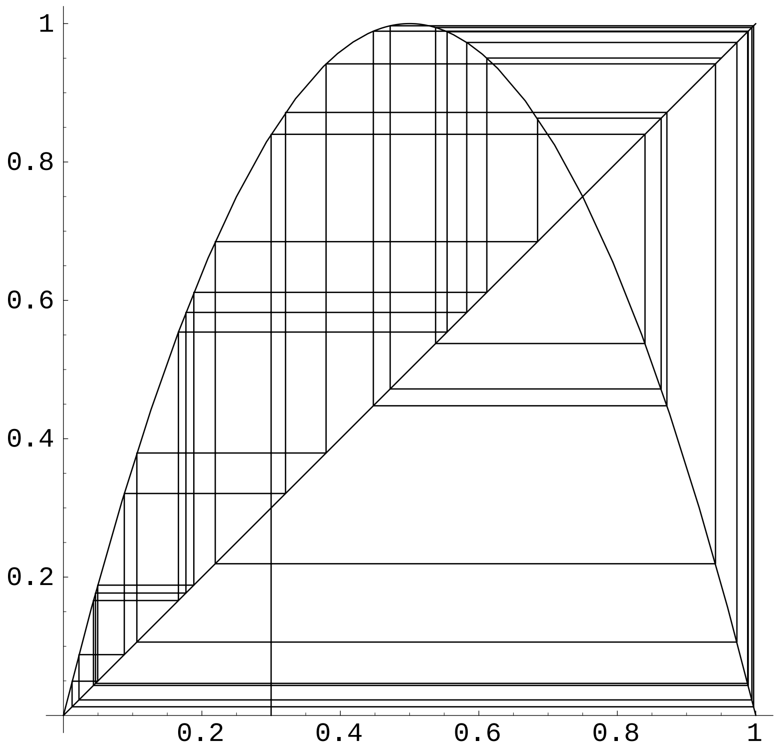
An Introduction to *Mathematica*



Phil Ramsden and Phillip Kent
The METRIC Project,

Mathematics Department, Imperial College

Version 3.2; 27 April 1999.

This booklet is a *Mathematica* notebook. It can be found online at

```
http://metric.ma.ic.ac.uk/mathematica
```

■ Outline of the course

Session 1: common and useful built-in *Mathematica* functions; variable assignment and function definition; the Front End and the Kernel; Notebooks.

Session 2: organisation of data in *Mathematica*; lists and expressions; simple programming; functions; nesting.

Session 3: the opportunity to develop your proficiency as a *Mathematica* user through work on an extended problem; Case Studies; *Mathematica* resources on the Internet.

■ Rationale

We're not attempting an exhaustive survey of *Mathematica*'s capabilities: we couldn't come close to doing justice to that task in the time we have. Equally, there are dozens of specialised uses for *Mathematica* (in pure and applied mathematics, physical science, engineering etc.) that we can't hope to address here (though some are touched on in our "Case Studies": see below). Instead, we focus on the key elements of the *Mathematica* system and how the system is used.

These course notes are not intended as a substitute for the manual, which is *The Mathematica Book* (Cambridge University Press, Third Edition, 1996), by Stephen Wolfram. The entire contents of the manual, and more, are available on *Mathematica*'s extensive online Help system, which you should certainly take time to explore.

In addition to these course notes we have prepared some Case Studies, or "common tasks in *Mathematica* for the academic user". These are an attempt to address just a few of the more specialised roles in which *Mathematica* is used.

This booklet includes information about the various free sources of *Mathematica* information on the Internet, and how to get in touch with the world-wide *Mathematica* user community. There are numerous other books besides the Wolfram "manual" about *Mathematica* itself, and its use in mathematics, science, engineering and finance (and some of these are available in other languages).

■ Session 1

In this session, we explore the arithmetical, symbolic and graphical capabilities of *Mathematica*. We cover global variable assignment and local variable substitution, and introduce you briefly to the idea of defining your own *Mathematica* functions (to be covered in more depth in Session 2). We also explore some key characteristics of *Mathematica*'s user interface. The final section (1.6) on Notebooks is optional: you may prefer to skip it for now and come back to it at a later time.

Each section consists of a piece of text followed by some exercises. Exercises marked with a five-pointed star (⋆) are central, and you should do all of these if you have the time. The other exercises, while useful, are more peripheral and can be skipped if necessary.

All *Mathematica* code is printed in these notes in `Courier` Font. We have used the ellipsis mark "..." to indicate where code has been missed out.

■ Getting help

If you get stuck, here are some ways to recover:

- Use the Help systems, which are especially useful for finding out about *Mathematica* functions. There is the system activated from the **Help** item in the menu (this gives access, amongst other things, to the entire *Mathematica* user manual), or you can use the special query character `?` to get information about any function, for example:

  ```
  ? Sqrt
  ```

- You can do "wildcard" searches as well. The following queries ask *Mathematica* to list all the function names beginning with, or ending with, `Plot`, respectively:

  ```
  ? Plot*
  ```

  ```
  ? *Plot
  ```

- If all you need is a reminder of the syntax of a command, type the command name, then hold down the shift and control keys and type K, for example

  ```
  Plot < shift - control - K >
  ```

- If a command doesn't work properly check the error messages (in blue text).

- If your input is simply returned unchanged, with no error messages to help, it means that *Mathematica* is unable to do anything with what you have typed. Check that you have spelt the command correctly, that the number of inputs is correct, that you haven't left out any commas, and that the types of the inputs (integer, real number, symbol, and so on) are appropriate. These are the most common causes of this error.

- If *Mathematica* seems to have stopped, **Abort** the calculation or (more drastic) **Quit** the Kernel, using the **Kernel** menu.

- If everything seems to have gone wrong **Quit** or **Exit** from *Mathematica* (via the **File** menu) and start again. It's a good idea to **Save** your work as you go along so that you can recover from these situations.

1.1 Arithmetic

At its simplest, *Mathematica* can be thought of as a highly sophisticated calculator. Like a calculator, it does arithmetic, for example:

```
2 + 5

2 * 5

2 5

2^5

100!

Sin[Pi / 3]

Sqrt[50]

2^(1 + 4)

Log[2, %]
```

etc. To get *Mathematica* to perform a calculation, hold down the *shift* key and press *return* (on some keyboards called *enter* or ↵). The shift-return operation sends "instructions" from the interface where you' re typing to the "engine" of *Mathematica* for processing: see Sections 1.6–1.7 for more about this. So to lay out code more clearly on the screen you can use "return" characters.

You'll notice right away two peculiarities of the syntax. One is that the names of all *Mathematica* functions, variables and constants begin with capital letters. This is important: *Mathematica* is completely *case-sensitive*, and it will simply be unable to interpret, for instance:

```
sin[pi / 3]
```

The other is that square brackets, `[...]` and round parentheses, `(...)` are both used in *Mathematica*, but *not* interchangeably. The former are always used to enclose the *arguments* of functions such as `Sin`. The latter are used only for the purpose of *grouping expressions*, either algebraically, as here, or procedurally, as you'll see in Session 2. Otherwise, the notation for arithmetic is straightforward, except to note that a space can implicitly mean multiplication.

The percent sign, `%`, is used to mean "the last output" (so the final expression in the above list will calculate the logarithm to base 2 of 32). You may have noticed that all inputs and outputs are numbered and can be referred back to using those numbers (see Section 1.6).

Note, too, that some of the above calculations can be laid out in a way that corresponds more closely to conventional mathematical notation, by using the **Basic Input** palette. This will probably have appeared automatically on the right of your screen. If it hasn't, find it in the **File** menu under **Palettes**. For example:

$$2^5$$

$$\mathrm{Sin}\left[\frac{\pi}{3}\right]$$

$$\sqrt{50}$$

$$2^{1+4}$$

and so on. This is especially useful when you need to build up large expressions.

All the above are examples of *exact* arithmetic, carried out using whole numbers, rationals, surds or rational multiples of constants such as `Pi`. *Mathematica* will also perform approximate, *floating-point* arithmetic of the sort that conventional calculators can do. Numbers entered with a decimal point are interpreted as approximations, even if they're integers, and all other numbers in the same expression (with the exception of symbolic constants such as `Pi`) will be converted to this latter form. For example:

```
100.0!
```

$$\sqrt{50.0}$$

$$3.35759^{100}$$

To force *Mathematica* to convert exact expressions to decimal ones, you can use the `N` command, as in:

$$\mathrm{N}\left[\sqrt{50}\right]$$

```
N[Sin[π / 3]]
```

$$\mathrm{N}\left[\sqrt{50},\ 25\right]$$

```
N[π, 200]
```

The last two cases illustrate one way in which *Mathematica* can be made to work to arbitrary precision.

Mathematica will handle complex numbers as well as real ones: see the exercises for some examples.

■ Exercises 1.1

⋆ 1. Type in and test all the code in this section.

⋆ 2. Try the following:

```
(3 - 2 I) * (1 + I)
```

$$(1 + 5\,\mathrm{I})^2$$

```
Conjugate[2 - 5 I]
```

```
Abs[12 - 5 I]
```

$\texttt{Arg}\left[1 + \sqrt{3}\ \texttt{I}\right]$

$\texttt{Exp}\left[\frac{1}{2}\ \texttt{Log[2]} + \frac{\pi}{4}\ \texttt{I}\right]$

$\texttt{Simplify}\left[\texttt{Exp}\left[\frac{1}{2}\ \texttt{Log[2]} + \frac{\pi}{4}\ \texttt{I}\right]\right]$

$\texttt{ComplexExpand}\left[\texttt{Exp}\left[\frac{1}{2}\ \texttt{Log[2]} + \frac{\pi}{4}\ \texttt{I}\right]\right]$

■ 1.2 Algebra and Calculus

As well as being an arithmetical calculator, *Mathematica* is also an algebraic one. For example:

$\texttt{Expand[(x + 2 y)}^2\ \texttt{(x - 3 y)}^5\texttt{]}$

```
Factor[%]
```

For more on the manipulation of algebraic expressions, see the exercises.

