Mathematical Methods for Physics using Maple

July 10, 2003

Introduction

Contents

1 About the course

One mathematical technique that is becoming increasingly important is the use of computers to perform *algebraic* calculations. This course will present some of the main mathematical techniques required in theoretical physics and explain how to apply them using the mathematical computation system Maple, which provides algebraic, symbolic, numerical and graphical facilities. It will also include some physical applications. It will therefore not be possible to give as much mathematical rigour or background as would be usual in a course on mathematical methods but instead there will be a significant practical component and you will be expected to use Maple yourself to solve problems, guided by the examples discussed during the lectures.

Some experience with computers and with programming in a language such as Pascal or C will be an advantage, but is not essential.

This week consists of a general introduction to Maple. In subsequent weeks I will focus on a particular mathematical topic and discuss in more detail the relevant Maple facilities. The mathematics will be based mainly on "Methods of Mathematical Physics" by Jeffreys and Jeffreys (Cambridge, 1956) and "Mathematical Methods of Physics" by Mathews and Walker (Benjamin, 1965), but some specialized topics (such as differential geometry and distribution theory) will also use other sources. The idea is that the presentation of computer algebra will be driven by the mathematics, but I will also briefly include some of the main mathematical background that does not relate to direct computation.

The next section of these notes consists of two worked examples to show Maple in action. The rest provides a brief introduction to Maple, in a way that I hope will be useful for later reference. I have tried to focus on the basic aspects, beyond which one can begin to use Maple's online help facility. It is in order to make these notes easier to use for reference that I have included a table of contents.

At the end of each set of notes will be some suggested exercises, and it is essential that you work through at least some of them using Maple. You can use any Maple system to which you have access, at any time, but versions of Maple earlier than version V may not be adequate for all the examples. We intend to provide access to Maple, and a supervised practical session, for those who wish to use it. There will be some informal continuous assessment of the exercises, although it is difficult to continuously assess such a course in a formal way. The main assessment will consist of one or two written examinations which will test your understanding of the mathematics and physics, and require you to write short programs in Maple, based on the weekly exercises. This should encourage you to take the exercises seriously!

2 Worked examples from elementary mechanics

The following examples involve only very simple physics and mathematics, so that we can concentrate on the use of Maple.

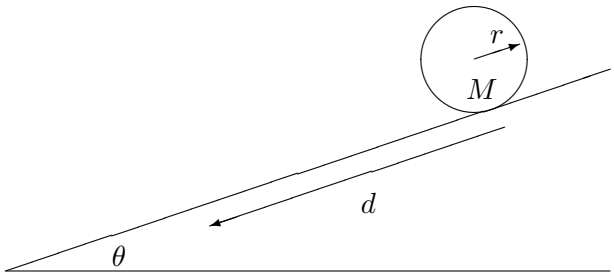
The Maple output that I will show in the notes will be in the standard text mode used by the Maple editor from within the review browser, which has the advantage that it can be imported into a text processor as text rather than graphics. The actual screen display on many systems (includ-

ing MS-DOS/MS-Windows) is prettier than Maple's textual simulation of mathematical notation as shown here. I will also reduce the screen width used by Maple (from 79 to 59 characters) so as to fit the text width of the notes. Text in "typewriter" font shows actual Maple input, preceded by the Maple prompt "`>`", and followed by the output that it produced. Rather than import screen images of plots into the notes, I will instead print (some) plots independently and include them as unnumbered pages.

I will explain more of the details as the course progresses. However, in the following examples I use *equations* (denoted by the equal sign "`=`") quite a lot, and an equation is treated by Maple as a particular kind of algebraic expression. A statement of the form "*name* `:=` *expression*" is an *assignment* and *not an equation*; it means "in future, use the name to represent the expression", where the expression named might in fact be an equation. I find it a useful technique in computer algebra to work with such "named equations". The symbol `%` in Maple represents the last value computed, `%%` represents the one before last and `%%%` the one before that.

2.1 Starting the ball rolling!

Let us consider the following problem. A cylinder of mass M and radius r rolls without slipping down a plane slope making an angle θ to the horizontal, as shown in the figure below. Find the distance d that it has travelled, its velocity v and acceleration a at a time t after it has been released from rest, and plot a graph of them.



I will solve this problem by using conservation of energy. The linear kinetic energy is

```
> LKE := 1/2*M*v^2;
                                    2
                        LKE := 1/2 M v
```

and the angular kinetic energy is

```
> AKE := 1/2*MI*w^2;
```

$$AKE := 1/2\ MI\ w^2$$

where `MI` is the moment of inertia, and `w` (representing ω) is the angular velocity, which is given by

```
> w := v/r;
```

$$w := v/r$$

With this value for ω, the angular kinetic energy is

```
> AKE;
```

$$\frac{1}{2}\,\frac{MI\ v^2}{r^2}$$

The gravitational potential energy is

```
> PE := M*g*h;
```

$$PE := M\ g\ h$$

where g represents the gravitational acceleration and h the height of the cylinder. Then, starting from rest at height $h = h_0$, conservation of energy leads to the equation

```
> Energy_eqn := PE + LKE + AKE = subs(h=h0, PE);
```

$$Energy\_eqn := M\ g\ h + 1/2\ M\ v^2 + 1/2\ \frac{MI\ v^2}{r^2} = M\ g\ h0$$

In terms of the distance d as a function of time t, represented by the Maple function `d(t)`:

```
> h := h0 - d(t)*sin(theta);
```

$$h := h0 - d(t)\ \sin(theta)$$

```
> v := diff(d(t), t);
```

$$v := \frac{d}{dt} d(t)$$

```
> Energy_eqn := Energy_eqn;
  Energy_eqn := M g (h0 - d(t) sin(theta))
```

$$+ \frac{1}{2} M \left(\frac{d}{dt} d(t)\right)^2 + \frac{1}{2} \frac{MI \left(\frac{d}{dt} d(t)\right)^2}{r^2} = M g h0$$

Note that Maple does not automatically simplify this expression by collecting like terms. Before doing so explicitly, let us first compute the moment of inertia.

The moment of inertia of a *tube* of mass density m per unit cross-sectional area, radius x and thickness dx is

```
> (2*Pi*x * dx * m) * x^2;
```

$$2 \pi x^3 dx m$$

Hence the moment of inertia of a *solid* cylinder of radius r is

```
> MI := Int(2*Pi*m*x**3, x = 0..r);
```

$$MI := \int_0^r 2 \pi m x^3 dx$$

I have chosen an "inert" form of integral that remains symbolic in order to check it, so now I will load the `value` function to evaluate inert forms, and apply it to get

```
> with(student, value):
```

```
> MI := value(MI);
                                        4
                        MI := 1/2 Pi m r
```

Similarly, the total mass of the cylinder is

```
> M = Int(2*Pi*m*x, x = 0..r);
                              r
                              /
                             |
                        M =  |  2 Pi m x dx
                             |
                            /
                            0

> value(");
                                   2
                             M = Pi m r
```

Note that I have expressed this as an equation, rather than an assignment, so that I can express m in terms of M by solving the equation:

```
> m := solve(", m);
                                    M
                             m := -----
                                     2
                                  Pi r
```

Re-evaluating `MI` to substitute the new value for m gives the final result:

```
> MI := MI;
                                      2
                           MI := 1/2 M r
```

Now let us simplify the ODE (the energy equation) to see what it looks like, using all the new values for the quantities on which it depends:

```
> Energy_eqn := simplify(Energy_eqn);
Energy_eqn :=
                                          /  d      \2
 M g h0 - M g d(t) sin(theta) + 3/4 M |---- d(t)|  = M g h0
                                          \ dt      /
```

and solve it, with the initial condition, to give the solution in explicit form:

```
> dsolve({Energy_eqn, d(0)=0}, d(t), explicit);
                                 2
                    d(t) = 1/3 t  g sin(theta)
```

It is convenient to assign the solution to a simple variable ready to plot it:

```
> d := rhs(");
                                2
                     d := 1/3 t  g sin(theta)
```

We also need the velocity and acceleration:

```
> v := diff(d, t);
                     v := 2/3 t g sin(theta)

> a := diff(v, t);
                      a := 2/3 g sin(theta)
```

Finally, let us try to plot these functions:

```
> plot({d, v, a}, t=0..3);
Error, (in plot) too many indeterminates in functions, {
theta, t, g}
```

This complaint is not really surprising! We need to assign numerical values to the parameters:

```
> theta := convert(30*degrees, radians);
                         theta := 1/6 Pi
```

I have done it this way just to illustrate a use of the `convert` function. Also, the approximate value of the gravitational acceleration in m/s/s is

```
> g := 10:
```

Now let us try again, this time also giving the plot a title:

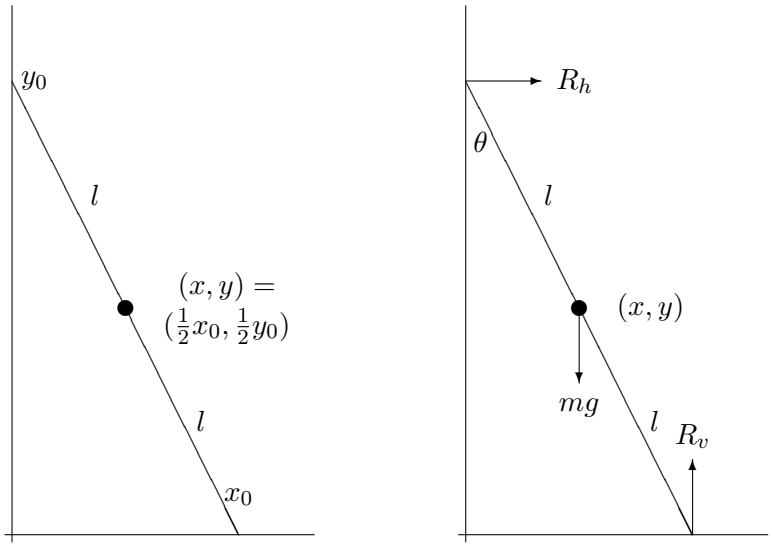
```
> plot({d, v, a}, t=0..3, title =
> `Distance, velocity and acceleration of rolling cylinder`);
```

After a brief pause, the effect of this instruction is to display a set of superimposed graphs on the screen. You may need to change the palette to get a visible plot, but this can only be done interactively. (Under MS-DOS, press function key *F10* and then press P to cycle the screen colours until the plot is clearly visible. The plot can also be interactively printed or saved to a file by using the Output menu.) When you have finished admiring the graphs, press *Esc* to return to Maple itself.

