Computer Algebra Support Project

An Introduction to Maple

David Harper

Astronomy Unit, School of Mathematical Sciences, Queen Mary and Westfield College, University of London, Mile End Road, London E1 4NS

Rachid Chalabi

Computer Laboratory, The University, PO Box 147, Liverpool, L69 3BX

Third edition
November 1992

Acknowledgement

I would like to thank Chris Wooff, Peter Mallinson and David Hodgkinson who have read this document and its earlier drafts and have offered many useful suggestions for its improvement. In addition, Sean Dunne of Manchester Computing Centre and Alasdair Buchanan of Leeds University have drawn my attention to several inconsistencies and mistakes. If any errors still lurk, I would be very grateful to hear about them. Please mail error reports and comments to *D.Harper@qmw.ac.uk*

David Harper
Astronomy Unit, School of Mathematical Sciences, QMW
May 1989, October 1990, November 1992, September 1996

I would like to thank the Computer Laboratory for providing the necessary environment in updating this guide.

Rachid Chalabi
Computer Laboratory, University of Liverpool, PO BOX 147, Liverpool L69 3BX
January 1992

Publication history

1st edition: May 1989; in TEX : June 1990.

2nd edition: January 1992.

3rd edition: November 1992. Minor revisions: September 1996.

Preface

This document describes Maple version 5.0 which was released in May 1991. Some of the examples, however, were produced using earlier versions and so they may differ slightly from the results obtained using a later version. If an example fails to work, please check the *Maple Reference Manual* for your version of Maple for clarification.

Copyright Notice

TABLE OF CONTENTS

1 Maple Basics

This section describes the basic features of the Maple computer algebra system. It explains how to perform elementary algebra on polynomials, transcendental functions and matrices.

1.1 Basic Syntax

Maple's syntax is similar to that of other computer algebra systems such as REDUCE and MACSYMA. It is also similar to programming languages like C and Pascal. The following examples show how simple arithmetic is performed. This example, and all other examples, are shown as they would appear in a real Maple session. The input line ends with a semicolon and Maple's response is indented to the right with no semicolon.

```
a := 5/2;
                     a := 5/2

b := 6!;
                     b := 720

c := a*b;
                     c := 1800

d := c**2;
                     d := 3240000

e := d-12345;
                     e := 3227655
```

In Maple, the arithmetic operators are:

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`**` *or* `^`	Exponentiation

All statements, expressions and commands must end with a semicolon or a colon. This is the signal to Maple to evaluate the statement. A Maple statement can extend across many lines until terminated with a colon or semicolon. This is particularly useful when we need to enter a long algebraic expression which will not fit easily onto a single line. It also allows us to put several commands on the same line, each command being terminated with a semicolon or colon.

If a Maple statement is terminated with a semicolon then Maple will display the result of evaluating the statement. If a colon is used instead, Maple will evaluate the statement but it will *not* display the result. This feature is usually used to suppress the result of long expressions during the intermediate stages of a calculation.

An algebraic expression may or may not be assigned to a variable. The assignment symbol is `:=` (not `=`, which has another meaning in Maple). Hence if we wish to assign the value 77 to the variable called `radius` then we enter

```
radius := 77;
```

and then `radius` has the value 77.

If we want to enter a very long number or name which must be broken across the end of a line, we must terminate each line with the back-slash character (\). When Maple encounters a back-slash at the end of a line, it ignores the end-of-line and reads the next line as a direct continuation of the first. For example, these commands all assign the value 12345678901234567890 to the variable `largenumber`.

```
largenumber:=12345678901234567890;

largenumber:=123456789012\
34567890;

largenumber:=1234\
567890\
123456\
7890;
```

It would be wrong to enter

```
largenumber:=123456789012
34567890;
```

since Maple regards an end-of-line without a back-slash as equivalent to a space, so this would be the same as entering

```
largenumber:=123456789012 34567890;
```

and this would cause an error since Maple requires algebraic expressions to be separated by infix operators such as `+` and `*`.

Each statement passed to Maple has a value associated with it as a result of its evaluation. We can obtain the value of the previous statement by using a special variable represented by the double-quote character `"`. This means that it is not always necessary to assign the result of a calculation to a variable, since we can always retrieve the previous result. The following example shows this.

```
(7*5+11)**3;
                    97336

"-6;
                    97330

"/2;
                    48665

"+1;
                    48666

"/6;
                    8111
```

At no stage in this example have we saved a result in a named variable, but the previous result is always available as the value of the special variable ". This technique is widely used in Maple and it is very useful when we only require the final result of a sequence of calculations, provided that each intermediate result depends upon no more than the previous three results. The " notation is extended to allow us to use the last three results, as follows:

`"`	previous result
`""`	penultimate result
`"""`	third-to-last result

Maple also allows comments. All text following the hash symbol (#) up to the end of a line is treated as a comment and ignored. For example:

```
# This is a Maple comment
y := x**2;  # Maple will set y to x**2 but ignore this text
```

1.2 The Function Library

Maple's power and versatility is due to the wide range of functions which can be used to simplify and manipulate mathematical expressions and to perform other tasks such as input and output to files. There are several hundred functions in the Maple library. Most of these are called *standard library functions* and can be called by name any time during a Maple session. The other functions are called *miscellaneous library functions* and must be loaded from the corresponding library file before they can be used. For example, the function `FFT` is a miscellaneous library function and so we must enter the command

```
readlib(FFT):
```

before the first time we use it during a Maple session.

We can view a list of standard library functions by entering the command

```
help(index,library);
```

and a list of the miscellaneous library functions by entering:

```
help(index,libmisc);
```

If we know the name of a library function, we can ask for a description of it directly. The information that the `help` function displays about a miscellaneous library function always includes a line such as:

```
You must use readlib(FFT): before invoking FFT.
```

Maple also has a number of function packages. Each package covers a branch of mathematics such as linear algebra, statistics or number theory. We can view a list of packages by entering the command

```
help(packages);
```

The functions in each package must be loaded from files before they can be used. The easiest way to do this is to load the entire package using the `with` function. For example, to load the linear algebra package called `linalg`, enter:

```
with(linalg);
```

Maple responds by listing the names of the functions in the package which are now defined:

```
     Warning: new definition for    norm
     Warning: new definition for    trace
[BlockDiagonal, GramSchmidt, JordanBlock, add, addcol, addrow, adj, adjoint,

angle, augment, backsub, band, basis, bezout, charmat, charpoly, col,

coldim, colspace, colspan, companion, concat, cond, copyinto, crossprod,

curl, definite, delcols, delrows, det, diag, diverge, dotprod, eigenvals,

eigenvects, equal, exponential, extend, ffgausselim, fibonacci, frobenius,

gausselim, gaussjord, genmatrix, grad, hadamard, hermite, hessian,

hilbert, htranspose, ihermite, indexfunc, innerprod, intbasis, inverse,

ismith, iszero, jacobian, jordan, kernel, laplacian, leastsqrs, linsolve,

matrix, minor, minpoly, mulcol, mulrow, multiply, norm, nullspace, orthog,

permanent, pivot, potential, randmatrix, range, rank, row, rowdim,

rowspace, rowspan, rref, scalarmul, singularvals, smith, stack, submatrix,

subvector, sumbasis, swapcol, swaprow, sylvester, toeplitz, trace,

transpose, vandermonde, vecpotent, vectdim, vector]
```

In the example above, this includes functions such as `inverse`, `sylvester` and `transpose`. We could now use these functions as if they were standard library functions.

Some packages include functions whose names are the same as Maple library functions which are already defined. When a package is loaded using the `with` function, Maple prints a warning message if any existing functions have been re-defined. In the example above, the function `trace` has been given a new definition. The previous definition (a standard library function) is no longer available.

If we do not need all the functions in a package, or if we want to avoid re-defining a library function, then we can invoke an individual function using a special notation. For example, if we wish to use the `inverse` function in the `linalg` package, we could enter

```
with(linalg,inverse);
```

instead of:

```
with(linalg);
```

The `inverse` function can always be accessed directly using the following notation:

```
linalg[inverse](...);
```

1.3 Variable Names

Like any other programming language, Maple allows us to assign values to named variables. This often makes programs more readable and it allows us to save results for use later in the calculation. The rules governing valid Maple names are quite simple:

- A name can be any string of characters including letters, numbers and the underscore character (_). It cannot start with a number. It can be up to 499 characters long.

- Maple distinguishes between upper and lower case letters. Thus the names

  ```
  pvar   Pvar   PVAR   pVaR
  ```

  all refer to different variables. This can cause confusion to new users since the same rule applies to names and commands defined by the Maple system itself. Most Maple functions and commands have names which are in lower case. For example, the command to leave Maple is `quit` but Maple regards `QUIT` as just another variable name.

  A few Maple functions and pre-defined variables have names which include upper-case letters. One example is `Pi` which represents the mathematical constant π.

  Users should also avoid using names starting with underscore as they may have been used by Maple as global names.

- Some names are already defined by Maple, either as keywords (e.g. `if`, `then`, `else`, `do`, `quit`) or as function names (e.g. `sin`, `cos`, `log`, `diff`, `bessel`) or as pre-defined mathematical values (e.g. `Pi`, `E`, `I`, `GAMMA`). You should avoid using these names as variables – it will often cause an error.

- It is possible to create names which contain characters such as blanks and algebraic operators by enclosing the name in back-quotes (‘). This technique is used extensively by the Maple system itself to create names such as `diff/bessel` (this is explained in Chapter 6) but it may also be used for effect. For example

  ```
  `Standard rate of tax ` := 0.25;

                     Standard rate of tax := 0.25
  ```

  Note how Maple removes the back-quotes but leaves the spaces in the string `Standard rate of tax`.

  During routine calculations, this technique is best avoided since it may be confusing, even to the author of the program!

1.4 String Manipulation – Constructing Names Dynamically

The period (.) character has a special meaning when used in Maple names. It performs concatenation and it can be used to build names. For example:

```
j:=2;
                          j := 2

q:=Wombat.j;
                      q := Wombat2

r:=Wombat.P;
                      r := WombatP

k:=4;
                          k := 4

s:=Wombat.j.k;
                     s := Wombat24

t:=Wombat.P.k;
                     t := WombatP4
```

The leftmost string is used literally as a stem, but all strings or expressions after the first period are evaluated and concatenated to the stem. We can force Maple to evaluate all expressions (including the first) before concatenation by using the `cat` function. For example:

```
j:=4; k:=1;
                          j := 4

                          k := 1

j.k;
                           j1

cat(j,k);
                           41

cat(j,Wombat,k);
                       4Wombat1
```

Note that the object that `cat` returns is always a string. The expression '41' in the example above cannot be used as the number 41 because it is a string.

Constructed names can also be used on the left-hand side of an expression. The same concatenation rules apply so that the first element of the name on the left-hand side is evaluated literally. This method allows us to create names dynamically and assign values to them. For example,

```
j:=4; k:=1;
                          j := 4

                          k := 1
```

```
p.j := x^j;
                                       4
                                p4 := x


q.j.(k+1) := (1+x^j)^(k+1);

                                           4 2
                              q42 := (1+x )
```

1.5 Numbers

In common with most other computer algebra systems, Maple represents numerical coefficients and indices as integers or ratios of integers by default. Integer arithmetic is exact - there is no truncation, so Maple can manipulate and generate integers of arbitrary length. This is well illustrated by the factorial operator, denoted in Maple by an exclamation mark `!` following its operand. We may evaluate the factorial of 250 as follows:

```
    250!;
323285626090910773232081455202436847099484371767378066674794242711282374755511\
120948881791537102819945092850735318943292673093171280899082279103027907128192\
167652724018926473321804118626100683292536513367893908956993571353017504051317\
876007724793306540233900616482555224881943657258605739922264125483298220484913\
772177665064127685880715312897877767295191399084437747870258917297325515028324\
178732065818848206247858265980884882554880000000000000000000000000000000000000\
0000000000000000000000000
```

This gives a number which is 493 digits long.

When evaluating fractions which are the ratios of integers, Maple will always extract the highest common factor from the numerator and the denominator in order to reduce the fraction to its lowest form. For example:

```
     12!;
                  479001600

     2**20;
                  1048576

     12!/2**20;
                  467775
                 --------
                   1024
```

The use of arbitrary-precision integer arithmetic ensures that numerical coefficients in Maple expressions are always exact. However, we sometimes want a numerical result in floating-point form. Maple can also perform floating-point arithmetic to any precision, but we must specify the number of significant figures which we wish Maple to retain in floating-point calculations. The special variable `Digits` controls the precision of floating-point calculations. Its initial value is 10, so floating-point results are calculated to 10 figures, but we can change `Digits` to any positive integer value we wish.

Floating-point numbers can be used in place of integers and fractions in any algebraic expression, as the following example shows:

```
1.2*x + 5*x**3;
                                    3
                  1.2 x + 5 x

expand("**2);
                             2           4         6
                  1.44 x  + 12.0 x  + 25 x

"*3/7;
                               2                    4           6
              .6171428572 x  + 5.142857143 x  + 75/7 x
```

Notice that floating-point numbers are 'contagious': integers and fractions multiplied by floating-point numbers produce floating-point numbers, even if the result could be represented by an integer or fraction. The coefficient of x^4 in the example above shows this.

Numerical values of expressions and functions can be obtained in floating-point form using the function `evalf` as illustrated by the following examples:

```
evalf(Pi);
                  3.141592654

evalf(2**(1/3));
                  1.259921050

Digits := 20;
                  Digits := 20

evalf(log(sin(Pi/8)));
                  -0.96054717892973049468

evalf(Pi,100);
                 3.14159265358979323846264338327950288419716939 9\
                    37510582097494459230781640628620899862803482 5\
                    342117068
```

In the last example, we have given a second argument to `evalf`. This temporarily overrides the current setting of `Digits` to display π to 100 digits.

