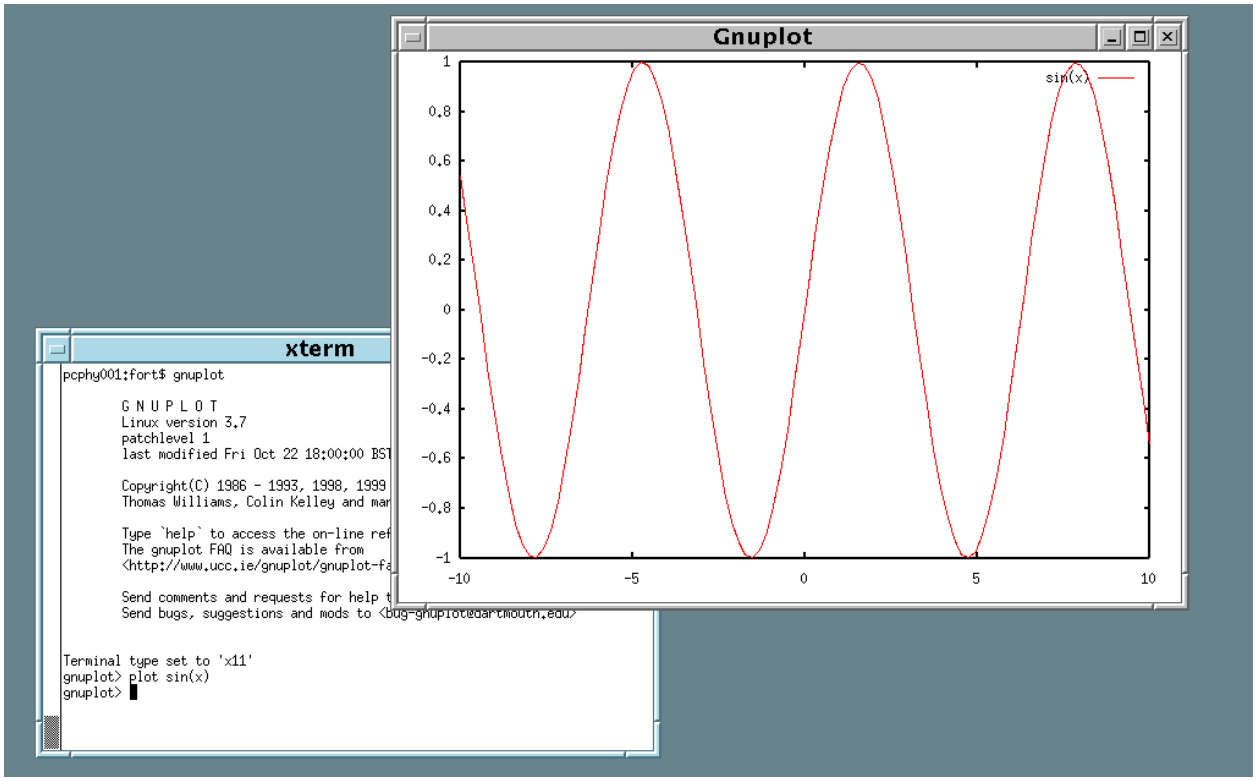
UNIX Utilities
&
Numerical Surprises

Dr MJ Rutter
mjr19@cam.ac.uk

Michaelmas 2001

Gnuplot

Gnuplot is a simple graph plotting program.



Here it is shown plotting `sin(x)`. It has a simple command line interface, and uses a separate window for the resulting graphics.