Equations in *Mathematica* are set up using a double equals sign, "==": this is because the single equals sign has a different meaning, which we introduce later on. The `Solve` command tries to find exact solutions to algebraic equations:

$\texttt{Solve[x}^2 - 3\ \texttt{x} + 2 == 0,\ \texttt{x]}$

$\texttt{Solve[x}^4 - 3\ \texttt{x}^3 + 5\ \texttt{x}^2 - 11\ \texttt{x} + 2 == 0,\ \texttt{x]}$

```
Solve[{x + 4 y == 5, 2 x - y == 8}, {x, y}]
```

Notice the use of curly brackets — braces — in the last `Solve` command. Curly brackets are used in *Mathematica* to group pieces of data together, forming structures called **lists**. These are studied in more depth in Session 2. For the moment, it is enough to note the kinds of circumstances when lists crop up. Here, we need to group the two equations, $x + 4y = 5$ and $2x - y = 0$, and the two unknowns, x and y.

For equations that do not have exact solutions, or for those whose exact solutions are unwieldy (such as quartic polynomials), there is the `NSolve` command which operates using a sophisticated repertoire of numerical methods :

$\texttt{NSolve[x}^7 + 3\ \texttt{x}^4 + 2 == 0,\ \texttt{x]}$

$\texttt{NSolve[x}^4 - 3\ \texttt{x}^3 + 5\ \texttt{x}^2 - 11\ \texttt{x} + 2 == 0,\ \texttt{x]}$

Mathematica will perform calculus operations too. Single-variable differentiation:

```
D[x^2, x]
```

or equivalently:

$\partial_x x^2$

Partial differentiation:

```
D[y x^2, x]
```

or equivalently:

$\partial_x (y x^2)$

Total differentiation:

```
Dt[y x^2, x]
```

Indefinite integration:

```
Integrate[x E^(x^2), x]
```

or equivalently:

$\int x E^{x^2} \, dx$

Definite integration:

```
Integrate[x E^(x^2), {x, -3, 3}]
```

or equivalently:

$\int_{-3}^{3} x E^{x^2} \, dx$

The `NIntegrate` command uses numerical integration methods: essential for those cases where analytical approaches would be difficult or inappropriate. For example

```
NIntegrate[E^(-x^2/2), {x, 0, 1}]
```

■ Exercises 1.2

★ 1. Type in and test all the code in this section.

2. Use *Mathematica* to find all the solutions in the complex plane of the equation cos $z = 2$.

3. Use *Mathematica* to express $\frac{1}{z+1}$ in terms of its real and imaginary parts, where $z = x + i\,y$, and x and y are real.

★ 4. Try the following:

Apart$\left[\frac{2\,x}{(1+x^2)\,(1+x)}\right]$

```
Together[%]
```

Expand$[(3 + 2\,x)^2\ (x + 2\,y)^2]$

```
Collect[%, x]
```

Expand$[(3 + 2\,x)^2\ (x + 2\,y)^2]$

```
Simplify[%]
```

Cancel$\left[\frac{x\text{^}2 + 5\,x + 6}{x+3}\right]$

Numerator$\left[\frac{x\text{^}2 + 5\,x + 6}{x+3}\right]$

★ 5. Open the Algebraic Manipulation palette (under **Palettes** in the **File** menu). This palette, unlike Basic Input, has the setting "Evaluate in Place". To find out what this means, first type, *without evaluating*,

$$\int \frac{2+3\,x+x^2}{2+2\,x+x^2}\,dx$$

Then select the fraction inside the integral, and click on the `Apart [■]` button. With the same piece of text selected, click on `Together [■]`. Try using `Expand [■]` on the numerator, and so on. Explore further. Investigate, too, the use of the **Evaluate in Place** instruction (under **Evaluation** in the **Kernel** menu).

6. Type

```
Sum[1 / r^2, {r, 1, 6}]
```

or

$$\sum_{r=1}^{6}\left(\frac{1}{r^2}\right)$$

Try summing from 1 to 20. Express the sum as a decimal.

Try summing from 1 to *n*, and from 1 to infinity (`Infinity` in *Mathematica,* or use the ∞ symbol from the Basic Input palette).

7. Solve the ordinary differential equation

$$\frac{d^2\,y}{dx^2} + y = 0$$

by typing

```
DSolve[y''[x] + y[x] == 0, y[x], x]
```

Solve this differential equation subject to the initial conditions $y(0) = 1$, $y'(0) = 0$, by typing

```
DSolve[{y''[x] + y[x] == 0, y[0] == 1, y'[0] == 0}, y[x], x]
```

Find a second-order linear ODE that *Mathematica* cannot solve.

■ 1.3 Assignment, substitution and function definition

As you've seen, the percent sign, `%`, gives us a useful way of referring to earlier output. And in fact you can refer to any output in this way using its "In/Out" number—see Section 1.6. However, it's inadvisable to rely on `%` in this way. The principal drawback is that if you save your work and call it up again, or even if you need to edit or debug work you've already done, the sequencing on which `%` depends can be disrupted. It's better instead to get into the habit of **naming** things which it's likely you'll want to use again, like this:

```
expression1 = 2 x / ((1 + x^2) (1 + x))

expression2 = Apart[expression1]

expression3 = Together[expression2]

TrueQ[expression1 == expression3]
```

An important thing to note is the use of the single equals sign, =, in commands such as

```
expression1 = 2 x / ((1 + x^2) (1 + x))
```

which means "let the symbol `expression1` have value $\frac{2x}{(1+x^2)(1+x)}$". This is to be distinguished from the double equals sign, ==, which, as you've seen, is used to set up equations. The final command,

```
TrueQ[expression1 == expression3]
```

means "test whether the equation `expression1 == expression3` is true for all values of the variable or variables". Notice the final `Q` in the function name: this is a convention for logical functions (those whose output is `True` or `False`).

Assigning values to symbols in this way is clearly very useful: indispensable, in fact. But one thing that's a disadvantage in some circumstances is that these assignments are completely *global*: unless we take steps, the symbol `expression1` will continue to call up the value $\frac{2x}{(1+x^2)(1+x)}$ in whatever future context we use it. This is not irreversible: we can make `expression1` into an unassigned symbol again by **clearing** its value:

```
Clear[expression1]
```

We could also quit our *Mathematica* session: that will clear all assignments pretty effectively, and leave everything clear for our next go! But these approaches are all fairly cumbersome, and it's sometimes more appropriate to avoid global assignments of this type and opt for **local substitution** instead.

Compare the following pieces of code, each of which aims at finding the value of the expression $x^2 - 5x + 9$ at $x = 3$. Here’s the first one:

```
x = 3
x^2 - 5 x + 9
Clear[x]
```

Here’s the second:

```
x^2 - 5 x + 9 /. x -> 3
```

In the first, it’s clear what we’ve done: the value 3 has been assigned to the symbol `x`, and the quadratic expression evaluated; finally, the symbol `x` has been cleared. The second piece of code is more obscure: it means “evaluate the expression $x^2 - 5x + 9$ *subject to the local substitution* $x = 3$”. The “`/.`” is a shorthand for the `ReplaceAll` command. It is not necessary to clear `x` afterwards, since `x` has never been assigned any value. Instead, all occurences of `x` in the expression $x^2 - 5x + 9$ have simply been replaced by 3, with no permanent effect on `x` at all.

The structure `x -> 3` is an example of what’s called a **rule**. You may recall that the output from `Solve` is generally in the form of a list of rules (more about that in Session 2, and Case Study 6).

A related idea to assignment is **function definition**. Here’s an example:

```
Clear[x]
f[x_] := x^2 - 5 x + 9
```

What’s happened here is this: the symbol `x` has been cleared, in case it had any value attached to it, and the function *f* has been defined, such that $f(x) = x^2 - 5x + 9$. We can now use this function like any built-in one, for example:

```
f[3]

f[z]

D[f[z], z]

f'[x]
```

Notice that we’ve used the compound symbol `:=` instead of = in the definition This is almost always appropriate for function definition, and = is almost always appropriate for variable assignment, though this is more complex than it seems and exceptions do exist.

Notice, too, that on the right-hand side of the defining statement the **underscore** symbol, _, has been used. This gives `x` the status of a **placeholder** or **dummy variable**, standing for all possible arguments. If you want to explore what happens when you leave the underscore out, try typing

```
Clear[x, f]
f[x] := x^2 - 5 x + 9
```

```
f[x]

f[3]

f[z]

D[f[x], x]

D[f[z], z]

f'[x]
```

Working with your own functions in *Mathematica* always involves two distinct stages: first you define the function, using the underscore character and (usually), "colon-equals". After that, *Mathematica* has "learnt" this new function, and for the rest of your session you can use it in just the same way as inbuilt functions such as `Sin` and `Sqrt`. Note that the first step, defining the function, doesn'tgenerate any output; this can be disconcerting the first few times you see it.

Our `f` in the above example corresponds exactly to a “function” in the mathematical sense. But in *Mathematica*, the term is rather broader. For example, the following is a “function” for comparing two expressions and deciding whether they appear to be algebraically equivalent (as far as *Mathematica* can make out):

```
algEquivQ[expr1_, expr2_] := TrueQ[Simplify[expr1 - expr2] == 0]
```

Having "taught" *Mathematica* the algEquivQ function, we can now use it. The expressions $(x+1)^2 - 1$ and $x(x+2)$ are algebraically equivalent...

```
algEquivQ[(x + 1)^2 - 1, x (x + 2)]
```

... whereas the expressions $(x+1)^2 - 1$ and $x^2(x+2)$ are not:

```
algEquivQ[(x + 1)^3 - 1, x^2 (x + 2)]
```

Notice that we've made all our functions start with lower case letters. This is a good idea in general, to avoid clashes between your own functions and internally defined *Mathematica* ones and to make it clear, to yourself and other users, which is which.

