cubic function. The most general cubic function has the form:

$$C_i(x) = a_i + b_i x + c_i x^2 + d_i x^3.$$

To determine the spline we must determine the coefficients, a_i, b_i, c_i, and d_i for each i. Since there are n intervals, there are $4n$ coefficients to determine. First we require that the spline be exact at the data:

$$C_i(x_{i-1}) = y_{i-1} \quad \text{and} \quad C_i(x_i) = y_i, \tag{19.2}$$

at every data point. In other words,

$$a_i + b_i x_{i-1} + c_i x_{i-1}^2 + d_i x_{i-1}^3 = y_{i-1} \quad \text{and} \quad a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 = y_i.$$

Notice that there are $2n$ of these conditions. Then to make $S(x)$ as smooth as possible we require:

$$\begin{aligned} C_i'(x_i) &= C_{i+1}'(x_i) \\ C_i''(x_i) &= C_{i+1}''(x_i), \end{aligned} \tag{19.3}$$

at all the internal points, i.e. x_1, $x_2, x_3, \ldots, x_{n-1}$. In terms of the points these conditions can be written as:

$$\begin{aligned} b_i + 2c_i x_i + 3d_i x_i^2 &= b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2 \\ 2c_i + 6d_i x_i &= 2c_{i+1} + 6d_{i+1} x_i. \end{aligned} \tag{19.4}$$

There are $2(n-1)$ of these conditions. Since each C_i is cubic, there are a total of $4n$ coefficients in the formula for $S(x)$. So far we have $4n-2$ equations, so we are 2 equations short of being able to determine all the coefficients. At this point we have to make a choice. The usual choice is to require:

$$C_1''(x_0) = C_n''(x_n) = 0. \tag{19.5}$$

These are called *natural* or *simple* boundary conditions. The other common option is called *clamped* boundary conditions:

$$C_1'(x_0) = C_n'(x_n) = 0. \tag{19.6}$$

The terminology used here is obviously parallel to that used for beams. That is not the only parallel between beams and cubic splines. It is an interesting fact that a cubic spline is exactly the shape of a (linear) beam restrained to match the data by simple supports.

Note that the equations above are all linear equations with respect to the unknowns (coefficients). This feature makes splines easy to calculate since *solving linear systems is what computers do best.*

Exercises

19.1 Plot the following data, then try a polynomial fit of the correct degree to interpolate this number of data points:

```
> t = [ 0  .1  .499  .5  .6  1.0  1.4 1.5 1.899 1.9 2.0]
> y = [ 0  .06 .17 .19 .21 .26  .29 .29 .30 .31 .31]
```

What do you observe. Give an explanation of this error, in particular why is the term *badly conditioned* used?

19.2 Plot the following data along with a spline interpolant:

```
> t = [ 0  .1  .499  .5  .6  1.0  1.4 1.5 1.899 1.9 2.0]
> y = [ 0  .06 .17 .19 .21 .26  .29 .29 .30 .31 .31]
```

How does this compare with the plot above? What is a way to make the plot better?

Lecture 20

Least Squares Interpolation: Noisy Data

Very often data has a significant amount of noise. The least squares approximation is intentionally well-suited to represent noisy data. The next illustration shows the effects noise can have and how least squares is used.

Traffic flow model

Suppose you are interested in the time it takes to travel on a certain section of highway for the sake of planning. According to theory, assuming up to a moderate amount of traffic, the time should be approximately:

$$T(x) = ax + b$$

where b is the travel time when there is no other traffic, and x is the current number of cars on the road (in hundreds). To determine the coefficients a and b you could run several experiments which consist of driving the highway at different times of day and also estimating the number of cars on the road using a counter. Of course both of these measurements will contain *noise*, i.e. random fluctuations.

We could simulate such data in MATLAB as follows:

```
 > x = 1:.1:6;
 > T = .1*x + 1;
 > Tn = T + .1*randn(size(x));
 > plot(x,Tn,'.')
```

The data should look like it lies on a line, but with noise. Click on the `Tools` button and choose `Basic fitting`. Then choose a `linear` fit. The resulting line should go through the data in what looks like a very reasonable way. Click on `show equations`. Compare the equation with $T(x) = .1x + 1$. The coefficients should be pretty close considering the amount of noise in the plot. Next, try to fit the data with a spline. The result should be ugly. We can see from this example that **splines are not suited to noisy data**.

How does MATLAB obtain a very nice line to approximate noisy data? The answer is a very standard numerical/statistical method known as *least squares.*

Linear least squares

Consider in the previous example that we wish to fit a line to a lot of data that does not exactly lie on a line. For the equation of the line we have only two free coefficients, but we have many data points. We can not possibly make the line go through every data point, we can only wish for it to come reasonably close to as many data points as possible. Thus, our line must have an error with respect to each data point. If $\ell(x)$ is our line and $\{(x_i, y_i)\}$ are the data, then

$$e_i = y_i - \ell(x_i)$$

is the error of ℓ with respect to each (x_i, y_i). To make $\ell(x)$ reasonable, we wish to simultaneously minimize all the errors: $\{e_1, e_2, \ldots, e_n\}$. There are many possible ways one could go about this, but the standard one is to try to minimize the *sum of the squares* of the errors. That is, we denote by $\mathcal{E}$ the sum of the squares:

$$\mathcal{E} = \sum_{i=1}^{n} (y_i - \ell(x_i))^2 = \sum_{i=1}^{n} (y_i - ax_i - b)^2 . \tag{20.1}$$

In the above expression x_i and y_i are given, but we are free to choose a and b, so we can think of $\mathcal{E}$ as a function of a and b, i.e. $\mathcal{E}(a, b)$. In calculus, when one wishes to find a minimum value of a function of two variables, we set the partial derivatives equal to zero:

$$\begin{aligned} \frac{\partial \mathcal{E}}{\partial a} &= -2 \sum_{i=1}^{n} (y_i - ax_i - b)\, x_i = 0 \\ \frac{\partial \mathcal{E}}{\partial b} &= -2 \sum_{i=1}^{n} (y_i - ax_i - b) = 0. \end{aligned} \tag{20.2}$$

We can simplify these equations to obtain:

$$\begin{gathered} \left(\sum_{i=1}^{n} x_i^2\right) a + \left(\sum_{i=1}^{n} x_i\right) b = \sum_{i=1}^{n} x_i y_i \\ \left(\sum_{i=1}^{n} x_i\right) a + nb = \sum_{i=1}^{n} y_i. \end{gathered} \tag{20.3}$$

Thus, the whole problem reduces to a 2 by 2 linear system to find the coefficients a and b. The entries in the matrix are determined from simple formulas using the data. The process is quick and easily automated, which is one reason it is very standard.

We could use the same process to obtain a quadratic or higher polynomial fit to data. If we try to fit an n degree polynomial, the software has to solve an $n \times n$ linear system, which is easily done. This is what MATLAB's basic fitting tool uses to obtain an n degree polynomial fit whenever the number of data points is more than $n + 1$.

Drag coefficients

Drag due to air resistance is proportional to the square of the velocity, i.e. $d = kv^2$. In a wind tunnel experiment the velocity v can be varied by setting the speed of the fan and the drag can be measured directly (it is the force on the object). In this and every experiment some random noise will occur. The following sequence of commands replicates the data one might receive from a wind tunnel:

```
> v = 0:1:60;
> d = .1234*v.^2;
> dn = d + .4*v.*randn(size(v));
> plot(v,dn,'*')
```

The plot should look like a quadratic, but with some noise. Using the tools menu, add a quadratic fit and enable the "show equations" option. What is the coefficient of x^2? How close is it to 0.1234?

Note that whenever you select a ploynomial interpolation in Matlab with a degree less than $n-1$ Matlab will produce a least squares interpolation.

You will notice that the quadratic fit includes both a constant and linear term. We know from the physical situation that these should not be there; they are remnants of noise and the fitting process. Since we know the curve should be kv^2, we can do better by employing that knowledge. For instance, we know that the graph of d versus v^2 should be a straight line. We can produce this easily:

```
> vs = v.^2;
> plot(vs,dn,'*')
```

By changing the independent variable from v to v^2 we produced a plot that looks like a line with noise. Add a linear fit. What is the linear coefficient? This should be closer to 0.1234 than using a quadratic fit.

The second fit still has a constant term, which we know should not be there. If there was no noise, then at every data point we would have $k = d/v^2$. We can express this as a linear system `vs'*k = dn'`, which is badly overdetermined since there are more equations than unknowns. Since there is noise, each point will give a different estimate for k. In other words, the overdetermined linear system is also inconsistent. When Matlab encounters such systems, it automatically gives a least squares solution of the matrix problem, i.e. one that minimizes the sum of the squared errors, which is exactly what we want. To get the least squares estimate for k, do

```
> k = vs'\dn'  .
```

This will produce a number close to .1234.

Note that this is an application where we have physical knowledge. In this situation extrapolation would be meaningful. For instance we could use the plot to find the predicted drag at 80 mph.

Exercises

20.1 Find two tables of data in an engineering textbook. Plot each and decide if the data is best represented by a polynomial, spline or least squares polynomial. Turn in the best plot of each. Indicate the source and meaning of the data.

20.2 The following table contains information from a chemistry experiment in which the concentration of a product was measured at one minute intervals.

T	0	1	2	3	4	5	6	7
C	3.033	3.306	3.672	3.929	4.123	4.282	4.399	4.527

Plot this data. Suppose it is known that this chemical reaction should follow the law: $c = a - b\exp(-0.2t)$. Following the example in the notes about the drag coefficients, change one of the variables so that the law is a linear function. Then plot the new variables and use the linear fit option to estimate a and b. What will be the eventual concentration? Finally, plot the graph of $a - b\exp(-0.2t)$ on the interval [0,10], along with the data.

Lecture 21

Integration: Left, Right and Trapezoid Rules

The Left and Right endpoint rules

In this section, we wish to approximate a definite integral:

$$\int_a^b f(x)\,dx$$

where $f(x)$ is a continuous function. In calculus we learned that integrals are (signed) areas and can be approximated by sums of smaller areas, such as the areas of rectangles. We begin by choosing points $\{x_i\}$ that subdivide $[a,b]$:

$$a = x_0 < x_1 < \ldots < x_{n-1} < x_n = b.$$

The subintervals $[x_{i-1}, x_i]$ determine the width Δ_i of each of the approximating rectangles. For the height, we learned that we can choose any height of the function $f(x_i^*)$ where $x_i^* \in [x_{i-1}, x_i]$. The resulting approximation is:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} f(x_i^*)\Delta_i.$$

To use this to approximate integrals with actual numbers, we need to have a specific x_i^* in each interval. The two simplest (and worst) ways to choose x_i^* are as the left-hand point or the right-hand point of each interval. This gives concrete approximations which we denote by L_n and R_n given by

$$L_n = \sum_{i=1}^{n} f(x_{i-1})\Delta_i \quad \text{and}$$

$$R_n = \sum_{i=1}^{n} f(x_i)\Delta_i\,.$$

```
function L = myleftsum(x,y)
% produces the left sum from data input.
% Inputs: x -- vector of the x coordinates of the partitian
%         y -- vector of the corresponding y coordinates
% Output: returns the approximate integral
n = size(x);
L = 0;
for i = 1:n-1
   L = L + y(i)*(x(i+1) - x(i));
end
```

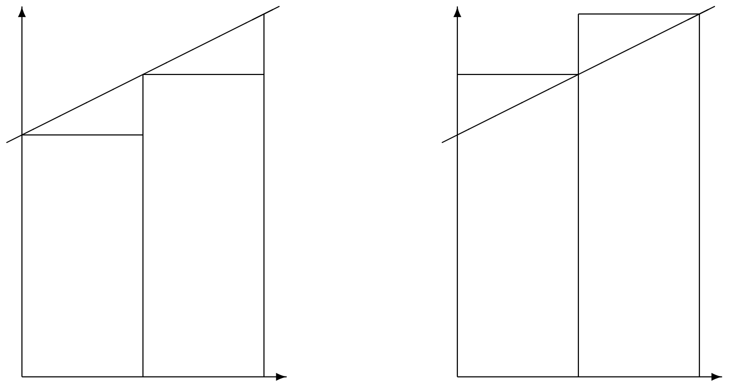

Figure 21.1: The left and right sums, L_n and R_n.

Often we can take $\{x_i\}$ to be *evenly spaced*, with each interval having the same width:

$$h = \frac{b-a}{n},$$

where n is the number of subintervals. If this is the case, then L_n and R_n simplify to:

$$L_n = \frac{b-a}{n} \sum_{i=0}^{n-1} f(x_i) \quad \text{and} \tag{21.1}$$

$$R_n = \frac{b-a}{n} \sum_{i=1}^{n} f(x_i). \tag{21.2}$$

The foolishness of choosing left or right endpoints is illustrated in Figure 21.1. As you can see, for a very simple function like $f(x) = 1 + .5x$, each rectangle of L_n is too short, while each rectangle of R_n is too tall. This will hold for any increasing function. For decreasing functions L_n will always be too large while R_n will always be too small.

The Trapezoid rule

Knowing that the errors of L_n and R_n are of opposite sign, a very reasonable way to get a better approximation is to take an average of the two. We will call the new approximation T_n:

$$T_n = \frac{L_n + R_n}{2}.$$

This method also has a straight-forward geometric interpretation. On each subrectangle we are using:

$$A_i = \frac{f(x_{i-1}) + f(x_i)}{2} \Delta_i \,,$$

which is exactly the area of the *trapezoid* with sides $f(x_{i-1})$ and $f(x_i)$. We thus call the method the trapezoid method. See Figure 21.2.

We can rewrite T_n as

$$T_n = \sum_{i=1}^{n} \frac{f(x_{i-1}) + f(x_i)}{2} \Delta_i.$$

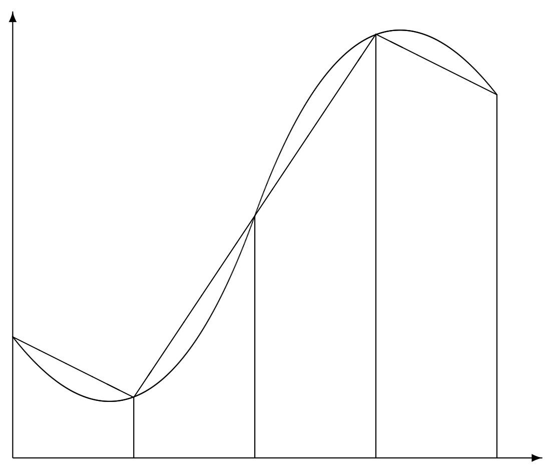

Figure 21.2: The trapezoid rule, T_n.

In the evenly spaced case, we can write this as

$$T_n = \frac{b-a}{2n}\big(f(x_0) + 2f(x_1) + \ldots + 2f(x_{n-1}) + f(x_n)\big). \tag{21.3}$$

Caution: The convention used here is to begin numbering the points at 0, i.e. $x_0 = a$; this allows n to be the number of subintervals and the index of the last point x_n. However, MATLAB's indexing convention begins at 1. Thus, when programming in MATLAB, the first entry in `x` will be x_0, i.e. `x(1)`$= x_0$ and `x(n+1)`$= x_n$.

If we are given data about the function, rather than a formula for the function, often the data are not evenly spaced. The following function program could then be used.

```
function T = mytrap(x,y)
% Calculates the Trapezoid rule approximation of the integral from input data
% Inputs: x -- vector of the x coordinates of the partitian
%         y -- vector of the corresponding y coordinates
% Output: returns the approximate integral
n = size(x);
T = 0;
for i = 1:n-1
   T = T + .5*(y(i)+y(i+1))*(x(i+1) - x(i));
end
```

Using the Trapezoid rule for areas in the plane

In multi-variable calculus you were supposed to learn that you can calculate the area of a region R in the plane by calculating the line integral:

$$A = -\oint_C y\,dx, \tag{21.4}$$

where C is the counter-clockwise curve around the boundary of the region. We can represent such a curve by consecutive points on it, i.e. $\bar{x} = (x_0, x_1, x_2, \ldots, x_{n-1}, x_n)$, and $\bar{y} = (y_0, y_1, y_2, \ldots, y_{n-1}, y_n)$.

Since we are assuming the curve ends where it begins, we require $(x_n, y_n) = (x_0, y_0)$. Applying the trapezoid method to the integral (21.4) gives

$$A = -\sum_{i=1}^{n} \frac{y_{i-1} + y_i}{2} (x_i - x_{i-1}).$$

This formula then is the basis for calculating areas when coordinates of boundary points are known, but not necessarily formulas for the boundaries such as in a land survey.

In the following script, we can use this method to approximate the area of a unit circle using n points on the circle:

```
% Calculates pi using a trapezoid approximation of the unit circle.
format long
n = 10;
t = linspace(0,2*pi,n+1);
x = cos(t);
y = sin(t);
plot(x,y)
A = 0
for i = 1:n
    A = A - (y(i)+y(i+1))*(x(i+1)-x(i))/2;
end
A
```

Exercises

21.1 For the integral $\int_1^2 \sqrt{x}\, dx$ calculate L_4, R_4, and T_4 with even spacing (by hand, but use a calculator) using formulas (21.1), (21.2) and (21.3). Find the precentage error of these approximations, using the exact value.

21.2 Write a function program `myints` whose inputs are f, a, b and n and whose outputs are L, R and T, the left, right and trapezoid integral approximations for f on $[a, b]$ with n subintervals. To make it efficient, first use `x = linspace(a,b,n+1)` to make the x values and `y = f(x)` to make the y values, then add the 2nd to nth y entries only once (use only one loop in the program).

Change to `format long` and apply your program to the integral $\int_1^2 \sqrt{x}\, dx$. Compare with the results of the previous exercise. Also find L_{100}, R_{100} and T_{100} and the relative errors of these approximations.

Turn in the program and a brief summary of the results.

Lecture 22

Integration: Midpoint and Simpson's Rules

Midpoint rule

If we use the endpoints of the subintervals to approximate the integral, we run the risk that the values at the endpoints do not accurately represent the average value of the function on the subinterval. A point which is much more likely to be close to the average would be the midpoint of each subinterval. Using the midpoint in the sum is called the *midpoint rule.* On the i-th interval $[x_{i-1}, x_i]$ we will call the midpoint $\bar{x}_i$, i.e.

$$\bar{x}_i = \frac{x_{i-1} + x_i}{2}.$$

If $\Delta_i = x_i - x_{i-1}$ is the length of each interval, then using midpoints to approximate the integral would give the formula:

$$M_n = \sum_{i=1}^{n} f(\bar{x}_i)\Delta_i \,.$$

For even spacing, $\Delta = (b-a)/n$, and the formula is:

$$M_n = \frac{b-a}{n} \sum_{i=1}^{n} f(\bar{x}_i) = \frac{b-a}{n} \left(\hat{y}_1 + \hat{y}_2 + \ldots + \hat{y}_n\right), \tag{22.1}$$

where we define $\hat{y}_i = f(\bar{x}_i)$.

While the midpoint method is obviously better than L_n or R_n, it is not obvious that it is actually better than the trapezoid method T_n, but it is.

Simpson's rule

Consider Figure 22.1. If f is not linear on a subinterval, then it can be seen that the errors for the midpoint and trapezoid rules behave in a very predictable way, they have opposite sign. For example, if the function is concave up then T_n will be too high, while M_n will be too low. Thus it makes sense that a better estimate would be to average T_n and M_n. However, in this case we can do better than a simple average. The error will be minimized if we use a weighted average. To find the proper weight we take advantage of the fact that for a quadratic function the errors EM_n and ET_n are exactly related by:

$$|EM_n| = \frac{1}{2}|ET_n| \,.$$

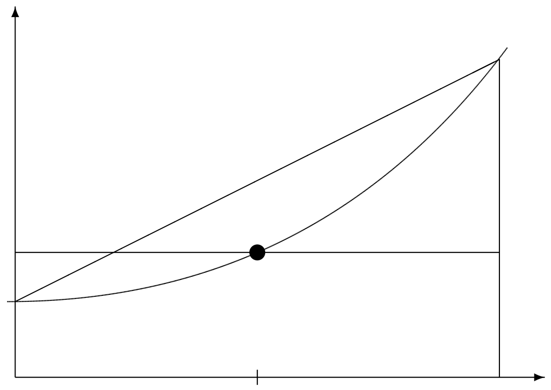

Figure 22.1: Comparing the trapezoid and midpoint method on a single subinterval. The function is concave up, in which case T_n is too high, while M_n is too low.