The `plot` command

The `plot` command takes various options. A more complete example is:

```
gnuplot> plot [xmin:xmax][ymin:ymax] f(x),g(x)
```

where one or both of the ranges may be omitted. It can also plot data files.

```
gnuplot> plot 'results.dat' with lines
```

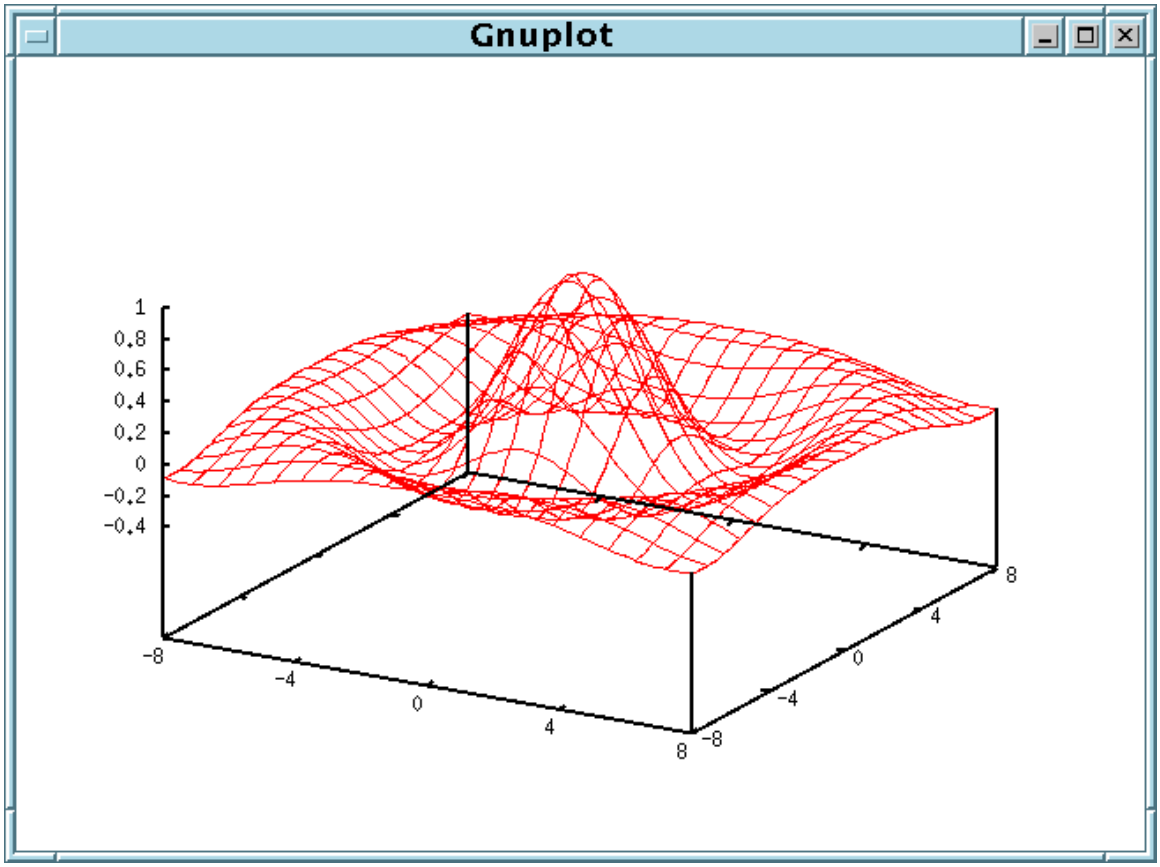
where `results.dat` is a simple file containing a column of `x` values and a column of `y` values.

Omit the '`with lines`' to obtain the points only, use '`with linespoints`' for both. Minimum unique abbreviations are usually accepted, e.g. '`w linesp`' for '`with linespoints`'

The online help system is quite extensive. Type '`help`' to investigate it.