In addition to `evalf`, Maple provides a function called `evalhf` which is used to evaluate an expression in "double precision" floating-point mode using the hardware floating-point capabilities of the host system. This produces results to a precision of typically 16 digits. Unlike `evalf`, the argument to `evalhf` *must* evaluate to a numeric value. The `evalhf` function is useful for gaining speed in intensive numerical computations, where accelerations of one or two orders of magnitude are possible.

Maple can also perform complex arithmetic. It recognises the special name `I` as the mathematical $i \equiv \sqrt{-1}$. The function `evalc` must be used to evaluate complex expressions.

```
1 - 3*I;
                  1 - 3 I
```

```
"**2;
                         2
              (1 - 3 I)

"/(3 + 4*I);
                        2
              (1- 3 I)
             ----------
               3 + 4 I

evalc(");
              48      14
           -  --  +   -- I
              25      25
```

Unless the function `evalc` is used, Maple will treat `I` as a simple variable with the same algebraic properties as any other variable.

1.6 Basic Expression Manipulation

Maple tends to perform very little simplification on expressions as they are entered. The way in which an expression is to be manipulated is left largely up to the user. The most elementary functions which can be applied to an expression are `expand`, `simplify`, `factor`, `collect`, `convert`, `normal`, `combine`, `radsimp` and `sort`.

The function `expand` multiplies all the factors in an expression. Its simplest use is in expanding products in polynomials.

```
(1+x)**3;
                        3
               (1 + x)

expand(");
                               2    3
                1 + 3 x + 3 x   + x

(1+x)*(1-x)*(x-2)**2;
                                        2
                (1 + x) (1 - x) (x - 2)

expand(");
                    2               4      3
               -3 x   - 4 x + 4 - x   + 4 x
```

Maple does not generally write polynomial terms in strictly ascending or descending order, as the example illustrates.

The `expand` function will also perform certain operations on expressions involving transcendental functions. For example:

```
expand(sin(x+y));
               sin(x) cos(y) + cos(x) sin(y)
```

```
expand(ln(x/(1-x)**2));
               ln(x) - 2 ln(1 - x)
```

The function `factor` is used to factorise multinomials. For example:

```
factor(-3*x**2-4*x+4-x**4+4*x**3);
                                        2
               - (x - 1) (1 + x) (x - 2)

factor(a**5-x**5);
                         4    3     2  2      3    4
               (a - x) (a  + a x + a  x  + a x  + x )
```

The function `simplify` applies a number of simplification rules to an expression. These include evaluation of radicals, special values of transcendental functions and relations between transcendental functions. The simplest use of the function is illustrated by the following examples.

```
simplify(ln(a**x));
               x ln(a)

simplify(sqrt(-9));
               3 I

simplify((-9)**(1/4));
                   1/4  1/2
               (-1)    3

evalc(");
                    1/2  1/2         1/2  1/2
               1/2 2    3    + 1/2 2    3    I

factor(");
                    1/2  1/2
               1/2 2    3    (1 + I)
```

It is possible to tailor `simplify` and `expand` to apply specified rules to functions which you have defined. This is described in Section 6.

The `collect` function applied to a polynomial in x will collect all the coefficients with the same power of x. For example:

```
collect( (x+1)*(x+2), x );
                            2
                           x  + 3 x + 2
expand((x+y+a)^3);

                  3      2        2          2
                 x  + 3 x  y + 3 x  a + 3 x y  + 6 x y a +

                         2    3      2          2    3
                     3 x a  + y  + 3 y  a + 3 y a  + a
```

```
collect(",x);
```

```
         3                2                2      2
        x  + (3 a + 3 y) x  + (6 y a + 3 a  + 3 y ) x +

                               2    3    3      2
                          3 y a  + a  + y  + 3 y  a
```

Maple also provides a function for carrying out general conversions. Two types of conversions are available: data-type conversion and form conversions. As an example of data-type conversions, we have:

```
convert([x,y], set);
```

```
                                {x, y}
```

```
convert(sin(x)+y-x,`*`);
```

```
                            - sin(x) y x
```

Form conversions are more numerous. Among some of the conversions, we have

```
convert(x^3-3*x^2+7*x+9,horner, x);
```

```
                       9 + (7 + (- 3 + x) x) x
```

```
convert(0.125,rational);
```

```
                                1/8
```

A complete list of the conversion types available is:

```
`+`          `*`        D         array      base       binary
confrac      decimal    degrees   diff       double     eqnlist
equality     exp        expln     expsincos  factorial  float
fraction     GAMMA      hex       horner     hostfile   hypergeom
lessthan     lessequal  list      listlist   ln         matrix
metric       mod2       multiset  name       octal      parfrac
polar        polynom    radians   radical    rational   ratpoly
RootOf       series     set       sincos     sqrfree    tan
vector
```

where any of the above represent the second argument to the function convert.

A different type of simplification is introduced by the `normal` function. It converts expression into rational form, where the numerator and denominator are relatively prime polynomials with integer coefficients. The following examples will clarify its use:

```
normal( (x^2-y^2)/(x-y)^3 );
```

```
                x + y
              --------
                     2
              (x - y)

normal( sin(x*(x+1)-x) );

                    2
                sin(x )
```

For many functions, the transformations applied by `expand` can be reversed by the `combine` function. As the name implies, this combines terms in sums, products and powers into a single term. Applied to lists, sets and relations, its action is recursive. One of the most useful `combine` options is `trig`, which converts powers and products of sines and cosines into functions of compound arguments. For example,

```
combine(sin(a)*cos(b) + cos(a)*sin(b),trig);

              sin(a+b)
```

To simplify expressions containing radicals, Maple provides the `radsimp` function. For instance, we have:

```
radsimp((1 + 2*x + x^2)^(1/2));

              1 + x
```

In Maple, polynomials are not always stored in sorted order. Internally, Maple stores them in the order they are created. To sort them, the user must call the `sort` function. A simple example is:

```
sort(1+x+x^2);
               2
              x  + x + 1
```

1.7 Mathematical Functions

Maple recognises a wide range of mathematical functions and their properties. A partial list of these functions is shown below.

Trigonometric and Hyperbolic Functions

sin	arcsin	sinh	arcsinh
cos	arccos	cosh	arccosh
tan	arctan	tanh	arctanh
sec	arcsec	sech	arcsech
csc	arccsc	csch	arccsch
cot	arccot	coth	arccoth

Logarithm and Exponential Functions

`log, ln`	Natural logarithm
`log10`	log to the base 10
`exp`	Exponential function
`W`	The function such that $W(x)e^{W(x)} \equiv x$

Gamma and Related Functions

`GAMMA`	Gamma function
`Beta`	Beta function
`lnGAMMA`	logarithm of Gamma function
`Psi`	Polygamma function (derivatives of `GAMMA`)

Integral Functions

`Ci`	Cosine integral
`Si`	Sine integral
`Ei`	Exponential integral
`Erf`	Error function
`Erfc`	Complementary error function $= 1 - \mathrm{erf}(x)$
`FresnelS`	Fresnel Sine integral, $\mathrm{S}(z) = \int_0^z \sin\left(\frac{\pi}{2}t^2\right)\,\mathrm{d}t$
`FresnelC`	Fresnel Cosine integral, $\mathrm{C}(z) = \int_0^z \cos\left(\frac{\pi}{2}t^2\right)\,\mathrm{d}t$
`Fresnelf`	$\left(\frac{1}{2} - \mathrm{S}(z)\right)\cos\left(\frac{\pi}{2}t^2\right) - \left(\frac{1}{2} - \mathrm{C}(z)\right)\sin\left(\frac{\pi}{2}t^2\right)$
`Fresnelg`	$\left(\frac{1}{2} - \mathrm{C}(z)\right)\cos\left(\frac{\pi}{2}t^2\right) + \left(\frac{1}{2} - \mathrm{S}(z)\right)\sin\left(\frac{\pi}{2}t^2\right)$
`dilog`	Dilogarithm function

Bessel Functions

`BesselI`	Modified Bessel function of the first kind, $I_\nu(x)$
`BesselJ`	Bessel function of the first kind, $J_\nu(x)$
`BesselK`	Modified Bessel function of the second kind, $K_\nu(x)$
`BesselY`	Bessel function of the second kind, $Y_\nu(x)$

Integer functions

`factorial`	Factorial function
`harmonic`	Harmonic numbers
`zeta`	Riemann ζ function and its derivatives
`bernoulli`	Bernoulli numbers
`binomial`	Binomial coefficients
`euler`	Euler numbers and polynomials
`fibonacci`	Fibonacci numbers
`stir1`	Stirling numbers of the first kind
`stir2`	Stirling numbers of the second kind

Simplification and expansion rules are defined for many of these functions. For example, when `expand` is applied to trigonometric functions, compound arguments are removed and the expression is re-written in terms of powers of sine and cosine of single variables. For example:

```
expand(sin(3*x));
                                         2
               4 sin(x) cos(x)  - sin(x)

expand(cos(x+y));
               cos(x) cos(y) - sin(x) sin(y)

expand(cot(x-a));
               cos(x) cos(a) + sin(x) sin(a)
              -------------------------------
               sin(x) cos(a) - cos(x) sin(a)
```

The `simplify` function can also apply appropriate identities and special values to some mathematical functions. For example:

```
simplify(sin(Pi/4));

                    1/2
               1/2 2

simplify(expand(sin(x+Pi/4)));

                           1/2                 1/2
               1/2 sin(x) 2    + 1/2 cos(x) 2
```

The knowledge of mathematical functions is also embodied in the functions `diff`, `evalc`, `evalf` and `series`. In the case of the `sin` function, Maple knows that

```
diff(sin(a(x)),x);

                    cos(a(x))*diff(a(x),x)
```

1.8 Arrays, Matrices and Vectors

Maple provides a facility for performing matrix algebra, vector algebra and calculus. Matrices and vectors are represented by Maple arrays. Arrays are in fact a special case of a Maple table, with zero or more dimensions, where each dimension is an integer range. An array must be declared before it can be used. The `array` function is used to create a Maple array. In its simplest form, its arguments are the dimensions of the array. To declare a 2×2 array, that is a 2×2 empty matrix, we would enter:

```
A := array(1..2, 1..2);
               A := array(1..2, 1..2, [])
```

The empty square brackets in Maple's response indicate that no values have yet been assigned to the components of the array. Similarily to the previous example, to declare a empty vector (i.e. a one dimensional array whose indices starts at one), one needs only to specify the arrays dimension. To declare a two-element vector, we would write:

```
V := array(1..2);

                        V := array(1 .. 2, [])
```

We can set individual components using array notation. Continuing the matrix example:

```
A[1,1] := x;
               A[1,1] := x

A[1,2] := y; A[2,1] := z; A[2,2] := w;
               A[1,2] := y
               A[2,1] := z
               A[2,2] := w
```

At this point, A represents the matrix:

$$\begin{pmatrix} x & y \\ z & w \end{pmatrix}$$

The same result can be achieved by assigning the components at the same time as we declare the array. We use another form of the `array` function:

```
A := array(1..2, 1..2, [[x,y],[z,w]]);
              [ x  y ]
          A:= [      ]
              [ z  w ]
```

Matrix algebra can be performed using the `evalm` function. A matrix (or vector) expression including the operators

`+`	Matrix addition
`-`	Matrix subtraction
`*`	Scalar × matrix multiplication
`&*`	Non-commutative matrix multiplication
`/`	Matrix inversion
`**` *or* `^`	Matrix exponentiation

can be evaluated using `evalm`. This allows matrix and vector algebra to be performed using almost the same notation as in scalar calculations. Matrix and array manipulation are described in greater detail in Chapter 4.

1.9 Expression Sequences

An expression sequence is a sequence of Maple expressions separated by commas. An expression sequence may be returned by some Maple functions (e.g. `solve`, `op`) and we can construct expression sequences explicitly using the Maple sequence operator `$`. This operator can be used in several ways, as illustrated by the following examples.

```
'p(i)' $ 'i'=1..5;
            p(1) , p(2) , p(3) , p(4) , p(5)

'j^2' $ 'j'=1..3;
                 1 , 4 , 9

x$4;
                 x , x , x , x

$3..6;
                 3 , 4 , 5 , 6
```

The previous examples represent the different uses of the `$` operator. More formally, we have the following calling sequence:

```
expr $i =m..n;
expr $ n;
$ m..n;
```

where the parameters `expr` and `m,n` represents expressions and `i` represent an unevaluated name. This indicates that both `expr` and `i` should be enclosed in single quotes to prevent a premature evaluation. This rule is to be remembered if one wants to avoid unexpected results. The third example above, which generates a sequence of x's, is very useful when used with the differentiation function `diff` which is described in the next chapter.

Maple also provides a sequence function `seq` which allows us to write the examples above in a different way:

```
seq('p(i)','i'=1..5);

seq('j^2','j'=1..3);

seq('i','i'=3..6);
```

However, the example which generates the sequence `x,x,x,x` cannot be produced using the `seq` function.

1.10 Sets and Lists

Maple provides set and list data types. These are used by a variety of Maple functions. Some functions such as `solve` (the equation solving function) yield a list or a set as their value. Lists and sets are sequences of Maple objects. These objects are often algebraic expressions but they can be any combination of Maple data types (see Chapter 5 for a discussion of data types.)

