A First Course in Computational Mathematics (MATLAB edition)

Richard Porter

2016

Everybody in this country should learn to program a computer, because it teaches you how to think

—Steve Jobs

1 Introduction to Computational Mathematics

1.1 Motivation

As we go through this course you may ask yourself, why am I doing this ? What has *this* got to do with mathematics ? What am I hoping to achieve ? Is this useful or important ? You may be asking this of yourself right now !

These are natural questions to ask about any course. Prior to coming to University you have probably been given a fairly fixed idea of what mathematics is: the manipulation of algebra which allows a solution to be found to a given problem.

For example, we are used to questions such as: find

$$\int_0^1 \frac{1}{1+x^2}\,dx;$$

or solve the differential equation

$$\frac{dy}{dx} + y = \sin(x), \qquad x > 0$$

for $y(x)$ given $y(0) = 1$. Hopefully you all know how to do these by hand.

Throughout your degree course you will approach advanced topics with a similar emphasis. The concepts will be harder to understand and the mathematics more technical but the idea will be that you will approach a problem and use increasingly sophisticated mathematical techniques, ideas and more complex algebra to provide a solution which you can readily analyse.

So is *all* mathematics like this ? What does mathematical research really involve ? Do academics solve problems in the same way ? Well, sometimes, yes. Certainly, before the advent of computational power this was all you could do. When was this ? Well, there are always been computational methods, even going back to the Greeks (e.g. computation of π). So what we really mean is before the advent of automated computational methods – i.e. electronic computers.

Computational power allows you to do calculations that you couldn't perform any other way. The more power you have the more you can do. So a lot of modern mathematical research (especially in applied mathematics and statistics) runs in parallel with the power offered to them by computers; the more powerful the computer the more mathematics you can do. So how does this happen in practice ?

Here are some simple demonstrations where you cannot work out the answer. E.g. find

$$\int_0^1 \frac{1}{1+x^{\sqrt{2}}}\,dx$$

and solve for $y(x)$ when it satisfies

$$\frac{dy}{dx} + \sin(y) = \sin(x), \qquad x > 0 \qquad \text{with } y(0) = 1.$$

It's not that these problems do not have answers or solutions, just that there are no analytic techniques known to extract them in closed analytic form.

So what do we do ? Do we give up ? No, we find *numerical methods* to solve these intractable problems and develop *algorithms* and write *computer programs* to produce the output we are after.

In fact, the value of the integral is 0.73916951 and the solution to the ODE above is plotted below

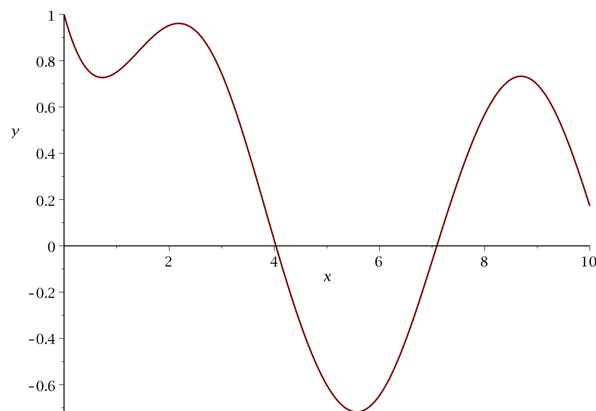


Figure 1.1: Computational solution of the ODE

These are simple examples, and this is where this course is aimed at (you can't run until you can walk).

But where does this lead ? How sophisticated does it get ?

1.1.1 Examples

Here are some simple examples, from the 'Fluids and Materials' group I work in, at the School of Mathematics, University of Bristol.

(1) Prof. Andrew Hogg and his co-workers have developed a mathematical model of ash cloud spreading from volcanic eruptions, used by the Icelandic Met Office:

`https://www.plumerise.bris.ac.uk/help/quickstart/`

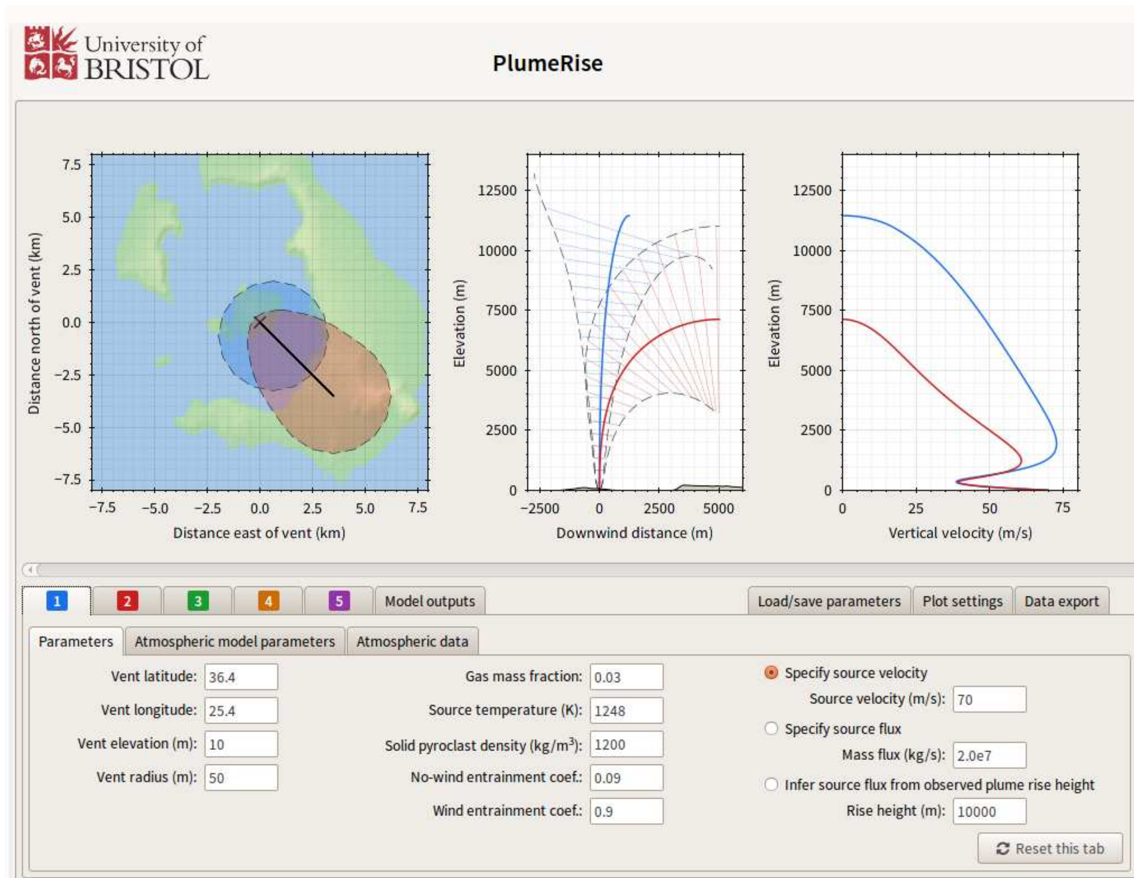


Figure 1.2: Output from a web interface for predicting various quantities of ash cloud spreading

(2) My PhD student, Imogen Noad, has been analysing a model for energy absorption from a wave energy converter called the OYSTER and has computed the optimal dimensions and power control:

(3) Prof. Rich Kerswell performs research into the structures of turbulence and to assist in these calculations, he uses simulations of the governing equations of fluid flow to visualise aspects of turblent flows.

In the movie on the course web page (a snapshot in figure 1.1.1) we see the stream-wise vorticity in a fully 3D flow moving left to right at a Reynolds number of 1900. The visualisation is of a turbuluent 'puff' introduced into the flow. The simulation contains about 1 million degrees of freedom.

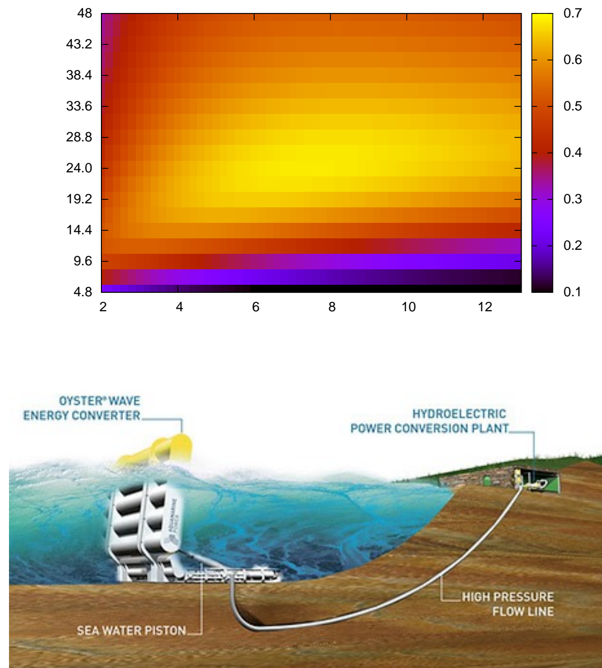


Figure 1.3: On the right a schematic of the OYSTER wave energy harvesting device and on the left a 'heatmap' showing power absorbed as a function of device length (vertical axis) and power control parameter (horizontal axis)

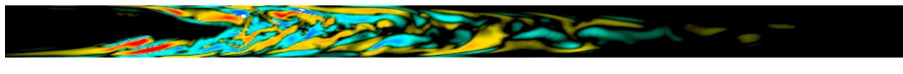


Figure 1.4: A snapshot of a simulation of turbulent fluid flow

1.1.2 Computational mathematics

So what is it that computers do that mathematicians rely on so heavily to make these sorts of calculations ? It's the fact that they can perform many millions or billions of elementary calculations every second; in principle we can do exactly what a computer does just using a calculator, but we cannot match the computer for speed. We just need to give the computer the right set of instructions, as if we were doing it by hand on a calculator and it will automate the process at a massively increased speed.

This is the best way of thinking about how to write **code** (the instructions the computer needs to perform calculations) as a recipe or a simulation of what you would do with manually with only a calculator and a piece of paper.

In order to write code we need to decide upon an **environment** which will allow the computer to understand and interpret the code we write and thus allow the computer to perform the tasks we ask of it.

This is designated the **computer language**: it is the way we talk to the computer. There are many examples of computer languages, many of which you may be familiar with. For example, C, C++, Python, Visual Basic, Pascal, Fortran,...

In this course we almost exclusively stick to one such language: **Matlab**. Now, not everyone uses the same language to code in, not even within Mathematics. In fact, the language you choose to code in turns out it's quite a personal thing. The reason we choose Matlab here is primarily because it has been specifically developed for mathematical applications (particular for matrices and vectors). It is used in industry in technical applications. It is also very good at producing nice graphical outputs. There also are some technical reasons which make Matlab a nice coding language but we will steer clear of those.

All coding language are different and have their own ways of performing certain tasks which set them apart from other languages.

But they all have the same fundamental constructs and in this course we aim to highlight these. In this way, if you ever need to learn a different coding language, you will be able to apply the same principles and simply have to apply the translation of **syntax**.

2 MATLAB

2.1 Getting Started: MATLAB as a big calculator

You can used this guide to help you get started in MatLab.

First, start Matlab (you will need to log-on first of course). On computers running Windows double-click on the Matlab icon, usually situated in:

```
Start>All Programs>Matlab
```

the location of the icon may slightly vary in different machines depending on how the software was installed. The Matlab desktop appears. Now you are ready to begin.

You can either work interactively on Matlab or you can write a script containing a sequence of instructions and run it afterwards (we cover this in Week 2). Here you will learn the basic interactive tools that you need to know to perform any numerical simulation in Matlab.

The Matlab desktop contains various windows. The most important is the central 'Command Window'. Click on it; you will be prompted by

```
>> |
```

The bar will be blinking. We call » the **command prompt** or **command line**. Each time you see this prompt, Matlab is ready to receive instructions.

The basic arithmetic operations are performed with the binary symbols `+`, `-`, `*`, `/`. For example,

```
>> 2 + 3 <enter>
ans =
    5
>> |
```

Similarly,

```
>> 4.5 - 6 <enter>
ans =
    -1.5000
```

```
>> 4.3*7.5 <enter>
ans =
    32.2500
>> 2/3 <enter>
ans =
    0.6667
>> |
```

Powers are taken using the operator ^; therefore, the syntax to compute 2^3 is

```
>> 2^3
ans =
    8
>> sqrt(2)
ans =
    1.4142
```

Note: from now on, we will suppress `<enter>` and the subsequent command prompt (» |) on the understanding that the command-line finishes with you pressing the return key.

Now try the following:

```
>> 2\3
ans =
    1.5000
```

The backslash is equivalent to the operations

```
>> (1/2)*3
ans =
    1.5000
```

Note: We can use the parentheses as in normal arithmetic. The backslash will become useful when we will introduce matrices.

2.2 Variables

The fundamental objects with which Matlab works are called **variables**. Their concept is slightly different from the definition that we are used to in mathematics. Numbers are stored in the computer's memory and we label that part of the computer's memory with a letter or string of characters (a **variable name**).

The syntax used to assign a value to a variable is simply

```
>> a = 2.5e-3
a =
    0.0025
```

Note: The notation `2.5e-3` stands for 2.5×10^{-3}.

Note: The symbol "=" means "assign", not "equal". This is a subtle but very important difference between the language of computing and the language of mathematics.

For example, if you type

```
>> a = a + 3
a =
    3.0025
```

this means "assign to the variable `a` its previous value plus 3." It does not mean the equation $a = a + 3$ so $3 = 0$!

If you like, there is a draw in a filing cabinet which is labelled "`a`". The command above means: open the draw with the value of `a` in it, add three, then put it back into the draw labelled `a`.

2.2.1 Arithemetic with variables

The usual arithmetic operations apply to variables in the same way as they do to numbers:

```
>> b = 4.5;
>> a*b
ans =
    13.5113
```

Note: Each time we press `<enter>` the output displayed is the content of a variable; if we do not allocate the result anywhere, the output is stored in the variable `ans`.

Note: A semicolon at the end of a statement suppresses Matlab from displaying the output.

2.2.2 Variables types

Let's go through this example which uses the Matlab functions `tanh` and `atanh` (tanh and inverse tanh).

```
>> s = 2
s =
    2
>> s = tanh(s)
s =
    0.9640
>> s = atanh(s)
s =
    2.0000
>> s = atanh(s)
s =
    0.5493+1.5708i
>> s = tanh(s)
s =
    2.0000+0.0000i
```

In step 1, s is assiged as an **integer**, but in step 2 it is changed into a **floating point number** or a **decimal number**. It remains so in step 3 even though this is the inverse of step 2. In step 4, it is changed into a **complex number** and in step 5, it remains complex, even though it is the inverse of step 4.

Note: In other computer languages, the **variable type** is a big deal, and once assigned can never be changed. This is due to the way the computer allocates storage space for that number (to carry on the analogy, the size of the draw has to fit the variable you want to put in it otherwise the filing cabinet gets jammed.)

2.3 Variables: storing vectors

We can assign not only scalars to variable names but also vectors, most commonly referred to as **arrays** in computing. You may think of an array as a container storing a collection of numbers. (By analogy with above, the array is like a filing cabinet with many draws, each draw being part of the array).

In the following examples we give to such a container the name "`a`":

```
>> a = [1 2 3 4 5]
a =
    1     2     3     4     5
```

Note: As we saw in the previous example, in Matlab we can take a variable name previously used for a scalar and assign it to a vector, or array. Other programming languages can be fussy about this cavaliar approach to storage.

Typing

```
>> a = [1,2,3,4,5]
a =
    1    2    3    4    5
```

or

```
>> a = 1:5
a =
    1    2    3    4    5
```

produces the same output. This is one of the annoying things in languages such as Matlab; there are several different ways of doing the same thing which have been designed with a particular purpose in mind.

Usefully, the outcome of writing

```
>> a = 0.5:0.25:2
a =
  0.5000   0.7500   1.0000   1.2500   1.5000   1.7500   2.0000
```

is a vector whose elements are equally spaced with a distance of 0.25. Formally this says: start at 0.5, and generate numbers in steps of 0.25 until you get to 2.

The location of an element of an array is identified by an integer index. (It's like the draw number in the filing cabinet).

For example, if we want to extract the second number in the vector `a` we type

```
>> a(2)
ans =
    0.7500
```

2.4 Checking on the values of variables in Matlab

The top right box lists all active variables and shows to what values they are currently assigned. This can be a very useful **debugging** tool.

If you want to clear the value of a variable (unassign the variable), type (to clear `a`)

```
>> clear a
```

To unassign values of all variables (a bit like a 'reboot'), type

```
>> clear all
```

To clear the main display area

```
>> clc
```

2.5 Formatting output in Matlab

By default Matlab displays only 5 significant figures. It is quite often useful to see more figures. In Matlab type:

```
>> format long
```

The other format options are `short` (default), `shortE` and `longE` where the last two employ the exponential representation of the number (e.g. `2E-2` to mean 0.02).

2.5.1 More on vectors/arrays

```
>> b = [0.8 1 43 3 0.5 1e-2 5];
>> a - b  <enter>
ans =
   -0.3000   -0.2500   -35.0000 -1.7500   1.0000   2.6400   -3.0000
```

The subtraction is performed element-by-element. More precisely, the output is a vector whose j-th entry is `a(j) - b(j)`. Vector addition works in the same way.

Important: When it comes to multiplying or dividing vectors by vectors we need to be a bit more precise about what we mean. That is for 2 vectors a and b, say, the operation ab makes no sense as it does with scalars.

If you want ab, a/b, a^b to mean each each element of `a` be multiplied, divided or raised to the power of by the corresponding element of `b` then you must put a dot before the operator; more precisely you need to use `.*`, `./` and `.^` respectively.

For example

```
>> a = [1 2 3];
>> b = [1 2 4];
>> a.*b
ans =
    1   4   12
>> a./b
ans =
    1.0000  1.0000  0.7500
>> a.^b
ans =
    1   4   81
```

Note: Again, integers are changed into decimals when the need suffices.

If you want to multiply or divide a vector by a scalar, which is a conventional operation, you can simply use conventional operators `*` and `/` (i.e. you do not need to type `.*` etc)

```
>> 2*a
ans =
     2    4    6
```

2.6 Matlab's own functions

Matlab has built-in functions that you can apply to both scalars and arrays (this is not conventional in mathematics). The Matlab statement that evaluates the sine of each element of a vector is

```
>> c = [1 2 3]
>> sin(c)
ans =
    0.8415    0.9093    0.1411
```

Other standard build-in functions that use the same syntax include `exp`, `cos`, `tan` and `log` (for the natural logarithm). Matlab has many other build-in functions. You can browse them by clicking on *fx*, usually located to the bottom left of the command Command Window.

2.7 Plotting in Matlab (and arrays)

Vectors provide a natural tool to plot the graph of a function. The graph is just the union of a number of short line segments joining points (x, y) in space.

That is, we want to form a collection of points x and a collection of the same number of points y to form a collection of points (x, y) which we can use to create the graph.

Suppose we want to plot $\sin(x)$ between 0 and 2π. First we discretise the abscissa by creating a vector whose elements are equally spaced in the interval $[0, 2\pi]$. The separation between two consecutive points should be small compared to the length of the interval:

```
>> x = [0:2*pi/100:2*pi]
```

This vector is made of 101 elements; the distance between two consecutive points is $2\pi/100$. Do not forget the semicolon after the statement! (otherwise you have 101 values outputted to the display !)

Note: that the constant π is pre-defined in the variable `pi`. The ordinates of the graph are easily computed

```
>> y = sin(x);
```

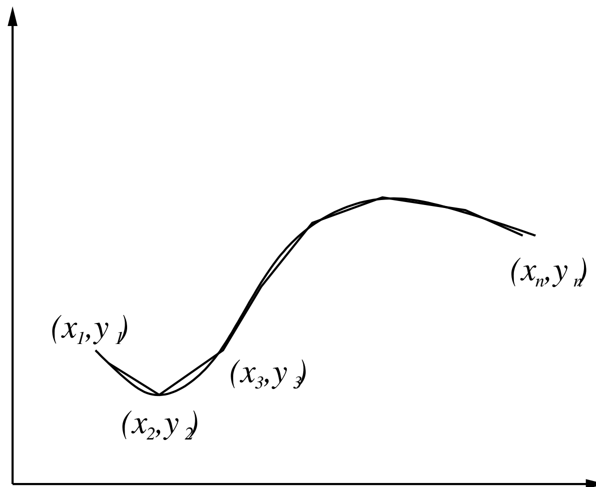


Figure 2.1: Plotting curves computationally are just the union of short line segments.

so that y is the array containing the values of the $\sin(x)$ evaluated at the each of the members of the array.

The basic instruction to plot the graph is

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

Annoyingly, again, you can achieve the same goal using the 'shortcut'

```
>> fplot(@sin, [0 2*pi])
```

but I include this purely for information. We will stick here to plotting arrays.

You may also plot more function in the same graph. For example,

```
>> z = cos(x)
plot(x,y,x,z)
```

The graph will appear in a separate window and will look like the plots in Fig. 2.2.

You may also want to save your graph. Here is how to do it. You go on the menu of the window with the figure — not the main `Matlab` frame. Then click on `File > Save as`. Finally, choose the most appropriate format (see Figs. 2.3 and 2.4).

2.8 Problems

Follow the lecture notes in §1.2.1 on how to get Matlab started and then follow the exercises listed below. You should try all exercises on the sheet.

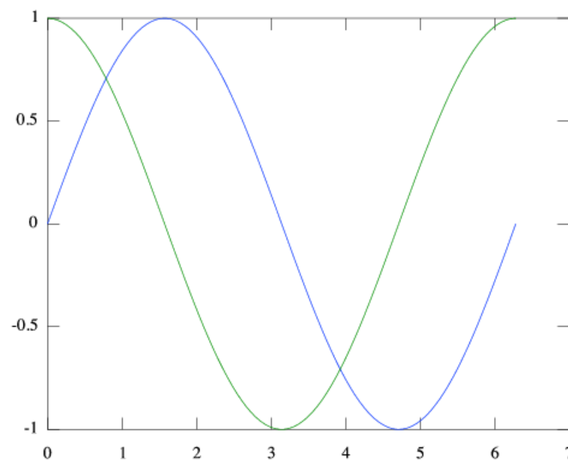


Figure 2.2: Plots of $\sin(x)$ and $\cos(x)$ in the interval $[0, 2\pi]$

There is no work to hand in from Week 1. Consequently, you do not need to save or print any work this week.

This worksheet is not supposed to be particularly challenging or interesting. It's been designed to get you famaliar with some of the basic things Matlab does.

1. Calculator operations: Try some simple calculations in the Matlab Command Window. Try adding, subtracting, multiply and dividing the two numbers 2.3 and 1.2

2. Order of priority. Try to work out what the computer decides takes order of priority of the operators `*`, `/`, `-` and `+` when doing multiple algebraic operations. Try
```
>> 2+3/5
>> (2+3)/5
>> 2+3*5/3
>> 2+3/5*3
>> 2/3+5
>> 2/3/5
>> 2/(3/5)
```
   Rule: If in doubt, use brackets !

3. Complex numbers. Matlab can cope with complex numbers. Try these commands
```
>> sqrt(-1)
>> sqrt(-1)^2
>> abs(3+4*i)
>> exp(pi*i)
>> i^i
```
   Note: `i` is reserved for the imaginary unit $i = \sqrt{-1}$ and `pi` is reserved for π.

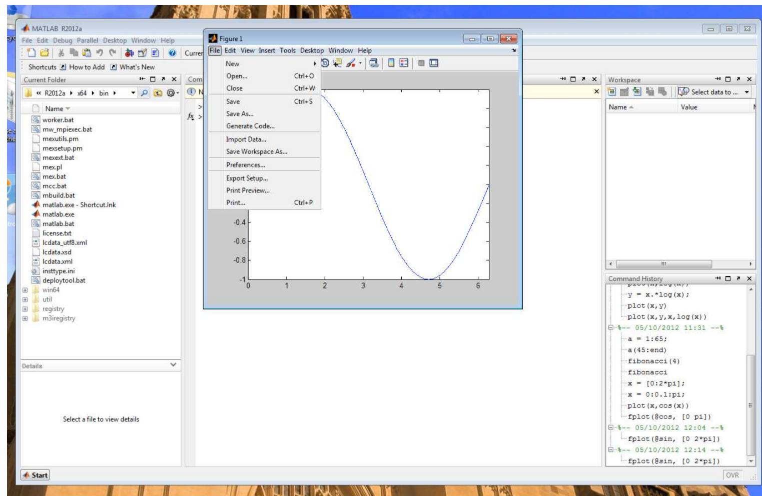


Figure 2.3: Saving a graph (a)

4. Formatting output. Get Matlab to output » `exp(-pi/2)` in the 4 different formats, `format short`, `shortE`, `long`, `longE`.

5. Basic built-in mathematical functions. Try `cos(pi)`, `sin(pi)`, `tan(pi)`, `cot(pi)` in the command line. Also `log(exp(3))`.

6. Variables. Try to get your head around the use of the equals sign to mean "assign the computation to the right of the equals sign to the variable on the left". It's not that difficult.

   Do the following in your head first (or: what is the value of the variables `a`, `b` and `c` at the end of the following list of commands ?)

   ```
   >> a = 1
   >> b = 2
   >> c = b-a
   >> a = b
   >> b = c
   >> c = b-a
   >> a = b
   >> b = c
   >> c = b-a
   ```

   and then confirm the output in Matlab.

7. Manual iteration. Try

   ```
   >> s = 1;
   >> s = sqrt(1+s)
   >> s = sqrt(1+s)
   ```

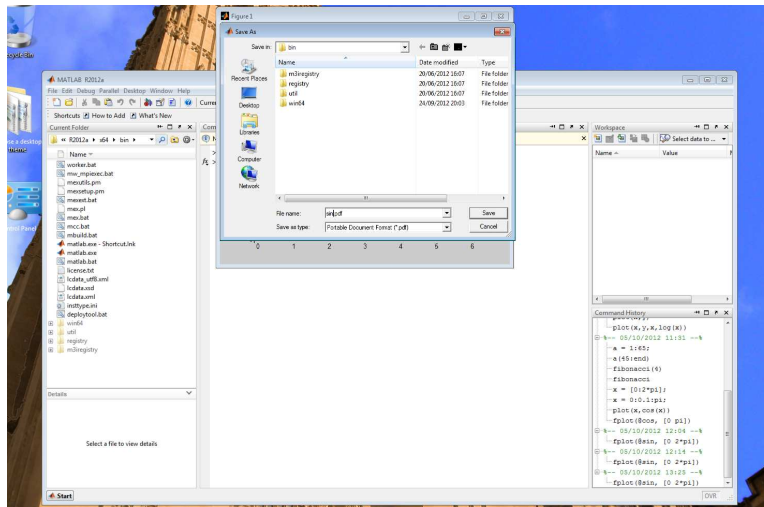


Figure 2.4: Saving a graph (b)

and repeat... use the up and down arrows to recall the previous commands.

Note: The use of ; to suppress the output.

Can you work predict the number that the output is tending to ?

8. Arrays. In Matlab, define an array `a` with elements 1, 2, 3 and another `b` with elements 3, 2, 1. Try adding and subtracting the arrays. Use Matlab's element-by-element operators `.*`, `./` and `.^`

```
>> a.*b
>> a./b
>> 1./a
>> b.^(-1)
```

to convince yourself that the output is what you expect it to be.

What does the command » `cos(a).*cos(b)` do ? Test that it is the same as

```
>> 0.5*(cos(a+b)+cos(a-b))
```

and the same as

```
>> (cos([4,4,4])+cos([-2,0,2]))/2
```

and

```
>> cos(1:3).*cos(3:-1:1)
```

Convince yourself why each of these is the same as the others.

9. Plotting. Plot the functions $\sinh(x)$ between $-2 \leq x \leq 2$ with 201 plotting points.

Now add plots of the functions $\cosh(x)$ and $\tanh(x)$ on the same graph, again between $-2 \leq x \leq 2$.

10. How would you plot $\sin^2(x)$ and $\sin(x^2)$ over the interval $0 \leq x \leq 2\pi$. ? Remember: `x` is stored as an array in the computer.

3 Week 2

3.1 Matlab scripts (.m files)

A **script** allows you to assemble a collection of commands in an ordered list to be read in line-by-line in Matlab. By writing the script in a file it can be saved, opened and edited like any normal text file. The advantages of this are clear.

How do you write a script file ?

Easiest is to use the Matlab interface:

- Click on the New file icon in the main Matlab window. You will get a blank file in a new text editor window
- Type in a list of commands, line by line as if you were typing them directly into the Matlab interface.
- Save the file as a .m file (which is the default).
- You can close, reopen the file, edit etc etc.
- Matlab will show you where in your filesystem (i.e. in which folder) you file has been saved. This is important to note if you save files in different folders for e.g. as you risk misplacing saved files !

3.1.1 Example

The area of a triangle with sides of length a, b, c is (Heron's formula)

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \qquad \text{where} \qquad s = \frac{1}{2}(a+b+c) \tag{3.1}$$

So in Matlab, open a new file and type in the lines

```
a = 3;
b = 4;
c = 6;
s = (a+b+c)/2;
Area = sqrt(s*(s-a)*(s-b)*(s-c))
```

and save as triangle.m (you do not need to close after editing; indeed you should leave it open so you can correct or amend in needed).

Now from Matlab type

```
>> triangle
Area =
    5.3327
```

(Note: if you change the current working folder in the Main Matlab window, Matlab will not be able to 'see' your saved file triangle.m and will therefore not run the script and instead give you an error:

```
>> triangle
Undefined function or variable 'triangle'.
```

It is best, at least until you are confident with your Matlab skills not to change any of the folders set by Matlab.

3.2 Principles of Programming: Automating repetitive tasks

This is at the heart of most computational tasks as we shall see throughout the course.

For example, developing simulations which advance in small steps in time, summing series, generating simulations of 100's or 1000's of numerical experiments.

3.2.1 Sequences

A **sequence** is an ordered list of numbers, s_n, say, for $n = 0, 1, 2 \ldots$. The subscript n is the **index**, which acts to label the sequence and the sequence can be finite or infinite. It does not need to start at 0, or advance in steps of 1. For example, we can define the sequence $a_2 = 1$, $a_0 = 0$, $a_{-2} = -1$.

A sequence may be defined explicitly, e.g. $s_n = 1/n$ for $n = 1, 2, \ldots$.

A sequence may also be defined **iteratively** by a **recurrence relation** in which an element of the sequence s_n, say, depending on other previous of the sequence.

In general we may write

$$s_n = f(s_{n-1})$$

(a **one-term** recurrence relation),

$$s_n = f(s_{n-1}, s_{n-2})$$

(**two term**) and so on.

3.2.2 Example

Define

$$s_{n+1} = \frac{1}{2}\left(s_n + \frac{2}{s_n}\right), \qquad n = 0, 1, 2, \ldots \tag{3.2}$$

Clearly need a 'starting value'. Here s_0 is needed, then s_1 is defined by (3.2) with $n = 0$, which then allows s_2 to be found with $n = 1$ and so on.

Let's choose $s_0 = 1$.

We can do this manually (in Matlab) using

```
>> s = 1;
>> s = 0.5*(s+2/s)
    1.5000
```

Note comments from Week 1: line 2 overwrites the previous value of s with the new value being $\frac{1}{2}(s + 2/s)$ and so in the first line s represents s_0 and in the second line s represents s_1 and the value of s_0 has been 'forgotten'.

Now we repeat

```
>> s = 0.5*(s+2/s)
    1.4167
```

which gives s_2 and so on.

We see the sequence appears to be converging.

This is a repetitive task. We are doing the same thing over and over again. So can we automate this repetitive task ?

3.2.3 For loops (or do loops)

Every programming language has a capacity to 'loop'; to repeat a single task or a set of tasks a set number of times.

Example

Let's say we want to iterate the recurrence relation (3.2) 5 times for e.g with $s_0 = 1$.

We write a new script which we will save as `rec1.m` which contains the lines

```
s = 1;
for n=1:5
    s = 0.5*(s+(2/s))
end
```

Note: you do not need ; to suppress the output on the `for` or `end` lines.

Note: The **indentation** is not needed, but is standard programming practice as it allows the structure of code to be easily identified (when code becomes much more complicated)

The lines `for n=1:5 ... end` means that the computer reads all the lines in between (in this case just one line `s = 0.5*(s+(2/s))`) five times, the first time with the variable `i` taking the value 1 first time round, then 2 and so on.

Note: In this e.g., the variable `i` plays no part apart from acting as a **counter**.

Note: Remember to use

```
>> format long
```

to get more than just 4 decimal places.

3.2.4 Some analysis

In (3.2) if we assume $s_n \to S$, say, as $n \to \infty$ (as the computation suggests) then $s_{n+1} \to S$ also and (3.2) becomes

$$S = \frac{1}{2}\left(S + \frac{2}{S}\right), \qquad \Rightarrow \qquad S^2 = 2 \qquad \Rightarrow \qquad S = \pm\sqrt{2}$$

Agrees with computations !

Q: If $s_0 < 0$ would you expect sequence converge to $-\sqrt{2}$? Why ?

Note: This a method for finding square roots ! How does a calculator find square roots of numbers ? Could you easily adapt the recurrence relation to find $\sqrt{a}$ where a is any positive number ?

3.2.5 Example: Fibonacci numbers

I.e. the sequence $1, 1, 2, 3, 5, 8, 13, \ldots$.

Defined by the two-term recurrence relation

$$F_{n+2} = F_{n+1} + F_n, \quad \text{for } n = 0, 1, 2, \ldots \text{ and where} \qquad F_0 = 1,\ F_1 = 1 \tag{3.3}$$

How do we code this in Matlab ?

- Open a new file in which we will write our Matlab script, to be saved as fib.m (say)
- Start to think about how to develop an **algorithm** (a list of instructions) which are ordered to automatically carry out the process of adding the previous two numbers together. Cannot simply ‘overwrite’, one number with a new number. Here, in order to proceed, you need to know **two** numbers to define the next in the sequence

- I.e. think about how many numbers you need to store; how many variables you need. Here, it is 3 – the two previous numbers in the sequence and the new number.

The Matlab script could look like this:

```
s = 1;           % sets F_0
t = 1;           % sets F_1
for n=2:10       % loop up to 10
    u = t+s      % Defines F_n
    s = t;
    t = u;
end
```

Note: `%` are used as **comments**. Everything that follows a `%` is ignored by Matlab but helps the person reading the script understand what is going on.

It is always advisable to comment your code !

What is going on in the 3 lines between `for` and `end` ? Well, first we note that `n` is again a passive counter.

- In the first loop `n = 1` and `u` is set to be the sum of `s` and `t`, as defines the recurrence relation (3.3). So at this point `u` is F_2.
- Once F_2 is known, for the next iteration for F_3 you will not need F_0. Here, `s` stores the value of F_0, so we are saying that `s` is redundant.
- If we let `s` be `t` and then let `t` be `u` then `s` now stores F_1 and `t` stores F_2.
- This means we have freed up `u` whilst `s` and `t` have become the previous two members of the sequence. That is, we are ready to apply exactly the same procedure again.
- That is, we can put this into a loop.

3.3 Example: Golden ratio and continued fractions

Consider the following definition

$$\phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \dots}}} \tag{3.4}$$

This is an example of a **continued fraction**.

Q: How can we find ϕ ? **A:** By a recurrence relation.

Assume ϕ_0 is given (!) and define

$$\phi_1 = 1 + \frac{1}{\phi_0}$$

and then

$$\phi_2 = 1 + \frac{1}{\phi_1} = 1 + \frac{1}{1 + \frac{1}{\phi_0}}$$

and so on... I.e. we have defined the recurrence relation

$$\phi_{n+1} = 1 + \frac{1}{\phi_n} \tag{3.5}$$

and *if* ϕ_n converges we might expect $\phi_n \to \phi$ as $n \to \infty$. There's a big if in this. And how does ϕ depend on ϕ_0 ?

3.3.1 The Matlab script

Open a new file and enter the script

```
p = 1;                % This is phi_0
for n = 1:10          % Doing 10 interations
    p = 1 + (1/p)     % can overwrite phi_n by phi_{n+1}
end
```

Save as cfrac.m. Then run in Matlab:

```
>> cfrac
```

and you will see a list of iterates appear.

3.3.2 Analysis of the limit

What does ϕ tend to ? Assume $\phi_n \to \phi$ and then (3.5) reads

$$\phi = 1 + 1/\phi, \qquad \Rightarrow \qquad \phi^2 - \phi - 1 = 0, \qquad \Rightarrow \qquad \phi = \frac{1 \pm \sqrt{5}}{2}$$

Two values, one positive and one negative. If $\phi_0 > 0$ then easy to see all $\phi_n > 0$ and so will tend to +ve root.