A 2D plot

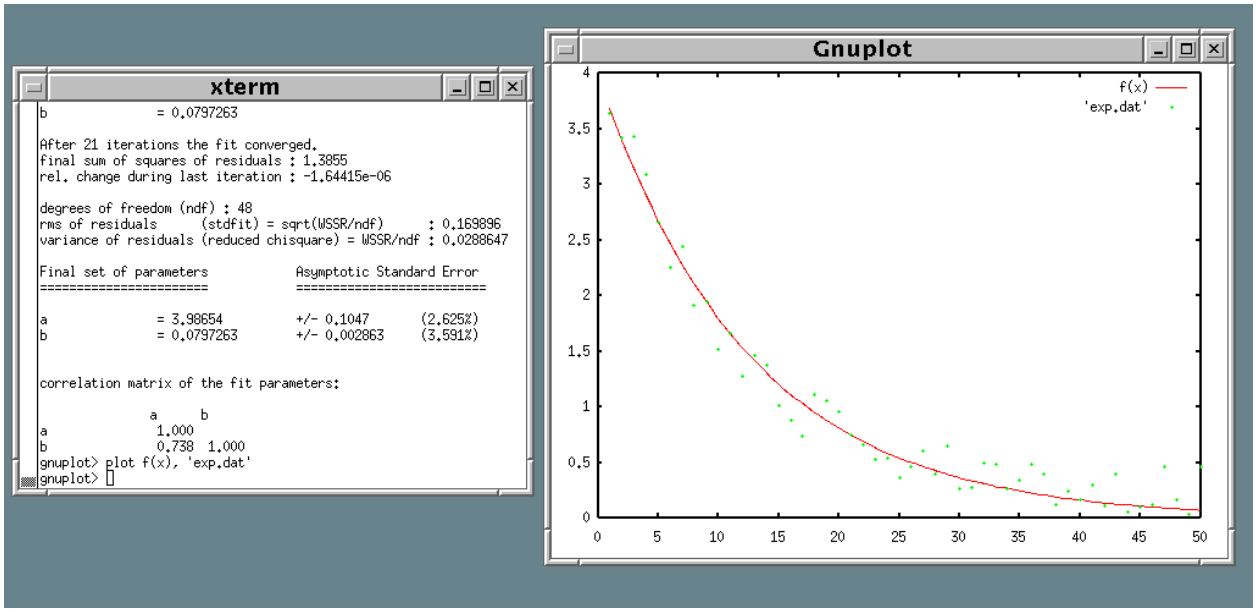
```
gnuplot> sinc(x)=sin(x)/x
gnuplot> set nokey
gnuplot> set xtics 4
gnuplot> set ytics 4
gnuplot> set isosamples 20,20
gnuplot> splot [-8:8][-8:8] sinc(sqrt(x*x+y*y))
```



Curve fitting

Gnuplot also does non-linear function fitting.

```
gnuplot> a=3
gnuplot> b=0.5
gnuplot> f(x)=a*exp(-b*x)
gnuplot> fit f(x) 'exp.dat' via a,b
```