2.2 The motion of a sliding ladder

This is a slightly less straightforward example.

Suppose that a ladder (or beam) of uniform mass m has its bottom end resting on a horizontal floor, its top end resting against a vertical wall, and that both its ends slide smoothly with no friction. It the ladder is placed in any position other than exactly vertical it will slide down to the ground. Find and plot the locus of its midpoint as it slides. Find the equation of motion of its midpoint as a function of time, and plot its motion (for some choice of parameters).



Let the cartesian coordinates of the midpoint be (x, y), which depend on time if the ladder is moving and so are actually $(x(t), y(t))$. Let the length of the ladder be $2l$, and the coordinates of the bottom and top of the ladder

be $(x_0, 0)$ and $(0, y_0)$. Then the coordinates of the midpoint must satisfy $x = \frac{1}{2}x_0, y = \frac{1}{2}y_0$. The geometry is shown in the first figure; the second figure shows the forces causing the ladder to slide.

By Pythagoras' theorem, the ladder must satisfy the equation

```
> x0^2 + y0^2 = (2*l)^2;
                          2     2      2
                        x0  + y0  = 4 l
```

together with the inequality $0 \leq x_0 \leq 2l$. Hence the equation of the mid-point is

```
> subs(x0 = 2*x, y0 = 2*y, ");
                          2      2      2
                       4 x  + 4 y  = 4 l

> geom_eqn := "/4;
                                  2    2    2
                     geom_eqn := x  + y  = l
```

constrained by the obvious inequality $0 \leq x \leq l$. Solving the equation explicitly and taking the positive solution gives

```
> solve(",y);  y_x := "[1];
                    2    2 1/2         2    2 1/2
                (- x  + l )   , - (- x  + l )

                                 2    2 1/2
                      y_x := (- x  + l )
```

I have given names to the mid-point equation and the explicit form of $y(x)$ so that I can use them again later. It is of course obvious what this locus is, but it is still nice to see it. The only contribution of the length parameter l is to set the scale for the problem, so there is no loss of generality by setting it to 1 (temporarily) for plotting graphs, as follows:

```
> l := 1:
> plot(y_x, x=0..l, y, title =
>   `Locus of the Center of a Sliding Ladder`);
> l := 'l':
```

I will analyse the dynamics using forces and torques; the same analysis using energy is left as an exercise. We must now regard x and y as functions of time t, but an elegant feature of Maple allows us to leave the dependence implicit initially and to use the operator notation D for differentiation. Initially I will keep the analysis as symmetrical as possible by regarding x and y as independent variables, and only later impose the geometrical relation expressed by `geom_eqn` determined above.

There are three forces acting on the ladder: its weight mg acting downwards, where g is the gravitational acceleration; a reaction force R_v acting upwards; and a reaction force R_h acting to the right. There is assumed to be no friction, so the only forces at the ends of the ladder are the reactions perpendicular to the surfaces on which the ladder rests.

The vertical and horizontal motions of the centre of mass of the ladder are therefore governed by the following two equations:

```
> Down_force_eqn := m*g - Rv = -m*(D@@2)(y);
                                                    (2)
          Down_force_eqn := m g - Rv = - m D   (y)

> Right_force_eqn := Rh = m*(D@@2)(x);
                                                 (2)
              Right_force_eqn := Rh = m D   (x)
```

where $D^{(2)}(x)$ might more conventionally be written as $\ddot{x}$, etc. The forces also generate a torque about the centre of the ladder which causes an angular acceleration governed by the following equation:

```
> Torque_eqn := x*Rv - y*Rh = MI*D(w);
            Torque_eqn := x Rv - y Rh = MI D(w)
```

where `MI` represents the moment of inertia of the ladder about its centre, and `w` represents the angular velocity ($\omega = \dot{\theta}$) so that `D(w)` represents the angular acceleration ($\dot{\omega} = \ddot{\theta}$).

Eliminating the vertical and horizontal forces leaves the torque equation as the main dynamical equation:

```
> solve(Down_force_eqn, {Rv});
                                       (2)
                    {Rv = m g + m D   (y)}

> Torque_eqn := subs(", Right_force_eqn, Torque_eqn);
```

```
                              (2)             (2)
 Torque_eqn := x (m g + m D   (y)) - y m D   (x) = MI D(w)
```

The moment of inertia MI about the centre of a ladder of length $2l$ and mass m is easily computed as in the previous example. Denoting the linear mass density by ρ, we have

```
> MI := int(rho*x^2, x=-l..l);
                                        3
                        MI := 2/3 rho l

> m = int(rho, x=-l..l);
                          m = 2 rho l

> solve({"}, rho);
                        {rho = 1/2 m/l}

> MI := subs(", MI);
                                      2
                         MI := 1/3 m l
```

Using this value in the torque equation, dividing out the mass, and simplifying the result, gives

```
> Torque_eqn := simplify(Torque_eqn/m);
                            (2)          (2)            2
  Torque_eqn := x g + x D   (y) - y D   (x) = 1/3 l  D(w)
```

The next step is to compute the angle θ, angular velocity ω and acceleration $\dot{\omega}$, all as functions (implicitly of time t). The composition operator in Maple is `@`, but unfortunately I can find no easy way to make it distribute over addition, hence the need for the slightly inelegant substitution below:

```
> tan @ theta = x/y;
                        tan@theta = x/y

> D(");
                  2                      D(x)   x D(y)
          (1 + tan )@theta D(theta) = ---- - ------
                                        y       2
                                               y
```

```
> subs((1+tan^2)@theta=1+(x/y)^2, ");
            /      2 \
            |     x  |            D(x)   x D(y)
            |1 + ----| D(theta) = ---- - ------
            |      2 |              y       2
            \     y  /                     y

> simplify("*y^2);
              2    2
            (x  + y ) D(theta) = D(x) y - x D(y)

> subs(geom_eqn, ");
                2
               l  D(theta) = D(x) y - x D(y)

> w := solve(", D(theta));
                           - D(x) y + x D(y)
                  w := - -----------------
                                  2
                                 l
```

The only *parameters* remaining in this problem are l and g, and we need to define them to be constants (rather than functions):

```
> constants := constants, l, g;
   constants :=
       false, gamma, infinity, true, Catalan, E, Pi, l, g
```

Then with the above definition of the angular velocity, the torque equation becomes

```
> Torque_eqn;
         (2)          (2)              (2)              (2)
x g + x D   (y) - y D   (x) = 1/3 y D   (x) - 1/3 x D   (y)
```

Collecting all terms on the left allows some simplification by cancellation to give the final fairly symmetrical ODE:

```
> Torque_eqn := lhs(") - rhs(") = 0;
                                (2)              (2)
   Torque_eqn := x g + 4/3 x D   (y) - 4/3 y D   (x) = 0
```

Expressing y in terms of x gives a differential equation for x alone:

```
> y := y_x;  Torque_eqn;
                              2    2 1/2
                    y := (- x  + 1 )

 x g + 4/3

       /      (2)                    2              2  2   \
       |     D   (x) x          D(x)           D(x)  x    |
     x |- -------------- - -------------- - --------------|
       |      2    2 1/2       2    2 1/2       2    2 3/2|
       \  (- x  + 1 )      (- x  + 1 )      (- x  + 1 )   /

                 2    2 1/2  (2)
      - 4/3 (- x  + 1 )    D   (x) = 0
```

Normalizing and discarding the common denominator gives

```
> numer(normal(lhs(")")) = 0;
              2    2 3/2      (2)     2  2           2  2
   3 x g (- x  + 1 )    + 4 D   (x) x  1  - 4 x D(x)  1

              (2)     4
        - 4 D   (x) 1  = 0
```

This equation looks better if the terms are collected with respect to the first and second derivatives, although this is not necessary for subsequent analysis:

```
> Torque_eqn := collect(", {(D@@2)(x), D(x)});
                           2  2      4   (2)
       Torque_eqn := (4 x  1  - 4 1 ) D   (x)

                        2    2 3/2           2  2
            + 3 x g (- x  + 1 )    - 4 x D(x)  1  = 0
```

This torque equation is actually an operator equation, and to proceed it is necessary to make the dependence on t explicit:[1]