■ Exercises 1.3

⋆ 1. Type in, and test, the first section of code, which assigns values to the symbols `expression1`, `expression2` and `expression 3` and tests the equivalence of `expression1` and `expression3`. Find, in turn, the value of each of these expressions when `x` is 5: do this by assignment *and* by local substitution.

Implement `expression1` as a function of `x`, and check that this function evaluates to what you would expect at 5.

⋆ 2. Define the function `algEquivQ` as in the text. Test it on the pairs:

(i) $x^2 + 2x + 1$ and $(x+1)^2$;

(ii) $\frac{y^2+5y+6}{y+3}$ and $y+2$;

(iii) $\cos 2t$ and $\cos^2 t - \sin^2 t$.

Try to find an equivalent pair for which `algEquivQ` fails. How about if you use `FullSimplify` instead of `Simplify`?

3. Write, and try out, your own function called `equalAtQ`, which tests whether two expressions in the same variable have equal value at a given value of the variable. Thus

```
equalAtQ[2 x^2, 6 x, {x, 3}]
```

should return `True`.

4. The functions `size` and `bigger`, defined below, make use of *Mathematica*'s `If` command.

```
size[x_] := If[x > 2000, "big", "small"]

bigger[a_, b_] := If[a < b, b, a]
```

Explore these two functions by typing, for example

```
size[1000]

size[10000]

bigger[3, 4]

bigger[3, 3]
```

and so on. Try:

```
bigger[Log[4], 2 Log[2]]
```

What seems to have gone wrong? Use these inputs to test the following two "improvements" of `bigger`:

```
bigger2[a_, b_] := If[TrueQ[a < b], b, a]

bigger3[a_, b_] := If[a < b, b, a, $Failed]
```

Which "improvement" do you think is better and why?

■ 1.4 Graphics

Mathematica incorporates a wide range of two-and three-dimensional graphics functions. The simplest is `Plot`, which generates two-dimensional Cartesian graphs, as in:

```
Plot[Sin[x], {x, -π, π}]

Plot[{Sin[x], Cos[x]}, {x, -π, π}]
```

(Notice again the use of curly brackets to form *lists*.) It's important to bear in mind that `Plot` always assumes that graphs are *continuous*. Functions with asymptotes will often come out wrong, therefore, with large, positive values joined to large, negative values across an asymptote (try plotting `Tan[x]` to see the effect).

You can take control of some of the characteristics of the plot by means of what are called **option settings**, for example:

```
Plot[Sin[x], {x, -π, π}, PlotRange -> {- 1.5, 1.5}]

Plot[Sin[x], {x, -π, π}, PlotRange -> {{-2 π, 2 π}, {-1.5, 1.5}}]

Plot[Sin[x], {x, -π, π}, AspectRatio -> 1]

Plot[Sin[x], {x, -π, π}, AspectRatio -> Automatic]

Plot[Sin[x], {x, -π, π}, AspectRatio -> Automatic,
 PlotRange -> {{-2 π, 2 π}, {-1.5, 1.5}}]
```

These use the substitution rules you met in Section 1.3. For a complete list of options for `Plot` together with their default settings, type:

```
Options[Plot]
```

For all that and more, type:

```
?? Plot
```

Note that many *Mathematica* functions feature option settings in this way: they are by no means confined to graphical functions such as `Plot`. Options are an important way of building in flexibility, and you can do this with your own functions too (see the manual: Section 2.3.10).

The function `ParametricPlot` can be used to generate plots of pairs of parametric equations, as in

```
ParametricPlot[{t + Sin[t], 1 + Cos[t]}, {t, 0, 4 π}]
```

The function `ListPlot` can be used to generate plots of sets of coordinates structured as lists, as in:

```
ListPlot[
 {{0.0, 1.2}, {1.0, 2.9}, {2.0, 5.3}, {3.0, 7.0}, {4.0, 8.8}}]
```

Two graphics can be combined on the same pair of axes by means of the `Show` command:

```
lineplot = Plot[2 x + 1, {x, 0, 4}]

dotplot = ListPlot[
  {{0.0, 1.2}, {1.0, 2.9}, {2.0, 5.3}, {3.0, 7.0}, {4.0, 8.8}}]

Show[lineplot, dotplot]
```

The principal three-dimensional plotting functions are `Plot3D`, `ParametricPlot3D` and `ContourPlot` (the last-named produces a two-dimensional contour plot of a function of two variables). These are explored further in the exercises for this section.

Notice, by the way, the way one of the inputs above is broken over two lines, to fit within the width of the page. *Mathematica* does this automatically, or you can override the default line breaks using the "return" key.

■ Exercises 1.4

⋆ 1. Type in, and test, all the code in this section. Note down in particular the effect of all the option settings for `Plot`. Explore this further if you need to. Test the effect on the `ListPlot` command of the options `PlotJoined -> True` and `PlotStyle -> PointSize[0.03 ]`.

2. Use `If` to define a function called `unitStep`, which evaluates to 0 for inputs equal to 0 or less, and to 1 otherwise. Generate a plot of this function for $-3 \le x \le 3$.

3. Use the `Show` command to generate a figure showing a straightforward function plot of the curve $y = x^2$ on the same axes as a *parametric* plot of the curve $x = y^2$. The scales should be the same on either axis.

4. Write a function called `plotWithInverse`, such that, for example

```
plotWithInverse[x^2 - x^4, {x, -3, 3}]
```

returns a plot of the curve $y = x^2 - x^4$ on the same pair of axes as a (parametrically defined) plot of the curve $x = y^2 - y^4$. The scales should be the same on either axis.

⋆ 5. Type the following:

```
Plot3D[(x^2 - y^2) E^(-x^2-y^2), {x, -2, 2}, {y, -2, 2}]

ContourPlot[(x^2 - y^2) E^(-x^2-y^2), {x, -2, 2}, {y, -2, 2}]

ContourPlot[(x^2 - y^2) E^(-x^2-y^2), {x, -2, 2}, {y, -2, 2},
  Contours -> {-0.1, -0.05, 0.0, 0.05, 0.1}]

ParametricPlot3D[{Cos[θ], Sin[θ], h}, {θ, 0, 2 π}, {h, -3, 3}]

ParametricPlot3D[{Cos[θ], Sin[θ], θ / 5}, {θ, 0, 8 π}]
```

See Case Study 3 for how contour and surface plots may be combined.

6. Generate a plot of the surface $u = x^2 + y^2$. Generate, too, a parametric plot of the unit sphere. Show these two figures on the same diagram.

■ 1.5 Data Fitting

Although *Mathematica* is not a dedicated statistical package, it does come with a large set of statistical capabilities. There is not nearly enough time to cover them all on this course, so we focus on one that colleagues find especially useful, namely regression and data fitting.

This is used where you have some data (from an experiment, say) and a *mathematical model* that you are pretty confident describes your data, but that contains some constants (known as *parameters*) whose values you do not know but wish to estimate. For example, suppose that we have the following data...

```
data1 = {{0.1, 1.33075}, {0.2, 1.40597}, {0.3, 1.70496},
  {0.4, 1.67045}, {0.5, 1.79961}, {0.6, 1.72460}, {0.7, 1.49204},
  {0.8, 1.36424}, {0.9, 1.14697}, {1.0, 0.815808}}
```

We can plot this data by typing

```
dataPlot1 = ListPlot[data1]
```

Suppose, too, that we believe the data to come from a model of the form $y = a + b\,x + c\,x^2$, but that we do not know the values of the constants a, b and c. Now, experimental data always has random errors associated with it. We'relooking, therefore, for an expression of the form $a + b\,x + c\,x^2$ that, while it is unlikely to fit the data perfectly, is the *best fit* available. *Mathematica* can generate this "best fit" expression:

```
bestFit1 = Fit[data1, {1, x, x^2}, x]
```

We can generate a plot of this function...

```
curvePlot1 = Plot[bestFit1, {x, 0, 1} ]
```

... and superimpose this on the original data:

```
Show[dataPlot1, curvePlot1]
```

■ Exercises 1.5

★ 1. Type out and test all the code in this section.

2. The data

```
data2 =
 {{0.3, 13.9112}, {0.6, 7.74621}, {0.9, 6.24733}, {1.2, 5.74747},
  {1.5, 5.61938}, {1.8, 5.819}, {2.1, 6.1611}, {2.4, 6.61437}}
```

is believed to come from a law of the form $y = a\,x + b / x$. Use the data to estimate the values of a and b, and generate a plot of the best fit curve superimposed on the data.

★ 3. *Mathematica* has a function called `MultipleListPlot` that is specially designed for plotting statistical data of this type. However, like many "specialist" commands it is only accessible if you load the library package that contains it, which you can do by typing

```
<< Graphics‘MultipleListPlot‘
```

Type in the two data sets

```
data3 = {{-2, -1.76596}, {-1, 1.37393}, {0, 2.88781},
  {1, 4.43359}, {2, 7.36544}, {3, 8.63811}, {4, 11.2147},
  {5, 13.158}, {6, 14.5677}, {7, 17.4601}}
```

and

```
data4 = {{-2, -2.2784}, {-1, 0.101452}, {0, 1.5432}, {1, 3.52656},
  {2, 4.83611}, {3, 5.79492}, {4, 8.68178}, {5, 11.0616},
  {6, 12.3439}, {7, 13.8627}}
```

Illustrate them both by typing

```
multPlot = MultipleListPlot[data3, data4]
```

You might need to enlarge the plot to see the distinct point symbols *Mathematica* has used to distinguish the two data sets.

Find the best fit straight lines for each data set and show graphs of them both on the same diagram as your points.

4. For additional statistical analysis (standard errors, *t*-statistics and so on) *Mathematica* has a function called **`Regress`**, that you can load by typing

```
<< Statistics‘LinearRegression‘
```

Try

```
Regress[data1, {1, x, x^2}, x]
```

You can find out more about this function using the help system if you need to.