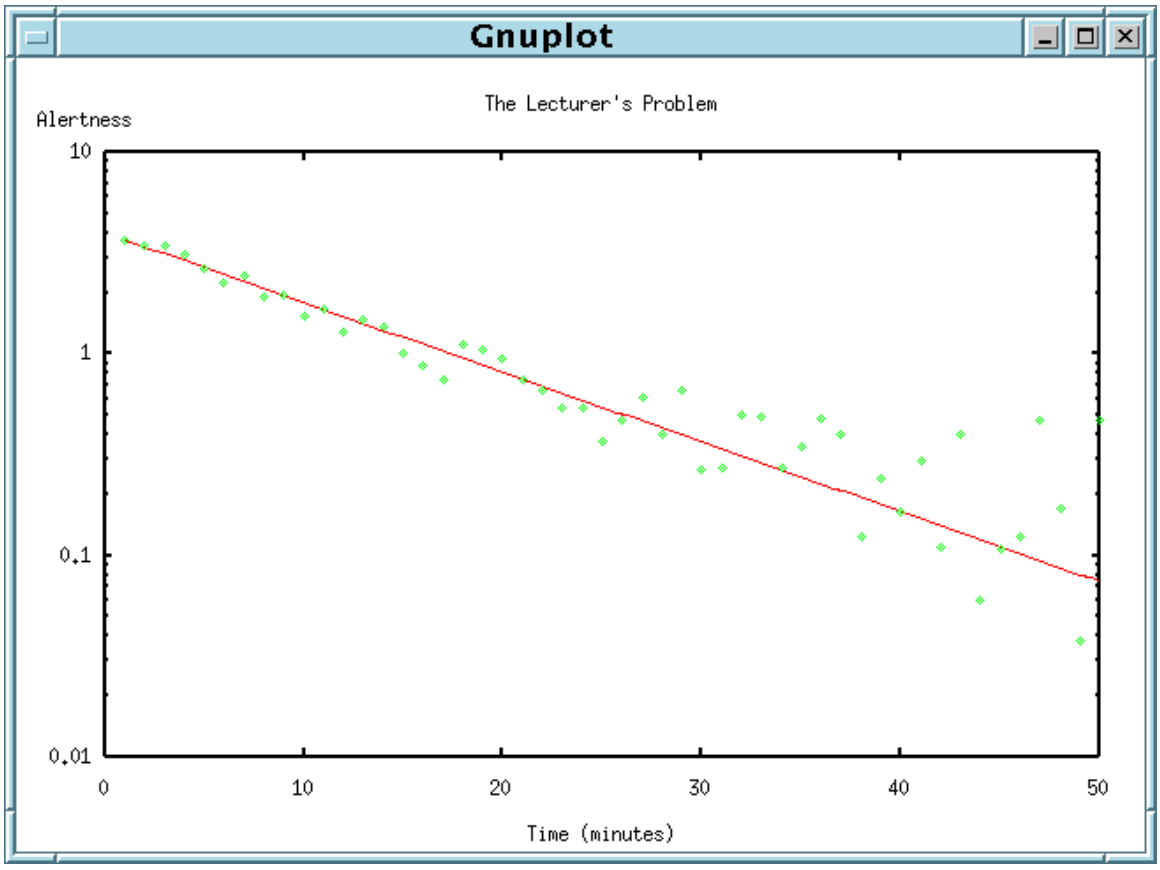



```
do i=1,50
  x=0.1*i ; call random_number(y)
  write(*,*)i,4*exp(-x)+0.5*y
end do
```

produced the data fitted. Notice the noise is sufficient to stop the fitted result ($a = 3.99 \pm 0.1$ and $b = 0.08 \pm 0.02$) being precisely the underlying function.

A labelled plot

```
gnuplot> set xlabel 'Time (minutes)'
gnuplot> set ylabel 'Alertness'
gnuplot> set title "The Lecturer's Problem"
gnuplot> set logscale y
gnuplot> set xtics 10
gnuplot> set nokey
gnuplot> replot
```



Gnuplot and other Programs

By default Gnuplot's output is sent to the screen. It can be sent to a file, in which case it is necessary to set both the file name and the type of output required:

```
gnuplot> set terminal postscript
Terminal type set to 'postscript'
Options are 'landscape noenhanced monochrome...'
gnuplot> set output 'my_plot.ps'
gnuplot> replot
gnuplot> set terminal x11
gnuplot> set output
gnuplot> replot
```