$\dfrac{1+\sqrt{5}}{2}$ is called the **Golden Ratio**.

3.3.3 Connection with Fibonacci numbers

The Fibonacci recurrence relation (3.3) divided F_{n+1} can be written

$$\frac{F_{n+2}}{F_{n+1}} = 1 + \frac{1}{\frac{F_{n+1}}{F_n}}$$

and if we let $\phi_n = F_{n+1}/F_n$ then we arrive at (3.5). So the Golden ratio is the limit of the ratio of two consecutive numbers in the Fibonacci sequence.

3.4 Matlab: Printing your work

Note: You will need to print your work in order to hand in homeworks to your tutor.

In Matlab, from your open script (.m) file, goto File and click on the tab Publish work. This will produce a new window which will list your script and the output of your script. You can then Print and collect from printer.

You will find it useful to use a comment at in line 1 of your script with your name on it so you can identify your printout !!

Use double %% Signs to produce a title (see §2.5.2 later).

3.5 Example: Approximating π

3.5.1 Derivation

Based on the method of inscribed polygons (Archimedes' Algorithm). Take a circle of radius 1. It has a perimeter of 2π.

Step 1: Inscribe a square (a regular 4-sided polygon). Using geometry, each side has length $d_0 = 2/\sqrt{2} = \sqrt{2}$. Hence perimeter is $4\sqrt{2}$.

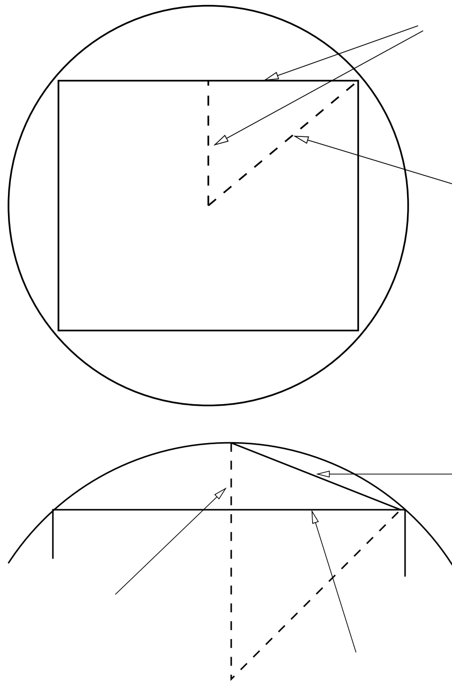
We know the perimeter of the circle must be greater than that of the inscribed square, so we know $4\sqrt{2} < 2\pi$.

In fact $4\sqrt{2} \approx 5.657$.

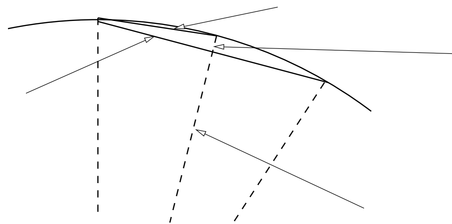
Step 2: Divide each side of the square into two equal segments to form an 8-sided regular polygon, inscribing the circle. Using geometry, we find the length d_1 of the side of the octagon is

$$d_1 = \sqrt{\left(\frac{1}{\sqrt{2}}\right)^2 + \left(1 - \frac{1}{\sqrt{2}}\right)^2}$$

whilst the perimeter is $8d_1$.



Step n: We think of developing this idea. Now assume a regular 2^{n+2}-sided polygon with sides of length d_n. Geometry gives us



$$d_n = \sqrt{\left(\frac{d_{n-1}}{2}\right)^2 + \left(1 - \sqrt{1 - \left(\frac{d_{n-1}}{2}\right)^2}\right)^2}$$

which can be simplified to

$$d_n = \sqrt{2 - \sqrt{4 - d_{n-1}^2}} \tag{3.6}$$

and the perimeter is $2^{n+2}d_n$.

So (3.6) is a recurrence relation with $d_0 = \sqrt{2}$ which defines the sequence $d_1, d_2, \ldots$.

We expect $2^{n+2}d_n \to 2\pi$ (or $2^{n+1}d_n \to \pi$) as $n \to \infty$ and for $2^{n+1}d_n < \pi$.

3.5.2 Matlab script

```
%% Richard Porter -- approximation of pi
d = sqrt(2);                         % This is d_0
for n=1:10                           % Iterate from d_1 up to d_10
    d = sqrt(2-sqrt(4-d^2));         % Iterative step
    circ = d*(2^(n+1))               % print approx to pi
end
```

3.6 Problems

1. Write out the following Matlab script which you should save as triangle2.m which prints out the three interior angles (in degrees) of the triangle with sides a, b and c using the cosine rule:
$$r_1^2 = r_2^2 + r_3^2 - 2r_2 r_3 \cos\theta_{23}$$
where r_1, r_2, r_3 are any of the three sides a, b, c and θ_{23} is the angle between sides r_2 and r_3.

```
a = 3;
b = 4;
c = 6;
thetaab = 180*acos((a^2 + b^2 - c^2)/(2*a*b))/pi
thetabc = 180*acos((b^2 + c^2 - a^2)/(2*b*c))/pi
thetaca = 180*acos((c^2 + a^2 - b^2)/(2*c*a))/pi
```

Run the script (by typing » `triangle2` in the Matlab command line).

Try editing the file and changing the values of `a`, `b` and `c` to test the script. Can you make it fail ? Why ?

2. a) Write a script (call it rec2.m) which outputs the first 5 iterates $s_1, s_2, \ldots, s_5$ to the recurrence relation
$$s_{n+1} = \frac{2}{3}\left(s_n + \frac{1}{s_n^2}\right), \qquad n = 0, 1, 2, \ldots$$
with $s_0 = 1$. Follow the example in §2.2.3 of the notes, using `for ... end` loops.

   b) Run your code by typing >> `rec2` into the Matlab command line. Make sure you type `format long` in the command line so Matlab displays enough decimal places. (Publish this and print it.)

   c) Edit your script to increase the number of iterates and find how many iterates it takes to converge to all 15 decimal places shown. (Write your answer on your printout.)

d) Can you determine the value to which the iterates are converging ? (Add your working to your printout.)

3. Follow the steps (a)–(d) in Exercise 2 for the recurrence relation

$$s_{n+1} = s_n + \cot(s_n), \qquad \text{for } n = 0, 1, 2, \ldots$$

with $s_0 = 1$. Give this one a new name (say rec3.m).

4. Follow the steps (a)–(d) in Exercise 2 for the recurrence relation

$$s_{n+1} = \sqrt{1 + s_n}$$

with $s_0 = 1$.

5. Consider the following two-term recurrence relation

$$T_{n+2} = 2xT_{n+1} - T_n, \qquad \text{for } n = 0, 1, 2, 3, \ldots$$

with $T_0 = 1$, $T_1 = x$ and x is a variable whose value we assume will be assigned in the script we write.

a) Follow §2.2.5 of the notes, using the file fib.m as a template, and write a script which outputs to the screen the iterates T_2 to T_{10}.

Note: you need to set the value of `x` within the script (i.e. the first line is `x = 1;` for e.g.) as the script does not know about values of `x` set within the main command window.

b) Run the script with `x = 1;` and show that all iterates are 1. How could you have predicted this ?

c) Run the script with `x = 0;` and again explain how you could predict the answer.

d) Run the script with `x = 0.4` and with `x = 1.4` and describe the qualitative difference in behaviour of the iterates. (Publish and printout the two different runs of your code and add comments to the printouts)

6. In the lecture we described a method for approximating π using *inscribed* polygons. The approximations to π were all lower bounds on the exact value of π as it was clear geometrically that the perimeter of the 2^n-sided polygons were all less than that of the circle they inscribed.

A similar approach to approximating π is to use *circumscribed* polygons; I.e. approximating the circle using a 2^n-sided polygon which encloses the circle.

You are given that the recurrence relation, being the analogue of §2.5 in Week 2 notes, is

$$d_{n+1} = \frac{2(\sqrt{d_n^2 + 4} - 2)}{d_n}, \qquad n = 0, 1, 2, \ldots$$

with $d_0 = 2$. The perimeter of the 2^{n+2}-sided polygon with side d_n circumscribing the circle of radius 1 is $2^{n+2}d_n$ and thus the approximation to π is $2^{n+1}d_n$

a) Write a script to implement this recurrence relation to approximate π at each step. Go up to d_{10}.

   [Hint: you can adapt two lines of the file pi1.m in §2.5.2 of the notes.]

b) Run your script to confirm that the iterates tend to π from above. (Publish and printout this)

c) Derive the recurrence relation above.

7. Continued fractions are pretty amazing. Here's one (there are others) that gives the value of π:

$$\pi = 2 + \cfrac{2}{1 + \cfrac{1}{\frac{1}{2} + \cfrac{1}{\frac{1}{3} + \cfrac{1}{\frac{1}{4} + \dots}}}}$$

Once we truncate this at the nth term, we can write the sequence as $s_0 = 1$, $s_{j+1} = (1/(n-j)) + (1/s_j)$ for $j = 0, 1, 2, \dots, n-1$ and the estimate to π will be $2 + 2/s_n$ (check this).

Write a Matlab script to implement this recurrence relation.

Week 3

3.7 Finite series

A finite **series** is a finite sum of a sequence of numbers a_j, $j = 1, \ldots, n$ say. We write

$$s_n = a_1 + a_2 + \ldots + a_n \equiv \sum_{j=1}^{n} a_j$$

For e.g, if $a_j = j$ then

$$s_n = 1 + 2 + 3 + \ldots + n = \sum_{j=1}^{n} j = \frac{1}{2}n(n+1) \tag{3.7}$$

(The last step is a well-known result which can be derived geometrically or is often used as an example of proof by induction.)

For a general $1 \leq k \leq n$ we write

$$s_k = \sum_{j=1}^{k} a_j$$

and it follows that $s_1 = a_1$ (for $k = 1$) and that

$$s_k = \sum_{j=1}^{k-1} a_j + a_k = s_{k-1} + a_k$$

Thus, the summing of series can be defined by a recurrence relation in which you simply add the next term to the sum at each step.

In e.g. (3.7) the recurrence relation is $s_1 = 1$ and $s_k = s_{k-1} + k$.

3.7.1 Matlab script

```
n = 10;              % setting n to 10 here, so it's easy to change
s = 1;               % sets up a variable to store the running total of the sum
for k=2:n
    s = s + k;       % Note: k is both a counter and being used in the loop
end
s                    % No semi-colon on this line -- output s to screen
```

E.g. in Matlab, save the script as sum1.m and run the script with

```
>> sum1
s =
    55
```

3.8 Infinite series

An **infinite series** is the sum of all elements of an infinite sequence.

E.g.

$$\sum_{j=1}^{\infty} \frac{1}{j^2} = S, \qquad \text{say} \tag{3.8}$$

In fact, it is known that

$$S = \frac{\pi^2}{6}$$

and so we may use the series as a means of calculating π as

$$\pi = \sqrt{6 \sum_{j=1}^{\infty} \frac{1}{j^2}}$$

There's a problem. We cannot continue summing the series to infinity, no matter how fast your computer is.

This gives rise to an important principle in Computational Mathematics: the notion of **approximation**. That is, not being able to perform certain numerical tasks exactly and having instead to accept that an approximation might be good enough. We already experienced this principle last week in the approximation of π.

Q: How do we approximate S ? **A:** It's fairly obvious that since $1/j^2 \to 0$ as $j \to \infty$ that if we stopping summing the series at some large enough value we will have a *finite series* which will be an approximation to S.

$$s_n = \sum_{j=1}^{n} \frac{1}{j^2}$$

and surmise that $s_n \to S$ as $n \to \infty$.

Note: We say the series S has been **truncated** after n terms and s_n is called the n**th partial sum**.

3.8.1 Matlab script

(To approximate π.) We follow the steps in §3.1.1 in successively adding terms to the series up to the nth term.

```
n = 100;
s = 1;
for k=2:n
    s = s + (1/k^2);
end
sqrt(6*s)
```

Call this `sum2.m` and run from Matlab with » `sum2`

3.9 Principles of Programming: Arrays

Arrays are indexed storage systems for storing sequences or multiple variables. All computing languages has capacity for storing information in arrays.

In mathematics, the analogue of an array is a vector (or matrix). So in mathematics the vector $\mathbf{x}$ has n **elements** and $\mathbf{x}$ encodes the elements as being stored as a collection $(x_1, x_2, \ldots, x_n)$.

Arrays are exactly the same. They have integer lengths and indexed elements. So an array `x` of length (often called **dimension**) n has elements `x(1),x(2),...,x(n)`.

3.9.1 Example of using arrays

Return to the previous example (3.8). In the Matlab script we iterated by overwriting the current partial sum with the new partial sum. But imagine we wanted to keep and store every value of s_k from $k = 1$ to $k = n$, say.

I.e. we want to keep the values $s_1, s_2, \ldots, s_n$. That's n indexed elements and so it is natural to use the array storage system.

How ? Well we need to make sure there is enough storage. So we need to set up an array of the correct dimension – here it is n.

Here's the Matlab script:

```
n = 100;
s = zeros(1,n);               % Creates an array s of dimension n such
                              % that s(1) = 0, s(2) = 0, ... s(n) = 0
s(1) = 1;                     % This is the value of the sum s_1 which we
```

```
                                % need to start the sequence
for k = 2:n
    s(k) = s(k-1) + 1/k^2;      % We are summing the series from 2 to n
                                % but not overwriting variables
end

% Now we are going to do something with the stored variables: a plot

x = 1:n;                        % Creates an array x of dimension/length n
                                % such that x(1) = 1, x(2) = 2, ... x(n) = n
plot(x,s-pi^2/6,'*')
```

Reminder: We used arrays in plotting in week 1

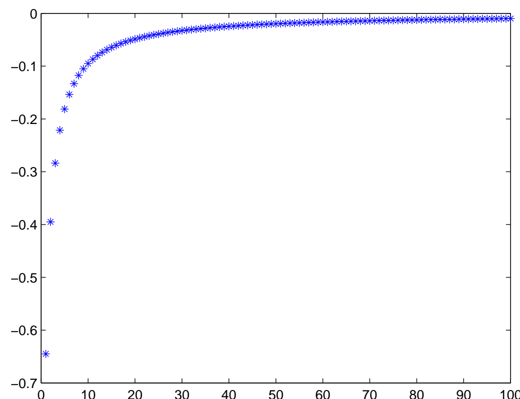
The array `x` hold the values of the integers from 1 to n. The array `s` holds the values of the partial sums s_k for $1 \leq k \leq n$. In Matlab, you can **plot** a graph of points stored in the two arrays in the first two arguments of the `plot` command. The third argument `'*'` is optional and prescribes the **linestyle of the plot**

Remark: Use the help system to look up other plotting options.

When the script is saved (as conv1.m, say) and then run from Matlab's command line

```
>> conv1
```

it produces a graph (see below) which shows that the value of the $s_k - \pi^2/6$ is tending to zero. I.e. you have **numerical evidence** that the series s_k are tending to $S = \pi^2/6$ as $k \to \infty$.



3.9.2 Analysis

Since we have had to approximate an infinite series by truncation, we might like to know how good the approximation is ?

Or, how big does n have to be to guarantee a desired accuracy ?

Remark: This sort of important question is at the heart of the subject of **numerical analysis**. We acknowledge that computers have limitations (here, they cannot sum to ∞) and often can only produce approximation. Often you want to know how good your approximation will be.

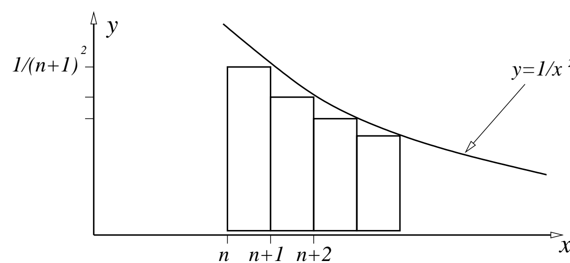
Here we compute $s_n = \sum_{j=1}^{n} \frac{1}{j^2}$ as an approximation to $S = \sum_{j=1}^{\infty} \frac{1}{j^2}$.

The bit we have neglected, the **error** is

$$E = \sum_{j=n+1}^{\infty} \frac{1}{j^2}$$

How big is this ?

We can interpret E as the sum of series of adjacent rectangles each of width 1 and height $1/(j+1)^2$ and positioned at $x = i$ as shown in the figure below. By drawing the curve



$1/x^2$ over the top of the set of rectangular steps we see that E is represented by the area under all the rectangles and this is less than the area below the curve $1/x^2$.

That is, graphically we have

$$E < \int_n^\infty \frac{1}{x^2} dx = \left[\frac{-1}{x} \right]_n^\infty = \frac{1}{n}$$

Exercise: Can you also show that $E > \frac{1}{n+1}$ so that $\frac{1}{n+1} < E < \frac{1}{n}$?

Answer: We can put another curve $y = 1/(x+1)^2$ under the rectangles so that

$$\int_n^\infty \frac{1}{(x+1)^2} dx < E$$

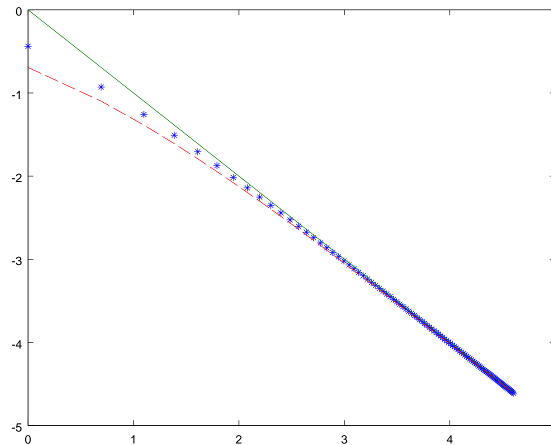
and this gives the result.

Summary: We conclude that if you truncate at 100 the error is in s_{100} is between 0.01 and 0.0101. That's useful.

Note: A more useful plot of the error is made by replacing the final `plot` line of §3.3.1 by the log-log plot

```
plot(log(x),log(s-pi^2/6),'*',log(x),-log(x),'-',log(x),-log(x+1),'--')
```

and we have added two curves given by the analytic bounds above. The gradient of the curve, being -1, indicates the error decays like $1/n$, useful if you cannot find bounds as we have done here.



Exercise: Can we find bounds on the size of $\pi^2/6$?

Use the same methods above to show that $1 < \pi^2/6 < 2$.

3.10 Functions defined by series

Consider the MacLaurin series expansion of the exponential

$$e^x = \sum_{j=0}^{\infty} \frac{x^j}{j!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots \tag{3.9}$$

Of course, there is an exponential function built into Matlab. E.g.

```
>> exp(1)
ans =
     2.71828
```

Imagine there weren't, or you had a similar series which couldn't be written in terms of elementary functions. How about we compute the series ?

- We cannot sum to infinity. So we will have to approximate.
- We can truncate, as we know here that $j!$ goes to zero faster than x^j. I.e. $x^j/j! \to 0$ as $j \to \infty$. Just as well or (3.9) would not be a convergent series.

3.10.1 Matlab script

This example computes the series truncated at the 15th term with $x = 1$. I.e. it should produce an approximation to e.

```
%% Call this myexp.m
n = 15;
x = 1;                          % set x = 1
s = 1;                          % first term in the series is 1
for j = 1:n
    s = s + x^j/factorial(j);   % Uses in-built Matlab factorial
end
s                               % Output s
```

Save and run this script

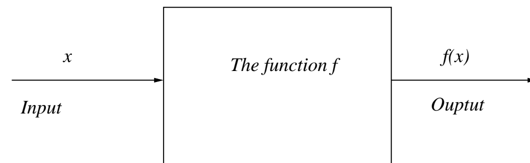
```
>> myexp
s =
   2.718281828458995
```

We can play around changing n and x.

This is OK to a point. But the exponential is a **function** and we have to manually change `x` (and `n`) within the script every time we want to change the value of e^x (and its accuracy).

3.11 Principles of Programming: Functions

What is a function ? In mathematics we are used to writing things like $f(x)$, but what does this actually mean ? So we can regard a function as a 'black box' which turns (a set



of) inputs into (a set of) outputs.

In computer programming languages a **function** uses the same principle; it is a list of instructions contained within a 'black box' (so that's a script to you and me) which turn input into output.

3.11.1 Example: the exponential function

Here's a new Matlab script to replace myexp.m.

```
%% A function, called myexp1
function s = myexp1(x,n)      % Two inputs: x and n, one output: s
s = 1;                        % The rest is the same
for j = 1:n
    s = s + x^j/factorial(j);
end
end                           % the function ends with end
```

Important note: The function name must be the same as the script name. I.e. here, we must save this file as myexp1.m

Now from the Matlab command line

```
>> myexp1(1,15)
ans =
    2.718281828458995
```

It works !

3.12 Products

A close relative of series are products...

E.g. a finite product of the sequence $a_j = j$, $j = 1, \ldots, n$ is

$$1.2.3.\ldots n = \prod_{j=1}^{n} j$$

and this, of course, defines $n!$. Just like with series, in general we define the finite product as

$$s_n = \prod_{j=1}^{n} a_j$$

and it follows that $s_1 = a_1$ and

$$s_k = a_k \prod_{j=1}^{k-1} a_j = a_k s_{k-1}, \qquad k = 2, 3, \ldots, n$$

3.12.1 Example: the factorial

For $s_n = n!$ we have $s_1 = 1$ and $s_k = k s_{k-1}$. Here's a Matlab function (myfact1.m) to compute $n!$:

```
function s = myfact(n)  % input n, output s
s = 1;
for k=2:n
    s = s*k;
end
end
```

Note: We can also consider **infinite products** with all of the remarks made above infinite series applying to them.

3.13 Problems 3: Loops, functions, arrays

1. Follow §3.1 the notes of Week 3 and write a script to compute the series

   $$\sum_{j=1}^{n} j^3$$

   where the value of n should be assigned within the script. Check your answers against the known formula:

   $$\left(\frac{1}{2}n(n+1)\right)^2.$$

2. a) Modify the script in the example in §3.3.1 of Week 3 notes, to compute and store each partial sum s_k for $k = 1, 2, \ldots, n$ in an array, where

$$s_k = \sum_{j=1}^{k} \frac{1}{j}$$

where the only output of the script is a `plot` of $s_k - \ln(k)$ (on the y-axis) against k (on the x-axis).

Note: In Matlab, the natural logarithm of x is `log(x)`.

b) Set $n = 100$ and from your plot convince yourself that $s_k - \ln(k) \to \gamma$ where γ is a constant that you should roughly estimate.

Publish and print the code and the graphical output.

c) Use graphical considerations to prove the following result

$$\ln(n+1) < s_n < 1 + \ln n$$

and show that this is commensurate with your estimate of γ in (b).

3. a) Similar to Exercise 2, write a script to compute and store each partial sum s_k for $k = 1, 2, \ldots, n$ in an array, where

$$s_k = \sum_{j=1}^{k} \frac{(-1)^j}{j}$$

Suppress all numerical output and instead, write your script to plot s_k on the y-axis against k on the x-axis.

b) Convince yourself from varying values of n set within your script that the infinite series $S = \sum_{j=1}^{\infty} \frac{(-1)^j}{j}$ is convergent.

c) Use the MacLaurin expansion of $\ln(1 + x)$ to calculate the exact value of S and compare with your results. You might do this graphically as we did in §3.3.1 of Week 3 notes.

4. In §3.6 of the notes we defined $n!$ by a recurrence relation

$$s_k = k s_{k-1}, \qquad \text{for } k = 2, 3, \ldots, n \text{ with } s_1 = 1$$

a) Download or copy the function `myfact1` which takes arguments (input) of n and outputs $n!$. Test the function.

Note: Remember from Week 3 notes, that you need to save the script with the same name as the function – here `myfact1.m`.

b) Modify your script so that the iterates $s_1, s_2, \ldots, s_n$ are all stored in an array `s` of dimension n. Test it with $n = 5$.

c) Now modify the script a second time by inserting the code

```
x = 1:n;
t = ((2*pi*x).^(1/2)).*((x./exp(1)).^x);
plot(x,t./s,'*')
```

after the `for ... end` loop and before the `end` of the function.

This piece of code should now produce a plot of t_k/s_k against k for $k = 1, \ldots, n$ where

$$t_k = \sqrt{2\pi k}(k/\mathrm{e})^k.$$

Run your code by calling `myfact1(20);`

Your plot should suggest that $t_k/s_k \to 1$ as $k \to \infty$.

Note: When you come to Publish your work to part (c), you will first need to choose Edit Publish Preferences (below Publish) and replace the command line `myfact1` with `myfact1(20);`

Now you can Publish and printout your code and the plot.

d) By comparing areas of curves under graphs, it is possible to establish the relation

$$\int_1^n \ln(x)\, dx < \sum_{j=1}^n \ln(j) < \int_0^n \ln(x+1)\, dx.$$

Use this to show that

$$\left(\frac{n}{\mathrm{e}}\right)^n < n! < \mathrm{e}\left(\frac{n+1}{\mathrm{e}}\right)^{n+1}$$

which is commensurate with the observation made in the numerical results.

5. Write a function called `mybicoeff` which takes the **two** arguments n, m and returns the value of

$$\left(\begin{array}{c} n \\ m \end{array}\right) = \frac{n!}{m!(n-m)!}.$$

using Matlab's in-built `factorial` function.

6. [HARDER] The binomial coefficients (Exercise 5) coincide with the elements of Pascal's triangle.

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

in which each number in the triangle is the sum of the two numbers above.

Hence, the mth the element of the nth row (counting from 0) is the binomial coefficient $\left(\begin{array}{c} n \\ m \end{array} \right)$.

Write a function which takes as its input n, m, and outputs the value of $\left(\begin{array}{c} n \\ m \end{array} \right)$, calculating this value using a recursion formula based on Pascal's triangle.

Week 4

4.1 Principles of Programming: Conditional Statements

All programming languages include the important capacity to make conditional statements. Essentially these allow different actions to be taken depending on different conditions being met.

This decision making process reflects how processes are naturally simulated in real life.

In Matlab, the conditional statements are controlled by the general process

```
if condition 1
    action 1
elseif condition 2
    action 2
...
else
    action to be performed if none of the previous conditions are met
end
```

4.1.1 Example: the modulus $|x|$

The modulus is defined by

$$|x| = \begin{cases} x, & \text{if } x \geq 0 \\ -x, & \text{if } x < 0 \end{cases}$$

In Matlab the function used to take the modulus of a number is `abs`. For e.g.

```
>> abs(-2)
ans =
     2
```

But, if we had wanted to write our own modulus function, we could write the script

```
%% My Modulus function
function s = mymod(x)
if x >= 0
```

```
    s = x;
else
    s = -x;
end
end
```

and save as mymod.m then run

```
>> mymod(-2)
ans =
     2
```

4.1.2 Relational operators

These are the **relational operators** used in conditional statements.

<=	less than or equal to
>=	greater than or equal to
>	greater than
<	less than
==	equal to
~=	not equal to

Note: The symbol = is used in Matlab to **assign** the value on the right hand side to the left hand side. The use of == as a relational operator in Matlab means *test* if the left hand side equals the right hand side.

Formally the use of relational operators give rise to two outcomes: **true** or **false** and these are stored by the computer as 1 or 0. So the statement `if 2 > 1` is the same as `if 1` and `if 1 < 2` is the same as `if 0`, meaning (respectively) if true/false.

4.1.3 Example: the signum function sgn(x)

Matlabs in-built function `sign(x)` performs the mathematical function sgn(x), defined by

$$\operatorname{sgn}(x) = \begin{cases} 1, & \text{if } x > 0 \\ -1, & \text{if } x < 0 \\ 0, & \text{if } x = 0 \end{cases}$$

In Matlab we can write the function script

```
%% My Signum function
function s = mysgn(x)
```

```
if x > 0
    s = 1;
elseif x < 0
    s = -1;
else
    s = 0;
end
end
```

4.2 Logical operators

Within conditional statements often you want to test more than one condition at the same time. Most frequently these take the form of **and** '`&&`' and **or** '`||`' operators.

E.g.: we use logical operators in normal language:

"If it is Friday **and** *it is the 13th then it is an unlucky day."*

"If it is Monday **or** *Tuesday* **or** *Wednesday* **and** *I don't have a 9 o'clock lecture then I can stay out late."*

Note: The ambiguity in the last statement, requiring the use of parentheses.

4.2.1 Example

Let a 'step' function be defined by $f(x) = \begin{cases} 1, & \text{if } 1 < x < 2 \\ 0, & \text{if } x \geq 2 \text{ or } x \leq 1 \end{cases}$

```
%% Function mystep
function s = mystep(x)
if x > 1 && x < 2
    s = 1;
elseif x <= 1 || x >=2      % Note: could just use else here as covers
                            % all remaining possibilities.
    s = 0;
end
end
```

Note: You can use logical operators within parentheses to test many conditions at once. So if we have

$$f(x) = \begin{cases} 1, & \text{if } 1 < x < 2 \text{ or } -3 < x < -2 \\ 0, & \text{otherwise} \end{cases}$$

we could write

```
%% Function mystep
function s = mystep(x)
if (x > 1 && x < 2) || (x > -3 && x < -2)
    s = 1;
else
    s = 0;
end
end
```

4.3 Example: Integer divisors

Problem: Given a positive integer n, list all the integer divisors of n.

Note: `ceil` and `floor` are Matlab functions which round up and down (respectively) to the nearest integers. E.g. `ceil(2.2)` gives 3 and `floor(2.8)` gives 2.

```
%% List the integer divisors of a number n
function intdiv(n)          % input n, no output required
for j = 2:n-1               % 1 and n are trivial, so don't include in loop
    if n/j == floor(n/j)    % test that j is an integer divisor
        j                   % if so, print it
    end
end
end
```

Q: Does `j` have to go all the way to $n-1$?

A: No, only need to get to $n/2$ (rounded down). So we can replace `j = 2:n-1` with `j = 2:floor(n/2)`.

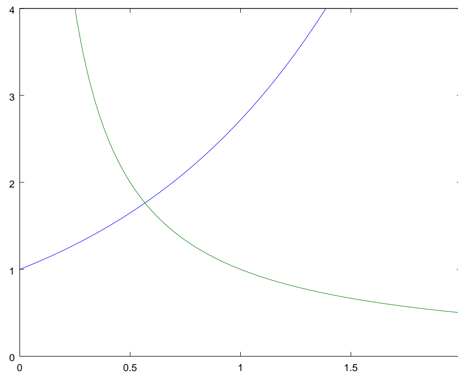
4.4 Application – root finding: the 'bisection method'

Some equations can be solved exactly: for example, the solutions of the quadratic equation $ax^2 + bx + c = 0$ are given explicitly. These are equivalent to the **roots** of the quadratic function $ax^2 + bx + c$.

But more general (nonlinear) equations are difficult to solve exactly. For example, solutions of cubic or higher order polynomial equations or equations involving transcendental functions.

E.g.

$$e^x = \frac{1}{x} \tag{4.1}$$

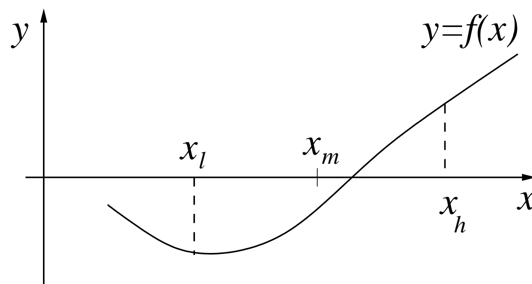


We can establish that there *is* a solution, by sketching graphs of e^x and $1/x$ on the same axes; when they intersect, at $x = x^*$, say, then x^* solves (4.1), or, we say $x = x^*$ is a root of the function

$$f(x) = e^x - \frac{1}{x}.$$

How can we find x^* ? We can't explicitly, so we instead try to approximate x^*.

We note that $f(x^*) = 0$ and, assuming $f(x)$ is continuous close enough to $x = x^*$ it must be that $f(x)$ changes sign from $x < x^*$ to $x > x^*$. This forms the basis of the ...



4.4.1 The bisection algorithm

For more complicated computational tasks, it helps to break the problem down into a sequence of small steps, and this then helps develop the code. This is often called an **algorithm**.

1. Assume you have values x_l and x_h such that $x_l < x^* < x_h$ and $f(x_l)$ has a different sign to $f(x_h)$.
2. The next step is to define the midpoint $x_m = \frac{1}{2}(x_l + x_h)$ of the interval (x_l, x_h) in which you know x^* exists. Now you test if $f(x)$ changes sign in the interval (x_m, x_h)

or in the interval (x_l, x_m). Whichever interval the change of sign is, is the new half-size interval containing the root x^*.

3. At this point you either have no need for x_l (if you have determined that $x^* \in (x_m, x_h)$) or x_h (if $x^* \in (x_l, x_m)$). So you re-label your new half-size interval so that the lower end of the interval is x_l and the upper end is x_h.
4. You are essentially back to step 1, but with tighter bounds on where x^* is. So you repeat using a `for` loop.
5. How many loops will we need ?

4.4.2 Analysis of accuracy of bisection method

Initially, the root lies in an interval $x_h - x_l$. So x^* is at most a distance $\frac{1}{2}(x_h - x_l)$ from x_m, the midpoint of the interval. I.e. the maximum error is $\frac{1}{2}(x_h - x_l)$.

After the first interation, the interval is halved in size, so the maximum error is $\frac{1}{4}(x_h - x_l)$; after k iterations the maximum error is

$$E = \frac{1}{2^{k+1}}(x_h - x_l).$$

If we want to find a root to within a **tolerance** or error of ϵ, say, then we are looking for the smallest k such that

$$\frac{1}{2^{k+1}}(x_h - x_l) < \epsilon$$

In other words

$$-(k+1)\log(2) < \log\left(\frac{\epsilon}{x_h - x_l}\right)$$

or

$$k > \log\left(\frac{x_h - x_l}{\epsilon}\right) \Big/ \log(2) - 1.$$

4.4.3 Matlab code

The bisection code with take the form of a Matlab function.

Our input is: x_l, x_h and ϵ (we'll call them `xl`, `xh`, `tol`) and, of course, we somehow need to define the function f – see in just a moment.

Our output is: approx to x^* and the number of iterations taken.

So we write a function in Matlab called, say, bisection.m:

```
%% Bisection method
function [xm,j] = bisection(f,xl,xh,tol) % Two outputs returned in an array
```

```
nmax = floor(log((xh-xl)/tol)/log(2)); % Max number of iterations needed
                                       % to reach the tolerance tol

for j=1:nmax                  % Start the loop. j is just a counter.
    xm = (xl + xh)/2;         % The midpoint of the interval (xl,xh)
    pl = f(xl)*f(xm);         % used to check if f chances sign in (xl,xm)

    if abs(f(xm)) < tol       % If f(xm) = 0 - within the tolerance tol - we
                              % have landed on the root and we stop the routine
        return                % Return means ‘finish the trip to the function’
    elseif pl < 0             % If f changes sign in (xl, xm) then
        xh = xm;              % pl < 0, and so replace xh by xm.
    else                      % otherwise f changes sign in (xm,xh) and
        xl = xm;              % replace xl by xm
    end
end
end
```

4.4.4 Using the routine

First we need to **define a function**. In example (4.1), we would write

```
>> f = @(x)(exp(x)-(1/x));
>> [root,n] = bisection(f,0.1,2,0.000001)
root =
    0.567143249511719
n =
    16
```

4.5 Newton's method for root finding

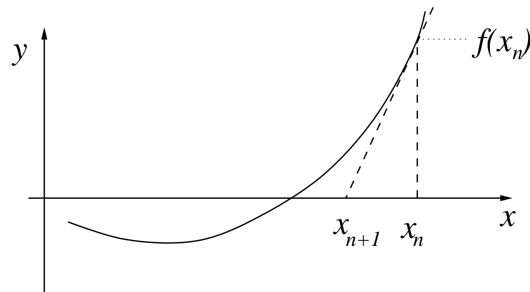
If you know not just $f(x)$ but also $f'(x)$ explicitly then an alternative (and often more rapidly convergent) method of finding roots is called Newton's method.

Newton's method is different to the bisection method in that it requires just one estimate of the root $x = x^*$ of $f(x) = 0$ rather than having to bracket the value of the root as before.

The method is best described using the following diagram:

We see that if $x = x_n$ $(n \geq 0)$ is a guess to $x = x^*$ then $x = x_{n+1}$ is closer to $x = x^*$ where, by geometry we see that

$$\frac{f(x_n)}{x_n - x_{n+1}} = \text{gradient of tangent at } x_n = f'(x_n).$$



Rearranging results in the iterative step

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{4.2}$$

To start the iteration we have to select an initial guess x_0, say.

4.5.1 Analysis

Questions: (i) Will the method always work ? (A: No); (ii) How fast will the method converge ? (A: Depends).