★ 5. The **`Fit`** command works whenever the model we are trying to fit to the data has the form

$$y = a_0\ f_0(x) + a_1\ f_1(x) + a_2\ f_2(x) + ... + a_n\ f_n(x).$$

Fitting a model of this kind is called *linear fitting.* Although this covers a lot of situations, there are circumstances in which we can’t use it. For example, consider the data set

```
data5 = {{-0.3, 4.4743}, {-0.2, 4.6063}, {-0.1, 4.76847},
  {0, 4.97541}, {0.1, 5.23563}, {0.2, 5.5614}, {0.3, 5.9760}}
```

This is believed to come from a model of the form $y = a + e^{b x}$, which does not conform to the above. To get the best fit for this data we have to load the package

```
<< Statistics‘NonlinearFit‘
```

then type

```
NonlinearFit[data5, a + E^(b*x), x, {a, b}]
```

Try typing

```
NonlinearFit[data5, a + E^(b*x), x, {a, b}, ShowProgress -> True]
```

What do you think is going on?

■ 1.6 The Front End and the Kernel

Mathematica isn’t just one program: it’s two. When you click on the *Mathematica* icon, what gets loaded is the **Front End**: the part of *Mathematica* that handles things like screen display of input and output, printing and the creation of files. When you do your first calculation, the **Kernel** gets loaded: the “calculating engine” of *Mathematica*.

For this reason, the first calculation always seems to take a very long time. Many users get into the habit of kicking off with something innocuous like

```
2 + 2
```

then going on to more serious calculations once the Kernel is up and running.

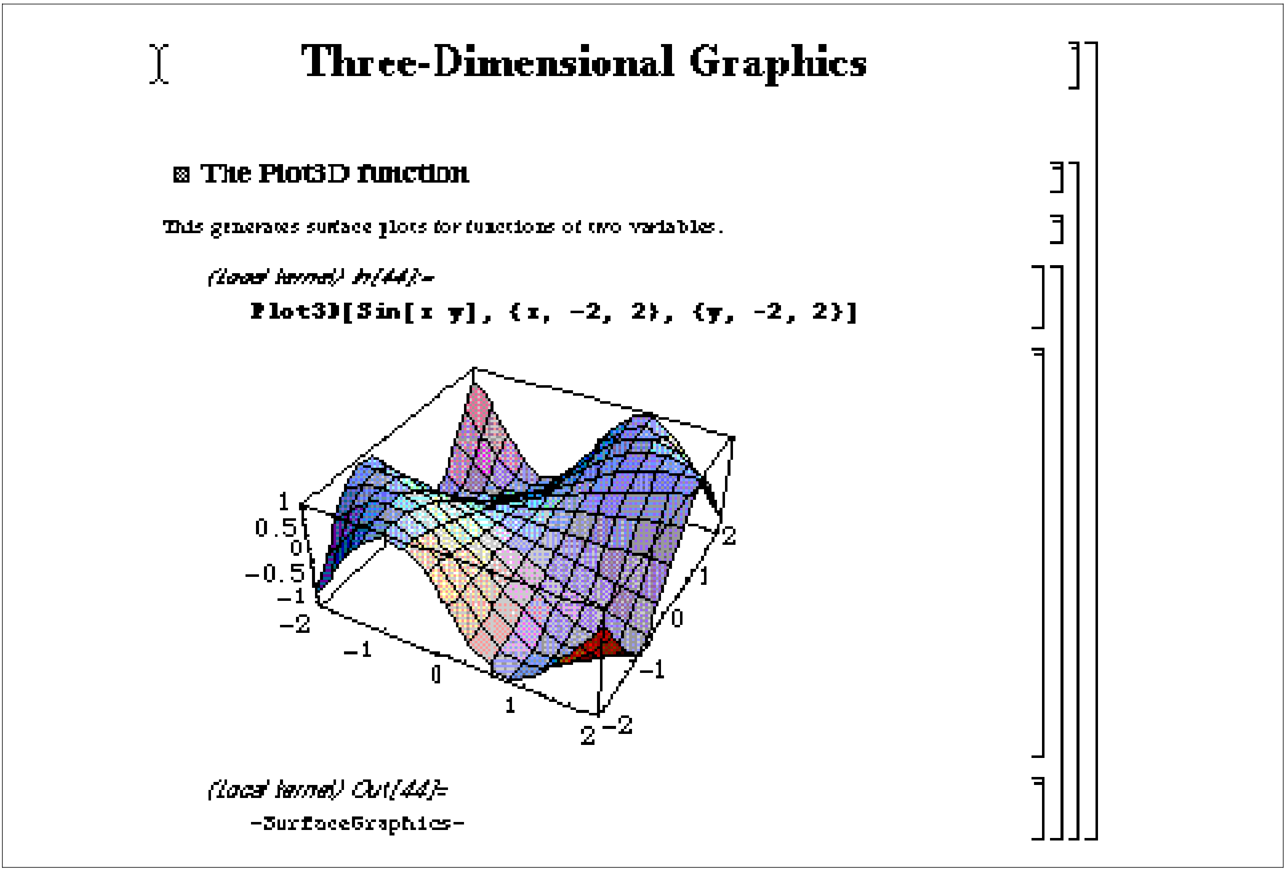
It’s possible to evaluate code that is already present, or to edit and re-evaluate: simply click in the text, edit if necessary, then “*shift-return*” in the normal way.

The detached relationship between the Front End and the Kernel is not very common for computer programs these days, and can take some getting used to. Those readers who used computers in the pre-micro days of the 1970s-80s will be familiar with the idea of typing locally, sending text down a wire to be processed by a mainframe computer elsewhere, and getting output back through the wire. Indeed, you could think of “shift-return” as a “send” instruction.

So, what happens when you press “shift-return” is the following: the text (*Mathematica* command) that you’ ve typed is sent to the Kernel as input, and it sends back the output from processing your command, which appears just underneath the input, and a “line number” which appears as a label for both input and output (the *In[. . .]* and *Out[. . .]*). It’s worth knowing that whereas the percent sign, `%`, means “the last output”, `%61`, say, means “output number 61” of the current Kernel session.

The Front End-Kernel split may at first sight seem to be no more than a complication. But it makes *Mathematica* very powerful and useful in a number of ways. First, the Front End and Kernel need not be located on the same machine, so you can use *Mathematica* in the comfortable environment of your personal PC or Macintosh whilst exploiting the computational power of a remote workstation. Instructions for setting up remote links like this depend on what platform (computer system) you’re using. Secondly, the Front End is designed to offer a sophisticated document interface, and this is claimed (by the developers, and with a certain amiunt of justice) to be of professional word processor quality. In the next section we describe the most basic features of “notebook” documents. We’ve labelled the section “advanced”, meaning that it’s not essential to go through it now, but it’s advisable to do so at some point.

■ *1.7 Advanced Topic: Notebooks*



Three-Dimensional Graphics

⊠ The Plot3D function

This generates surface plots for functions of two variables.

```
(local kernel) In[44]:=
    Plot3D[Sin[x y], {x, -2, 2}, {y, -2, 2}]
```

```
(local kernel) Out[44]=
    -SurfaceGraphics-
```

Figure 1: excerpt from a *Mathematica* notebook

As you work in *Mathematica*, a document is built up. Known as a **notebook**, this is a bit like a word processing document in a Windows or Mac application such as Word or WordPerfect: you can save it, you can select, cut and paste within it, you can mouse to different points in the text and edit in place, etc. But notebooks are in some ways more complex than word-processor documents, because of the different roles text can play. Often, perhaps most of the time, you'llsimply want to be typing code to evaluate. But you may also wish to add annotations or explanations, or to set up titles and section headings, and *Mathematica* needs a way of distinguishing these "inactive" forms of text both from one another and from "active" code. It does this by dividing the text in the notebook into disjoint **cells**, *marked by square brackets in the right margin*. In this way, you can build up complex documents of the type shown in Figure 1.

By default, *Mathematica* assumes that anything you type is code. To change that assumption, click on the cell bracket in the right margin and choose **Style** from the **Format** menu. You can then choose an appropriate style. If you wish to change the size, font or alignment associated with a cell style, or even within a particular cell, this is entirely possible.

You'll notice from Figure 1 that *brackets around brackets* exist, covering more than one cell. These define what are called **cell groups**. In all versions, a piece of input will be automatically grouped with its output, and later versions allow even more automatic grouping. To manually group a collection of cells (or of already existing cell groups), select all their cell brackets and choose **Cell Grouping** (or, in earlier versions,

Group Cells) from the **Cell** menu. There's no limit to the depth of the hierarchies you can build up in this way.

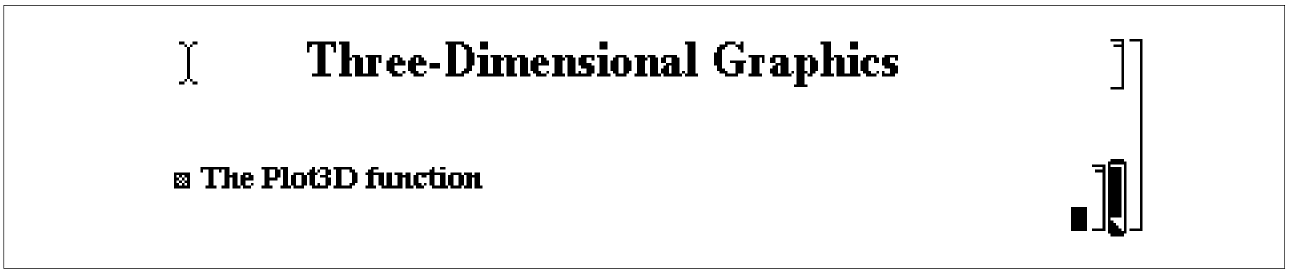


Figure 2: Excerpt from a *Mathematica* notebook, partially closed

A group can be **closed** by double-clicking on its grouping bracket; this hides all the cells except the first. It's often helpful to hide large collections of cells behind section headings: this allows documents to be skim-read for contents without scrolling right through them. A closed cell group can be **opened** by double-clicking on the grouping bracket (which is distinguishable by a small hook). Figure 2 shows a partially closed version of the notebook in Figure 1, with the grouping bracket selected.