A list is an *ordered* sequence of objects enclosed in square brackets. The same object can appear more than once in a list. The following examples are all valid lists.

```
[1, 2, 3, 4]
[1, 2, 1, 4]
[a, sin(x), 5]
[a=1, b=6..infinity]
[]
```

The last example is the empty list. Elements of a list can be extracted using the `op` function or using subscript notation similar to array notation. In addition, the `nops` function gives the number of elements in a list. We can use list elements in the same way as any Maple expression. For example:

```
p := [a, 7, sin(x)];
                p := [a, 7, sin(x)]

nops(p);
                3

op(1,p);
                a


p[1];
                a

op(3,p);
                sin(x)

p[3] + p[2] - 2*p[1];

                sin(x) + 7 - 2 a
```

A list may contain other lists. For example:

```
p := [a, 7, [u, v]];

                p := [a, 7, [u, v]]

nops(p);
                3

p[3];
                [u, v]
```

```
p[3][1];
                  u
```

Operations on `list`, such as concatenation, are done in Maple through the operands of the `list`. They can be retrieved in one block, as seen in the following example:

```
p[];
                  a, 7, [u, v]
```

The `list` structure is used in the `array` data type. It constitutes the third operand of the data type. Its components could be accessed via the indexing operation.

A set is an *unordered* sequence of objects enclosed in braces (curly brackets). Each element is unique, since Maple automatically deletes duplicate elements. The order of elements in a set is determined by Maple in an 'arbitrary' way. The order in which Maple lists the elements of a set may not be the same as the order in which they were entered.

Elements can be selected from a set in the same way as a list. For example:

```
p := {a, 7, sin(x)};
               p := {a, sin(x), 7}

p := {a, 7, sin(x), a**2, 7};
                                2
               p := {a, sin(x), a , 7}

nops(p);
               4

op(3,p);
                2
               a

p[3];
                2
               a
```

Lists are not extensible. Once defined, their components can be changed using the `subsop` function (described in Chapter 5) but it is not possible to add extra components. However, sets can be manipulated using the set operators `union`, `intersect` and `minus`. For example:

```
p := {a, 7, a**2, sin(x)};
                               2
              p := {a, sin(x), a , 7}

q := {b, cos(x), 7};
              q := {b, cos(x), 7}

p union q;
                                      2
              {b, cos(x), a, sin(x), a , 7}

p intersect q;
```

```
                {7}

r := p minus q;
                              2
                {a, sin(x), a }
```

The function `member` can be used to test whether an object is a member of a set or list. Continuing the previous example:

```
member(a**2,r,'s');
               true

s;
               3
```

The variable s is set to the position of a^2 within the list r. The name s is enclosed in single quotes to prevent Maple from trying to evaluate it before assigning the value 3 to it.

When `member` is used in this way on a list, it will only return the position of the first occurrence (if any) of the object within the list. This restriction cannot occur with sets of course, because each element is unique.

1.11 Substitution

It is often useful to be able to substitute values temporarily for one or more variables within an expression. For example, we may want to know the value of $(a+bx)^2$ when $a=1$ and $b=4$. One way to do this is as follows:

```
p := expand((a+b*x)**2);
                    2               2  2
             p := a  + 2 a b x + b   x

a := 1; b := 4;
             a := 1
             b := 4

p;
                              2
             1 + 8 x + 16 x
```

However, the original value of p has now been lost. We cannot even regain it by re-typing our first command since a and b no longer stand for themselves.

The substitution function `subs` provides a more flexible method. With the original value of p above, we could continue as follows:

```
subs(a=1,b=4,p);
                             2
            1 + 8 x + 16 x
```

In this case, p still retains its original value and a and b have not been altered.

The list of substitution values can be of any length. We may also substitute for simple expressions such as $\sin x$. For example:

```
p := expand(sin(2*x+y));
                                                        2
      p := 2 cos(y) sin(x) cos(x) + 2 sin(y) cos(x)  - sin(y)

subs(sin(x)=sx,cos(x)=cx,sin(y)=sy,cos(y)=cy,p);

                                  2
               2 cy sx cx + 2 sy cx  - sy
```

In such cases, the substitution may not be complete since Maple substitutes by pattern matching and this depends upon the internal representation of the expression. In other words, only sub-expressions that can be extracted with the function `op` are substituted. This "syntactic substitution" is not followed by an evaluation.

Maple provides another substitution function, `subsop`, which can be used to replace specified operands of an expression. This is described in greater detail in Chapter 5.

1.12 Sums and Products

Maple can evaluate sums and products of sequences over ranges which may be definite, indefinite or infinite. The functions `sum` and `product` are provided for this purpose and are used in similar ways. The following examples illustrate the use of `sum` and `product`.

To add the first 50 integers:

```
sum('i','i'=1..50);
                       1275
```

To add the first N integers, where N is arbitrary:

```
sum('i','i'=1..N);
                          2
               1/2 (N + 1)   - 1/2 N - 1/2

factor(");
               1/2 N (N + 1)
```

To sum $x^n/n!$ for n from 0 to infinity:

```
sum('x**n/n!','n'=0..infinity);
                         exp(x)
```

To form the product of (x+1), (x+2), ..., (x+6):

```
product('(x+i)','i'=1..6);
     (x + 1) (x + 2) (x + 3) (x + 4) (x + 5) (x + 6)
```

Notice that the index variable and the expression are enclosed in single quotes to prevent Maple from trying to evaluate them before calling the function. This is a precaution in case the index variable has previously been assigned a value. It is good practice to include the quotes when using `sum` and `product`.

1.13 Program Control and Looping

Maple provides a statement analogous to the Fortran block-IF statement to enable us to execute a sequence of commands only if a specified condition is met. The if-statement may take one of several forms:

1 Simple if-statement.

`if` *expression* `then` *statement sequence* `fi;`

2 If-statement with an else clause.

```
if  expression then
     statement sequence
else
     statement sequence
fi;
```

3 If-statement with an else-if clause.

```
if  expression then
     statement sequence
elif  expression then
     statement sequence
fi;
```

4 If-statement with both else and else-if clauses.

```
if  expression then
     statement sequence
elif  expression then
     statement sequence
else
     statement sequence
fi;
```

where *expression* represents any expression which has a logical (Boolean) value and *statement sequence* represents any valid Maple statement or sequence of statements separated by colons or semicolons. The

```
elif  expression then
     statement sequence
```

construction may be repeated any number of times.

Note that an `if` statement must always be terminated by a matching `fi;`. The keywords `else`, `elif` and `fi` terminate the statement sequence of the preceding conditional clause and so they act as a statement terminator for the last statement of that sequence, in place of a semicolon or colon.

Each `if` statement has a value which is the value of the last statement of the conditional clause which is executed. If none of the conditional clauses is executed (because none of the logical conditions are true) then the statement returns the special value `NULL`, which represents an empty (zero-length) expression sequence.

The following examples illustrate Maple `if` statements:

- A very simple case.

```
if a=b then print(`a and b are equal`) fi;
```

- Two Maple statements are executed if a=b.

```
if a=b then
     print(`a and b are equal`);
     c := sin(a)
fi;
```

- A simple case statement.

```
if a=b then
     print(`a and b are equal`)
elif a<b then
     print(`a is less than b`)
else
     print(`a is greater than b`)
fi;
```

- Nested `if` statements. Note that each `if` needs a closing `fi`.

```
if a=b then
     if n>32 then
          a := 0; b := b+1
     else
          a := a+1; b := b-1
     fi
else
     a := 0; b := 1
fi;
```

In these examples, indentation has been used to make the program more readable. It is also easier to spot a missing `fi`, especially when `if` statements are nested.

Maple also provides a facility for repetition of a sequence of statements. This is the `do` statement. Its two general forms are as follows:

`for` *name* `from` *i1* `by` *i2* `to` *i3* `while` *expression*
  `do` *statement sequence* `od;`

`for` *name* `in` *x* `while` *expression*
  `do` *statement sequence* `od;`

In the first form, *name* is the name of an index variable. The `from` clause specifies the starting value of this variable, the `by` clause specifies the index increment, and the `to` clause specifies the end value of the index. In addition, the `while` clause specifies any expression which has a logical (Boolean) value.

Any of the `for`, `from`, `by`, `to` and `while` clauses may be omitted. Default values `from 1` and `by 1` will be used by Maple if the `from` or `by` clauses are omitted.

The sequence of Maple statements between `do` and `od;` will be executed whilst the index variable takes values from *i1* to *i3* in steps of *i2*, and whilst the expression in the `while` clause (if present) is true. If no termination condition (a `to` or `while` clause) is specified, the loop may be an endless loop.

In the second form of the do statement, the index of iteration takes each value from the list of items represented by x. This may be a list or set, but it can also be any valid Maple expression since the values are selected from x using the op selector function described in Chapter 5.

Execution of a do loop is interrupted if the index variable has taken its final value or if the while clause becomes false. It can also be interrupted if the statement sequence contains a quit command, or a RETURN statement if the do loop is within a procedure definition. We can also interrupt a do loop by including a break command. This tells Maple to leave the current do loop immediately and continue at the next command following the loop.

We can tell Maple to skip to the next iteration of a do loop using the next command. When this is encountered, execution of the statement sequence is halted and Maple proceeds with the next iteration of the current loop. The termination condition will be tested before the iteration begins, of course, so next may result in an exit from the do loop if the termination condition is met.

Note that a do loop must always have a matching od; at the end. Like the fi which terminates an if block, od terminates the statement sequence of a do loop and acts as the statement terminator for the last statement in the sequence. The following examples illustrate the Maple do statement.

1 Simple repetition.

```
for i from 1 by 3 to 10 do
     print(i)
od;
                              1
                              4
                              7
                             10
```

2 The by clause is omitted. The default is by 1.

```
for i from 1 to 5 do
     print(i**2)
od;
                              1
                              4
                              9
                             16
                             25
```

3 The while clause is included.

```
for i from 1 to 5 while i**2<20 do
     print(i**3)
od;
                              1
                              8
                             27
                             64
```

4 Repetition without an iteration index.

```
p := true;
while p do
     x := f(x,y);
     if x<0 then p := false fi
od;
```

This do statement repeats until $f(x,y)$ returns a negative (numerical) value.

5 Repetition with a break statement.

```
for j from 1 by 2 do
     if j>10 then break fi;
     ......
     ......
od;
```

is equivalent to:

```
for j from 1 by 2 to 10 do
     ......
     ......
od;
```

6 Repetition with a next command.

```
for k from 1 to 5 do
     if k=3 then next fi;
     print(k)
od;
                    1
                    2
                    4
                    5
```

7 Repetition using the in clause.

```
for x in [a,b,c,d] do
  print(x, sin(x))
od;
              a , sin(a)
              b , sin(b)
              c , sin(c)
              d , sin(d)
```

After a do loop has been executed, the loop index variable will be left with its value at the termination of the loop. In example 6 above, the variable k will have the value 6 which caused the loop to terminate. In example 7, the variable x will have the value d which is the last member of the list.

1.14 Operator Notation

Maple provides a notation for defining and manipulating mathematical operators. Consider the mapping

$$f : x, y \mapsto \sqrt{x^2 + y^2}.$$

This can be written in Maple using the `->` notation as follows:

```
f := (x,y) -> sqrt(x^2+y^2);
```

```
                                      2     2
                  f := (x,y) -> sqrt(x   + y )
```

In this form, the name `f` can be used as if it were a function, so we can write

```
f(3,4);
```

```
                              5
```

```
f(p,q);
```

```
                          2     2 1/2
                        (p   + q )
```

```
simplify(f(sin(x),cos(x)));
```

```
                              1
```

Internally, Maple stores this mapping as a procedure. Maple procedures are described more fully in Chapter 6. In this section, we will concentrate upon the operator notation and its uses.

Consider the mapping

$$f : x \mapsto \sin x,$$

which can be defined in Maple using the command

```
f := (x) -> sin(x);
```

and the mapping

$$g : x \mapsto \arcsin x,$$

which is written in Maple's operator notation as

```
g := (x) -> arcsin(x);
```

Maple denotes operator composition using the at-sign, `@`. Thus

```
f @ g;
```

```
                         sin@arcsin
```

```
simplify(");
```

```
                           x -> x
```

Repeated application of an operator, i.e. composition with itself N times, may be written using the `@@` operator (cf. multiplication using `*` and exponentiation using `**`). For example, `f@f` can be written as `f@@2`, so we may continue the previous example:

```
(f@@2)@g;
```

```
                         (2)
                      sin    @arcsin
```

```
simplify(");
```

```
                           sin
```

```
f @ (g@@2);
```

```
                                   (2)
                      sin@arcsin
```

```
simplify(");
```

```
                          arcsin
```

Maple is also able to calculate the derivatives of mappings using the `D` operator. This is explained in more detail in Chapter 2.

2 Calculus

Calculus is one of the most important branches of applied mathematics. Most physical problems involve differentiation and/or integration and therefore Maple offers facilities for performing these operations upon a very wide range of mathematical functions. In addition, Maple can solve certain types of ordinary differential equation.

2.1 Differentiation

Maple can calculate the derivative of any expression containing polynomial terms and functions 'known' to the system (in the sense of Section 1.7). Differentiation is performed using the `diff` operator. Its simplest form is

```
diff(expression, variable);
```

which differentiates `expression` with respect to `variable`. For example:

```
diff(sin(x),x);
                              cos(x)
```

Higher derivatives may be calculated with a single `diff` simply by repeating the variable. To form the second derivative of $\sin x$ we enter

```
diff(sin(x),x,x);
                              -sin(x)
```

and to form the third derivative, enter:

```
diff(sin(x),x,x,x);
                              -cos(x)
```

We can use the sequence operator `$` to simplify input of higher derivatives. Remember that `x$3` is equivalent to `x,x,x` so we may write the third derivative of $\sin x$ as:

```
diff(sin(x),x$3);
                              -cos(x)
```

Hence

```
diff(expression, variable$N);
```

returns the $\mathtt{N}^{\mathrm{th}}$ derivative of `expression` with respect to `variable`.