(i) Example of divergence

Consider $f(x) = \tanh(x)$. We know that the only root of $\tanh(x) = 0$ is $x = 0$.

Then $f'(x) = \text{sech}^2(x) \equiv 1/\cosh^2(x)$ and Newton's method reads:

$$x_{n+1} = x_n - \sinh x_n \cosh x_n = x_n - \tfrac{1}{2}\sinh 2x_n$$

Now we can see graphically that

$$|\sinh x| > |x|, \qquad \text{for all } x \neq 0.$$

But if $|x|$ is larger than some critical value, we can see that $|\sinh 2x| > 4|x|$.

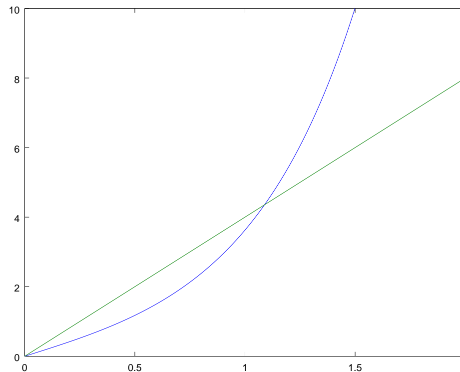
Imagine then that $x_0 > 0$ is an initial guess for which $\sinh 2x_0 > 4x_0$. Then the 1st step of Newton gives

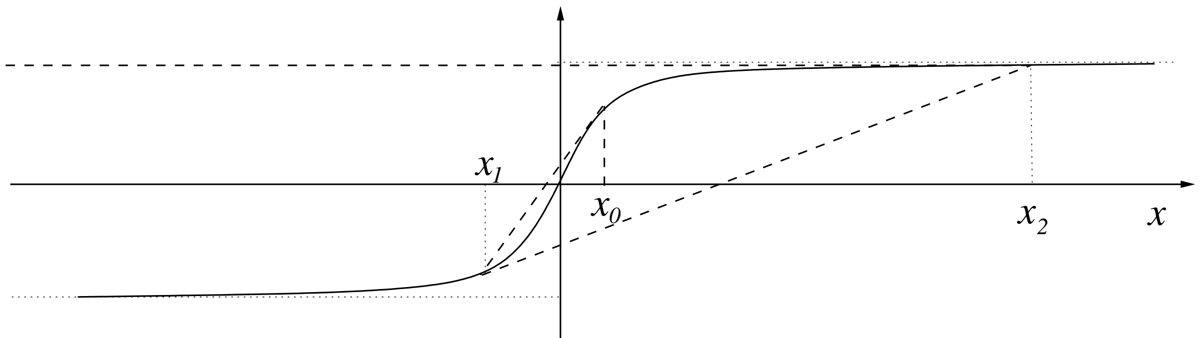
$$x_1 = x_0 - \tfrac{1}{2}\sinh 2x_0 < -x_0$$

Now $|x_1| > |x_0|$ and then, as before, next iterate is

$$x_2 = x_1 - \tfrac{1}{2}\sinh 2x_1 > -x_1$$

Thus, the iterates are growing and not tending to 0, as we want ! To see why look at the picture of Newton's method in action.





(ii) Examples of fast and slow convergence

If $f(x) = x^2 - 2$ we know the roots are $x^* = \pm\sqrt{2}$. Now $f'(x) = 2x$ and Newton's method gives

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right)$$

We have seen this before in Week 2 and shown convergence to the two roots (which one depends on choosing x_0 to be positive or negative) is rapid. That is, more rapid than bisection would give you.

Choose now $f(x) = x^m$, $m \geq 1$. We know the roots are $x^* = 0$. Now $f'(x) = mx^{m-1}$ and Newton's method gives

$$x_{n+1} = x_n - \frac{x_n^m}{mx_n^{m-1}} = x_n\left(1 - \frac{1}{m}\right).$$

This will converge for all $m \geq 1$:

- For $m = 1$ it converges in the first step. Why ?

- For $m = 2$, the iterative step says $x_{n+1} = \frac{1}{2}x_n$ and we halve the distance to the root $x^* = 0$ at each step. If m is large, we only improve by a small factor of $(1 - 1/m)$ at each step, and this is slower than bisection.

4.5.2 The Algorithm

As before, let's set out a series of basic steps needed to formulate a code to implement the Newton method.

1. Define the function, f and f' in the Matlab command window – see §4.4.4. Call them `f` and `fd`, say.
2. Set up a function with inputs, `f`, `fd`, `x0`, `tol`, `nmax`.

   We need `tol` as a tolerance to say that we are close enough that we want to stop.

   We need `nmax` to specify the maximum number of iterations and stop the code from potential going on forever without stopping.
3. What do we want our output to be ? The approximation to the root, and the number of iterations. Just like the bisection code.
4. Inside the code, we need to perform the iterative step (4.2). I.e. set up a loop for `j` from `1` to `nmax`.
5. Let's use `x` to represent x_n and `y` to represent x_{n+1}.
6. In each loop we want to test if we have found a root. Sensible to test if

   $$|x_{n+1} - x_n| < \texttt{tol}$$

   and if this is satisfied `return` from the loop.
7. If get to `nmax` and root not converged, output a warning to the user. Use `disp('message')` to do this.

4.6 Problems 4: Conditional statements, root finding

1. Follow §4.1.1 the notes of Week 4 and write a function script mysinc.m which computes the mathematical function

   $$f(x) = \begin{cases} \dfrac{\sin x}{x}, & x \neq 0, \\ 1, & x = 0. \end{cases}$$

   We know that $\lim_{x \to 0} \dfrac{\sin x}{x} = 1$. So what is wrong with defining $f(x) = \sin x / x$?

2. Write a function script which takes x as its input and computes the value of the 'square wave' function

$$f(x) = \begin{cases} 1, & 2n\pi \leq x < (2n+1)\pi, \\ 0, & (2n+1)\pi \leq x < (2n+2)\pi \end{cases}$$

for all integers n. [Hint: you may find the Matlab command `mod` useful. For example: `>> mod(7,3)` returns the value of 7(mod 3) = 1.]

3. Download the bisection method from the course website `https://people.maths.bris.ac.uk/~marp/compmaths` and save in your Matlab folder.

   a) Test the code with the function $f(x) = e^x - 1/x$ as described in §4.3.4 of the notes.

   b) Deduce that there is just one root to the equation $\sin(x) = x$. Test that the bisection method will find this root.

   c) Plot curves of the functions $\sin(x)$ and $1/x$ on the same graph for $0 < x < 12$. Use `» axis([0,12,-2,2])` to limit the vertical extent of the plot to $-2 < y < 2$. Printout your graph and annotate it with answers to the questions below:

   (i) Explain why there are an infinite number of roots $x = x_n^*$, say, to the equation $x\sin(x) = 1$ and write down an approximate formula for large x_n^*; (ii) use the bisection method to find (to 6 decimal places) the values of the first 3 positive roots.

4. a) Follow the algorithm in §4.4.2 of the notes on Newton's method to write a function `newton` which takes the five inputs: f, f', `tol`, `nmax`, `x0` and returns two outputs: the approximate root of $f(x) = 0$ and the number of iterations taken.

   b) Test your code on the functions: (i) $f(x) = e^x - 1/x$ and (ii) $f(x) = \sin x - x$. In both cases choose $x_0 = 1$, `nmax = 100` and `tol = 1e-6`. Why does the rate of convergence change ?

   c) Test your code on the function $f(x) = \tanh(x)$. Find values of x_0 for which Newton's method is convergent and others where it is not convergent.

   d) Use your code to find the positive root of the equation $\sinh(2x) = 4x$ accurate to 6 decimal places. How does this value relate to part (c) ?

   Publish your output to part (d) (remembering to use Edit Publish Options to include three lines: the definition of `f`, `fd` and the command used to call `newton`.)

5. In this exercise, we make a modification to Newton's method, called the 'secant method', which is useful if the derivative, f', is not easy to derive.

   See `http://en.wikipedia.org/wiki/Secant_method` for a description of this method.

The recursion formula for the secant method is

$$x_{n+2} = \frac{x_n f(x_{n+1}) - x_{n+1} f(x_n)}{f(x_{n+1}) - f(x_n)}$$

where two initial guesses x_0 and x_1 are needed. Clearly this is a two-term recurrence relation.

a) Write a function, say, `secant`, which takes f, x_0, x_1, `nmax` and `tol` as input and outputs the value of the root and the number of iterations taken.

b) Test your code on the function $f(x) = x^2 - 2$, where you know the roots are $x^* = \pm\sqrt{2}$.

c) Can you find an example where the iterates diverge instead of converging on the root ?

d) Show that the secant method applied to $f(x) = x^2 - 2$ explicitly reduces to the recurrence relation

$$x_{n+2} = \frac{x_n x_{n+1} + 2}{x_n + x_{n+1}}.$$

Do you expect convergence to $\sqrt{2}$ to be faster than the iteration

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right)$$

discussed in Week 2 of the course ?

6. (HARD) Write a Matlab function which inputs a general array of numbers and outputs the array ordered from smallest to largest number. You will find the Matlab function `length(a)` useful in returning the size of an array `a`.

5 Week 5

5.1 Computers have limits

The theme of the lecture and examples this week's lecture is to explore how far you can push a computational task, when to expect that things will go wrong and how to mitigate against things going wrong.

We have already seen that computers allow us to make approximations (to e.g. π, e, $\sqrt{2}$, infinite series, roots of functions) and we pick up this idea of approximation here.

In particular, we are concerned here with the possibility that we are asking computers to do things they are not equipped to do.

5.1.1 Floating point precision

One of the key issues is how computers store numbers. For example, in Matlab is you ask for the value of π you get:

```
>> pi
   3.141592653589793
```

But π is irrational and its decimal representation continues forever. It is obvious that the computer cannot store the infinite number of decimal places required for π.

So the computer stores essentially the number of decimal places you see when you ask for `format long` (roughly 16 decimal places). This is called the **floating point precision**.

So even π in Matlab is approximate, and the fact that you cannot store numbers *exactly* can cause problems, as we will demonstrate.

5.1.2 Overflow and Underflow

The other issue we will see is concerned again with infinity, an abstract concept which a computer cannot cope with. The question to be asked is:

What is the biggest number the computer can understand ?

Again, it is to do with storage and it turns out that in Matlab the computer can store numbers up to about 10^{308}. Which is quite big, but sometimes not big enough !

Conversely the computer cannot register very small numbers and can only go as far as about 10^{-308} before it decides this is practically zero. Again, there are occasions when this might not be small enough.

Problems associated with very large and very small numbers are referred to as **overflow** and **underflow**.

5.1.3 A simple demonstration: loss of precision

On the Matlab command line type

```
>> s = 2;
>> s = sqrt(s)
```

and repeat the last line another 46 times. The output is

```
s =
    1.000000000000005
```

Now we reverse the process and square 47 times by repeating the command

```
>> s = s^2
```

We should end up with `s = 2` but we actually get

```
s =
    1.988737457549722
```

Why is this ? Well the storage of the 47th square root of 2 has lost valuable information about the decimal places after the 17th position needed to reconstruct 2 when subsequently squared 20 times.

In the language of computational mathematics we say that such an outcome is symptomatic of **rounding errors**.

Perhaps the example above seems is bit silly and artificial and in some ways so is the next, but it makes the point a different way.

5.1.4 Another example of loss of precision

Consider the function

$$(x-1)^7 = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \tag{5.1}$$

We can see that this function passes through zero at $x = 1$.

We can plot this function around $x = 1$ in Matlab with the following

```
>> x = [1-2.0e-8:1.0e-10: 1+2.0e-8];
>> y = (x-1).^7;
>> plot(x,y)
```

We have plotted over the small range of values $-2^{-8} < (x-1) < 2^{-8}$, and we see that the curve is what we expect it to be:

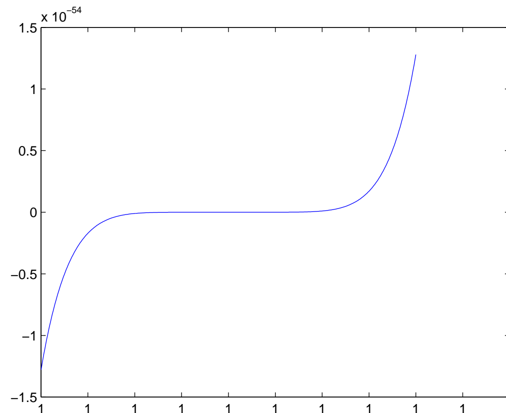


Figure 5.1: The graph of $(x-1)^7$ close to $x = 1$.

Now we use the RHS of (5.1) instead,

```
>> x = [1-2.0e-8:1.0e-10: 1+2.0e-8];
>> z = x.^7-7*x.^6+21*x.^5-35*x.^4+35*x.^3-21*x.^2+7*x-1;
>> plot(x,z)
```

and this gives the following graph: This is just **numerical noise**.

Why has this happened ? The two functions are mathematically equivalent and yet the computer calculates completely different values. The answer lies in **rounding errors** again.

The LHS is calculated by taking a number x close to 1. Let's say

$$x = 1 + \epsilon, \qquad \Rightarrow (x-1)^7 = (1 + \epsilon - 1)^7 = \epsilon^7$$

and the computer loses no storage information in computing a small number to a high power. For example,

```
>>  0.00000001^7
ans =
    1.000000000000000e-56
```

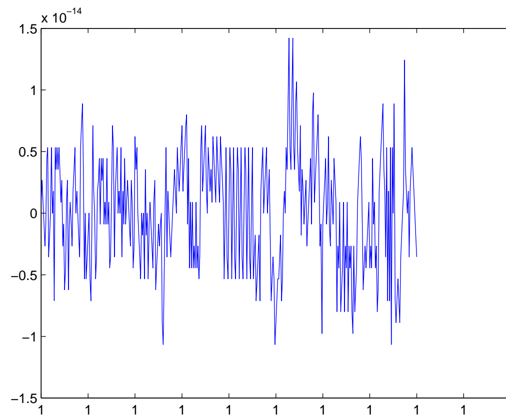


Figure 5.2: The graph of $x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$ close to $x = 1$.

On the RHS, for a number x close to 1, say $x = 1 + \epsilon$ the calculation the computer performed by the computer is

$$(1+\epsilon)^7 - 7(1+\epsilon)^6 + 21(1+\epsilon)x^5 - 35(1+\epsilon)x^4 + 35(1+\epsilon)x^3 - 21(1+\epsilon)x^2 + 7(1+\epsilon) - 1$$

which exactly equals ϵ^7, a number of the size of 10^{-57}, say. But the computer sees the calculation as *approximately*

$$1 - 7 + 21 - 35 + 35 - 21 + 7 - 1$$

(being equal to zero) and for each number in the series above, the computer cannot store the required digits for the sum total of each of the 7 terms to come to exactly 10^{-56}.

5.1.5 Yet another example

This example illustrates the same point as above. Consider the simultaneous equations for unknowns x and y:

$$\left.\begin{aligned} x + y &= 1 \\ x + (1+\epsilon)y &= 1 \end{aligned}\right\} \tag{5.2}$$

where $\epsilon > 0$. Clearly the solution is $x = 1$, $y = 0$, independent of ϵ.

Now imagine we wanted to solve this algorithmically, we would write the two equations in matrix/vector form:

$$\begin{pmatrix} 1 & 1 \\ 1 & 1+\epsilon \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad \Rightarrow \qquad \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} (1+\epsilon)/\epsilon & -1/\epsilon \\ -1/\epsilon & 1/\epsilon \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

after taking inverses. So

$$x = (1+\epsilon)/\epsilon - 1/\epsilon, \qquad \text{and} \qquad y = 1/\epsilon - 1/\epsilon$$

We can make the next step in the calculation using standard algebraic manipulation and it allows us to deduce that $x = 1$ and $y = 0$. But the computer cannot do algebraic manipulations and so as part of a numerical algorithm the computer would compute the two terms for x and y and subtract one from the other.

Mimicking this step in Matlab we have (when $\epsilon = 10^{-14}$)

```
>> e = 1e-14;
>> x = (1+e)/e - (1/e)
x =
     1
```

which is OK and (when $\epsilon = 10^{-15}$)

```
>> e = 1e-15;
>> x = (1+e)/e - (1/e)
x =
     1.1250000000000000
```

which is clearly nonsense. Why ? Because $1 + 10^{-15}$ cannot be accurately stored in the computer because of its floating point storage limitations. It gets worse, for $\epsilon = 10^{-16}$,

```
>> e = 1e-16;
>>  x = (1+e)/e - (1/e)
x =
     0
```

because, as far as the computer is concerned $1 + 10^{-16}$ is the same as 1 in the way it is stored.

5.2 Overflow and underflow

5.2.1 An example

Here's a bit of mathematics which you should be able to do:

$$\int_0^L \tanh(x)\,dx = [\ln(\cosh(x))]_0^L = \ln(\cosh(L)) - \ln(1) = \ln(\cosh(L)) \tag{5.3}$$

Fine. The LHS is the area under the curve of tanh and since $\tanh(x) \to 1$ rapidly as x increases, we can see that the integral is approximately equal to 'L minus a bit'.

In Matlab, there is a built in capacity to calculate integrals numerically, that is without knowing how to analytically work out the answer as we have done in (5.3).

So we can write

```
>> f = @(x)(tanh(x));
>> integral(f,0,100)
ans =
    99.306852819426894
>> integral(f,0,800)
ans =
    7.993068517616830e+02
```

Which are correct. We can check by comparing with our exact answer in (5.3)

```
>> log(cosh(100))
ans =
    99.306852819440053
>> log(cosh(800))
ans =
    Inf
```

Inf ? So here, Matlab has told us that a number in the calculation has got so big (bigger than 10^{308}), that it cannot store the number and let's us know that this has happened by returning the message `Inf`.

But we know the answer was just shy of 800. So what was the problem ? The problem is that $\cosh(800)$ is greater in size than 10^{308} and the computer calculates this *before* taking logarithms.

Of course, we can avoid this computationally, by being **clever**. In this case you can identify the issue in advance and work out a way to overcome it, by writing

$$\ln(\cosh(L)) = \ln(\tfrac{1}{2}(\exp^{L} + \exp^{-L})) = \ln(\exp^{L}) + \ln(\tfrac{1}{2}(1 + \exp^{-2L})) = L + \ln(\tfrac{1}{2}(1 + \exp^{-2L}))$$

Note: We see from that that for L very large

$$\int_0^L \tanh(x)\, dx \approx L - \ln(2).$$

as the exponential $\exp^{-2L}$ is vanishingly small and this is commensurate with the calculations made previously.

In Matlab, we confirm the revised version works:

```
>> 800+log(0.5*(1+exp(-2*800)))
ans =
    7.993068528194401e+02
```

Note: In Matlab, $\exp^{-1600}$ is less than 10^{-308} and so beyond the computers storage capacity. In Matlab, numbers less than 10^{-308} are stored as 0 (which can cause problems) allowing the calculation to be made cleanly.

5.3 The exponential function

Let's go back to the example in Week 3 of the scripting of the function myexp1.m for computing the exponential function using the McLaurin series representation

$$\exp^x = \sum_{j=0}^{\infty} \frac{x^j}{j!} \tag{5.4}$$

The Matlab code we wrote

```
%% A function, called myexp1
function s = myexp1(x,n)      % Two inputs: x and n
s = 1;                        % The rest is the same
for j = 1:n
    s = s + x^j/factorial(j);
end
end
```

We tested this with a command like » `myexp1(1,20)` which outputs `2.718281828459046` and agrees exactly with » `exp(1)`.

So all is good. Or is it ? Now let's try to use our code to compute $\exp^{100}$.

```
>> exp(100)
ans =
     2.688117141816136e+43
>> myexp1(100,100)
myexp1 =
     1.415460872100881e+43
>> myexp1(100,150)
myexp1 =

     2.688113827113892e+43
>> myexp1(100,200)
myexp1 =
   NaN
```

Note: NaN is an error message shorthand for **Not a Number**. It means we have asked the computer to do something it cannot do.

The problem:

$$\exp^{100} = \sum_{j=0}^{\infty} \frac{100^j}{j!}$$

and 100^j grows rapidly as j increases and j needs to be very large before $j! > 10^j$ and the series starts to converge. Which is why we need to increase our truncation number from 100 to 150 to 200.

However, when j is very large, we are dividing large numbers by large numbers. In particular we note that in Matlab that

```
>> 100^200
ans =
    Inf
>> factorial(200)
ans =
    Inf
```

So our current algorithm for computing the exponential will not work because the computer needs to calculate `Inf/Inf` (which is where you get `NaN`) before the series has converged.

The solution: We need to avoid dividing large numbers by large numbers. We start with

$$\exp^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

which we can write as

$$\exp^x = 1 + 1 \cdot \frac{x}{1} + 1 \cdot \frac{x}{1} \cdot \frac{x}{2} + 1 \cdot \frac{x}{1} \cdot \frac{x}{2} \cdot \frac{x}{3} + \dots$$

Algorithmically, this series can be computed using

$$t_0 = 1, \qquad t_n = t_{n-1} x / n, \qquad \text{and} \qquad s_0 = 1, \qquad s_n = s_{n-1} + t_n$$

where s_n is the nth partial sum.

In Matlab code

```
% A function, called myexp2
function s = myexp2(x,n)      % Two inputs: x and n
s = 1;                        % This is the running total of the sum
t = 1;                        % This is the new clever bit.
for j = 1:n
    t = t*x/j;
    s = s + t;
end
end
```

and we see that we avoid dividing large numbers by large numbers.

Back to our troublesome example:

```
>> myexp2(100,200)
myexp2 =
     2.688117141816134e+43
```

which now works.

5.4 Problems

1. a) In the Matlab command line, evaluate $\tanh(800)$ using the built-in function `tanh`. Compare your result against the evaluation of $\sinh(800)/\cosh(800)$. What has gone wrong ?

   b) Use the definition of sinh and cosh in terms of exponentials to find an alternative way of computing $\tanh(800)$ and test it in Matlab.

2. Consider the following infinite series

$$S = \sum_{j=1}^{\infty} \frac{\sinh(j)}{j^2 \cosh(j + 1/j)}$$

   a) Prove that this series is convergent.

   b) Write a Matlab function script series1.m to compute the nth partial sum s_n, say (that is, the series truncated at $j = n$).

   [Hint: Adapt the code sum2.m from Week 3 used for the computation of truncated infinite series, into a function with n the input and s_n the output.]

   c) Run your code with the tuncation parameter $n = 100$, $n = 200$, $n = 400$. Do your results appear to be converging ?

   d) Now try $n = 800$. You should find that there is a problem with the output.

   Why is this ? [Hint: see exercise 1.]

   e) Copy your code to a new function script, series2.m say, and rewrite the computation of the series to overcome the problem you have identified. Test the code with $n = 400$, $n = 800$ and $n = 1600$ to demonstrate that the new function script runs successfully.

   [*You should* Publish *the output of part (e) with* $n = 1600$, *remembering to use the* Edit Publish Options *to execute the command* `series2(1600)`. *If you don't get part (e) working, printout your code from (c) with* $n = 400$ *in the same way. Annotate your printout with the answer to part (a).*]

   f) Your results from part (e) should still show that the partial sums are converging slowly.

We know from the Week 3 notes that

$$\sum_{j=1}^{\infty} \frac{1}{j^2} = \frac{\pi^2}{6}.$$

Using this identity, suggest a new way of computing the series S which improves the rate of convergence and estimate that improved rate of convergence.

Test your idea numerically by writing a piece of code to implement the improved convergence scheme.

3. The method of Archimedes for approximating π by using inscribed and circumscribed polygons (e.g. Week 2), but starting with hexagons leads to the recurrence relation

$$s_{n+1} = \frac{\sqrt{s_n^2 + 1} - 1}{s_n}, \qquad n = 0, 1, \ldots$$

with $s_0 = 1/\sqrt{3}$. The value of π is approximated at each step by

$$\pi \approx 6 \times 2^n \times s_n$$

and the expectation is that the approximation improves as n increases.

a) Write a script to implement the recurrence relation above up to the 20th iterate and, at each step of the relation, output the approximation to π.

b) Run your script, and identify that the code is not doing what it should be doing.

c) Find an alternative method for performing the iterative step above that avoids the issues that you have identified in part (b) and test it to confirm that the computations do what you expect them to do.

4. You are presented with the following calculation:

$$\int_{-1}^{1} \frac{1}{x}\, dx = [\ln |x|]_{-1}^{1} = 0$$

Try to confirm this in Matlab by defining a Matlab function $1/x$ and using the inbuilt Matlab numerical integration command `integral`:

```
>> f = @(x)(1./x);
>> integral(f,-1,1)
```

What happens ? What is wrong here ?

5. The McLaurin series for $\cos(x)$ is

$$\cos(x) = \sum_{j=0}^{\infty} \frac{(-1)^j x^{2j}}{(2j)!}.$$

a) Follow the example in Week 5 notes on the exponential function to write a Matlab function script which avoids numerical difficulties associated with computing convergent series for large values of x.

The input should be x and the number of terms, n taken in the series and the output should be the nth partial sum approximation to $\cos(x)$.

b) Test your function. Why is there still an issue with your function for large x ?

How could you overcome it ?

6. The Catalan numbers are defined by

$$C_n = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix} \equiv \frac{(2n)!}{(n+1)!n!}.$$

a) Confirm that $C_0 = C_1 = 1$ and show that

$$C_n = \prod_{k=2}^{n} \frac{n+k}{k}.$$

for $n \geq 2$.

b) Which of the two representations is more computationally robust and why ?

c) Write a function in Matlab which inputs n and returns the value of C_n using the method in part (b).

[*You should* Publish *the output of part (c) with* $n = 10$ *(remember* Edit Publish Options*) and annotate with your answers to parts (a), (b).*]

7. [HARD] So-called spherical Bessel functions $j_n(x)$ are, arguably, most easily defined by the following two-term recurrence relation

$$j_{n+1}(x) = (2n+1)j_n(x)/x - j_{n-1}(x), \qquad n = 1, 2, \ldots$$

with

$$j_0(x) = \frac{\sin(x)}{x}, \qquad j_1(x) = \frac{\sin(x) - x\cos(x)}{x^2}.$$

Write a Matlab function (with n and x inputs) to compute and print the first n of these. Compare your code with the exact results of $j_0(0.3), \ldots, j_{10}(0.3)$:

```
0.985067355537799
0.099102888040642
0.005961524868620
0.000255859769695
0.000008536424265
0.000000232958256
0.000000005378443
0.000000000107607
0.000000000001899
0.000000000000030
0.000000000000000
```

[Hint: You should observe that things go wrong very quickly.]

6 Week 6

This week we continue reinforce the programming ideas we have accumulated in the previous few weeks (loops, conditional statements, arrays, functions, plotting) and how to think about turning problems into code. This lecture will illustrate this with examples based on ...

6.1 Simulations

Simulations are important in many areas of statistical science. For example, they are relevant to climate change where one builds uncertainty into elements of a model which is advanced into the future. One simulation produces one particular outcome. But in the same way that tossing a coin once and noting it lands on 'heads' doesn't mean all coins always land on heads, one needs to run the experiment many times to produce statistically meaningful results.

6.1.1 A Matlab function for coin tossing

Let's start with a simple example of coin tossing where the input is n, the number of simulations and the output is `heads` and `tails`, the probabilities predicted by the simulation of heads and tails being tossed.

```
function [heads,tails] = cointoss(m)  % input m, the number of tosses
                                      % output: P(heads), P(tails)
heads = 0;                % heads and tails are counters
tails = 0;
for k=1:m
    if rand > 0.5         % rand produces a random number between 0 and 1
        heads = heads+1;  % add to the heads counter or the tails counter
    else
        tails = tails+1;
    end
end
heads = heads/m;     % normalise the count to give simulated probabilities
tails = tails/m;
end
```

When you run this code, you find that as n increases, the two probabilities tend towards 0.5. As you'd expect.

You can plot the results in a bar graph by running

```
>> [h,t] = cointoss(1000);
>> bar([h,t])
```

where the command `bar` can be used in several ways (see help). Here it plots a bar graph of the array `[h,t]` against the axis.

6.2 The Galton Board

The **Galton board** is a device for statistical experiments named after English scientist Sir Francis Galton. A number of balls are dropped in the top of an inverted funnel and falls under gravity bouncing off pegs as it falls. The idea is that the bead has a 50/50 chance of being diverted to the left/right as it collides with each peg on its descent.

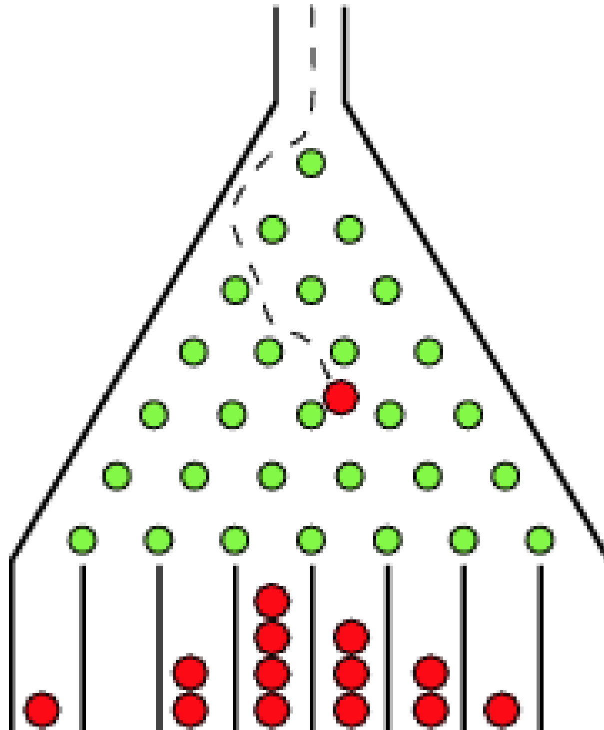


Figure 6.1: A diagram of a Galton Board, with green pegs and red balls. See `http://www.youtube.com/watch?v=6YDHBFVIvIs` for a video demonstration.

Where is the ball likely to fall ?

6.2.1 Mathematics of the Galton board

If there are n rows of pegs there will be $n+1$ possible final locations for the balls; let's label them $0, 1, 2, \ldots, n$.

At each peg, the probability of going left or right is equal and $\frac{1}{2}$. So, a simple application of combinatorial arithmetic shows, for example, there is only path to location 0, there are n paths to location 1, and so on, and we quickly see that there are in general

$$\left(\begin{array}{c} n \\ j \end{array} \right) = \frac{n!}{j!(n-j)!}$$

different paths to location j. At each peg, the probability of going left and right is $\frac{1}{2}$ and there are n pegs so the probability of ending up at location j is

$$\frac{1}{2^n} \left(\begin{array}{c} n \\ j \end{array} \right) = \frac{n!}{2^n j!(n-j)!}.$$

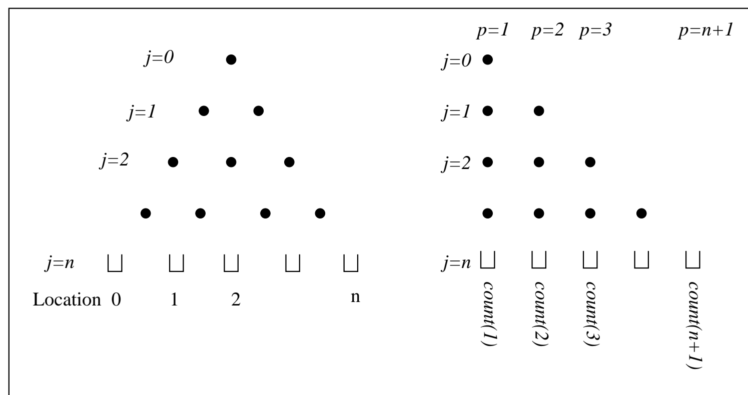


Figure 6.2: On the left panel, the physical configuration, the labelling of rows and the final locations on the lower row. On the right panel, how we label the pegs for the code: the rows are numbered the same, but columns are labelled from $p = 1$ to $p = n + 1$. Now at each step either stay in the same colum or move a step to the right. The array count acts as a 'bin' to pick up dropped balls.

6.2.2 Matlab code

Now we want to test our result statistically. I.e. we drop a large number of balls through the system to even out the randomness.

The input is n, the number of rows of pegs and m, the number of simulations we make.

The output will be a plot of the simulation against the probabilities. Figure **??**(b) shows how we organise the galton board for the purposes of coding the simulation. The ball starts at position $p = 1$ and then at every step from $j = 1$ to $j = n$ the ball either stays with the same value of p or moves to the right to $p + 1$. When $j = n$, the array `count` then stores the final position by adding one to the value of `count(p)`. This is embedded in a loop over m simulations.

```
function galton(n,m)    % Input: n = # peg rows, m = # simulations
                        % Output is a plot

count = zeros(1,n+1);   % set up counters for each location

for k=1:m               % Do m simluations
    p = 1;              % p tracks the location of the ball; starts at 1
    for j=1:n           % drop down n pegs
        if rand > 0.5   % at each peg go right (p=p+1) or left (p=p)
            p = p+1;
        end
    end
    count(p) = count(p)+1; % ball at bottom, location p, add to count(p)
end

% Compute the expected values according to the formula

count2 = zeros(1,n+1);
for j=0:n
    count2(j+1) = factorial(n)/(2^n*factorial(j)*factorial(n-j));
end

% Produce a plot of simulation against probabilities

x = 1:n+1;                          % array x(1) = 1, x(2) = 2 etc.
plot(x,count/m,'*',x,count2,'-')    % Note: divide simulations by total
                                    % number of balls.
```

Note: The `plot` command overlays two plots on the same graph, one plotting discrete data points ($*$) and one a line joining data point ($-$).

Here's a demonstration of the output from running the code:

```
>> galton(10,1000)
```

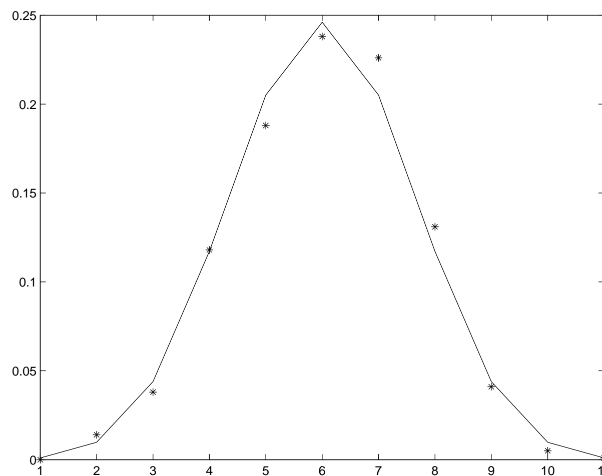


Figure 6.3: An output of the code `galton` with $n = 10$, $m = 1000$, simulation are $*$ and lines are expected values.

6.2.3 An improved code with nicer graphical output

Look at the code on the website to see a different way of presenting the results of the simulation as bar graphs.

6.3 Random walks in 1D

The example above is very close to a one-dimensional 'Simple Random Walk on $\mathbb{Z}$': at each time-step (say, each second), you decide with equal probability to go forwards or backwards one step. The random walk has long been used to describe a drunk man leaving a pub.

Here's a good example of what this might look like
`http://www.youtube.com/watch?v=7a7l7IHXZwk`

6.3.1 Theory and Brownian motion

At each time step, j, suppose you step $s_j = +1$ or -1 to the right/left with equal probability. The total distance covered after n steps is

$$x_n = \sum_{j=1}^{n} s_j$$

The mean position (averaged over all realisations) is

$$\langle x_n \rangle = \left\langle \sum_{j=1}^{n} s_j \right\rangle = 0.$$

I.e. on average, you will find yourself at the origin. This doesn't tell you a lot. Measure instead the mean square displacement,

$$\langle x_n^2 \rangle = \left\langle \sum_{j=1}^{n} s_j \sum_{k=1}^{n} s_k \right\rangle = \left\langle \sum_{j=1}^{n} s_j^2 + \sum_{k=1}^{n} \sum_{j=1,\neq k}^{n} s_j s_k \right\rangle = n$$

In order words the mean distance from the origin averaged over all realisation is

$$\sqrt{\langle x_n^2 \rangle} = \sqrt{n}$$

and if you quadruple n, the total number of time steps, on average you are twice as far from the origin. This is a process which is characteristic of **Brownian Motion**.

6.3.2 The limit $n \to \infty$ and Normal distributions

With a shift of variables from the Galton Board game, $j = \frac{1}{2}(n+m)$ we have the probability of being a displacement m from the origin after n steps as

$$P_n(m) = \frac{n!}{2^n(\frac{1}{2}(n-m))!(\frac{1}{2}(n+m))!}$$

We note that if $m = 0$ and we write $n = 2N$ for algebraic convenience we see that $P_{2N}(0) = (2^{-2N})(2N)!/(N!N!)$ which are related to the Catalan numbers (Worksheet 5).