[1] `g(t)` should have simplified to just `g` because `g` has been declared constant, in the same way that `l(t)` has simplified – this appears to be a bug in Maple V R1.1. It does simplify when `g` is later given an explicit numerical value.

```
> Torque_eqn := Torque_eqn(t);
                        2  2      4   (2)
 Torque_eqn := (4 x(t)  l  - 4 l ) D   (x)(t)

                             2    2 3/2                  2  2
      + 3 x(t) g(t) (- x(t)  + l )    - 4 x(t) D(x)(t)  l

      = 0
```

The analytical solution given by

```
dsolve(Torque_eqn, x(t));
```

is horrible (try it for yourself), so instead let us solve the ODE numerically and plot a graph of $x(t)$. First we need to fix the parameters:

```
> l := 1:  g := 10:
```

It is clear from the ODE that $\ddot{x} = 0$ at $x = 0$, so that there is an (unstable) equilibrium when the ladder is exactly vertical. Therefore, let us start the ladder at rest at a small angle to vertical, i.e. with x having a value that is a small fraction of its maximum value l $(= 1)$.

```
> F := dsolve({Torque_eqn, x(0)=0.1, D(x)(0) = 0}, x(t),
>   numeric);
F := proc(t) `dsolve/numeric/result2`(t,1559200,[2]) end
```

The numerical solution of a DE in Maple is a procedure, which when called with a particular value of the independent variable returns a *sequence* of values consisting of the independent variable followed by the dependent variables, in this case $t, x(t)$, of which we only want $x(t)$ to plot. One way to plot the values returned by such a procedure is to embed it in another (anonymous) procedure that extracts the second element returned by F, thus:

```
> plot(proc(t) F(t)[2] end, t=0..1.4, x, title =
>   `Horizontal Position of the Center of a Sliding Ladder`);
```

Note that this plot takes a few minutes to produce. The plot command could easily be extended so as to plot both $x(t)$ and $y(t)$ together (which would, of course, take twice as long).

3 About Maple and its use

Maple is a system for mathematical computation designed and implemented by the Symbolic Computation Group at the University of Waterloo in Ontario, Canada. It is named after the Canadian national emblem. Maple began development in 1980, and is now one of the major 6 such systems. It runs on a wide range of computers including relatively cheap personal micro-computers. I will refer to systems like Maple as "Computer Algebra Systems" (CASs), even though they all do more than "just algebra", because it is the algebraic and symbolic facilities that distinguish them from other computational systems and languages.

Maple provides good facilities for algebraic or symbolic computation, and fairly good facilities for numerical computation and graphics. It has a good online help facility, which is particularly useful when one is learning how to use it, and a fairly friendly user interface, which almost unavoidably varies between different computing systems.

The current version of Maple is version V, and a second release of Maple V is becoming available for various computing systems. To be specific, I will describe Maple V Release 1.1 for 386/486-based micro-computers running under MS-DOS Version 5.00. This runs satisfactorily on a 25MHz 386 with 4Mb of (conventional plus extended) main memory and a hard disk. The installed system takes about 8Mb, and ideally Maple needs some free disk space to use as "swap space" to provide "virtual memory" unless there is a lot of physical main memory. Maple appears to be intended to drive a colour screen. With a monochrome (black and white) screen (at least with my LCD display) I found it a little tricky to make the user interface and graphics fully useable. However, it is possible to get output that looks quite similar to conventional printed mathematics, and useful well-defined graphics. Most important aspects of Maple are independent of the particular system on which it runs.

3.1 Maple documentation

Maple V is one of the better documented CASs. There are three books about it written by the main Maple developers and published by Springer-Verlag. The "Maple V Language Reference Manual" and the "Maple V Library Reference Manual" were published in 1991, and "First Leaves: A Tutorial Introduction To Maple V" was published in 1992. Most of Maple consists of library routines, and these are *all* documented in the online help

system, i.e. the whole of the library reference manual is available within Maple itself in an essentially identical format, so that other than for reading in bed, on the beach, etc, there is little need for this manual.

For this course it should not be essential to refer to any documentation other than the course notes, and I will give a summary of the Maple language later. However, it is useful to read "First Leaves" to get a feel for how one uses Maple, and the language reference manual is useful for details of the language that are not available online. The language reference manual is probably the most useful of the three books to actually buy.

I will also provide an introduction to Maple written by the UK National Computer Algebra Support Officer while that project was running, which is freely distributed.

The internal operation and theory of CASs is outside the scope of this course but, for anyone interested, the most appropriate book to read in the context of Maple, and the most recently published, is "Algorithms for Computer Algebra" by K. O Geddes, S. R. Czapor and G. Labahn (Kluwer, 1992), but unfortunately this book is very expensive. A much cheaper book, which does not relate specifically to Maple but is nevertheless very useful, is "Computer Algebra: Systems and Algorithms for Algebraic Computation" by J. H. Davenport, Y. Siret and E. Tournier (Academic Press, 1988 – second edition available now or soon).

3.2 The DOS-Maple user interface

Much, but not all, of the information in this section is specific to the MS-DOS implementation of Maple, whereas I hope that the rest of the notes will be generally applicable. There is also a new version of Maple for Microsoft Windows, which should be fairly easy to use by anyone familiar with Windows.[2] Here I will describe primarily the MS-DOS version (which can also be run under Windows in an MS-DOS window).

3.2.1 Starting and running Maple

To start Maple under MS-DOS type the command `maple`. (To start it under Windows open the appropriate program group and then open the Maple icon.)

[2]It looks similar to the X-Windows version described briefly below (which appears to be the direction in which Maple development is heading), but it is not identical.

When Maple starts, it briefly displays an initial banner which then disappears to leave a screen that is blank except for a triangular "prompt" near the bottom left, and a "status line" along the bottom. At the right end of the status line is a display of the amount of memory in kilobytes that Maple is currently using, which increases and decreases while Maple is running, and the total amount of time in seconds that Maple has spent actually computing. As an alternative to typing commands, Maple provides menus that operate in the conventional MS-DOS way: pressing the function key *F10* brings up a menu bar at the top of the screen, from which menus and options can be selected either by using the cursor keys and the *Return* key or by pressing a letter key, which usually corresponds to the first letter of the menu entry.

3.2.2 Stopping Maple

To exit from Maple when it is waiting for input press function key *F3* and then acknowledge the query by pressing the *Y* key.

To interrupt a computation and return Maple to input mode press the *Ctrl*, *Alt* and left *Shift* keys simultaneously. Alternatively, the usual ways of killing an MS-DOS program should work, but are not recommended because they may cause disk corruption.

3.2.3 Online help

Pressing function key *F1* displays a menu of topics, for which subtopic menus can be selected to at most two levels by pressing the right-arrow (→) key, and help on the currently selected topic is provided when the *Return* key is pressed. Pressing *F2* causes Maple to search for help on the specified topic. Alternatively, you can type a command of the form `?`*topic* or just `?`. However the help system is invoked, once a suitable topic has been chosen then help on it is displayed using the Maple "browser" facility, which is explained further below.

3.2.4 Session review

Maple automatically keeps a "session log" – an internal record of its input and output – which can be saved to disk from the Session menu or reviewed interactively by pressing function key *F5*. The browser then allows previous input to be selected and re-input, possibly after editing it. Maple output in the session log looks slightly different from that originally shown on the

screen, because it uses only standard ASCII characters and not the enhanced IBM character set, so that it is portable (machine independent) and also can be edited using any standard text editor. Output of the session log directly to a file (called "Capture") can also be turned on and off from the Session menu.

3.2.5 The help and session review browser

The browser allows scrolling up and down using the usual keys and searching using function key *F2*. It also allows a block of lines to be selected and either input to Maple or edited. To do this, move the highlight to the first line of the block and press the *Ins* key, then move to the last line of the block and press *Ins* again. Pressing *Del* cancels the block selection. When a block is selected, pressing the *Return* key will read the whole block into Maple, or pressing function key *F5* will load the block into the Maple editor. After editing, pressing *F2* (save) and then *F3* (quit) will return you to the browser containing only the edited block which is automatically selected, so that pressing *Return* will read the whole edited block into Maple. Note that the session log itself is never changed and only a copy of (part of) it is actually edited. This facility allows examples in the help documentation to be input into Maple and executed "live".

The Maple editor provides its own online help, which documents all of its commands, by pressing function key *F1*. The same editor is used for all (except command line) editing within Maple.

3.2.6 Command line input

Most interactive input to Maple is typed in response to the triangular prompt symbol at the bottom left of the main screen area and appears on the screen there. Maple does not process such input until the *Return* key is pressed, and until that time it can be edited using the following keys:

$\rightarrow$ **or** $\leftarrow$ moves the cursor right or left one character without erasing;

Home **or** *End* moves the cursor to the beginning or end of the line;

Backspace (⟵) **or** *Del* deletes the character before or at the cursor;

Esc deletes the whole line;

Ins turns insertion mode off and on – insertion is initially on.

Maple remembers the last 100 lines of interactive input, and the ↑ and ↓ cursor keys scroll through this list, from which a selected line can be edited as above and then re-input. To search for a particular line, type the first few characters of it and then press function key *F2*.

3.2.7 Expression editing

Any valid Maple expression can be edited in the Maple text editor by using the Edit menu, or by giving an instruction of the form

`xed(`*expression*`):`

On saving and quitting from the editor, the expression is re-input to Maple. If *expression* was actually a variable that evaluated to the expression being edited then the edited expression is assigned back to the same variable.