Thus we take the following weighted average of the two, which is called Simpson's rule:

$$S_{2n} = \frac{2}{3}M_n + \frac{1}{3}T_n \,.$$

If we use this weighting on a quadratic function the two errors will exactly cancel.

Notice that we write the subscript as $2n$. That is because we usually think of $2n$ subintervals in the approximation; the n subintervals of the trapezoid are further subdivided by the midpoints. We can then number all the points using integers. If we number them this way we notice that the number of subintervals must be an even number.

The formula for Simpson's rule if the subintervals are evenly spaced is the following (with n intervals, where n is even):

$$S_n = \frac{b-a}{3n}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\right) \,.$$

Note that if we are presented with data $\{x_i, y_i\}$ where the x_i points are evenly spaced with $x_{i+1} - x_i = \Delta x$, it is easy to apply Simpson's method:

$$S_n = \frac{\Delta x}{3}\left(y_0 + 4y_1 + 2y_2 + 4y_3 + \ldots + 2y_{n-2} + 4y_{n-1} + y_n\right). \qquad (22.2)$$

Notice the pattern of the coeficients. The following program will produce these coefficients for n intervals, if n is an even number. Try it for $n = 6, 7, 100$.

```
function w = mysimpweights(n)
% computes the weights for Simpson's rule
% Input: n -- the number of intervals, must be even
% Output: a vector with the weights, length n+1
if rem(n,2) ~= 0
    error('n must be even')
end
w = ones(n+1,1);
for i = 2:n
    if rem(i,2)==0
        w(i)=4;
    else
        w(i)=2;
    end
end
```

Simpson's rule is incredibly accurate. We will consider just how accurate in the next section. The one drawback is that the points used must either be evenly spaced, or at least the odd number points must lie exactly at the midpoint between the even numbered points. In applications where you can choose the spacing, this is not a problem. In applications such as experimental or simulated data, you might not have control over the spacing and then you cannot use Simpson's rule.

Error bounds

The trapezoid, midpoint, and Simpson's rules are all approximations. As with any approximation, before you can safely use it, you must know how good (or bad) the approximation might be. For these methods there are formulas that give upper bounds on the error. In other words, the worst case errors for the methods. These bounds are given by the following statements:

- Suppose f'' is continuous on [a,b]. Let $K_2 = \max_{x\in[a,b]} |f''(x)|$. Then the errors ET_n and EM_n of the Trapezoid and Midpoint rules, respectively, applied to $\int_a^b f\,dx$ satisfy:

$$\begin{aligned}|ET_n| &\leq \frac{K_2(b-a)^3}{12n^2} = \frac{K_2(b-a)}{12}\Delta x^2 \quad \text{and}\\ |EM_n| &\leq \frac{K_2(b-a)^3}{24n^2} = \frac{K_2(b-a)}{24}\Delta x^2\,.\end{aligned}$$

- Suppose $f^{(4)}$ is continuous on [a,b]. Let $K_4 = \max_{x\in[a,b]} |f^{(4)}(x)|$. Then the error ES_n of Simpson's rule applied to $\int_a^b f\,dx$ satisfies:

$$|ES_n| \leq \frac{K_4(b-a)^5}{180n^4} = \frac{K_4(b-a)}{180}\Delta x^4.$$

In practice K_2 and K_4 are themselves approximated from the values of f at the evaluation points.

The most important thing in these error bounds is the last term. For the trapezoid and midpoint method, the error depends on Δx^2 whereas the error for Simpson's rule depends on Δx^4. If Δx is just moderately small, then there is a huge advantage with Simpson's method.

When an error depends on a power of a parameter, such as above, we sometimes use the **order notation**. In the above error bounds we say that the trapezoid and midpoint rules have errors of order $O(\Delta x^2)$, whereas Simpson's rule has error of order $O(\Delta x^4)$.

In MATLAB there is a built-in command for definite integrals: `quad(f,a,b)` where the `f` is an inline function and `a` and `b` are the endpoints. Here `quad` stands for quadrature, which is a term for numerical integration. The command uses "adaptive Simpson quadrature", a form of Simpson's rule that checks its own accuracy and adjusts the grid size where needed. Here is an example of its usage:

```
 > f = inline('x.^(1/3).*sin(x.^3)')
 > I = quad(f,0,1)
```

Exercises

22.1 Using formulas (22.1) and (22.2), for the integral $\int_1^2 \sqrt{x}\,dx$ calculate M_4 and S_4 (by hand, but use a calculator). Find the relative error of these approximations, using the exact value. Compare with exercise 21.1.

22.2 Write a function program `mymidpoint` that calculates the midpoint rule approximation for $\int f$ on the interval $[a,b]$ with n subintervals. The inputs should be f, a, b and n. Use your program on the integral $\int_1^2 \sqrt{x}\,dx$ to obtain M_4 and M_{100}. Compare these with the previous exercise and the true value of the integral.

22.3 Write a function program `mysimpson` that calculates the Simpson's rule approximation for $\int f$ on the interval $[a,b]$ with n subintervals. It should use the program `mysimpweights` to produce the coefficients. Use your program on the integral $\int_1^2 \sqrt{x}\,dx$ to obtain S_4 and S_{100}. Compare these with the previous exercise and the true value.

Lecture 23

Plotting Functions of Two Variables

Functions on Rectangular Grids

Suppose you wish to plot a function $f(x, y)$ on the rectangle $a \leq x \leq b$ and $c \leq y \leq d$. The graph of a function of two variables is of course a three dimensional object. Visualizing the graph is often very useful.

For example, suppose you have a formula:

$$f(x, y) = x \sin(xy)$$

and you are interested in the function on the region $0 \leq x \leq 5$, $\pi \leq y \leq 2\pi$. A way to plot this function in MATLAB would be the following sequence of commands:

```
> f = inline('x.*sin(x.*y)','x','y')
> [X,Y] = meshgrid(0:.1:5,pi:.01*pi:2*pi);
> Z = f(X,Y)
> mesh(X,Y,Z)
```

This will produce a 3-D plot that you can rotate by clicking on the rotate icon and then dragging with the mouse.

Instead of the command `mesh`, you could use the command:

```
> surf(X,Y,Z)
```

The key command in this sequence is `[X Y] = meshgrid(a:h:b,c:k:d)`, which produces *matrices of x and y values* in `X` and `Y`. Enter:

```
> size(X)
> size(Y)
> size(Z)
```

to see that each of these variables is a 51×51 matrix. To see the first few entries of `X` enter:

```
> X(1:6,1:6)
```

and to see the first few values of `Y` type:

```
> Y(1:6,1:6)
```

You should observe that the x values in `X` begin at 0 on the left column and increase from left to right. The y values on the other have start at π at the top and increase from top to bottom. Note that this arrangement is flipped from the usual arrangment in the x-y plane.

In the command `[X Y] = meshgrid(a:h:b,c:k:d)`, h is the increment in the x direction and k is the increment in the y direction. Often we will calculate:

$$h = \frac{b-a}{m} \quad \text{and} \quad k = \frac{d-c}{n},$$

where m is the number of *intervals* in the x direction and n is the number of intervals in the y direction. To obtain a good plot it is best if m and n can be set between 10 and 100.

For another example of how `meshgrid` works, try the following and look at the output in `X` and `Y`.

```
 > [X,Y] = meshgrid(0:.5:4,1:.2:2);
```

Scattered Data and Triangulation

Often we are interested in objects whose bases are not rectangular. For instance, data does not usually come arranged in a nice rectangular grid; rather, measurements are taken where convenient.

In Matlab we can produce triangles for a region by recording the coordinates of the vertices and recording which vertices belong to each triangle. The following script program produces such a set of triangles:

```
% mytriangles
% Program to produce a triangulation.
% V contains vertices, which are (x,y) pairs
V = [ 1/2 1/2  ;  1   1 ; 3/2 1/2 ;  .5  1.5 ;  0    0
      1    0  ;  2   0 ; 2    1  ; 1.5 1.5  ;  1    2
      0    2  ;  0    1]
% x, y are row vectors containing coordinates of vertices
x = V(:,1)';
y = V(:,2)';
%  Assign the triangles
T = delaunay(x,y)
```

You can plot the triangles using the following command:

```
 > trimesh(T,x,y)
```

You can also prescribe values (heights) at each vertex directly (say from a survey):

```
 > z1 = [ 2 3 2.5 2 1 1 .5 1.5 1.6 1.7 .9 .5 ];
```

or using a function:

```
 > f = inline('abs(sin(x.*y)).^(3/2)','x','y');
 > z2 = f(x,y);
```

The resulting profiles can be plotted:

```
 > trimesh(T,x,y,z1)
 > trisurf(T,x,y,z2)
```

Each row of the matrix `T` corresponds to a triangle, so `T(i,:)` gives triangle number `i`. The three corners of triangle number `i` are at indices `T(i,1)`, `T(i,2)`, and `T(i,3)`. So for example to get the y-coordinate of the second point of triangle number 5, enter:

```
 > y(T(5,2))
```

To see other examples of regions defined by triangle get `mywedge.m` and `mywasher.m` from the class web site and run them. Each of these programs defines vectors `x` and `y` of x and y values of vertices and a matrix `T`. As before `T` is a list of sets of three integers. Each triple of integers indicates which vertices to connect in a triangle.

To plot a function, say $f(x, y) = x^2 - y^2$ on the washer figure try:

```
 > mywasher
 > z = x.^2 - y.^2
 > trisurf(T,x,y,z)
```

Note again that this plot can be rotated using the icon and mouse.

Exercises

23.1 Plot the function $f(x,y) = xe^{-x^2-y^2}$ on the rectangle $-3 \leq x \leq 3$, $-2 \leq y \leq 2$ using meshgrid. Make an appropriate choice of h and k and if necessary a rotation to produce a good plot. Turn in your plot and the calculation of k and h.

23.2 Download the programs `mywedge.m` and `mywasher.m` from the web site. Plot $f(x,y) = xe^{-x^2-y^2}$ for each of these figures. Also plot a function that is zero on all nodes except on one boundary node and one interior node, where it has value 1.

Lecture 24

Double Integrals for Rectangles

The center point method

Suppose that we need to find the integral of a function, $f(x, y)$, on a rectangle:

$$R = \{(x, y) : a \le x \le b, c \le y \le d\}.$$

In calculus you learned to do this by an iterated integral:

$$I = \iint_R f\, dA = \int_a^b \int_c^d f(x, y)\, dydx = \int_c^d \int_a^b f(x, y)\, dxdy.$$

You also should have learned that the integral is the limit of the Riemann sums of the function as the size of the subrectangles goes to zero. This means that the Riemann sums are approximations of the integral, which is precisely what we need for numerical methods.

For a rectangle R, we begin by subdividing into smaller subrectangles $\{R_{ij}\}$, in a systematic way. We will divide $[a, b]$ into m subintervals and $[c, d]$ into n subintervals. Then R_{ij} will be the "intersection" of the i-th subinterval in $[a, b]$ with the j-th subinterval of $[c, d]$. In this way the entire rectangle is subdivided into mn subrectangles, numbered as in Figure 24.1.

A Riemann sum using this subdivision would have the form:

$$S = \sum_{i,j=1,1}^{m,n} f(x_{ij}^*)A_{ij} = \sum_{j=1}^{n} \left(\sum_{i=1}^{m} f(x_{ij}^*)A_{ij} \right),$$

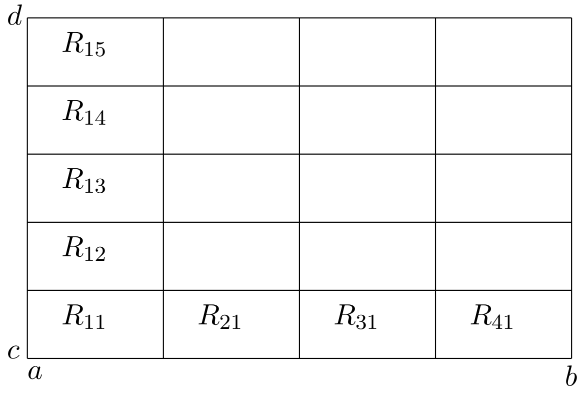

Figure 24.1: Subdivision of the rectangle $R = [a, b] \times [c, d]$ into subrectangles R_{ij}.

where $A_{ij} = \Delta x_i \Delta y_j$ is the area of R_{ij}, and x^*_{ij} is a point in R_{ij}. The theory of integrals tells us that if f is continuous, then this sum will converge to the same number, no matter how we choose x^*_{ij}. For instance, we could choose x^*_{ij} to be the point in the lower left corner of R_{ij} and the sum would still converge as the size of the subrectangles goes to zero. However, in practice we wish to choose x^*_{ij} in such a way to make S as accurate as possible even when the subrectangles are not very small. The obvious choice for the best point in R_{ij} would be the center point. The center point is most likely of all points to have a value of f close to the average value of f. If we denote the center points by c_{ij}, then the sum becomes

$$C_{mn} = \sum_{i,j=1,1}^{m,n} f(c_{ij}) A_{ij}.$$

Here

$$c_{ij} = \left(\frac{x_{i-1} + x_i}{2}, \frac{y_{i-1} + y_i}{2} \right).$$

Note that if the subdivision is evenly spaced then $\Delta x \equiv (b-a)/m$ and $\Delta y \equiv (d-c)/n$, and so in that case

$$C_{mn} = \frac{(b-a)(d-c)}{mn} \sum_{i,j=1,1}^{m,n} f(c_{ij}).$$

The four corners method

Another good idea would be to take the value of f not only at one point, but as the average of the values at several points. An obvious choice would be to evaluate f at all four corners of each R_{ij} then average those. If we note that the lower left corner is (x_i, y_j), the upper left is (x_i, y_{j+1}), the lower right is (x_{i+1}, y_i) and the upper right is (x_{i+1}, y_{i+1}), then the corresponding sum will be

$$F_{mn} = \sum_{i,j=1,1}^{m,n} \frac{1}{4} \left(f(x_i, y_j) + f(x_i, y_{j+1}) + f(x_{i+1}, y_i) + f(x_{i+1}, y_{j+1}) \right) A_{ij},$$

which we will call the *four-corners* method. If the subrectangles are evenly spaced, then we can simplify this expression. Notice that $f(x_i, y_j)$ gets counted multiple times depending on where (x_i, y_j) is located. For instance if (x_i, y_j) is in the interior of R then it is the corner of 4 subrectangles. Thus the sum becomes

$$F_{mn} = \frac{A}{4} \left(\sum_{\text{corners}} f(x_i, y_j) + 2 \sum_{\text{edges}} f(x_i, y_j) + 4 \sum_{\text{interior}} f(x_i, y_j) \right),$$

where $A = \Delta x \, \Delta y$ is the area of the subrectangles. We can think of this as a weighted average of the values of f at the grid points (x_i, y_j). The weights used are represented in the following matrix

$$W = \begin{pmatrix} 1 & 2 & 2 & 2 & \cdots & 2 & 2 & 1 \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ 1 & 2 & 2 & 2 & \cdots & 2 & 2 & 1 \end{pmatrix}. \tag{24.1}$$

We could implement the four-corner method by forming a matrix (f_{ij}) of f values at the grid points, then doing entry-wise multiplication of the matrix with the weight matrix. Then the integral would be obtained by summing all the entries of the resulting matrix and multiplying that by $A/4$. The formula would be

$$F_{mn} = \frac{(b-a)(d-c)}{4mn} \sum_{i,j=1,1}^{m,n} W_{ij} f(x_i, y_j).$$

Notice that the four-corners method coincides with applying the trapezoid rule in each direction. Thus it is in fact a *double trapezoid* rule.

The double Simpson method

The next improvement one might make would be to take an average of the center point sum C_{mn} and the four corners sum F_{mn}. However, a more standard way to obtain a more accurate method is the Simpson double integral. It is obtained by applying Simpson's rule for single integrals to the iterated double integral. The resulting method requires that both m and n be even numbers and the grid be evenly spaced. If this is the case we sum up the values $f(x_i, y_j)$ with weights represented in the matrix

$$W = \begin{pmatrix} 1 & 4 & 2 & 4 & \cdots & 2 & 4 & 1 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ 2 & 8 & 4 & 8 & \cdots & 4 & 8 & 2 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ \vdots & \vdots & \vdots & \vdots & \ldots & \vdots & \vdots & \vdots \\ 2 & 8 & 4 & 8 & \cdots & 4 & 8 & 2 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ 1 & 4 & 2 & 4 & \cdots & 2 & 4 & 1 \end{pmatrix}. \tag{24.2}$$

The sum of the weighted values is multiplied by $A/9$ and the formula is

$$S_{mn} = \frac{(b-a)(d-c)}{9mn} \sum_{i,j=1,1}^{m,n} W_{ij} f(x_i, y_j).$$

MATLAB has a built in command for double integrals on rectangles: `dblquad(f,a,b,c,d)`. Here is an example:

```
 > f = inline('sin(x.*y)./sqrt(x+y)','x','y')
 > I = dblquad(f,0.5,1,0.5,2)
```

Below is a MATLAB function which will produce the matrix of weights needed for Simpson's rule for double integrals. It uses the function `mysimpweights` from Lecture 22.

```
function W = mydblsimpweights(m,n)
% Produces the m by n matrix of weights for Simpson's rule
% for double integrals
% Inputs: m -- number of intervals in the row direction.
%              must be even.
%         n -- number of intervals in the column direction.
%              must be even.
% Output: W -- a (m+1)x(n+1) matrix of the weights
if rem(m,2)~=0 | rem(n,2)~=0
    error('m and n must be even')
end
u = mysimpweights(m);
v = mysimpweights(n);
W = u*v'
```

Exercises

24.1 Download the program **`mylowerleft`** from the web site. Modify it to make a program **`mycenter`** that does the center point method. Implement the change by changing the "meshgrid" to put the grid points at the centers of the subrectangles. Look at the mesh plot produced to make sure the program is putting the grid where you want it. Use both programs to approximate the integral

$$\int_0^2 \int_1^5 \sqrt{xy}\, dy\, dx,$$

using $(m, n) = (10, 18)$. Evaluate this integral exactly to make comparisons.

24.2 Write a program **`mydblsimp`** that computes the Simpson's rule approximation. Let it use the program **`mydblsimpweights`** to make the weight matrix (24.2). Check the accuracy of the program on the integral in the previous problem.

24.3 Using **`mysimweights`** and **`mydblsimpweights`** as models make programs **`mytrapweights`** and **`mydbltrapweights`** that will produce the weights for the trapezoid rule and the weight matrix for the four corners (double trapezpoid) method (24.1).

Lecture 25

Double Integrals for Non-rectangles

In the previous lecture we considered only integrals over rectangular regions. In practice, regions of interest are rarely rectangles and so in this lecture we consider two strategies for evaluating integrals over other regions.

Redefining the function

One strategy is to redefine the function so that it is zero outside the region of interest, then integrate over a rectangle that includes the region.