If we use *Stirling's formula* (Worksheet 3),

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^n, \qquad \text{as } n \to \infty$$

it can be shown (the algebra is too fierce to repeat here) that

$$P_n(m) \sim \frac{2}{\sqrt{2\pi n}} \mathrm{e}^{-m^2/2n}, \qquad \text{as } n \to \infty$$

which is a Normal probability distribution with a variance $\sigma^2 = n$ characterised by a Gaussian function.

Remark: The *Central limit theorem* proves that any random process with a finite mean and a non-zero variance tends to a Gaussian in the limit as $n \to \infty$, so the details of how the random walk is made (e.g. uniform steps, in 1D) are unimportant to the main results.

6.4 Simulation: A two-dimensional random walk

Let's now do a random walk on $\mathbb{Z}^2$. That is, we take a prescribed number of steps of length 1 and at each step we go with equal probability north, south, east or west.

We want to simulate this many times and then statistically analyse the results.

Motivated by the previous section on 1D random walks, we compute the mean squared distance from the origin (our random walk starting position) after a given number of steps. We call this the D^2 average (`d2av` in the code).

6.4.1 Code

The input is the number of steps taken and the number of simulations performed.

The output is going to be a plot of the simulated mean squared distance taken after the jth step against j.

In the code, for each simulation, we start at the origin (`x=0` and `y = 0`), define a random number (`p`) between 0 and 4 and use this to move with equal probability in each of the four directions, north, south, east and west. After each move, we update the D^2 average for that step (the jth step).

```
function rwalk2d(n,m)   % input: n = #steps and m = #simulations

d2av = zeros(n,1);      % Set up an array of length 1+nsteps

for k = 1:m             % k counts the simulations
    x = 0;              % Each new simulation starts at the origin
    y = 0;
    for j = 1:n         % doing n steps
        p = rand*4;     % p = random variable from 0 to 4
        if ( p >= 3 && p < 4 )
            x = x+1;   % right
        elseif ( p >= 2 && p < 3 )
            y = y+1;   % up
        elseif ( p >= 1 && p < 2 )
            x = x-1;   % left
        else
            y = y-1;   % down
        end
        d2av(j) = d2av(j) + x^2 + y^2;   % update average D^2
    end
end
```

```
d2av(:) = d2av(:)/m;       % Normalise by dividing array by m
%d2av(:) = d2av(:).^0.5;  % uncomment to get the mean distance

time = 1:n;           % Set up an array for x-axis for plotting.
plot (time,d2av,'-','LineWidth',2)

end
```

6.4.2 Results

And we are ready to do some simulations

```
>> rwalk2d(100,500)
```

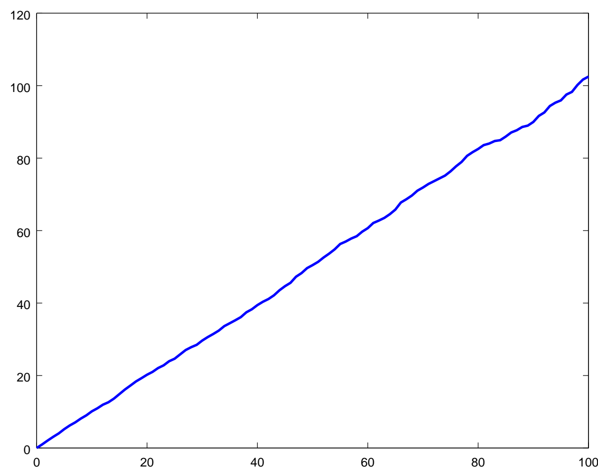


Figure 6.4: An output of the code `galton` with `n=100` and `m=1000`. The mean squared distance is on the vertical against number of steps on the horizontal. You can see it's almost a perfect linear fit, in accordance with the theory.

6.4.3 More advanced plotting

In Matlab, we've seen that we can plot multiple lines on the same graph by defining two (or more) pairs of arrays which are called sequentially by the `plot` command. E.g.

```
>> x = [0:0.01:1];
>> y1 = sin(x);
>> y2 = x.^2;
>> plot(x,y1,x,y2)
```

Issuing the commands » `plot(x,y1)` following by » `plot(x,y2)` sequentially only overwrites the first plot with the second.

Another method, which can be very useful is to use the commands `hold on` and `hold off` which allows new curves to be overlaid onto the old curves. Associated with these are `clf` which clears the graphics screen of all existing curves.

Example: Look at the code animrwalk2d.m on the course website for a version of the 2D random walk which modifies the previous code to animate each individual walk.

6.5 Problem

1. a) Test what » `ceil(rand*6)` returns from the command line.

   b) Write a function called `dice` which averages over m simulations the scoring of the sum of two randomly-thrown six-sided dice.

   The outline of the function is given below. You need to fill in the part which simulates the throwing of the dice and updating the counter array `count` which is designed so that `count(j)` records the number of times the dice sum j is thrown.

```
function dice(m)                       % input: m = # of simulations

count = zeros(12,1);                   % array for logging dice scores

% Fill in missing code from here ...   %
%                                      %
% ... to here.                         %

count(:) = count(:)/m;                 % normalise by # of simulations

x = 1:12;                              % x axis for plotting
count2 = [0,1,2,3,4,5,6,5,4,3,2,1]/36; % array of expected results
plot(x,count,'*',x,count2,'-')         % plot

end
```

   c) After testing your code, Publish your code with a value of $m = 500$. Remember as always to Edit Publish Options to set the call to `dice(500)`. Your printout

should contain a listing of the code and a plot of the simulated and expected results.

d) Adapt the code so that the output is plotted as a bar graph (see the online code `galton.m` and how it compares with the code given in the notes).

2. In a one-dimensional random walk, at each of the n steps taken, the position is moved (from an initial position at the origin) randomly one step to the left or right with equal probability.

   a) Download the code `rwalk2d.m` from the course website and copy it into a new file, `rwalk1d.m` (remembering to change the name of the `function` also).

   Adapt the code to simulate the one-dimensional random walk described above. The inputs to the function should be n, the number of steps, and m, the number of simulations.

   Your code should compute the mean squared distance – the D^2 average – (see §6.4 of notes). The output should be a plot of the number of steps taken on the horizontal axis against the normalised D^2 average on the vertical axis.

   Hint: This task mainly involves just deleting everything that operates in the y-direction to leave dynamics in the x-direction.

   b) Test the code with the function with different values of n and m. You should observe that the D^2 average approximately is linear with the steps taken.

   c) Publish your code with $n = 100$ and $m = 500$. Remember, as always, to Edit Publish Options to change the call line to `rwalk1d(100,500)`. Your printout should include just the code and a plot of your results.

3. Download the code `animrwalk2d.m` and save as `animrwalk1d.m`. Implement the changes made in Exercise 3 so that your new code plots an animation of a random walk in 1D, with the step number along the horizontal axis and the vertical axis used to show the distance from the origin.

4. In §6.2.1 of the notes the probability of ending up at location j, where $j = 0, 1, 2, \ldots, n$, on the Galton board is given as

$$\frac{1}{2^n}\binom{n}{j} = \frac{n!}{2^n j!(n-j)!}$$

   (a) Since the total probability of all outcomes must be 1, it must be that

$$\sum_{j=1}^{n}\binom{n}{j} = 2^n$$

   Can you prove this ?

   (b) Develop a modified theory where the probability of a ball being deflected left at each peg is $p \neq \frac{1}{2}$.

Download and adapt the code `galton.m` from the course website so that its inputs are m, n and p and the output is a bar graph of the simulated results versus expected results for biased pegs. Test with $p = \frac{1}{4}$, $n = 10$ and $m = 500$.

5. (a) Download the code `rwalk2d.m` from the course website and use it as a template for a new simulation of random walks in which the position at each step is of unit length but in a random direction.

   (b) Do the same thing to the code `animrwalk2d.m`. This will be the best thing you will have done so far.

6. [PRIZE PROBLEM 1] Write code to simulate a player determined to get a Yahtzee (`https://en.wikipedia.org/wiki/Yahtzee`) from three rolls of five dice. In particular, your code should numerically determine the probability of getting a Yahtzee if that's what you set out to get. The person/team with the best answer and code submitted to me by email before 5pm Friday 4th Novemeber wins a small prize.

7 Week 8

7.1 Introduction to 3D plotting

A three-dimensional surface can often (not always, e.g. a sphere) be represented in Cartesian coordinates by writing the height, z of the surface, as a function of the two horizontal coordinates x and y. I.e., we write

$$z = f(x, y)$$

So how can we use Matlab to visualise such surfaces ? Here's how. First you have to define the range of values that x and y take and, as with line plotting, we do with by setting up arrays which define the discrete points at which the surface plot is evaluated. For e.g. if we want to plot the function $z = \mathrm{e}^{-x^2-y^2}$ over $0 \le x \le 1$, $0 \le y \le 1$ we use

```
» x = 0:0.01:1;
» y = 0:0.01:1;
» [X,Y] = meshgrid(x,y);
» Z = exp(-X.^2-Y.^2);
```

The 3rd line defines the new variables `X` and `Y` as a **double array** contain all possible combinations of `x` and `y` and consequently `Z` is also a double array which evaluates the height of the surface above the array of points (`X`,`Y`).

To plot the surface, one can type

```
» surf(Z)
```

which simply plots the matrix `Z` (i.e. the x and y coordinates are the indexes of the matrix). In this example, better to plot z against x and y using

```
» surf(X,Y,Z), shading interp
» view([1,1,1])
```

The shading option removes grid lines on the plot and the `view` command changes the position from which you view the surface. For example, `» view([0,0,1])` shows the surface plot from above. A nicer feature of Matlab is the ability to interactively rotate the view of the plot with the mouse and this can be toggled using

```
» rotate3d on
» rotate3d off
```

If you need to restrist the viewing area of the plot, use the extended version of axis to add to min/max values of z to those of x and y, e.g.

```
» axis([0 1 0 1 0 2])
```

Finally, to plot contour lines either on the surface itself or projected onto the horizontal plane use, for example

```
» contour3(X,Y,Z,20)
» contour(X,Y,Z,20)
```

where 20 can be varied to give the number of contour levels plotted.

There are many more plotting command/options that can be used, but these are probably the most basic and useful. Of course, these commands can be incorporated into Matlab scripts and functions.

7.2 Simulations in time and space

In week 6 we looked at random walks in which the evolution of time was modelled as a series of discrete steps in which the position of the random walker moved a set distance at each discrete step.

In many areas of mathematics one is interested in problems which depend upon **both space and time**. These types of problems can often be treated computationally in a similar manner to the random walk approach by describing the continuous evolution of time as discrete steps in time whilst simultaneously 'chopping up' continuous space into a series of discrete **cells**.

Then the state of the system at a point in space and time is described not in terms of the continuous variables x and t, say, but by integers, j and k (say), which encode the cell number and the number of steps in time taken. As we consider this computationally we need to restrict the values of j and k and so we let $1 \leq j \leq n$ and $1 \leq k \leq m$ so there are n cells and m time steps.

Under this discretisation, we can assign a double array (or a $m \times n$ matrix) $u(k, j)$ to store information about the state of the system at time step k in cell j.

One can now devise rules which describe how the state of the system advances forward in time from one time step to the next using information already stored about the state. These rules can often be derived from the continuous system one is trying to model or they can be invented either arbitrarily or so as to follow empirical evidence/observations. We shall see examples of both types here.

To keep things relatively simple, the update to time step $k + 1$ in cell j will be of the

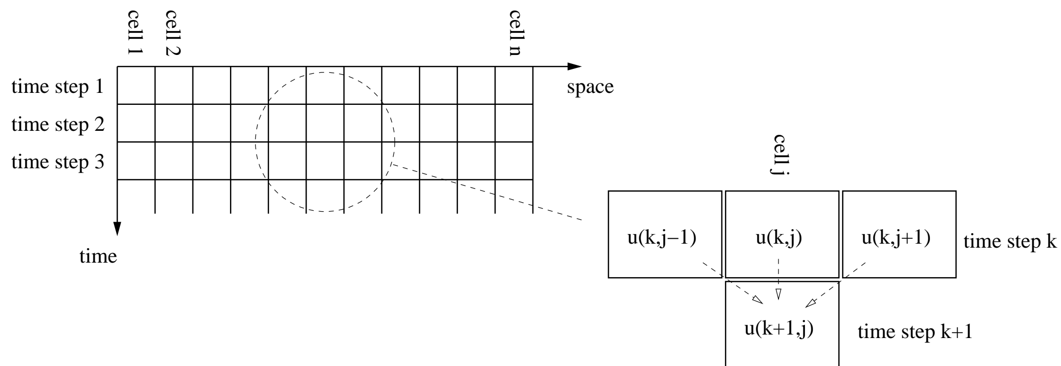


Figure 7.1: Diagram illustrating discretisation of space and time into rows of cells and how the systems updates to the new time step

taken to be of the form:

$$u(k+1,j) = f(u(k,j-1), u(k,j), u(k,j+1))$$

where f represents an (assumed given) function of the state of the system at time k in cells $j-1$, j, $j+1$. In other words, we advance forward each step in time using information about the current state and the state of the two neighbouring cells.

Initially, which is to say at time step $k = 1$, the state of the system is assumed to be given. I.e.

$$u(1,j) = a_j$$

for some numbers a_j, $1 \leq j \leq n$.

Note: For cells $j = 1$ and $j = n$ we cannot use the update rule as, in each case, we are missing a neighbour cell. So we have to treat these 'edge' cells differently to the interior cells. This may be informed by the model we are trying to approximate computationally.

7.2.1 1D Cellular Automata

Cellular Automata are discrete space/time systems in which the state of the system at a particular time step and in a particular cell is described as being either "on" or "off". Mathematically, then $u(k,j) = 1$ or 0. The rules for updating are based on the the state of the current cell and its two neighbours as a **binary operation**.

We give an example of such an operation below in which the three numbers in the row titled "present state" refer to the left/centre/right cell state.

Remark: One can see that this binary system belongs to a class of 256 possible combinations. Stephen Wolfram first considered these models and came up with the classification system: the one above is called "Rule-90" or the "**Sierpinski triangle rule**".

present states	000	001	010	011	100	101	110	111
update to	0	1	0	1	1	0	1	0

Table 7.1: Rules for updating the cells based on the states of the cell and its left/right neighbours.

Note: In terms of coding this example, it looks complicated at first glance. But we note that the update amounts to a single condition being met:

"If there is just one neighbour alive at the previous step, then make the current cell alive"

Algorithm and Matlab code

Below is a Matlab script which implements the Sierpinski triangle cellular automata system.

We have to specify the number of cells in the simulation (n) and the number of time steps taken in the simulation (m). We store information about the state of the jth cell at the kth time step in a **matrix** (a two-dimensional array), here denoted by $u(k,j)$. The entries of this matrix will be either 0 or 1 and at the end of the script the surface of the matrix is plotted so that red/blue on the grid corresponds to 1/0 or alive/dead.

In the first step $k = 1$ we assign the just the middle cell $u(1, n/2)$ to be alive.

Here's a Matlab script to do the job:

```
%% Sierpinski triangle Cellular Automata model
m = 80;           % number of time steps in simulation
n = 80;           % number of cells

u = zeros(m,n);   % matrix of time steps against cells

u(1,n/2) = 1;     % set the middle one of the first row to zero

for k=1:m-1              % loop over time steps 1 to m-1
    for j=2:n-1          % loop over all *interior* cells

        b = u(k,j-1)+u(k,j+1)   % b is the sum of the two neighbours

        if b == 1        % implement update rule
            u(k+1,j) = 1;
        end
```

```
        u(k+1,1) = 0;   % update edge cells to zero
        u(k+1,n) = 0;

    end
end

surf(u)             % surface plot the matrix of cell values
view([0,0,1])       % view plot from above
```

Output:

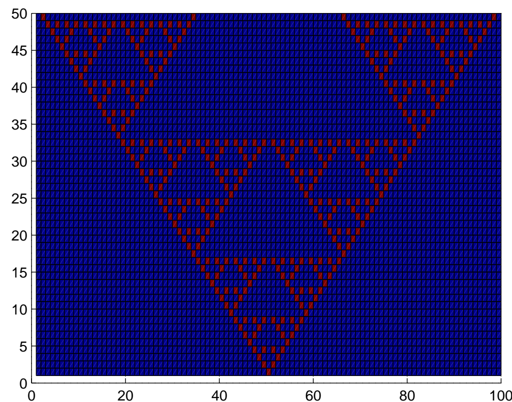


Figure 7.2: Output of the Sierpinski triangle code. Time axis is up, space axis horizontal.

7.2.2 Example: Diffusion

Diffusion is the process of spreading of the state of a system. For example, if you drop a blob of dye in a narrow shallow channel of water of the dye spreads out over time along the channel due to Brownian motion, otherwise known as molecular diffusion.

The continuous model which describes diffusion is too complicated to state here (it's a partial differential equation you would study in the 2nd year APDE2 unit). But a discrete time/cell based model that arises from a particular approximation of the continuous model is given by the update rule:

$$u(k+1,j) = u(k,j) + D(u(k,j+1) - 2u(k,j) + u(k,j-1))$$

where D is a parameter which controls how fast the diffusion happens (at least numerically).

The Matlab script is similar to before:

```
%% Modelling diffusion
m = 80;             % set number of time steps
```

```
n = 80;          % set number of cells

D = 0.4;         % set diffusion parameter D

u = zeros(m,n);  % set up double array for u all full of zeros

u(1,n/2+2) = 1;
u(1,n/2+1) = 1;
u(1,n/2) = 1;    % set 5 cells around the centre cell to 1; all others 0
u(1,n/2-1) = 1;
u(1,n/2-2) = 1;

for k=1:m-1      % step forwards in time
    for j=2:n-1  % update interior cells according to rule

        u(k+1,j) = u(k,j)+D*(u(k,j-1)-2*u(k,j)+u(k,j+1));

    end

    u(k+1,1) = 0; % update cells at the edges to be zero
    u(k+1,n) = 0;

end

surf(u),shading interp    % plot the surface of the matrix u
view([1,1,1])             % view surface from a jaunty angle
```

Output:

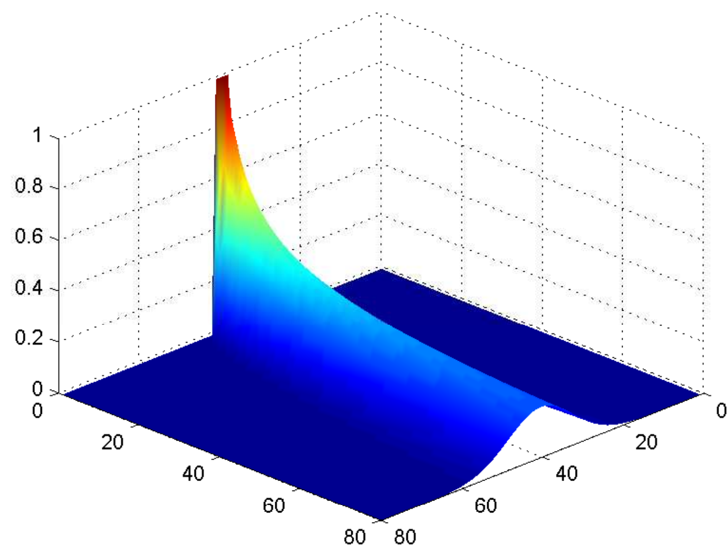


Figure 7.3: Output of the diffusion script `mydiffusion.m`. Notice that as time increases the initially sharp peak spreads out.

8 Week 8

Amongst the most useful mathematical tasks a computer can perform is the ability to provide solutions to problems in calculus that are accessible using analytic approaches.

Specifically this week we shall consider: (i) how to use computational methods to compute definite integrals and (ii) to solve ordinary differential equations (ODEs).

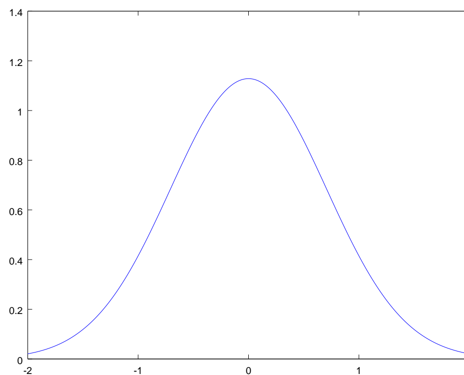
8.1 Numerical integration

(Often referred to known as **quadrature**.) Much of the time (e.g. in Calculus 1) you will be presented with integrals which can be done analytically using standard methods such as substitution, integration by parts etc. But life is not always so easy and there are many occasions where integrals which cannot be integrated explicitly. For e.g.

$$I = \frac{2}{\sqrt{\pi}} \int_0^x \mathrm{e}^{-t^2}\, dt,$$

cannot be expressed in terms of 'elementary functions'.

It so happens that the integral above is so useful it is given a special name, the **error function** written $\mathrm{erf}(x)$.

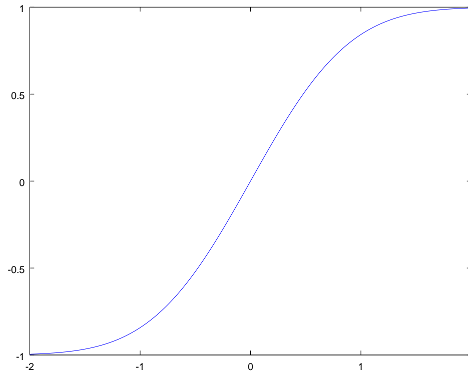


Note: We can always interpret a definite integral as the area under a curve. Thus, the error function is defined as the area under the graph of the function $f(t) = (2/\sqrt{\pi})\mathrm{e}^{-t^2}$ across $0 < t < x$.

Remark: We can deduce some basic properties of the error function from its definition:

- $\mathrm{erf}(0) = 0$;

- $\text{erf}(x)$ is monotonically increasing.
- $\text{erf}(-x) = -\text{erf}(x)$ (i.e. it is an **odd function**);
- $\text{erf}(\infty) = 1$ since $\displaystyle\int_0^\infty e^{-t^2}\,dt = \frac{\sqrt{\pi}}{2}$ is a known result (c.f. Probability 1).
- Can use information above to sketch a rough graph of $\text{erf}(x)$.



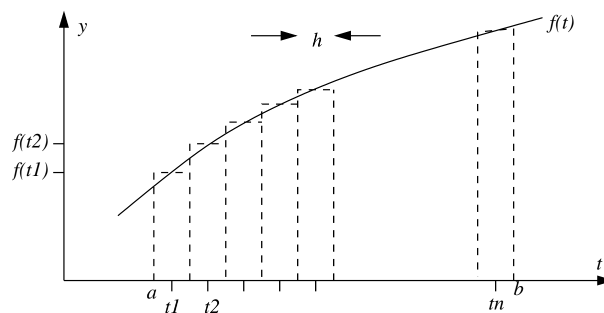
Just because the function is not given in terms of elementary functions doesn't mean you cannot find out things about it... however we still have a need to calculate I given x (or find $\text{erf}(x)$)... so...

8.2 The rectangle mid-point rule

In general we wish to find the (approximate) value of

$$I = \int_a^b f(t)\,dt$$

The simplest way of approximating I is to dividing the area under the curve of $f(t)$ into a number n, say, rectangles of equal width as shown in the figure below:



It follows that the width of each rectangle is $h = (b - a)/n$.

We let t_j, $j = 1, 2, \ldots, n$ denote the midpoint of the base of each rectangle, so that

$$t_j = a + (j - \tfrac{1}{2})h, \qquad j = 1, 2, \ldots, n$$

The height of the curve at $t = t_j$ is $f(t_j)$ and we set this to the height of the approximating rectangle.

Now, summing over all rectangles we have

$$I \approx I_{rect} = \sum_{j=1}^{n} h f(t_j) \equiv h \sum_{j=1}^{n} f(a + (j - \tfrac{1}{2})h).$$

This approximation is known as the **rectangle mid-point rule**.

Intuitively we expect that as n increases (so h decreases), the approximation to I should improve. I.e. the method suggests that $I_{rect} \to I$ as $n \to \infty$.

8.2.1 Matlab code

The input needed is the function f, the end points a and b, and n – the parameter which controls the approximation.

The output is the value of I_{rect} the approximation to I.

```
function s = rectint(f,a,b,n)   % function, input: f = function, a,b
                                % end points, n = number of rectangles.
                                % output is s = approx to integral.

h = (b-a)/n;                    % the width of each rectangle

s = 0;                          % running total of area of rectangles
for j=1:n                       % loop over all rectangles
    s = s+h*f(a+(j-0.5)*h);     % add the areas of each rectangle
end

end
```

8.2.2 Analysis: Estimating the error

This is not particularly rigorous, but it can be done rigorously (analysis, numerical analysis). We refer to fig. 8.2 and note that the **error**, E, is

$$E \equiv I - I_{rect} = \sum_{j=1}^{n} \int_{t_j - h/2}^{t_j + h/2} f(t) - f(t_j)\, dt.$$

(that is, the error is the sum of all the left over bits between the actual curve and the rectangular approximation.)

Now, since h is supposed to be small, for $t_j - \frac{1}{2}h < t < t_j + \frac{1}{2}h$, $f(t)$ can be well approximated by its **Taylor series** representation (c.f. Calculus 1) centred around t_j

$$f(t) = f(t_j) + (t - t_j)f'(t_j) + \frac{(t - t_j)^2}{2!}f''(t_j) + \dots$$

where the additional terms get increasingly small (higher order terms or H.O.T.). Using this in formula for E gives

$$E \approx \sum_{j=1}^{n} \int_{t_j - h/2}^{t_j + h/2} (t - t_j)f'(t_j) + \frac{(t - t_j)^2}{2!}f''(t_j)\, dt$$

but the first term integrates to zero (check for yourself) and so

$$E \approx \sum_{j=1}^{n} \left[\frac{(t - t_j)^3 f''(t_j)}{6} \right]_{t_j - h/2}^{t_j + h/2} = \frac{h^3}{24} \sum_{j=1}^{n} f''(t_j).$$

Now we argue that the sum 'scales with n' meaning if n is doubled the value of the sum will be approximately doubled too (this can all be done properly). So the sum is like a constant (independent of n) times n. Since $n = (b - a)/h$ the upshot is that the error is given by

$$E \approx Ch^2$$

where C is some constant independent of h.

What this suggests is that doubling n, the number of intervals between a and b – and therefore halving h – implies that the error between the exact and the approximate result reduces by a factor of 4. Such schemes are called **second-order accurate**.

8.2.3 Running the code: results

First, in the command line window which calls the function, we need to define the function. For e.g.

```
>> f = @(t)((2/sqrt(pi))*exp(-t^2));
```

sets the function $f(t) = (2/\sqrt{\pi})\mathrm{e}^{-t^2}$. We can then call the routine with, for e.g.

```
>> rectint(f,0,1,100)
     0.84270
```

n	I	E
20	0.7071522185	4.5437e-05
40	0.7071181401	1.1358e-05
80	0.7071096209	0.2839e-05
160	0.7071074911	0.0709e-05

Table 8.1: Convergence of I with doubling n. Notice the error is approximately quartered as n is doubled.

This is an approximation to erf(1).

We can now test the mathematical estimate of the error in the method, by double the number N of intervals in the method and testing the error. Let's use function where we can do the integral exactly. E.g.

$$\int_{\pi/4}^{\pi/2} \sin(t)\, dt = \frac{1}{\sqrt{2}} = 0.707106781186547$$

Then call our code with, for e.g.

```
>> f = @(t)(sin(t))
>> rectint(f,pi/4,pi/2,20)
       0.707152218544050
```

8.3 The trapezium rule

The rectangle rule is pretty good, but that does not stop us from looking for more accurate methods of approximation and there are many much more sophisticated ways of extending the results.

The simplest extension is the following and fairly (I hope) obvious.

Instead of approximating the area under a curve by a series of narrow rectangles with flat roofs, we approximate the curve itself by piecewise linear line segments under which are generated trapezoids whose areas we can calculate the areas and sum over:

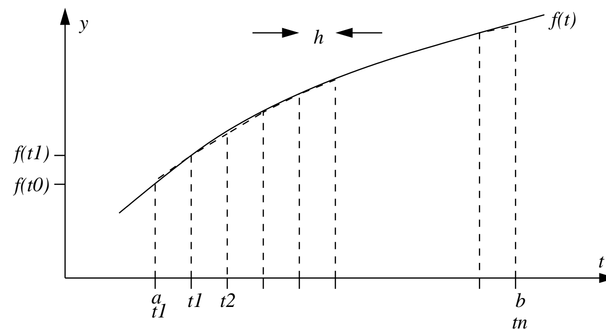
As before we divide the interval $a < t < b$ into n strips with $h = (b - a)/n$.

We modify the definition of t_j from before by writing

$$t_j = a + jh, \qquad \text{for } j = 0, 1, 2, \ldots, n$$

so that $t_0 = a$ and $t_n = b$. Now the area under one of the trapezoids between t_{j-1} and t_j is

$$\frac{h}{2}\left(f(t_{j-1}) + f(t_j)\right)$$



and so the total area under the sum of all trapezoids is

$$
\begin{aligned}
I \approx I_{trap} = \sum_{j=1}^{n} \frac{h}{2}\left(f(t_{j-1}) + f(t_j)\right) &= \sum_{j=0}^{n-1} \frac{h}{2} f(t_j) + \sum_{j=1}^{n} \frac{h}{2} f(t_j) \\
&= \frac{h}{2}\left(f(t_0) + f(t_n)\right) + h \sum_{j=1}^{n-1} f(t_j) \\
&= \frac{h}{2}\left(f(a) + f(b)\right) + h \sum_{j=1}^{n-1} f(a + jh)
\end{aligned}
$$

This is called **the trapezium rule.**

Worksheet Exercise: Implement this method in Matlab and use the numerical results to analyse the error of the method. Do you reckon it should be better than rectangle mid-point rule ?

8.4 Numerical integration of ordinary differential equations (ODEs)

At School and in Calculus 1 we are shown methods for solving ODEs. But these methods only work for ODEs belonging to certain classes where methods exist for integrating the ODE.

For example, we can solve

$$y'(t) + y(t) = \sin(t), \qquad t > 0$$

exactly using **integrating factors** (because the ODE is **linear**), but this method does not apply to

$$y'(t) + \sin(y(t)) = \sin(t), \qquad t > 0,$$

(which is **non-linear**). Indeed no general method of solution exists for this ODE. Yet a solution exists (see graph in Week 1).

How do we proceed ? Let us consider here only **1st order ODEs** (the numerical methods can be extended to higher order ODEs).

Consider a **general form** for a 1st order ODE:

$$\frac{dy}{dt} \equiv y'(t) = f(t, y(t)), \qquad t > t_0 \tag{8.1}$$

where f is a given function of the independent variable t and the dependent variable y (t_0 is assumed given).

As you should know, an additional condition is needed to fix the 'integration constant'. For the ODE in (8.1) this is typically expressed as

$$y(t_0) = y_0 \tag{8.2}$$

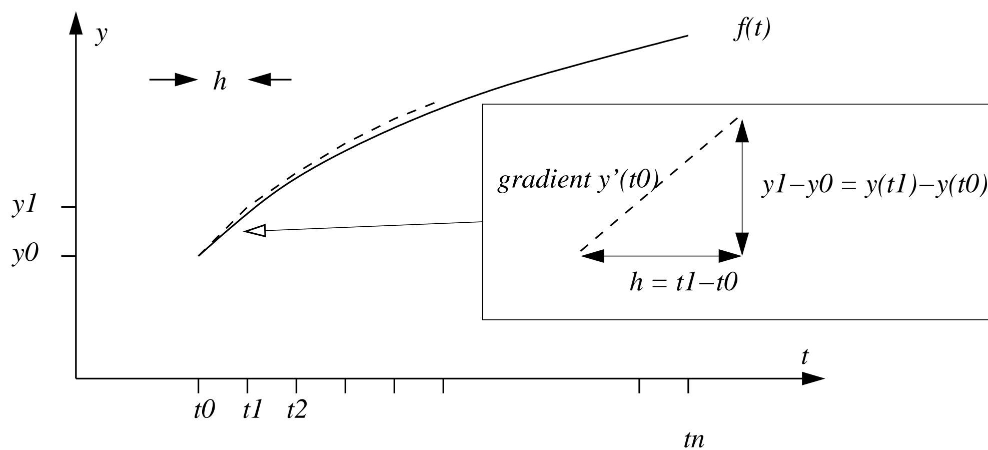
where y_0 is a given number and is often called the **initial condition** as often t is interpreted as the time variable.

8.4.1 Euler's method

We see immediately that since we are given t_0, $y(t_0) = y_0$ and f that we know the intial gradient of the solution, from (8.1),

$$y'(t_0) = f(t_0, y_0).$$

The idea is that we shoot forwards along a small **straight line** from t_0 to a new point $t_1 = t_0 + h$ where h is small where we can approximate $y(t_1)$ and then define $y'(t_1)$. And then we can repeat this step, thus reconstructing the solution through a series of short line segments.



That is, in general we define

$$t_j = t_0 + jh, \qquad j = 1, 2, \ldots, n$$

say, and we note that

$$y(t_{j+1}) - y(t_j) \approx hy'(t_j) \tag{8.3}$$

which can be established graphically...

... or which comes from a Taylor series expansion

$$y(t_{j+1}) = y(t_j) + hy'(t_j) + \frac{h^2}{2}y''(t_j) + \dots$$

in which h is assumed to be small enough that h^2, h^3 etc are negligible. (That's a big assumption !).

Using (8.2) with (8.3) we end up with the iterative scheme

$$y(t_{j+1}) = y(t_j) + hf(t_j, y(t_j)), \qquad j = 1, 2, \dots, n$$

with $t_j = t_0 + jh$ and $y(t_0) = y_0$.

So actually it's just a straightforward one-term recurrence relation.

8.4.2 Matlab code

The input is f, t_0, y_0, h and n, the number of steps we take in the iterative scheme. We note that the final value of t which we integrate the solution to is $t_n = t_0 + nh$.

The output will be a plot of $y(t_j)$ against t_j.

```
function myeuler(f,t0,y0,h,n)   % input f = function, t0 = initial t,
                                % y0 = y(t_0), h = step size, n =
                                % number of steps. No output returned

t = zeros(n+1,1);               % set up arrays for values of t_j
y = zeros(n+1,1);               % and y(t_j), and j needs to go from 0 to n

t(1) = t0;                      % initialise the arrays with t0
y(1) = y0;                      % and y0, supplied as input to the function

for j=1:n                       % iterate; can be done other ways
    t(j+1) = t(j)+h;
    y(j+1) = y(j)+h*f(t(j),y(j));
end

plot(t,y,'-')                   % plot y against t with a line.
end
```

Note: We go up to `n+1` because the first element of the arrays `t` and `y` are reserved for t_0 and y_0

8.4.3 Results

Let's take an example we can solve for exactly. E.g.

$$\frac{dy}{dt} = -\frac{t}{y}, \qquad t > 0$$

with $y(0) = 1$. The exact solution is

$$y^2 + t^2 = 1, \qquad \text{or} \qquad y = \sqrt{1 - t^2}$$

and we note that the solution only exists up to $t = 1$.

In Matlab we note $f(t, y) = -t/y$ to align the ODE with the general form expressed in (8.1) set up the function with

```
>> f = @(t,y)(-t/y);
```

and run the code with

```
>> myeuler(f,0,1,0.01,100)
Warning ... complex issues
```

Note: We see that there is an error recorded and this is because the approximation has violated the conditions/restrictions of the original problem.

The output looks like this:

8.1 Problems

1. a) Download the code rectint.m from the course web page and test it on the two examples in §8.2.3 of the notes:

$$\text{(i) } \frac{2}{\sqrt{\pi}} \int_0^1 e^{-t^2}\, dt, \qquad \text{and (ii) } \int_{\pi/4}^{\pi/2} \sin t\, dt$$

   using values of $n = 10, 20, 40, 80$. You should observe that the error in (ii) between the numerical results and the exact result decreases like $1/n^2$.

   b) Test `rectint` on the function $f(t) = t$ from $0 < t < 1$ using $n = 10, 20, 40, 80$. Why is the error always zero ?

2. a) Write a Matlab function called `trapint` which computes the approximation to the integral

$$I = \int_a^b f(t)\, dt$$

   using the **trapezium rule**.

   You should follow the method outlined in §8.3 of the notes and may find it useful to adapt the code rectint.m from the course web page.