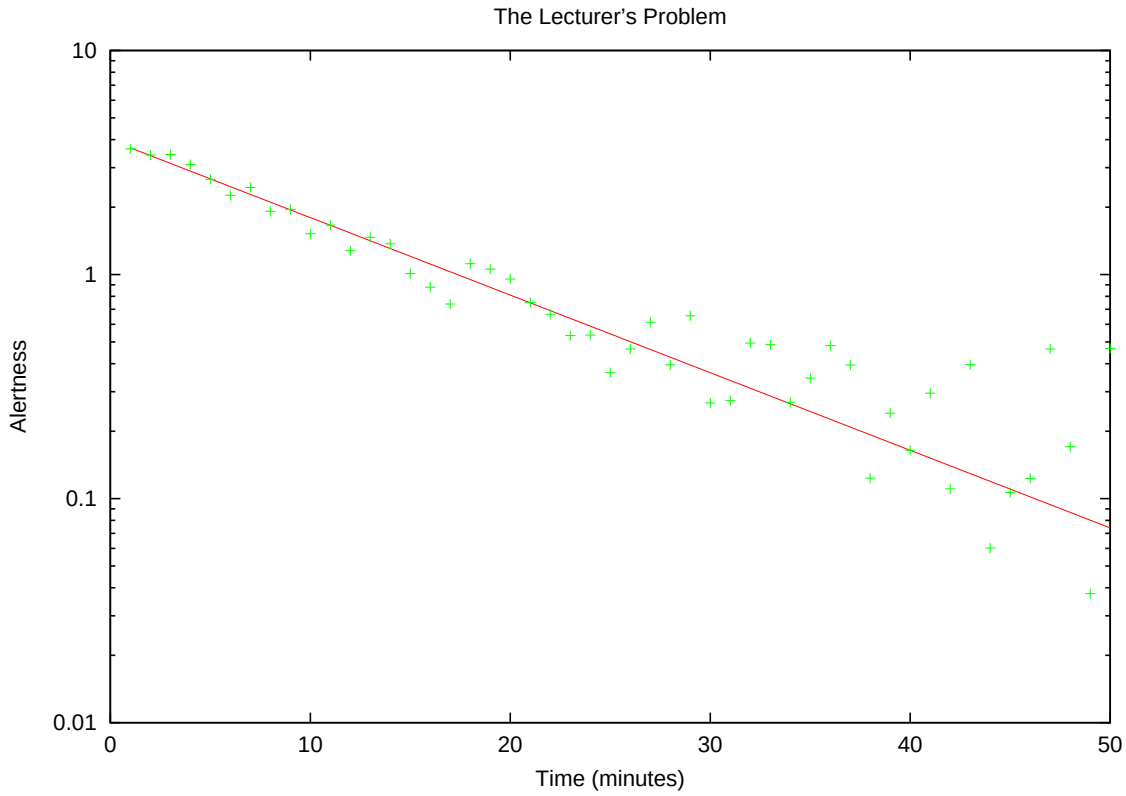
`replot` repeats the last `plot` command.

Useful terminal types include:

- `postscript` for B&W printers
- `postscript color` for colour printers
- `postscript eps color` for LaTeX and other EPS-loving programs
- `cgm` for MS Word and other CGM-loving programs