For example, suppose we need to approximate the value of

$$I = \iint_T \sin^3(xy)\,dx\,dy$$

where T is the triangle with corners at $(0,0)$, $(1,0)$ and $(0,2)$. Then we could let R be the rectangle $[0,1]\times[0,2]$ which contains the triangle T. Notice that the hypotenuse of the triangle has the equation $2x+y=2$. Then make $f(x)=\sin^3(xy)$ if $2x+y\le 2$ and $f(x)=0$ if $2x+y>2$. In Matlab we can make this function with the command:

```
 > f = inline('sin(x.*y).^3.*(2*x + y <= 2)')
```

In this command `<=` is a *logical* command. The term in parentheses is then a *logical statement* and is given the value 1 if the statement is true and 0 if it is false. We can then integrate the modified `f` on $[0,1]\times[0,2]$ using the command:

```
 > I = dblquad(f,0,1,0,2)
```

As another example, suppose we need to integrate $x^2\exp(xy)$ inside the circle of radius 2 centered at $(1,2)$. The equation for this circle is $(x-1)^2+(y-2)^2=4$. Note that the inside of the circle is $(x-1)^2+(y-2)^2\le 4$ and that the circle is contained in the rectangle $[-1,3]\times[0,4]$. Thus we can create the right function and integrate it by:

```
 > f = inline('x.^2.*exp(x.*y).*((x-1).^2 + (y-2).^2 <= 4)')
 > I = dblquad(f,-1,3,0,4)
```

Integration Based on Triangles

The second approach to integrating over non-rectangular regions, is based on subdividing the region into triangles. Such a subdivision is called a *triangulation.* On regions where the boundary consists of line segments, this can be done exactly. Even on regions where the boundary contains curves, this can be done approximately. This is a very important idea for several reasons, the most important of which is that the finite elements method is based on it. Another reason this is important is that often the values of f are not given by a formula, but from data. For example, suppose you are surveying on a construction site and you want to know how much fill will be needed to bring the level up to the plan. You would proceed by taking elevations at numerous points across the site. However, if the site is irregularly shaped or if there are obstacles on the site, then you cannot make these measurements on an exact rectangular grid. In this case, you can use triangles by connecting your points with triangles. Many software packages will even choose the triangles for you (MATLAB will do it using the command `delaunay`).

The basic idea of integrals based on triangles is exactly the same as that for rectangles; the integral is approximated by a sum where each term is a value times an area

$$I \approx \sum_{i=1}^{n} f(x_i^*) A_i \,,$$

where n is the number of triangles, A_i is the area of the triangle and x^* a point in the triangle. However, rather than considering the value of f at just one point people often consider an average of values at several points. The most convenient of these is of course the corner points. We can represent this sum by

$$T_n = \sum_{i=1}^{n} \bar{f}_i A_i \,,$$

where $\bar{f}$ is the average of f at the corners.

If the triangle has vertices (x_1, y_1), (x_2, y_2) and (x_3, y_3), the formula for area is

$$A = \frac{1}{2} \left| \det \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix} \right| . \tag{25.1}$$

A function `mythreecorners` to compute using the three corners method is given below.

Another idea would be to use the center point (centroid) of each triangle. If the triangle has vertices (x_1, y_1), (x_2, y_2) and (x_3, y_3), then the centroid is given by the simple formulas

$$\bar{x} = \frac{x_1 + x_2 + x_3}{3} \quad \text{and} \quad \bar{y} = \frac{y_1 + y_2 + y_3}{3} . \tag{25.2}$$

```
function  I = mythreecorners(f,V,T)
% Integrates a function based on a triangulation, using three corners
% Inputs: f -- the function to integrate, as an inline
%         V -- the vertices. Each row has the x and y coordinates of a vertex
%         T -- the triangulation. Each row gives the indices of three corners
% Output: the approximate integral

x = V(:,1); % extract x and y coordinates of all nodes
y = V(:,2);
I=0;
p = size(T,1);
for i = 1:p
   x1 = x(T(i,1));  % find coordinates and area
   x2 = x(T(i,2));
   x3 = x(T(i,3));
   y1 = y(T(i,1));
   y2 = y(T(i,2));
   y3 = y(T(i,3));
   A = .5*abs(det([x1, x2, x3; y1, y2, y3; 1, 1, 1]));
   z1 = f(x1,y1);  % find values and average
   z2 = f(x2,y2);
   z3 = f(x3,y3);
   zavg = (z1 + z2 + z3)/3;
   I = I + zavg*A;   % accumulate integral
end
```

Exercises

25.1 Make up an interesting or useful convex irregular region and choose well-distributed points in it and on the boundary as a basis for a triangulation. Make sure to include a good number of both interior and boundary points. (Graph paper would be handy.) Record the coordinates of these points in the matrix V in a copy of the program `mytriangles.m` from Lecture 23. Plot the triangulation. Also create values at the vertices using a function and plot the resulting profile.

25.2 Modify the program `mythreecorners.m` to a new program `mycenters.m` that does the centerpoint method for triangles. Run the program on the region produced by `mywasher.m` with the function $f(x,y) = \frac{x+y}{x^2+y^2}$ and on the region produced by `mywedge.m` with the function $f(x,y) = \sin(x) + \sqrt{y^2}$.

Lecture 26

Gaussian Quadrature*

Exercises

26.1

26.2

Lecture 27

Numerical Differentiation

Approximating derivatives from data

Suppose that a variable y depends on another variable x, i.e. $y = f(x)$, but we only know the values of f at a finite set of points, e.g., as data from an experiment or a simulation:

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n).$$

Suppose then that we need information about the derivative of $f(x)$. One obvious idea would be to approximate $f'(x_i)$ by the **Forward Difference**:

$$f'(x_i) = y_i' \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i}.$$

This formula follows directly from the definition of the derivative in calculus. An alternative would be to use a **Backward Difference**:

$$f'(x_i) \approx \frac{y_i - y_{i-1}}{x_i - x_{i-1}}.$$

Since the errors for the forward difference and backward difference tend to have opposite signs, it would seem likely that averaging the two methods would give a better result than either alone. If the points are evenly spaced, i.e. $x_{i+1} - x_i = x_i - x_{i-1} = h$, then averaging the forward and backward differences leads to a symmetric expression called the **Central Difference**:

$$f'(x_i) = y_i' \approx \frac{y_{i+1} - y_{i-1}}{2h}.$$

Errors of approximation

We can use Taylor polynomials to derive the accuracy of the forward, backward and central difference formulas. For example the usual form of the Taylor polynomial with remainder (sometimes called Taylor's Theorem) is:

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

where c is some (unknown) number between x and $x+h$. Letting $x = x_i$, $x + h = x_{i+1}$ and solving for $f'(x_i)$ leads to:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{h}{2}f''(c).$$

Notice that the quotient in this equation is exactly the forward difference formula. Thus the error of the forward difference is $-(h/2)f''(c)$ which means it is $O(h)$. Replacing h in the above calculation

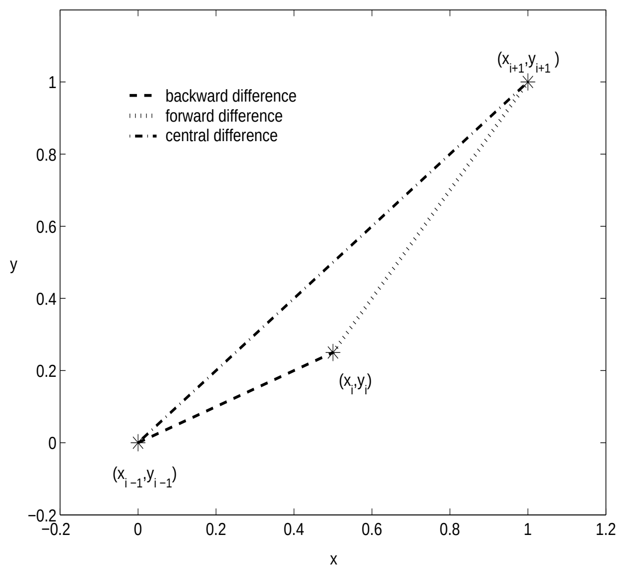

Figure 27.1: The three difference approximations of y_i'.

by $-h$ gives the error for the backward difference formula; it is also $O(h)$. For the central difference, the error can be found from the third degree Taylor polynomial with remainder:

$$f(x_{i+1}) = f(x_i + h) = f(x_i) + hf'(x_i) + \frac{h^2}{2}f''(x_i) + \frac{h^3}{3!}f'''(c_1) \quad \text{and}$$
$$f(x_{i-1}) = f(x_i - h) = f(x_i) - hf'(x_i) + \frac{h^2}{2}f''(x_i) - \frac{h^3}{3!}f'''(c_2)$$

where $x_i \le c_1 \le x_{i+1}$ and $x_{i-1} \le c_2 \le x_i$. Subtracting these two equations and solving for $f'(x_i)$ leads to:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} - \frac{h^2}{3!}\frac{f'''(c_1) + f'''(c_2)}{2}.$$

This shows that the error for the central difference formula is $O(h^2)$. Thus, central differences are significantly better and so: **It is best to use central differences whenever possible.**

There are also central difference formulas for higher order derivatives. These all have error of order $O(h^2)$:

$$f''(x_i) = y_i'' \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2},$$
$$f'''(x_i) = y_i''' \approx \frac{1}{2h^3}\left[y_{i+2} - 2y_{i+1} + 2y_{i-1} - y_{i-2}\right], \quad \text{and}$$
$$f^{(4)}(x_i) = y_i^{(4)} \approx \frac{1}{h^4}\left[y_{i+2} - 4y_{i+1} + 6y_i - 4y_{i-1} + y_{i-2}\right].$$

Partial Derivatives

Suppose $u = u(x, y)$ is a function of two variables that we only know at grid points (x_i, y_j). We will use the notation:

$$u_{i,j} = u(x_i, y_j)$$

frequently throughout the rest of the lectures. We can suppose that the grid points are evenly spaced, with an increment of h in the x direction and k in the y direction. The central difference formulas for the partial derivatives would be:

$$u_x(x_i, y_j) \approx \frac{1}{2h}\left(u_{i+1,j} - u_{i-1,j}\right) \quad \text{and}$$
$$u_y(x_i, y_j) \approx \frac{1}{2k}\left(u_{i,j+1} - u_{i,j-1}\right).$$

The second partial derivatives are:

$$u_{xx}(x_i, y_j) \approx \frac{1}{h^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right) \quad \text{and}$$
$$u_{yy}(x_i, y_j) \approx \frac{1}{k^2}\left(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right),$$

and the mixed partial derivative is:

$$u_{xy}(x_i, y_j) \approx \frac{1}{4hk}\left(u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}\right).$$

Caution: Notice that we have indexed u_{ij} so that as a matrix each row represents the values of u at a certain x_i and each column contains values at y_j. The arrangement in the matrix does not coincide with the usual orientation of the xy-plane.

Let's consider an example. Let the values of u at (x_i, y_j) be recorded in the matrix:

$$(u_{ij}) = \begin{pmatrix} 5.1 & 6.5 & 7.5 & 8.1 & 8.4 \\ 5.5 & 6.8 & 7.8 & 8.3 & 8.9 \\ 5.5 & 6.9 & 9.0 & 8.4 & 9.1 \\ 5.4 & 9.6 & 9.1 & 8.6 & 9.4 \end{pmatrix} \tag{27.1}$$

Assume the indices begin at 1, i is the index for rows and j the index for columns. Suppose that $h = .5$ and $k = .2$. Then $u_y(x_2, y_4)$ would be approximated by the central difference:

$$u_y(x_2, y_4) \approx \frac{u_{2,5} - u_{2,3}}{2k} \approx \frac{8.9 - 7.8}{2 \cdot 0.2} = 2.75.$$

The partial derivative $u_{xy}(x_2, y_4)$ is approximated by:

$$u_{xy}(x_2, y_4) \approx \frac{u_{3,5} - u_{3,3} - u_{1,5} + u_{1,3}}{4hk} \approx \frac{9.1 - 9.0 - 8.4 + 7.5}{4 \cdot .5 \cdot .2} = -2.$$

Exercises

27.1 Suppose you are given the data in the following table.

t	0	.5	1.0	1.5	2.0
y	0	.19	.26	.29	.31

a. Give the forward, backward and central difference approximations of $f'(1)$.
b. Give the central difference approximations for $f''(1)$, $f'''(1)$ and $f^{(4)}(1)$.

27.2 Suppose values of $u(x, y)$ at points (x_i, y_j) are given in the matrix (27.1). Suppose that $h = .1$ and $k = .5$. Approximate the following derivatives by central differences:
a. $u_x(x_2, y_4)$
b. $u_{xx}(x_3, y_2)$
c. $u_{yy}(x_3, y_2)$
d. $u_{xy}(x_2, y_3)$

Lecture 28

The Main Sources of Error

Truncation Error

Truncation error is defined as the error caused directly by an approximation method. For instance, all numerical integration methods are approximations and so there is error, even if the calculations are performed exactly. Numerical differentiation also has a truncation error, as will the differential equations methods we will study in Part IV, which are based on numerical differentiation formulas. There are two ways to minimize truncation error: (1) use a higher order method, and (2) use a finer grid so that points are closer together. Unless the grid is very small, truncation errors are usually much larger than roundoff errors. The obvious tradeoff is that a smaller grid requires more calculations, which in turn produces more roundoff errors and requires more running time.

Roundoff Error

Roundoff error always occurs when a finite number of digits are recorded after an operation. Fortunately, this error is extremely small. The standard measure of how small is called **machine epsilon**. It is defined as the smallest number that can be added to 1 to produce another number on the machine, i.e. if a smaller number is added the result will be rounded down to 1. In IEEE standard double precision (used by MATLAB and most serious software), machine epsilon is 2^{-52} or about 2.2×10^{-16}. A different, but equivalent, way of thinking about this is that the machine records 52 floating binary digits or about 15 floating decimal digits. Thus there are never more than 15 significant digits in any calculation. This of course is more than adequate for any application. However, there are ways in which this very small error can cause problems.

To see an unexpected occurence of round-off try the following commands:

```
 > (2^52+1) - 2^52
 > (2^53+1) - 2^53
```

Loss of Precision

One way in which small roundoff errors can become bigger is by multiplying the results by a large number (or dividing by a small number). For instance if you were to calculate:

$$1234567(1 - .9999987654321)$$

then the result should be

$$1.52415678859930.$$

But, if you input:
```
 > 1234567 * ( 1 - .9999987654321)
```
you will get:
```
 > 1.52415678862573
```
In the correct calculation there are 15 significant digits, but the MATLAB calculation has only 11 significant digits.

Although this seems like a silly example, this type of *loss of precision* can happen by accident if you are not careful. For example in $f'(x) \approx (f(x+h) - f(x))/h$ you will lose precision when h gets too small. Try:

```
> format long
> f=inline('x^2','x')
> for i = 1:30
>    h=10^(-i)
>    df=(f(1+h)-f(1))/h
>    relerr=(2-df)/2
> end
```

At first the relative error decreases since truncation error is reduced. Then loss of precision takes over and the relative error increases to 1.

Bad Conditioning

We encountered bad conditioning in Part II, when we talked about solving linear systems. Bad conditioning means that the problem is unstable in the sense that small input errors can produce large output errors. This can be a problem in a couple of ways. First, the measurements used to get the inputs cannot be completely accurate. Second, the computations along the way have roundoff errors. Errors in the computations near the beginning especially can be magnified by a factor close to the condition number of the matrix. Thus what was a very small problem with roundoff can become a very big problem.

It turns out that matrix equations are not the only place where condition numbers occur. In any problem one can define the condition number as the maximum ratio of the relative errors in the output versus input, i.e.

$$\text{condition \# of a problem} = \max\left(\frac{\text{Relative error of output}}{\text{Relative error of inputs}}\right).$$

An easy example is solving a simple equation:

$$f(x) = 0.$$

Suppose that f' is close to zero at the solution x^*. Then a very small change in f (caused perhaps by an inaccurate measurement of some of the coefficients in f) can cause a large change in x^*. It can be shown that the condition number of this problem is $1/f'(x^*)$.

Exercises

28.1 [1] The function $f(x) = (x-2)^9$ could be evaluated as written, or first expanded as $f(x) = x^9 + 18x^8 + \cdots$ and then evaluated. To find the expanded version, type

```
> syms x
> expand((x-2)^9)
> clear
```

To evaluate it the first way, type

```
> f1=inline('(x-2).^9','x')
> x=1.92:.001:2.08;
> y1=f1(x);
> plot(x,y1,'blue')
```

To do it the second way, convert the expansion above to an inline function `f2` and then type

```
> y2=f2(x);
> hold on
> plot(x,y2,'red')
```

Carefully study the resulting graphs. Should the graphs be the same? Which is more correct? MATLAB does calculations using approximately 16 decimal places. What is the largest error in the graph, and how big is it relative to 10^{-16}? Which source of error is causing this problem?

[1]From *Numerical Linear Algebra* by L. Trefethen and D. Baum, SIAM, 1997.

Review of Part III

Methods and Formulas

Polynomial Interpolation:
An exact fit to the data.
For n data points it is a $n-1$ degree polynomial.
Only good for very few, accurate data points.
The coefficients are found by solving a linear system of equations.

Spline Interpolation:
Fit a simple function between each pair of points.
Joining points by line segments is the most simple spline.
Cubic is by far the most common and important.
Cubic matches derivatives and second derivatives at data points.
Simply supported and clamped ends are available.
Good for more, but accurate points.
The coefficients are found by solving a linear system of equations.

Least Squares:
Makes a "close fit" of a simple function to all the data.
Minimizes the sum of the squares of the errors.
Good for noisy data.
The coefficients are found by solving a linear system of equations.

Interpolation vs. Extrapolation: Polynomials, Splines and Least Squares are generally used for *Interpolation*, fitting between the data. *Extrapolation*, i.e. making fits beyond the data, is much more tricky. To make predictions beyond the data, you must have knowledge of the underlying process, i.e. what the function should be.

Numerical Integration:
Left Endpoint:

$$L_n = \sum_{i=1}^{n} f(x_{i-1})\Delta x_i$$

Right Endpoint:

$$R_n = \sum_{i=1}^{n} f(x_i)\Delta x_i.$$

Trapezoid Rule:

$$T_n = \sum_{i=1}^{n} \frac{f(x_{i-1}) + f(x_i)}{2}\Delta x_i.$$

Midpoint Rule:

$$M_n = \sum_{i=1}^{n} f(\bar{x}_i)\Delta x_i \qquad \text{where} \qquad \bar{x}_i = \frac{x_{i-1} + x_i}{2}.$$

Numerical Integration Rules with Even Spacing:
For even spacing: $\Delta x = \frac{b-a}{n}$ where n is the number of subintervals, then:

$$L_n = \Delta x \sum_{i=0}^{n-1} y_i = \frac{b-a}{n} \sum_{i=0}^{n-1} f(x_i),$$

$$R_n = \Delta x \sum_{i=1}^{n} y_i = \frac{b-a}{n} \sum_{i=1}^{n} f(x_i),$$

$$T_n = \Delta x\big(y_0 + 2y_1 + \ldots + 2y_{n-1} + y_n\big) = \frac{b-a}{2n}\big(f(x_0) + 2f(x_1) + \ldots + 2f(x_{n-1}) + f(x_n)\big),$$

$$M_n = \Delta x \sum_{i=1}^{n} \bar{y}_i = \frac{b-a}{n} \sum_{i=1}^{n} f(\bar{x}_i),$$

Simpson's rule:

$$\begin{aligned} S_n &= \Delta x\big(y_0 + 4y_1 + 2y_2 + 4y_3 + \ldots + 2y_{n-2} + 4y_{n-1} + y_n\big) \\ &= \frac{b-a}{3n}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\right). \end{aligned}$$

Area of a region:

$$A = -\oint_C y\,dx,$$

where C is the counter-clockwise curve around the boundary of the region. We can represent such a curve, by consecutive points on it, i.e. $\bar{x} = (x_0, x_1, x_2, \ldots, x_{n-1}, x_n)$, and $\bar{y} = (y_0, y_1, y_2, \ldots, y_{n-1}, y_n)$ with $(x_n, y_n) = (x_0, y_0)$. Applying the trapezoid method to the integral (21.4):

$$A = -\sum_{i=1}^{n} \frac{y_{i-1} + y_i}{2}\left(x_i - x_{i-1}\right)$$

Accuracy of integration rules:
Right and Left endpoint are $O(\Delta)$
Trapezoid and Midpoint are $O(\Delta^2)$
Simpson is $O(\Delta^4)$

Double Integrals on Rectangles:
Centerpoint:

$$C_{mn} = \sum_{i,j=1,1}^{m,n} f(c_{ij}) A_{ij}.$$

where

$$c_{ij} = \left(\frac{x_{i-1} + x_i}{2}, \frac{y_{i-1} + y_i}{2}\right).$$

Centerpoint – Evenly spaced:

$$C_{mn} = \Delta x \Delta y \sum_{i,j=1,1}^{m,n} z_{ij} = \frac{(b-a)(d-c)}{mn} \sum_{i,j=1,1}^{m,n} f(c_{ij}).$$

Four corners:

$$F_{mn} = \sum_{i,j=1,1}^{m,n} \frac{1}{4}\left(f(x_i, y_j) + f(x_i, y_{j+1}) + f(x_{i+1}, y_i) + f(x_{i+1}, y_{j+1})\right) A_{ij},$$