3.2.8 Using files

The word "file" is the name given to a collection of information stored on a secondary storage device such as a magnetic disk. Files are stored in "directories" (sometimes called "folders"), which are essentially files with a special structure. It will probably be necessary occasionally to use the facilities provided by MS-DOS (or Windows) to manipulate files. If Maple is run directly from MS-DOS then a new MS-DOS command processor can be run from within Maple by pressing function key *F4*, and then when the MS-DOS command `exit` is executed control will be returned to Maple, exactly as it was immediately before MS-DOS was called. Alternatively, a single MS-DOS command can be passed from within Maple to MS-DOS for execution by preceding it with a exclamation mark, i.e. by executing a Maple command of the form

`!`*MS-DOS command*

which will return to Maple immediately after executing the command. This use of the "`!`" character to call MS-DOS is analogous to the use of "`?`" to call the Maple help facility.

MS-DOS commands are not case sensitive. The most useful are probably those in the table below. Most of this list is an exact subset of the output of the MS-DOS help command, which can be used to find all MS-DOS commands or to find more information about any particular command. Alternatively, invoking any MS-DOS command with the switch "`/?`" will display help on the command, i.e. give to MS-DOS a command of the form

Table 1: Main MS-DOS commands

`HELP`	Provides Help information for MS-DOS commands.
`DIR`	Displays a list of files and subdirectories in a directory.
`CD`	Displays the name of or changes the current directory.
`X:`	Change to disk drive X.
`COPY`	Copies one or more files to another location.
`DEL`	Deletes one or more files.
`REN`	Renames a file or files.
`MD`	Creates a directory.
`RD`	Removes a directory.
`FORMAT`	Formats a disk for use with MS-DOS.
`EXIT`	Quits the `COMMAND.COM` program (command interpreter).

command `/?`

It is also possible to run the Maple editor on a file outside Maple by giving MS-DOS a command of the form

`mapledit` *file*

The main use of files with Maple (at least for this course) is to develop Maple programs in files that can be read into Maple instead of typing instructions interactively. A file (or directory) in MS-DOS is identified by a *name* consisting of up to 8 letters optionally followed by a point "`.`" and an *extension* consisting of up to 3 letters. (In fact, digits and some other characters are allowed in file and directory identifiers, but it is probably safest to use letters.) The purpose of the extension is to indicate the type of a file, and therefore it makes sense to use an extension such as "`.map`" for Maple source files, but Maple itself does not enforce this. The only special extension that Maple recognises is "`.m`", which indicates a Maple binary file. These files are not human-readable and there is no need to use them for this course, so *DO NOT USE THE EXTENSION ".M"*. It is also reasonable to use no extension at all, in which case the point (`.`) can also be omitted.

The best way to start developing a file of Maple code is to work interactively and then save the interactive input. You can either save the whole session, or edit it before saving, as described above. Either way, you can then edit the saved file as described below. A file of Maple code is read into Maple by giving a command of the form

`read` `` `Maple file identifier` ``;

Note that the quotes must actually be typed as show here,[3] and that they are both *backquotes*[4] (which could alternatively be called *left quotes* or *opening quotes*). The backquote is not the same as the forward quote (') or the double quote ("), which both have other meanings in Maple. Unfortunately, the precise location of such symbols differs among different keyboards.

Maple defines a format for file identifiers[5] that is the same on all systems, and hence not always the same as that used by the operating system. In the case of MS-DOS the difference is that Maple uses forward slashes "/" in place of the backslashes "\" used by MS-DOS to separate directory components. However, for files in the *current directory* it suffices to specify only the file name (and extension), so it is best to keep all your Maple files in one directory and make this the (MS-DOS or Windows) current directory before starting Maple. File identifiers in Maple should always be specified as described here; when given as part of a command to the main input prompt the backquotes should be used as shown above for the `read` command, but when given to a prompt from a menu the backquotes should not be included.

By default, a Maple input file is not displayed or *echoed* to the screen – only the output that it produces is displayed. To have the input also displayed, the value of the interface variable `echo` must be increased to at least 2, which can be done using the Interface menu or the `interface` command. There is an inverse operation of `read` called `save`, and screen output can also be re-directed to a file by using the commands `writeto` and `appendto` – see the online help for details.

Files can be edited using the Maple editor from within Maple, most easily via the Edit menu, or alternatively by using the commands `fed` or `fred`.

3.2.9 The graphics display drivers

There are two drivers, for two- and three-dimensional plots, and online help about the drivers is available through the commands `?2Dgraphics` and `?3Dgraphics`. After a plot has been displayed, pressing function key *F10*

[3] This is not strictly true, but it is always safest!

[4] Here, and in the rest of this set of notes, I will use the symbol ` for the backquote to make it stand out and because it looks more like the symbol normally used on keyboards and screens, but elsewhere I will use the symbol ‘, as I used in the introductory examples, which is the true typewriter font backquote character.

[5] It is essentially the UNIX format.

produces a menu that allows various features of the plot to be changed interactively and the plot optionally to be output. (However, you will probably not have access to a suitable output device, and printed output will probably not be required for this course.) This interactive facility can be particularly useful to change things like the viewing direction for a 3D surface plot.

3.3 Maple on other systems

The "Maple V Language Reference Manual" describes the use of Maple under UNIX and X Windows, which are both relevant to the SUN workstation implementation, and under MS-DOS. (These are clearly seen as the main systems on which Maple will be run!)

Under UNIX, online information about running Maple should be available by giving the UNIX command

```
man maple
```

and the standard version of Maple should be executed by giving the UNIX command

```
maple
```

This version of Maple may not provide very good graphics.

Maple will read the initialization file `.mapleinit` in your home directory. Maple should respond normally to the standard interrupt and suspend signals from the keyboard, and the system escape mechanism (`!` etc.) described above under MS-DOS should also work. A double interrupt will stop Maple completely (as will the keyboard quit signal).

X Windows is a portable windowed user interface that runs under UNIX, and to run Maple under X Windows (which is probably advisable if possible) use the command

```
xmaple
```

X-Maple provides a window onto a "worksheet", which is divided into "cells", each containing one input statement and its output. Such worksheet formatting can be controlled by options in the Utilities menu. A caret ^ indicates the text insertion point, which can be moved with the mouse or cursor keys. Long lines can be entered without being read by Maple by ending each line not with *Return* but with *Newline*, *Enter* or *Control-J*, which allows multi-line input to be reviewed and edited. To read the input

into Maple, select (highlight) it with the mouse and then press the *Return* key; otherwise the effect of *Return* is to read in only the current line as usual. This technique can also be used to re-read any text in the Maple worksheet.

Text can be selected in various ways: double clicking the left mouse button selects a word; triple clicking it selects a line; clicking the left button at the start and the right button at the end selects a block, as does dragging. Selected text is automatically copied to an internal buffer (the clipboard), and can be inserted *at the caret* by clicking the middle mouse button.

You can initiate a search for particular text in the worksheet by pressing *Control-S* (or *Control-R* to search backwards). There are buttons on the window to interrupt, pause or quit Maple. Help can be accessed only via the `?` command, but the help appears in a separate window, in which the text search and copying facilities can be used. Similarly, plots appear in separate windows, and should be of good quality. There are boxes showing the location of the mouse pointer within the graph.

On any other system, there is a good chance that the command `maple` will run Maple if it is available, but it may require a little effort to configure it to produce good graphics.

4 The Maple language

You have already seen Maple in action in the previous simple worked examples. The purpose of this section is to explain more carefully how Maple works, and provide a reference. In most cases further details are available in the online help system, and the important thing is to know what facilities are available. Examples of the facilities outlined below will be given as the course progresses, hence much of the following description will be very brief.

The Maple language is a modern high-level language, which unfortunately is not very similar to any other programming language. It is generally "Algol-like", and is perhaps closest to Algol 68. Among current popular languages it has similarities to Pascal, but without any type declarations or begin/end statements; statements are grouped into blocks automatically where appropriate.

The Maple language is interpreted, as for example BASIC and LISP usually are, rather than compiled, as for example Pascal and C usually are. Maple source code is stored within Maple in a compressed binary form, and it is possible to input and output it in this form, but it is not compiled.

Like most interpreted languages, Maple is intended to be used interactively, meaning that the user types an instruction and then, depending on the result that it produces, decides what to do next. However, sequences of instructions identical to those that would be typed interactively can be entered from a file. Variable *type declarations* are not very convenient to use interactively, nor are they as relevant as in a compiled language, and it is common for an interpreted language not to use type declarations. Even though variables do not themselves have types, data do, and a variable inherits the type of the datum assigned to it, so that variable typing is handled dynamically.

4.1 Maple input format

Maple is a free-format language like Pascal, meaning that the beginnings and ends of lines of program text are of no significance (except for comments – see below). Because statements are not terminated or separated by the ends of lines it is necessary to use another character, and Maple uses both the semicolon "`;`" (as do many languages) and also the colon "`:`" (as do some versions of BASIC). The difference is that if "`:`" is used to terminate a "top-level" statement input interactively then any output that it might have produced is suppressed. I recommend normally using "`;`" except for top-level statements that you specifically require to produce no output. Maple allows an empty statement, so that it makes no difference if a statement separator is included where none is required. (Pascal, for example, is much more fussy about this!) Hence, if in doubt, include a separator.

When input is interactive, Maple *reads* each physical line when the *Return* key is pressed, but only *executes* those statements that have been terminated within the line, and saves any final un-terminated statement to be continued on the next line.