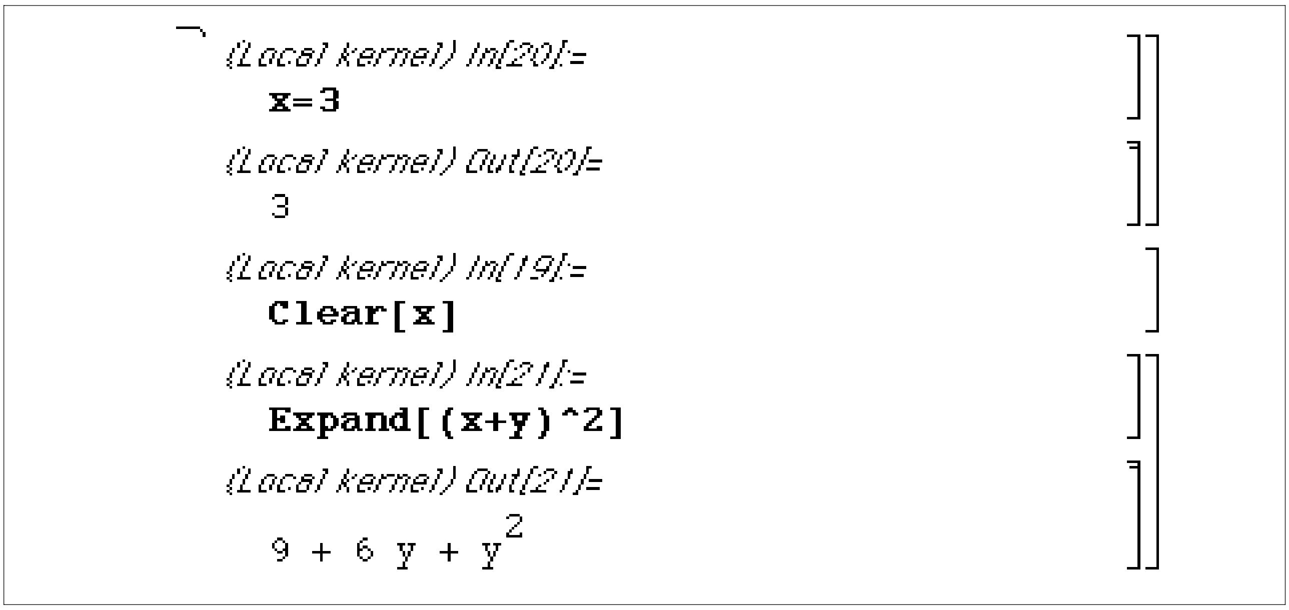
Notebooks, then, are complex documents. Their management is the task of the Front End, which therefore has to handle multiple types of text organised in complicated hierarchical ways. By contrast, all the Kernel does is keep a strictly *chronological* record of your calculations, whose ordering is reflected in the *In[. . .]* and *Out[. . .]* messages you see. So you have to be careful: just because a certain calculation comes last in the notebook doesn't mean it's the latest one as far as the Kernel's concerned.

■ *Exercises 1.7*

1. Start a fresh *Mathematica* notebook and reproduce Figure 1. By closing the appropriate cell group, reproduce Figure 2 as well.
2. Experiment with local style changes: reproduce a Text cell that looks more or less like this:

This *is* a **Text** cell, <u>but</u> one in which I have `experimented` with various **fonts**, sizes and ***<u>typefaces</u>***.

3. Experiment with *style sheets*: with (say) your "Figure 1" notebook on screen, find **Style Sheet** in the **Format** menu, and try several options.
4. Experiment with the various options under **Format → Screen Style Environment**
5. The following excerpt (Figure 3) from a *Mathematica* notebook seems to show something going wrong. Examine it carefully, and explain why the `Clear[x]` command doesn't seem to have worked. What has really happened here?



```
(Local kernel) In[20]:=
  x=3
(Local kernel) Out[20]=
  3
(Local kernel) In[19]:=
  Clear[x]
(Local kernel) In[21]:=
  Expand[(x+y)^2]
(Local kernel) Out[21]=
                2
  9 + 6 y + y
```

Figure 3: Excerpt from a *Mathematica* notebook

■ Session 2

In this session, we introduce the *expression*, the principal *Mathematica* data structure, and the *list*, one of its most useful manifestations. We examine the way *Mathematica* handles matrices and vectors. We look again at the idea of defining your own *Mathematica* functions, and explore different approaches to that task, focussing on the commands `Do`, `While`, `Map`, `Apply`, `Nest` and `FixedPoint`. There is optional material in Sections 2.5–2.7 on pure functions, local scoping of variables and recursive function definition.

As in Session 1, each section consists of a piece of text followed by some exercises. Exercises marked by a ⋆ are especially important or central, and you should do all of these if you get the time. The other exercises, while useful, are more peripheral and can be skipped if necessary.

■ 2.1 Lists and expressions

The standard way of storing multiple items of data in *Mathematica* is the **list**. An example might be `{1, x^2, 0.937, 3 + 2I, Factorial}` (admittedly a rather artificial one). Notice that there are no restrictions on the type of data you can hold in a list: here, for example, each data item is of a different type.

Defining your own lists is easy. You can, for example, type them in full, like this:

```
oddList = {1, 3, 5, 7, 9, 11, 13, 15, 17}
```

Alternatively, if (as here) the list elements correspond to a rule of some kind, the command `Table` can be used, like this:

```
oddList = Table[2 n + 1, {n, 0, 8}]
```

We can pick out, say, the 5th element in `oddList` by typing:

```
oddList[[5]]
```

(Note the double square brackets. This is a shorthand notation; the function we've used here is indexed in the manual under its "full" name, `Part`). We can generate a list containing the elements of `oddList` in reverse order by typing:

```
Reverse[oddList]
```

We can add new elements to lists by using `Append` or `Prepend`, as in:

```
Append[oddList, 19]
```

```
Prepend[oddList, -1]
```

Lists can be joined together:

```
Join[{-5, -3, -1}, oddList, {19, 21, 23}]
```

See the exercises for some more commands that are useful when handling lists.

Lists are important things, but there’s a sense in which there’s nothing very special about them: they’re simply an example of what’s called an **expression**. Internally, *Mathematica* represents `oddList` *not* the way we see it on the screen but like this:

```
List[1, 3, 5, 7, 9, 11, 13, 15, 17]
```

You can see this internal representation if you type

```
FullForm[oddList]
```

In a similar way, the internal representation of the equation

```
x == y == z
```

is

```
Equal[x, y, z]
```

So the essential structure of *equations* is exactly the same as that of *lists*. The same is true of virtually anything we can type into, or get out of, *Mathematica.* As it says in the manual: “everything is an expression” (section 2.1.1).

This enables us to use some of the commands you’ve just met on things that aren’t lists, as in:

```
eqn1 = (x == y == z)

eqn1[[3]]

Reverse[eqn1]

Append[eqn1, 0]

FullForm[eqn1]
```

Lists proper have a variety of uses in *Mathematica*. Most simply, they are a way of grouping together data we want to keep in one place, or refer to by one name. An example of a specialised use is to represent *vectors* referred to a Cartesian basis. Then the scalar product of two vectors is represented, as in standard mathematical notation, by a *dot*, as in:

```
{3, 0, -1} . {1, -2, 4}
```

Note, though, that *Mathematica*’s “scalar product” function has uses that go beyond vector work. Suppose, for example, we wished to generate a degree-8 polynomial whose coefficients corresponded to the elements of `oddList`. One way to do this is:

```
xPowers = Table[x^n, {n, 0, 8}]
oddList . xPowers
```

This is a good example of where *Mathematica*'s flexibility comes in handy. It doesn't matter at all that the elements of `oddList` are numeric whereas those of `xPowers` are symbolic expressions.

A *matrix* is represented as a *list of lists*, that is, as a list of the rows, each row itself a list. This is covered more fully in the exercises.

■ Exercises 2.1

★ 1. As above, define

```
oddList = Table[2 n + 1, {n, 0, 8}]
```

and try out the commands `[[...]]`, `Reverse`, `Append`, `Prepend` and `Join` on it in the ways suggested.

Repeat for

```
eqn1 = (x == y == z)
```

Try out the following too:

```
Length[oddList]

Delete[oddList, 4]

Insert[oddList, 100, 3]

ReplacePart[oddList, 100, 3]

RotateLeft[oddList]

RotateRight[oddList]

RotateLeft[oddList, 5]

Rest[oddList]

Take[oddList, 4]

Drop[oddList, 4]
```

Find out which of the above commands can sensibly be applied to `eqn1`, and how.

★ 2. Type the following to define a highly complex "list of lists of lists of lists":

```
multi =
{{{{a, b}, {c, d}}, {{e, f}, {g, h}}},
 {{{i, j}, {k, l}}, {{m, n}, {o, p}}}}
```

Test the effect of the following commands:

```
Flatten[multi]

Flatten[multi, 1]
```

```
Flatten[multi, 2]
```

Type

```
veryFlat = Flatten[multi]
```

and try out

```
Partition[veryFlat, 3]
```

```
Partition[veryFlat, 4]
```

etc. What does Partition do?

What combination of `Partition` commands will turn `veryFlat` back into `multi`?