Four Corners – Evenly spaced:

$$\begin{aligned} F_{mn} &= \frac{A}{4}\left(\sum_{\text{corners}} f(x_i,y_j) + 2\sum_{\text{edges}} f(x_i,y_j) + 4\sum_{\text{interior}} f(x_i,y_j)\right) \\ &= \frac{(b-a)(d-c)}{4mn}\sum_{i,j=1,1}^{m,n} W_{ij} f(x_i,y_j). \end{aligned}$$

where

$$W = \begin{pmatrix} 1 & 2 & 2 & 2 & \cdots & 2 & 2 & 1 \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ 1 & 2 & 2 & 2 & \cdots & 2 & 2 & 1 \end{pmatrix}.$$

Double Simpson:

$$S_{mn} = \frac{(b-a)(d-c)}{9mn}\sum_{i,j=1,1}^{m,n} W_{ij} f(x_i,y_j).$$

where

$$W = \begin{pmatrix} 1 & 4 & 2 & 4 & \cdots & 2 & 4 & 1 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ 2 & 8 & 4 & 8 & \cdots & 4 & 8 & 2 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 2 & 8 & 4 & 8 & \cdots & 4 & 8 & 2 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ 1 & 4 & 2 & 4 & \cdots & 2 & 4 & 1 \end{pmatrix}.$$

Integration based on triangles:
– Triangulation: Dividing a region up into triangles.
– Triangles are suitable for odd-shaped regions.
– A triangulation is better if the triangles are nearly equilateral.
Three corners:

$$T_n = \sum_{i=1}^{n} \bar{f}_i A_i$$

where $\bar{f}$ is the average of f at the corners of the i-th triangle.
Area of a triangle with corners (x_1,y_1), (x_2,y_2), (x_3,y_3):

$$A = \frac{1}{2}\left|\det\begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix}\right|.$$

Centerpoint:

$$C = \sum_{i=1}^{n} f(\bar{x}_i,\bar{y}_i)A_i, \quad \text{with}$$

$$\bar{x} = \frac{x_1+x_2+x_3}{3} \quad \text{and} \quad \bar{y} = \frac{y_1+y_2+y_3}{3}.$$

Forward Difference:
$$f'(x_i) = y'_i \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i}.$$

Backward Difference:
$$f'(x_i) \approx \frac{y_i - y_{i-1}}{x_i - x_{i-1}}.$$

Central Difference:
$$f'(x_i) = y'_i \approx \frac{y_{i+1} - y_{i-1}}{2h}.$$

Higher order central differences:
$$\begin{aligned}
f''(x_i) = y''_i &\approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}, \\
f'''(x_i) = y'''_i &\approx \frac{1}{2h^3}\left[y_{i+2} - 2y_{i+1} + 2y_{i-1} - y_{i-2}\right], \\
f^{(4)}(x_i) = y^{(4)}_i &\approx \frac{1}{h^4}\left[y_{i+2} - 4y_{i+1} + 6y_i - 4y_{i-1} + y_{i-2}\right].
\end{aligned}$$

Partial Derivatives: Denote $u_{i,j} = u(x_i, y_j)$.
$$\begin{aligned}
u_x(x_i, y_j) &\approx \frac{1}{2h}\left(u_{i+1,j} - u_{i-1,j}\right), \\
u_y(x_i, y_j) &\approx \frac{1}{2k}\left(u_{i,j+1} - u_{i,j-1}\right), \\
u_{xx}(x_i, y_j) &\approx \frac{1}{h^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right), \\
u_{yy}(x_i, y_j) &\approx \frac{1}{k^2}\left(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right), \\
u_{xy}(x_i, y_j) &\approx \frac{1}{4hk}\left(u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}\right).
\end{aligned}$$

Sources of error:
Truncation – the method is an approximation.
Roundoff – double precision arithmetic uses ≈ 15 significant digits.
Loss of precision – an amplification of roundoff error due to cancellations.
Bad conditioning – the problem is sensitive to input errors.
Error can build up after multiple calculations.

Matlab

Data Interpolation:
Use the plot command `plot(x,y,'*')` to plot the data.
Use the Basic Fitting tool to make an interpolation or spline.
If you choose an $n-1$ degree polynomial with n data points the result will be the exact polynomial interpolation.
If you select a polynomial degree less than $n-1$, then MATLAB will produce a least squares approximation.

Integration
`> quadl(f,a,b)` .... Numerical integral of $f(x)$ on $[a, b]$.

`> dblquad(f,a,b,c,d)` .... Integral of $f(x, y)$ on $[a, b] \times [c, d]$.
Example:

```
> f = inline('sin(x.*y)/sqrt(x+y)','x','y')
> I = dblquad(f,0,1,0,2)
```

Matlab uses an advanced form of Simpson's method.

Integration over non-rectangles:
Redefine the function to be zero outside the region. For example:

```
> f = inline('sin(x.*y).^3.*(2*x + y <= 2)')
> I = dblquad(f,0,1,0,2)
```

Integrates $f(x, y) = \sin^3(xy)$ on the triangle with corners $(0, 0)$, $(0, 2)$, and $(1, 0)$.

Triangles:
Matlab stores triangulations as a matrix of vertices `V` and triangles `T`.

`> T = delaunay(V)` .... Produces triangles from vertices.

```
> trimesh(T,x,y)
> trimesh(T,x,y,z)
> trisurf(T,x,y)
> trisurf(T,x,y,z)
```

Part IV
Differential Equations

Lecture 29

Reduction of Higher Order Equations to Systems

The motion of a pendulum

Consider the motion of an ideal pendulum that consists of a mass m attached to an arm of length ℓ. If we ignore friction, then Newton's laws of motion tell us:

$$m\ddot{\theta} = -\frac{mg}{\ell}\sin\theta,$$

where θ is the angle of displacement.

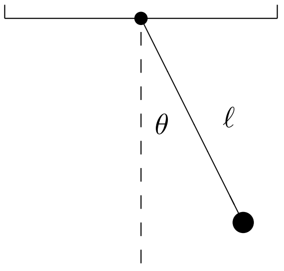

Figure 29.1: A pendulum.

If we also incorporate moving friction and sinusoidal forcing then the equation takes the form:

$$m\ddot{\theta} + \gamma\dot{\theta} + \frac{mg}{\ell}\sin\theta = A\sin\Omega t.$$

Here γ is the coefficient of friction and A and Ω are the amplitude and frequency of the forcing. Usually, this equation would be rewritten by dividing through by m to produce:

$$\ddot{\theta} + c\dot{\theta} + \omega\sin\theta = a\sin\Omega t, \tag{29.1}$$

where $c = \gamma/m$. $\omega = g/\ell$ and $a = A/m$.

This is a second order ODE because the second derivative with respect to time t is the highest derivative. It is nonlinear because it has the term $\sin\theta$ and which is a nonlinear function of the dependent variable θ. A solution of the equation would be a function $\theta(t)$. To get a specific solution we need side conditions. Because it is second order, 2 conditions are needed, and the usual conditions are initial conditions:

$$\theta(0) = \theta_0 \qquad \text{and} \qquad \dot{\theta}(0) = v_0. \tag{29.2}$$

Converting a general higher order equation

All of the standard methods for solving ordinary differential equations are intended for first order equations. For this reason, it is inconvenient to solve higher order equations numerically. However, most higher-order differential equations that occur in applications can be converted to a *system* of first order equations and that is what is usually done in practice.

Suppose that an n-th order equation can be solved for the nth derivative, i.e. it can be written in the form:

$$x^{(n)} = f\left(t, x, \dot{x}, \ddot{x}, \ldots, \frac{d^{n-1}x}{dt^{n-1}}\right), \tag{29.3}$$

then it can be converted to a first-order system by this standard change of variables:

$$\begin{aligned} y_1 &= x \\ y_2 &= \dot{x} \\ &\vdots \\ y_n &= x^{(n-1)} = \frac{d^{n-1}x}{dt^{n-1}}. \end{aligned} \tag{29.4}$$

The resulting first-order system is the following:

$$\begin{aligned} \dot{y}_1 &= \dot{x} = y_2 \\ \dot{y}_2 &= \ddot{x} = y_3 \\ &\vdots \\ \dot{y}_n &= x^{(n)} = f(t, y_1, y_2, \ldots, y_n). \end{aligned} \tag{29.5}$$

For the example of the pendulum (29.1) the change of variables has the form

$$\begin{aligned} y_1 &= \theta \\ y_2 &= \dot{\theta}, \end{aligned} \tag{29.6}$$

and the resulting equations are:

$$\begin{aligned} \dot{y}_1 &= y_2 \\ \dot{y}_2 &= -cy_2 - \omega \sin(y_1) + a \sin(\Omega t). \end{aligned} \tag{29.7}$$

In vertor form this is

$$\dot{\mathbf{y}} = \begin{pmatrix} y_2 \\ -cy_2 - \omega \sin(y_1) + a \sin(\Omega t) \end{pmatrix}.$$

The initial conditions are converted to:

$$\mathbf{y}(0) = \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} \theta_0 \\ v_0 \end{pmatrix}. \tag{29.8}$$

As stated above, the main reason we wish to change a higher order equation into a system of equations is that this form is convenient for solving the equation numerically. Most general software for solving ODEs (including MATLAB) requires that the ODE be input in the form of a first-order system. In addition, there is a conceptual reason to make the change. In a system described by a higher order equation, knowing the position is not enough to know what the system is doing. In the case of a second order equation, such as the pendulum, one must know both the angle and the angular velocity to know what the pendulum is really doing. We call the pair $(\theta, \dot{\theta})$ the *state* of the system. Generally in applications the vector $\mathbf{y}$ is the state of the system described by the differential equation.

Using Matlab to solve a system of ODE's

In Matlab there are several commands that can be used to solve an initial value problem for a system of differential equations. Each of these correspond to different solving methods. The standard one to use is `ode45`, which uses the algorithm "Runga-Kutta 4 5". We will learn about this algorithm later.

To implement `ode45` for a system, we have to input the vector function f that defines the system. For the pendulum system (29.7), we will assume that all the constants are 1, except $c = .1$, then we can input the right hand side as:

```
> f = inline('[y(2);-.1*y(2)-sin(y(1))+sin(t)]','t','y')
```

The command `ode45` is then used as follows:

```
> [T Y] = ode45(f,[0 20],[1;-1.5]);
```

Here `[0 20]` is the time span you want to consider and `[1;-1.5]` is the initial value of the vector `y`. The output `T` contains times and `Y` contains values of the vector $\mathbf{y}$ at those times. Try:

```
> size(T)
> T(1:10)
> size(Y)
> Y(1:10,:)
```

Since the first coordinate of the vector is the position (angle), we are mainly interested in its values:

```
> theta = Y(:,1);
> plot(T,theta)
```

In the next two sections we will learn enough about numerical methods for initial value problems to understand roughly how Matlab produces this approximate solution.

Exercises

29.1 Transform the given ODE into a first order system:

$$\dddot{x} + \ddot{x}^2 - 3\dot{x}^3 + \cos^2 x = e^{-t}\sin(3t).$$

Suppose the initial conditions for the ODE are: $x(1) = 1$, $\dot{x}(1) = 2$, and $\ddot{x}(1) = 0$. Find a numerical solution of this IVP using `ode45` and plot the first coordinate (x). Try time intervals `[1 2]` and `[1 2.1]` and explain what you observe. (Remember to use entry-wise operations, `.*` and `.^`, in the definition of the vector function.)

Lecture 30

Euler Methods

Numerical Solution of an IVP

Suppose we wish to numerically solve the initial value problem:

$$\dot{\mathbf{y}} = f(t, \mathbf{y}), \qquad \mathbf{y}(a) = \mathbf{y}_0, \tag{30.1}$$

on an interval of time $[a, b]$.

By a numerical solution, we must mean an approximation of the solution at a finite number of points, i.e.:

$$(t_0, \mathbf{y}_0), (t_1, \mathbf{y}_1), (t_2, \mathbf{y}_2), \ldots, (t_n, \mathbf{y}_n),$$

where $t_0 = a$ and $t_n = b$. The first of these points is exactly the initial value. If we take n steps as above, and the steps are evenly spaced, then the time change in each step is:

$$h = \frac{b-a}{n}, \tag{30.2}$$

and the times t_i are given simply by $t_i = a + ih$. This leaves the most important part of finding a numerical solution; determining $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n$ in a way that is as consistent as possible with (30.1). To do this, first write the differential equation in the indexed notation:

$$\dot{\mathbf{y}}_i \approx f(t_i, \mathbf{y}_i), \tag{30.3}$$

and will then replace the derivative $\dot{\mathbf{y}}$ by a difference. There are many ways we might carry this out and in the next section we study the simplest.

The Euler Method

The most straight forward approach is to replace $\dot{\mathbf{y}}_i$ in (30.3) by its forward difference approximation. This gives:

$$\frac{\mathbf{y}_{i+1} - \mathbf{y}_i}{h} = f(t_i, \mathbf{y}_i).$$

Rearranging this gives us a way to obtain $\mathbf{y}_{i+1}$ from $\mathbf{y}_i$ known as Euler's method:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + hf(t_i, \mathbf{y}_i). \tag{30.4}$$

With this formula, we can start from $(t_0, \mathbf{y}_0)$ and compute all the subsequent approximations $(t_i, \mathbf{y}_i)$. This is very easy to implement, as you can see from the following program (which can be downloaded from the class web site):

```
function [T , Y] = myeuler(f,tspan,y0,n)
% function [T , Y] = myeuler(f,tspan,y0,n)
% Solves dy/dt = f(t,y) with initial condition y(a) = y0
% on the interval [a,b] using n steps of Euler s method.
% Inputs: f -- a function f(t,y) that returns a column vector of the same
%              length as y
%         tspan -- a vector [a,b] with the start and end times
%         y0 -- a column vector of the initial values, y(a) = y0
%         n  -- number of steps to use
% Outputs: T -- a n+1 column vector contianing the times
%          Y -- a (n+1) by d matrix where d is the length of y
%               Y(t,i) gives the ith component of y at time t
% parse starting and ending points
a = tspan(1);
b = tspan(2);
h = (b-a)/n;        % step size
t = a; y = y0;      % t and y are the current variables
T = a; Y = y0';      % T and Y will record all steps
for i = 1:n
    y = y + h*f(t,y);
    t = a + i*h;
    T = [T; t];
    Y = [Y; y'];
end
```

To use this program we need a function, such as the vector function for the pendulum:

```
> f = inline('[y(2);-.1*y(2)-sin(y(1)) + sin(t)]','t','y')
```

Then type:

```
> [T Y] = myeuler(f,[0 20],[1;-1.5],5);
```

Here `[0 20]` is the time span you want to consider, `[1;-1.5]` is the initial value of the vector `y` and `5` is the number of steps. The output `T` contains times and `Y` contains values of the vector as the times. Try:

```
> size(T)
> size(Y)
```

Since the first coordinate of the vector is the angle, we only plot its values:

```
> theta = Y(:,1);
> plot(T,theta)
```

In this plot it is clear that $n = 5$ is not adequate to represent the function. Type:

```
> hold on
```

then redo the above with 5 replaced by 10. Next try 20, 40, 80, and 200. As you can see the graph becomes increasingly better as n increases. We can compare these calculations with MATLAB's built-in function with the commands:

```
> [T Y]= ode45(f,[0 20],[1;-1.5]);
> theta = Y(:,1);
> plot(T,theta,'r')
```

The problem with the Euler method

You can think of the Euler method as finding a linear approximate solution to the initial value problem on each time interval. An obvious shortcoming of the method is that it makes the approximation based on information at the beginning of the time interval only. This problem is illustrated well by the following IVP:

$$\ddot{x} + x = 0, \qquad x(0) = 1, \quad \dot{x}(0) = 0 \tag{30.5}$$

You can easily check that the exact solution of this IVP is

$$x(t) = \cos(t).$$

If we make the standard change of variables:

$$y_1 = x, \qquad y_2 = \dot{x},$$

we get:

$$\dot{y}_1 = y_2, \qquad \dot{y}_2 = -y_1.$$

Then the solution should be $y_1(t) = \cos(t)$, $y_2(t) = \sin(t)$. If we then plot the solution in the y_1–y_2 plane, we should get exactly a unit circle. We can solve this IVP with Euler's method:

```
 > g = inline('[y(2);-y(1)]','t','y')
 > [T Y] = myeuler(g,[0 4*pi],[1;0],20)
 > y1 = Y(:,1);
 > y2 = Y(:,2);
 > plot(y1,y2)
```

As you can see the approximate solution goes far from the true solution. Even if you increase the number of steps, the Euler solution will eventually drift outward away from the circle because it does not take into account the curvature of the solution.

The Modified Euler Method

An idea which is similar to the idea behind the trapezoid method would be to consider f at both the beginning and end of the time step and take the average of the two. Doing this produces the Modified (or Improved) Euler method represented by the following equations:

$$\begin{aligned} \mathbf{k}_1 &= hf(t_i, \mathbf{y}_i) \\ \mathbf{k}_2 &= hf(t_i + h, \mathbf{y}_i + \mathbf{k}_1) \\ \mathbf{y}_{i+1} &= \mathbf{y}_i + \frac{1}{2}(\mathbf{k}_1 + \mathbf{k}_2) \end{aligned} \tag{30.6}$$

Here $\mathbf{k}_1$ captures the information at the beginning of the time step (same as Euler), while $\mathbf{k}_2$ is the information at the end of the time step.

The following program implements the Modified method. It may be downloaded from the class web site.

```
function [T , Y] = mymodeuler(f,tspan,y0,n)
% Solves dy/dt = f(t,y) with initial condition y(a) = y0
% on the interval [a,b] using n steps of the modified Euler's method.
% Inputs: f -- a function f(t,y) that returns a column vector of the same
%               length as y
%         tspan -- a vector [a,b] with the start and end times
%         y0 -- a column vector of the initial values, y(a) = y0
%         n  -- number of steps to use
% Outputs: T -- a n+1 column vector contianing the times
%          Y -- a (n+1) by d matrix where d is the length of y
%                Y(t,i) gives the ith component of y at time t
% parse starting and ending points
a = tspan(1);
b = tspan(2);
h = (b-a)/n;        % step size
t = a; y = y0;      % t and y are the current variables
T = a; Y = y0';      % T and Y will record all steps
for i = 1:n
    k1 = h*f(t,y);
    k2 = h*f(t+h,y+k1);
    y = y + .5*(k1+k2);
    t = a + i*h;
    T = [T; t];
    Y = [Y; y'];
end
```

We can test this program on the IVP above:

```
 > [T Ym] = mymodeuler(g,[0 4*pi],[1;0],20)
 > ym1 = Ym(:,1);
 > ym2 = Ym(:,2);
 > plot(ym1,ym2)
```

Exercises

30.1 Download the files `myeuler.m` and `mymodeuler.m` from the class web site.

1. Type the following commands:
   ```
    > hold on
    > f = inline('sin(t)*cos(x)','t','x')
    > [T Y]=myeuler(f,[0,12],.1,10);
    > plot(T,Y)
   ```
   Position the plot window so that it can always be seen and type:
   ```
    > [T Y]=myeuler(f,[0,12],.1,20);
    > plot(T,Y)
   ```
   Continue to increase the last number in the above until the graph stops changing (as far as you can see). Record this number and print the final graph. Type `hold off` and kill the plot window.
2. Follow the same procedure using `mymodeuler.m` .
3. Describe what you observed. In particular compare how fast the two methods converge as n is increased (h is decreased).

Lecture 31

Higher Order Methods

The order of a method

For numerical solutions of an initial value problem there are two ways to measure the error. The first is the error of each step. This is called the Local Truncation Error or LTE. The other is the total error for the whole interval $[a, b]$. We call this the Global Truncation Error or GTE.