If the expression is a function of more than one independent variable then `diff` returns the partial derivative. In fact, it is defined to be a partial differentiation function. The following examples illustrate partial differentiation using `diff`.

```
p := exp(a*x)*sin(b*y);
                   p := exp(a x) sin(b y)

diff(p,x);
                   a exp(a x) sin(b y)

diff(p,y);
                   b exp(a x) cos(b y)

diff(p,x,y);
                   a b exp(a x) cos(b y)

diff(p,x,x,y);
                    2
                   a  b exp(a x) cos(b y)

diff(p,x$2,y);
                    2
                   a  b exp(a x) cos(b y)
```

We can also apply `diff` to equations, sets, lists and unevaluated functions. The argument list of the function must contain an expression which depends upon the differentiation variable, otherwise the function is assumed to be independent of the variable, and hence its derivative is zero. The following examples illustrate this.

```
diff(p(x),x);
                     d
                    --  p(x)
                    dx

diff(p(x),x,x);
                     2
                    d
                   --- p(x)
                     2
                   dx

diff(p(x),y);
                   0
```

No evaluation can occur since the mathematical properties of p and g are not defined, so `diff` returns an unevaluated result. In the last example, the output is expressed in terms of the differential operator `D` that acts on functions.

In fact `diff` can be tailored to apply rules to evaluate derivatives of user-defined functions. This is described in Section 6.

2.2 The Abstract Differentiation Operator "D"

In the previous chapter, we introduced Maple's `->` notation which can be used to define mappings. Maple provides an abstract differentiation operator to define derivatives of mappings. It can also be used to provide a completely general notation for partial and total derivatives.

The `D` operator denotes derivatives and is applied to expressions which represent functions. For example,

```
f := (x) -> sin(x);
```

```
                              f := sin
```

```
D(f);
```

```
                                cos
```

```
(D@@2)(f);
```

```
                               - sin
```

As the example demonstrates, `D(f)` returns a function which is the derivative of `f`, and the composition operator `@@` can be applied to `D` to denote higher derivatives.

The more general form of the `D` operator enables us to define partial derivatives. For example, consider the mapping

$$q : x, y \mapsto \exp(xy),$$

which can be written in Maple's `->` notation as

```
q := (x,y) -> exp(x*y):
```

We can calculate the partial derivative of this mapping with respect to its first argument as follows:

```
D[1](q);
```

```
                       (x,y) -> y exp(x y)
```

and the partial derivative with respect to its second argument is

```
D[2](q);
```

```
                       (x,y) -> x exp(x y)
```

Higher and mixed derivatives are calculated either by composition of the `D` operator or by extending the subscript list of the operator. Thus `D[1,2]` calculates the mixed second derivative of a function with respect to its first and second arguments. Applied to the current example, we find:

```
D[1,2](q);
```

```
               (x,y) -> exp(x y) + y x exp(x y)
```

The `D` operator allows us to denote partial and total derivatives unambiguously. Consider the operator `q` defined above, applied to the arguments u and v. In mathematical notation, we have

$$q(u, v) = \exp(uv),$$

and we may calculate the partial derivatives

$$q_u = \frac{\partial q}{\partial u}$$

and

$$q_v = \frac{\partial q}{\partial v}.$$

Suppose, however, that v is a function of u. In Maple's operator notation, we can still distinguish between the partial derivative with respect to the first argument, q_u, and the total derivative which is

$$\frac{\mathrm{d}q}{\mathrm{d}u} = q_u + q_v \frac{\mathrm{d}v}{\mathrm{d}u}.$$

The first is

```
D[1](q)(u,v);

                              v exp(u v)
```

and the second is

```
D[1](q)(u,v(u)) + D[2](q)(u,v(u)) * diff(v(u),u);

                                              /  d       \
               v(u) exp(u v(u)) + u exp(u v(u)) |---- v(u)|
                                              \ du       /
```

2.3 Integration

The integration of algebraic expressions is one of the principal applications of computer algebra systems. Maple can integrate a very wide range of functions. It can also evaluate definite integrals, employing numerical methods (quadrature) if an analytic result cannot be obtained.

The function `int` is used to perform integration. To evaluate the indefinite integral of F with respect to x we enter

```
int(F,x);
```

and (hopefully) Maple responds with the result. For example:

```
int(x**3,x);
                     4
                    x
                   ---
                    4

int(x**n,x);
                     (n+1)
                    x
                  -----
                   n+1

int(cos(x),x);
```

```
                    sin(x)

int(1/x,x);
                    ln(x)

int(1/sqrt(1-x**2),x);
                    arcsin(x)

int(",x);
                                         2  1/2
                    x arcsin(x) + (1 - x )

int(ln(arcsin(x)),x);
                    /
                    |
                    | ln(arcsin(x)) dx
                    |
                   /
```

In the last example, Maple is unable to evaluate the indefinite integral, and it indicates this failure by displaying an unevaluated integral. This means that the integral does not exist in closed form or, if it does, Maple cannot determine the closed form of the integral.

When Maple evaluates an indefinite integral, it does not add a constant of integration. If a constant of integration is required, it must be supplied by the user.

We can use `int` to evaluate definite integrals using the general form

```
int(F,x=a..b);
```

where a and b are the lower and upper bounds of the range of integration. The following examples illustrate definite integration.

```
int(x**3,x=a..b);
                     4       4
                    b       a
                   ---  -  ---
                    4       4

int(sin(x),x=0..p);
                    - cos(p) + 1

int(1/sqrt(1-x**2),x=0..1);
                    1/2 Pi

int(exp(-(a*x)**2),x=0..infinity);
                                  1/2
                         abs(a) Pi
                    1/2 -----------------
                                 2
                                a

int(sin(x**3),x=0..sqrt(Pi));
```

```
                                     1/2
                                   Pi
                                  /
                                 |        3
                                 |   sin(x ) dx
                                 |
                                /
                                0
evalf(");
                  .3761585203
```

The last example shows how Maple can be instructed to use numerical integration (quadrature) if a result cannot be obtained in closed form. The numerical value of the integral is returned with a precision of `Digits` digits. In the case where `Digits` is 16 or less, Maple may make use of `evalhf` to evaluate the integral more quickly.

3 Equation Solving

Maple is able to solve a variety of types of mathematical equation:

- One equation in a single unknown, i.e. roots of a function.
- Simultaneous linear equations.
- Least-squares solution of over-determined systems of simultaneous linear equations.
- Simultaneous non-linear equations.
- Ordinary differential equations.
- Simultaneous ordinary differential equations, using Laplace transforms.
- Recurrence equations.

A note of caution is required here. As with indefinite integration, equation solving is often a difficult mathematical problem and explicit solutions often cannot be found. This is especially true of systems of simultaneous non-linear or differential equations. In such cases, Maple will attempt to find a partial solution of the system. Even when a full algebraic solution can be found, it may be very complicated and of limited usefulness. As an example, the solution to the general cubic equation can easily be found by Maple, but the result which is returned by the `solve` function is a set of three very lengthy expressions.

3.1 One Equation in a Single Unknown

This problem is identical to that of finding the roots of a function. One of the commonest types of function is a polynomial. Maple can find the roots of any polynomial up to degree 4. It can also find roots of equations involving transcendental functions.

The `solve` function is used to solve algebraic equations. For example, to solve the general quadratic we enter

```
solve(a*x**2+b*x+c,x);
                          2          1/2                  2          1/2
                  - b + (b  - 4 a c)             - b - (b  - 4 a c)
             1/2 ----------------------- , 1/2 -----------------------
                           a                             a
```

In this example, the right-hand side of the equation is implicitly equal to zero, but we could specify the right-hand side explicitly using the = symbol. For example:

```
solve(sin(x)**2=1/2,x);
                        1/4 Pi , - 1/4 Pi
```

Maple returns the solution as a sequence of roots. To use these in subsequent calculations, we can make a set by enclosing the command in curly brackets. The previous example could be re-written as

```
p := {solve(sin(x)**2=1/2,x)};
                          p := { 1/4 Pi , - 1/4 Pi}
```

and we can find the number of distinct roots using the `nops` function:

```
nops(p);
                         2
```

The roots themselves are available as the elements of the set `p` and can be selected using square-bracket notation or the `op` function.

3.2 Simultaneous Linear Equations

The `solve` function can also be used to solve a set of simultaneous linear equations when the set is fully-determined, i.e. the number of (linearly independent) equations is equal to the number of unknowns. The equations and unknowns must be passed to `solve` as sets. For example:

```
eqns := { 3*x+y=1, x-y-z=4, x+2*y+3*z=8};
                    eqns = { x + 2 y + 3 z = 8,
                             x - y - z = 4,
                             3 x + y = 1}

unknowns := {x,y,z};
                    unknowns := {x, y, z}

p:=solve(eqns, unknowns);
                 p := { y = -8, x = 3, z = 7}
```

We can extract the solutions from the set using the `op` function described in Section 1. For example, to obtain the value of y in the previous example we enter

```
op(1,p);
                         y = -8

op(2,");
                         -8
```

because each element of the set of solutions is a Maple object of type = and has two parts; a left-hand side and a right-hand side, numbered 1 and 2 respectively. We seek the actual value, which is the right-hand side.

If we try to solve an over-determined or under-determined system of equations, Maple will indicate that no solutions can be found. In order to solve an over-determined system, we can use the function `leastsqrs` which is part of the Maple linear algebra package called `linalg`. We must first declare that we wish to use `leastsqrs` by entering:

```
with(linalg, leastsqrs);
                    [leastsqrs]
```

It is used in a similar way to `solve` except that all of the equations to be solved must be implicitly equal to zero and must not be given a right-hand side using =.

3.3 Simultaneous Non-Linear Equations

The `solve` function can also be used to solve systems of non-linear equations, though a solution may not be found in many cases since simultaneous non-linear equations do not generally have explicit solutions. In some cases, `solve` will return a partially solved system in which the solutions are expressed in terms of the roots of an auxiliary equation. For example:

```
eqns := { x+y=5, x**3+y**2=11};
                                          3    2
                    eqns := { x + y = 5, x  + y  = 11}

solve(eqns, {x,y});
                       3        2
       { x = -RootOf(_Z  - 16 _Z  + 75 _Z - 114) + 5,

                      3        2
         y = RootOf(_Z  - 16 _Z  + 75 _Z - 114)}
```

We can see that the equation has three roots which can be obtained by solving the cubic polynomial in the dummy variable _Z. At this stage, we must manipulate the result returned by `solve` in order to obtain the full solution.

The Gröbner basis package `grobner` contains a number of functions to help the user to find solutions of simultaneous non-linear equations. In particular, `grobner[gbasis]` returns the reduced minimal Gröbner basis of a set of equations. The `grobner[gsolve]` function produces a collection of reduced Gröbner bases of the original system in which the variables have been eliminated and separated as far as possible, so that the new equations may be solvable using `solve`.

3.4 Ordinary Differential Equations

In addition to solving algebraic equations, Maple can solve a number of different types of ordinary differential equations (ODEs). Like integration, solving ODEs is not always a simple task and many equations do not have closed-form solutions. Maple can employ a range of powerful methods to solve ODEs including Laplace transform techniques for systems of simultaneous ODEs.

The function `dsolve` is provided for solving ordinary differential equations. We must specify the differential equations to be solved using unevaluated-function notation. The following example solves the equation

$$\frac{d^2f}{dx^2} + a^2 f = 0$$

where f is regarded as a function of the independent variable x.

```
diffeq := diff(f(x),x$2) + a**2*f(x) = 0;

                         /   2       \
                         |  d        |     2
              diffeq :=  |----- f(x) | + a  f(x) = 0
                         |   2       |
                         \ dx        /

dsolve(diffeq,f(x));
```

```
f(x) = _C1 exp(I a x) + _C2 exp(- I a x)
```

Notice that in this case, Maple returns a solution explicitly, with $f(x)$ on the left-hand side of a Maple object of type =. To extract the solution (the right-hand side) we must use the op function. Continuing the example above:

```
op(2,");
                    _C1 exp(I a x) + _C2 exp(- I a x)
```

Maple has introduced two arbitrary constants into the solution; _C1 and _C2. Such constants are always included if no boundary conditions have been specified, and they are always of the form _Cn where n is a number. If we had assigned values to _C1 and _C2 before calling solve, Maple would have used _C3 and _C4 as arbitrary constants. It will always choose unused names.