3. Define two lists having some elements in common, such as:

```
list1 = {1, 2, 3, 4, 5, 6}
```

```
list2 = {5, 6, 7, 8, 9, 10}
```

Try out the following:

```
Union[list1, list2]
```

```
Intersection[list1, list2]
```

```
Complement[Union[list1, list2], list2]
```

4. Use *Mathematica* to find the cosine of the angle between the vectors

```
v1 = {1, 3, -1}
v2 = {2, -3, 0}
```

5. Define the matrices `mat1` and `mat2` like this:

```
mat1 = {{1, -2, 0}, {3, 5, -3}}
mat2 = {{1, 3, -4}, {0, 2, 1}, {-2, 0, 3}}
```

Try out the following:

```
MatrixForm[mat1]
```

```
mat1 . mat2
```

```
Transpose[mat1]
```

```
Det[mat2]
```

```
Inverse[mat2]
```

```
MatrixPower[mat2, 5]
```

```
NullSpace[mat1]
```

Use the last output to calculate the *rank* of `mat1`.

■ 2.2 One-time code versus reusable functions

The following is some *Mathematica* code for calculating the population variance of a large list of randomly generated data.

```
thisData = Table[Random[Real], {1000}];

(* sample size *)
n = Length[thisData];

(* calculate mean *)
 total = 0;
 Do[total = total + thisData[[i]], {i, 1, n}];
 mean = total/n;

(* sum of squares of deviations *)
 total = 0;
 Do[total = total + (thisData[[i]] - mean)^2, {i, 1, n}];

(* return the population variance *)
total/(n - 1)
```

Notice that in the first line we have suppressed the output by using a semicolon. This saves time, and the assignment still takes place. Notice, too, that any text between the symbols `(*` and `*)` is a **comment**, ignored by *Mathematica*. Finally, note the use of the “looping” command `Do` to make *Mathematica* perform the same operation several times.

This code works well enough as far as it goes. But what if we wanted to calculate the variance of several sets of data? It's *possible* to use the above code more than once (for example, by mousing back to the relevant bit of the notebook, or by using the Cut and Paste facilities you met in Session 1). But it is awkward to do so. A much better approach is to define a function which takes a data set as input and outputs the variance (*note: the following is all one command, so be sure not to "shift-return" until you get to the end!)*:

```
myVariance[data_List] :=
 (
  (* sample size *)
   n = Length[data];

  (* calculate mean *)
   total = 0;
   Do[total = total + data[[i]], {i, 1, n}];
   mean = total/n;

  (* sum of squares of deviations *)
   total = 0;
   Do[total = total + (data[[i]] - mean)^2, {i, 1, n}];

  (* return the population variance *)
   total/(n - 1)
 )
```

(the brackets will size themselves automatically).

Notice, again, the use of the underscore character to make the variable name `data` a placeholder, standing for any possible input. Three syntactical features of this definition need special remark.

(1) The use of the word `List` after the underscore is a **type declaration**; it instructs *Mathematica* to expect `data` to be in the form of a list (if it isn't, *Mathematica* will return the expression unevaluated).

(2) The use of semicolons: previously, we've introduced the semicolon as a means of *suppressing output*, but here its role is rather different. Put simply, when a function consists of more than one command, the commands *must* be separated by semicolons. It's as though the commands are being "strung together" into a single larger command with just one final output.

(3) The reason for the outermost pair of parentheses `(...)` is not obvious. What they are doing is to group all the function code on the right-hand side of the `:=`.

The `myVariance` function can be used like this:

```
thisData = Table[Random[Real], {1000}];
myVariance[thisData]
```

```
thatData = Table[Random[Real, 100], {500}];
myVariance[thatData]
```

It'sgood to get into the habit of incorporating code you intend to reuse into function definitions. In long, complex chunks of code, this has the added advantage of *making dependencies explicit,* allowing you to keep track of what quantities depend on what other quantities.

■ Exercises 2.2

★ 1. Type in, and test, the two versions of the variance code above.

2. Write a *Mathematica* function called `myMax`, which takes as its argument a list of numbers, and returns the maximum number in the list. Test this function.

★ 3. Write a function called `tangent`. Your function definition should begin

```
tangent[expression_, x_, a_] :=
```

The function should return an expression in `x` corresponding to the tangent to the graph of `expression` at the point `x` = `a`.

4. The following is an attempt to write two functions: `myMean`, which calculates the mean of a list of data, and `myVariance2`, which calculates the variance of the data and calls `myMean` in the process.

```
myMean[data_List] :=
 (n = Length[data];
  total = 0;
  Do[total = total + data[[i]], {i, 1, n}];
  total
  ─────
    n  )
myVariance2[data_List] :=
 (n = Length[data];
  total = 0;
  Do[total = total + (data[[i]] - myMean[data])^2, {i, 1, n}];
  total
  ─────
  n - 1 )
```

Both of the functions work, but one of them is very inefficiently written. Identify the inefficient one, and alter the code so that `myVariance2` still calls `myMean`, but the inefficiency has been removed.

■ 2.3 Apply and Map

The various pieces of variance code that you met in the last section all work upon *lists* of data. Although they differ in their details, what they have in common is that they all use the `Do` command to pick out elements of the list in turn and perform actions of some kind using them or upon them.

But this turns out to be a pretty inefficient way of dealing with lists in *Mathematica*: by using "whole list" operations we can often improve execution time appreciably (by a factor of 2 or 3), and have more compact, elegant code. The keys to handling lists, and indeed expressions in general, as *single entities* (rather than picking them apart and handling their elements separately) are the commands `Apply` and `Map`.

For example, suppose we want to add all the numbers in the list

```
oddList = {1, 3, 5, 7, 9, 11, 13, 15, 17}
```

One way of doing this is to do what we did in the last section, namely

```
total = 0;
Do[total = total + oddList[[i]], {i, 1, 9}];
total
```

Or we might try:

```
Sum[oddList[[i]], {i, 1, 9}]
```

or

$$\sum_{i=1}^{9} \texttt{oddList[[i]]}$$

But the most efficient approach of all is this:

```
Apply[Plus, oddList]
```

This simply generates the expression `Plus[1,3,5,7,9,11,13,17]`, and then evaluates it. `Apply` replaces the *head* of the expression `oddList`, namely `List`, with `Plus`.

`Apply` is the best thing to use, then, when you have a list of things that you want to *combine* in some way. But what if you have a list of things that you want to treat *separately*, doing the same thing to each? For example, suppose we want to generate a list called `squareList` consisting of all the squares of the elements of `oddList`, in order. One way to do this is to use the `Do` command, like we did in the last section:

```
squareList = {};
Do[squareList = Append[squareList, oddList[[i]]^2], {i, 1, 9}];
squareList
```

More elegantly, we might try:

```
squareList = Table[oddList[[i]]^2, {i, 1, 9}]
```

But it’s most efficient of all to do this:

```
square[x_] := x^2
squareList = Map[square, oddList]
```

This sets up a function called `square` which is then applied to each of the elements of `oddList` separately. Here’s a more complex example, where a “magnitude” function is mapped over a list of coordinate pairs:

```
ptsList = Table[{Random[], Random[]}, {10}]
mag[{x_, y_}] := √(x² + y²)
Map[mag, ptsList]
```

Using `Map` gives us a way of processing each of the elements in a list “simultaneously”, without needing to trawl through the list element by element.

■ Exercises 2.3

★ 1. Type in each of the three sets of code for adding all the elements of a list. Test them on the list

```
bigOddList = Table[2 n + 1, {n, 0, 9999}];
```

Type in each of the three sets of code for squaring the elements of a list. Test them on `bigOddList`.

`bigOddList` is sufficiently large that the differences in execution time are noticeable for the different codes. However for precise comparison *Mathematica* provides the `Timing` command:

```
Timing[Apply[Plus, bigOddList]]

Timing[Sum[bigOddList[[i]], {i, 1, 10000}]]
```

2. Write a function called `derivativeList`, which takes as its arguments a list of expressions in `x`, and returns a list consisting of the derivatives of the expressions with respect to `x`. Thus, the input

```
derivativeList[{x², Log[x], Cos[x]}]
```

should return

$\left\{2\,x,\ \frac{1}{x},\ -\mathrm{Sin}[x]\right\}$

★ 3. Write a function called `myVariance3`, which calculates the variance of a list of data. This time, your function should make use of `Map` and `Apply`, instead of using `Do` to iterate through the list. Use the `Timing` command and the lists `thisData` and `thatData` from Section 2.2 to compare the execution time of your new function with `myVariance` and `myVariance2`.

4. Write a function called `squareBothSides`, which squares both sides of an equation. Thus

```
squareBothSides[x - y == 4]
```

should return

$(x - y)^2 == 16$

(Remember that we can treat all *Mathematica* expressions like lists.) Go on to write a function called `doToBothSides`, which performs *any* given operation on both sides of an equation. Thus

```
doToBothSides[Sin, x - y == π / 2]
```

should return

```
Sin[x - y] == 1
```

■ 2.4 Nesting functions

Clearly `Do` is a useful *Mathematica* command. As you've seen, we can use it to process lists of data, either by combining all the data elements or by doing the same thing to each one. However, as you've also seen, there are more efficient and more economical ways of doing both those things.

Another use of the `Do` command is when we want to apply the same operation repeatedly to one piece of data, as in this implementation of a (well-known, and rather inefficient) iterative algorithm for finding the square root of 5.0:

```
sqrt5Iterate1[n_] :=
 (x = 0.0;
  Do[x = (x + 5.0)/(x + 1.0), {n}];
  x)
```

But even in this case there's an alternative. It involves no great saving in execution time, but it is, perhaps, a little more economical and elegant as far as the code is concerned. This is it:

```
g[x_] := (x + 5.0)/(x + 1.0);
sqrt5Iterate2[n_] := Nest[g, 0.0, n]
```

What's happened is that a function `g` has been defined, and the *Mathematica* command `Nest` applies `g` repeatedly, using a starting value of 0.0. This has exactly the same effect as before, but avoids the use of `Do`. Notice that `Nest` automatically *returns* the final value as output. `Do`, by contrast, doesn't return anything: it has no output. That is why we have to finish `sqrt5Iterate1` by explicitly calling the value of `x`.