For the Euler method the LTE is of order $O(h^2)$, i.e. the error is comparable to h^2. We can show this directly using Taylor's Theorem:

$$\mathbf{y}(t+h) = \mathbf{y}(t) + h\dot{\mathbf{y}}(t) + \frac{h^2}{2}\ddot{\mathbf{y}}(c)$$

for some c between t and $t+h$. In this equation we can replace $\dot{\mathbf{y}}(t)$ by $f(t, \mathbf{y}(t))$ which make the first two terms of the right hand side be exactly the Euler method. The error is then $\frac{h^2}{2}\ddot{\mathbf{y}}(c)$ or $O(h^2)$. It would be slightly more difficult to show that the LTE of the modified Euler method is $O(h^3)$, an improvement of one power of h.

We can roughly get the GTE from the LTE by considering the number of steps times the LTE. For any method, if $[a, b]$ is the interval and h is the step size, then $n = (b-a)/h$ is the number of steps. Thus for any method, the GTE is one power lower in h than the LTE. Thus the GTE for Euler is $O(h)$ and for modified Euler it is $O(h^2)$.

By the **order** of a method, we mean the power of h in the GTE. Thus the Euler method is a 1st order method and modified Euler is a 2nd order method.

Fourth Order Runga-Kutta

The most famous of all IVP methods is the classic Runga-Kutta method of order 4:

$$\begin{aligned}
\mathbf{k}_1 &= hf(t_i, \mathbf{y}) \\
\mathbf{k}_2 &= hf(t_i + h/2, \mathbf{y}_i + \mathbf{k}_1/2) \\
\mathbf{k}_3 &= hf(t_i + h/2, \mathbf{y}_i + \mathbf{k}_2/2) \\
\mathbf{k}_4 &= hf(t_i + h, \mathbf{y}_i + \mathbf{k}_3) \\
\mathbf{y}_{i+1} &= \mathbf{y}_i + \frac{1}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right).
\end{aligned} \tag{31.1}$$

Notice that this method uses values of $f(t, \mathbf{y})$ at 4 different points. In general a method needs n values of f to achieve order n. The constants used in this method and other methods are obtained from Taylor's Theorem. They are precisely the values needed to make all error terms cancel up to $h^{n+1}f^{(n+1)}(c)/(n+1)!$.

Variable Step Size and RK45

If the order of a method is n, then the GTE is comparable to h^n, which means it is approximately Ch^n, where C is some constant. However, for different differential equations, the values of C may be very different. Thus it is not easy beforehand to tell how big h should be to get the error within a given tolerence. For instance, if the true solution oscillates very rapidly, we will obviously need a smaller step size than for a solution that is nearly constant.

How can a program then choose h small enough to produce the required accuracy? We also do not wish to make h much smaller than necessary, since that would increase the number of steps. To accomplish this a program tries an h and tests to see if that h is small enough. If not it tries again with a smaller h. If it is too small, it accepts that step, but on the next step it tries a larger h. This process is called **variable step size**.

Deciding if a single step is accurate enough could be accomplished in several ways, but the most common are called **embedded methods**. The Runga-Kutta 45 method, which is used in `ode45`, is an embedded method. In the RK45, the function f is evaluated at 5 different points. These are used to make a 5th order estimate $\mathbf{y}_{i+1}$. At the same time, 4 of the 5 values are used to also get a 4th order estimate. If the 4th order and 5th order estimates are close, then we can conclude that they are accurate. If there is a large discrepency, then we can conclude that they are not accurate and a smaller h should be used.

To see variable step size in action, we will define and solve two different ODEs and solve them on the same interval:

```
 > f1 = inline('[-y(2);y(1)]','t','y')
 > f2 = inline('[-5*y(2);5*y(1)]','t','y')
 > [T1 Y1] = ode45(f1,[0 20],[1;0]);
 > [T2 Y2] = ode45(f2,[0 20],[1;0]);
 > y1 = Y1(:,1);
 > y2 = Y2(:,1);
 > plot(T1,y1,'bx-')
 > hold on
 > plot(T2,y2,'ro-')
 > size(T1)
 > size(T2)
```

Why order matters

Many people would conclude on first encounter that the advantage of a higher order method would be that you can get a more accurate answer than for a lower order method. In reality, this is not quite how things work. In engineering problems, the accuracy needed is usually a given and it is usually not extremely high. Thus getting more and more accurate solutions is not very useful. So where is the advantage? Consider the following example.

Suppose that you need to solve an IVP with an error of less than 10^{-4}. If you use the Euler method, which has GTE of order $O(h)$, then you would need $h \approx 10^{-4}$. So you would need about $n \approx (b-a) \times 10^4$ steps to find the solution.

Suppose you use the second order, modified Euler method. In that case the GTE is $O(h^2)$, so you would need to use $h^2 \approx 10^{-4}$, or $h \approx 10^{-2}$. This would require about $n \approx (b-a) \times 10^2$ steps.

That is a hundred times fewer steps than you would need to get the same accuracy with the Euler method.

If you use the RK4 method, then h^4 needs to be approximately 10^{-4}, and so $h \approx 10^{-1}$. This means you need only about $n \approx (b-a) \times 10$ steps to solve the problem, i.e. a thousand times fewer steps than for the Euler method.

Thus the real advantage of higher order methods is that they can run a lot faster at the same accuracy. This can be especially important in applications where one is trying to make real-time adjustments based on the calculations. Such is often the case in robots and other applications with dynamic controls.

Exercises

31.1 There is a Runga-Kutta 2 method, which is also known as the midpoint method. It is summarized by the following equations:

$$\begin{aligned} \mathbf{k}_1 &= hf(t_i, \mathbf{y}_i) \\ \mathbf{k}_2 &= hf(t_i + h/2, \mathbf{y}_i + \mathbf{k}_1/2) \\ \mathbf{y}_{i+1} &= \mathbf{y}_i + \mathbf{k}_2. \end{aligned} \tag{31.2}$$

Modify the program `mymodeuler` into a program `myRK2` that does the RK2 method. Compare `myRK2` and `mymodeuler` on the following IVP with time span $[0, 4\pi]$:

$$\ddot{x} + x = 0, \qquad x(0) = 1, \quad \dot{x}(0) = 0.$$

Using `format long` compare the results of the two programs for $n = 10, 100, 1000$. In particular, compare the errors of $\mathbf{x}(4\pi)$, which should be $(1, 0)$.

Lecture 32

Multi-step Methods*

Exercises

32.1

Lecture 33

ODE Boundary Value Problems and Finite Differences

Steady State Heat and Diffusion

If we consider the movement of heat in a one dimensional object, it is known that the heat, $u(x,t)$, at a given location x and given time t satisfies the partial differential equation:

$$u_t - u_{xx} = g(x,t), \tag{33.1}$$

where $g(x,t)$ is the effect of any external heat source. The same equation also describes the diffusion of a chemical in a one-dimensional environment. For example the environment might be a canal, and then $g(x,t)$ would represent how a chemical is introduced.

In many applications, we would only be interested in the steady state of the system, supposing $g(x,t) = g(x)$ and $u(x,t) = u(x)$. In this case

$$u_{xx} = -g(x).$$

This is a linear second-order ordinary differential equation. We could find its solution exactly if $g(x)$ is not too complicated. If the environment or object we consider has length L, then typically one would have conditions on each end of the object, such as $u(0) = 0$, $u(L) = 0$. Thus instead of an initial value problem, we have a **boundary value problem**.

Beam With Tension

Consider a simply supported beam with modulus of elasticity E, moment af inertia I, a uniform load w, and end tension T (see Figure 33.1). If $y(x)$ denotes the deflection at each point x in the beam, then $y(x)$ satisfies the differential equation:

$$\frac{y''}{(1+(y')^2)^{3/2}} - \frac{T}{EI}y = \frac{wx(L-x)}{2EI}, \tag{33.2}$$

with boundary conditions: $y(0) = y(L) = 0$. This equation is nonlinear and there is no hope to solve it exactly. If the deflection is small then $(y')^2$ is negligible and the equation simplifies to:

$$y'' - \frac{T}{EI}y = \frac{wx(L-x)}{2EI}, \tag{33.3}$$

This is a linear equation and we can find the exact solution. We can rewrite the equation as

$$y'' - ay = bx(L-x), \tag{33.4}$$

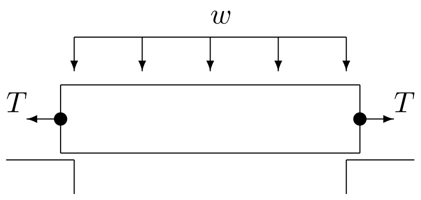

Figure 33.1: A simply supported beam with a uniform load w and end tension T.

where,

$$a = \frac{T}{EI} \qquad \text{and} \qquad b\frac{w}{2EI}, \tag{33.5}$$

the exact solution is:

$$y(x) = \frac{2b}{a^2}\frac{e^{\sqrt{a}L}}{e^{\sqrt{a}L}+1}e^{-\sqrt{a}x} + \frac{2b}{a^2}\frac{1}{e^{\sqrt{a}L}+1}e^{\sqrt{a}x} + \frac{b}{a}x^2 - \frac{bL}{a}x + \frac{2b}{a^2}. \tag{33.6}$$

Finite Difference Method – Linear ODE

A finite difference equation is an equation obtained from a differential equation by replacing the variables by their discrete versions and derivatives by difference formulas.

First we will consider equation (33.3). Suppose that the beam is a W12x22 structural steel I-beam. Then $L = 120$ in., $E = 29 \times 10^6$lb./in.^{2} and $I = 121$in.^{4}. Suppose that the beam is carrying a load of $100,000$ lb. so that $w = 120,000/120 = 10,000$ and a tension of $T = 10,000$ lb.. For these choices we calculate from (33.5) $a = 2.850 \times 10^{-6}$ and $b = 1.425 \times 10^{-6}$. Thus we have the following BVP:

$$y'' = 2.850 \times 10^{-6} y + 1.425 \times 10^{-6} x(120 - x), \qquad y(0) = y(120) = 0. \tag{33.7}$$

First subdivide the interval $[0, 120]$ into four equal subintervals. The nodes of this subdivion are $x_0 = 0$, $x_1 = 30$, $x_2 = 60$, …, $x_4 = 120$. We will then let $y_0, y_1, \ldots, y_4$ denote the deflections at the nodes. From the boundary conditions we have immediately:

$$y_0 = y_4 = 0.$$

To determine the deflections at the interior points we will rely on the differential equation. Recall the central difference formula:

$$y''(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}.$$

In this case we have $h = (b - a)/n = 30$. Replacing all the variables in the equation (33.4)by their discrete versions we get:

$$y_{i+1} - 2y_i + y_{i-1} = h^2 a y_i + h^2 b x_i (L - x_i).$$

Substituting in for a, b and h we obtain:

$$\begin{aligned} y_{i+1} - 2y_i + y_{i-1} &= 900 \times 2.850 \times 10^{-6} y_i + 900 \times 1.425 \times 10^{-6} x_i (120 - x_i) \\ &= 2.565 \times 10^{-3} y_i + 1.282 \times 10^{-3} x_i (120 - x_i). \end{aligned}$$

This equation makes sense for $i = 1, 2, 3$. At $x_1 = 30$, the equation becomes:

$$\begin{aligned} y_2 - 2y_1 + y_0 &= 2.565 \times 10^{-3} y_1 + 1.282 \times 10^{-3} \times 30(90) \quad \text{or} \\ y_2 - 2.002565 y_1 &= 3.463. \end{aligned} \tag{33.8}$$

Note that this equation is linear in the unknowns y_1 and y_2. At $x_2 = 2$ we have:

$$\begin{aligned} y_3 - 2y_2 + y_1 &= .002565 y_2 + 1.282 \times 10^{-3} \times 60^2 \quad \text{or} \\ y_3 - 2.002565 y_2 + y_1 &= 4.615. \end{aligned} \tag{33.9}$$

At $x_3 = 3$:

$$y_4 - 2.002565 y_3 + y_2 = 3.463. \tag{33.10}$$

Thus (y_1, y_2, y_3) is the solution of the linear system:

$$\left(\begin{array}{ccc|c} -2.002565 & 1 & 0 & 3.463 \\ 1 & -2.002565 & 1 & 4.615 \\ 0 & 1 & -2.002565 & 3.463 \end{array} \right).$$

We can easily find the solution of this system in Matlab:

```
> A = [ -2.002565 1 0 ; 1 -2.002565 1 ; 0 1 -2.002565]
> b = [ 3.463 4.615 3.463 ]'
> y = A\b
```

To graph the solution, we need define the x values and add on the values at the endpoints:

```
> x = 0:30:120
> y = [0 ; y ; 0]
> plot(x,y,'d')
```

Adding a spline will result in an excellent graph.

The exact solution of this BVP is given in (33.6). That equation, with the parameter values for the W12x22 I-beam as in the example, is in the program `myexactbeam.m` on the web site. We can plot the true solution on the same graph:

```
> hold on
> myexactbeam
```

Thus our numerical solution is extremely good considering how few subintervals we used and how very large the deflection is.

An amusing exercise is to set $T = 0$ in the program `myexactbeam.m`; the program fails becasue the exact solution is no longer valid. Also try $T = .1$ for which you will observe loss of precision. On the other hand the finite difference method works when we set $T = 0$.

Exercises

33.1 Derive the finite difference equations for the BVP (33.7) on the same domain ($[0, 120]$), but with eight subintervals and solve (using Matlab) as in the example. Plot your result, together with the exact solution (33.6) from the program `myexactbeam.m`.

33.2 Derive the finite difference equation for the BVP:

$$y'' + y' - y = x \qquad y(0) = y(1) = 0$$

with 4 subintervals. Solve them and plot the solution using Matlab.

Lecture 34

Finite Difference Method – Nonlinear ODE

Heat conduction with radiation

If we again consider the heat in a bar of length L, but this time we also consider the effect of radiation as well as conduction, then the steady state equation has the form:

$$u_{xx} - d(u^4 - u_b^4) = -g(x), \tag{34.1}$$

where u_b is the temperature of the background, d incorporates a coefficient of radiation and $g(x)$ is the heat source.

If we again replace the continuous problem by its discrete approximation then we get:

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} - d(u_i^4 - u_b^4) = -g_i = -g(x_i). \tag{34.2}$$

This equation is nonlinear in the unknowns, thus we no longer have a system of linear equations to solve, but a system of nonlinear equations. One way to solve these equations would be by the multivariable Newton method. Instead, we introduce another interative method, known as the *relaxation* method.

Relaxation Method for Nonlinear Finite Differences

We can rewrite equation (34.2) as:

$$u_{i+1} - 2u_i + u_{i-1} = h^2 d(u_i^4 - u_b^4) - h^2 g_i.$$

From this we can solve for u_i in terms of the other quantities:

$$2u_i = u_{i+1} + u_{i-1} - h^2 d(u_i^4 - u_b^4) + h^2 g_i.$$

Next we add u_i to both sides of the equation:

$$3u_i = u_{i+1} + u_i + u_{i-1} - h^2 d(u_i^4 - u_b^4) + h^2 g_i.$$

and then divide by 3 to get the equation:

$$u_i = \frac{1}{3}\left(u_{i+1} + u_i + u_{i-1}\right) - \frac{h^2}{3}\left(d(u_i^4 - u_b^4) + g_i\right).$$

Now for the main idea. We will begin with an initial guess for the value of u_i for each i, which we can represent as a vector $\mathbf{u}^0$. Then we will use the above equation to get better estimates, $\mathbf{u}^1$, $\mathbf{u}^2$, ..., and hope that they converge to the correct answer.

If we let

$$\mathbf{u}^j = (u_0^j, u_1^j, u_2^j, \ldots, u_{n-1}^j, u_n^j)$$

denote the jth approximation, then we can obtain that $j+1$st from the formula:

$$u_i^{j+1} = \frac{1}{3}\left(u_{i+1}^j + u_i^j + u_{i-1}^j\right) - \frac{h^2}{3}\left(d((u_i^j)^4 - u_b^4) + g_i\right).$$

Notice that g_i and u_b do not change. In the resulting equation, we have u_i at each successive step depending on its previous value and the equation itself.

Boundary Conditions

If we have fixed boundary conditions. i.e. $u(0) = a$, $u(L) = b$, then we will make $u_0^j = a$ and $u_n^j = b$ for all j.

If we have an insulated end, say at the right end, then we want to replace $u'(L) = 0$, by a discrete version. If we use the most accurate version, the central difference, then we should have:

$$u'(L) = u'(x_n) = \frac{u_{n+1} - u_{n-1}}{2h} = 0.$$

or more simply:

$$u_{n+1} = u_{n-1}.$$

However, u_{n+1} would represent $u(x_{n+1})$ and x_{n+1} would be $L+h$, which is outside the bar. This however is not an obstacle in a program. We can simply extend the grid to one more node, x_{n+1}, and let u_{n+1} always equal u_{n-1}. This idea is carried out in the program of the next section.

Another practical way to implement an insulated boundary is to let the grid points straddle the boundary. For example if $u'(0) = 0$, then you could let the first two grid points be at $-h/2$ and $h/2$. Then you can let

$$u_0 = u_1.$$

This will again force the central difference at $x = 0$ to be 0.

A way to think of an insulated boundary that makes sense of the point $L+h$ is to think of two bars joined end to end, where you do the mirror image of the first bar to the second bar. If you do this, then no heat will flow across the joint, which is exactly the same effect as insulating.

Implementing the Relaxation Method

In the following program we solve the finite difference equations (34.2) with the boundary conditions $u(0) = 0$ and $u'(L) = 0$. We let $L = 4$, $n = 4$, $d = 1$, and $g(x) = \sin(\pi x/4)$. Notice that the grid includes the point $L+h$. Notice that the vector u always contains the current estimate of the values of $\mathbf{u}$.

```
% mynonlinheat (lacks comments)
L = 4;    n = 4;
h = L/n;  hh = h^2/3;
u0 = 0;   ub = .5;  ub4 = ub^4;
x = 0:h:L+h;   g = sin(pi*x/4);   u = zeros(1,n+2);
steps = 4;
u(1)=u0;
for j = 1:steps
     u(2:n+1) = (u(3:n+2)+u(2:n+1)+u(1:n))/3 + hh*(-u(2:n+1).^4+ub4+g(2:n+1));
     u(n+2) = u(n);
end
plot(x,u)
```

If you run this program the result will not seem reasonable.

We can plot the initial guess by adding the command `plot(x,u)` right before the `for` loop. We can also plot successive iterations by moving the last `plot(x,u)` before the `end`. Now we can experiment and see if the iteration is converging. Try various values of *steps* and n to produce a good plot. You will notice that this method converges quite slowly. In particular, as we increase n, we need to increase *steps* like n^2.

Exercises

34.1 Modify the script program `mynonlinheat` to plot the initial guess and all intermediate approximations and add complete comments to the program. Also modify it so that the boundary conditions are

$$u_x(0) = 0 qquad u(L) = 0$$

Print the program and a plot.

34.2 The steady state temperature $u(r)$ (given in polar coordinates) of a disk subjected to a radially symmetric heat load $g(r)$ and cooled by conduction to the rim of the disk and radiation to its environment is determined by the boundary value problem:

$$\frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r} = d(u^4 - u_b^4) - g(r), \qquad (34.3)$$

$$u(R) = u_R \qquad u'(0) = 0. \qquad (34.4)$$

Here u_b is the background temperature and u_R is the temperature at the rim of the disk.

Suppose $R = 5$, $d = .1$, $u_R = u_b = 10$, $g(r) = (r-5)^2$ and derive the finite difference equations for this BVP. Then solve for u_i as we did above to get a relaxation update formula.

The class web site has a program `myheatdisk.m` that implements these equations. Notice that the equations have a singularity at $r = 0$. How does the program avoid this problem? How does the program implement $u'(0) = 0$? Run the program and print the plot.

Next change the radiation coefficient d to .01. How does this affect the final temperature at $r = 0$? How does it affect the rate of convergence of the method? Turn in both plots, the derivation, and answers to the questions.

Lecture 35

Parabolic PDEs - Explicit Method

Heat Flow and Diffusion

The conduction of heat and diffusion of a chemical happen to be modeled by the same differential equation. The reason for this is that they both involve similar processes. Heat conduction occurs when hot, fast moving molecules bump into slower molecules and transfer some of their energy. In a solid this involves moles of molecules all moving in different, nearly random ways, but the net effect is that the energy eventually spreads itself out over a larger region. The diffusion of a chemical in a gas or liquid simliarly involves large numbers of molecules moving in different, nearly random ways. These molecules eventually spread out over a larger region.