We can specify boundary conditions if we wish. If the previous example is to be solved such that

$$f(0) = 0$$

and

$$f(\frac{\pi}{2a}) = 1$$

then we would enter:

```
 dsolve({diffeq, f(0)=0, f(Pi/(2*a))=1}, f(x));
                   f(x) = sin(a x)
```

If we wish to specify both f and f' at a point, we must solve the general equation and then substitute the value of t into f and f' and solve for _C1 and _C2. Consider the inhomogeneous equation in the following example.

```
diffeq := diff(f(x),x$2) + 4*f(x) = sin(x);
                         /   2      \
                         |  d       |
                diffeq := |----- f(x)| + 4 f(x) = sin(x)
                         |   2      |
                         \ dx       /

dsolve(diffeq, f(x));
                                              3
                 f(x) = - 1/3 sin(2 x) cos(x)  + 1/2 sin(2 x) cos(x)

                                              2
                 + 1/3 cos(2 x) sin(x) cos(x)  - 1/3 sin(x) cos(2 x)

                 + _C1 sin(2 x) + _C2 cos(2 x)

ff := op(2,");
                                              3
                 f(x) = - 1/3 sin(2 x) cos(x)  + 1/2 sin(2 x) cos(x)

                                              2
                 + 1/3 cos(2 x) sin(x) cos(x)  - 1/3 sin(x) cos(2 x)
```

```
                    + _C1 sin(2 x) + _C2 cos(2 x)
```

Suppose we want the solution such that $f(0) = 0$ and $f'(0) = a$. We proceed as follows:

```
subs(x=0,ff);
                   1/6 sin(0) cos(0) + _C1 sin(0) + _C2 cos(0)

simplify(");
                      _C2

eqs := {"=0};
                      eqs := {_C2 = 0}

subs(x=0,diff(ff,x));
                  4             2       2             2             2
      - 1/3 cos(0)  - 1/3 sin(0)  cos(0)  + 2/3 cos(0)  + 1/6 sin(0)

      + 2 _C1 cos(0) - 2 _C2 sin(0

simplify(");
                      1/3 + 2 _C1

eqs := eqs union {"=a};
                      eqs := {_C2 = 0, 1/3 + 2 _C1 = a}

solve(eqs,{_C1,_C2});
                      {_C2 = 0, _C1 = - 1/6 + 1/2 a}

subs(op("),ff);
                                    3
           - 1/3 sin(2 x) cos(x)  + 1/2 sin(2 x) cos(x)
                                          2
           + 1/3 cos(2 x) sin(x) cos(x)

           - 1/3 sin(x) cos(2 x) + (- 1/6 + 1/2 a) sin(2 x)

combine(",trig);

                1/3 sin(x) - 1/6 sin(2 x) + 1/2 sin(2 x) a
```

This is the solution we require. The use of the `op` function in the penultimate statement is explained in Chapter 5. In this case, it simply strips the brackets from the result returned by the `solve` function.

3.5 Systems of Ordinary Differential Equations

The dsolve function can also be used to solve systems of simultaneous ordinary differential equations. It uses Laplace transform methods. As in the case of simultaneous algebraic equations, we provide dsolve with a set of equations and a set of unknowns. As an example, consider the linked first-order ODEs:

$$\frac{dp}{dx} + q = 0$$
$$\frac{dq}{dx} - p = 0$$

We can solve these using Maple as follows:

```
eqns := {diff(p(x),x)+q(x)=0, diff(q(x),x)-p(x)=0};

               / d      \              / d      \
       eqns := {|---- p(x)| + q(x) = 0, |---- q(x)| - p(x) = 0}
               \ dx     /              \ dx     /

unknowns := {p(x),q(x)};
     unknowns := {p(x), q(x)}

dsolve(eqns, unknowns, laplace);
     { q(x) = q(0) cos(x) + p(0) sin(x),

       p(x) = p(0) cos(x) - q(0) sin(x) }
```

Notice that we included a third argument, the name laplace, in the call to dsolve to tell Maple to use Laplace transforms to solve the equations. In addition, the arbitrary constants introduced by Maple are $p(0)$ and $q(0)$. We could have specified these as part of the set of equations to be solved, but here they appear explicitly in the solution. If we did not specify the name laplace as an argument, the arbitrary constants $p(0)$ and $q(0)$ would have been replaced by _C1 and _C2.

When the Laplace transform method is used to solve systems of second (or higher) order ODEs, Maple introduces additional boundary conditions in the form of derivatives of the unknowns, using the differential operator D. It represents the first derivative as $D(y)(0)$ where y is the function name and 0 the coordinate. Hence the value of the second derivative of $y(x)$ at $x = 0$ is denoted by

```
 (2)
D   (y)(0)
```

If we only require an approximate solution, we can tell Maple to solve the equations using power series. As an example, suppose we wish to solve the previous example with boundary conditions $p(0) = 0$, $q(0) = 1$ neglecting terms of the 6th order or higher.

```
Order := 6;
     Order := 6

eqns := {diff(p(x),x)+q(x)=0, diff(q(x),x)-p(x)=0, p(0)=0, q(0)=1};
               / d      \              / d      \
      eqns := {|---- p(x)| + q(x) = 0, |---- q(x)| - p(x) = 0, p(0) = 0,
               \ dx     /              \ dx     /
```

```
               q(0) = 1}

unknowns := {p(x),q(x)};
     unknowns := {p(x), q(x)}

dsolve(eqns, unknowns, series);
                           3           5      6
     { p(x) = - x + 1/6 x  - 1/120 x  + O(x ),

                        2         4      6
       q(x)= 1 - 1/2 x  + 1/24 x  + O(x ) }
```

We can see that p is the truncated power series for $\sin x$ and q is the truncated power series for $\cos x$. The function O is a dummy 'order' function indicating the order of the first neglected term.

4 Matrix Algebra

One of the most widely-used packages in Maple is the linear algebra package `linalg`. It provides a wide range of functions for manipulating matrices and vectors and systems of simultaneous linear equations. This section describes how to use Maple to perform matrix algebra.

4.1 Representation of Matrices

Maple represents matrices using arrays. To create a matrix, we use the `array` function described in Chapter 1. We can initialise the elements of a matrix either using the extended form of the `array` function or using array element notation to assign values to individual elements. The following examples illustrate the two different approaches.

```
a:=array(1..2,1..2);
                  a := array(1 .. 2, 1 .. 2, [])

a[1,1]:=p; a[1,2]:=q; a[2,1]:=r; a[2,2]:=s;
                  a[1,1] := p
                  a[1,2] := q
                  a[2,1] := r
                  a[2,2] := s
```

This is equivalent to

```
a:=array(1..2,1..2,[[p,q],[r,s]]);

                       [ p   q ]
                  a := [       ]
                       [ r   s ]
```

and they both define a as the 2×2 matrix:

$$\begin{pmatrix} p & q \\ r & s \end{pmatrix}$$

If the elements of a matrix are not assigned values, then they retain their symbolic values as indexed names. For example:

```
a:=array(1..2,1..2);
                  a := array(1..2, 1..2, [])

p:=a[1,1];
                  p := a[1,1]
```

To display the elements of a matrix, it is not sufficient to type the name of the matrix. We must use the `eval` function. The following examples show how matrices may be displayed. Note the use of the colon as a statement terminator to suppress output.

- A matrix which has not been initialised.

```
a:=array(1..2,1..2):

eval(a);

                 [ ?[1, 1]  ?[1, 2] ]
                 [                  ]
                 [ ?[2, 1]  ?[2, 2] ]
```

- A matrix whose elements have all been initialised.

```
a:=array(1..2,1..2,[[p,q],[r,s]]):

eval(a);
                 [ p   q ]
                 [       ]
                 [ r   s ]
```

- A matrix which has been partially initialised.

```
a:=array(1..2,1..2):

a[1,1]:=p:

eval(a);

                 [ p          ?[1,2] ]
                 [                   ]
                 [ ?[2, 1]    ?[2,2] ]

a[1,2];
                 a[1,2]
```

In the last example, Maple introduces the `?` notation which indicates that some elements of the matrix have not been given explicit values. However, when we ask for the value of one of these uninitialised elements,we get the correct indexed name.

It is possible to assign values to some or all of the elements of a matrix using an alternative form of the `array` function. For example, we could set the (1,1) and (1,2) elements of a 2×2 matrix when we declare it by using the command:

```
a:=array(1..2,1..2,[(1,1)=p, (1,2)=q]);
                    [ p            q ]
                a :=[                ]
                    [ a[2,1]  a[2,2] ]
```

The `linalg` package offers a variety functions to define special types of matrix. The most general of these is the `matrix` function which can be used in several ways. Its simplest form is equivalent to the `array` function. Both `array(1..m,1..n)` and `matrix(m,n)` create an `m` by `n` matrix whose entries are initially undefined. Note that the dimensions specified in the `matrix` function are **not** given as ranges but just as the upper bounds. The lower bounds are assumed to be equal to one.

If `matrix` is given a third argument which is the name of a procedure, then this procedure will be used to calculate the initial values of the matrix components.

```
A := matrix(m,n,f);
```

is equivalent to

```
A := array(1..m,1..n):

for i from 1 to m do
  for j from 1 to n do
    A[i,j] := f(i,j)
  od
od:
```

and the resulting array is initialised so that $A_{ij} = f(i,j)$. For example,

```
matrix(3,3,(i,j)->i*x+j*y);

                   [  x + y    x + 2 y    x + 3 y  ]
                   [                               ]
                   [ 2 x + y  2 x + 2 y  2 x + 3 y ]
                   [                               ]
                   [ 3 x + y  3 x + 2 y  3 x + 3 y ]
```

in which `(i,j)->i*x+j*y` is Maple's operator notation for a function which maps its arguments (i, j) to the function $ix + jy$.

4.2 Special Types of Matrix

Maple recognises the concept of sparse and symmetric matrices, and it allows us to declare such properties as part of the `array` function. For example, we could declare the symmetric 2x2 matrix

$$\begin{pmatrix} p & q \\ q & r \end{pmatrix}$$

using the statement:

```
a:=array(symmetric,1..2,1..2,[(1,1)=p, (1,2)=q, (2,2)=r]);
                          [ p   q]
                    a :=[       ]
                          [ q   r]
```

Notice that the (2,1) element of the matrix was not specified, but Maple is able to deduce it from the symmetry property of the matrix.

If we declare a matrix to be symmetric and do not assign explicit values, Maple still makes use of the symmetry of the matrix when dealing with indexed names. For example:

```
a:=array(symmetric,1..2,1..2):
```

```
a[1,2];
                          a[1,2]

a[2,1];
                          a[1,2]

a[1,2]-a[2,1];
                            0
```

If we now assign values to the elements of the matrix, the symmetry is maintained. Continuing the previous example:

```
a[1,2]:=p;
                     a[1,2] := p

a[2,1];
                           p

a[2,1]:=q;
                     a[2,1] := q

a[1,2];
                           q
```

Sparse matrices can be declared in a similar way to symmetric matrices. For example, to declare a sparse 2×2 matrix we enter:

```
a:=array(sparse,1..2,1..2);

               a := array(sparse, 1..2, 1..2, [])
```

Maple regards any element of a sparse matrix to be zero unless it has been explicitly assigned a value. The following example illustrates this.

```
a:=array(sparse,1..2,1..2,[(1,1)=p]);

                     [ p   0 ]
                 a :=[       ]
                     [ 0   0 ]

a[1,2];
                           0

a[1,2]:=q:

op(a);
                     [ p   q ]
                     [       ]
                     [ 0   0 ]
```

Maple only stores non-zero elements of a sparse matrix. This saves a considerable amount of memory, especially during calculations with large sparse matrices. Some matrix manipulation routines (e.g. the determinant function `det`) have special algorithms for dealing with sparse matrices.

Maple also recognises `antisymmetric`, `diagonal` and `identity` and can manipulate such matrices according to their special properties. When dealing with matrices of a named type, Maple employs a special function called an *indexing function* to reference array components. We can define our own indexing function to deal with matrix types of our own definition. This is explained in the *Maple Language Reference Manual*.

4.3 Matrix Algebra Using "evalm"

Basic matrix algebra can be performed using the library function `evalm`. The `evalm` function recognises the algebraic operators (+,-,*,/,**) together with '&*'. Note that '*' denotes commutative multiplication and should only be used to multiply matrices by scalars. In order to perform non-commutative multiplication between matrices we use '&*'. For example,

```
evalm(a &* b);
```

returns the matrix product $a.b$ whilst

```
evalm(a*b);
```

may be evaluated as

```
evalm(b*a);
```

since the each argument of a procedure is evaluated by Maple before being passed to the procedure, and Maple regards `'*'` as commutative. Hence Maple's expression evaluator may decide to evaluate the argument as $b \cdot a$ before passing it to the procedure, rather than $a \cdot b$.

We can use `evalm` to evaluate any expression involving scalars and matrices. If a and b are 3×3 matrices then

```
evalm(a+b);
```

returns their sum (performed component-wise),

```
evalm(a &* b);
```

or its equivalent

```
evalm(&*(a,b));
```

gives their matrix product in the sense $a.b$,

```
evalm(a**2);
```

gives the matrix which is the square of a, and

```
evalm(a**(-1));
```

or

```
evalm(1/a);
```

yields the matrix inverse of a.

5 The Structure of Maple Expressions

This section describes how Maple represents expressions and shows how expressions can be analysed and manipulated term by term. Unlike some computer algebra systems, Maple provides a range of powerful functions to make manipulation of expressions very easy indeed.

All Maple expressions have a *type*. For objects like lists, sets and integers, the type is obvious. The function `whattype` displays the top-level type of any Maple expression. For example:

```
whattype(3);
                    integer

whattype(3/7);
                    fraction

whattype([r,1,sin(a)]);
                    list

whattype({r,1,sin(a)});
                    set
```

The type of an algebraic expression depends upon its mathematical representation. If it is a sum of terms then its type is '+'. For example:

```
p := expand((1+x)**5);
                         2        3       4    5
    p := 1 + 5 x + 10 x  + 10 x  + 5 x  + x

whattype(p);
                         +
```

However, if it is a product of factors then its type is '*'. For example, `whattype(5*x^4)` yields '*'. If it is an exponentiated expression then its type is '^'. For example, `whattype(x^4)` yields '^'. If it is a simple variable then its type is `string`, so `whattype(x)` yields `string` if `x` has not been assigned a value.