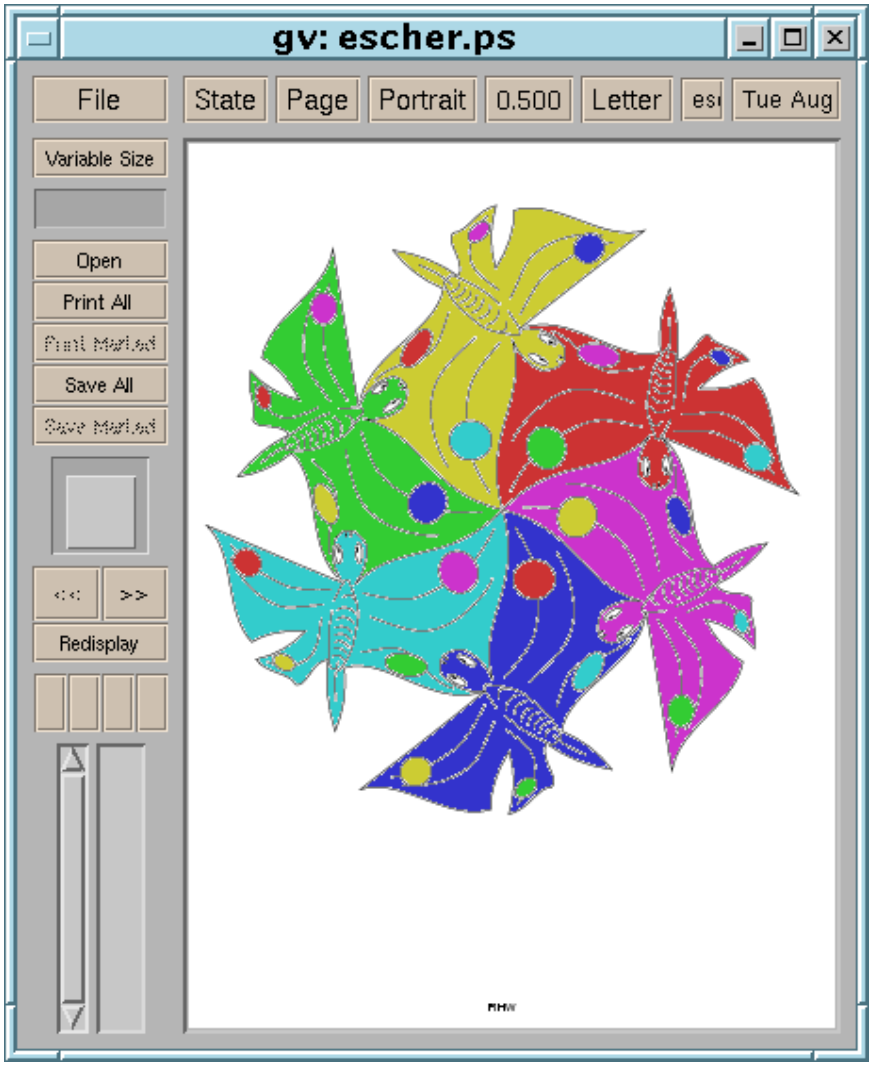
What You Get is Better than What You See

Gnuplot will use rotated text and different fount sizes when using some terminal types. Hence the labelled graph using the `post eps color` terminal type produces:



Viewing PostScript

It is often useful to preview PostScript before printing it. A program called ‘gv’ does this:



Menus hide under the first six buttons on the top row.

Numerical Surprises

People often naively apply the laws of algebra to numerical problems, sometimes with unexpected consequences. As this is not only your first, but also your *last* course in Computational Physics, a few of the pitfalls should be highlighted.

Most of these are of no consequence as far as the set projects are concerned. They may be of useful general interest though.

Discrete maths: the problem

'Real' algebra deals with a number space which is continuous and infinite in extent. Computers, even when dealing with double precision, use a space which is finite and discrete.

There are only 2^{64} unique ways of setting the 64 bits in a double-precision number. Hence there can be no more than 2^{64} different double-precision numbers.

As floating point binary maths is tedious for humans, we shall consider a model numerical system with similar properties, but in base 10!

A simple model

Let us assume that every number we write shall be written as a four digit signed $mantissa$ and a two digit signed exponent.

$$\pm 0.XXXX * 10^{\pm XX}$$

e.g.

$$0.1000 * 10^{1}$$

or

$$0.6626 * 10^{-33}$$

(As is conventional, the mantissa, M, is restricted to $0.1 \leq M < 1$.)

Representable numbers

Using this notation, we can represent at most four million distinct numbers. Having merely six digits which can range from 0 to 9, and two signs, there are only 4,000,000 possible combinations.

The largest number we can represent is $0.9999 * 10^{99}$, the smallest $0.1000 * 10^{-99}$, or $0.0001 * 10^{-99}$ if we do not mind having fewer than four digit precision in the mantissa.