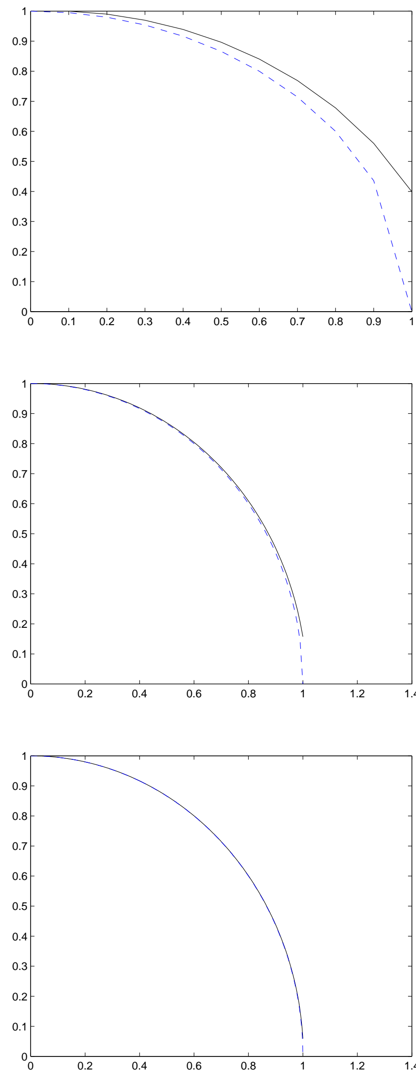


Figure 8.1: Euler method (solid) against exact solution (dashed) for step sizes of 0.1, 0.01 and 0.001.

b) Test your code on the two examples in Exercise 1(a). Deduce how the error decreases with increasing n. In Matlab, erf(1) is given by `» erf(1)`

[Publish *your code applied to the example (ii) from Exercise 1(a) with* $n = 80$ *using* `» format long`*. By hand, add results for* $n = 10, 20, 40$ *and your answer to (b). Remember to use* Edit Publish Options *to add the definition of* f *and the inputs to your function.*]

3. Simpson's rule (`http://en.wikipedia.org/wiki/Simpson's_rule`) is an extension of the rectangle mid-point rule and the trapezium rule. The basis of the method is to fit quadratic polynomials between regular points on the curve and integrate these explicitly to approximate the area under the curve.

   The integral I is approximated by the following formula:

$$I = \int_a^b f(t)\,dt \approx I_{simp} = \frac{h}{3}\left[f(a) + f(b) + 4\sum_{j=1}^{n/2} f(t_{2j-1}) + 2\sum_{j=1}^{n/2-1} f(t_{2j})\right]$$

   where $t_j = a + jh$ for $j = 0, 1, \ldots, n$ and $h = (b-a)/n$. **Note:** n must be even.

   a) Write Matlab code to approximate integrals using Simpson's rule.

   b) Test your code with the two examples given in Exercise 1(a).

      You should be able to deduce from your results from example (ii) that the error decreases like $1/n^4$.

4. In the lectures we have considered how to numerically integrate functions over a finite interval from $a < t < b$. But what if we wanted to evaluate

$$\int_0^\infty f(t)\,dt$$

   numerically ? One method is to transform the infinite integral to a finite integral with a transformation $t = u/(1-u)$.

   Make this substitution, and then apply it to the function $f(t) = 1/(1+t^2)$. Use one of your numerical integration routines from Exercises 1,2 or 3 to approximate the value of the resulting integral numerically and compare with the exact result.

5. a) Download the code myeuler.m described in §8.4 of the notes for approximating solutions to the 1st order ODE

$$y'(t) = f(t, y(t)), \qquad t > t_0, \qquad \text{with } y(t_0) = y_0,$$

      using Euler's method.

      Test the code on the ODE

$$y'(t) + \sin(y) = \sin(t), \qquad t > 0, \qquad y(0) = 1.$$

      plotting the solution to $t = 10$. Use step sizes $h = 0.08, 0.04, 0.02, 0.01$ to assess the accuracy of your results. You will find the commands » `hold on`, » `hold off` and » `clf` useful for overlaying graphical output (and then clearing it) from each set of results.

   b) Euler's method can be improved upon. One such scheme, an example of a general class of so-called 'Runga-Kutta methods' is called the **Midpoint method**

(http://en.wikipedia.org/wiki/Midpoint_method) and replaces the iterative step in §8.4 of the notes with

$$y_{j+1} = y_j + hf(t_j + \tfrac{1}{2}h, y_j + \tfrac{1}{2}hf(t_j, y_j)).$$

Adapt the code myeuler.m to implement the iterate step above and test your code with the ODE in part (a).

[Publish *your code with a step size of* $h = 0.01$. *Your output should be a plot of the solution between* $0 < t < 10$. *You will need to* Edit Publish Options *to set the function* $f(t, y)$ *and the function input parameters*.]

6. [HARD] Think of an efficient way of approximating the numerical value of the improper integral

$$\int_0^1 \ln(\sin x)\, dx.$$

9 Week 9

9.1 Linear Algebra

9.1.1 Background

Many complex mathematical problems are often ultimately reduced to solving problems in linear algebra. Specifically, a problem might be characterised by a large number of unknowns which satisfy an equally large number of equations. Often in areas of applied mathematics, one is able to model the problem in terms unknown functions which satisfy complicated differential equations (for example) and additional conditions or constraints which are impossible to solve explicitly. In such cases, one typically chooses to approximate the unknown functions by characterising them in terms of a discrete set of unknown numbers (this can be done in a number of different ways) and the subsequent transformation of the underlying differential equations which allows them to operate in this new discrete description of the problem gives rise to a system of equations. In many important application areas these are linear systems of equations.

9.1.2 Linear systems of equations

We are used to problems like this (simultaneous equations):

$$a_{11}x + a_{12}y = b_1$$
$$a_{21}x + a_{22}y = b_2$$

E.g. multiply the top equation by a_{21}/a_{11} and subtract from the bottom equation and you get

$$y = \frac{a_{11}b_2 - a_{21}b_1}{a_{11}a_{22} - a_{11}a_{12}}$$

and similarly,

$$x = \frac{a_{22}b_1 - a_{12}b_2}{a_{11}a_{22} - a_{11}a_{12}}$$

So what about a system of 3 equations in 3 unknowns ? Possible, but trickier... What about n equations in n unknowns (call them $x_1, x_2, \ldots x_n$) ?

$$\begin{array}{ccccccc}
a_{11}x_1 & + & a_{12}x_2 & +\ldots & +a_{1n}x_n & = & b_1 \\
a_{21}x_1 & + & a_{21}x_2 & +\ldots & +a_{2n}x_n & = & b_2 \\
\vdots & & \vdots & \ddots & \vdots & & \vdots \\
a_{n1}x_1 & + & a_{n1}x_2 & +\ldots & +a_{nn}x_n & = & b_n
\end{array}$$

So we can write this compactly as

$$\mathsf{A}\mathbf{x} = \mathbf{b} \tag{9.1}$$

where A is an $n \times n$ **matrix** and $\mathbf{x}$ and $\mathbf{b}$ are $n \times 1$ **column vectors**:

$$\mathsf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

In the first example, we show that the solution to the 2×2 system of equations can be written

$$\begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{a_{11}a_{22} - a_{12}a_{21}} \begin{pmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

and we have inverted the equation $\mathsf{A}\mathbf{x} = \mathbf{b}$ to deduce $\mathbf{x}$ explicitly. How has this happened ?

Well, the matrix on the right-hand side multiplying the vector $\mathbf{b} = (b_1, b_2)^T$ (here T means **transpose**) has the property that

$$\frac{1}{a_{11}a_{22} - a_{12}a_{21}} \begin{pmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

and the matrix on the right hand side is called the (2×2) **identity matrix**, I. The identity matrix has the property that $\mathsf{I}\mathbf{x} = \mathbf{x}$. Accordingly we define the 2×2 matrix

$$\frac{1}{a_{11}a_{22} - a_{12}a_{21}} \begin{pmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{pmatrix} = \mathsf{A}^{-1}$$

and call this the **inverse** of A. This matrix has the property that $\mathsf{A}^{-1}\mathsf{A} = \mathsf{I}$.

Note: The **determinant** $\det(\mathsf{A}) = a_{11}a_{22} - a_{21}a_{12}$ and the inverse does not exist if $\det(\mathsf{A}) = 0$.

Note: If $\mathsf{A}\mathbf{x} = \mathbf{b}$ and $\det(\mathsf{A}) = 0$ then either: (i) if $\mathbf{b} = \mathbf{0}$ then there are infinitely-many solutions; or (ii) if $\mathbf{b} \neq \mathbf{0}$ then either none or infinitely-many solutions exist.

These ideas generalise to systems larger than just 2×2. Thus we take (13.2) and pre-multiply by its inverse A^{-1} and it follows that

$$\mathsf{A}^{-1}\mathbf{b} = \mathsf{A}^{-1}\mathsf{A}\mathbf{x} = \mathsf{I}\,\mathbf{x} = \mathbf{x}$$

This is fine. We know what A^{-1} is if the matrix A is 2×2. But what about an $n \times n$ matrix ? The short answer is you don't want to know !

The point is, we want to solve equations like (13.2) for systems of size n which might be very large. The notion of an inverse is very useful, but we have to remember that writing

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

is largely *symbolic* and the actual process of computing A^{-1} is only really practicable with the aid of automated algorithms.

Because of its importance in applications, this is a subject which forms a large part of numerical analysis, partly because the process of inverting a linear system of equations is so numerically expensive (at its basic level, doubling the size of the matrix implies the time taken by a computer to invert goes up by a factor of 8).

Luckily...

9.1.3 Matlab and linear algebra

... most of the time, you don't need to know how such things are done. Matlab actually stands for 'matrix laboratory' and was initially conceived as a programming tool to aid computations which are largely based on linear algebra. Let's see what we can do:

We've already seen that arrays can store numbers in a manner corresponding to matrices and vectors.

Vectors:

Start by defining **row vectors**, $\mathbf{a} = (1,2,3)$, $\mathbf{b} = (4,1,6)$.

```
>> a = [1 2 3];
>> b = [4 1 6];
>> length(a)        % length of a vector

>> b(2) = 5         % overwite 2nd element of vector

>> a*b'             % * is matrix multiplication, ' is transpose
```

which is $\mathbf{a}\mathbf{b}^T = 1 \times 4 + 2 \times 5 + 3 \times 6$. This is the same as the **dot product**

```
>> dot(a,b)
```

but $\mathbf{a}^T\mathbf{b}$ is a 3×3 matrix: $\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} (4,5,6) = \begin{pmatrix} 4 & 5 & 6 \\ 8 & 10 & 12 \\ 12 & 15 & 18 \end{pmatrix}$:

```
>> a'*b
```

Notes: (i) » `a*b` is meaningless; (ii) » `a.*b` is not a linear algebra command. Here it produces the vector `[ 4 10 18 ]`.

The vector **cross product** is

```
>> cross(a,b)
```

is $\mathbf{a} \times \mathbf{b}$ (only meaningful for 3-vectors).

Column vectors are defined by

```
>> b = [ 4 ; 5 ; 6 ]

>> b(1)                % print 1st element

>> a*b                 % matrix multiplication
```

is now equivalent to the dot product.

Norms:

These indicate the '**size**' of a vector. Two of the the main practical norms are: (i) the **2-norm**

$$\|\mathbf{a}\| = \|\mathbf{a}\|_2 = \sqrt{\sum_{j=1}^{n} a_j^2}$$

is the usual one, where $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ (in two or three dimensions the 2-norm represents the length of the vector); and (ii) the **1-norm**

$$\|\mathbf{a}\|_1 = \sum_{j=1}^{n} |a_j|.$$

By default, Matlab assumes the 2-norm so

```
>> norm(a)

>> norm(a,2)

>> sqrt(a*a')
```

are all the same whilst the 1-norm of **a** is

```
>> norm(a,1)
```

Matrices:

Take $\mathsf{C} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix}$ then

```
>> C = [ 1 2 3 ; 2 3 4 ; 3 4 5 ]

>> C(2,2)                   % access (2,2) element
```

```
>> C(:,2)                  % isolate 2nd column

>> C(3,:)                  % ... or 3rd row

>> C'                      % Transpose the matrix
```

Multiplication is fairly obvious:

```
>> C*a'                    % This is matrix C into vector a transpose

>> D = [ 0 1 ; 1 1 ; 1 0 ] % D is a 3 by 2 matrix

>> C*D                     % This is C times D

>> D*C
```

The last one is illegal because the matrix sizes don't match.

Inverses, determinants:

```
>> det(C)                  % determinant of matrix C

>> inv(C)                  % inverse of matrix C

>> inv(D)
```

The last one is illegal because you cannot invert a non-square matrix.

Solving systems of equations:

If we want to solve $C\mathbf{x} = \mathbf{a}$ then $\mathbf{x} = C^{-1}\mathbf{a}$ can be done » `x = inv(C)*a` but is most easily (and efficiently) done in Matlab using

```
>> x = C\a                 % Note the BACKSLASH
```

Special Matrices:

```
>> P = zeros(4,4)          % 4x4 matrix of zeros

>> P = ones(4,5)           % 4x5 matrix of ones

>> P = eye(3)              % 3x3 identity matrix

>> P = rand(4,4)           % 4x4 random matrix
```

9.2 Eigenvalues and eigenvectors of a matrix A

Unless you've done them at school, you won't yet have seen **eigenvalues** and **eigenvectors**. Although it isn't yet so obvious why they are important, you will see throughout your course that they crop up in all areas of mathematics. And like inverses, they are not generally easy to compute unless you are working with systems of equations of order 2. Hence computational methods/knowing that how to use a computer to find them is important.

Definition: If A is an $n \times n$ matrix, then an **eigenvalue** λ is a scalar and its associated **eigenvector**, $\mathbf{v} \neq \mathbf{0}$, is a $n \times 1$ vector satisfying

$$\mathsf{A}\mathbf{v} = \lambda \mathbf{v} \tag{9.2}$$

The geometric interpretation of this is that the vector formed by the matrix/vector multiplication $\mathsf{A}\mathbf{v}$ points in the same direction of $\mathbf{v}$ but stretches $\mathbf{v}$ by the factor λ.

Note: we are not *given* either λ or $\mathbf{v}$; they *belong to the matrix* and have to be found. In fact, there are n such distinct eigenvalue/eigenvector pairs for an $n \times n$ matrix.

Note: We also see that if $\mathbf{v}$ is an eigenvector, then so is $\mu\mathbf{v}$ for any scalar μ. So their directions are defined but their size is arbitrary.

We also see that $\mathbf{v} = 0$ satisfies (13.3). This is not an interesting result; nor does it define an eigenvector.

In order to find eigenvalues and eigenvectors, we write (13.3) as

$$(\mathsf{A} - \lambda \mathsf{I})\mathbf{v} = \mathbf{0} \tag{9.3}$$

Since we want $\mathbf{v} \neq \mathbf{0}$ we must have that

$$\det(\mathsf{A} - \lambda \mathsf{I}) = 0 \tag{9.4}$$

and this essentially determines an nth order polynomial that λ has to satisfy and this leads to the n eigenvector/eigenvalue pairs.

9.2.1 Example by hand

Consider the 2×2 matrix

$$\mathsf{A} = \begin{pmatrix} 1 & 1 \\ 4 & 1 \end{pmatrix}$$

What are the eigenvalue/eigenvector pairs ? We use (10.5). So λ given by

$$\det\left(\begin{pmatrix} 1 & 1 \\ 4 & 1 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right) = 0$$

or

$$\det \begin{pmatrix} 1-\lambda & 1 \\ 4 & 1-\lambda \end{pmatrix} = 0$$

or

$$(1-\lambda)^2 - 4 = 0 \qquad \Rightarrow \lambda^2 - 2\lambda - 3 \Rightarrow (\lambda-3)(\lambda+1) = 0$$

and so $\lambda = 3$ or $\lambda = -1$. To find the eigenvector associated with $\lambda = 3$ return to (12.3), writing $\mathbf{v} = (v_1, v_2)^T$:

$$\begin{pmatrix} 1-3 & 1 \\ 4 & 1-3 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \mathbf{0}$$

and both lines of the two equations represented by this matrix/vector multiplication gives the same equation: $-2v_1 + v_2 = 0 \;\Rightarrow\; v_2 = 2v_1$. Therefore, if $v_1 = 1$, for e.g. $v_2 = 2$ and so the eigenvector associated with $\lambda = 3$ is $\mathbf{v} = (1, 2)^T$.

Exercise: Repeat with the eigenvalue $\lambda = -1$ to show that its eigenvector is $\mathbf{v} = (1, -2)^T$.

Note: As soon as the matrix is bigger than 2×2, the polynomial equation for λ is higher than quadratic and our chances of finding roots by hand are more remote which is why we need to use computational methods.

9.2.2 Using Matlab

```
>> P = rand(4,4)            % set up a 4x4 random matrix

>> eig(P)
```

gives a column vector containing the eigenvalues, or

```
>> [V,d] = eig(P)
```

produces a matrix V of eigenvectors and a column vector **d** of eigenvalues.

Example:

```
>> v1 = V(:,1)              % first column of V assigned to vector v1

>> P*v1

>> d(1)*v1
```

The last two lines are: (i) the matrix times the eigenvector and (ii) the eigenvalue times the eigenvector... they are the same as this is how they are defined !

9.2.3 Worksheet example

It can be shown (not here) that if an $n \times n$ matrix A has eigenvalues λ_i with modulus less than one (i.e. $|\lambda_i| < 1$) then

$$(\mathsf{I} - \mathsf{A})^{-1} = \mathsf{I} + \mathsf{A} + \mathsf{A}^2 + \mathsf{A}^3 + \ldots = \mathsf{I} + \sum_{j=1}^{\infty} \mathsf{A}^j$$

This can be thought of as the matrix analogue of the MacLaurin series result for a scalar variable x:

$$(1 - x)^{-1} = 1 + x + x^2 + x^3 + \ldots$$

provided $|x| < 1$.

9.3 Problems

1. In Matlab with your own choice of 3-vectors $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, confirm numerically the following vector results:

$$\begin{array}{rl} \text{(i)} & \mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \cdot \mathbf{c})\mathbf{b} - (\mathbf{a} \cdot \mathbf{b})\mathbf{c}; \\ \text{(ii)} & \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) = \mathbf{b} \cdot (\mathbf{c} \times \mathbf{a}) = \mathbf{c} \cdot (\mathbf{a} \times \mathbf{b}); \\ \text{(iii)} & [\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})]\mathbf{a} = (\mathbf{a} \times \mathbf{b}) \times (\mathbf{a} \times \mathbf{c}). \end{array}$$

2. Use Matlab to solve the system of equations

$$\begin{aligned} 3x_1 - x_2 + 4x_3 &= 0 \\ x_2 - x_3 &= 1 \\ 2x_1 + 6x_2 - x_3 &= 1. \end{aligned}$$

   [*Add your answers on your printout to Q4.*]

3. A position in space represented by the vector $\mathbf{x} = (x, y, z)^T$ is rotated through an angle α around the x-axis, an angle β around the y-axis and then an angle γ around the z-axis to a new position $\mathbf{r} = (r, s, t)^T$ via the mapping

$$\mathbf{r} = \mathsf{R}_z(\gamma)\mathsf{R}_y(\beta)\mathsf{R}_x(\alpha)\mathbf{x}$$

   where all angles are measured anti-clockwise and

$$\mathsf{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}, \qquad \mathsf{R}_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix},$$

$$\mathsf{R}_z(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

a) Write a Matlab function, `vecmap` which takes $\mathbf{x}$, α, β and γ as its input and outputs $\mathbf{r}$. The function should start `function r = vecmap(x,alpha,beta,gamma)`

b) Let $\mathbf{x} = (1,1,1)^T$ be a column vector. Call `r = vecmap(x,pi,pi,pi)` to see that the code works. How do we know whether it has worked for this case ?

c) An orthonormal matrix, R, satisfies the relation $\mathsf{R}^T = \mathsf{R}^{-1}$. Show that each of the rotation matrices above are orthonormal. Hence deduce a simple inverse mapping from $\mathbf{r}$ to $\mathbf{x}$.

4. a) Consider the following recurrence relation: $\mathbf{y}_j = \mathbf{x} + \mathsf{A}\mathbf{y}_{j-1}$, for $j = 1,2,\ldots,m$. with $\mathbf{y}_0 = \mathbf{x}$ where A, $\mathbf{x}$ is fixed a matrix, vector. By hand, demonstrate that

$$\mathbf{y}_m = \mathbf{x} + \mathsf{A}\mathbf{x} + \mathsf{A}^2\mathbf{x} + \ldots \mathsf{A}^m\mathbf{x} \equiv \mathbf{x} + \sum_{j=1}^{m} \mathsf{A}^j\mathbf{x}$$

b) Write a Matlab script `invexp` which performs the following tasks: (i) defines a 2×2 random matrix A and a column vector $\mathbf{x} = (1,1)^T$; (ii) computes and prints the eigenvalues of A; (iii) computes and prints the values of $(\mathsf{I} - \mathsf{A})^{-1}\mathbf{x}$; (iv) computes and prints the vector $\mathbf{y}_m$ defined in part (a) where m should be set to a value of at least 200.

c) Run your script. You should confirm that $\mathbf{y}_m$ is a good approximation to $(\mathsf{I} - \mathsf{A})^{-1}\mathbf{x}$ when both eigenvalues of A are less than one in modulus, otherwise the expression for $\mathbf{y}_m$ fails to converge as m increases.

[Publish *results for an example where* $\mathbf{y}$ *does converge to the value of* $(\mathsf{I} - \mathsf{A})^{-1}\mathbf{x}$]

5. Prove that if a matrix has zero determinant then at least one of its eigenvalues must be zero.

6. In Matlab, set `» A = rand(4,4);` and `» B = rand(4,4);` and confirm numerically which of these propositions are true:

a) $\det(\mathsf{A})$ equals the product of the eigenvalues of A.

b) The eigenvalues of A^T are the same as A.

c) The eigenvalues of A^{-1} are reciprocals of the eigenvalues of A.

d) The eigenvalues of AB are the products of the eigenvalues of A and B.

e) The eigenvalues of $\mathsf{A} + \mathsf{B}$ are the sum of the eigenvalues of A and B.

[*Add your answers on your printout to Q4.*]

7. Try this script, which overlays plots of eigenvalues in the complex plane from 100 simulations of $n \times n$ random matrices.

```
n = 20                     % matrix size
clf                        % clear the graphics frame
hold on                    % allow overlaying of graphical output
```

```
for j=1:100                 % do 100 simulations
    A = rand(n,n);          % Each loop is a new random matrix of size n
    d = eig(A);             % Compute eigenvalues
    x = real(d)/sqrt(n);    % x and y are their real and imaginary parts
    y = imag(d)/sqrt(n);    % normalised by sqrt(n)
    plot(x,y,'*')           % overlay the eigenvalues on complex plane
end
hold off                    % take off overlaying of graphical output
```

Try increasing the value of n to 120. It's kind of interesting and related to something called Girko's Circular Law. Girko's Circular Law requires the matrix to be formed by random numbers taken from a standard normal distribution with zero mean. See if you can find how to adapt the Matlab script to do this and see how it changes the results.

10 Week 10

10.1 The power method/iteration

The purpose of the Power method (or iteration) is to determine the largest eigenvalue (the dominant eigenvalue) of a given matrix and its associated eigenvector. It is particularly useful if a matrix A is very large and sparse (meaning that many of its entries are zero) since under such circumstances it is generally much more numerically efficient (quicker) to use the power iteration than to find eigenvalues by other direct methods.

10.1.1 Derivation

We assume an $n \times n$ matrix A has eigenvalues $\lambda_1, \ldots, \lambda_n$, ordered such that

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|. \tag{10.1}$$

We also assume that A is *diagonalisable*, meaning that the eigenvectors $\mathbf{v}_j$, $j = 1, 2, \ldots, n$ satisfying $\mathsf{A}\mathbf{v}_j = \lambda_j \mathbf{v}_j$ are linearly independent. (This is what you expect in the normal course of events.) This means that the set $\{\mathbf{v}_j\}$ forms a basis in $\mathbb{R}^n$.

We pick a random $n \times 1$ vector $\mathbf{x}_0$ in $\mathbb{R}^n$ which is capable of being expressed as a linear combination of the eigenvectors,

$$\mathbf{x}_0 = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \ldots + c_n\mathbf{v}_n$$

for some $c_j \neq 0$. Then define $\mathbf{x}_1$ by

$$\mathbf{x}_1 = \mathsf{A}\mathbf{x}_0 = \sum_{j=1}^{n} c_j \mathsf{A}\mathbf{v}_j = \sum_{j=1}^{n} c_j \lambda_j \mathbf{v}_j = \lambda_1 \sum_{j=1}^{n} c_j \left(\frac{\lambda_j}{\lambda_1}\right) \mathbf{v}_j$$

Continue with $\mathbf{x}_2 = \mathsf{A}\mathbf{x}_1$ and so on and repeating this process iteratively k times we end up with

$$\mathbf{x}_k = \mathsf{A}\mathbf{x}_{k-1} = \ldots = \mathsf{A}^k\mathbf{x}_0 = \lambda_1^k \sum_{j=1}^{n} c_j \left(\frac{\lambda_j}{\lambda_1}\right)^k \mathbf{v}_j$$

and on account of the assumption (10.6), we see that $(\lambda_j/\lambda_1)^k \to 0$ as $k \to \infty$ if $j \neq 1$ else $(\lambda_j/\lambda_1)^k = 1$ of $j = 1$. I.e.

$$\mathbf{x}_k \to \lambda_1^k c_1 \mathbf{v}_1, \qquad \text{as } k \to \infty$$

The rate of convergence is controlled by the ratio $|\lambda_2/\lambda_1|$; the smaller this is, the faster the method will converge.

We see that this allows the eigenvector $\mathbf{v}_1$ to be determined since $\mathbf{x}_k/\|\mathbf{x}_k\| \approx \mathbf{v}_1/\|\mathbf{v}_1\|$ defines the eigenvector as a unit vector. Once the eigenvector is known we may use the definition $\mathsf{A}\mathbf{v}_1 = \lambda_1\mathbf{v}_1$ to give

$$\mathsf{A}\mathbf{v}_1 \cdot \mathbf{v}_1 = \lambda_1\mathbf{v}_1 \cdot \mathbf{v}_1 \qquad \Rightarrow \qquad \lambda_1 = \frac{\mathsf{A}\mathbf{v}_1 \cdot \mathbf{v}_1}{\mathbf{v}_1 \cdot \mathbf{v}_1}$$

(This is called the *Rayleigh Quotient*).

10.1.2 Summary

For a matrix A choose a random vector $\mathbf{x}_0$ s.t. $\|\mathbf{x}_0\| = 1$. Then iterate $\mathbf{x}_k = \mathsf{A}\mathbf{x}_{k-1}/\|\mathsf{A}\mathbf{x}_{k-1}\|$. It follows that $\mathbf{x}_k \to \mathbf{v}_1$ and $\lambda_1 \to \mathsf{A}\mathbf{x}_k \cdot \mathbf{x}_k$ as $k \to \infty$, the largest eigenvalue/eigenvector pair of A.

10.1.3 Algorithm

This forms the basis of the power method algorithm:

- function: input A, *kmax*; output $\mathbf{x}_k$, λ. So `function [x,d] = power(A,kmax)` so `x` represent $\mathbf{x}_k$ and `d` represents λ.
- Determine size, n, of A. Use `[n,m] = size(A);`.
- Choose a random n-dimensional column vector to represent $\mathbf{x}_0$ and then normalise it such that $\|\mathbf{x}_0\| = 1$. I.e. `x = rand(n,1);` followed by `x = x/norm(x);`.
- loop from $k = 1$ to $k = kmax$
- Iterate with: $\mathbf{x}_k = \mathsf{A}\mathbf{x}_{k-1}/\|\mathsf{A}\mathbf{x}_{k-1}\|$ so that we re-normalise the new iterate at each step. In the code, we can overwrite the previous iterate `x` with the new one.
- end loop
- $\lambda = \mathsf{A}\mathbf{x}_k \cdot \mathbf{x}_k$ (or $\lambda = \mathbf{x}_k^T\mathsf{A}\mathbf{x}_k$).
- end function

10.2 Google's PageRank algorithm

The power method is particularly useful in the theory of directed graphs. One prominent example of this is applied to the world wide web and how search engines rank the importance of web pages. The most well known of these is the **"PageRank" algorithm** invented by Larry Page and Sergey Brin in 1998 while they were graduate students at Stanford and underpinned the success of Google[1]. The particular idea that Brin and Page had was that the importance of a web page was determined by the links that are made to that page, but not in a trivial way.

For the purposes of illustration, imagine the world wide web consists of four web pages labelled, 1, 2, 3, 4. (In fact there are over 20 billion – you can see the scale of the problem !).

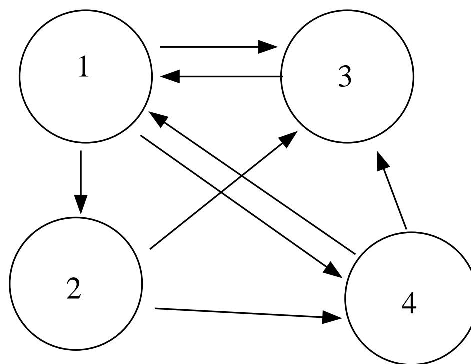


Figure 10.1: A model web with 4 pages with links connecting them. Can we rank the pages in importance ?

As we don't know anything about the page ranking in advance, let's assume initially (i.e. at step zero) that they all have the same page rank, and call it $p_i^{(0)} = \frac{1}{4}$ for $i = 1, \ldots, 4$. This is done so that the cumulative page rank

$$\sum_{j=1}^{4} p_j^{(0)} = 1.$$

If we let $\mathbf{p}^{(0)} = (p_1^{(0)}, p_2^{(0)}, p_3^{(0)}, p_4^{(0)})^T$ then the condition above is the same as $\|\mathbf{p}^{(0)}\|_1 \equiv \sum_{j=1}^{4} p_j^{(0)} = 1$ since all elements of $\mathbf{p}^{(0)}$ are positive.

Now we make a step forwards in which each node j donates its current page rank equally amongst all of the nodes it links to.

[1]It's almost impossible to believe that before Google, web searches were almost completely pointless and performed only in an act of desperation or blind hope. But I remember these times well, and also remember the exact moment someone told me to try Google.

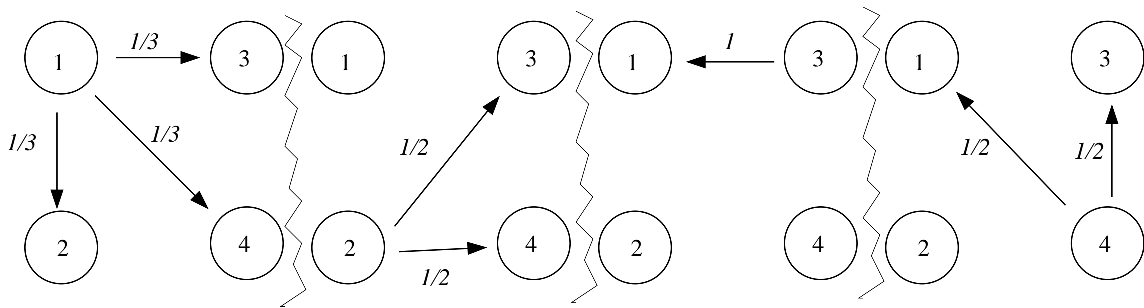


Figure 10.2: Redistributing the weighting of the pages.

We update the page ranking to reflect this so that mathematically you have reached a new page rank state $\mathbf{p}^{(1)}$ with

$$p_1^{(1)} = 1.p_3^{(0)} + \tfrac{1}{2}p_4^{(0)}, \qquad p_2^{(1)} = \tfrac{1}{3}p_1^{(0)}, \qquad p_3^{(1)} = \tfrac{1}{3}p_1^{(0)} + \tfrac{1}{2}p_2^{(0)} + \tfrac{1}{2}p_4^{(0)}, \qquad p_4^{(1)} = \tfrac{1}{3}p_1^{(0)} + \tfrac{1}{2}p_2^{(0)}$$

which can be written

$$\mathbf{p}^{(1)} = \mathsf{A}\mathbf{p}^{(0)}, \qquad \text{and here } \mathsf{A} = \begin{pmatrix} 0 & 0 & 1 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{pmatrix}$$

Note: General rule is, A_{ij} is either 0 if there are no links from j to i or $1/l_j$ if there is a link from j to i where l_j is the total number of links from page j.

Now we simply iterate this process so $\mathbf{p}^{(2)} = \mathsf{A}\mathbf{p}^{(1)} = \mathsf{A}^2\mathbf{p}^{(0)}$ and in general

$$\mathbf{p}^{(k+1)} = \mathsf{A}\mathbf{p}^{(k)} = \mathsf{A}^k\mathbf{p}^{(0)} \tag{10.2}$$

We find that $\mathbf{p}^{(k)} \to \mathbf{P}$ as $k \to \infty$, and thus $\mathbf{P}$ satisfies

$$\mathbf{P} = \mathsf{A}\mathbf{P}. \tag{10.3}$$

In other words $\mathbf{P}$ is an eigenvector of A and is associated with an eigenvalue $\lambda = 1$.

Remark: This is no fluke. It can be proved that $\lambda = 1$ is the *largest* eigenvalue of A (because of its special structure in that all columns add up to unity – it's called a stochastic matrix) and that $\|\mathbf{p}^{(k)}\|_1 = 1$ for all $k \geq 1$ provided $\|\mathbf{p}^{(0)}\|_1 = 1$.

This latter property is easily seen from the way in which the algorithm works; values given to the system initially are conserved and simply redistributed at each step.

The outcome of this theory is that the iterative step $\mathbf{p}^{(k+1)} = \mathsf{A}\mathbf{p}^{(k)}$ being applied is nothing more than the power method, in which the iterates are converging to the eigenvector $\mathbf{P}$ associated with the largest eigenvalue 1.

10.2.1 The random surfer

There is another way of looking at the iterative step which helps first identify and then remedy a problem with the method described above. That is from the point of view of random surfers on the web. Thus the step from k to $k+1$ is the same as a surfer at node j who randomly clicks on one of the links with a probability given by $1/l_j$. Thus over a large number of iterations, we are simulating, on average, how often a web surfer making random clicks visits a particular node and this is used as the basis for ranking the importance of the page.

This does introduce a problem. E.g. the web has disconnected elements.

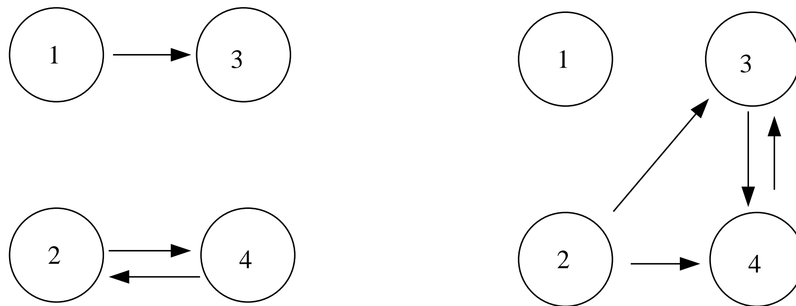


Figure 10.3: A web with disconnected elements.

Both examples have problems, but let's look at the sketch on the right. Now $\mathbf{p}^{(k+1)} = \mathsf{A}\mathbf{p}^{(k)}$ implies that the element $p_1^{(k)} = p_1^{(0)}$ for all k and its value therefore stays at $\frac{1}{4}$. Or, in other words, the random surfer is stuck on node 1 and is not able to infer anything about the importance of this site in relation to the others. This problem will occur in any web in which there are a set of pages disconnected from another set, as in the illustration on the left above.

So what Brin & Page proposed was to allow the random surfer with probabilty $1-d \approx 0.15$ (so $d \approx 0.85$ – Brin and Page called this parameter a *damping parameter*) to jump to a new node anywhere on the web rather than randomly following clicks from pages that are connected.

Now, instead of just the matrix A, we use

$$\mathsf{M} = d\mathsf{A} + (1-d)\mathsf{B}, \qquad \text{where } \mathsf{B} = \frac{1}{n}\begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & \ddots & & \vdots \\ \vdots & & & \\ 1 & \dots & & 1 \end{pmatrix}$$

where n is the number of nodes on the web. So we now use the rule

$$\mathbf{p}^{(k+1)} = \mathsf{M}\mathbf{p}^{(k)} = \mathsf{M}^k\mathbf{p}^{(0)} \tag{10.4}$$

with $\|\mathbf{p}^{(0)}\|_1 = 1$ as before.

Note: It can be shown that M is still stochastic and so $\mathbf{p}^{(k)} \to \mathbf{P}$ (a different vector now) as before in the same way.

10.2.2 Matlab code

This is essentially the same as the power method. We input ϵ as the *tolerance* (a prescribed accuracy we must target before stopping the iteration) and output the converged pagerank vector and the number of iterations taken to converge.