5.1 The Tree Structure of Expressions

Expressions that are not simple variables or numbers can be analysed further using the `op` function which was introduced in Section 1. Consider an expression of type `'+'`, i.e. a sum of terms. Maple represents this as a tree structure with the operator `'+'` as the node and the operands (the terms of the sum) as branches stemming from this node. Thus the expression $a + b + c$ can be represented graphically as follows:

```
          +
        / | \
       /  |  \
      /   |   \
     a    b    c
```

The expression at the end of each branch can itself be a tree structure if it is not a simple variable or number. For example, the expression $a + bd + c$ is represented as follows:

```
          +
        / | \
       /  |  \
      /   |   \
     a    *    c
         / \
        /   \
       /     \
      b       d
```

The `whattype` function displays the type of a tree, and the `op` function lists each of the operands (branches) of the sub-expression. The following example illustrates how `op` may be used in different ways on an expression.

```
p := a+b+c;
                    p := a + b + c

whattype(p);
                        +

op(p);
                    a, b, c

nops(p);
                    3

op(1,p);
                    a

op(2,p);
                    b

op(3,p);
                    c
```

With only one argument, `op` lists the operands of an expression. In this example, the operands are a, b and c whilst the operator is `'+'`. The expression could be regarded as the result of applying the addition operator to a list of arguments (a, b, c).

The `nops` function gives the number of operands in the expression. In this example there are three operands. We can extract them one at a time using the `op` function with two arguments. The first argument is the number of the operand which we require, so if we want the first operand of the expression p then we use `op(1,p)`. Maple signals an error if we try to extract a non-existent operand, so it is always advisable to use `nops` to find the number of operands.

The `whattype` function is useful for finding the type of an expression interactively, but more often we want to know whether an expression is of a given type. The `type` function is provided for this purpose. To find out whether an expression p is of type X, we enter `type(p,X)` which yields the value `true` or `false`. We can test for a wide variety of types in this way. Among the recognised types that can be used as the second argument of `type` are `integer, fraction, float, numeric, string, ‘+‘, ‘*‘` and `‘^‘`. In all 17 basic data types are implemented in the Maple kernel. Note that algebraic operators must be enclosed in back-quotes otherwise Maple signals an error.

The following example illustrates the use of `type` together with `op` and `nops`.

```
p := a+b*d+c;
                    p := a + b d + c

type(p,‘+‘);
                    true

type(p,integer);
                    false

nops(p);
                    3

op(p);
                    a , b d , c

q:=op(2,p);
                    q := b d

type(q,‘+‘);
                    false

type(q,‘*‘);
                    true

nops(q);
                    2

op(q);
                    b , d

op(1,q);
                    b
```

Notice how we have used `type`, `op` and `nops` repeatedly to analyse p completely. First, we see that p is a sum of three terms (a, bd and c) and then we see that the second term of p is itself a product of two terms (b and d).

Some types recognised by the `type` function are supersets of other types and are called composites types.

For example, the type `numeric` includes the more specific types `integer`, `rational` and `float`. Hence `type(3,numeric)` yields `true` as does `type(3,integer)`, though of course `type(3,float)` is `false`.

An expression such as $\sin(x+3)$ or $f(a,b,c)$ is of type `function` and the use of `op` with a single argument yields a list of the function's arguments but *not* the function name. The function name is available as the special operand numbered zero. For example:

```
p := f(a,b,c);
                    p := f(a,b,c)

whattype(p);
                    function

nops(p);
                    3

op(p);
                    a , b , c

op(1,p);
                    a

op(0,p);
                    f
```

5.2 Type definition

In the previous section, we encountered some of the types known to the `type` function. These types belongs to the *system types* category, which is defined in the Maple kernel. This category contains Maple's internal data structures. For objects defined by the user or in the Maple library, the `type` function extends its knowledge of new types by making use of the *procedural type*. As the name suggests, this type testing is done by using a Maple library procedure '`type/xxx`' where `xxx` is the name of the type. For example in the case of positive integers, the library function `type/posint` has the following definition:

```
`type/posint`:=proc(n)
options `Copyright 1990 by the University of Waterloo`;
    type(n,integer) and (0 < n)
end
```

Maple uses the *assigned type* for argument checking in procedure definitions. This method is preferable to using nested if statements to perform argument checking as it is less costly in computer times. In the case of the `int` function, Maple implements the type checking in the following way:

```
`type/intargs`:={[{algebraic, procedure}, name],

           [{algebraic, procedure}, name = algebraic .. algebraic],

           [{algebraic, procedure}, name = algebraic .. algebraic, name]}

    ....

    if not type([args],`intargs`) or (nargs = 3) and (args[3] <> 'continuous')
```

```
 then
ERROR(`invalid arguments`)
```

The assigned type `type/intargs` defines three "templates" for arguments of the `int` function. They correspond to the following three examples:

```
int(sin(x), x);

int(sin(x), x=0..Pi);

int(q(x), x=a..b, continuous);
```

All the types we have encountered so far belong to the category *simple types* in the sense that they are identified by a single name. Types that are identified by more than a single type are called *structured types*. They provide Maple with a more thorough type testing facility. A structured type is a Maple expression, other than a string, which has been built using any of the Maple operators, constants, and the names of simple types. A typical example is the expression `array,table` which is used by the `copy` function. This specifies the alternation of the types table and array. The following examples illustrate the range of structured types available to Maple:

```
type(log(x),log(name));

                                   true

type(x*y,`*`(algebraic));

                                   true

type(2*x*y,`*`(algebraic));

                                   true

type(2*x*y,`*`(name));

                                  false

rel:=a=b,c=d,e=f:

type({rel},set(relation));

                                   true

type({rel,2=x+3},set(name=name));

                                  false

mat:=array([[s,0,0],[0,s,0],[0,0,s]]):

type(mat,'matrix(anything,square)');

                                   true
```

5.3 Mapping Functions onto Expressions

The `map` function allows us to map a Maple function onto successive operands of an expression. For example, we may wish to map a function f onto an expression such as $a+b+c$ to obtain: $f(a)+f(b)+f(c)$ The function f may be a mathematical operator (e.g. `diff`) or function (e.g. `sin`) or it may be a procedure which performs some simplification or transformation upon its arguments (e.g. `simplify`).

We may wish to map f onto a product of terms such as abc to obtain $f(a)f(b)f(c)$ or onto

```
[a , b , c]
```

to obtain:

```
[f(a) , f(b) , f(c)]
```

This can be achieved using the `map` function, as the following examples illustrate.

```
p:=a+b+c;
                    p := a + b + c

map(f,p);
                  f(a) + f(b) + f(c)

map(sin,p);
               sin(a) + sin(b) + sin(c)

p := a*b*arcsin(c);
               p := a b arcsin(c)

map(sin,p);
                 sin(a) sin(b) c

map(exp,p);
           exp(a) exp(b) exp(arcsin(c))
```

If the function to be mapped requires more than one argument, then these can be supplied as extra arguments to `map` itself. For example:

```
p:=a+b+c;
               p := a + b + c

map(f,p,x,y,z);
               f(a,x,y,z) + f(b,x,y,z) + f(c,x,y,z)
```

In this example, the extra arguments x, y and z are passed as arguments to each occurrence of f.

The `map` function can be applied to any Maple expression. It operates by mapping the function onto each operand of the expression. For example, if p is a Maple expression of type `'+'` (i.e. a sum of terms) and $N=$ `nops`(p) then

```
map(f,p,a,b);
```

returns the sum

```
f(op(1,p),a,b) + f(op(2,p),a,b) + ... + f(op(N,p),a,b)
```

whilst if p is of type `'*'` then the result is a product:

```
f(op(1,p),a,b) * f(op(2,p),a,b) * ... * f(op(N,p),a,b)
```

We are not restricted to applying `map` to sums and products. We could apply it to lists or sets. For example:

```
 map(f,[a,b,c]);
                   [f(a) , f(b) , f(c)]
```

We could also apply it to an expression of type `'^'` if we wished. For example:

```
p:=a**x;
                                x
                         p := a

op(1,p);
                           a

op(2,p);
                           x

q:=map(f,p);
                                   f(x)
                         q := f(a)

op(1,q);
                           f(a)

op(2,q);
                           f(x)
```

Notice that `map` only maps the function onto the top-level operands of an expression. Hence

```
map(f, a+b*c+d);
```

yields

```
                  f(a) + f(b*c) + f(d)
```

and not:

```
                 f(a) + f(b)*f(c) + f(d)
```

5.4 Searching for Sub-Expressions

Often, we may wish to know whether an expression contains a particular variable or sub-expression. For example, we may want to test an expression to see if it depends upon a variable. The function `has` is used to test for the presence of a variable or sub-expression. It searches each sub-expression of its first argument recursively for any occurrence of its second argument. Thus `has(f,x)` returns the value `true` if and only if a sub-expression of f is equal to x. For example,

```
has((a+b)**2,a+b)
```

is `true` but

```
has(expand((a+b)**2),a+b)
```

is `false` because the two expressions, though mathematically identical, have different tree-structure representations in Maple. The following examples illustrate the use of `has`.

```
p:=(a+b)**4;
                                           4
                         p := (a + b)

has(p,a);
                            true

has(p,c);
                            false

has(p,a+b);
                            true

has(p,4);
                            true

p:=expand(p);
                  4      3         2  2         3    4
          p := a  + 4 a  b + 6 a  b  + 4 a b  + b

has(p,a);
                            true

has(p,a+b);
                            false

q:=1+y+sin(x)**2;
                                          2
           q := 1 + y + sin(x)

has(q,x);
                            true

has(q,sin(x));
                            true
```

```
has(q,sin);
                              true
```

5.5 Substituting for operands of an expression

Maple provides a function called `subsop` which allows the user to replace any operand of an expression. Its use is similar to that of the `subs` function described in Chapter 1, but instead of specifying the variable or sub-expression that is to be replaced by *name*, we can specify it by its *operand number*/ For example,

```
p := expand((1+x)^3);

                                        2    3
                     p := 1 + 3 x + 3 x  + x

op(2,p);

                               3 x

subsop(2=a,p);

                                   2    3
                          1 + a + 3 x  + x

op(2,");

                                a
```

We can substitute for more than one operand at a time. Continuing the example,

```
subsop(2=a,3=b,4=c,p);

                          1 + a + b + c
```

Recalling that the function name in an expression such as `sin(x)` is operand zero, we can replace the function name by applying `subsop` to operand zero. For example,

```
p := sin(x);

                           p := sin(x)

subsop(0=cos,p);

                              cos(x)
```

5.6 Analysis of Polynomials

Maple provides several functions for the analysis and manipulation of polynomials and similar algebraic expressions. We can determine the highest and lowest powers of a variable within a polynomial and we can extract the coefficient of a specified polynomial term. The following example illustrates the functions `degree`, `ldegree` and `coeff`.

```
p:=x+3*x**2+b*x**3+x**7;
                                  2       3    7
               p := x + 3 x  + b x  + x

degree(p,x);
                    7

ldegree(p,x);
                    1

coeff(p,x,2);
                    3

coeff(p,x,3);
                    b

coeff(p,x,4);
                    0

degree(p,b);
                    1

coeff(p,b,1);
                     3
                    x
```

The function `degree` returns the highest power of a given variable within a polynomial, and `ldegree` returns the lowest power. The function `coeff` returns the coefficient of the specified power of the given variable within a polynomial.

In order for these functions to work correctly, the expression must be a polynomial in the given variable. This can be achieved by first applying the `expand` function to the expression to ensure that factors are fully multiplied out.

An expression can be re-ordered so that it is explicitly a polynomial in a given variable. The function `collect` re-arranges a polynomial by collecting all the coefficients with the same power of a variable. For example:

```
y:=expand((a+b+x)**2);

                   2                    2             2
           y := a  + 2 a b + 2 a x + b  + 2 b x + x

collect(y,x);
```

```
                      2                      2    2
                     x  + (2 b + 2 a) x + b  + a  + 2 a b
```

A polynomial can be converted to Horner (nested) form using one form of the `convert` function. For example:

```
y := expand((1+x)**6);
                           2       3       4      5    6
     y := 1 + 6 x + 15 x  + 20 x  + 15 x  + 6 x  + x

convert(y,horner,x);

     1 + (6 + (15 + (20 + (15 + (6 + x) x) x) x) x) x
```

6 Defining Maple Procedures

Many of Maple's mathematical facilities are written as procedures in the Maple language itself, and we can define our own procedures in a similar way. Procedures can be used to set up new simplification rules and to create new mathematical functions and define their properties. They can also be used to make it easier to repeat sequences of Maple commands.

Maple procedures are similar to procedures (or subroutines) in other programming languages:

- They can contain any valid sequence of Maple commands.
- They can use local variables which are only defined within the procedure, but they can also access and alter global variables.
- They can be passed arguments which may be any valid Maple data type.
- They can return a result which may be any valid Maple data type.
- They can be called with an over-specified or under-specified argument list.
- They can be called recursively; a procedure may call itself.

6.1 Elementary Definitions

A Maple procedure is defined using the `proc` keyword. The following example defines a procedure called f which squares its argument.

```
f := proc (x) x**2 end;
```

The keyword `proc` introduces a procedure definition. Then follows the formal argument list enclosed in parentheses. In this example, there is only one parameter, x. The next item is a Maple statement which defines the value of the procedure in terms of its formal arguments. The definition is terminated by the keyword `end`. After this definition, we could use f as follows:

```
f(3);
                                    9

f(a+b);
                                       2
                                  (a+b)

f(f(a+b));
                                       4
                                  (a+b)
```

We could define a procedure to return the maximum of two values as follows.

```
max := proc (a,b) if a>b then a else b fi end;
```

Note how the `end` keyword acts as a statement terminator (in place of the semicolon) as well as marking the end of the procedure definition. In fact, between `proc (a,b)` and `end` we could write a sequence of Maple statements, and the value of the procedure is simply the value of the last statement. For example, we could re-define `max` to print out its two arguments before calculating the maximum:

```
max := proc (a,b)
         print(a,b);
         if a>b then a else b fi
         end;
```

Note that there is *no semicolon* after the formal argument list nor after the last statement of the definition. Calling this procedure might produce the following result:

```
p:=max(1,2);
                       1 , 2

                       p := 2
```

6.2 Local Variables

It is usually necessary to save intermediate results within a procedure definition unless it is a very short procedure. The special 'previous result' variables `"`, `""` and `"""` can be used in a procedure definition but they refer only to results evaluated within that procedure and not to any value outside the procedure.