Algebra

$$\boldsymbol{a + b = a \nRightarrow b = 0}$$

$$0.1000 * 10^{1} + 0.4000 * 10^{-3} = 0.1000 * 10^{1}$$

$$\boldsymbol{(a + b) + c \neq a + (b + c)}$$

$$\begin{aligned}
&(0.1000 * 10^{1} + 0.4000 * 10^{-3}) + 0.4000 * 10^{-3} \\
&= \quad 0.1000 * 10^{1} + 0.4000 * 10^{-3} \\
&= \quad 0.1000 * 10^{1} \\
&0.1000 * 10^{1} + (0.4000 * 10^{-3} + 0.4000 * 10^{-3}) \\
&= \quad 0.1000 * 10^{1} + 0.8000 * 10^{-3} \\
&= \quad 0.1001 * 10^{1}
\end{aligned}$$

$$\boldsymbol{\sqrt{a^2} \neq |a|}$$

$$\sqrt{(0.1000 * 10^{-60})^2} = \sqrt{0.0000 * 10^{-99}} = 0$$

Base two

Computers actually use base 2 arithmetic always. Binary fractions follow trivially from decimal fractions:

$$0.625 = 2^{-1} + 2^{-3} = 0.101_2$$

but note that some finite decimal fractions are not finite binary fractions

$$0.2_{10} = 0.0011001100110011\ldots_2$$

(although any finite binary fraction is a finite decimal fraction of the same number of digits)

Thus a computer cannot store 0.2 exactly in a finite number of binary digits.

IEEE 754

IEEE 754 defines a way both of representing and manipulating floating point numbers on a computer. Its use is almost universal.

In a similar fashion to the above decimal format, it defines a sign bit, a mantissa of fixed length, and an exponent.

	Precision	
	Single	Double
Bits, total	32	64
Bits, exponent	8	11
Bits, mantissa	23	52
Largest value	c.$1.7 * 10^{38}$	c.$9 * 10^{307}$
Smallest non-zero	c.$6 * 10^{-39}$	c.$1 * 10^{-308}$
Precision (decimal digits)	c.7	c.15

Getting it right

$$\boldsymbol{a + b = b + a}$$

Very reassuring. IEEE 754 states that addition must return the closest representable number to the true result. Returning to our base 10 examples

$$0.1000 * 10^1 + 0.5432 * 10^0 = 0.1543 * 10^1$$

for this number is the closest representable number to the true answer of $0.15432 * 10^1$.

Similar guarantees exist for subtraction, multiplication and division.

Backwards and Forwards

$$\sum_{n=1}^{N} \frac{1}{n}$$

Consider summing this series forwards $(1..N)$ and backwards $(N..1)$ using single precision arithmetic.

N	forwards	backwards	exact
100	5.187378	5.187377	5.187378
1000	7.485478	7.485472	7.485471
10000	9.787613	9.787604	9.787606
100000	12.09085	12.09015	12.09015
1000000	14.35736	14.39265	14.39273
10000000	15.40368	16.68603	16.69531
100000000	15.40368	18.80792	18.99790

The smallest number such that $15 + x \neq x$ is about 5×10^{-7}. Therefore, counting forwards, the total stops growing after around two million terms.

This is better summed by doing a few hundred terms explicitly, then using a result such as

$$\sum_{n=a}^{b} \frac{1}{n} \approx \log\left(\frac{b+0.5}{a-0.5}\right) + \frac{1}{24}\left((b+0.5)^{-2} - (a-0.5)^{-2}\right) + O(a^{-4})$$

The Quadratic Formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$30x^2 + 60.01x + 30.01 = 0$$

Roots are -1 and $-1\frac{1}{3000}$.

Single precision arithmetic and the above formula may give no roots or repeated roots!

number	nearest representable single prec. no.
30	30.0000000000
30.01	30.0100002289. . .
60.01	60.0099983215. . .

Even with no further rounding errors, whereas $4 * 30 * 30.1 = 3601.2$ and $60.01^2 = 3601.2001$, $60.0099983215\ldots^2 = 3601.199899\ldots$.

The final roots may be distinct in single-precision arithmetic, but the intermediate values are not being stored with sufficient precision.