Inside the code are two new coding tricks worthy of note:

(i) We call a function (here it is called `Adefine` and sets the matrix A) from within the function `pagerank`.

The process of breaking code down into small chunks which each perform single tasks is very useful when it comes to writing more complex numerical codes.

(ii) We use a conditional loop `while` to loop continuously while a condition is met. The general syntax is

```
while condition is met
   ... perform actions
end
```

The looping stops when the condition is broken.

Here's the pagerank code demonstrated in the lectures:

```
function [p,k] = pagerank(tol) % input: tol = tolerance for iteration
                               % output: p = pagerank vector, k = # of loops

d = 0.85;           % set the damping factor
A = Adefine         % call function Adefine to set matrix A
[n,m] = size(A);    % set n = size of matrix A (m = # of cols; not needed)
B = ones(n,n)/n;    % set the matrix B as in the notes

p = rand(n,1);      % p = random column vector
p = p/norm(p,1);    % and normalise using the 1-norm

p_old = zeros(n,1); % set p_old zero column vector so that ||p-p_old|| > tol
                    % the first time we go into the while loop

k = 0;              % k is a loop counter

while (norm(p-p_old) > tol)   % while: the conditional loop... will
```

```
                                % continue to loop until the distance
                                % between p and p_old, the previous iterate,
                                % is less than tol.

    p_old = p;                  % about to update p, so p_old becomes p
    p = (d*A+(1-d)*B)*p;        % iterate according to step in notes
    k = k+1;                    % add one to k everytime we loop

end                             % end while loop: iteration converged
end
```

10.2.3 Alternative methods

Method 2:

Since $\|\mathbf{p}^{(k)}\|_1 = 1$, it follows that

$$\mathsf{B}\mathbf{p}^{(k)} = \frac{1}{n}\mathbf{1}$$

where $\mathbf{1}$ is the $n \times 1$ vector with one in each entry. So the iterative step (12.3) is equivalent to

$$\mathbf{p}^{(k+1)} = d\mathsf{A}\mathbf{p}^{(k)} + \frac{1-d}{n}\mathbf{1} \tag{10.5}$$

Remark: This is particularly efficient because, in a real web, the matrices are obviously very very big, but because pages often only link to a handful of other pages, the matrix A is very sparse (this means that it is mainly zeros). There are very efficient methods for matrix multiplication of sparse matrices. In contrast, in our first method, the matrix M is full (i.e. not at all sparse).

Method 3:

Since we know $\mathbf{p}^{(k)} \to \mathbf{P}$ as $k \to \infty$ (10.5) now tends to

$$\mathbf{P} = d\mathsf{A}\mathbf{P} + \frac{1-d}{n}\mathbf{1}$$

and so we have a direct method of computing the page ranking vector without the need for iteration:

$$\mathbf{P} = \frac{1-d}{n}(\mathsf{I} - d\mathbf{A})^{-1}\mathbf{1} \tag{10.6}$$

BUT, because inverting matrices (even sparse matrices) is numerically expensive, this is not a practical method for large systems.

10.3 Problems

1. a) Write a Matlab function `power`, say, to implement the power method for finding the largest eigenvalue and its associated eigenvector. Based your function on the algorithm in §10.1 of the notes.

   b) Define a random 4×4 Matrix A in Matlab by `>> A = rand(4,4);`. In the Matlab command window, find the eigenvalues and eigenvectors of A. [**Note:** Find an example where these are all real.]

   c) Use your Matlab function `power` with the same random matrix A in (b) to confirm that it returns the largest eigenvalue and its associated eigenvector.

2. Download the Matlab functions pagerank.m and Adefine.m from the course web page and test the code on the following model of the world wide web with 5 pages:

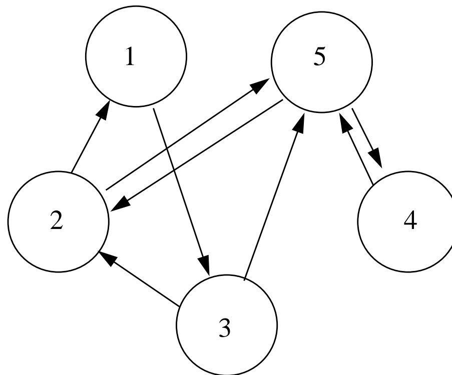


Figure 10.4: A model web with 5 pages with links connecting them.

   a) Edit the definition of your matrix within the function `Adefine` to encode the links in the web above (follow §10.2 of the notes).

   b) Run your `pagerank` code with a tolerance of 10^{-5} to determine the order the pages are ranked.

   c) Prove that if A is a stochastic matrix (column elements sum to 1) and $\|\mathbf{p}\|_1 = 1$ then $\|\mathbf{q}\|_1 = 1$ also where $\mathbf{q} = \mathsf{A}\mathbf{p}$ and $\mathbf{p}$, $\mathbf{q}$ contain only positive elements.

3. In §10.2.2 of the notes we noted a revised iterative scheme (Method 2):

$$\mathbf{p}^{(k+1)} = d\mathsf{A}\mathbf{p}^{(k)} + \frac{1-d}{n}\mathbf{1}$$

   where $\mathbf{1}$ is the $n \times 1$ vector with one in each entry.

   a) Make a copy of the function pagerank.m from Exercise 2 and edit this to implement the revised iterative scheme above.

b) Test your code on the same example as in Exercise 2 with a tolerance of 10^{-5} to determine the order in which pages are ranked (they should be the same as in Exercise 2).

[Publish *your code and the output of part (b), using* Edit Publish Options *to revise the function call appropriately.*]

4. In §10.2.2 of the notes (Method 3), we showed that a direct calculation of the page rank vector, $\mathbf{P}$, is given by

$$\mathbf{P} = \frac{1-d}{n}(\mathsf{I} - d\mathsf{A})^{-1}\mathbf{1}$$

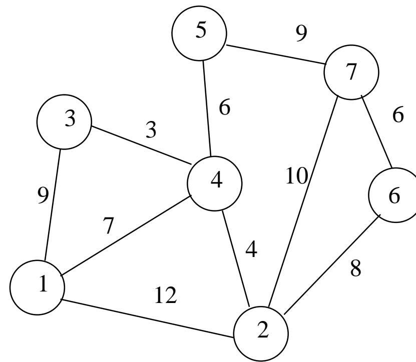
a) Write a function which implements this scheme to determine $\mathbf{P}$. Within your function, A should be defined by a call to the function `Adefine` (as in the code pagerank.m).

b) Test your code on the same example as in Exercise 2 to determine the order in which pages are ranked (they should be the same as in Exercise 2).

[Publish *your code and the output of part (b).*]

5. [PRIZE PROBLEM 2][2]

Consider a road network connecting n junctions which are numbered as nodes $1, 2, \ldots, n$ on a graph. The lines of the graph connecting the nodes represent roads and are given numerical values which represent the travel time along those roads (e.g. figure below). Assume that the ordering of the nodes has been performed so that you set out from node 1 and your destination is node n. The task is to determine the route and the shortest time to your destination. The information about the road network is stored in an $n \times n$ matrix A. Thus A_{jk} stores the time from node j to k.



Write a Matlab function to calculate the shortest route to your destination. The input is the matrix A assumed to be supplied by the user in the format described above. The output of the function is the shortest time plus information about the route to be taken. You can use the network above to test your code; I will test your code on my own network.

[2] You can work individually or in teams of up to 5 people. Email your code and your name(s) to me by 5pm Thursday 8th December. A small Xmassy prize will be awarded to the winning entry.

11 Week 11

11.1 Introduction to dynamical systems

Dynamical systems are an important part of applied mathematics. Broadly, they describe the evolution of a system of variables from an initial state through either differential equations (so-called **continuous** dynamical systems) or difference equations (so-called **discrete** dynamical systems). Often they are of interest as the equations are derived from modelling physical phenomena. They can also exhibit fascinating behaviour as we shall see...

11.1.1 Example of a continuous dynamical system

A very simple example of a continuous dynamical system might be a **population model** in which the rate of growth of a population $p(t)$ of a species is proportional to $p(t)$. In other words $p(t)$ satisfies the first order ordinary differential equation (ODE)

$$\frac{dp}{dt} = \alpha p, \qquad t > t_0 \tag{11.1}$$

with $p(t_0) = p_0$ the population at time t_0. We can integrate this by hand to get $p(t) = p_0 \exp^{\alpha t}$ and the population grows exponentially.

This model is generally accurate if there are infinite resources for the population to flourish and no threats from overcrowding, predators etc.

11.1.2 Example of a discrete dynamical system

A discrete dynamical system version of the above would be the following

$$p_n = \alpha p_{n-1}, \qquad n > 0 \tag{11.2}$$

with p_0 given. The solution of this **difference equation** (we recognise it numerically as a recurrence relation) is easy to derive by hand. It's $p_n = \alpha^n p_0$.

Remark: In both cases the system is evolving, in the first with the continous time variable t and in the second with the time-like discrete integer n.

11.1.3 Matlab's ODE solver

To solve a first order ODE such as the one in (11.1) we can use one of Matlab's in-built ODE solver `ode45` as demonstrated in the following Matlab script:

```
%% Population model. Calls separate function popfun
global alpha;                     % shared value of alpha elsewhere in the code
alpha = 1;                        % set value of alpha
[t,y] = ode45(@popfun,[0 1],1);   % solve ODE system over range 0 < t < 1 and
                                  % with initial condition of y(1) = 1
plot(t,y)                         % plot solution against time
```

which calls another Matlab function called `popfun.m`

```
function yd = popfun(t,y)  % input is t and y; output is yd, the derivative
global alpha;              % pick up values of constants from calling script
yd = alpha*y;              % define derivative
end
```

Notes on the code above

- `global` allows the value of a variable set in one part of the Matlab code to be shared with other parts (including the command window). We need this here because α is needed in the function `popfun`.
- `ode45` takes three arguments: (i) the name of function defining the derivative (preceeded by `@`); (ii) the range of values of t over which the ODE is integrated, defined as an array; (iii) the value of the initial condition. The output is an *array* `t` of discrete time steps at which the solver `ode45` computes the solution and the *array* `y` which contains the value of the solution at those time steps.
- The solver `ode45` calls a function which has to be written and saved by the user as a separate Matlab function. It assumes the input is t and y and the output is the computed value of dy/dt. In the example above `popfun.m` does this.

11.2 Prey vs. Predator

Let's now consider a more complicated population model which involves two species whose populations interact with one another.

We let $r(t)$ and $f(t)$ represent the size of the population of **rabbits** and **foxes** as a function of time t. At time $t = 0$, say, the population of rabbits and foxes are given as r_0 and f_0; i.e. $r(0) = r_0$ and $f(0) = f_0$.

A model for the evolution of these populations is given by the **coupled non-linear ODEs**

$$\frac{dr}{dt} = \alpha r - \beta r f, \qquad \frac{df}{dt} = \delta r f - \gamma f, \tag{11.3}$$

for $t > 0$ with $r(0) = r_0$ and $f(0) = f_0$. Here α, β, γ, δ are (assumed positive) constants.

Notes on the model: If there are no foxes the first equation just becomes the population model in (11.1) with a growth rate of α. On the other hand if there are no rabbits then the second equation becomes $df/dt = -\gamma f$ and this integrates to $f(t) = f_0 \exp^{-\gamma t}$ and the foxes die exponentially with the death rate γ.

The two constants β and δ represent interactions between the two populations; how the foxes reduce the rabbit population and how rabbits boost the fox population. These terms should intuitively both be proportional to the sizes of the rabbit and fox populations (hence they both involve the product $r(t)f(t)$).

Remark: The system described above is called the '**Predator-prey model**' or the '**Lokta-Volterra equations**'. The same coupled ODEs apply in the modelling of other physical applications and were originally devised as a model for stock market trading interactions.

11.2.1 Equilibrium solutions

A system is said to be in **equilibrium** if it does not vary with time. I.e. $r(t) = r^*$ and $f(t) = f^*$ for all time t. Substituting this in (11.3) gives

$$0 = \alpha r^* - \beta r^* f^*, \qquad 0 = \delta r^* f^* - \gamma f^*,$$

and we can see that this is satisfied by *either*

$$f^* = \alpha/\beta, \quad r^* = \gamma/\delta$$

or by

$$f^* = 0, \quad r^* = 0.$$

Q: As time increases will the initial rabbit and fox populations evolve towards one of these two equilibrium solutions ?

11.2.2 Solving the ODEs

Equations (11.3) cannot be solved analytically in closed form. Why ? Primarily because they are non-linear. Here, we also have two equations for two unknowns (the equations are coupled) which is far more complicated than in your Calculus 1 course (coupled equations are covered in 2nd year ODEs).

However, we can integrate these equations numerically !

11.2.3 Matlab Code

Here's a script `lv.m` from the website

```
%% Lokta-Volterra model. Calls separate function lvfun
global alpha;   % define variables globally over all parts of the code
global beta;
global gamma;
global delta;
alpha = 1;      % set constants
beta = 0.5;
gamma = 0.8;
delta = 0.3;
[t,y] = ode45(@lvfun,[0 20],[1 1]); % solve ODE system over range
                                    % 0 < t < 20 and with initial condition
                                    % of y(1) = 1, y(2) = 1
plot(t,y(:,1),'b-',t,y(:,2),'r--') % plot variation of rabbits (blue line)
                                   % and foxes (red dashed) versus time

figure(2)       % set up second figure
plot(y(:,1),y(:,2)) % plot variation of rabbits (x-axis) against
                    % foxes (y-axis). Called the PHASE PORTRAIT
```

and this script calls the function `lvfun.m`

```
function yd = lvfun(t,y)      % input is t and VECTOR y; output is VECTOR yd
global alpha;                 % pick up values of constants from calling code
global beta;
global gamma;
global delta;
yd = zeros(2,1);              % output of derivatives must be a COLUMN VECTOR
yd(1) = alpha*y(1)-beta*y(1)*y(2);    % define derivatives: y(1) is r(t)
yd(2) = delta*y(1)*y(2)-gamma*y(2);   % and y(2) is f(t).
end
```

Notes on the code

- The coupled ODE system means that instead of scalar variables we need to use an **array** to store the state variables $r(t)$ and $f(t)$ in `ode45`. This means that the initial conditions are now passed into the function as an array containing the initial values of each of the state variables (r_0 and f_0). It also means the output `y` is no longer a solution array but a solution **matrix** with columns corresponding to each of the state variables evaluated at each time step in the solution.

- `figure(2)` sets up a second graphics window so the solution can be viewed in different ways.
- To run the script type » `lv`. The output will be a plot of two lines both with time t on x-axis against `y(:,1)` representing the first state variable (rabbits) and `y(:,.2)` representing the 2nd state variable (foxes). In a second figure `y(:,1)` is plotted against `y(:,.2)` – this is called a **phase portrait**.

11.2.4 Results

The output of the code in §11.2.3 is shown in Fig 11.1. We can see that the solution is actually cyclic and that when the fox population is low the rabbit population increases and vice versa.

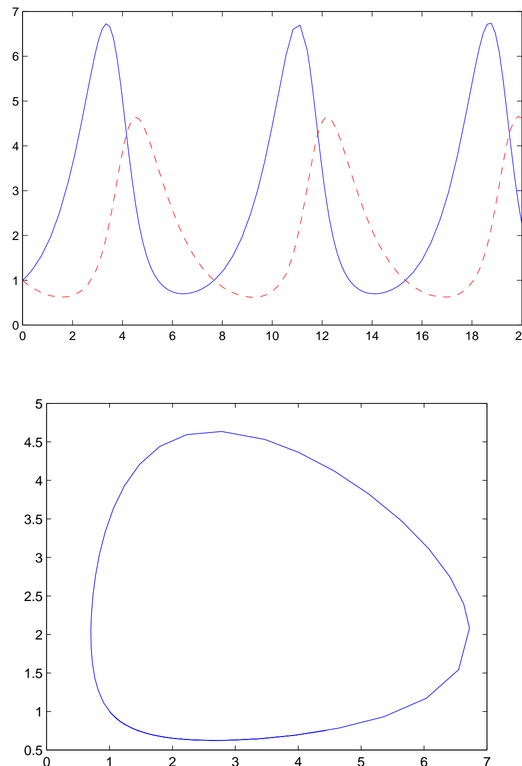


Figure 11.1: Left panel: Rabbits (solid) and foxes (dashed) against time. Right panel: A rabbits versus foxes solution plotted on a phase portrait. The horizontal axis is rabbits; the vertical axis is foxes

Even better is to plot many of these curves on the same graph (see `lv3.m` and Fig 11.3) by varying the initial conditions. This allows you to visualise all possible solutions on a single graph. Note that solutions oscillate around the equilibrium solution which is at $r^* = 8/3$, $f^* = 2$ in this example.

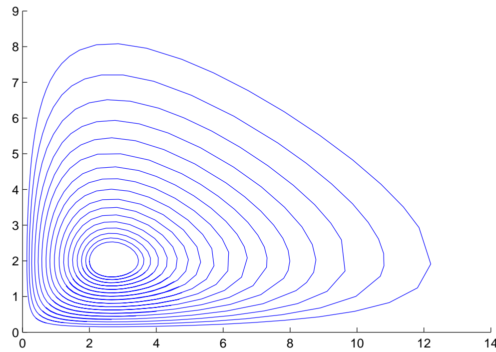


Figure 11.2: Multiple rabbits versus foxes solutions plotted on a phase portrait.

Finally, a **space curve** involves visualising the solution in 3D using Matlab command `plot3` where the last line of the code in §11.2.3 is replaced with

```
plot3(t,y(:,1),y(:,2))  % Note: command plot3 plots t,r(t),f(t) in 3D
rotate3d on             % mouse controlled rotation of the plot
```

11.3 Discrete population model: the logistic map

Instead of (11.2) we could consider a model which simplifies the coupled nature of the predator-prey model by providing a quadratic term which contributes to the demise of the population in addition to the growth term:

$$p_n = \alpha p_{n-1} - \beta p_{n-1}^2$$

If we rescale the variables by writing $p_n = (\alpha/\beta)x_n$ we find

$$x_n = \alpha x_{n-1}(1 - x_{n-1}), \qquad n \geq 1 \tag{11.4}$$

and we choose $x_0 \in (0, 1)$.

11.3.1 Equilibrium solutions

As in the previous model, assume that a constant solutions $x_n = x^*$ exists for all n. Subsituting into (11.4) we gives $x^* = \alpha x^*(1 - x^*)$ and this is satisfied by *either*

$$x^* = 1 - 1/\alpha \qquad or \qquad x^* = 0$$

Q: Will the iterates tend to one of these two constant states as $n \to \infty$? There's only one way to find out ...

11.3.2 Matlab code

Here's the code I came up with to do this

```
alpha = 2.8             % define alpha
n = 75;                 % total number of iterates
x = zeros(1,n);         % set up array
x(1) = 0.5;             % set x_0
for j=2:n               % iterate so that x(n) is last iterate
    x(j) = alpha*x(j-1)*(1-x(j-1));
end
xx = 1:n;               % set up array for x-axis which is the iterate number
plot(xx,x,'-')          % plot xx against x
axis([0 n 0 1])         % put it in a plot of height 0 to 1
end
```

11.3.3 Results

Call » `lmap`. We find that if the value of α is less than 3 the iterates converge to $1 - 1/\alpha$. However, if $3 < \alpha < 3.449$ the iterates now jump between two values, neither of which is $1 - 1/\alpha$. For $3.449 < \alpha < 3.544$ they jump between 4 values and these **period doubling** events continue increasingly rapidly until at $\alpha \approx 3.5699$ the sequence is completely **chaotic**.

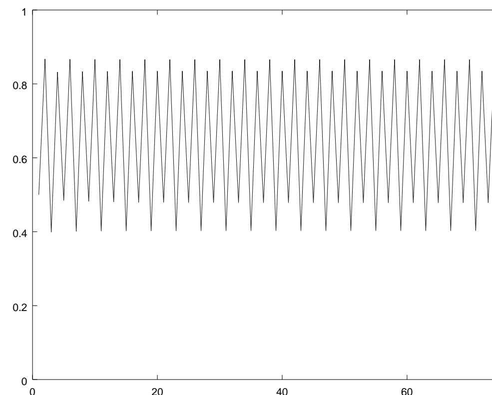


Figure 11.3: Output of » `lmap` with $\alpha = 3.47$ converging to a period-4 solution.

A more complicated extension of this code is also available on the web pages (see `lmap2`) which plots the converged iterates over a range of values of α.

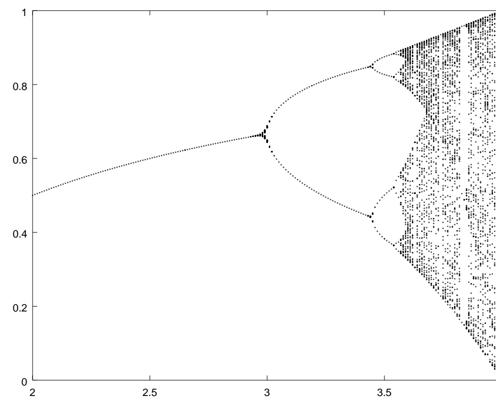


Figure 11.4: Output of » `lmap2(2,0.01,4,0,1)`. The inputs are alpha (start, step, end) and the min/max range vertical range of values.

11.4 The Lorenz Attractor and Chaos

One of the most famous examples of chaos in non-linear systems and often synonymous with the phenonemon known as "the Butterfly effect". This is because the system of equations (below) derived by Edward Lorenz in 1963 as a simplified mathematical model for atmospheric convection have the property two solutions which differ in their initial conditions by a very small amount can evolve to have completely different long time behaviours. This is linked in popular culture to that a butterfly flapping its wings in the South America can be the cause of a typhoon in Asia.

The Lorenz system is given by the following

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y-x)\\ \frac{dy}{dt} &= x(\rho-z)-y\\ \frac{dz}{dt} &= xy-\beta z\end{aligned}$$

Here x, y, z are *not* positions in 3D space; they represent quantities relating to atmospheric convection and temperature gradients.

The constants σ, β and ρ are also related to physical parameters pertaining to fluid dynamics (Prandtl number, a geometric factor and Rayleigh number). It is usual to adopt $\sigma = 10$, $\beta = 8/3$. We shall also take $\rho = 28$.

11.4.1 Equilibrium solutions

If we look for solutions which do not vary with time, (x^*, y^*, z^*) we find that either

$$(x^*, y^*, z^*) = (\pm\sqrt{\beta(\rho-1)}, \pm\sqrt{\beta(\rho-1)}, \rho-1)$$

or

$$(x^*, y^*, z^*) = (0, 0, 0)$$

Q: Again, we can ask if solutions will tend to either of these equilibrium solutions.

11.4.2 Integrating the ODEs

Should be of no surprise that we cannot solve these ODEs by hand (they are non-linear) We can simply adapt the Predator-Prey code to accommodate **3** variables instead of **2**.

Look at the code `lorenz.m` which produces the two figures below:

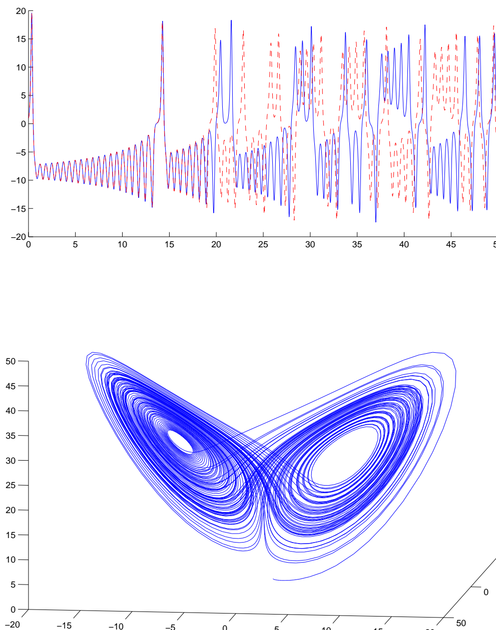


Figure 11.5: Left panel: Variation of $x(t)$ with t for two slightly different initial conditions. Right panel: Phase portrait of the solution as a space curve using `plot3`

11.5 Problems

1. A population $p(t)$ can be modelled (Calculus 1, sheet 10) by the ODE

$$\frac{dp}{dt} = \alpha p - \delta p^2, \qquad t > 0$$

with $p(0) = p_0$ where α, δ represent birth and death rates.

   a) Find all equilibrium solutions, $p(t) = p^*$, a constant.

   b) Adapt the Matlab script `pop.m` (and function `popfun.m`) to plot the solution $p(t)$ over $0 < t < 8$ with $\alpha = 1$ and $\delta = 0.5$ for three initial conditions: (i) $p_0 > 2$; (ii) $p_0 = 2$; and (iii) $0 < p_0 < 2$. (e.g. use `hold on` to allow 3 solution curves to be overlaid onto a single graph)

2. The ODE (Duffing's equation)

$$\frac{d^2x}{dt^2} + \delta\frac{dx}{dt} + \alpha x + \beta x^3 = \gamma \cos t \tag{11.5}$$

describes the displacement $x(t)$ of a point mass under periodic forcing (amplitude γ) attached to a spring which has a damping rate (δ) and non-linear stiffness characterised by α and β.

   a) Confirm that the coupled ODE system below is equivalent to the ODE (11.5)

$$\frac{dx}{dt} = v, \qquad \frac{dv}{dt} = -\delta v - \alpha x - \beta x^3 + \gamma \cos t, \qquad t > 0$$

   b) If $\gamma = 0$ find all equilibrium solutions $x(t) = x^*$, $v(t) = v^*$.

   c) Adapt the Matlab script `lv.m` (and function `lvfun.m`) so that it plots the solution $x(t)$ as function of time t with initial condition $x(0) = 1$, $v(0) = 1$ over $0 < t < 200$ and overlays a plot of the solution with $x(0) = 1$, $v(0) = 1.0001$. Your script should also produce a phase portrait of $x(t)$ against $v(t)$ in a separate plot.

   d) Run your code in two cases: (i) $\alpha = 1$, $\beta = 0$, $\gamma = 0.3$, $\delta = 0.2$; (ii) $\alpha = -1$, $\beta = 1$, $\gamma = 0.3$, $\delta = 0.2$.

   [Publish *your script from part (c) using the parameters in (d)(ii) and add answers to (a) and (b) by hand.*]

3. Download `lmap2.m` from the website and see how the code works to produce Fig. 11.4 in the notes. Investigate different ranges of α to see detail of the logistic map.

4. A pendulum of length l is turned upside down and its lower end vibrated vertically with amplitude a and angular frequency Ω. The ODE describing a small angle $\theta(t)$ made by the pendulum with respect to the vertical can be written in terms of a coupled ODE system

$$\frac{d\theta}{dt} = v, \qquad \frac{dv}{dt} = (\alpha + \beta \cos t)\theta, \qquad t > 0 \tag{11.6}$$

in terms of the auxiliary variable $v(t)$ where $\alpha = \Omega^2 g/l$ ($g = 9.81\text{ms}^{-1}$ is gravity) and $\beta = a/l$ with initial conditions $\theta(0) = \theta_0$, $v(0) = v_0$.

a) What are the equilibrium solutions of (11.6) ?

b) Derive the solution by hand when $a = 0$ and hence show that the equilibrium solution is the only solution which doesn't tend to infinity as $t \to \infty$.

c) Adapt the Matlab code `lv.m` (and `lvfun`.m) to plot the solution $\theta(t)$ as function of time t with initial condition $\theta(0) = 1$, $v(0) = 0$ over $0 < t < 100$. Run your code in two cases: (i) $\alpha = 0.1$, $\beta = 0.5$; and (ii) $\alpha = 0.1$, $\beta = 0.4$. Comment on qualitative the difference in the two sets of results.

d) In addition to $\alpha = 0.1$ try $\alpha = 0.01$ and $\alpha = 0.05$ and values of β ranging between 0.1 and 1 to suggest when it is possible to stabilise an inverted pendulum.

5. The discrete Duffing system is described by the coupled recurrence relations

$$x_n = v_{n-1}, \qquad v_n = -\delta x_{n-1} - \alpha v_{n-1} - \beta v_{n-1}^3, \qquad n \geq 1$$

with $x_0 = 1$, $v_0 = 1$.

a) Write a Matlab script which compute and stores 1000 iterates and then plots n against x_n and, in a separate figure, plots the phase portrait x_n against v_n using dots. [HINT: adapt `lmap.m` code from the notes]

b) Run your script for $\alpha = -2.75$, $\beta = 1$, $\delta = 0.2$. You should observe chaotic behaviour.

[Publish *your script and results to part (b).*]

6. Download `lorenz.m` from the website and see how the code works to produce Figs. 11.5 in the notes.

7. [HARD] Write a Matlab function which computes the path of a pendulum moving in the plane above n magnets equally spaced on a ring of unit radius. The inputs are the initial coordinates (x_0, y_0) of the pendulum (which starts from rest). The output is a plot of $(x(t), y(t))$, for $0 < t < 50$ where the dynamics are given by the equations

$$\frac{d^2x}{dt^2} + 0.025\frac{dx}{dt} + x = \sum_{j=1}^{n} \frac{(x_j - x)}{((x - x_j)^2 + (y - y_j)^2 + 0.01)^{3/2}}$$
$$\frac{d^2y}{dt^2} + 0.025\frac{dy}{dt} + y = \sum_{j=1}^{n} \frac{(y_j - y)}{((x - x_j)^2 + (y - y_j)^2 + 0.01)^{3/2}}$$

and $x_j = \cos(2j\pi/n)$, $y_j = \sin(2j\pi/n)$ for $j = 1, \ldots, n$. Run your code for $n = 3$ and try values of $(x_0, y_0) = (1, 1)$ and small variations about this point. You should notice that the pendulum trajectories are chaotic.

12 Week 12

12.1 The 'Netflix Prize' Problem

Many online sales-based businesses employ "recommender systems" to improve the service they deliver or to tempt users into spending more money on their products. Imagine that you have viewed a set of items or, perhaps, bought a set of items from a web site. A recommender system would attempt to determine which of the other items for sale on that web site that would be most likely to attract your attention and then advertise these items at online checkouts or in ad boxes.

How would a recommender system work ? The online retailer has access to a complete database of users and their habits and the system will work best if it can somehow infer information about your preferences based on the preferences of others. Such an approach is called *collaborative filtering*.

Originally set up in 1999 as an online DVD rental firm, Netflix understood the importance of recommender systems for retaining and expanding its userbase. In 2006 Netflix initiated a prize worth $1m to be awarded to persons or groups of people who could improve on their then current recommender system, called "cinematch". Specifically, the prize demanded that there was a 10% improvement in a specifically prescribed number which determined the efficacy of the system.

The prize was claimed by 2 separate groups: "BellKor's Pragmatic Chaos" and "The Ensemble" in July 2009: both teams had passed the 10% barrier. The prize was awarded to "BellKor's Pragmatic Chaos" as they had submitted their results just 24 minutes earlier than "The Ensemble". The leaderboard can be viewed here:
`http://www.netflixprize.com/leaderboard.html`

12.2 The Netflix Problem

In 2006 the Netflix business had accumulated over 480,000 users and possessed a catalogue of over 18,000 films on DVD. On their system, users could rate films from 1 (poor) to 5 (excellent). Netflix had over 100m user ratings in their database. Netflix were using these user-supplied ratings to recommend other films to users that they hadn't seen yet.

The "real" Netflix problem involves extra data (in particular, the time at which the ratings are made is an important factor). Our model problem outlined below ignores this

level of detail but does include the crux of the general method developed into recommender systems in response to the Netflix problem.

12.3 Model Problem

We are given n users and m films. Users rate films and it's therefore useful to store these ratings in a matrix R, say. That is, the rating user i gives to film j is R_{ij}.

Here's our example with 5 users and 4 films:

$$\mathsf{R} = \begin{pmatrix} 5 & 1 & & 3 \\ 2 & 2 & & \\ 1 & & 3 & \\ & & 3 & 4 \\ 4 & 4 & & 4 \end{pmatrix}$$

Note: The blanks in the matrix are *not* zeros. They are *unknown*. The point of the Netflix algorithm is to decide what these values should be !!

Remark: This isn't as ridiculous as it seems. If we look at R_{53} say, we can imagine that its value should be at least 3, probably 4 or more, but certainly not less than 3. How ? Intuitively, we look at that user and how he/she has rated other films; then we look at the overall rating of that film, but take into account how other users have rating other films as well. This is an important thought process as it allows us to develop a clear way of modelling this mathematically.

12.4 Matrix Factorisation

We suppose that films can be divided into K different categories (e.g. romance, comedy, action, ...). This can be coarse or fine-detailed and the value of K varies accordingly.

The idea is expressed as follows: (i) users can be matched to categories (a number assigns the strength of the relationship between a user and a category); and (ii) films can be matched to categories (this is just film classification). It is worthy of note that in the algorithm below we don't need to know (and hence impose) either of these: the algorithm decides how this is done with no external intervention. This is quite remarkable.

Thus we suppose that

$$\mathsf{R} \approx \mathsf{P}\mathsf{Q}^T = \hat{\mathsf{R}} \tag{12.1}$$

where $\hat{\mathsf{R}} \in \mathbb{R}^{n\times m}$ is the approxiation to R formed by the product of the matrices

$$\mathsf{P} \in \mathbb{R}^{n\times k}, \qquad \mathsf{Q} \in \mathbb{R}^{m\times k}.$$

Here, the entries of P and Q encode the strength of the map between user and category and film and category respectively.

Note: P and Q are matrices with real entries even though R is sparsely populated with integers. Consequently, we expect $\hat{\mathsf{R}}$ to be real and fully populated. Indeed, $\hat{\mathsf{R}}$ will include the predicated ratings not present in R.

The particular decomposition employed in (13.2) is called **matrix factorisation**.

So how do with find P and Q ?

First, we let

$$T = \{(i,j) \mid \mathsf{R}_{ij} \neq 0\}$$

be the set of indices of R which include ratings. Assuming $\hat{\mathsf{R}}$ exists and approximates R, then a measure of the accumulated error is

$$E = \sum_{(i,j)\in T} \mathsf{E}_{ij}^2$$

where

$$\mathsf{E}_{ij} = \mathsf{R}_{ij} - \hat{\mathsf{R}}_{ij} \equiv \mathsf{R}_{ij} - \left(\mathsf{P}\mathsf{Q}^T\right)_{ij} \equiv \mathsf{R}_{ij} - \sum_{k=1}^{K} \mathsf{P}_{ik}\mathsf{Q}_{kj}^T \equiv \mathsf{R}_{ij} - \sum_{k=1}^{K} \mathsf{P}_{ik}\mathsf{Q}_{jk}.$$

Note: The error used by Netflix (the root mean squared error) to measure success is

$$E_{RMS} = \sqrt{E/|T|}, \qquad \text{where } |T| = \text{the number of elements in the set } T.$$

It should also be noted that minimising E is the the same E_{RMS}.

We should choose P and Q to minimise E. That is, the elements in T of the product $\mathsf{P}\mathsf{Q}^T$ should be as close as possible to those of R.

This is not something that can be solved exactly but it can be approached numerically by using a iterative method called a gradient descent method, in which we shuffle downhill on the surface $E(\mathsf{P},\mathsf{Q})$ with small steps trying to find a minimum. This bit is now rather too technical for 1st year maths, but I'll include it before getting to the final, much simpler, equation so that I can't be accused of missing out steps.

Specifically we can write

$$\mathsf{P}_{ik} \to \mathsf{P}_{ik} - \alpha \frac{\partial E}{\partial \mathsf{P}_{ik}}, \qquad \mathsf{Q}_{jk} \to \mathsf{Q}_{jk} - \alpha \frac{\partial E}{\partial \mathsf{Q}_{jk}} \tag{12.2}$$

where α is a small scalar which acts as a numerical parameter controlling how big the steps downhill are and where

$$\frac{\partial E}{\partial \mathsf{P}_{ik}} = \sum_{(i,j)\in T} \left(-2\mathsf{R}_{ij}\mathsf{Q}_{jk} + 2(\mathsf{P}\mathsf{Q}^T)_{ij}\mathsf{Q}_{jk}\right) \equiv -2 \sum_{(i,j)\in T} \mathsf{E}_{ij}\mathsf{Q}_{jk}$$

and

$$\frac{\partial E}{\partial \mathsf{Q}_{jk}} = \sum_{(i,j)\in T} \left(-2\mathsf{R}_{ij}\mathsf{P}_{ik} + 2(\mathsf{P}\mathsf{Q}^T)_{ij}\mathsf{P}_{ik}\right) \equiv -2 \sum_{(i,j)\in T} \mathsf{E}_{ij}\mathsf{P}_{ik}.$$

If we define E_{ij} to be zero when R_{ij} has no entry, then the update step (13.3) simply becomes

$$P \to P + 2\alpha EQ, \qquad Q \to Q + 2\alpha E^T P. \tag{12.3}$$

This just involves matrix multiplication. It tells us how to update P and Q to reduce the error between the non-zero entries of R and their approximate counterparts $\hat{R}$.