Maple is a case-sensitive language, which means that it regards the same letter in upper case (e.g. "`X`") and lower case (e.g. "`x`") as two different characters. This is like C and unlike Pascal. Hence care must be taken to use the right case. Most words of the Maple language itself are in lower case, but some library functions have the first letter only capitalized and a few are written using all capitals. A few functions come in two forms, e.g. "`int`" and "`Int`" as used in the worked examples, the second of which has the first letter of its name capitalized and is an "inert" version that itself never computes anything. Inert functions just provide data structures that are understood by other functions including the normal Maple two-dimensional output function (referred to as the "pretty-printer").

When composing Maple instructions in a file it is a good idea to include *comments* for the benefit of yourself and other human readers. The sharp sign "`#`" and all characters following it up to the end of a physical line are treated as a comment and ignored by Maple. (This is about the only case where a physical end-of-line has any significance.) It is a good idea to include plenty of comments when programming, but only use comments that actually convey useful information. The comments should be written at the same time as the code and not afterwards, and they should be treated as equally important and kept up-to-date when the code is edited. Maple allows completely blank lines, which are useful to separate logically distinct segments of code.

4.2 Identifiers, names and strings

Ojects such as variables and functions are given names or identifiers by which to refer to them, and a "string" is the name given in programming to a sequence of characters that are to be manipulated literally, for example as a title for a plot. In Maple, identifiers and strings are distinguished only by the way that they are used, and any string can be used as an identifier. A string can be up to 499 characters long. A letter followed by zero or more letters or digits is automatically a string, and it is normally best to use only such simple strings as identifiers. The underscore character counts as a letter for this rule, but identifiers beginning with an underscore are used internally by Maple and so should not be introduced by the user. More generally, a string enclosed in backquotes can contain any characters, as described above in the context of file identifiers.

4.3 Statements and expressions

A *program* consists of a sequence of instruction *statements* input from the keyboard and/or files, which are *executed* by Maple in the order in which they are input. However, this is strictly true only for "top-level" statements, and for the statements within any group controlled by a selection or repetition statement, because (as language purists will be pleased to note) there is no "go to" type of statement in Maple. In fact, the Maple language itself is extremely small, but it has a lot of operators and even more library functions.

Much of the work of Maple is accomplished using *expressions*, and an expression is a valid statement but not vice-versa. The only statements

that have a value associated with them are expressions. If an expression is input at top level as a statement and terminated with a "`;`" then its value is automatically output, so that it is not necessary to call the `print` function explicitly – this is only necessary to output values from lower-level expressions.

4.4 Variables, assignment and unassignment

Variables in mathematics are used in two way; sometimes they represent an unspecified value and are called *unknowns*, and sometimes they represent a specified value, when their meaning in an expression is that they are to be replaced by the value that they represent. In a programming language, a variable is given a specific value by an *assignment* statement, which in Maple (as in Pascal) has the form

variable `:=` *expression* .

In "conventional" programming languages, variables can be used only in this second way – they can be used in expressions *only* after they have been assigned a value. However, in Maple, a variable with no assigned value can be used in an expression, in which case it represents an unknown and just remains symbolic. If it is later assigned a value then any previously constructed expression that contained the symbolic variable will now evaluate to the expression with the variable replaced by its value.

Assignment always transfers information "from right to left", and the asymmetric nature of the assignment operator serves as a useful reminder of the asymmetry of the assignment process itself. A variable (on its own) on the left of an assignment statement is not evaluated, so that a new value can be assigned to a variable by simply executing a new assignment statement.

Any expression can be forced to remain symbolic, i.e. to evaluate to itself, by enclosing it in single forward quotes. It is a special case of this that allows a variable to be *unassigned* or *cleared* in Maple by an assignment of the form

variable `:= '`*variable*`'` .

4.5 Constants and number format

Numbers are always constants. Maple works with essentially three types of number:

integers are input as arbitrarily long sequences of digits (only);

fractions are input in the form "*integer* / *integer*";

floating-point numbers are input in either the form "*integer* . *integer*" or the form "`Float(`*mantissa*`,` *exponent*`)`" (note the capital F), which represents $mantissa \times 10^{exponent}$.

Any number can be made negative by preceding it with the (unary) minus operator "`-`". (This character is the same as the hyphen used when typing text; do not confuse it with the underscore character which is often placed on the same key and accessed with the *Shift* key.) A positive number can optionally be preceded by the (unary) plus operator "`+`". This can occasionally be useful to emphasize that a number is positive, or to type a pair of positive and negative numbers symmetrically.

The concept of a "symbolic constant" is also useful. Any unevaluated variable can be regarded by the user as a symbolic constant but, to make Maple recognise it as a constant, its name should be included in the sequence of constant names assigned to the reserved variable `constants`. The symbolic constants that Maple recognises by default are

`false, gamma, infinity, true, Catalan, E, Pi, I` .

More generally, Maple recognises as a constant any expression that contains only numbers and recognised symbolic constants.

In fact, the constant `I` is implemented as an `alias` for `(-1)^(1/2)` to give it the properties of the imaginary number, rather than explicitly as a symbolic constant. A form of symbolic constant is also provided by the `macro` function.

4.6 Names

Simple names are strings, but composite names are also allowed, which can either have a functional form

name`(` *sequence* `)`

or an indexed form

name`[` *sequence* `]`

where *sequence* is a sequence of one or more expressions. This construction can be nested. Names can also be constructed using the catenation operator "`.`" or the library function `cat`.

4.6.1 Labels

These have the form "`%`*positive integer*". They are introduced automatically by the pretty-printer, and their values can then be used as if they were assigned variables.

5 Maple expressions

In general, expressions are constructed from constants, variables, operators and functions. Operators take arguments called *operands*, the number of which is determined by their *arity*: a *nullary* operator takes no operands (and hence behaves like a variable or constant); a *unary* operator takes one operand; a *binary* operator takes two operands. Unary operators are normally placed in front of their operand, and are called *prefix operators*. An operator that is placed behind its operand is called *postfix*; there is only one in Maple, the factorial operator "`!`". Binary operators are placed between their two operands and are called *infix*. The only difference between operators and functions is their input and output format: functions are always prefix and their arguments must be enclosed in parentheses and separated by commas. Expressions can be either algebraic (which includes numerical) or Boolean (logical), and those containing relational operators can be regarded as either, depending on the context.

5.1 Operators

The following tables, taken from the "Maple V Language Reference Manual", provide a useful summary of (nearly all) of the operators known to Maple. The words (as opposed to special symbols) used to identify operators are *reserved words*, which means that they cannot be used for any other purpose.

The recall operators (`"` etc.) recall the *result* or *value* of the previous statement. If it was an assignment then the value assigned is recalled; if it was a control statement then the result of the last statement executed is recalled, and this applies also to nested control statements. However, only the results of procedure calls, and not of any statements executed within the procedures, are recalled, and null expression sequences and loop control expressions are not recalled.

By default, any operator whose name begins with "`&`" is *neutral* or *inert*, meaning that it performs no computation. However, most operators (includ-

Table 2: Nullary operators

Operator	**Meaning**
`"`	previous expression value
`""`	expression value two back
`"""`	expression value three back

Table 3: Unary operators

Operator	**Meaning**
`+`	unary plus (prefix)
`-`	unary minus (prefix)
`!`	factorial (postfix)
`$`	sequence operator (prefix)
`not`	logical not (prefix)
`&`*string*	neutral operator (prefix)
`%`	label (prefix)

Table 4: Binary operators

Operator	**Meaning**	**Operator**	**Meaning**
`+`	addition	`<`	less than
`-`	subtraction	`<=`	less or equal
`*`	multiplication	`>`	greater than
`/`	division	`>=`	greater or equal
`**`	exponentiation	`=`	equal
`^`	exponentiation	`<>`	not equal
`$`	sequence operator	`->`	arrow operator
`@`	composition	`mod`	modulo
`@@`	repeated composition	`union`	set union
`&*`	noncommutative mult.	`minus`	set difference
`&`*string*	neutral operator	`intersect`	set intersection
`.`	concatenation; decimal	`and`	logical and
`..`	range	`or`	logical or
`,`	expression separator	`:=`	assignment

ing all user-defined ones) correspond internally to a call of a function with the same name as the operator, and if such a function has been defined then it is applied. Operators can also be given a small set of specific properties by using the `define` function. Inert operators merely provide data structures that can be interpreted by other functions. For example, the inert operator "`&*`" is primarily used with matrices, to be described later in the course, and is interpreted by the `evalm` function; the `mod` operator interprets various inert operators not listed above. The functional composition operator "`@`" and repeated composition operator "`@@`" work analogously to "`*`" and "`**`" for multiplication and repeated multiplication (exponentiation) respectively.

The precedence and associativity of operators is designed to match their usual mathematical usage, but parentheses "`( )`" can always be used to force a particular grouping of sub-expressions. It is generally better (and quicker) to use parentheses than to look up the precise precedence and associativity of operators. However, arithmetic operators have higher precedence than relational operators, which have higher precedence than Boolean operators, so that the use of parentheses in Boolean expressions is not compulsory in the way that it is in Pascal.

The *relational operators* "`=`" etc. are used to *compare* the values of other expressions, and they can be interpreted as algebraic expressions themselves, or evaluated to give Boolean values, depending on the context. Usually, the correct interpretation is automatic, but the function `evalb` can always be applied to force Boolean evaluation. Maple supports arithmetic on relational expressions in the obvious ways. When an "ordered comparison" using one of the operators $<, \leq, >, \geq$ is evaluated to give a Boolean value, the difference of the operands must evaluate to a number so that the ordering can be determined; otherwise, comparison of symbolic expressions is allowed. However, no simplification is performed by default, so that mathematically equal expressions may not be regarded as equal by Maple unless they are both simplified into canonical form.