The Logistic Map

$$x_{n+1} = 4x_n(1 - x_n)$$

n	single	double	correct
0	0.5200000	0.5200000	0.5200000
1	0.9984000	0.9984000	0.9984000
2	0.0063896	0.0063898	0.0063898
3	0.0253952	0.0253957	0.0253957
4	0.0990019	0.0990031	0.0990031
5	0.3567998	0.3568060	0.3568060
10	0.9957932	0.9957663	0.9957663
15	0.7649255	0.7592756	0.7592756
20	0.2214707	0.4172717	0.4172717
30	0.6300818	0.0775065	0.0775067
40	0.1077115	0.0162020	0.0161219
50	0.0002839	0.9009089	0.9999786
51	0.0011354	0.3570883	0.0000854

With just three operations per cycle, this series has even double precision producing rubbish after just 150 elementary operations.

Making Life Complex

Processors deal with real numbers only. Many scientific problems are based on complex numbers. This leads to major problems.

Addition is simple

$$(a + ib) + (c + id) = (a + c) + (b + d)i$$

and subtraction is similar.

Multiplication is slightly tricky:

$$(a + ib) \times (c + id) = (ac - bd) + (bc + ad)i$$

What happens when $ac - bd$ is less than the maximum number we can represent, but ac is not?

What precision problems do we have if ac is approximately equal to bd?

The Really Complex Problem

$$(a+ib)/(c+id) = \frac{(ac+bd)+(bc-ad)i}{c^2+d^2}$$

This definition is almost useless!

If N is the largest number we can represent, then the above formula will produce zero when dividing by any number x with $|x| > \sqrt{N}$.

Similarly, if N is the smallest representable number, it produces infinity when dividing by any x with $|x| < \sqrt{N}$.

This is not how languages like FORTRAN, which support complex arithmetic, do division: they use a more complicated algorithm which we shall quietly ignore.

Powers

$$(-32)^{0.2}$$

has a reasonable real value: -2. No binary computer can be reasonably expected to find this using trivial numerics though. As 0.2 cannot be stored precisely as a binary fraction, and the behaviour of

$$(-32)^{0.2+\epsilon}$$

for small ϵ is unfriendly, a binary computer can be expected to fail. Fortran considers raising a negative real or integer value to a any real power to be an error.

Raising negative numbers to integer or complex powers is fine

Hope

If simple operations are so awkward, how can one solve complex problems?

The first answer is don't try. Rely on a library written by someone who understands numerical analysis well, and which does decent error checking for you.

The second answer is that complex problems can be easier than simple ones! Often complex algorithms are inherently stable, unlike the logistic map which is inherently unstable.

The Unstable

```
real x
integer i

write(*,*)'Input initial value'
read(*,*)x

do i=1,10
  write(*,*)i,x
  x=4*x*(1-x)
enddo

end
```

Try with initial values between 0.1 and 0.9.

Initial value: 0.100 Tenth value: 0.962
Initial value: 0.101 Tenth value: 0.630

The Stable

```
real x
integer i

write(*,*)'Input initial value'
read(*,*)x

do i=1,10
  write(*,*)i,x
  x=0.5*(x+2/x)
enddo

end
```

Try with initial values between 0.1 and 0.9.

Initial value: 0.100 Tenth value: 1.414
Initial value: 0.101 Tenth value: 1.414
Initial value: 0.500 Tenth value: 1.414

If the '2' is replaced by a different value, this algorithm will converge to its square root instead. Thus a square root has been reliably calculated using only the operations $+$, $*$ and $/$.

The End

By now you have seen a reasonable amount of background material on computing, as well as a detailed introduction to UNIX and Fortran, together with some opportunity for practising your (new?) computing skills.

You are thus well placed to benefit from Dr Haynes' lectures on Computational Physics, and, thereafter, to demonstrate your knowledge by solving one of the set problems.