We can now apply this iteratively, and this is done in the following code.

12.5 Matlab Code

This is the ultimate code for this course. It combines linear algebra, loops, functions, function calls from within functions, while loops, conditional statements. And in the worksheet exercise will ask you to extend it to make it simulation based.

```
function Rh = netflix(tol,k)  % input tol = tolerance, k = # of categories
                              % Output Rh = completed netflix ratings

alpha = 0.002;   % rate at which the algorithm descends towards the minimum

R = Rdefine      % set up the ratings matrix by calling Rdefine

[n,m] = size(R); % n = # of users, m = # of films

P = rand(n,k);   % P and Q are set to random matrices
Q = rand(m,k);

E = zeros(n,m);  % set aside space for the error matrix

Rh = P*Q';                         % Rhat = P*Q^T at n=0 step
Rh_old = zeros(n,m);               % Rhat_old = zeros at n=0 step

while (norm(Rh - Rh_old) > tol)    % loop until iterates converge

    for i=1:n                      % The error matrix can only sample
        for j=1:m                  % errors against the non-zero data in R
            if R(i,j) ~= 0
                E(i,j) = R(i,j)-Rh(i,j);
            end
        end
    end

    PP = P+2*alpha*E*Q;            % update P & Q to PP and QQ:
```

```
    QQ = Q+2*alpha*E'*P;

    P = PP;                         % Overwrite P & Q with new values, PP & QQ
    Q = QQ;

    Rh_old = Rh;                    % going to update Rh, so store old value
    Rh = P*Q';                      % New value of Rhat

end

end
```

And here's the output of running this code on the example sketched in §11.1.1 with $K = 2$ and a tolerance of 10^{-4}.

```
>> netflix(1e-4,2)
R =

   5   1   0   3
   2   2   0   0
   1   0   3   0
   0   0   3   4
   4   4   0   4

ans =

   4.9963   0.9966   1.5487   3.0093
   2.0003   2.0013   2.1451   2.0023
   1.0012   3.0229   2.9975   2.0083
   5.4960   2.4633   3.0047   3.9907
   3.9993   3.9998   4.2874   4.0026
```

Notice how the existing ratings are almost perfectly reconstructed. Unfortunately, this is not all good news. The results are quite significantly dependent on the initial random matrices. Moreover, they also depend on the value of K. E.g. a simulation with $K = 3$ gives the following output:

```
ans =

   5.0037   1.0026   1.6216   2.9956
   2.0003   2.0026   1.9175   1.9919
   1.0048   3.4123   2.9948   2.1507
   4.3998   3.3513   3.0064   3.9890
   3.9920   3.9941   3.7041   4.0156
```

It seems K must be chosen carefully. Too small and there are not enough categories, and users and films are too coarsely correlated; too big and there are too many and too many combinations of solutions appear.

In addition to the choice of K we probably must average over many simulations.

12.6 Improvements to the scheme

12.6.1 Regularization

In order to reduce overfitting of data, a regularization method can be implemented. Here, we write

$$E = \sum_{(i,j)\in T} \mathsf{E}_{ij}^2 + \beta \left(\|\mathsf{P}\|^2 + \|\mathsf{Q}\|^2 \right)$$

where $\|\mathsf{P}\|^2 = \sum_{i,k} \mathsf{P}_{ik}^2$ represents a matrix norm and β is a small real number. The effect of the additional term is to find solutions in which P and Q are also minimised.

The upshot is that the iterative scheme (12.3) can be replaced by

$$\mathsf{P} \to \mathsf{P} + 2\alpha(\mathsf{E}\mathsf{Q} - \beta\mathsf{P}), \qquad \mathsf{Q} \to \mathsf{Q} + 2\alpha(\mathsf{E}^T\mathsf{P} - \beta\mathsf{Q}).$$

In practice we use $\beta \approx 0.01$.

12.6.2 Isolating bias

The idea here is to subtract user and film averages from the matrix R so that all that is left to analyse is bias.

Thus, we first need to compute and subtract the average entry from all entries to R. Next we need to compute and remove the average from entries in each row and each column. All that is left is user and film bias. Now we apply the matrix factorisation method to infer the missing bias in the matrix. Finally we add back in the subtracted row, column and matrix averages.

12.6.3 Updated code

Both improvements are implemented in the code online called `netflix2.m` The solutions are much more robust and less prone to variations in each simulation.

13 Solutions to Problems

13.1 Solutions 1

1. Simple, e.g.

   ```
   >> 2.3+1.2
   ans =
       3.5000
   ```

   and change + for -,*,/

2. The output is, in order, `2.6000,1,7,3.8,5.6667,0.1333,3.3333`

   Rule: Division takes priority over multiplication, which takes priority over addition and subtraction.

3. The answers are `0 + 1.0000i, -1, 5, -1.0000 + 0.0000i, 0.2079`

4. `format short` we get `0.2079`, using `format shortE`, we get `+2.0788e-01`, then `long` gives `0.207879576350762` and `longE` gives `2.078795763507619e-01`.

   Notice that we have implied that $i^i = e^{-\pi/2}$. You can prove this ultra geeky result as follows:

   $$x = i^i \quad \Rightarrow \quad \ln x = i \ln i \quad \Rightarrow \quad \ln x = i \ln e^{i\pi/2} \quad \Rightarrow \quad \ln x = i \cdot i\pi/2 \quad \Rightarrow \quad \ln x = -\tfrac{1}{2}\pi.$$

5. E.g.

   ```
   >> cos(pi)
   ans =
       -1
   ```

   and the other answers are `1.2256e-16, -1.2246e-16, -8.165619676597685e+15, 3.`

   Note: The answer to `cot(pi)` is wrong; 10^{15} is a big number, but it's not infinity. If you try » `cot(0)` the Matlab output is `Inf`, an error message similar to what you would expect on a calculator. Similarly, the results of sin(`pi`) and tan(`pi`) are not identically zero. Why ? Because the exact value of π, which is irrational, cannot be stored by a computer – the computer stores the first 16 decimal places. Which means that the evaluation of functions of π are also approximate.

6. The values after each command are: `a = 1, b = 2, c = 1, a = 2, b = 1, c = -1, a = 1, b = -1, c = -2`. At the end, `a = 1, b = -1, c = -2`

7. Using `format long` here.

```
>> s = 1;
>> s = sqrt(1+s)
s =
   1.414213562373095
>> s = sqrt(1+s)
s =
   1.553773974030037
>> s = sqrt(1+s)
s =
   1.598053182478617
>> s = sqrt(1+s)
s =
   1.611847754125252
```

If we assume s tends to a limit then $s = \sqrt{1+s}$ and so $s^2 - s - 1 = 0$ which gives $s = \frac{1}{2}(1 \pm \sqrt{5})$ and the plus root is 1.618033988.

8. The output from the commands are all arrays of size 3: `3 4 3`, `1.0000 0.5000 0.3333`, `0.3333 0.5000 1.0000`

   The command » `cos(a).*cos(b)` gives the output `-0.5349 0.1732 -0.5349`. It's an array in which the first entry is the product of the cosines of the first two elements of the arrays `a` and `b`. And so on. The other formulae produce the same result because they are either mathematically equivalent (by trig identities) or equivalent in terms of syntax. For e.g. `3:-1:1` produces the array `[3,2,1]` which is the same as `b`.

9. Here you type in Matlab, for e.g. (the square brackets are optional)

```
>> x = [-2:0.02:2];
>> y = sinh(x);
>> y1 = cosh(x);
>> y2 = tanh(x);
>> plot (x,y,x,y1,x,y2)
```

   Note: use semi-colons to suppress output ! You get the graph in figure 13.1(a)

10. Here, you follow as above with

```
>> x = [0:pi/100:2*pi];
>> y = sin(x).^2;
>> y1 = sin(x.^2);
```

   Why ? Well `x` is an array and you want to produce an array of points `y` which are opreations on each element of `x`. Therefore you need to use the `.` in front of each standard command.

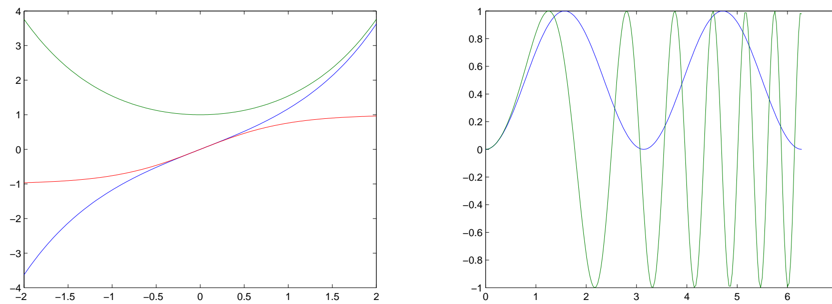


Figure 13.1: On the left, output from Q9 and on the right from Q10

13.2 Solutions 2

1. Just copy the code into a script and the output should be `thetaab = 117.2796`, `thetabc = 26.3843`, `thetaca = 36.3361`.

   You can easily produce spurious results by using values of `a,b,c` that cannot belong to a triangle. E.g. $a > b + c$. The Matlab script gives you complex values.

2. (a) Matlab script:

```
s = 1;
for n=1:5
    s = 2*(s+(1/s^2))/3
end
```

   (b) The results are

```
s =
   1.333333333333333
s =
   1.263888888888889
s =
   1.259933493449977
s =
   1.259921050017770
s =
   1.259921049894873
```

   (c) The value of s converges to all displayed decimal places after 6 iterations. The exact value is 1.259921049894873.

   (d) Assuming $s_n \to S$ then $s_{n+1} \to S$ also and $S = \frac{2}{3}(S + 1/S^2)$ reduces to $S = 2^{1/3}$.

3. (a) Matlab script

```
s = 1;
```

```
for n=1:5
    s = s + cot(s)
end
```

(b) Iterates s_1 to s_5 are

```
1.642092615934331
1.570675277161251
1.570796326795488
1.570796326794897
1.570796326794897
```

(c) Clearly converged before the 5th iteration.

(d) Assume $s_n \to S$, then $S = S + \cot(S)$ so $\cot(S) = 0$ so $S = \frac{1}{2}\pi + n\pi$. The particular value to which s_n converge depends on the initial value s_0. [*In fact this is Newton's root finding method applied to the function* $\cos(x)$ *– see Week 4*].

4. This is the automated version of the example we did manually on Worksheet 1.

(a) The script is

```
s = 1;
for n=1:5
    s = sqrt(1+s)
end
```

(b) The first 5 iterates are

```
1.414213562373095
1.553773974030037
1.598053182478617
1.611847754125252
1.616121206508117
```

(c) Running it for more iterates, it eventually converges to 1.618033988749895.

(d) Assuming $s_n \to S$ we have $S = \sqrt{1-S}$ so $S^2 - S - 1 = 0$ and $S = \frac{1}{2}(1 \pm \sqrt{5})$. We tend to the positive value of S because s_0 is positive, then so must all subsequent iterates.

5. (a) The script is

```
x = 1;
s = 1;
t = x;
for n=2:10
    u = 2*x*t - s
    s = t;
    t = u;
end
```

(b) With $x = 1$, the recurrence relation is $T_{n+2} = 2T_{n+1} - T_n$ with $T_0 = T_1 = 1$ and it's evident that $T_n = 1$ satisfies this for all n.

(c) Likewise, $x = 0$ means $T_{n+2} = -T_n$ with $T_0 = 1$ and $T_1 = 0$, so $T_n = 0$ for n odd and $T_{2n} = (-1)^n$, alternate.

[*In computing, it is always useful to find special cases which you know the answer to and against which you can check your code.*]

(d) For $x = 0.4$ the iterates are bounded between -1 and 1 and for $x = 1.4$ they are diverging towards infinity.

[*The recurrence relation in this question defines values of the Chebychev polynomials, when* $-1 \le x \le 1$, *explicitly defined by* $T_n(x) = \cos(n \cos^{-1}(x))$. *For* $|x| > 1$, *the definition can be extended using* cosh *in place of* cos, *but these grow as n increases.*]

6. (a) The script is

```
d = 2;          % change value of d_0 to 2
for n=1:10
    d = (2/d)*(sqrt(d^2+4)-2);  % change iterative step
    circ = d*(2^(n+1))
end
```

(b) The results are as follows from d_1 to d_{10}.

```
3.313708498984761
3.182597878074529
3.151724907429259
3.144118385245867
3.142223629942345
3.141750369169704
3.141632080702249
3.141602510241972
3.141595117718365
3.141593269631745
```

Clearly coming at π from above. Relatively slow ...

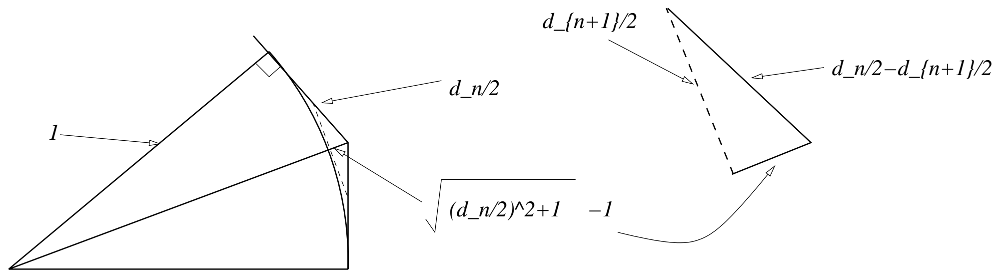
(c) When $n = 0$ and we have a square the length of the side of the square is $d_0 = 2$ and the perimeter is $8 > 2\pi$.

Now if we make an octagon, we can use simple geometry to show that $d_1 = 2(\sqrt{2} - 1)$ and the perimeter is thus $16(\sqrt{2} - 1) > 2\pi$.

The general step from a 2^{n+2}-sided polygon of side d_n to a 2^{n+3}-sided polygon of side d_{n+1} essentially follows the step from square to octagon. The sketch below outlines the geometrical arguments we infer that

$$\left(\frac{d_n}{2} - \frac{d_{n+1}}{2}\right)^2 = \left(\frac{d_{n+1}}{2}\right)^2 + \left(\sqrt{\left(\frac{d_n}{2}\right)^2 + 1} - 1\right)^2$$

and this is rearranged to get d_{n+1} in terms of d_n as given.



7. Here's a Matlab script

```
%%pirat.m
s = 1;
n = 10;
for j=0:n-1
  s = (1/(n-j))+(1/s);
end
s = 2+(2/s)
```

where s takes the value of s_0 originally, we set the truncation size so the code knows what it is. Then we loop from $j = 0$ through to $j = n - 1$. At each step we calculate the next term in the sequence and overwrite the old value.

With $n = 10$ we get $\pi \approx 3.14513$, with $n = 100$ it's 3.14163. So slow convergence.

13.3 Solutions 3

1. Changing k in the notes to k^3, the Matlab script could look like this:

```
n = 10;           % setting n to 10 here, so it's easy to change
s = 1;            % store the running total of the sum
for k=2:n
    s = s + k^3;  % k is both a counter and being used in the loop
end
s                 % The only output to screen
```

Change the value of `n` to check it equals the given formula. With `n=10` the answer is 3025.

Remark 1: It's an odd thing that

$$\sum_{j=1}^{n} j^3 = \left(\sum_{j=1}^{n} j \right)^2$$

Remark 2: In Matlab there are shortcuts using useful in-built functions. Here you could have written

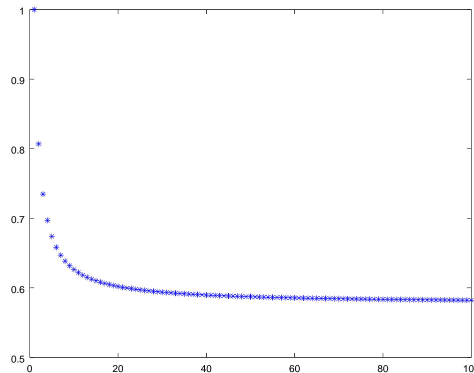
```
n = 10;
x = 1:n;
s = sum(x.^3)
```

However, I think it's easiest to stick to the long-hand methods – less to remember and it's more explicit.

2. (a) An elementary modification of the script in the notes:

```
n = 100;
s = zeros(1,n);
s(1) = 1;
for k=2:n
    s(k) = s(k-1)+(1/k);    % used to be (1/k^2)
end
x = 1:n;
plot(x,s-log(x),'*')        % used to be s-pi^2/6
```

(b) The curve appears to tend to a number just below 0.6.



(c) As in the notes, can prove this graphically by drawing n rectangles of unit width side by side starting at $x = 1$ and of decreasing heights 1, 1/2, 1/3 and so on. Then s_n is the area under the sum of the rectangles and this area can be bounded below by the area under the curve $1/x$ from $x = 1$ to $x = n + 1$ and above by the 1 plus the area under the curve $1/(x - 1)$ from $x = 2$ to $x = n + 1$. I.e.

$$\int_1^{n+1} \frac{1}{x}\,dx < s_n < 1 + \int_2^{n+1} \frac{1}{x-1}\,dx$$

which gives $\ln(n+1) < s_n < 1 + \ln(n)$. Substracting $\ln(n)$ from both sides and noting that $\ln(n+1) - \ln(n) \to 0$ as $n \to \infty$ we conclude that $\gamma = \lim_{n\to\infty}\{s_n - \ln(n)\} \in (0, 1)$.

Remark 1: In fact $\gamma = 0.57721566490...$ is called the Euler-Mascheroni constant. It's quite an interesting number:

`http://en.wikipedia.org/wiki/Euler-Mascheroni_constant`

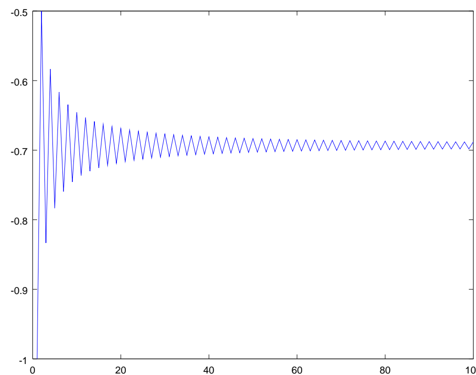
Remark 2: You can prove the series is divergent by writing out the series in the following way

$$\begin{aligned}\sum_{n=1}^{\infty} \frac{1}{n} &= 1 + \underbrace{\frac{1}{2}}_{\geq\frac{1}{2}} + \underbrace{\frac{1}{3} + \frac{1}{4}}_{\geq\frac{1}{2}} + \underbrace{\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}}_{\geq\frac{1}{2}} + \ldots \\ &\geq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \ldots\end{aligned}$$

3. (a) Similar to Exercise 2. The modified Matlab script is

```
n = 100;
s = zeros(1,n);
s(1) =-1;                   % s_1 = -1 instead of +1
for k=2:n
    s(k) = s(k-1)+((-1)^k/k);   % (-1)^k/k instead of 1/k
end
x = 1:n;
plot(x,s,'-')               % nicer to use a solid line
```

(b) Yes, the values of the iterates seem to be oscillating towards -0.7. In fact it's easy to deduce $-1 < S < -\frac{1}{2}$.



(c) MacLaurin expansion of $\ln(1+x)$ is

$$x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots$$

(you should be able to do this, right ?)

Using $x = 1$ we see that $-\ln(2) = -1 + \frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \ldots$ which is S. So the limit of the sum is $-\ln(2) = -0.69315$.

4. (a) Matlab script is

```
function s = myfact1(n)
s = 1;          % s_1
for k=2:n
    s = s*k;    % iterating... use ; to suppress output
end
end
```

(b) Modification to storage in arrays.

```
function s = myfact1(n)
s = zeros(1,n);         % needs to be size n to fit s_1 to s_n
s(1) = 1;               % s_1... need to shift indices forward by one !
```

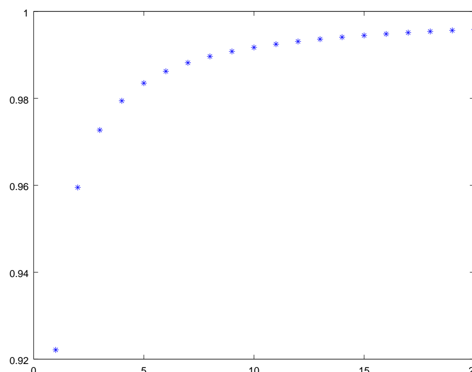
```
for k=2:n
    s(k) = k*s(k-1);    % iterating and storing
end
end
```

(c) Add the following lines between `end` and `end`

```
x = 1:n;
t = ((2*pi*x).^(1/2)).*((x./exp(1)).^x);
plot(x,t./s,'*')
```

Note: The array `t` is defined by operations on the array `x` and therefore uses element-by-element operators `.*`, `./` etc...

Running this with $n = 20$ produces a graph suggesting that $t_k/s_k \to 1$ as $k \to \infty$.



(d) Using the formula, which can be established by bounding the area occupied by rectangles of unit width and height $\ln(i)$, $i = 1, \ldots, n$, we get

$$[x\ln(x) - x]_1^n < \ln(1) + \ln(2) + \ldots + \ln(n) < [x\ln(x) - x]_1^{n+1}$$

and so

$$n \ln n - n < \ln(n!) < (n+1)\ln(n+1) - n$$

which means that

$$\ln(n^n) - \ln(\mathrm{e}^n) < \ln(n!) < \ln((n+1)^{n+1}) - \ln(\mathrm{e}^n)$$

and so

$$\left(\frac{n}{\mathrm{e}}\right)^n < n! < \mathrm{e}\left(\frac{n+1}{\mathrm{e}}\right)^{n+1}$$

It's clear $t_n > (n/\mathrm{e})^n$ and $t_n < (n+1)^{n+1}/\mathrm{e}^n$ for large n so this fits with the numerical observations.

Remark: The approximation t_n to $n!$ called Stirling's formula.

5. Simple to code as

```
function s = mybicoeff(n,m)
s = factorial(n)/(factorial(m)*factorial(n-m));
end
```

6. This is a hard example, requiring nested loops. It's all about how you organise the triangle in an array and overwrite one line with the next. There are two methods:

```
function s = pascal(n,m)
c = zeros(1,n+1);
c(1) = 1;
for k=1:n
    for j=n:-1:1
        c(j+1) = c(j+1)+c(j);
    end
end
s = c(m+1);
end
```

in which the inner loop goes backwards in steps of 1 from n to 1 or

```
function s = pascal2(n,m)
c = zeros(1,n+1);
c(n+1) = 1;
for k=1:n
    for j=n+1-k:n
        c(j) = c(j)+c(j+1);
    end
end
s = c(m+1);
end
```

Call with `pascal1(5,2)` for e.g.

13.4 Solutions 4

1. The Matlab script could look like this:

```
function s = mysinc(x)
if abs(x) > 0            % or x ~= 0
    s = sin(x)/x;
else
    s = 1;
end
end
```

or

```
function s = mysinc(x)
if x == 0
    s = 1;
else
    s = sin(x)/x;
end
end
```

$\sin(x)/x$ does not define a function at $x = 0$.

2. (a) This function script will do it. Not completely trivial.

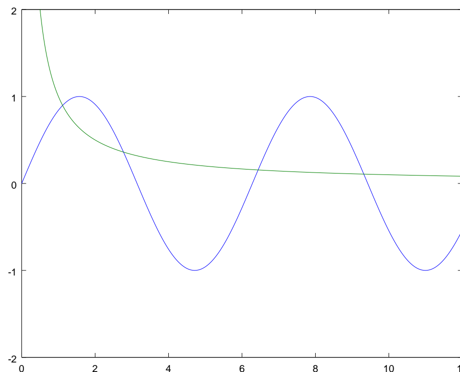
```
function s = squarewave(x)
if mod(x,2*pi) < pi
    s = 1;
else
    s = 0;
end
end
```

3. (a) E.g. `» [root,n] = bisection(f,0.1,2,1e-6)` gives `root = 0.567143` and `n = 19` (the number of iterations taken.)

   (b) $\sin(x) = x$ satisfied by $x = 0$. Clear that $\sin(x)/x \leq 1$ for $x \neq 0$ so there are no other roots.

   (c) Reminder on how to plot. Type

```
>> x = [0:0.05:12];
>> y1 = sin(x);
>> y2 = 1./x;
>> plot(x,y1,x,y2)
>> axis([0 12 -2 2])
```



(i) The points of intersection of the curves $\sin(x)$ and $1/x$ correspond to roots of $\sin(x) = 1/x$ or $x\sin(x) = 1$. There are clearly going to be an infinite number and $x_n^* \to n\pi$ as $n \to \infty$.

Remark: You can do even better by considering $x_n^* = n\pi + \epsilon_n$, where $|\epsilon_n| \ll 1$ (meaning the size of ϵ_n is much smaller than 1). Susbstitute this into the relation to give $\epsilon_n \approx (-1)^n/(n\pi)$ after arguing that $\epsilon_n^2 \ll |\epsilon_n|$. Thus, a more accurate version of the asymptotic relation is $x_n^* \approx n\pi + (-1)^n/(n\pi)$ as $n \to \infty$.

(ii) Use the graph to bracket roots. First 3 roots are 1.114156, 2.772603, 6.439115.

4. (a) Here's a Matlab script which will do the job

```
function [y,n] = newton(f,fd,x,nmax,tol)
for n=1:nmax
    y = x-f(x)/fd(x);     % cannot simply overwrite a as need to
    if abs(x-y) < tol     % check if the distance between two successive
                          % iterates is less than tol. If so, return.
        return
    end
    x = y;
end
disp('The tolerance has not been reached within max iterations')
end
```

(b) (i) Set up the function call with » `f = @(x)(exp(x)-(1/x));` and » `fd = @(x)(exp(x)+(1/x^2);` and call » `[root, n] = newton(f,fd,1,100,1e-6)`. This returns `root = 0.567143` in `n=4` iterations.

Remark: Newton is fast: compare with 3(a), where bisection took 19 iterations.

(ii) This equation only has the root $x = 0$ (see Q3(b)). The output of the newton method is `root = 1.4e-06` in `n=33` iterations.

Here Newton is slow, as $\sin(x) - x \sim -x^3/6$ when x is small.

(c) This follows the analysis in the notes.

E.g. Choose $x_0 = 1$, define f and f' and » `[root, n] = newton(f,fd,1,100,1e-6)` gives `root = 0` in `n=6` iterations. But change x_0 to 1.1 and the code returns the warning message "`The tolerance has not been reached within max iterations`" and `root = NaN` !!

There is a critical value of x_0 somewhere between 1 and 1.1.

(d) We define `f = @(x)(sinh(2*x)-4*x);` and `fd = @(x)(2*cosh(2*x)-4);` and then calling » `[root, n] = newton(f,fd,1,100,1e-6)` gives `root = 1.088659` in `n=4` iterations.

We showed in the notes that if x_0 was such that $\sinh 2x_0 > 4x_0$ then Newton's method applied to $f(x) = \tanh(x)$ would diverge and that if $\sinh 2x_0 < 4x_0$ it would converge. Thus, the critical value corresponds to the solution of $\sinh 2x = 4x$, which we have found to be 1.088, lying between 1 and 1.1, as observed in part (c).

5. (a) We've done two-term recurrence relations before (e.g. Fibonacci in Week 2). So follow that method, and also the notes in Week 4 on the bisection method concerning the output of more than one variable from a function. You will end up with something like this:

```
function [b,n] = secant(f,a,b,nmax,tol)
for n=1:nmax
    c = (a*f(b)-b*f(a))/(f(b)-f(a));
    a = b;
    b = c;
    if abs(a-b) < tol
        return
    end
end
disp('The tolerance has not been reached within the max iterations')
end
```

(or you could use arrays if so inclined).

(b) Testing the code. E.g.

```
>> f = @(x)(x^2 -2);
>> [root,n] = secant(f,1,2,100,1e-6)
root =  1.41421356237310
n =  6
```

(c) If you choose x_0 and x_1 so that they bracket a root, there's no way the secant method will fail to converge. But if x_0 and x_1 are not close enough to a root, you have the same potential problems as Newton's method. E.g. try finding roots of the function $\tanh(x)$ (as in the notes),

```
>> [root,n] = secant(f,1.2,2.1,100,1e-6)
warning: division by zero
```

and things are evidently pear-shaped.

(e) The equation follows easily from the secant method applied to $f(x) = x^2 - 2$.

Firstly, the two methods are different in that one requires x_0 only and the other x_0 and x_1. So to make things fair, we should use a x_1 in the secant method which is generated from the one-term method.

Apart from that we can confirm that the second (one-term) relation is actually Newton's method applied to $f(x) = x^2 - 2$. The rule of thumb is that Newton's method is better than secant, and if you look up rates of convergence you find that the order of convergence of Newton is 2 and secant is $\frac{1}{2}(1 + \sqrt{5}) \approx 1.61$ and so Newton wins. (If at one step, you are a distance ϵ away from the root, at the next step you'd be a distance ϵ^α where α is the order. Thus larger values of α implies faster convergence).

6. This is known as a 'bubble sort' code. You loop from the first member of the array to last swapping elements of the array in pairs when one is larger than the next. To ensure that the smallest number at the bottom of the array has a chance to rise to the top of the array you need to run the pair-swapping loop $n-1$ times where n is the length of the array.

   Here's code that does it:

```
%% Bubble sort code
function a = bubble(a)  % input: a is an array, output modified array a
n = length(a);          % n is the length of the array
for j=1:n-1
    for k=1:n-1
        if a(k) > a(k+1)    % if local pair ordered wrong
            tmp = a(k+1);   % swap the local pair.
            a(k+1) = a(k);
            a(k) = tmp;
        end
    end
end
end
```

   call this with, for e.g. `» a = bubble([10, 4, 6, 12, 1])`.

13.5 Solutions 5

1. (a) `» tanh(800)` returns a value of 1. `» sinh(800)/cosh(800)` returns `NaN` (Not a Number). This is because both sinh(800) and cosh(800) are too big for the computer to store, and Matlab assigns `Inf` to their values.

   (b) Use $\tanh(x) = \dfrac{\exp^x - \exp^{-x}}{\exp^x + \exp^{-x}} = \dfrac{1-\exp^{-2x}}{1+\exp^{-2x}}$.

   Computing this definition works because $\exp^{-1600}$ is stored by Matlab as zero.

   Remark: The definition in (b) would fail if $x = -800$. So you could not use the definition universally; you would have to replace it by $(\exp^{2x} - 1)/(\exp^{2x} + 1)$ if $x < 0$.

2. (a) The argument goes that $\cosh(j+1/j) > \cosh(j)$ and so $\dfrac{\sinh(j)}{\cosh(j+1/j)} < \tanh(j) < 1$. It follows that

$$S = \sum_{j=1}^{\infty} \frac{\sinh(j)}{j^2 \cosh(j+1/j)} < \sum_{j=1}^{\infty} \frac{1}{j^2} = \frac{\pi^2}{6}. \tag{13.1}$$

   Hence S is convergent.

   (b) Here's the code

```
function s = series1(n)
s = 0;
for j=1:n
    s = s+sinh(j)/(j^2*cosh(j+(1/j)));
end
end
```

(c,d) For $n = 100, 200, 400, 800$ we get `0.77691, 0.78184, 0.78432, NaN`. Converging (slowly) until 800. Reason explained in answer to Q1.

(e) We use the definitions of cosh and sinh in terms of exponentials to write

$$\frac{\sinh(j)}{\cosh(j+1/j)} = \frac{1-\exp^{-2j}}{\exp^{1/j}+\exp^{-2j-1/j}} \tag{13.2}$$

and note $\exp^{\pm 1/j} \to 1$ as $j \to \infty$. The code for series2.m simply uses this relation to replace the LHS with the RHS.

For $n = 800, 1600$ we get `0.78556, 0.78619`.

(f) [HARD] Noting that (13.2) tends to 1 as $j \to \infty$, we modify the series by writing

$$s_n = \sum_{j=1}^{n} \left(\frac{\sinh(j)}{j^2 \cosh(j+1/j)} - \frac{1}{j^2} \right) + \frac{\pi^2}{6} \tag{13.3}$$

Then $s_n \to S$ as before, but, whereas the terms in (13.1) decay like $1/j^2$, the terms in the sum above are

$$\frac{1}{j^2}\left(\frac{1-\exp^{-2j}}{\exp^{1/j}+\exp^{-2j-1/j}} - 1\right) \sim \frac{1-\exp^{-2j} - (1+j^{-1}+\frac{1}{2}j^{-2}+\ldots) - \exp^{-2j}(1-j^{-1}+\frac{1}{2}j^{-2}+\ldots)}{\exp^{1/j}+\exp^{-2j-1/j}}$$

after expanding $\exp^{1/j}$ and $\exp^{-1/j}$ in a MacLaurin series and retaining terms up to $1/j^2$ (i.e. ignoring $1/j^p$ for $p \geq 3$ as they are much smaller than $1/j^2$). Now the RHS of the above is like

$$\frac{j^{-2} - 2\exp^{-2j}}{\exp^{1/j}+\exp^{-2j-1/j}} \sim \frac{1}{j^2}$$

as $j \to \infty$ (to leading order). With the $1/j^2$ already present in the series, this means that the terms in the series in (13.3) decay like $1/j^4$, much faster than $1/j^2$.

Results of implementing the revised scheme in our code: $n = 800, 1600$ are `0.78681`, `0.78681`, so converged to 5-d.p.'s after 800 terms.

3. (a) Here's a script, just like back in Week 2:

```
n = 20;
s = 1/sqrt(3);
for j=1:n
    s = (sqrt(s^2+1)-1)/s;
    approxpi = 6*2^j*s
end
```

(b) Looks like it's converging until you get to about the 13th iterate. Then the numbers start to 'wobble'. Iterates 18, 19 and 20 are `3.14159267`, `3.14175586` and `3.14159267`.

(c) Arh ! The problem is that $s_n \to 0$ and so the numerator $\sqrt{1+s_n}-1$ and the denominator s_n in the recurrence relation both tend to zero. And this causes problems. Instead, multiply the relation in the question top and bottom by $\sqrt{s_n^2+1}+1$ to get

$$s_{n+1} = \frac{(\sqrt{s_n^2+1}-1)(\sqrt{s_n^2+1}+1)}{s_n(\sqrt{s_n^2+1}+1)} = \frac{s_n}{\sqrt{s_n^2+1}+1}$$

Now s_n still tends to zero, but the relation becomes approximately $s_{n+1} = \frac{1}{2}s_n$ as $n \to \infty$, which is well-behaved (i.e. not dividing small numbers by small numbers).

4. If you try the `integral` command you get an error message. The evaluation of the integral is nonsense because there is a singularity in the function $1/x$ at zero and you cannot ignore this. There *is* a version of this integral which is correct in which you define the integral to be a *Cauchy principal-value* integral. But you have to know why you have decided to use such an integral.

5. (a) Here's some code. Bit trickier than exponential because you have to jump over even powers of x here, and switch signs at each step.

```
function s = mycos(x,n)
s = 1;
t = 1;
for j=1:n
    t =-t*x*x/((2*j)*(2*j-1));
    s = s+t;
end
end
```

(b) Testing... e.g. `>> mycos(pi/2,20)` returns `4.2648e-17` which seems pretty accurate. However `mycos(31*pi/2,200)` or `mycos(31*pi/2,2000)` both give `-8892` which is a long way from zero.

The difficulty here is the alternating sign in the series. So when large positive and negative numbers are being added together to get small numbers then the computer just cannot store the number of decimal places needed for this to be done accurately.

OK. So the point is that, in reality, you wouldn't try and do what the question asked you to do. Instead you would Taylor expand about the point $2n\pi$, say, closest to the value of x you wanted to compute.

6. (a) $C_0 = 1$, $C_1 = 1$ obviously. Then from the definition

$$C_n = \frac{(n+2)(n+3)\dots(n+n)}{1.2.\dots n} = \prod_{k=2}^{n} \frac{(n+k)}{k} = \prod_{k=2}^{n} \left(1+\frac{n}{k}\right)$$

(b) The representation above is better computationally as it avoids dividing large numbers by large numbers when n is large.

(c) Here's some code

```
function c = catalan(n)
c = 1;
for k=2:n
    c = c*(k+n)/k;
end
```

The Catalan numbers C_2 to C_{10} are 2, 5, 14, 42, 132, 429, 1430, 4862, 16796. (There are better ways of computing them.)