Perhaps it's best to see `Do` as a utility, "multi-purpose" command, and things like `Table`, `Map`, `Apply` and `Nest` as more finely-tuned, specialised tools. It's rare that an application of `Do` will be the best way to get what you want out of *Mathematica*, but it may often be the first that springs to mind.

Suppose that we want to carry out our square root iteration not for a fixed and predicted number of steps but for as many steps as necessary until it has converged. One way to do this is using the `While` command, as follows:

```
x = 5.0;
tmp = 0.0;
While[TrueQ[x =!= tmp], (* while x is not equal to tmp *)
 (tmp = x; x = (x + 5.0)/(x + 1.0))]; (* iterate *)
x
```

The iteration is carried out until the condition “the current and previous iterates are different (within the precision of machine accuracy)” ceases to be true. “Machine accuracy” is, as the name suggests, machine-dependent and depends on the quality of the arithmetical processing hardware. On most machines it’s around 16 digits.

This kind of iteration, too, can be more elegantly done. The command `FixedPoint` is exactly like `Nest`, except that it returns not the *n*th iterate but the *final* one after convergence has been established. To get our iterative approximation to $\sqrt{5}$, all we need to do is type:

```
g[x_] := (x + 5.0)/(x + 1.0)
FixedPoint[g, 0.0]
```

Some iterative algorithms may take a long time to converge to machine accuracy, but may converge perfectly satisfactorily for practical purposes long before that. We can still use `FixedPoint`; all we have to do is change the `SameTest` option, like this:

```
g[x_] := (x + 5.0)/(x + 1.0)
prettyDamnCloseQ[x_, y_] := TrueQ[Abs[x - y] < 0.0001]
FixedPoint[g, 0.0, SameTest -> prettyDamnCloseQ]
```

■ Exercises 2.4

★ 1. Test out the `sqrt5Iterate1` and `sqrt5Iterate2` functions. Examine the effect of replacing the `Nest` command with `NestList`.

Compare, too, the two approaches to conditional stopping: the one that uses `While` and the one that uses `FixedPoint`. Examine the effect of replacing `FixedPoint` with `FixedPointList`.

★ 2. Write a function called `sqrtIterate` such that `sqrtIterate[a, n]` will return the *n*th iterated square root approximation for *any* number *a*, again using the algorithm

$$x \to \frac{x+a}{x+1.0}$$

with starting value 0.0. Use `Nest` rather than `Do`.

Write a function called `sqrtApprox` such that `sqrtApprox[a]` returns, as a decimal approximation to $\sqrt{a}$, the final iterate of the above algorithm once convergence has occurred. Use `FixedPoint` rather than `While`.

3. Write a function called `derivs` which takes as its argument an expression, a variable name and an integer *n*, and returns a list containing the expression, its first derivative, its second derivative, and so on up to its *n*th derivative, all with respect to the variable. Thus

   ```
   derivs[Log[1 + x], x, 3]
   ```

   should return

   $$\left\{\text{Log}[1+x],\ \frac{1}{1+x},\ -\frac{1}{(1+x)^2},\ \frac{2}{(1+x)^3}\right\}$$

   Write a function called `mySeries` (calling `derivs`) such that

   ```
   mySeries[expr, {x, x0, n}]
   ```

   returns the Taylor series for `expr` about `x` = `x0` up to the term in `x^n`.

■ *2.5 Advanced Topic: Pure functions*

In Section 2.3 you met the following piece of code:

```
square[x_] := x^2
squareList = Map[square, oddList]
```

This is a pretty efficient way of squaring all the elements of `oddList`, but not the most efficient—quite. What's wasteful about it is that we seem to have to define a whole new *Mathematica* function, `square`: that's because the first argument of `Map` must always be a the name of a function.

Well, it's not quite true that `Map` demands the *name* of a function as its first argument. It will also accept what's known as a **pure function**, and it's this that gives us a way round the problem. The same piece of code, recast in pure function form, looks like this:

```
squareList = Map[(#^2)&, oddList]
```

Instead of going to the trouble of defining the `square` function separately, we've used the rather odd-looking expression `(#`^{2}`)&`. This is an example of a pure function; its key characteristics are the following:

- the use of the *hash mark*, `#`, to stand for the argument of the function;
- the use of the *ampersand*, `&`, at the very end of the expression to signify that it is a pure function.

Here's the "magnitude" example from Section 2.3 rewritten in pure function form:

$$\texttt{Map}\left[\sqrt{\#[[1]]^2+\#[[2]]^2}\ \&,\ \texttt{ptsList}\right]$$

This uses the `Part` command, written in short form as `[[...]]`, to extract coordinate values.

Pure functions behave exactly like ordinary functions, and can be used in the same ways—as inputs to `Apply` and `Nest`, for example.

Functions of more than one variable can also be dealt with in this way, using the symbols `#1`, `#2`, etc. As the notation suggests, you can think of these symbols as "argument number 1", "argument number 2", ... Here's an example:

```
Apply[√(#1² + #2² + #3²)&, {a, b, c}]
```

The iteration code (from Section 2.4)

```
g[x_] := (x + 5.0)/(x + 1.0)
prettyDamnCloseQ[x_, y_] := TrueQ[Abs[x - y] < 0.0001]
FixedPoint[g, 0.0, SameTest -> prettyDamnCloseQ]
```

could have been written like this:

```
FixedPoint[(# + 5.0)/(# + 1.0)&, 0.0,
 SameTest -> (TrueQ[Abs[#1 - #2] < 0.0001]&) ]
```

Here, both the iterated numerical function `g` and the convergence test `prettyDamnCloseQ` have been replaced by pure functions, the latter being a function of two variables.

■ *Exercises 2.5*

1. Test the code in this section.
2. Rewrite the functions `myVariance3` and `squareBothSides` (both from Exercises 2.3) and `sqrtIterate` and `sqrtApprox` (from Exercises 2.4) so that they use pure functions.
3. Write a function `magnitude` that calculates the magnitude (square root of the sum of the squares of the elements) of a list of any length.

■ *2.6 Advanced Topic: local scoping*

By default, variables and constants in *Mathematica* are **global**. For example, consider the following function definition, which makes use of the formula

$$s^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)$$

```
neatVariance1[data_] := (
  n = Length[data];
  1/(n - 1) (Apply[Plus, Map[(#²)&, data]] - 1/n (Apply[Plus, data]²))
  )
```

This is probably the most efficient form of the variance code so far, but it does have a problem, and one that you'll find in many of the functions you've met in this section. Every time the function is called, a value gets attached to the symbol `n`. This value is *global*: it applies outside the context of the function, and will be used every time the symbol `n` occurs subsequently, until `n` is reassigned or cleared or until the session is terminated.

This can cause serious problems: interference between different functions and so on. Unless you specifically want your function to make global assignments like this, it's best to override the default and make `n` **local**. We do that by means of the command `With`, as follows:

```
neatVariance2[data_] :=
  With[{n = Length[data]},
   1/(n - 1) (Apply[Plus, Map[(#^2)&, data]] - 1/n (Apply[Plus, data]^2))
  ]
```

This tells *Mathematica* to replace all occurrences of the symbol `n` with `Length[data]` *where those occurrences are inside the function*, but to leave `n` unassigned, or assigned as it was before, elsewhere. As rewritten, the function will neither *change* any already existing global value of `n` nor *use* that value during the calculation.

There are two sets of circumstances in which local scoping is important but `With` won't work. One is when we want to make *changes* to the value of a symbol during the execution of the function: in other words, when the symbol represents a *variable* rather than a *constant*. The other is when we want the symbol not to have a value: when we want it to be purely symbolic. In either case, we can use the `Module` command.

Here's an example of the first case, in which the value attached to the symbol changes during the execution of the function. Consider the following code, which you first met in Section 2.4:

```
sqrt5Iterate1[n_] :=
  (x = 0.0;
   Do[x = (x + 5.0)/(x + 1.0), {n}];
    x)
```

In order that values of `x` assigned during the execution of this function should not clash with other occurrences of that symbol, this should really be rewritten

```
sqrt5Iterate1[n_] :=
  Module[{x = 0.0},
   Do[x = (x + 5.0)/(x + 1.0), {n}];
    x]
```

Here, 0.0 is used merely as the *initial* value of `x`. (Of course, you'll recall that this whole function can be rewritten using `Nest`!)
For an example of the "purely symbolic" case, consider the following code for calculating derivatives from first principles:

```
firstPrincD1[expr_, x_] := Limit[(expr /. (x -> x + h) - expr)/h, h -> 0]
```

This works fine provided `h` hasn't already been given a global value; if it has, the code fails badly. To make it work even in those circumstances, it should be written like this:

```
firstPrincD2[expr_, x_] :=
  Module[{h}, Limit[(expr /. (x -> x + h) - expr)/h, h -> 0]]
```

■ ***Exercises 2.6***

1. Test the `neatVariance1` function on some suitable randomly generated data. Now type

   ```
   n
   ```

   and comment on what you observe. Clear any global value for `n` by typing

   ```
   Clear[n]
   ```

   and then test `neatVariance2`. Type

   ```
   n
   ```

   again, and comment on what you find.

2. Test both forms of the `sqrtIterate` function in the same way.

3. Begin by making sure that no value is attached to the symbol `h` by typing

   ```
   Clear[h]
   ```

   Now test the `firstPrincD1` function on some suitable expression. Test it again, but this time assign a value to `h` first by typing

   ```
   h = 0.3
   ```

   Clear the symbol `h` and test the `firstPrincD2` code in the same way.

4. Review your code from earlier exercises in this session. Rewrite some of it with local scoping of variables and constants where appropriate.

■ *2.7 Advanced Topic: iteration and recursion*

Suppose we wanted to rewrite *Mathematica*'s `Factorial` function from scratch— for non-negative integers, at any rate. One way would be the following:

```
myFactorial1[n_] := Product[i, {i, 1, n}]
```

or

$$\texttt{myFactorial1[n\_] :=} \prod_{i=1}^{n} i$$

This is an example of **iterative** code: the value of `i` is made to increase sequentially, and the product is built up as it does so .