The Boolean operators "`and`" and "`or`" are what is sometimes called "conditional", because their first operand is always evaluated first, and their second operand is not evaluated at all if the value of the first operand already determines the value of the expression (as in C, Lisp and REDUCE). Hence, the following code works as one (presumably) expects:

```
if type(a, numeric) and a < 0 then ...
```

(which it would not if "`and`" worked as in Pascal). Boolean algebra in Maple uses a non-standard *three-valued logic*, in which the three constant values

are `true`, `false` and `FAIL`. `FAIL` is treated essentially the same as `false`, but beware of the asymmetry caused by the fact that "`not FAIL` = `FAIL`".

5.2 Structured data

In a sense any non-trivial algebraic expression is structured, but here I primarily want to consider explicitly structured data types analogous to those provided in conventional languages.

An unusual data type provided by Maple is the *expression sequence* (*sequence* for short), which is just a sequence of general expressions separated by commas, *without* any kind of bracketing or other delimiters, of the form

$$expression_1 \ , \ expression_2 \ , \ \ldots \ , \ expression_n \ .$$

An expression sequence is always simplified to a single un-nested sequence. An empty sequence is input as the reserved word `NULL`. The function `seq` may be used to construct sequences in a way analogous to a `for` loop. There is also a *sequence operator* "`$`", which is generally less efficient than `seq` but more convenient interactively, particularly for specifying multiple derivatives, to be described later in the course.

More conventional data structures are *sets* and *lists*. A set in Maple is represented as a sequence enclosed in braces thus

$$\{ \ sequence \ \}$$

as is normal in mathematics, and a list is represented as a sequence enclosed in square brackets thus

$$[\ sequence \ ]$$

(as in Prolog, for example). Both can be empty. A set is an unordered collection of data in which no element may appear more than once, whereas a list is an ordered collection of data in which any element may appear any number of times. Maple therefore does not preserve the order of elements of a set but re-orders them into the current system order and removes any duplicate elements, whereas it preserves all the elements in a list exactly as they are given. There is no operator to concatenate (join) two lists – this is achieved by building a new list from the sequences contained within the two old lists. (See the exercises.)

Another unconventional data type is the *range*, which has the form

$$expression \ .. \ expression \ .$$

This notation is as used in Pascal, except that ranges in Maple can be used more generally, and two *or more* consecutive points are recognised as the range operator. The range operator itself is inert, and it requires some other operation to interpret its meaning. It can be used in the functions `int`, `Int`, `sum` and `Sum` to express definite integration and summation. The sequence operator "`$`" expands a range into a sequence, and the concatenation operator "`.`" expands a range and concatenates onto each element of the expansion sequence.

The most general structured data types in Maple are arrays and tables, which will be described later.

Elements of data structures can be selected in Maple in very general ways (reminiscent of the somewhat arcane language APL). The conventional indexing notation

name[*expression sequence*]

(as used in Pascal and C) can be applied to an expression sequence, set, list, array, table, or to an unevaluated name in which case the indexed name remains symbolic, as described earlier. Elements of the indexing sequence can be ranges, in which case a sequence rather than a single element is extracted, and if the indexing sequence is empty then the whole structure is extracted as a sequence. More general element selection can be performed by the library function `op`.

The left and right operands of any relational or range operator can be extracted by applying the library functions `lhs` and `rhs` respectively.

5.3 Simplification, evaluation and unevaluated expressions

Simplification is the process of converting algebraic data into a form that is (in some sense) simpler, either for the user or (more usually) for the algebra system. Some CASs, notably REDUCE, usually fully simplify expressions at every stage of a calculation, but Maple does not. Maple always performs some minimal simplifications that are always likely to be appropriate, but otherwise the user must explicitly apply various simplification functions (`simplify` and its relatives) in order to change the form of any data. The default simplification is to perform all possible arithmetic, some elementary function evaluations, and to re-order expressions into the system order. But by default Maple does not expand, factorize, or apply trigonometric identities, for example.

Evaluation is the process of replacing a variable by its value, if it has one. Usually, a chain of assignments is followed to its end. Evaluation of any expression can be prevented by enclosing it in single (forward) quotes (apostrophes), and this construction can be nested; the effect of evaluating such an "unevaluated expression" is to remove one level of quotes. Even though the variables within an unevaluated expression are not evaluated, the expression nevertheless receives the usual default Maple simplification. Particular forms of evaluation can be forced by using the family of library functions with names beginning with `eval`. The main use of unevaluated expressions is to assign to a variable its unevaluated name in order to clear any assignment to it, or to explicitly use the unevaluated name of a variable in order to ignore any assignment to it. Unassigned variables are necessary, for example, as loop control variables, and when a procedure returns data via its arguments – see below.

6 Procedures and functions

A *procedure* is a block of instructions that perform a well-defined task, and information can be input to the procedure via *arguments* that can be used to supply either data or control information. Some languages, such as Pascal, make a distinction between a procedure that has a data value associated with its name, which is then called a *function*, and one that does not, but other languages including C and Maple do not make this distinction. A Maple procedure can either return a value to be associated with its name or not. The instructions within a procedure are called the procedure *body*.

The idea behind using procedures is that once defined they can be executed or *called* from any point in a program, including from within a procedure, in order to perform their task as a service to the program at that point. Procedures can call themselves, which is called *recursion*. When a procedure is written it cannot be known exactly what the state of Maple will be when the procedue is executed – in particular, what variables will be in use and what their values will be. It is therefore essential, in order to write reliable programs, that procedures have their own *local* variables that are independent of variables used elsewhere in the program that might have the same name. This is particularly important with recursive procedures.

Most modern programming languages by default treat procedure dummy arguments as local variables that are initialized by the actual argument values used when the procedure is called. This is called "call-by-value",

and such dummy arguments can be used freely within a procedure without having any affect outside it, and they never return any information to the calling program. There is also usually a mechanism that allows procedure arguments to pass information freely both into and out of a procedure, by arranging that the dummy argument in some way "shadows" the actual argument (which must be a variable and not a more general expression) used when the procedure is called. The usual mechanism is "call-by-reference", in which a pointer to the data rather than the data itself is passed to the procedure. This is how "`var`" arguments in Pascal work (and how procedure arguments in versions of FORTRAN before FORTRAN 90 always work). There is a different but related mechanism called "call-by-name", which was introduced in Algol 60, and it is essentially this which is the only argument-passing mechanism provided in Maple. (Actually, it is "call-by-evaluated-name", because arguments are first fully evaluated and then textually substituted into the procedure body.) Hence procedure arguments cannot be treated as local variables (which necessitates a "FORTRAN style" of programming) and they can be used to return information.

The final peculiarity of Maple procedures is that by default they are anonymous, i.e. they have no name. Anonymous procedures are occasionally useful, but usually they are given a name by simply assigning an anonymous procedure definition to a variable. (This is quite an elegant mechanism once one gets used to it.) A procedure definition takes the form

`proc (` *nameseq* `) local` *nameseq*`; options` *nameseq*`;` *statseq* `end`

Note that although the procedure definition ends with `end` there is no "begin", and the procedure body is essentially bracketed by the `proc`–`end` pair. The `local` and `options` clauses are both optional. The `local` clause defines a sequence of local variables,[6] whose types are nevertheless not declared. (An anonymous procedure applied to no argument provides a mechanism for writing an anonymous block with local variables as available in REDUCE and C.) The `options` clause accepts a number of keywords, among which the most important is probably `remember`. If this is specified, it causes Maple to remember the result of each call of the procedure, and when the procedure is called again it takes the value from the remember table if possible rather than re-computing it. However, if the result can be affected by global variables then remembering may not work correctly.

[6]This only applies to variables textually appearing in the procedure body, unlike with Lisp-like languages.

The default value associated with ("returned by") a procedure call is the value of the last non-null statement in the procedure body, as would be recalled by a `"` immediately before the `end` of the procedure. If it should be necessary to prevent the procedure from returning a value then the return value can be made explicitly `NULL`. If the function `RETURN` is executed in a procedure body then execution of the procedure is terminated and the value associated with the procedure call is the value of the argument of the `RETURN` function. A rather more drastic way to terminate execution of a procedure is to execute the `ERROR` function, which completely terminates the current computation, returns control to the Maple prompt, and displays an error message including the name of the procedure executed followed by the value of the argument of the `ERROR` function, which will usually be a text string (in backquotes).

In a CAS it is important for a procedure call to be able to remain unevaluated in situations where evaluation is not possible. A procedure apparently remains unevaluated by returning the form of its call, and in Maple this is most easily achieved by returning the expression

```
'procname(args)'
```

exactly as shown. The reserved identifiers `procname` and `args` within a procedure evaluate respectively to the name of the current procedure and its argument sequence, and the quotes are necessary to prevent it being evaluated immediately and causing a recursive procedure call. The reserved identifier `args` provides an alternative way to refer to procedure arguments (and the number of actual arguments need not agree with the number of dummy arguments). The reserved identifier `nargs` evaluates to the number of arguments supplied when a procedure is called.