We can declare named variables which we wish to use within a procedure definition. The `local` keyword is used to do this. For example, this procedure calculates the maximum of three numbers:

```
max := proc (a,b,c)
         local x;
         if a>b then a else b fi;
         x:=";
         if c>x then c else x fi
         end;
```

The variable x is declared as a local variable. It can be used in the same way as any other Maple variable but it is defined only within the procedure definition. In particular, it bears no relation to any variable x which may have been used outside the procedure definition.

To declare two or more local variables, a statement such as

```
local x,y,speed,height,mass;
```

could be used. The `local` statement must follow immediately after the formal argument list. Note that it is ended by a semicolon but there is *no semicolon* between the formal argument list and the keyword `local`.

6.3 Local Variables Versus Global Variables

If we use a variable name within a procedure that is neither a formal argument nor a local variable, then it is regarded as a *global variable* which exists outside the procedure. For example:

```
a:=5;
                             a := 5

f := proc (x) x+a end;

f(3);
                               8

f(x);
                             x + 5

f(a);
                              10
```

In this case, a was not declared to be a local variable in the procedure f so its value was taken from outside the procedure definition. We could assign another value to a and the procedure definition would then use the new value. Continuing the previous example:

```
f(m);
                             m + 5

a:=r;
                             a := r

f(m);
                             m + r

f(a);
                              2 r
```

It is advisable to use global variables with caution, since it is possible to alter them within a procedure definition. For example, consider this procedure which sums the squares of a sequence of numbers.

```
f := proc (a,b)
       local s;
       s := 0;
       for j from a to b do
         s := s + j**2
       od;
       s
       end;
```

The loop variable j has not been declared to be a local variable within the procedure, so the value of the global variable j will be used instead and will be altered each time that the procedure is called. Continuing the example:

```
s := t**2;
                                     2
                              s := t

j := sin(x);
                              j := sin(x)

f(1,10);
                                385

j;
                                 11

s;
                                 2
                                t
```

The global variable j has been left with its value at the end of the loop within the procedure definition, whilst the global variable s is unaltered since s was declared to be a local variable within the procedure. If we wanted to avoid altering j outside the procedure, we should have declared `local s,j;` within the procedure.

6.4 Arguments of Procedures

When a Maple procedure is invoked with a list of arguments, each argument is evaluated and the result is used in place of the corresponding formal argument specified in the procedure definition. The arguments are only evaluated *once*, at the point when the procedure is invoked.

It is possible to call a procedure and specify a number of actual arguments which is different from the number of formal arguments in the procedure definition. If we supply too many arguments, the extra arguments are simply ignored. If we supply too few, then Maple signals an error if (but only if) the corresponding formal argument is referenced whilst executing the procedure, as the following example illustrates.

```
f := proc (n,a,b) if n>0 then a**2+b**2 else a**2 fi end;

f(1,x,y);
                                     2    2
                                    x  + y

f(0,x,y);
                                       2
                                      x

f(0,x);
                                       2
                                      x

f(1,x);
    Error, (in f) too few parameters in function call
```

Maple also provides another method for referencing arguments of procedures. Regardless of the formal argument list, the actual parameters can be obtained as the elements of the expression sequence with the special name args. For example, we can refer to the first actual argument as args[1] and to the second actual argument as args[2]. The number of arguments supplied when the procedure was called is available within any procedure definition as the special variable nargs. This number may be zero if the procedure was called with no arguments. The following example procedure illustrates the use of args and nargs. It calculates the maximum of an arbitrary number of items.

```
max := proc ()
         local biggest,j;
         biggest := args[1];
         for j from 1 to nargs do
           if args[j]>biggest then biggest:=args[j] fi
         od;
         biggest
         end;
```

Notice that this definition has no formal arguments since we are referencing the actual arguments via args. The parentheses following the proc keyword must still be included, even though they are empty.

6.5 Testing the Type of an Argument

In all of the previous examples, we have assumed that the actual arguments passed to the procedures have been valid. Consider the following definition of the multinomial function which calculates

$$\frac{(a+b+c+\cdots+n)!}{a!\,b!\,c!\,\cdots\,n!}$$

where $a, b, c, \cdots, n$ are all non-negative integers.

```
multinomial := proc ()
                 local j,m,n;
                 m := sum('args[j]', 'j'=1..nargs);
                 n := product('args[j]!', 'j'=1..nargs);
                 m!/n
                 end;
```

This definition would be incorrect if the procedure were called in any of the following ways:

```
multinomial(1,3,-5);
multinomial(5,sin(y));
multinomial();
```

It is therefore advisable to include several checks within the definition to ensure that at least one argument has been given and that all of the arguments are non-negative integers. After the local declaration, we could include the lines:

```
if nargs=0 then ERROR(`no arguments given`) fi;
for j from 1 to nargs do
  if not (type(args[j],integer) and args[j]>=0) then
    ERROR('invalid argument `,args[j]) fi
od;
```

This piece of code uses the ERROR function, which causes Maple to stop executing this procedure (and all others in the calling chain) and return to the top level of Maple. An error message will be displayed to show where the error occurred. For example:

```
multinomial(1,3,7);
                          1320

multinomial(x,3,7);
  Error, (in multinomial) invalid argument, x

multinomial();
  Error, (in multinomial) no arguments given
```

The ERROR function can be used in any Maple procedure when we need to leave the procedure because of a serious (irrecoverable) error.

6.6 Returning a Value from a Procedure

The RETURN keyword allows us to leave a procedure and return a value before we reach the end of the procedure. This is often used in conjunction with an if statement to leave a procedure as soon as some condition has been fulfilled. The following example of a binomial-expansion procedure illustrates the use of RETURN. It calculates the expansion of

$$(1+x)^n$$

up to the term in x^m.

```
binomx := proc(x,n,m)
  local i,thisterm,res,factr;
  if not type(m,integer) then
    ERROR(`Third arg must be integer`) fi;
  if m<0 then ERROR(`Third arg must not be negative`) fi;
  if m=0 then RETURN(1) fi;
  res:=1;thisterm:=1;factr:=n+1;
  for i from 1 to m do
    factr:=factr-1;
    thisterm := thisterm*factr*x/i;
    res := res + thisterm
  od;
  res
  end;
```

Note the use of RETURN to leave if $m = 0$, in which case only the leading term (1) is required. RETURN is also used at the end of the definition, though it need not be explicitly called here.

Any Maple datatype can be returned from a procedure, either via the last statement of a procedure definition or via a RETURN statement. In the case of *tables* and *arrays* however, the value must be returned using the op function. This is illustrated by the following example which returns an array made up of the coefficients of successive powers of a variable in a polynomial.

```
xpoly := proc (p,x)
              local norder, A, j;
              norder := degree(p,x);
              A := array(0..norder);
              for j from 0 to norder do
                A[j] := coeff(p,x,j)
              od;
              eval(A)
              end;
```

Notice that we return `eval(A)` rather than just `A`, because tt A has been assigned a value which is an array. If we had said `A` then we would get the literal value `A` each time we called the procedure instead of the array of coefficients. With the definition above, we could use the procedure as follows:

```
p := expand((1+w)**3);
                                     2    3
                p := 1 + 3 w + 3 w  + w

xpoly(p,w);
                            array(0 .. 3,, [
                                0 = 1
                                1 = 3
                                2 = 3
                                3 = 1
                            ])
```

When we wish to return a table or an array from a procedure, we must use `op` in conjunction with `RETURN`.

6.7 "Fail Return" from Procedures

Sometimes, we wish to signal that a procedure has failed to compute a result. For example, the integration function `int` does this when it is unable to evaluate an integral analytically. It simply returns the unevaluated function call. We can do this within a procedure by using the special variable `args` (the actual argument list of the current procedure) and `procname` (the name of the current procedure) and by including the line

```
RETURN('procname(args)');
```

in the procedure. If this line is executed within a procedure, then the unevaluated procedure call is returned as the result. This is usually used to indicate that the function cannot be evaluated with the given arguments, though the arguments are not actually invalid so a return via `ERROR` is inappropriate.

6.8 Tailoring Maple's Built-in Functions

Some of Maple's functions can be tailored so that they recognise functions and datatypes defined by the user. For example, if we wish to apply a differentiation rule to Bessel functions then we can define a procedure called 'diff/bessel' which Maple will apply automatically to evaluate the derivative of any function named bessel.

The Maple functions which can be tailored in this way are:

```
convert
diff
expand
simplify
taylor
type
```

As an example, consider the hypergeometric function $F(a, b; c; z)$. Its derivative with respect to its fourth argument is defined by

$$\frac{d}{dz}F(a,b;c;z) = \frac{ab}{c}F(a+1,b+1;c+1;z)$$

so we can tailor the diff function to apply this rule by defining a procedure called diff/F. The function call

```
diff(F(a,b,c,f),z);
```

results in a call to the function diff/F as follows:

```
'diff/F'(a,b,c,f,z)
```

which would return the value:

$$\frac{ab}{c}\frac{df}{dz}F(a+1,b+1;c+1;f)$$

The definition of diff/F is as follows:

```
`diff/F` := proc(a,b,c,f,z)
              local df;
              if nargs<>5 then
                ERROR(`F has wrong number of arguments`) fi;
              if (has(a,z) or has(b,z) or has(c,z)) then
                ERROR(`Independent variable `,z,
                ` appears outside last parameter`) fi;

              df := diff(f,z);
              (a*b*df/c)*F(a+1,b+1,c+1,f)
              end;
```

Note that Maple converts second and higher derivatives into recursive calls to diff so that a call such as

```
diff(F(1,2,3,x),x,x);
```

is evaluated as if it were

```
diff(diff(F(1,2,3,x),x),x);
```

and so there is never any ambiguity in the argument list passed to `diff/F` about which arguments are those of the function F and which are the differentiation variable(s). The last argument is always the independent variable and all previous arguments belong the the function F.

With the definition of `diff/F` above we can calculate derivatives of hypergeometric functions. For example:

```
diff(F(1,2,3,x),x);
                 2/3 F(2, 3, 4, x)
diff(F(1,2,3,sin(x)),x);
                 2/3 cos(x) F(2, 3, 4, sin(x))
diff(F(1,2,3,x),x,x);
                F(3, 4, 5, x)

diff(F(x,2,3,x),x);
Error, (in diff/F), independent variable, x, appears outside last
parameter
```

The last example illustrates how to trap invalid cases. The differentiation rule used by `diff/F` is not applicable if the differentiation variable appears in one of the first three arguments.

6.9 Example - A Conversion Function

This short Maple procedure converts a Poisson series term into a list consisting of the coefficient, the function (sin or cos) and the argument. It can be called via the `convert` function with a command such as

```
convert(x**2*cos(3*p-q),poisson);

                          2
                       [ x  , cos , 3 p - q ]
```

and it returns zero if the expression is not in the correct form. Notice the use of anonymous procedures in the calls to the `map` function. This avoids the need to define *named* procedures to perform relatively trivial tasks.

The function will operate upon a sum of Poisson terms to produce a sum of lists. Maple allows arithmetic operators to act upon lists, but it has no simplification rules for the sum or product of two lists. This fact could be used to good effect.

```
# convert/poisson :- changes a term or sum of terms from the form
#
#  a*sin(x) or a*cos(x)
#
#  into
#
#  [a,sin,x] or [a,cos,x]
#
readlib(zip):
`convert/poisson` := proc(x)
  local xlist,blist,flist,tlist,fn;
```

```
  if   not has(x,{sin,cos}) then RETURN(0)
  elif type(x,`+`) then RETURN(map(procname,x))
  elif type(x,function) then
    if   op(0,x)=sin then RETURN([1,sin,op(1,x)])
    elif op(0,x)=cos then RETURN([1,cos,op(1,x)])
    else RETURN(0)
    fi
  elif type(x,`*`) then
    xlist := [op(x)];
    blist := map(proc(x) type(x,function) and (op(0,x)=sin or op(0,x)=cos)
                  end, xlist);
    flist := zip(proc(x,y) if not x then y else NULL fi end, blist, xlist);
    tlist := zip(proc(x,y) if x then y else NULL fi end, blist, xlist);
    if has(flist,{sin,cos}) or nops(tlist)>1 then
      ERROR(`incompletely simplified argument `,x) fi;
    fn := op(1,tlist);
    RETURN([convert(flist,`*`),op(0,fn),op(1,fn)])
  else RETURN(0)
  fi
 end;
```

6.10 Truncation of multinomial series

The routines defined in this section can be used to truncate series which are expressed in powers of small quantities. Maple's `series` data type provides a limited facility for this kind of operation, but `series` objects are not amenable to general algebraic manipulation.

```
truncate := proc(poly,olist,omax)
  if not type(omax,numeric) then ERROR(`third argument not numeric`) fi:

  if type(poly,`+`) then RETURN(map(procname,poly,olist,omax)) fi:

  if termorder(poly,olist)>omax then 0 else poly fi

  end:


termorder := proc(poly,olist)
  local x,ix;
  if not type(olist,list) or type(olist,set) then
    ERROR(`second argument is not a list or set`) fi:

  for x in olist do
    if not (type(x,`=`) and type(op(1,x),string) and type(op(2,x),numeric))
      then ERROR(`invalid order specifier`,x)
    fi
  od:

  ix := 0:

  for x in olist do
    ix := ix + op(2,x)*degree(poly,op(1,x))
```

```
  od:

  RETURN(ix)
  end:
```

There are two procedures, `truncate` and `termorder`. Their use is illustrated in the following example:

```
p := expand((1+x+y)^10):

truncate(p,[x=1,y=2],4);

        2                2        2                         3        4
1 + 45 x  + 90 x y + 45 y  + 360 x  y + 10 x + 10 y + 120 x  + 210 x
```

Here, x is treated as a first-order small quantity and y is treated as second-order, and the expansion of $(1+x+y)^{10}$ is truncated to fourth order. This eliminates terms higher than x^4 and y^2 and also cross-products such as x^3y which is considered to be of fifth order.