7. Here's the Matlab code:

```
function s = sph(x,n)
s = zeros(n+1,1);
s(1) = sin(x)/x;
s(2) = (sin(x)-x*cos(x))/x^2;
for j=1:n-1
    s(j+2) = ((2*j+1)*s(j+1)/x)-s(j);
end
end
```

and the output is

```
0.985067355537799
0.099102888040642
0.005961524868621
0.000255859769704
0.000008536424483
0.000000232964777
0.000000005617358
0.000000010454061
0.000000517085671
0.000029291067308
0.001854583843824
```

This is a well-known phenomena associated with certain types of recurrence relation which can be shown to be exponentially unstable. So errors propagate rapidly, even from the first iterative step. It turns out that such relations have to be computed by running the recurrence relation in reverse !

13.6 Solutions 6

1. Lots of code to look at and understand.

2. (a) Returns a random integer between 1 and 6. I.e. simulation the throwing of a dice.

   (b) Here's the complete code

```
function dice(m)                    % input

count = zeros(12,1);                % array for logging scores

for k=1:m                           % m simulations
    d1 = ceil(rand*6);              % dice1 random score from 1 to 6
    d2 = ceil(rand*6);              % dice2    "     "     "  "  "  "
    count(d1+d2) = count(d1+d2)+1; % add one to the score counter
end

count(:) = count(:)/m;              % normalise over the simulations

x = 1:12;                           % x axis for plotting
count2 = [0,1,2,3,4,5,6,5,4,3,2,1]/36;  % array of expected results
plot(x,count,'*',x,count2,'-')      % plot

end
```

   (c) The output will look something like the left-hand panel in fig. 13.2

   (d) See the website for the code `dice2.m` in which changes to the code in part (b) are made to produce a bar graph (right-hand panel in fig. 13.2). In the revised code, we use a single array `count` with two columns, one for the simulated results and the other for the expected results. This allows us to use `bar`.

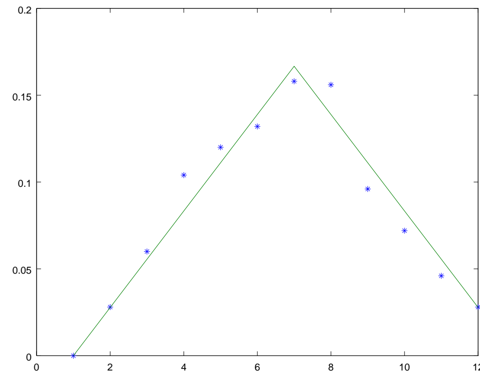
3. (a) Here's a script, which is a modified version of `rwalk2d.m` in which we have stripped out the movement in the y-direction.

```
function rwalk1d(n,m)  % input: n = # steps, m = # simulations

d2av = zeros(n,1);     % Set up an array of length n

for k=1:m              % k counts the walks
    x = 0;
    for j=1:n          % walking
        p = rand*2;    % p is random number between 0 and 2
        if p > 1
            x = x+1;   % right
        else
            x = x-1;   % left
        end

        d2av(j) = d2av(j) + x^2;   % update average D^2
```



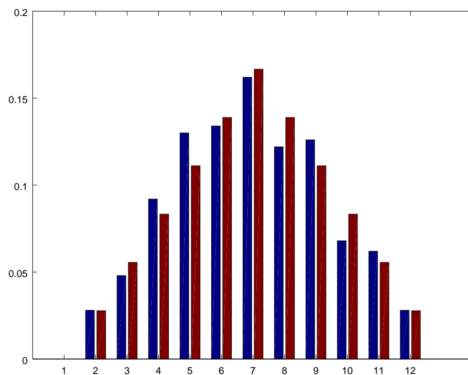


Figure 13.2: Output from Q2(c,d). Two visualisations of simulations versus expected results for the throwing of two dice.

```
    end
end

d2av(:) = d2av(:)/m;   % normalise array by the number of walks

time = 1:n;            % Set up an array for x-axis for plotting.
plot (time,d2av,'b-','LineWidth',2)
```

```
end
```

(b,c) Following theory in the notes, expect the D^2 average to be linear with number of steps taken. Simulations look like they agree, see figure 13.3

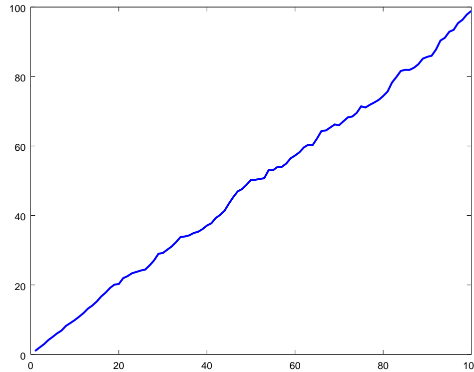


Figure 13.3: Output from Q3(c). D^2 average against steps taken.

4. The idea is that you strip off the movement in the y-direction and leave yourself with an animation which models a random walk in 1D. The only subtlety here is that you want to plot the walk in the vertical direction against the step number on the horizontal axis. See the code `animrwalk1d.m` on the web page to see how it ends up.

   Running `animrwalk1d(100,40)` gives fig. 13.4:

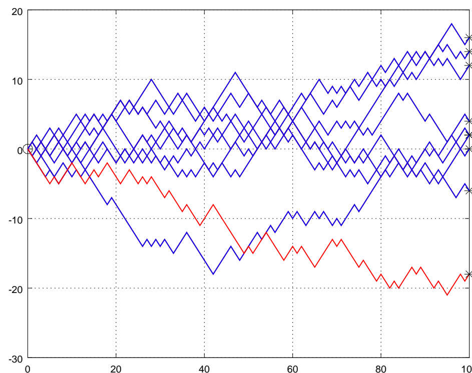


Figure 13.4: Output from Q4: Snapshot taken from random walk animation in 1D. Continue to see that the density profile becomes parabolic confirming that the vertical spread is like $\sqrt{n}$.

5. (a) Start with the formula

$$(1+x)^n = \binom{n}{0} + \binom{n}{1} x + \binom{n}{2} x^2 + \ldots + \binom{n}{n} x^n$$

and simply substitute $x = 1$ to get the result.

(b) Fairly easy to see that if you end up at location zero, you have gone left at every step and location 1 you have gone left at $n-1$ steps and right at 1 step. So in general at location j you have gone left $n-j$ times and right j times. So the probability of landing at j is

$$\frac{p^{n-j}(1-p)^j n!}{j!(n-j)!}.$$

The revised code galton2.m is on the course web page and calling `galton2(10,500,0.25)` gives fig. 13.5

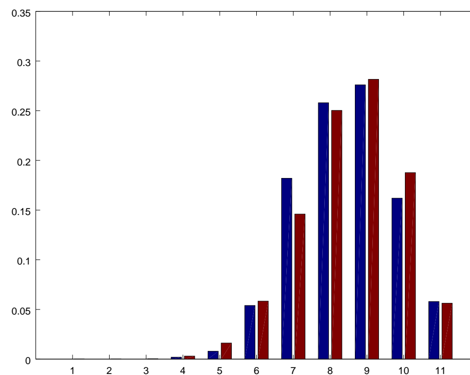


Figure 13.5: Output from Q5(b). Blue: simulated results, Red: expected results.

6. (a,b) Here we take out the conditional statements moving the walk either north, south east or west and replace with the much simpler random step:

```
p = rand*2*pi;  % p = random variable from 0 to 2*pi
x = x+cos(p);   % update x
y = y+sin(p);   % update y
```

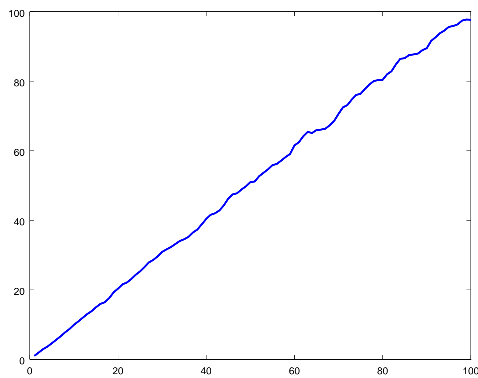
Everything else is the same. See the code rwalk2db.m and animrwalk2db.m on the course web page, in which you need to change the lines above to reflect the use of arrays in the animation code.

13.7 Solutions 8

1. For example (i) you can get the exact result by typing >> `erf(1)` and the answer is 0.842700792949715. The table below shows the convergence and error

   Example (ii): convergence and error tabulated in §8.2.3 of the notes.

   (b) The code produces the exact answer of 0.5 irrespective of the value of n. Why ? If you draw a graph of $f(t) = t$ and overlay the rectangle mid-point rule on it, you will see immediately that the area under $f(t)$ and the rectangles are the same, irrespective of the number of rectangles.



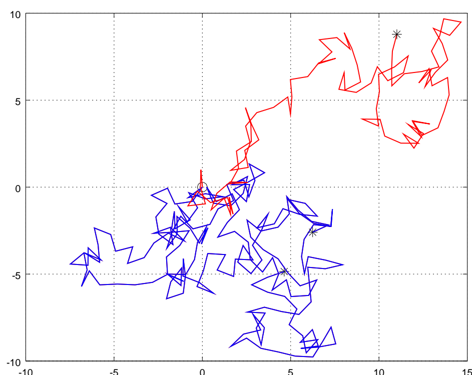


Figure 13.6: Output from Q6(a,b). Left panel is the D^2 average against time (or steps) and the right-hand panel is a snapshot taken from a simulation.

2. (a) Here's some code adapted from rectint.m

```
function s = trapint(f,a,b,n)   % function, input: f = function, a,b
                                % end points, n = number of trapezoids.
                                % output is s = approx to integral.
h = (b-a)/n;                    % the width of each trapezoid
s = 0.5*h*(f(a)+f(b));          % set running total
for j=1:n-1                     % loop over all rectangles and sum
```

n	I_{rect}	E
10	0.8430469175	3.4612e-04
20	0.8427872862	0.8649e-04
40	0.8427224139	0.2162e-04
80	0.8427061980	0.0540e-05

Table 13.1: Convergence of I_{rect} and error $E = I_{rect} - \text{erf}(1)$ with n. The error is approximately quartered as n is doubled.

```
    s = s+h*f(a+j*h);
end
end
```

(b) Results of using the trapezium rule on the function in Exercise 1(a)(ii) for different n shown in Table 13.2.

n	I_{trap}	E
10	0.706743261	3.6351e-04
20	0.707015908	0.9087e-04
40	0.707084063	0.2271e-04
80	0.707101101	0.0567e-04

Table 13.2: Convergence of I_{trap} and error $E = I_{trap} - 1/\sqrt{2}$ with n. The error is approximately quartered as n is doubled.

The trapezium rule appears to be converging with an error proportional to $1/n^2$.

Remark: It can be shown that the trapezium rule has errors which decay like $1/n^3$ as $n \to \infty$ *if* $f'(a) = f'(b)$. This is particularly useful if you are integrating periodic functions over a period.

3. (a) Here's the code for Simpson's Rule (it could be written a lot more efficiently)

```
function s = simpint(f,a,b,n)   % function, input: f = function, a,b
                                % end points, n = number of subintervals
                                % output is s = approx to integral.
h = (b-a)/n;                    % the width of each subinterval
s = (h/3)*(f(a)+f(b));          % running total
for j=1:n/2                     % loop over first sum
    s = s+(4*h/3)*f(a+(2*j-1)*h);
end
for j=1:n/2-1                   % loop over second sum
    s = s+(2*h/3)*f(a+2*j*h);
end
end
```

(b) Table 13.3 gives results from Simpson's rule with example (ii) from Exercise 1(a). The results indicate that the error is proportional to $1/n^4$.

n	I_{simp}	E
10	0.70710693077	1.495860e-07
20	0.70710679053	0.093439e-07
40	0.70710678177	0.005839e-07
80	0.70710678122	0.000364e-07

Table 13.3: Convergence of I_{simp} and error $E = I_{simp} - 1/\sqrt{2}$ with n. The ratio of two consecutive errors is approximately 16 indicating that that error is reduced by 1/16th as n is doubled.

4. There's different ways of dealing with improper integrals and this is just one. We make the substitution $t = u/(1-u)$ so that $dt = du/(1-u)^2$ and so the semi-infinite integral is mapped to

$$\int_0^1 f\left(\frac{u}{1-u}\right)\frac{1}{(1-u)^2}\,du$$

With $f(t) = 1/(1+t^2)$ the integrand in the above becomes

$$\frac{1}{(1+u^2/(1-u)^2)}\frac{1}{(1-u)^2} = \frac{1}{u^2+(1-u)^2}$$

In Matlab we write `» f = @(u)(1/(u^2 + (1-u)^2))` with `» rectint(f,0,1,100)`, say. Using this command get 1.570812993. The exact answer is

$$\int_0^\infty \frac{1}{1+t^2}dt = \tan^{-1}(\infty) = \pi/2 \approx 1.570796326$$

So it works pretty well.

5. (a) Output from Euler's method in fig. 13.8. Only small visual difference between different step sizes.

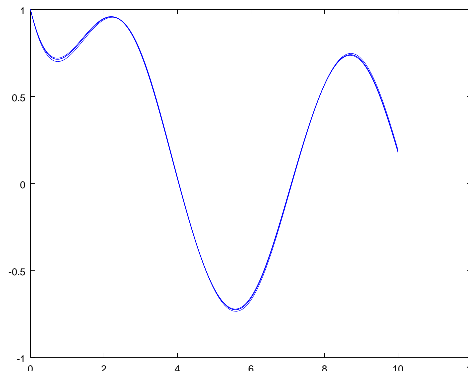


Figure 13.7: The solution over $0 < t < 10$ with $h = 0.08$, 0.04 and 0.02.

(b) To implement the Midpoint method (see rk2.m) we simply have to replace a line of the Euler code with

```
y(j+1) = y(j)+h*f(t(j)+0.5*h,y(j)+0.5*h*f(t(j),y(j)));
```

You get a solution curve like those shown in fig. 13.8.

Remark: The Midpoint method is generally more accurate than Euler's method, though we haven't tested this here.

6. The problem is that $\ln(\sin x) \to -\infty$ as $x \to 0$. We could use brute force and use the rectangle mid-point, trapezium or Simpson's rule to find the numerical value. Two problems emerge. First, the integrand cannot be evaluated at $x = 0$, so trapezium/Simpson's will not work unless we replace the lower limit of zero with a very small number. Second, we are going to need a lot of subintervals (a very small value of h) to resolve the area under the integrand as it tends to minus infinity. E.g. with `» f = @(t)(log(sin(t))` we need `» simpint(f,1e-14,1,100000)` (that's 100,000 subintervals) to get an answer `-1.0567` accurate to 4 decimal places.

   The solution ? We note that the MacLaurin series gives $\sin(x) \approx x$ as $x \to 0$, and so $\ln(\sin(x)) \approx \ln(x)$ as $x \to 0$. Then we write

$$\begin{aligned}\int_0^1 \ln(\sin(x))\,dx &= \int_0^1 \ln(\sin(x)) - \ln(x)\,dx + \int_0^1 \ln(x)\,dx \\ &= \int_0^1 \ln(\sin(x)/x)\,dx + [x\ln(x) - x]_0^1 \\ &= \int_0^1 \ln(\sin(x)/x)\,dx - 1\end{aligned}$$

   and as $x \to 0$ the integrand tends to $\ln(1) = 0$. In Matlab `» f = @(t)(log(sin(t)/t))` with `» simpint(f,1e-14,1,10)-1` gives the answer `-1.05672`. That is, with just 10 subintervals we have got the same level of accuracy as the brute force approach.

13.8 Solutions 9

1. Can choose your own numbers, but here's my example of how it works:

```
>> a = [1 2 3];
>> b = [3 2 1];
>> c = [2 3 1];
>> cross(a,cross(b,c))
ans =
   13   -8    1
>> dot(a,c)*b-dot(a,b)*c
ans =
   13   -8    1
>> dot(a,cross(b,c))*a
ans =
   12   24   36
```

```
>> cross(cross(a,b),cross(a,c))
ans =
   12   24   36
```

2. So set matrix and RHS vector and then invert

```
>> A = [3 -1 4 ; 0 1 -1; 2 6 -1];
>> b = [0 ; 1 ; 1]
>> x = A\b
x =
   2.22222
  -0.88889
  -1.88889
```

3. (a) Here's the code... a bit long but there are no obvious short-cuts.

```
function r = vecmap(x,alpha,beta,gamma)
Rx = zeros(3,3);
Ry = zeros(3,3);
Rz = zeros(3,3);
Rx(1,1) = 1;
Rx(2,2) = cos(alpha);
Rx(2,3) = -sin(alpha);
Rx(3,3) = cos(alpha);
Rx(3,2) = sin(alpha);
Ry(2,2) = 1;
Ry(1,1) = cos(beta);
Ry(1,3) = sin(beta);
Ry(3,3) = cos(beta);
Ry(3,1) =-sin(beta);
Rz(3,3) = 1;
Rz(1,1) = cos(gamma);
Rz(1,2) =-sin(gamma);
Rz(2,2) = cos(gamma);
Rz(2,1) = sin(gamma);
r = Rz*Ry*Rx*x;
end
```

(b) Set `x = [1 ; 1 ; 1];` and call `r = vecmap(x,pi,pi,pi)` to find $\mathbf{r} = (1,1,1)^T$. So mapping leaves position vector unchanged. Easy to see geometrically that rotation by π around each axis will get you back to the starting position. Alternatively, it's easy to see from their definition that $\mathsf{R}_z(\pi)\mathsf{R}_y(\pi)\mathsf{R}_x(\pi) = \mathsf{I}$.

(c) The condition $\mathsf{R}^T = \mathsf{R}^{-1}$ is equivalent to $\mathsf{R}\mathsf{R}^T = \mathsf{I}$ and for each of the three matrices this is easy to confirm. Indeed, once you can done one of them the others are basically the same.

If we have $\mathbf{r} = \mathsf{R}_z(\gamma)\mathsf{R}_y(\beta)\mathsf{R}_x(\alpha)\mathbf{x}$ then it follows that

$$\mathbf{x} = \mathsf{R}_x^{-1}(\alpha)\mathsf{R}_y^{-1}(\beta)\mathsf{R}_z^{-1}(\gamma)\mathbf{r} = \mathsf{R}_x^T(\alpha)\mathsf{R}_y^T(\beta)\mathsf{R}_z^T(\gamma)\mathbf{r}$$

and this is the inverse mapping.

4. (a) Here's the code

```
A = rand(2,2);      % set A to be a random 2x2 matrix
d = eig(A)          % print eigenvalues of A
x = [1 ; 1];        % define x to be the column vector (1,1)^T
inv(eye(2)-A)*x     % print (I-A)^{-1}x
y = x;              % set y to first term the sequence, y_0
m = 200;            % set m
for j=1:m
  y = x + A*y;      % update y using the recurrence relation given
end
y                   % print the value of the series
```

(b) Run the script: E.g. of output when eigenvalues (d)less than unity in modulus:

```
d =

   0.923550826838525
  -0.811043090074474

ans =

   12.9566641630964
   13.2144570358461

y =

   12.9566626842223
   13.2144555262378
```

5. $\det(\mathsf{A}) = 0$ if and only if $\det(\mathsf{A} - 0\mathsf{I}) = 0$ and so there is a $\lambda = 0$.

6. (a) True. Do ≫ `det(A)` and ≫ `d = eig(A);` followed by ≫ `d(1)*d(2)*d(3)*d(4)`

   (b) True. ≫ `d = eig(A)` and ≫ `e = eig(A')` give the same answers.

   (c) True. ≫ `d = eig(A)` and ≫ `e = eig(inv(A));` followed by ≫ `1./e` to reciprocate the output.

   (d) False. ≫ `d = eig(A*B)` is not the same as ≫ `e = eig(A).*eig(B)`.

   (e) False. ≫ `d = eig(A+B)` is not the same as ≫ `e = eig(A)+eig(B)`.

7. This:

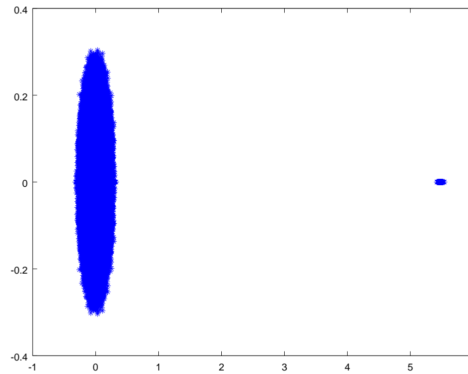


Figure 13.8: Output from the code given in Q7 with $n = 120$. Observe two distinct regions of eigenvalues – one occupying a circular domain of fixed radius approximately 0.3 and a set of real eigenvalues which vary in position with n.

13.9 Solutions 10

1. (a) Here's the code for the power method. Follow the algorithm almost word for word from the notes.

```
function [x,d] = power(A,kmax)  % input: A = matrix, kmax = max iterations
                                % output: x = eigenvector, d = eigenvalue
[n,m] = size(A);                % n = size of A, m = #of cols; not needed.
x = rand(n,1);                  % defining x0 to be a random col vector
x = x/norm(x);                  % make the vector into a unit vector
for k=1:kmax                    % iterate kmax times
    x = A*x;                    % x_{k+1} = A x_k
    x = x/norm(x);              % normalise x_{k+1}
end
d = x'*A*x;                     % eigenvalue is x^T A x
end
```

(b) Do `» A = rand(4,4);` followed by `» [V,d] = eig(A)` (until we get real values) gives an output something like this:

```
V =
  -0.447674  -0.247639  -0.359160   0.402170
  -0.441766  -0.425613   0.016426  -0.480301
  -0.573914   0.868542  -0.332106  -0.426586
  -0.524456  -0.056237   0.872032   0.652377
d =
   1.93222
  -0.30277
   0.58965
   0.38541
```

So the largest is the first, $d = 1.93222$ and the corresponding eigenvector is the first column of V or $\mathbf{v} = (-0.447674, -0.441766, -0.573914, -0.524456)^T$.

(c) Using the same A as in part (b) we type:

```
>> [e,f] = power(A,20)
e =
   0.44767
   0.44177
   0.57391
   0.52446

f =  1.9322
```

Note that the eigenvector is the same as that in part (b) apart from a minus sign, but this is because of the scalar invariance property of eigenvectors: i.e. if $\mathbf{v}$ is an eigenvector then so is $-\mathbf{v}$.

2. (a) The matrix A is defined by

$$\mathsf{A} = \begin{pmatrix} 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 & 0 \end{pmatrix}$$

($\mathsf{A}_{ij} = 1/l_j$ if there is a link from page j to page i and l_j is the total number of links from page j.)

This should be hard-coded into the function Adefine.m

(b) The output should be the matrix A defined above (this will indicate if your definition in Adefine.m is correct) and the vector

```
   0.12780
   0.23012
   0.13863
   0.17120
   0.33224
```

indicating a page ranking of $5, 2, 4, 3, 1$. It took me 29 iterations.

(c) If $q_i > 0$ and $p_i > 0$ are the ith elements of the n-vectors $\mathbf{q}$ and $\mathbf{p}$ then $\mathbf{q} = \mathsf{A}\mathbf{p}$ implies

$$q_i = \sum_{j=1}^{n} \mathsf{A}_{ij} p_j$$

and so

$$\|\mathbf{q}\|_1 = \sum_{i=1}^{n} q_i = \sum_{j=1}^{n} \left(\sum_{i=1}^{n} \mathsf{A}_{ij} \right) p_j = \sum_{j=1}^{n} p_j = \|\mathbf{p}\|_1 = 1.$$

3. (a) See pagerank2.m. We change the line defining B to something like:

```
b = ones(n,1)/n;          % Now b stores a column vector of ones/n
```

and the iterative step to

```
p = d*A*p+(1-d)*b;        % REVISED ITERATION
```

(b) The same answer as Exercise 2(b).

4. (a) Here's the code...

```
function p = pagerank3       % input: nothing; output: p = pagerank vector

d = 0.85;                    % sets the damping factor
A = Adefine                  % call function Adefine to set matrix A
[n,m] = size(A);             % set n = size of matrix A (m = # cols; not needed)
b = ones(n,1)/n;             % set the column vector of ones/n
p = (1-d)*inv(eye(n)-d*A)*b; % perform matrix inversion
end
```

(b) The same answer as Exercise 2(b).

13.10 Solutions 11

1. a) Solving $0 = \alpha p^* - \beta p^{*2}$ gives $p^* = 0$ and $p^* = \alpha/\beta$.

   b) Here's the script

```
%% Population model. Calls separate function ppopfun
global alpha;                        % set alpha, beta as global variables
global beta;
alpha = 1;                           % set constants
beta = 0.5;
clf;                                 % clear graphics frame
hold on;                             % hold graphics on
[t,y] = ode45(@ppopfun,[0 8],2.5); % solve ODE system over 0 < t < 8
plot(t,y,'b-')                       % plot solution against time
[t,y] = ode45(@ppopfun,[0 8],2);   % & different initial condition
plot(t,y,'r--')
[t,y] = ode45(@ppopfun,[0 8],1.5); % & different initial condition
plot(t,y,'g-.')
hold off
```

   which calls function

```
function yd = ppopfun(t,y) % input is t and y, output is yd = dy/dt
global alpha;              % pick up values of constants from calling
```

```
global beta;                % code
yd = alpha*y-beta*(y^2);    % define derivative
end
```

The output is shown in Fig. 13.9 and shows that all initial conditions tend to $p* = \alpha/\beta$ equilibrium solution.

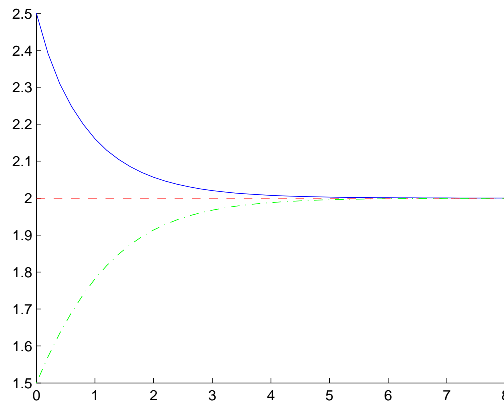


Figure 13.9: Solution for the three different initial conditions.

2. a) Simple: differentiate the first equation and substitute in the second. (This is a method you'll see used in Mechanics 1 course).

   b) $v^* = 0$ and $\alpha x^* + \beta x^{*3} = 0$ gives thee equilibrium point $(x^*, v^*) = (0, 0)$ plus $(x^*, v^*) = (\pm\sqrt{-\alpha/\beta}, 0)$ if α and β have opposite signs.

   c) Code is hybrid of `lv.m` and `lorenz.m`

```
%% Duffing oscillator
global alpha;    % define variables globally over all parts of the code
global beta;
global gamma;
global delta;
alpha =-1;       % set constants
beta = 1;
gamma = 0.3;
delta = 0.2;
clf              % clear graphics
hold on          % allow overwrite of curves
[t,y] = ode45(@duffingfun,[0 200],[1 1]); % solve ODE system over
                 % 0 < t < 200 and with initial condition y(1)=y(2)=1
plot(t,y(:,1),'b-')                  % plot blue solid
[t,y] = ode45(@duffingfun,[0 200],[1 1.0001]); % solve again with
                 % slightly different initial conditions
plot(t,y(:,1),'r--')                 % plot red dashed
hold off
```

```
figure(2)                              % set up second figure
plot(y(:,1),y(:,2))                    % plot the phase portrait
```

where the function called by `ode45` is

```
function yd = duffingfun(t,y) % input is t and vector y, output vector yd
global alpha;                 % pick up values of constants from calling
global beta;                  % code
global gamma;
global delta;
yd = zeros(2,1);          % output of derivatives must be a column vector
yd(1) = y(2);             % define derivatives
yd(2) =-delta*y(2)-alpha*y(1)-beta*(y(1)^3)+gamma*cos(t);
end
```

d) Output of code shown in Fig. 13.10. The solution is chaotic and is sensitive to initial conditions.

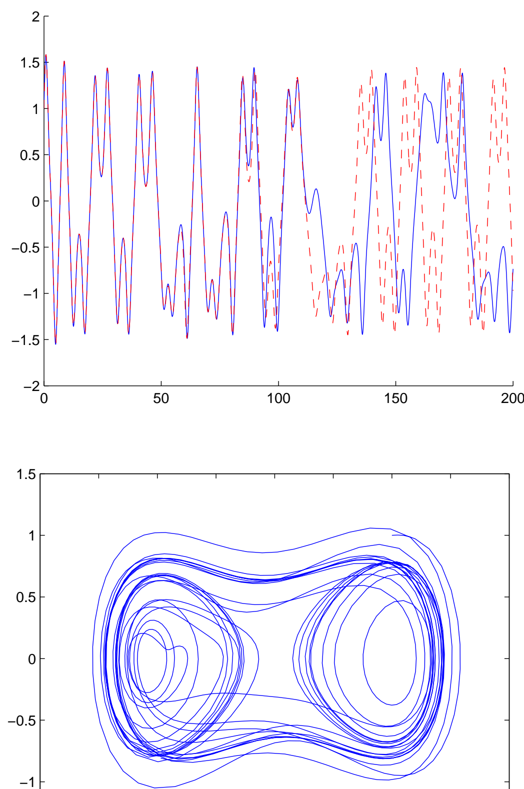


Figure 13.10: Left panel: Variation of $x(t)$ with t for two slightly different initial conditions. Right panel: Phase portrait $x(t)$ versus $v(t)$.

4. a) There is one equilibrium solution: $v^* = 0$ and $\theta^* = 0$.

b) When $a = 0$, $\beta = 0$ and so the two ODEs can be combined into $d^2\theta/dt^2 = \alpha\theta$ and its solution (by Calculus 1 methods) with $\theta(0) = \theta_0$ and $d\theta/dt(0) = v_0$ is

$$\theta(t) = \theta_0 \cosh(t\sqrt{\alpha}) + (v_0/\sqrt{\alpha}) \sinh(t\sqrt{\alpha})$$

If $\theta_0 = 0$ and $v_0 = 0$ then $\theta(t) = 0$ for all time. Otherwise the solution tends to infinity on account of the exponential behaviour of the hyperbolic trig functions.

c) Look on the web page for the code for this.

Solutions (see Fig. 13.11) show that when $\beta = 0.5$ the solution is bounded and periodic but when $\beta = 0.4$ the solution is unbounded and tends to infinity.

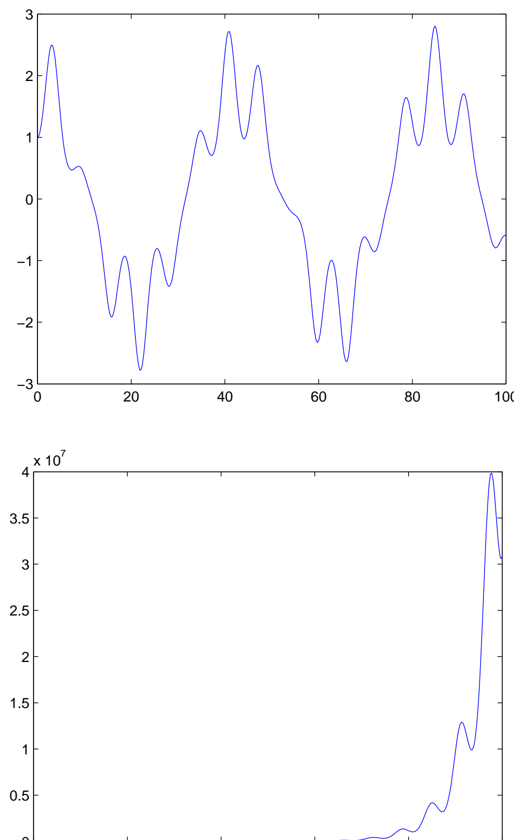


Figure 13.11: $\theta(t)$ against t. Left panel: $\alpha = 0.1$, $\beta = 0.5$. Right panel: $\alpha = 0.1$, $\beta = 0.4$.

d) Trying other values of α and β show that there is a "tongue" of values of α and β which emerge from the β-axis where the solution remains bounded (or stable) and all other values lead to unbounded (unstable) solutions.

5. a) Here's the code, which is adapted from `lmap.m`

```
n = 1000;         % set number of iterates
alpha = -2.75;    % set constants
```

```
beta = 1;
delta = 0.2;
x = zeros(1,n); % set up storage for x_n and y_n
v = zeros(1,n);
x(1) = 1;       % set x_0 and v_0
v(1) = 1;
for j=2:n       % iterate so that x(n), v(n) is final iterate
  x(j) = v(j-1);
  v(j) =-delta*x(j-1)-alpha*v(j-1)-beta*(v(j-1)^3);
end
xx = 1:n;       % set up integers on x axis
plot(xx,x,'-')  % plot x_n against n
figure(2)       % second figure is phase portrait
plot (x,v,'.')
```

b) See Fig. 13.12. The results of n against x_n shows chaotic behaviour and the iterates belong to 'the attractor' (you don't need to understand what this means) shown in the second figure. All quite cool.

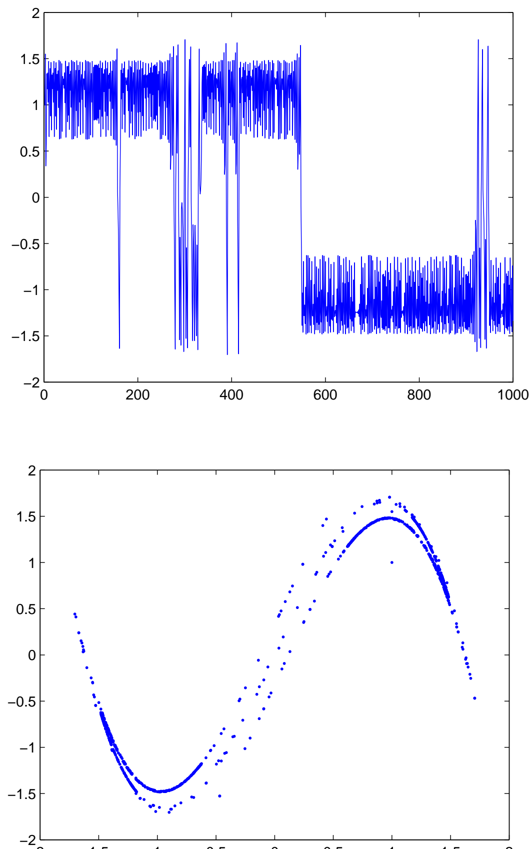


Figure 13.12: Left panel: n against x_n. Right panel: x_n, against v_n.

7. The first thing that is needed is to write the two coupled second-order ODEs in terms of four coupled first-order ODEs. *This is because MATLAB can only solve systems of first order ODEs.* So we write, for e.g.

$$\frac{dx}{dt} = u, \qquad \frac{du}{dt} + 0.025u + x = \sum_{j=1}^{n} \frac{(x_j - x)}{((x - x_j)^2 + (y - y_j)^2 + 0.01)^{3/2}}$$

and

$$\frac{dy}{dt} = v, \qquad \frac{dv}{dt} + 0.025v + y = \sum_{j=1}^{n} \frac{(y_j - y)}{((x_j - x)^2 + (y - y_j)^2 + 0.01)^{3/2}}$$

and the system is described in terms of four variables (x, u, y, v). Look at the code published on the web page. It solves the system of ODEs for $0 < t < 50$ with initial conditions $x(0) = x_0$, $y(0) = y_0$ and $u(0) = 0$, $v(0) = 0$ (initially at rest) and plots $x(t)$ against $y(t)$.

Fig. 5 shows a plot of a typical solution, for $n = 3$ and $x_0 = y_0 = 1$. When you change the values of x_0 and y_0 by a small amount you will find that the dynamics change completely and the pendulum can be attracted, seemingly at random, to any of the 3 magnets.

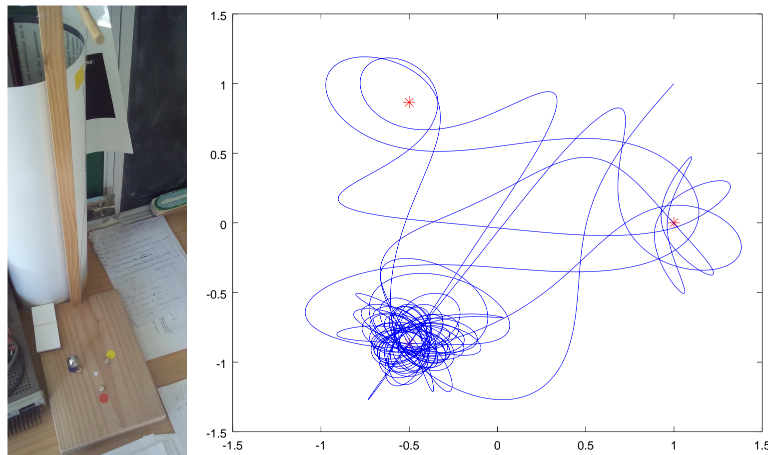


Figure 13.13: A homemade 3-magnet pendulum experiment sitting on the desk in my office and chaotic motion of the pendulum predicted by the numerical solution which starts at $(1, 1)$ and is eventually attracted one of the 3 magnets.