Procedures are normally called using the functional notation of mathematics (whether or not they return a value) and hence provide one way to write functions that perform computation rather than remaining symbolic. They can also be called as operators if their names begin with "`&`", in which case the name must be backquoted when the procedure is defined. However, there are two other ways to write anonymous functions (which can also be assigned to variables). The arrow notation

`(` *varseq* `) -> (` *resultseq* `)`

is very close to the mapping notation of mathematics. The parentheses can be omitted it either sequence has only one element. Sequences of variables

or results could be interpreted as vectors. If necessary, a `local` clause can be included immediately after the arrow, as for procedure definitions, but the body of an arrow-operator can be only a single expression (or sequence) and not a statement. There is also an angle-bracket notation that looks rather like the Dirac "bra-ket" notation used in quantum mechanics.

The function `unapply` is a convenient way of turning an expression into a function; it is analogous to the `lambda` function in Lisp.

The functions `trace` and `printlevel` provide assistance in finding programming errors in procedures.

7 Maple control structures

Statements in a computer program are executed in the textual order in which they are read, except that statements within a *control structure* are executed as determined by the control statement. There are essentially two types of control statement. A *selection* or "if" statement determines whether a particular sequence of statements is executed or not, and in its general form can pick one out of several statement sequences to be executed. A special case of selection is provided by the "case" statement of Pascal or the "switch" statement of C, for example, but this is not provided in Maple. A *repetition* or "loop" statement determines the order in which statements are executed, by allowing a statement sequence to be repeated a number of times that can be determined either in advance or while the sequence is being executed. This facility is provided in most languages by respectively "for" and "while/repeat" statements.

Some languages also provide a "goto" statement to transfer execution to a specified labelled statement. However, this leads to "unstructured" programs, and unstructured programs tend to contain errors. Moreover, "goto" statements do not make much sense interactively, and there are neither "goto" statements nor statement labels in Maple. One reasonable use for a "goto" statement is to provide additional control over the execution of a loop, but Maple provides structured statements to terminate a loop early, similar to those in C.

There are essentially only two control statements in Maple, which include as special cases most of those provided by other languages, and provide all the control that is ever necessary. Maple control structures automatically group the statements that they control into groups or blocks, so that there is no need for any specific block delimiters and none are provided in Maple.

Within control structures redundant statement separators/terminators are ignored.

7.1 Selection statement

The general selection statement in Maple has the form

`if` *expr* `then` *statseq* `elif` *expr* `then` *statseq* `else` *statseq* `fi`

Note that the final delimiter of an "if" statement is "fi", the reverse of "if", thereby giving a symmetrical bracketing. This idea comes from Algol 68.

The "else" clause can be omitted if its body statement sequence would be empty, and any number of "elif–then" clauses can be used, including zero if the body statement sequence would be empty. Hence this control statement can be used to select for execution one statement sequence out of any number of statement sequences, from one upwards. The interpretation of an "elif–then" clause is as if it were "else if–then", and indeed this latter form is perfectly valid, but each explicit "if–then" nested within a selection statement requires its own terminating "fi", and it can be difficult to put them in the right places, and hard to read. Therefore, only use explicitly nested selection statements when it is necessary (as it sometimes is) to correctly express the required program logic.

An *expr* appearing after an "if" or "elif" keyword is automatically evaluated as a Boolean or logic-valued expression. A *statseq* appearing after a "then" or "else" keyword is a sequence of statements (all statements are allowed) separated by statement separators – I recommend using "`;`" for uniformity and compatibility with other languages. It is not necessary to terminate the last statement in the sequence, but it is not an error to do so.

7.2 Repetition statement

The general repetition statement in Maple has one of the forms

`for` *name* `from` *expr* `by` *expr* `to` *expr* `while` *expr* `do` *statseq* `od`

or

`for` *name* `in` *expr* `while` *expr* `do` *statseq* `od`

All components of these statements are optional except "`in` *expr*" and "`do` *statseq* `od`", which on its own provides uncontrolled or "infinite" repetition. Note that the final delimiter of a "do" statement is "od", the reverse of

"do". The meaning of *expr* after the "while" keyword and *statseq* are as described above for selection statements.

An *expr* appearing after a "from", "by" or "to" keyword should evaluate to an integer, which specifies respectively the initial value, increment and final value for a control variable that will be used to count (in a general sense) the number of repetitions of the controlled statement sequence. If the "from" or "by" components are omitted then the appropriate values default to 1. If the "for" clause is included then *name* is used as the control variable; otherwise an anonymous internal variable is used. The conditions controlling repetition are checked on each iteration *before* the statement sequence is executed, so that it is possible for the statement sequence not to be executed at all.[7]

In the second form of repetition statement the (explicit or anonymous) control variable takes the value of each top-level operand of the expression specified after the "in" keyword, i.e. each element of the sequence `op(`*expr*`)`, or of *expr* if it is itself a sequence.

7.2.1 Loop termination

A repetition statement can be interrupted at any point within its controlled statement sequence by selectively executing any of the following statements:

`break` terminates the loop and continues execution at the first statement immediately following the repetition statement (exactly as in C);

`next` terminates the current execution of the loop by skipping to the end of the statement sequence and then continuing with the test for the next iteration (exactly as with `continue` in C);

`RETURN` terminates both the loop and also the procedure body in which the loop must be contained;

`quit, done, stop` terminate both the loop and also Maple itself!

It is also worth repeating here that pressing the *Ctrl*, *Alt* and left *Shift* keys together (under MS-DOS or MS-Windows) will stop the current computation (including a loop) and return to the Maple prompt, without terminating Maple itself.

[7] This is normal in modern languages, unlike the "`DO`" loop in FORTRAN 66.

8 Data types and type testing

Every expression in Maple is represented as a tree structure in which each node (and leaf) has a particular data type. A terminal node or leaf corresponds to a primitive data type, whereas a node that is itself a tree corresponds to a derived data type consisting of an operator with operands. The type of any node can be determined by the `type` function; the number of branches from any node, which correspond to operands of the operator at the node, can be determined by the `nops` function; the branches or operands themselves are returned by the `op` function. The operands of most operators are indexed starting from 1, but some types have a 0^{th} operand. The type identifiers described below are those recognised by the `type` function, which include composite types that are *sets* of related types.

8.1 Data types

The primitive data types in Maple are `integer` and `string`, which are treated as operators with only one operand. All other data types are represented in terms of some operator.

The type `fraction` is represented (in lowest terms) by an integer numerator and a positive integer denominator; `rational` = {`integer`, `fraction`}.

The type `float` is represented by an integer mantissa and an integer (restricted to machine word size) exponent of 10; `numeric` = {`integer`, `fraction`, `float`}.

The type `name` = {`string`, `indexed`}, where an `indexed` name has the form "*name* [*seq*]" and the 0^{th} operand is the *name*.

Algebraic expressions can be of type `` `+` ``, `` `*` `` or `` `^` `` (or `` `**` ``) only, because $a-b$ is represented as $a+(-b)$ and a/b is represented as $a*(b^{(-1)})$. This representation is called *sum-of-products* form. A negative term in a sum is represented with a negative numerical coefficient. Type `series` corresponds to generalized power series.

Relational expressions can have types `` `=` `` (or `equation`), `` `<>` ``, `` `<` `` or `` `<=` `` only, because $>$ and $\geq$ are always represented in terms of $<$ and $\leq$; `relation` = {`` `=` ``, `` `<>` ``, `` `<` ``, `` `<=` ``}. Boolean expressions can have types `` `and` ``, `` `or` `` or `` `not` ``, and the general type `boolean` is the union of all these types together with the names `true`, `false` and `FAIL`.

A range has type `range` or `` `..` ``. An expression sequence has type `exprseq`; to apply the function `op` to an *exprseq* it should be first put into a

list to avoid a syntax error. Sets and lists have types `set` and `list` respectively.

The `array` data type is a special case of the `table` data type. A procedure definition has type `procedure`. An (unevaluated) function form has type `function`, a special case of which is type `` `!` `` for an unevaluated factorial function; the 0^{th} operand is the name of the function. Unevaluated or inert operators are treated as function forms.

An unevaluated concatenation has type `` `.` ``, and an unevaluated expression has type `uneval`.

There is a composite type `algebraic` = {`numeric`, `name`, `` `+` ``, `` `*` ``, `` `^` ``, `series`, `function`, `` `.` ``, `uneval`}.

8.2 Manipulating data structures

Data types can be tested using `type`, and their components can be accessed using `op` and `nops`. The following functions allow data structures to be changed, either by changing their components or their types.

The function `map(`*function*`,` *expr*`,` *argseq*`)` replaces each operand of *expr* by the result of applying *function* to it as its first argument, passing the optional *argseq* as subsequent arguments.

The function `subs(`*eqns*`,` *expr*`)` replaces sub-expressions within *expr*. If *eqns* is a single equation then all occurrences of its lhs in *expr* are replaced by its rhs; if *eqns* is a sequence of equations then they are substituted sequentially; if *eqns* is a set or list then they are substituted simultaneously (i.e. "in parallel"). Note that `subs` actually only performs textual replacement of *explicit operands* in the expression tree, and not of mathematical sub-expressions; the latter more sophisticated operation requires the use of the `match` function.

Values returned by `subs` may need to be explicitly re-evaluated (by applying `eval`), particularly to get substituted function forms re-evaluated; by default returned values only receive the usual default Maple simplification. The function `subsop` is similar to `subs`, but replaces operands via their index rather than their value, except that *eqns* can only be a sequence of (one or more) equations, not a set or list, and that the substitution is always performed simultaneously, and never sequentially (although `subsop` calls can, of course, be nested).