In three dimensions, the equation that governs both of these processes is the heat/diffusion equation:

$$u_t = c\Delta u$$

where c is the coefficient of conduction or diffusion, and

$$\Delta u(x, y, z) = u_{xx} + u_{yy} + u_{zz}.$$

The symbol Δ in this context is called the *Laplacian.* If there is also a heat/chemical source, then it is incorporated a function $g(x, y, z)$ in the equation as:

$$u_t = c\Delta u + g.$$

In some problems the z dimension is irrelevent, either because the object in question is very thin, or u does not change in the z direction. In this case the equation is:

$$u_t = c\Delta u = c(u_{xx} + u_{yy}).$$

Finally, in some cases only the x direction matters. In this case the equation is just

$$u_t = u_{xx}, \tag{35.1}$$

or

$$u_t = cu_{xx} + g(x), \tag{35.2}$$

if there is a heat source represented by a function $g(x)$.

In this lecture we will learn a straight-forward technique for solving (35.1) and (35.2). It is very similar to the finite difference method we used for nonlinear boundary value problem in the previous lecture.

It is worth mentioning a related equation:

$$u_t = c\Delta(u^\gamma)$$

called the porus-media equation. Here $\gamma > 1$. This equation models diffusion in a solid, but porus, material, such as sandstone or an earthen structure. We will not solve this equation numerically, but the methods introduced here would work. Many equations that involve 1 time derivative and 2 spatial derivatives are **parabolic** and the methods introduced in this lecture will work for most of them.

Explicit Method Finite Differences

There are two independent variables in the one dimensional heat/diffusion equation $u_t = cu_{xx}$, t and x, and so we have to discretize both. Since we are considering $0 \leq x \leq L$, let's subdivide $[0, L]$ into m equal subintervals, i.e. let

$$h = L/m$$

and

$$(x_0, x_1, x_2, \ldots, x_{m-1}, x_m) = (0, h, 2h, \ldots, L-h, L).$$

Similarly, if we are interested in solving the equation on an interval of time $[0, T]$, let

$$k = T/n$$

and

$$(t_0, t_1, t_2, \ldots, t_{n-1}, t_n) = (0, k, 2k, \ldots, T-k, T).$$

We will then denote the approximate solution at the grid points by:

$$u_{ij} \approx u(x_i, t_j).$$

The equation $u_t = cu_{xx}$ can then be replaced by the difference equations:

$$\frac{u_{i,j+1} - u_{i,j}}{k} = \frac{c}{h^2}(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}). \tag{35.3}$$

Here we have used the forward difference for u_t and the central difference for u_{xx}. This equation can be solved for $u_{i,j+1}$ to produce:

$$u_{i,j+1} = ru_{i-1,j} + (1-2r)u_{i,j} + ru_{i+1,j}, \tag{35.4}$$

for $1 \leq i \leq m-1$, $0 \leq j \leq n-1$, where,

$$r = \frac{ck}{h^2}. \tag{35.5}$$

The formula (35.4) allows us to calculate all the values of u at step $j+1$ using the values at step j.

Notice that $u_{i,j+1}$ depends on $u_{i,j}$, $u_{i-1,j}$ and $u_{i+1,j}$. That is u at grid point i depends on its previous value and the values of its two nearest neighbors at the previous step (see Figure 35.1).

Boundary and Initial Conditions

To solve the partial differential equation (35.1) or (35.2) we need boundary conditions. Just as in the previous section we will have to specify something about the ends of the domain, i.e. at $x = 0$ and $x = L$. Two of the possibilities are just as they were for the steady state equation, fixed boundary conditions and insulated boundary conditions. We can implement both of those just as we did for the ODE boundary value problem.

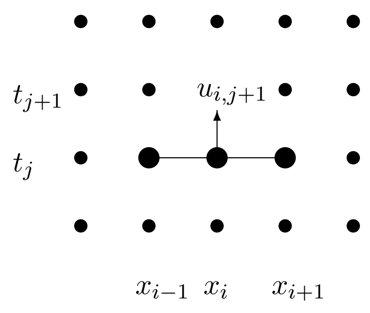

Figure 35.1: The value at grid point i depends on its previous values and the previous values of its nearest neighbors.

At third possibility is called **variable boundary conditions**. This is represented by time-dependent functions,

$$u(0,t) = g_1(t) \qquad u(L,t) = g_2(t).$$

In a heat problem, g_1 and g_2 would represent heating or cooling applied to the ends. These are easily implemented in a program by letting $u_{0,j} = g_1(t_j)$ and $u_{m,j} = g_2(t_j)$.

In addition to boundary conditions, we need an initial condition for this problem. This represents the state of the system when we begin, i.e. the initial temperature distribution or initial concentration profile. This is represented by:

$$u(x,0) = f(x).$$

To implement this in a program we let:

$$u_{i,0} = f(x_i).$$

Implementation

The following program (also available on the web page) implements the explicit method. It incorporates moving boundary conditions at both ends. To run it you must define functions f, g_1 and g_2. Notice that the main loop has only one line. The values of u are kept as a matrix. It is often convenient to define a matrix first as a matrix of the right dimension containing all zeros, and then fill in the calculated values as the program runs.

Run this program using $L = 2$, $T = 20$, $f(x) = .5x$, $g_1(t) = 0$, and $g_2(t) = cos(t)$. Note that $g1(t)$ must be input as `g1 = inline('0*t')`.

```
function [t x u] = myheat(f,g1,g2,L,T,m,n,c)
% function [t x u] = myheat(f,g1,g2,L,T,m,n,c)
% solve u_t = c u_xx  for 0<=x<=L, 0<=t<=T
%  BC:  u(0, t) = g1(t);  u(L,t) = g2(t)
%  IC:  u(x, 0) = f(x)
%  Inputs:
%  f -- inline function for IC
%  g1,g2 -- inline functions for BC
%  L -- length of rod
%  T -- length of time interval
%  m -- number of subintervals for x
%  n -- number of subintervals for t
%  c -- rate constant in equation
%  Outputs:
%  t -- vector of time points
%  x -- vector of x points
%  u -- matrix of the solution, u(i,j)~=u(x(i),t(j))

h = L/m;  k = T/n;
r = c*k/h^2;  rr = 1 - 2*r;
x = linspace(0,L,m+1);
t = linspace(0,T,n+1);

%Set up the matrix for u:
u = zeros(m+1,n+1);

% assign initial conditions
u(:,1) = f(x);

% assign boundary conditions
u(1,:) = g1(t);  u(m+1,:) = g2(t);

% find solution at remaining time steps
for j = 1:n
    u(2:m,j+1) = r*u(1:m-1,j) + rr*u(2:m,j) + r*u(3:m+1,j);
end

% plot the results
mesh(x,t,u')
```

Exercises

35.1 Modify the program `myheat.m` to have an insulated boundary at $x = L$ (rather than $u(L,t) = g_2(t)$). Run the program with $L = 2\pi$, $T = 20$, $c = .5$, $g_1(t) = \sin(t)$ and $f(x) = -\sin(x/4)$. Experiment with m and n. Get a plot when the program is stable and one when it isn't. Turn in your program and plots.

Lecture 36

Solution Instability for the Explicit Method

As we saw in experiments using `myheat.m`, the solution can become unbounded unless the time steps are small. In this lecture we consider why.

Writing the Difference Equations in Matrix Form

If we use the boundary conditions $u(0) = u(L) = 0$ then the explicit method of the previous section has the form:

$$u_{i,j+1} = ru_{i-1,j} + (1-2r)u_{i,j} + ru_{i+1,j}, \qquad 1 \le i \le m-1, \quad 0 \le j \le n-1, \tag{36.1}$$

where $u_{0,j} = 0$ and $u_{m,j} = 0$. This is equivalent to the matrix equation:

$$\mathbf{u}_{j+1} = A\mathbf{u}_j \tag{36.2}$$

where $\mathbf{u}_j$ is the column vector: $(u_{1,j}, u_{2,j}, \ldots, u_{m,j})'$ representing the state at the jth time step and A is the matrix:

$$A = \begin{pmatrix} 1-2r & r & 0 & \cdots & 0 \\ r & 1-2r & r & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & r & 1-2r & r \\ 0 & \cdots & 0 & r & 1-2r \end{pmatrix}. \tag{36.3}$$

Unfortunately, this matrix can have a property which is very bad in this context. Namely, it can cause exponential growth of error unless r is small. To see how this happens, suppose that $\mathbf{U}_j$ is the vector of correct values of u at time step t_j and $\mathbf{E}_j$ is the error of the approximation $\mathbf{u}_j$, then

$$\mathbf{u}_j = \mathbf{U}_j + \mathbf{E}_j.$$

From (36.2), the approximation at the next time step will be:

$$\mathbf{u}_{j+1} = A\mathbf{U}_j + A\mathbf{E}_j,$$

and if we continue for k steps:

$$\mathbf{u}_{j+k} = A^k\mathbf{U}_j + A^k\mathbf{E}_j.$$

The problem with this is the term $A^k\mathbf{E}_j$. This term is exactly what we would do in the power method for finding the eigenvalue of A with the largest absolute value. If the matrix A has **ew**'s with absolute value greater than 1, then this term will grow exponentially. In Figure 36.1 L is the largest absolute value of an **ew** of A as a function of the parameter r for various sizes of the matrix A. As you can see, for $r > 1/2$ the largest absolute **ew** grows rapidly for any n and quickly becomes greater than 1.

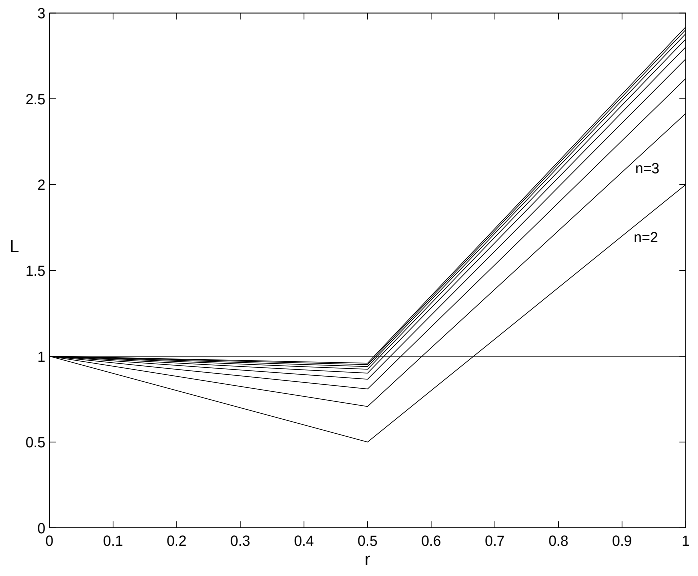

Figure 36.1: Maximum absolute eigenvalue L as a function of r for the matrix A from the explicit method for the heat equation calculated for matrices A of sizes $n = 2 \ldots 10$. Whenever $L > 1$ the method is unstable, i.e. errors grow exponentially with each step. When using the explicit method $r < 1/2$ is a safe choice.

Consequences

Recall that $r = ck/h^2$. Since this must be less than 1/2, we have

$$k < \frac{h^2}{2c}.$$

The first consequence is obvious: k must be relatively small. The second is that h cannot be too small. Since h^2 appears in the formula, making h small would force k to be extremely small!

A third consequence is that we have a converse of this analysis. Suppose $r < .5$. Then all the eigenvalues will be less than one. Recall that the error terms satisfy:

$$\mathbf{u}_{j+k} = A^k \mathbf{U}_j + A^k \mathbf{E}_j.$$

If all the eigenvalues of A are less than 1 in absolute value then $A^k \mathbf{E}_j$ grows smaller and smaller as k increases. This is really good. Rather than building up, the effect of any error diminishes as time passes! From this we arrive at the following principle: **If the explicit numerical solution for a parabolic equation does not blow up, then errors from previous steps fade away!**

Finally, we note that if we have other boundary conditions then instead of equation (36.2) we have:

$$\mathbf{u}_{j+1} = A\mathbf{u}_j + r\mathbf{b}_j, \tag{36.4}$$

where the first and last entries of $\mathbf{b}_j$ contain the boundary conditions and all the other entries are zero. In this case the errors behave just as before, if $r > 1/2$ then the errors multiply and if $r < 1/2$ the errors fade away.

We can write a function program **`myexppmatrix`** that produces the matrix A in (36.3), for given inputs n and r. Without using loops we can use the `diag` command to set up the matrix.:

```
function A = myexpmatrix(n,r)
u = (1-2*r)*ones(n,1);
v = r*ones(n-1,1);
A = diag(u) + diag(v,1) + diag(v,-1);
```

Test this using $n = 6$ and $r = .4, .6$. Check the eigenvalues and eigenvectors of the resulting matirices:

```
A = myexpmatrix(6,.6)
[v e] = eig(A)
```

What is the "mode" represented by the eigenvector with the largest absolute eigenvalue? How is that reflected in the unstable solutions?

Exercises

36.1 Let $L = \pi$, $T = 20$, $f(x) = .1\sin(x)$, $g_1(t) = 0$, $g_2(t) = 0$, $c = .5$, and $m = 20$, as used in the program `myheat.m`. Calculate the value of n corresponding to $r = 1/2$. Try different n in `myheat.m` to find precisely when the method works and when it fails. Is $r = 1/2$ the boundary between failure and success? Hand in a plot of the last success and the first failure.

36.2 Write a script program that produces the graph in Figure 36.1 for $n = 4$. Your program should: define r values from 0 to 1, for each r create the matrix A using `myexppmatrix`, calculate the eigenvalues of A, find the max of the absolute values, and plot these numbers versus r.

Lecture 37

Implicit Methods

The Implicit Difference Equations

By approximating u_{xx} and u_t at t_{j+1} rather than t_j, and using a backwards difference for u_t, the equation $u_t = cu_{xx}$ is approximated by:

$$\frac{u_{i,j+1} - u_{i,j}}{k} = \frac{c}{h^2}(u_{i-1,j+1} - 2u_{i,j+1} + u_{i+1,j+1}). \qquad (37.1)$$

Note that all the terms have index $j + 1$ except one and isolating this term leads to:

$$u_{i,j} = -ru_{i-1,j+1} + (1 + 2r)u_{i,j+1} - ru_{i+1,j+1}, \qquad 1 \leq i \leq m - 1 \qquad (37.2)$$

where $r = ck/h^2$ as before.

Now we have $\mathbf{u}_j$ given in terms of $\mathbf{u}_{j+1}$. This would seem like a problem, until you consider that the relationship is linear. Using matrix notation, we have:

$$\mathbf{u}_j = B\mathbf{u}_{j+1} - r\mathbf{b}_{j+1},$$

where $\mathbf{b}_{j+1}$ represents the boundary condition. Thus to find $\mathbf{u}_{j+1}$, we need only solve the linear system:

$$B\mathbf{u}_{j+1} = \mathbf{u}_j + r\mathbf{b}_{j+1} \qquad (37.3)$$

where $\mathbf{u}_j$ and $\mathbf{b}_{j+1}$ are given and B is the matrix:

$$B = \begin{pmatrix} 1+2r & -r & & & \\ -r & 1+2r & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & 1+2r & -r \\ & & & -r & 1+2r \end{pmatrix}. \qquad (37.4)$$

Using this scheme is called the **implicit method** since $\mathbf{u}_{j+1}$ is defined implicitly.

Since we are solving (37.1), the most important quantity is the maximum absolute eigenvalue of B^{-1}, which is 1 divided by the smallest **ev** of B. Figure 37.1 shows the maximum absolute **ev**'s of B^{-1} as a function of r for various size matrices. Notice that this absolute maximum is always less than 1. Thus errors are always diminished over time and so this method is always stable. For the same reason it is also always as accurate as the individual steps.

Both this implicit method and the explicit method in the previous lecture make $O(h^2)$ error in approximating u_{xx} and $O(k)$ error in approximating u_t, so they have total error $O(h^2 + k)$. Thus although the stability condition allows the implicit method to use arbitrarily large k, to maintain accuracy we still need $k \sim h^2$.

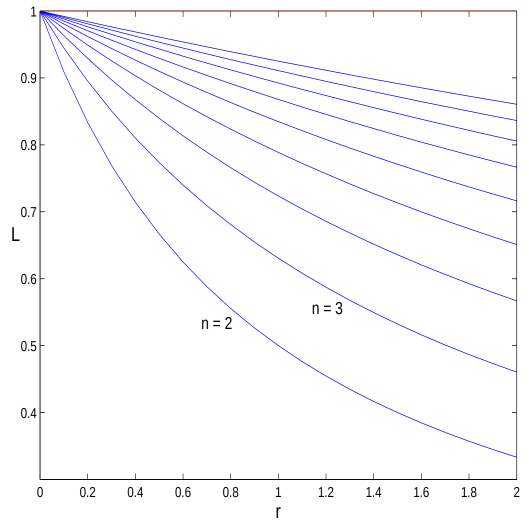

Figure 37.1: Maximum absolute eigenvalue L as a function of r for the matrix B^{-1} from the implicit method for the heat equation calculated for matrices B of sizes $n = 2 \dots 10$. Whenever $L < 1$ the method is stable, i.e. it is always stable.

Crank-Nicholson Method

Now that we have two different methods for solving parabolic equation, it is natural to ask, "can we improve by taking an average of the two methods?" The answer is yes.

We implement a weighted average of the two methods by considering an average of the approximations of u_{xx} at j and $j+1$. This leads to the equations:

$$\frac{u_{i,j+1} - u_{i,j}}{k} = \frac{\lambda c}{h^2}(u_{i-1,j+1} - 2u_{i,j+1} + u_{i+1,j+1}) + \frac{(1-\lambda)c}{h^2}(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}). \qquad (37.5)$$

The implicit method contained in these equations is called the **Crank-Nicholson method**. Gathering terms yields the equations:

$$-r\lambda u_{i-1,j+1} + (1+2r\lambda)u_{i,j+1} - r\lambda u_{i+1,j+1} = r(1-\lambda)u_{i-1,j} + (1-2r(1-\lambda))u_{i,j} + r(1-\lambda)u_{i+1,j}.$$

In matrix notation this is:

$$B_\lambda \mathbf{u}_{j+1} = A_\lambda \mathbf{u}_j + r\mathbf{b}_{j+1},$$

where

$$A_\lambda = \begin{pmatrix} 1-2(1-\lambda)r & (1-\lambda)r & & & \\ (1-\lambda)r & 1-2(1-\lambda)r & (1-\lambda)r & & \\ & \ddots & \ddots & \ddots & \\ & & (1-\lambda)r & 1-2(1-\lambda)r & (1-\lambda)r \\ & & & (1-\lambda)r & 1-2(1-\lambda)r \end{pmatrix} \qquad (37.6)$$

and

$$B_\lambda = \begin{pmatrix} 1+2r\lambda & -r\lambda & & & \\ -r\lambda & 1+2r\lambda & -r\lambda & & \\ & \ddots & \ddots & \ddots & \\ & & -r\lambda & 1+2r\lambda & -r\lambda \\ & & & -r\lambda & 1+2r\lambda \end{pmatrix}. \tag{37.7}$$

In this equation $\mathbf{u}_j$ and $\mathbf{b}_{j+1}$ are known, $A\mathbf{u}_j$ can be calculated directly, and then the equation is solved for $\mathbf{u}_{j+1}$.

If we choose $\lambda = 1/2$, then we are in effect doing a central difference for u_t, which has error $O(k^2)$. Our total error is then $O(h^2 + k^2)$. With a bit of work, we can show that the method is always stable, and so we can use $k \sim h$ without a problem.

To get optimal accuracy with a weighted average, it is always necessary to use the right weights. For the Crank-Nicholson method with a given r, we need to choose:

$$\lambda = \frac{r - 1/6}{2r}.$$

This choice will make the method have truncation error of order $O(h^4 + k^2)$, which is really good considering that the implicit and explicit methods each have truncation errors of order $O(h^2 + k)$. Surprisingly, we can do even better if we also require:

$$r = \frac{\sqrt{5}}{10} \approx 0.22361,$$

and, consequently:

$$\lambda = \frac{3 - \sqrt{5}}{6} \approx 0.12732.$$

With these choices, the method has truncation error of order $O(h^6)$, which is absolutely amazing.