But there is another possible approach. Consider the following code:

```
myFactorial2[0] = 1;
myFactorial2[n_] := n * myFactorial2[n - 1]
```

This may look like a circular definition. However, it is rescued from circularity by two things: the fact that $n - 1$ is less than n and the fact that a simple *non-circular* definition exists for one case, namely $n = 0$.

When functions call themselves in this way, we have what’s called **recursion**. Many problem situations present us with a choice between iteration and recursion, and some seem tailor-made for the latter.

Recursion, though it’s often elegant and pleasing, is rather inefficient in many computer languages and impossible in some (older dialects of Fortran, for example). In *Mathematica*, though, it usually works rather well.

■ *Exercises 2.7*

1. Test both forms of the factorial code in this section. Test, also, the following:

   ```
   myFactorial3[0] = 1;
   myFactorial3[n_] := myFactorial3[n] = n * myFactorial3[n - 1]
   ```

   Can you explain what’s going on here? What are the advantages and disadvantages of this approach?

   (*Hint*: when you've done some preliminary testing of each, compare the outputs of `??myFactorial2` and `??myFactorial3`.)

2. Write and test a recursive function for the Fibonacci sequence

   1, 1, 2, 3, 5, 8, ...

   (Each term of this sequence is generated by adding the previous two.) Thus

   ```
   fibonacci[6]
   ```

   should return the 6th term in the sequence, and so on.

3. Write a recursive function called `myDet`, which must not call *Mathematica*’s `Det` function, for calculating matrix determinants by cofactor expansion.

■ Session 3

In this session, there are three things you can opt to do.

Option 1: a major programming exercise with the idea of putting into practice the *Mathematica* you've been learning in Sessions 1 and 2. This forms Section 3.1

Option 2: one or more of the various Case Studies which are available separately: a list of these forms Section 3.2.

Option 3: *Mathematica* surgery. You bring along any problems in your own work for which *Mathematica* might be useful, and we'll try to help you implement *Mathematica* appropriately.

In addition, we have described a number of the Internet resources available to *Mathematica* users. These form Section 3.3.

■ 3.1 Extended exercise: the Logistic Map

The logistic map is one of the simplest, and most famous, of nonlinear dynamical systems. We won't cover any of the theory here (which has been described in a vast number of books, articles and backs of breakfast cereal packets) beyond mentioning a few interesting things to look at.

We are interested in the map

$$x \to ax(1-x)$$

which is equivalent to the iterative equation:

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

The parameter range of interest is $0 \le a \le 4$, because for that range if $0 \le x_0 \le 1$ then $0 \le x_n \le 1$ for all n.

Before starting, you should read Section 2.5 on "pure functions": it may help you to program more elegantly.

1. Write a function called `logisticIterates` that outputs a list containing the iterates from 0 to n of $x_{n+1} = ax_n(1-x_n)$, starting from a suitable x_0. x_0, a and n should be inputs to the function, and thus

   ```
   logisticIterates[0.3, 3.1, 4]
   ```

   should return

   ```
   {0.3, 0.651, 0.704317, 0.645589, 0.709292}
   ```

   Generate a list of the iterates from 0 to 21 of the logistic map with $x_0 = 0.1$, $a = 2.75$. Use `ListPlot` with appropriate option settings to generate a "time series" plot of the type shown in Figure 4.

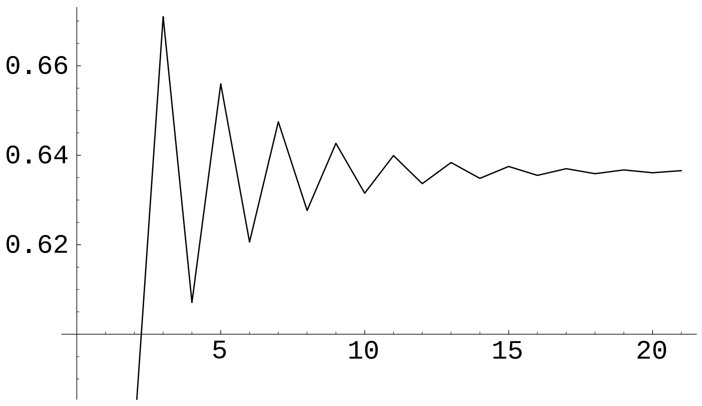


Figure 4: Time series for the Logistic Map, x_n versus n. $a = 2.75$, $x_0 = 0.1$.

Write a function called, say, `tsPlot`, such that

```
  tsPlot[0.1, 2.75, 21]
```

generates the above figure automatically.

Investigate the behaviour of the logistic map for different values of a, focussing on the intervals

$$0 \le a \le 1;$$
$$1 \le a \le 2;$$
$$2 \le a \le 3;$$
$$3 \le a \le 3.569946;$$
$$3.569946 \le a \le 4.$$

2. When looking at convergent behaviour such as that described in the table above it's helpful to discard the early iterates: write an "attractor" function that takes $x0$, a, n and m as inputs, and outputs a list of m iterates beginning with the nth (so it has to calculate, but not output, iterates 1 up to n–1). Write another time series function to plot the attractor data.

3. Write a function to produce bifurcation diagrams for the logistic map (a bifurcation diagram is a plot of the limit points—long-term x-values—of the map against the parameter a). This is a non-trivial programming task; it may be helpful to recall the way the `Flatten` function works (see Exercises 2.1).

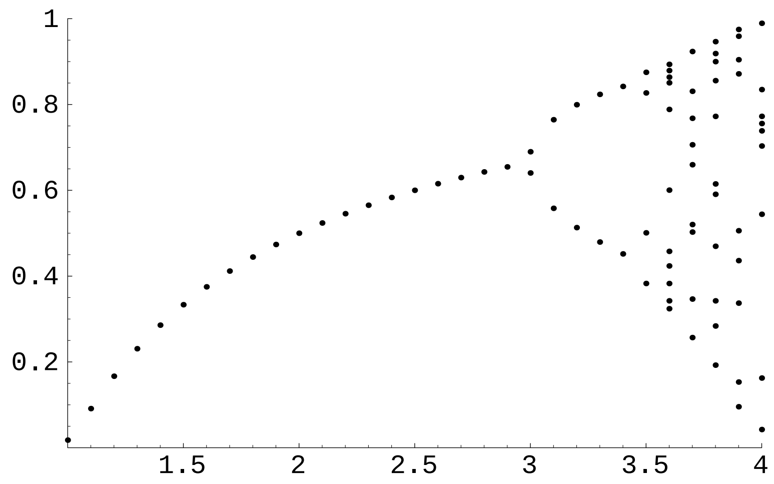


Figure 5: Bifurcation diagram generated with *Mathematica*: the interval between values of a is 0.1; for the attractor the first 40 iterates are discarded, 10 retained.

4. Write a function to produce *cobweb diagrams* for the logistic map, such as the one on the front cover of this booklet. For this task, which is again non-trivial, you'll need to think about what needs to be done to your `logisticIterates` data so that the appropriate coordinate points are generated in the right order.

   Further ideas you might explore: Generalize the code you have written so far so that it can be applied to any map, not just the logistic map. Use this new code to investigate the *complex* map

$$z \to z^2 + c,$$

   where c is a complex parameter and $z_0 = 0$. Write *Mathematica* code for plotting the Julia Sets and the Mandelbrot Set in the Argand Diagram.

■ 3.2 Case studies

Separate from this booklet we have prepared a number of Case Studies, or "common tasks for the academic user of *Mathematica*". Here are the current titles:

1. Handling experimental data.

2. Animations and movies.

3. Contour and surface plotting.

4. Moving data between *Mathematica* and Excel.

5. Exporting graphics to other applications.

6. Equations and rules.

They're accessible on the WWW at:

`http://metric.ma.ic.ac.uk/mathematica/studies`

■ 3.3 Resources for *Mathematica* users on the Internet

Besides numerous books about *Mathematica*, there is a great deal of information freely available, and an active user community, on the Internet. For these links go to the WWW page at

`http://metric.ma.ic.ac.uk/mathematica/resources.html`

The central information point for *Mathematica* is the WWW server at Wolfram Research Inc. (Champaign, Illinois, USA), creators of the *Mathematica* system:

`http://www.wolfram.com`

It's a good idea to browse around that site, check out the latest books and order your *Mathematica* baseball caps, T-shirts and coffee mugs. You can even interact with a *Mathematica* program over the Web, also known as "The Integrator":

`http://www.integrals.com`

Maintained on the Wolfram server is a very large collection called *MathSource* of *Mathematica* programs and notebooks contributed by *Mathematica* users world-wide:

`http://www.wolfram.com/mathsource`

If you have a specialist interest you may well find some useful stuff there. Another useful section is the Technical Support Frequently Asked Questions (FAQ):

`http://www.wolfram.com/FAQs`

There are two useful news/email discussion groups for *Mathematica* users: *MathGroup* is dedicated to *Mathematica*, and is accessible via both email and the newsgroup `comp.soft-sys.math.mathematica`.

For details see:

`http://smc.vnet.net/MathGroup.html`

MathGroup is moderated, which means less junk than usual and its archives are available online:

`http://www.wolfram.com/mathgroup`

Also there is a (prototype) search engine for the archive at:

`http://smc.vnet.net/mathgroupsearch.html`

The newsgroup `sci.math.symbolic` carries discussions about *Maple, Mathematica* and other symbolic software. It isn't moderated and you get what you pay for: for example it is a host to frequent discussions of that burning question: "which program is best?". Finally, remember that with the Web, often the best way to track something down is to use a search engine. Digital's "AltaVista" is currently one of the best around:

`http://www.altavista.digital.com`

Last but not least, do visit the WWW server of the METRIC project:

`http://metric.ma.ic.ac.uk`

where we'll be putting news of, for example, future *Mathematica* training courses.