8.3 Type testing

Maple is polymorphic, meaning that the Maple language itself does not impose any restrictions on the type of data assigned to any variable, and it is left to functions and procedures to decide (dynamically, i.e. when they are executed) what type or types of data they will accept. Thus, whereas a language like Pascal provides type declarations that take effect at compile time, Maple provides type testing that takes effect at execution time (as do most CASs).

Type testing is only relevant when writing procedures. The type of data passed to procedure arguments can be tested for two purposes: to ensure that it will not cause an error, or to decide what to do with it. The former is mainly in order to provide a good user interface; it is good programming practice and is essential in "professional quality" code, but it is not essential for a first introduction to algebraic programming, and I will not stress it. The latter is essential, but it will not be necessary very often in this course. Therefore, this description of type testing facilities will be brief.

The type mechanism in Maple is quite sophisticated and elegant. There are three type-testing functions: `type(`*value*`,` *type*`)` is a predicate that returns `true` or `false` depending on whether *value* is of type *type*; the function `whattype(`*value*`)` returns the top-level system type of *value*; the predicate `hastype(`*value*`,` *type*`)` returns `true` if any sub-expression of *value* has type *type*.

A *type* corresponds to a computational domain, and in Maple there is an algebra of types that allows them to be manipulated like other algebraic objects. A type can be a simple type identifier or a pattern that reflects a *structured type*, which consists of types combined so as to match the structure of the type. In particular, when used as a type specifier, a constant of a particular type indicates *any* constant of the same type. Most constructs mean the same when used with types as they do otherwise, with the notable exception of "{...}", which when used with types means *any one of the types in the set*. A *hierarchical type* specifier of the form "*type1*(*type2*)" matches any *type1* expression all sub-expressions of which are *type2*. As a special case, the type `polynom(`*type2*`)` indicates polynomials whose coefficients are all *type2*.

The top-level type of a datum is called its *surface type*; a type that includes information from deeper in the expression tree is called a *nested type*. For example, type `constant` is a nested type, because it is necessary to check a whole expression tree to ensure that it contains no non-constant

sub-expressions. Some additional pre-defined types are shown in the table below, where the inclusions show how they relate to each other.

Table 5: Proper subsets of types

`radical`	$\subset$	`` `^` ``
`polynom`	$\subset$	`ratpoly`
`ratpoly`	$\subset$	`algebraic`
`` `!` ``	$\subset$	`function`
`&+`	$\subset$	`` `+` ``
`&*`	$\subset$	`` `*` ``
`numeric`	$\subset$	`constant`
`constant`	$\subset$	`algebraic`
`{relation, `and`, `or`, `not`}`	$\subset$	`boolean`
`set(...)`	$\subset$	`set`
`list(...)`	$\subset$	`list`
`array`	$\subset$	`table`
`{algebraic, boolean, set, list, table}`	$\subset$	`anything`

Types are simplified by Maple in the same way that other algebraic expressions are, but using also the special relations among types. However, sometimes this can be undesirable, and care must be taken to express a type in such a way that it will not be incorrectly simplified; often using a hierarchical type (in functional form) rather than a type expression (in operator form) will avoid the simplification.

Frequently, all the type testing that is required can be expressed by constructing a list of all the arguments of a procedure by using "`[args]`" and using the `type` function to compare that against a model of the required argument number and types, also made into a list. For example, a suitable argument type check for an integration function that can be used for either indefinite or definite integration (as can `int` and `Int` provided in the Maple library) would be

```
if not type([args],
            [algebraic, {name, name=algebraic..algebraic}])
  then ERROR(...)
```

which means that the first argument (the integrand) can be any algebraic expression, and the second argument can be either a variable identifier (the integration variable) or an equation with the integration variable on the

left and the integration range on the right – note the use of a set of types to indicate that any element of the set is acceptable. This form of type testing automatically tests the number of arguments – the wrong number of arguments does not match the specified type structure. Only in complicated cases, such as a variable number of arguments or arguments with types that depend on each other, would more explicit type testing be required. There are a few more ways of specifying types, especially of function forms – see the online help or Maple manuals.

9 Arrays and tables

A *table* is a very general data structure in Maple (implemented as a hash table), a special case of which is the *array*. Tables are mostly useful for system programming tasks that will not be considered in this course. Arrays are mostly used to represent vectors and matrices, and will be discussed later in that context. Because of the representation used, sparse arrays can be implemented easily and efficiently in Maple via an indexing function that is part of the general table data structure.

10 Some Maple library functions

This section lists a few of the more useful but perhaps less obvious library functions, all of which are described in the online help system:

`assign`	convert equations into assignments;
`binomial`	binomial coefficients;
`coeff, coeffs`	extract polynomial coefficients;
`collect`	collect coefficients of like powers;
`combine`	combine terms into single terms;
`convert`	convert an expression to a different form;
`D, diff, Diff`	differentiation;
`degree`	degree of a polynomial;
`denom`	denominator of an expression;
`discrim`	discriminant of a polynomial (multiple roots);
`divide, Divide`	exact polynomial division;
`dsolve`	differential equation solver;
`edit`	structure-oriented expression editor;
`eval, Eval`	evaluation functions;

`expand, Expand`	expand an expression;
`factor(s), Factor(s)`	multivariate polynomial factorizer;
`frac`	fractional part of a number;
`freeze, thaw`	replace an expression by a name;
`fsolve`	floating-point equation solver;
`gcd(ex), Gcd(ex)`	greatest common divisor (extended);
`i`*function*	integer version of *function*;
`indets`	indeterminates in an expression;
`int, Int`	integration;
`lcoeff`	leading coefficient of multivariate polynomial;
`ldegree`	low degree or order of polynomial;
`length`	length of an object;
`limit`	limits of functions;
`linalg`	linear algebra and matrix package;
`member`	membership predicate for a set or list;
`mtaylor`	multivariate Taylor series expansion;
`normal, Normal`	normalize a rational expression;
`numer`	numerator of an expression;
`O`	order of a function;
`order`	truncation order of a series;
`orthopoly`	orthogonal polynomial package;
`plot(3d), plots`	plotting packages;
`powseries`	power series package;
`product, Product`	repeated product;
`Psi`	Ψ and polygamma functions;
`quo, Quo, rem, Rem`	polynomial quotient and remainder;
`radsimp`	simplify radicals;
`resultant, Resultant`	resultant of two polynomials;
`RootOf`	representation for roots of equations;
`round`	round a number to an integer;
`rsolve`	recurrence equation solver;
`select`	select from a list, set, sum or product;
`series`	generalized series expansion;
`sign, signum`	sign determination;
`simplify`	simplify an expression;
`singular`	find singularities of an expression;
`solve`	solve equations;
`sort`	sort a list of values or a polynomial;

`sqrt`	square root;
`sum, Sum`	repeated sum;
`taylor`	Taylor series expansion;
`trace, untrace`	trace procedures for debugging;
`trigsubs`	handle trigonometric identities;
`trunc`	truncate a number to an integer;
`type`	type-checking function;
`unapply`	convert an expression into a function;
`with`	use a library package.

11 Exercises

1. Work through the first worked example on your own, making sure that you understand what is happening. Now really "start the ball rolling" by repeating the whole calculation for a sphere rather than a cylinder. All that you need to change is the computation of the moment of inertia: imagine the sphere to be sliced into discs and use the formula for the moment of inertia of a cylinder in terms of the mass density m (a disc is a very short cylinder).

2. Repeat the computation of the motion of the ladder, but use energy conservation (as used for the rolling cylinder and ball) instead of forces. You should be able to reproduce the results in the worked example, but this is harder than for the ball problem.

3. Experiment interactively with the problem of joining or *appending* two lists. The trick is to use the sequences contained within the lists. (Look up the descriptions of the library function `op` and the indexing operator `[]`.) Write a procedure to append lists, which should work like this:

```
append([1, 2], [3, 4])  -->  [1, 2, 3, 4]
```

Try to add error checking so that the procedure does not attempt to append arguments that are not lists.

4. Experiment interactively with the problem of removing or *deleting* an element from a list. (Note that you cannot reliably do this by converting the list to a set and using the `minus` operator. Why not? Consider a list such as `[1, 2, 1]`.) The trick here is to append the

parts of the list before and after the element to be deleted. The library function `member` determines where in a list an element appears, and `nops` returns the total length of a list. Write a procedure to delete (the first occurrence of) an element from a list, which should work like this:

```
delete(3, [1, 2, 3, 4, 3])  -->  [1, 2, 4, 3]
delete(5, [1, 2, 3, 4, 3])  -->  [1, 2, 3, 4, 3]
```

Try to add error checking so that the procedure does not try to delete an element from an argument that is not a list.

5. This question arose from an attempt to use Maple for my daughter's homework.

Write a set of functions that act as subset predicates, i.e. that return Boolean values and implement the relations $\subset, \subseteq, \supset, \supseteq$, *assuming* that their arguments are both sets. Use function names that begin with `&` so that the functions can be used as binary infix operators. For example, they should behave like this:

```
> {1,2} &subset {1,2,3}
                              true
> {1,2} &subset {1,2}
                              false
```

The operations can be defined in terms of set intersection and equality.

Now generalize your definitions so that if either operand is manifestly not a set then an error is reported, and if the operands are not both explicit sets but could later evaluate to sets then the operators remain symbolic. Try to do this without rewriting the same code several times.