To appreciate the implications, suppose that we need to solve a problem with 4 significant digits. If we use the explicit or implicit method alone then we will need $h^2 \approx k \approx 10^{-4}$. If L and T are comparable to 1, then we need $m \approx 100$ and $n \approx 10{,}000$. Thus we would have a total of $1{,}000{,}000$ grid points, almost all on the interior. This is a lot.

Next suppose we solve the same problem using the optimal Crank-Nicholson method. We would need $h^6 \approx 10^{-4}$ which would require us to take $m \approx 4.64$, so we would take $m = 5$. For k we need $k = .22361h^2/c$. If $c = 1$, this gives $k \approx .22 \times 10^{-4/3}$ so $n = 100$ is sufficient This gives us a total of 500 interior grid points, or, a factor of 2,000 fewer than the explicit or implicit method alone.

Exercises

37.1 Modify the program `myexppmatrix` from exercise 36.2 into a function program `myimpmatrix` that produces the matrix B_λ, for given inputs n, r and λ. Use the `diag` command to set up the matrix. Modify your script from exercise 36.3 to use B_λ^{-1} for $\lambda = 1/2$. Turn in the programs and the plot.

Lecture 38

Finite Difference Method for Elliptic PDEs

Examples of Elliptic PDEs

Eliptic PDE's are equations with second derivatives in space and no time derivative. The most important examples are Laplace's equation:

$$\Delta u = u_{xx} + u_{yy} + u_{zz} = 0$$

and the Poisson equation:

$$\Delta u = f(x, y, z).$$

These equations are used in a large variety of physical situations such as: steady state heat problems, steady state chemical distributions, electrostatic potentials, elastic deformation and steady state fluid flows.

For the sake of clarity we will only consider the two dimensional problem. A good model problem in this dimension is the elastic deflection of a membrane. Suppose that a membrane such as a sheet of rubber is stretched across a rectangular frame. If some of the edges of the frame are bent, or if forces are applied to the sheet then it will deflect by an amount $u(x, y)$ at each point (x, y). This u will satify the boundary value problem:

$$\begin{aligned} u_{xx} + u_{yy} &= f(x, y), \qquad (x, y) \text{ in } R \\ u(x, y) &= g(x, y), \qquad (x, y) \text{ on } \partial R, \end{aligned} \tag{38.1}$$

where R is the rectangle, ∂R is the edge of the rectangle, $f(x, y)$ is the force density (pressure) applied at each point and $g(x, y)$ is the deflection at the edge.

The Finite Difference Equations

Suppose the rectangle is described by:

$$R = \{a \le x \le b, c \le y \le d\}.$$

We will divide R in subrectangles. If we have m subdivisions in the x direction and n subdivisions in the y direction, then the step size in the x and y directions respectively are

$$h = \frac{b-a}{m} \quad \text{and} \quad k = \frac{d-c}{n}.$$

We obtain the finite difference equations for (38.1) by replacing u_{xx} and u_{yy} by their central differences:

$$\frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{k^2} = f(x_i, y_j) = f_{ij}, \tag{38.2}$$

for $1 \leq i \leq m-1$ and $1 \leq j \leq n-1$. The boundary conditions are introduced by:

$$u_{0,j} = g(a, y_j), \quad u_{m,j} = g(b, y_j), \quad u_{i,0} = g(x_i, c), \quad u_{i,n} = g(x_i, d). \tag{38.3}$$

Direct Solution of the Equations

Notice that since the edge values are prescribed, there are $(m-1) \times (n-1)$ grid points where we need to determine the solution. Note also that there are exactly $(m-1) \times (n-1)$ equations in (38.2). Finally, notice that the equations are all linear. Thus we could solve the equations exactly using matrix methods. To do this we would first need to express the u_{ij}'s as a vector, rather then a matrix. To do this there is a standard procedure; it is to let $\mathbf{u}$ be the column vector we get by placing one column after another from the columns of (u_{ij}). Thus we would list $u_{:,1}$ first then $u_{:,2}$, etc.. Next we would need to write the matrix A that contains the coefficients of the equations (38.2) and incorporate the boundary conditions in a vector $\mathbf{b}$. Then we could solve an equation of the form

$$A\mathbf{u} = \mathbf{b}. \tag{38.4}$$

Setting up and solving this equation is called the direct method.

An advantage of the direct method is that solving (38.4) can be done relatively quickly and accurately. The drawback of the direct method is that one must set up $\mathbf{u}$, A and $\mathbf{b}$, which is confusing. Further, the matrix A has dimensions $(m-1)(n-1) \times (m-1)(n-1)$, which can be rather large.

Iterative Solution

A prefered alternative to the direct method is to solve the finite difference equations iteratively. To do this, first solve (38.2) for u_{ij}:

$$u_{ij} = \frac{1}{2(h^2 + k^2)} \left(k^2(u_{i+1,j} + u_{i-1,j}) + h^2(u_{i,j+1} + u_{i,j-1}) - h^2k^2 f_{ij} \right). \tag{38.5}$$

Using this formula, along with (38.3), we can update u_{ij} from its neighbors, just as we did for the nonlinear boundary value problem. If this method converges, then the result is an approximate solution.

The iterative solution is implemented in the program `mypoisson.m` on the class web site. Download and read it. You will notice that `maxit` is set to 0. Thus the program will not do any iteration, but will plot the initial guess. The initial guess in this case consists of the proper boundary values at the edges, and zero everywhere in the interior. To see the solution evolve, gradually increase `maxit`.

Exercises

38.1 Modify the program `mypoisson.m` (from the class web page) in the following ways:

1. Change $x = b$ to be an insulated boundary, i.e. $u_x(b, y) = 0$.
2. Change the force $f(x, y)$ to a negative constant $-p$.

Experiment with various values of p and `maxit`. Obtain plots for small and large p. Tell how many iterations are needed for convergence in each. For large p plot also a couple of intermediate steps.

Lecture 39

Finite Elements

Triangulating a Region

A disadvantage of finite difference methods is that they require a very regular grid, and thus a very regular region, either rectangular or a regular part of a rectangle. Finite elements is a method that works for any shape region because it is not built of a grid, but on triangulation of the region, i.e. cutting the region up into triangles as we did in a previous lecture.

The following figure shows a triangularization of a region.

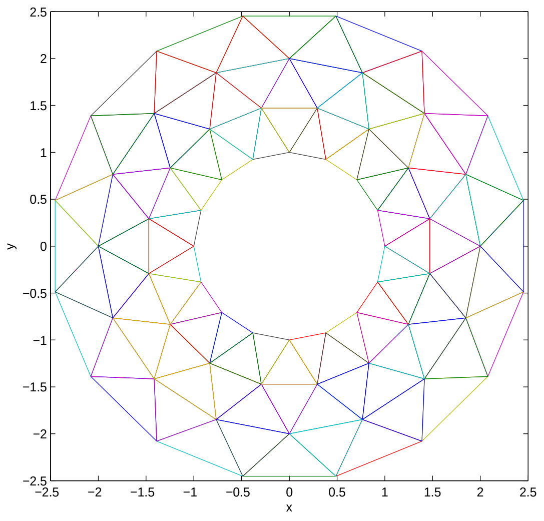

Figure 39.1: An annular region with a triangulation. Notice that the nodes and triangles are very evenly spaced.

This figure was produced by the script program `mywasher.m`. Notice that the nodes are evenly distributed. This is good for the finite element process where we will use it.

Open the program `mywasher.m`. This program defines a triangulation by defining the vertices in a matrix `V` in which each row contains the x and y coordinates of a vertex. Notice that we list the interior nodes first, then the boundary nodes.

Triangles are defined in the matrix `T`. Each row of `T` has three integer numbers indicating the indices of the nodes that form a triangle. For instance the first row is `43 42 25`, so T_1 is the triangle

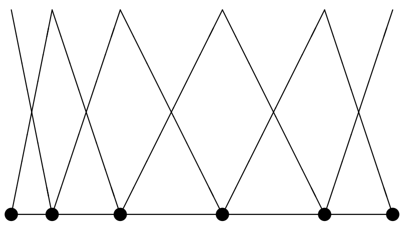

Figure 39.2: The finite elements for the triangulation of a one dimensional object.

with vertices $\mathbf{v}_{43}$, $\mathbf{v}_{42}$ and $\mathbf{v}_{25}$. The matrix T in this case was produced by the MATLAB command `delaunay`. The command produced more triangles than desire and the unwanted ones were deleted.

We can produce a three dimensional plot of the region and triangles by changing the last line to `trimesh(T,x,y,c)`.

What is a finite element?

The finite element method is a mathematically complicated process. However, a finite element is actually a very simple object.

An element is a piecewise-defined function Φ_j which is:
– Linear (or some other simple function) on each triangle.
– There is exactly one element corresponding each node.
– An element equals 1 at the node j and 0 at all other nodes.
If there are n vertices's, then there are also n elements. We will use the index j to number both vertices's and the corresponding elements. Thus the set of elements is:

$$\{\Phi_j\}_{j=1}^n.$$

As we have defined it, an element, Φ_j, is a function whose graph is a pyramid with its peak over node j.

If we consider one dimension, then a triangulation is just a subdivision into subintervals. In Figure 39.2 we show an uneven subdivision of an interval and the elements corresponding to each node.

What is a finite element solution?

A finite element (approximate) solution is a linear combination of the elements:

$$U = \sum_{j=1}^{n} C_j \Phi_j. \tag{39.1}$$

Thus finding a finite element solution amounts to finding the best values for the constants $\{C_j\}_{j=1}^n$.

In the program `mywasher.m`, the vector `c` contains the node values C_j. These values give the height at each node in the graph. For instance if we set all equal to 0 except one equal to 1, then the function is a finite element. Do this for one boundary node, then for one interior node.

Notice that a sum of linear functions is a linear function. Thus the solution using linear elements is a piecewise linear function. Also notice that if we denote the j-th vertex by $\mathbf{v}_j$, then

$$U(\mathbf{v}_j) = C_j. \tag{39.2}$$

Thus we see that **the constants C_j are just the values at the nodes.**

Take the one-dimensional case. Since we know that the solution is linear on each subinterval, knowing the values at the endpoints of the subintervals, i.e. the nodes, gives us complete knowledge of the solution. Figure 39.3 could be a finite element solution since it is piecewise linear.

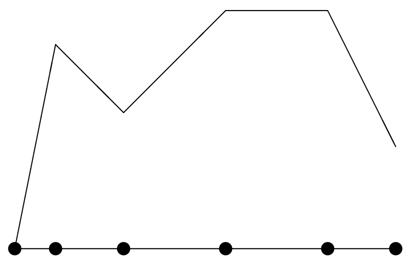

Figure 39.3: A possible finite element solution for a one dimensional object. Values are assigned at each node and a linear interpolant is used in between.

In the two-dimensional case, the solution is linear on each triangle, and so again, if we know the values $\{C_j\}_{j=1}^n$ at the nodes then we know everything.

Experiment with finite elements

By changing the values in `c` in the program we can produce different three dimensional shapes based on the triangles. The point then of a finite element solution is to find the values at the nodes that best approximate the true solution. This task can be subdivided into two parts: (1) assigning the values at the boundary nodes and (2) assigning the values at the interior nodes.

Values at boundary nodes

Once a triangulation and a set of finite elements is set up, the next step is to incorporate the boundary conditions of the problem. Suppose that we have fixed boundary conditions, i.e. of the form

$$u(\mathbf{x}) = g(\mathbf{x}), \qquad \text{for} \quad \mathbf{x} \in \partial D,$$

where D is the object (domain) and ∂D is its boundary. Then the boundary condition directly determines the values on the boundary nodes.

In particular, suppose that $\mathbf{v}_\ell$ is a boundary node. Since $\Phi_\ell(\mathbf{v}_\ell) = 1$ and all the other elements are zero at node $\mathbf{v}_\ell$, then to make

$$U(\mathbf{v}_\ell) = \sum_{j=1}^{n} C_j \Phi_j(\mathbf{v}_\ell) = g(\mathbf{v}_\ell),$$

then we must choose

$$C_\ell = g(\mathbf{v}_\ell).$$

Thus **the constants C_j for the boundary nodes are set at exactly the value of the boundary condition at the nodes.**

Thus, if there are m interior nodes, then

$$C_j = g(\mathbf{v}_j), \qquad \text{for all} \quad m+1 \le j \le n.$$

In the program `mywasher` the first 32 vertices correspond to interior nodes and the last 32 correspond to boundary nodes. By setting the last 32 values of `c`, we achieve the boundary conditions. We could do this by adding the following commands to the program:

```
c(33:64) = .5;
```

or more elaborately we might use functions:

```
c(33:48) = x(33:48).^2 - .5*y(33:48).^2;
c(49:64) = .2*cos(y(49:64));
```

Exercises

39.1 Generate an interesting or useful 2-d object and a well-distributed triangulation of it. Plot the region. Plot one interior finite element and one boundary finite element. Assign values to the boundary using a function (or functions) and plot the region with the boundary values.

Lecture 40

Determining Internal Node Values

As seen in the previous section, a finite element solution of a boundary value problem boils down to finding the best values of the constants $\{C_j\}_{j=1}^{n}$ which are the values of the solution at the nodes. The interior nodes values are determined by *variational principles*. Variational principles usually amount to **minimizing internal energy**. It is a physical principle that systems seek to be in a state of minimal energy and this principle is used to find the internal node values.

Variational Principles

For the differential equations that describe many physical systems, the internal energy of the system is an integral. For instance, for the steady state heat equation:

$$u_{xx} + u_{yy} = g(x,y) \tag{40.1}$$

the internal energy is the integral:

$$I[u] = \iint_R u_x^2 + u_y^2 + 2g(x,y)u(x,y)\,dA, \tag{40.2}$$

where R is the region on which we are working. It can be shown, that $u(x,u)$ is a solution of (40.1) if and only if it is minimizer of $I[u]$ in (40.2).

The finite element solution

Recall that a finite element solution is a linear combination of finite element functions:

$$U(x,y) = \sum_{j=1}^{n} C_j \Phi_j(x,y),$$

where n is the number of nodes. To obtain the values at the internal nodes, we will plug $U(x,y)$ into the energy integral and minimize. That is, we find the minimum of:

$$I[U]$$

for all choices of $\{C_j\}_{j=1}^{m}$, where m is the number of internal nodes. In this as with any other minimization problem, the way to find a possible minimum is to differentiate the quantity with respect to the variables and set those to zero. In this case the free variables are $\{C_j\}_{j=1}^{m}$. Thus to find the minimizer we should try to solve the following set of equations:

$$\frac{\partial I[U]}{\partial C_j} = 0 \qquad \text{for} \quad 1 \le j \le m. \tag{40.3}$$

We call this set of equations the **internal node equations**. At this point we should ask whether the internal node equations can be solved, and if so, is the solution actually a minimizer (and not a maximizer). The following two facts answer these questions. These facts make the finite element method practical:
– **for most applications the internal node equations are linear**
– **for most applications the internal node equations give a minimizer**

We can demonstrate the first fact using an example.

Application to the steady state heat equation

If we plug the candidate finite element solution $U(x, y)$ into the energy integral for the heat equation (40.2), we obtain

$$I[U] = \iint_R U_x(x,y)^2 + U_y(x,y)^2 + 2g(x,y)U(x,y)\,dA. \tag{40.4}$$

Differentiating with respect to C_j we obtain the internal node equations

$$0 = \iint_R 2U_x \frac{\partial U_x}{\partial C_j} + 2U_y \frac{\partial U_y}{\partial C_j} + 2g(x,y)\frac{\partial U}{\partial C_j}\,dA, \tag{40.5}$$

for $1 \le j \le m$. Now we have several simplifications. First note that since

$$U(x,y) = \sum_{j=1}^{n} C_j \Phi_j(x,y),$$

we have

$$\frac{\partial U}{\partial C_j} = \Phi_j(x,y)$$

Also note that

$$U_x(x,y) = \sum_{j=1}^{n} C_j \frac{\partial}{\partial x}\Phi_j(x,y),$$

and so

$$\frac{\partial U_x}{\partial C_j} = (\Phi_j)_x.$$

Similarly, $\frac{\partial U_y}{\partial C_j} = (\Phi_j)_y$. The integral (40.5) then becomes

$$0 = 2\iint U_x(\Phi_j)_x + U_y(\Phi_j)_y + g(x,y)\Phi_j(x,y)\,dA,$$

$1 \le j \le m$.

Next we use the fact that the region R is subdivided into triangles $\{T_i\}_{i=1}^{p}$ and the functions in question have different definitions on each triangle. The integral then is a sum of the integrals:

$$0 = 2\sum_{i=1}^{p} \iint_{T_i} U_x(\Phi_j)_x + U_y(\Phi_j)_y + g(x,y)\Phi_j(x,y)\,dA,$$

$1 \le j \le m$.

Now note that the function $\Phi_j(x, y)$ is linear on triangle T_i and so

$$\Phi_{ij}(x,y) = \Phi_j|_{T_i}(x,y) = a_{ij} + b_{ij}x + c_{ij}y.$$

This gives us the following simplification:

$$(\Phi_{ij})_x(x,y) = b_{ij} \quad \text{and} \quad (\Phi_{ij})_y(x,y) = c_{ij}.$$

Also, U_x restricted to T_i has the form:

$$U_x = \sum_{k=1}^{n} C_k b_{ik} \quad \text{and} \quad U_y = \sum_{k=1}^{n} C_k c_{ik}.$$

The internal node equations then reduces to:

$$0 = \sum_{i=1}^{p} \iint_{T_i} \left(\sum_{k=1}^{n} C_k b_{ik} \right) b_{ij} + \left(\sum_{k=1}^{n} C_k c_{ik} \right) c_{ij} + g(x,y)\Phi_{ij}(x,y)\, dA,$$

$1 \le j \le m$. Now notice that $\left(\sum_{k=1}^{n} C_k b_{ik}\right) b_{ij}$ is just a constant on T_i, thus we have:

$$\iint_{T_i} \left(\sum_{k=1}^{n} C_k b_{ik} \right) b_{ij} + \left(\sum_{k=1}^{n} C_k c_{ik} \right) c_{ij} = \left[\left(\sum_{k=1}^{n} C_k b_{ik} \right) b_{ij} + \left(\sum_{k=1}^{n} C_k c_{ik} \right) c_{ij} \right] A_i,$$

where A_i is just the area of T_i. Finally, we apply the Three Corners rule to make an approximation to the integral

$$\iint_{T_i} g(x,y)\Phi_{ij}(x,y)\, dA\,.$$

Since $\Phi_{ij}(x_k, y_k) = 0$ if $k \neq j$ and even $\Phi_{ij}(x_j, y_j) = 0$ if T_i does not have a corner at (x_j, y_j), we get the approximation

$$\Phi_{ij}(x_j, y_j) g(x_j, y_j) A_i/3.$$

If T_i does have a corner at (x_j, y_j) then $\Phi_{ij}(x_j, y_j) = 1$.

Summarizing, the internal node equations are:

$$0 = \sum_{i=1}^{p} \left[\left(\sum_{k=1}^{n} C_k b_{ik} \right) b_{ij} + \left(\sum_{k=1}^{n} C_k c_{ik} \right) c_{ij} + \frac{1}{3} g(x_j, y_j)\Phi_{ij}(x_j, y_j) \right] A_i,$$

for $1 \le j \le m$. While not pretty, these equations are in fact linear in the unknowns $\{C_j\}$.

Experiment

Download the program `myfiniteelem.m` from the class web site.

This program produces a finite element solution for the steady state heat equation without source term:

$$u_{xx} + u_{yy} = 0$$

The boundary values are input directly on the line:

```
c = [ 0 0 0 0 0 0 0 1 1 1 0 0 ];
```

Try different values for `c`. You will see that the program works no matter what you choose.

Exercises

40.1 Study for the final!

Review of Part IV

Methods and Formulas

Initial Value Problems

Euler's method:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + hf(t_i, \mathbf{y}_i).$$

Modified (or Improved) Euler method:

$$\begin{aligned}\mathbf{k}_1 &= hf(t_i, \mathbf{y}_i)\\ \mathbf{k}_2 &= hf(t_i + h, \mathbf{y}_i + \mathbf{k}_1)\\ \mathbf{y}_{i+1} &= \mathbf{y}_i + \frac{1}{2}(\mathbf{k}_1 + \mathbf{k}_2)\end{aligned}$$

Boundary Value Probems

Finite Differences:
Replace the Differential Equation by Difference Equations on a grid.

Explicit Method Finite Differences for Parabolic PDE (heat):

$$u_{i,j+1} = ru_{i-1,j} + (1-2r)u_{i,j} + ru_{i+1,j},$$

where $h = L/m$, $k = T/n$, and $r = ck/h^2$. The stability condition is $r < 1/2$.

Implicit Method Finite Differences for Parabolic PDE (heat):

$$u_{i,j} = -ru_{i-1,j+1} + (1+2r)u_{i,j+1} - ru_{i+1,j+1},$$

which is always stable and has truncation error $O(h^2 + k)$.

Crank-Nicholson Method Finite Differences for Parabolic PDE (heat):

$$-ru_{i-1,j+1} + 2(1+r)u_{i,j+1} - ru_{i+1,j+1} = ru_{i-1,j} + 2(1-r)u_{i,j} + ru_{i+1,j},$$

which is always stable and has truncation error $O(h^2 + k^2)$ or better if r and λ are chosen optimally.

Finite Difference Method for Elliptic PDEs:

$$\frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{k^2} = f(x_i, y_j) = f_{ij},$$

Part V
Appendices

Appendix A

Sample Exams

MATH344 Midterm Exam – Spring 04

1. For $f(x) = x^2 - 5$, do 2 iterations of Newton's method, starting with $x_0 = 2.0$. What is the relative error of x_2? About how many more steps would be needed to make the error less than 10^{-16}?

2. Write a **Matlab** program to do n steps of the bisection method for a function f with starting interval $[a, b]$. Let f, a, b and n be the inputs and the final x the output.

3. Given a data set represented by vectors $\mathbf{x}$ and $\mathbf{y}$, describe how you would use **Matlab** to get a Least Squares Approximation, Polynomial Interpolation and Spline Interpolation?

4. Given that the LU decomposition of $A = \begin{bmatrix} 3 & 3 \\ 1 & 2 \end{bmatrix}$ is $LU = \begin{bmatrix} 1 & 0 \\ 1/3 & 1 \end{bmatrix} \begin{bmatrix} 3 & 3 \\ 0 & 1 \end{bmatrix}$, solve $A\mathbf{x} = \mathbf{b}$ where $\mathbf{b} = (1, 2)'$.

5. Suppose $A^{-1} = \begin{bmatrix} -1 & 2 \\ 1 & -1 \end{bmatrix}$. Using $\mathbf{v}_0 = (1, 1)'$ as the starting vector do 2 iterations of the Inverse Power Method for A. What do the results mean?

6. Let $A = \begin{bmatrix} 1 & .5 \\ 2 & 1 \end{bmatrix}$. Find the LU factorization with pivoting.

7. What is the condition number of a matrix? How do you find it with **Matlab**? What are the implications of the condition number when solving a linear system? What is the engineering solution to a problem with a bad condition number?

8. Write a **Matlab** program to do n iterations of the Power Method. Let the matrix A and n be inputs and let [e v] (the eigenvalue and eigenvector) be the outputs.

9. Write a **Matlab** program to that solves a linear system $A\mathbf{x} = \mathbf{b}$ using LU decomposition. Let A, $\mathbf{b}$ and *tol* be the inputs and $\mathbf{x}$ the output. If the error (residual) is not less than *tol*, then display a warning.

10. List your 10 least favorite **Matlab** commands.

X. What problem did Dr. Young totally botch in class?

MATH 344 – Midterm Exam – Winter 05

1. For $f(x) = x^2 - 5$, do 2 iterations of the bisection method, starting with $[a, b,] = [2, 3]$. What is the relative error? About how many more steps would be needed to make the error less than 10^{-6}?
2. Write a MATLAB program to do n steps of Newton's method for a function f with starting interval $[a, b]$. Let f, f', x_0 and n be the inputs and the final x the output.
3. Suppose $f(x)$ has been defined as an inline function. Give MATLAB commands to plot it on the interval $[0, 10]$.
4. Write a function program which will find the roots of a function f on an interval $[a, b]$.
5. Suppose $A = \begin{bmatrix} -1 & 2 \\ 1 & -1 \end{bmatrix}$. Using $\mathbf{v}_0 = (1, 1)'$ as the starting vector do 2 iterations of the Power Method for A. What do the results mean?
6. Let $A = \begin{bmatrix} -1 & 5 \\ 2 & 2 \end{bmatrix}$. Find the LU factorization with pivoting.
7. What is the condition number of a matrix? How do you find it with MATLAB? What are the implications of the condition number when solving a linear system? What is the engineering solution to a problem with a bad condition number?
8. Write a MATLAB program to do n iterations of the QR Method. Let the matrix `A` and `n` be inputs and let `e` be the output.
9. Write a MATLAB program to that solves a linear system $A\mathbf{x} = \mathbf{b}$ using LU decomposition. Let A, $\mathbf{b}$ and *tol* be the inputs and $\mathbf{x}$ the output. If the error (residual) is not less than *tol*, then display a warning.
10. Give the MATLAB commands, or sequences of commands for solving a linear system $A\mathbf{x} = \mathbf{b}$ in as many ways as you know. Which of these are the worst and best?

X. When using Newton's method, how does one measure its effectiveness?

MATH 344 – Midterm Exam – Spring 2006

1. What 10 commands in MATLAB are the least useful?

2. For $f(x) = x^3 - 6$, do 2 iterations of Newton's method, starting with $x_0 = 2$.

3. What are the main *differences* in the uses of: Polynomial Interpolation, Splines and Least Squares Fitting?

4. Find the LU decomposition of A using pivoting if needed:
$A = \begin{bmatrix} 3 & -2 \\ 6 & 1 \end{bmatrix}$

5. Write a MATLAB function program that calculates the sum of the squares of the first n integers.

6. What is the command in MATLAB to produce the eigenvalues and eigenvectors of a matrix. Which method does it use? What will be the form of the output?

7. What is the condition number of a matrix? How do you find it with MATLAB? What are the implications of the condition number when solving a linear system?

8. Find the eigenvalues and eigenvectors of the matrix:
$A = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix}$

9. Write a MATLAB function program that takes an input n, produces a random $n \times n$ matrix A and random vector $\bar{b}$, solves $A\bar{x} = \bar{b}$ (using the built in command) and outputs the residual (number).

10. Write a MATLAB script program that will use Newton's method to find a root of the system of functions $f_1(x, y) = x^3 - y^2 + 1$ and $f_2(x, y) = y^3 + x - 1$ starting from the initial guess $(0, 0)$.

X. In this class, every problem leads to ... .

MATH344 Final Exam – Spring 2004

1. Estimate the integral $\int_0^\pi \sin x \, dx$ using L_4, R_4 and T_4. Calculate the exact value and the percentage errors of each of the approximations.
2. Write a MATLAB program to do the midpoint method for integration. Let the inputs be the function f, the endpoints a, b and the number of subintervals n.
3. Write a MATLAB program to do n steps of the Euler method for a differential equation $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$, with $\mathbf{x}(a) = \mathbf{x}_0$. Let the first line be:
`function [t, x] = myeuler(f,x0,a,b,n)`.
4. What is the condition number of a matrix? How do you find it with **MatLab**? What are the implications of the condition number when solving a linear system? What is the engineering solution to a problem with a bad condition number?.
5. Write the equation $\theta" + a\theta' + b\sin\theta = c\sin t$ as a linear system of first order equations and set up the Euler method for the system.
6. Describe RK45. What is the command for it in MATLAB?
7. When and why does the explicit finite difference method for the heat/diffusion equation become unstable?
8. Derive the implicit finite difference equations for solving the heat/diffusion equation $u_t = cu_{xx}$.
9. Set up the finite difference equations for the BVP: $u_{xx} + u_{yy} = f(x, y)$, on the rectangle $0 \le x \le a$ and $0 \le y \le b$, with $u = 0$ on all the boundaries. Explain how the difference equations could be solved as a linear system.
10. Write a MATLAB program to do Newton's method with starting point x_0 that does n steps or stops when a tolerance is reached. Include f' in the inputs. Let the first line be:
`function x = mynewton(f,f1,x0,n,tol)`.
11. What is the geometric meaning of eigenvalue and eigenvector. What is the MATLAB command to find them, with correct syntax. Describe an application of ew's and ev's.
12. Give the MATLAB command(s) for as many different ways as you know to solve a system of linear equations. Rank them.
13. Discuss uses of Polynomial Interpolation, Splines and Least Squares Interpolations.
14. What is a finite element and a finite element solution?
15. Very generally, how are the boundary and interior values of the finite element solution obtained.

X. When you have two different methods of approximating something, how can you get an even better approximation?

MATH344 – Final Exam – Winter 2005

1. Estimate the integral $\int_0^{16} \sqrt{x}\,dx$ using L_4, R_4, T_4 and S_4. Calculate the exact value and the percentage errors of each of the approximations.
2. Write a MATLAB function program to do the trapezoid method for integration. Let the inputs be the function f, the endpoints a, b and the number of subintervals n.
3. Write a MATLAB program to do n steps of the modified Euler method for a differential equation $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$, with $\mathbf{x}(a) = \mathbf{x}_0$. Let the first line be:
   `function [t, x] = myeuler(f,x0,a,b,n)`.
4. Write a MATLAB script program that calculates $\sum_{i=0}^{\infty} \frac{.5^i}{i^2} \sin(i)$ by adding terms until the sum stops changing. How do you know that it would stop?
5. Write the IVP: $\theta" + a\theta' + b\sin\theta = c\sin t$, $\theta(0) = \pi/4$, $\theta'(0) = 0$ as a system of first order equations. Give all the MATLAB commands needed to solve this IVP.
6. Describe RK45. What is the command for it in MATLAB?
7. Explain why order matters in engineering problems.
8. Derive the explicit finite difference equations for solving the heat/diffusion equation $u_t = cu_{xx}$, with boundary conditions, $u(0,t) = a$, $u(L,t) = b$, and $u(x,0) = f(x)$.
9. Set up the finite difference equations for the BVP: $u_{rr} + \frac{1}{r}u_r = f(r)$, on the interal $0 \le r \le R$, with $u(R) = 0$ and $u_r(0) = 0$. Explain how to avoid the problem at $r = 0$.
10. Write a MATLAB program to do Newton's method with starting point x_0 that does n steps or stops when a tolerance is reached. Include f' in the inputs. Let the first line be:
    `function x = mynewton(f,f1,x0,n,tol)`.
11. Write a MATLAB program to do n iterations of the inverse power method. Let the matrix A and n be the inputs and `[e v]` be the output. What is the meaning of e and v?
12. Describe methods for approximating double integrals.
13. What are: Polynomial Interpolations, Splines and Least Squares Interpolations. How do you get them from MATLAB?
14. Discuss: triangulation, finite element and finite element solution.
15. How are the boundary and interior values of the finite element solution obtained.

X. When I can't get my MATLAB program to work, I ... .

MATH344 – Final Exam – Spring 2006

1. Approximate the integral $\int_0^\pi \sin x\, dx$ using M_4 and S_4. Which do you expect to be more accurate?
2. Write a MATLAB function program to do the Trapezoid Rule for integration of data. Let the inputs be vectors x and y, where it is assumed that y is a function of x and x is not necessarily evenly spaced.
3. Describe and give formulas for 2 methods to approximate double integrals based on triangles.
4. Explain how to incorporate an insulated boundary in a finite difference method.
5. Set up the finite difference equations for the BVP: $u_{rr} + \frac{1}{r}u_r = f(r)$, on the interval $0 \leq r \leq R$, with $u(R) = 4$ and $u_r(0) = 0$. Explain how to avoid the problem at $r = 0$.
6. Do 2 steps of Newton's method to solve the equation $f(x) = x^2 - 5 = 0$, with starting point $x_0 = 2$. Find the percentage errors of x_0, x_1 and x_2.
7. Write a MATLAB script program that will use Newton's method to find a root of the system of functions $f_1(x, y) = x^3 + y - 1$ and $f_2(x, y) = y^3 - x + 1$ starting from the initial guess $(.5, .5)$.
8. Write a MATLAB function program to do n steps of the Modified Euler method for a differential equation $\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x})$, on the time interval $[a, b]$, with $\mathbf{x}(a) = \mathbf{x}_0$.
9. Write the IVP: $\theta'' + .5\theta' + \sin\theta = \sin 2t$, $\theta(0) = 1$, $\theta'(0) = 0$ as a system of first order equations. Give all the MATLAB commands needed to solve this IVP on the interval $0 \leq t \leq 10$.
10. What is variable step size? How is it implemented RK45?
11. What are main differences between the Finite Difference Method and Finite Elements Method?
12. If $U(x) = \sum_{j=1}^{n} C_j \Phi_j(\bar{x})$ is a finite element solution, what is the meaning of C_j? What is the final step of finding a finite element solution?
13. Write a MATLAB program to do n iterations of the power method. Let the matrix A and n be the inputs and `[e v]` be the output. What is the meaning of e and v?
14. Let $A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$. Find the LU factorization with pivoting.
15. What is a polynomial interpolation? How do you get one in MATLAB?

X. When you have two different methods to make an approximation, how can you get a potentially more accurate method?

Appendix B

Glossary of Matlab Commands

Mathematical Operations

`+`	Addition. Type `help plus` for information.
`-`	Subtraction. Type `help minus` for information.
`*`	Scalar or matrix multiplication. Type `help mtimes` for information.
`/`	Scalar or right matrix division. Type `help slash` for information. For matrices, the command `A/B` is equivalent to `A*inv(B)`.
`^`	Scalar or matrix powers. Type `help mpower` for information.
`.*`	Element by element multiplication. Type `help times` for information.
`.^`	Element by element exponentiation. Type `help power` for information.
`./`	Element by element division.

Built-in Mathematical Constants

`eps`	Machine epsilon, i.e. approximately the computer's floating point roundoff error.
`i`	$\sqrt{-1}$.
`Inf`	∞.
`NaN`	Not a number. Indicates an invalid operation such as 0/0.
`pi`	$\pi = 3.14159\ldots$.

Built-in Mathematical Functions

`abs(x)`	Absolute value$\|x\|$.
`acos(x)`	Inverse cosine $\arccos x$.
`asin(x)`	Inverse sine $\arcsin x$.

`atan(x)`	Inverse tangent $\arctan x$.
`cos(x)`	Cosine $\cos x$.
`cosh(x)`	Hyperbolic cosine $\cosh x$.
`cot(x)`	Cotangent $\cot x$.
`exp(x)`	Exponential function $e^x = \exp x$.
`log(x)`	Natural logarithm $\ln x = \log_e x$.
`sec(x)`	Secant $\sec x$.
`sin(x)`	Sine $\sin x$.
`sinh(x)`	Hyperbolic sine $\sinh x$.
`sqrt(x)`	Square root $\sqrt{x}$.
`tan(x)`	Tangent $\tan x$.
`tanh(x)`	Hyperbolic tangent $\tanh x$.
`max`	Computes maximum of the rows of a matrix.
`mean`	Computes the average of the rows of a matrix.
`min`	Computes the minimum of the rows of a matrix.

Built-in Numerical Mathematical Operations

`fzero`	Tries to find a zero of the specified function near a starting point or on a specified interval.
`inline`	Define a function in the command window.
`ode113`	Numerical multiple step ODE solver.
`ode45`	Runga-Kutta 45 numerical ODE solver.
`quad`	Numerical integration using an adaptive Simpson's rule.
`dblquad`	Double integration.
`triplequad`	Triple integration.

Built-in Symbolic Mathematical Operations

`collect`	Collects powers of the specified variable is a given symbolic expression.
`compose`	Composition of symbolic functions.
`diff`	Symbolic differentiation.
`double`	Displays double-precision representation of a symbolic expression.
`dsolve`	Symbolic ODE solver.
`expand`	Expands an algebraic expression.
`factor`	Factor a polynomial.
`int`	Symbolic integration; either definite or indefinite.

`limit`	Finds two-sided limit, if it exists.
`pretty`	Displays a symbolic expression in a nice format.
`simple`	Simplifies a symbolic expression.
`subs`	Substitutes for parts a a symbolic expression.
`sym` or `syms`	Create symbolic variables.
`symsum`	Performs a symbolic summation, possibly with infinitely many entries.
`taylor`	Gives a Taylor polynomial approximation of a given order at a specified point.

Graphics Commands

`contour`	Plots level curves of a function of two variables.
`contourf`	Filled contour plot.
`ezcontour`	Easy contour plot.
`loglog`	Creates a log-log plot.
`mesh`	Draws a mesh surface.
`meshgrid`	Creates arrays that can be used as inputs in graphics commands such as `contour`, `mesh`, `quiver`, and `surf`.
`ezmesh`	Easy mesh surface plot.
`plot`	Plots data vectors.
`ezplot`	Easy plot for symbolic functions.
`plot3`	Plots curves in 3-D.
`polar`	Plots in polar coordinates.
`quiver`	Plots a vector field.
`semilogy`	Semilog plot, with logarithmic scale along the vertical direction.
`surf`	Solid surface plot.
`trimesh`	Plot based on a triangulation
`trisurf`	Surface plot based on a triangulation

Special Matlab Commands

`:` Range operator, used for defining vectors and in loops. Type `help colon` for information.

`;`	Suppresses output. Also separates rows of a matrix.
`=`	Assigns the variable on the left hand side the value of the right hand side.
`ans`	The value of the most recent unassigned.
`cd`	Change directory.
`clear`	Clears all values and definitions of variables and functions. You may also use to

clear only specified variables.

`diary`	Writes a transcript of a MATLAB session to a file.
`dir`	Lists the contents in the current working directory. Same as `ls`.
`help`	
`inline`	Define an inline function.
`format`	Specifies output format, e.g. `> format long`.
`load`	Load variables from a file.
`save`	Saves workspace variables to a file.

Matlab Programming

`==`	Is equal?
`~=`	Is not equal?
`<`	Less than?
`>`	Greater than?
`<=`	Less than or equal?
`break`	Breaks out of a `for` or `while` loop.
`end`	Terminates an `if`, `for` or `while` statement.
`else`	Alternative in an `if` statement.
`error`	Displays and error message and ends execution of a program.
`for`	Repeats a block of commands a specified number of times.
`function`	First word in a function program.
`if`	Checks a condition before executing a block of statements.
`return`	Terminates execution of a program.
`warning`	Displays a warning message.
`while`	Repeats a block of commands as long as a condition is true.

Commands for Matrices and Linear Algebra

Matrix arithmetic:

`>  A = [ 1  3 -2 5 ;  -1  -1 5 4 ; 0 1 -9  0]` .... Manually enter a matrix.

```
> u = [ 1  2  3  4]'
> A*u
> B = [3 2 1; 7 6 5; 4 3 2]
```

`> B*A` .... multiply B times A.

`> 2*A`	multiply a matrix by a scalar.
`> A + A`	add matrices.
`> A + 3`	add a number to every entry of a matrix.
`> B.*B`	component-wise multiplication.
`> B.^3`	component-wise exponentiation.

Special matrices:

`> I = eye(3)`	identity matrix
`> D = ones(5,5)`	
`> O = zeros(10,10)`	
`> C = rand(5,5)`	random matrix with uniform distribution in $[0,1]$.
`> C = randn(5,5)`	random matrix with normal distribution.
`> hilb(6)`	
`> pascal(5)`	

General matrix commands:

`> size(C)`	gives the dimensions $(m \times n)$ of A.
`> norm(C)`	gives the norm of the matrix.
`> det(C)`	the determinant of the matrix.
`> max(C)`	the maximum of each row.
`> min(C)`	the minimum in each row.
`> sum(C)`	sums each row.
`> mean(C)`	the average of each row.
`> diag(C)`	just the diagonal elements.
`> inv(C)`	inverse of the matrix.

Matrix decompositions:

`> [L U P] = lu(C)`	
`> [Q R] = qr(C)`	
`> [U S V] = svm(C)`	singular value decomposition.

Bibliography

[1] Ayyup and McCuen, 1996.

[2] E. Johnston, J. Mathews, *Calculus*, Addison-Wesley, 2001