The procedure `truncate` uses `termorder` to determine the order of each term, so most of the work of truncation is performed by the latter. Each variable (x and y in our example) has an order, as defined in the list which is the second argument of `truncate`. This list is passed directly to `termorder` together with the single term whose order is sought.

For each variable in the order-list, `termorder` uses the `degree` function to extract the degree of the variable in the term being examined. It multiplies the degree by the order of the variable and adds the results for all the named variables to yield the total order of the term.

6.11 A generalised degree function

Maple's built-in `degree` function can be used to determine the degree of an expression in a specified variable, as illustrated in the previous section. However, it fails if the exponent of the variable is not numeric. For example,

```
degree(3*x^4, x);
                                4
```

but

```
p := x^J;
                                 J
                                x

degree(p,x);
                               FAIL
```

The procedure listed below can determine the degree of an expression in which exponents may be numeric or symbolic.

```
pdegree := proc(p,x)
  local pwrset,totpwr,j;
```

```
if not type(x,name) then
  ERROR(`second argument must be a name`)
fi:

# Check to see if the function contains the variable.
if not has(p,x) then
  RETURN(0)
fi:

# If it's a sum, return the lowest power which occurs.
if type(p,`+`) then
  pwrset := map(pdegree,convert(p,set),x);
  RETURN(min(op(pwrset)))
fi:

# If it's a name, check to see if it's equal to the variable.
if type(p,name) then
  if p=x then
    RETURN(1)
  else
    RETURN(0)
  fi
fi:

# If it's a product, the power is the sum of the powers in each term.
if type(p,`*`) then
  totpwr := 0;
  for j from 1 to nops(p) do
    totpwr := totpwr + pdegree(op(j,p),x)
  od:
  RETURN(totpwr)
fi:

# If it's a power, then the result is the exponent times the power in
# the exponand, if you see what I mean. And if 'exponand' is a real word !
if type(p,`^`) then
  RETURN(op(2,p)*pdegree(op(1,p),x))
fi:

# What else could it be ? If it's a function, then we're sunk. We can't
# compute the degree of 'x' in cos(x), for example.
FAIL

end:
```

This procedure can be used in cases where `degree` would fail. For example,

```
 p:=x^J;
```

```
                              J
                        p := x
```

```
 pdegree(p,x);
```

```
                                   J

pdegree(p*x,x);

                                 J + 1

pdegree(p^2,x);

                                  2 J

(1+x^J)^K;

                                      J K
                               (1 + x )

pdegree(",x);

                              K min(0, J)
```

The procedure is defined recursively to enable it to deal with more complicated expressions such as the last example above.

7 File Handling

This section describes Maple's facilities for writing results to files and reading commands from files. It also describes how to save all or part of the current Maple session so it can be resumed again later.

7.1 Maple File Names

Within Maple, file names are represented using a convention intended to make programs portable between different operating systems. The convention is derived from UNIX-like file names which may include directory names. An example filename is

```
maple/prog1.mpl
```

which is the file `prog1.mpl` in the user's directory `maple`. On a VAX/VMS system, this would be the file `[.maple]prog1.mpl`. Similarly, a Maple filename such as

```
maple/dynamics/kepler.mpl
```

is equivalent to the VAX/VMS file `[.maple.dynamics]kepler.mpl`.

On all machines, the directory may be omitted if the file is on the user's default directory. Thus the Maple filename

```
prog2.mpl
```

corresponds to `prog2.mpl` on a VAX/VMS machine.

Filenames should be enclosed in back-quotes to prevent Maple from trying to evaluate them as algebraic expressions.

Filenames which end with `.m` (e.g. `myfile.m` or `maple/ellipse.m`) are treated in a special way by Maple, and such names should not be used for files containing text input or output. The use of `.m` files is explained in Section 7.5.

7.2 Reading Commands from Files

The `read` function can be used to read a sequence of Maple commands or expressions from a file instead of the terminal. This is very useful for setting up frequently-used procedures or expressions during Maple sessions and for running Maple programs in a batch-processing system.

Any valid Maple commands or expressions can be read from a file, including the `quit` command. When the end of the file is reached, Maple reverts to reading its input from the previous input source. This may be the terminal or it may be another file, since files may contain `read` commands themselves.

For example, suppose the file `ellipse.mpl` contains a sequence of Maple commands. The command

```
read `ellipse.mpl`;
```

will read the commands from the file and execute them. Maple then continues to take commands from the previous source.

7.3 Writing Output to Files

The `writeto` function is used to re-direct Maple's output to a file. Maple can only write to one device (the terminal or a file) at a time, and so during an interactive session when output is being written to a file, nothing appears on the terminal screen.

For example, to write output to the file `example.out` we would use the command

```
writeto(`example.out`);
```

and all output, including Maple's `>` prompts, is sent to that file *instead* of to the terminal screen. We could terminate output to the current file and re-direct output to the terminal by entering the command:

```
writeto(terminal);
```

This command has the effect of closing the file `example.out`, and if we subsequently entered the command

```
writeto(`example.out`);
```

again, the previous copy of the file would be deleted (or a new generation created under VAX/VMS) because `writeto` creates a new file and does not append output to an existing one. In order not to overwrite an existing file, the command `appendto` is used instead. In this last example, the command

```
appendto(`example.out`);
```

should have been used.

7.4 Saving Expressions and Procedure Definitions

The `save` command allows us to save any Maple expression or procedure definition to a file during a Maple session, so that it can be read from that file using the `read` command in a later session. If we save to a file whose name does not end `.m` then the expressions or procedures will be written to the file in the same syntax as an ordinary Maple input file. The output can be read and altered using a text file editor if required. Some lines may end with a back-slash (\) character to indicate continuation of a long name or number across the end of a line.

We can save a list of named variables and procedures as illustrated in the following example:

```
p:=1+sin(x);
                    p := 1 + sin(x)

q:=int(1/",x);
                                  1
               q := -2 ----------------
                           tan(1/2 x) + 1

max:=proc(x,y) if x>y then x else y fi end;
      max := proc (x,y) if y<x then x else y fi end

save p,q,max,`myfile.sav`;
```

The file `myfile.sav` now contains the lines

```
p:=1+sin(x);
q:=-2*1/(tan(1/2*x)+1);
max := proc (x,y) if y<x then x else y fi end;
```

and it could be read in a later Maple session with the command:

```
read `myfile.sav`;
```

7.5 Saving and Reading Maple Internal-Format Files

Maple treats files whose names end in `.m` (e.g. `myfile.m`) as *internal-format* files. They contain Maple expressions and procedure definitions in a special binary format which does not resemble text but which can be read into Maple more quickly and efficiently than text input files. Most of Maple's own library of functions is stored in files in this way, and each routine is only read from the file when it is first used.

If the `save` command is used with a file name ending in `.m` then Maple generates an internal-format file instead of a text file. The `.m` file can be read using the `read` command, but Maple does not echo the expressions or procedure definitions on the terminal screen as it does when reading a text file.

We can save the entire Maple session, including assigned variables and procedure definitions, by entering the `save` command with only a filename. Maple assumes that if no list of expressions or procedures has been given, then it must save everything. We could save a Maple session with a command such as

```
save `session.m`;
```

and resume it again by entering

```
read `session.m`;
```

as the first command of a new Maple session.

To find out which variable names have been assigned values during the current Maple session, we can use the `anames` function. This is sometimes useful for checking which variables have been assigned values after an `.m` file has been read.

8 Maple graphics

Maple provides facilities for producing graphical representations of functions in both two and three dimensions. The reader should note that this normally requires the use of a display capable of rendering high-resolution images.

Personal computers such as Apple Macintosh and 80386-based IBM/PC-compatibles are generally supplied with such displays. In the case of PC-compatibles, a VGA card is preferable.

Users of UNIX machines normally need to run a version of Maple which makes use of the X Window System to produce graphical output. This normally entails issuing a command such as `xmaple` to start a Maple session, rather than just `maple` which uses a plain "teletype" interface.

You should refer to the installation documents or your local computer centre for advice on machine-specific questions such as this. In the remainder of this section, we will assume that you have discovered how to get graphical output in Maple on your own computer system.

8.1 Plotting graphs in two dimensions

Maple provides a function called `plot` which draws the graph of one or more functions in a single variable. For example, to plot a graph of $\sin x$ in the range $0 \le x \le 2\pi$, you would type

```
plot(sin(x), x=0..2*Pi);
```

If a set of functions is provided in place of a single function, then Maple draws all of the functions on the same graph, overlaying them. This example shows how to plot the Bessel functions $J_0(x), J_1(x), J_2(x)$ on the same graph. We use the sequence operator to construct the set of functions.

```
fns := {'BesselJ(n,x)' $ 'n'=0..2}:

plot(fns, x=0..5);
```

The same function can also be used to produce parametric plots. For example, to draw the Archimedean spiral given by

$$x = t\cos t,$$
$$y = t\sin t,$$

in the parameter range $0 \le t \le 10$, you would use the command

```
plot([t*cos(t), t*sin(t), t=0..10]);
```

in which the curve is specified as a list of three values: the x-coordinate and the y-coordinate written in terms of the parameter, and then the parameter and its range.

When drawing graphs of functions, Maple uses a technique called adaptive plotting in which it decides automatically how many points to use depending on how rapidly the function varies in each part of the range. When the function is varying slowly, it plots fewer points, but when the function changes rapidly over a small interval, then Maple will use correspondingly more points within that interval to ensure that the behaviour of the function is accurately represented.

Using the same command, Maple can plot lists of points given as numerical data values. By default, Maple will join the points with straight lines in the order in which you have listed them. For example,

```
square := [ 0,0, 0,1, 1,1, 1,0, 0,0 ]:

plot(square);
```

will draw a square whose corners are the points $(0,0$, $(0,1)$, $(1,1)$ and $(1,0)$. We could have specified the data points as a *list of lists* in which each sub-list is a pair of (x,y) coordinates, like this:

```
square := [ [0,0], [0,1], [1,1], [1,0], [0,0] ]:

plot(square);
```

which is exactly equivalent to the previous example.

If we do not want Maple to join the points with straight lines, we can specify the option `style=POINT`. For example,

```
square := [ [0,0], [0,1], [1,1], [1,0], [0,0] ]:

plot(square, style=POINT);
```

places cross marks at the corners of the square but does not join them with lines.

8.2 Polar plots in two dimensions

As part of the `plots` package, Maple provides the `polarplot` function which allows you to plot a curve from its description in polar i.e. (r,θ), representation. The simplest form of this function is illustrated by a command to draw the polar curve $r(\theta) = 3\cos(2\theta)$ for the range $-\pi \leq \theta \leq +\pi$:

```
polarplot(3*cos(2*theta), theta=-Pi..Pi);
```

If the range of the parameter θ is to be a complete cycle over 2π, then the range can be omitted. The previous example could have been written simply as

```
polarplot(3*cos(2*theta));
```

`polarplot` can also generate parametric polar plots in which both r and θ are specified as functions of a third parameter. The usage is analogous to cartesian parametric plots; the curve is defined by a list of three values, the r and θ coordinates expressed in terms of the parameter, then the parameter and its range. For example:

```
polarplot([exp(t/10), sin(t), t=0..Pi]);
```

As with the cartesian plot function, `polarplot` allows you to specify a set of several polar curves to be drawn on the same graph. For example,

```
polarplot({cos(theta),cos(2*theta)});
```

8.3 Surfaces in three dimensions

By analogy with the plot function, Maple provides plot3d which can be used to draw surfaces in three dimensions. The simplest form requires a function of two variables which represents the height above the (x, y) plane, and the ranges of the corresponding x and y variables. For example,

```
plot3d(BesselJ(0,x)*BesselJ(1,y), x=-5..5, y=-5..5);
```

plots the surface

$$z = J_0(x)J_1(y),$$

in the range $-5 \leq x \leq +5, -5 \leq y \leq +5$.

Maple can draw three-dimensional figures in many different ways according to the options supplied as extra arguments of the plot3d function (or selected from a menu on some types of machine). These control colouring, viewpoint, perspective, inclusion (or not) of axes or a bounding box, and the style in which the surface is to be drawn. The style can be chosen from the following list:

- POINT – the surface is drawn as a set of discrete points whose x and y coordinates lie on the grid (see below).

- WIREFRAME – the surface is drawn as a "wire frame" using straight lines to join neighbouring grid points.

- HIDDEN – similar to WIREFRAME except that hidden-line removal is applied to give a more realistic (and less confusing) view of the surface.

- PATCH – the surface is drawn as a set of opaque coloured patches. This yields the most realistic and satisfying view of the surface.

Surfaces can also be described in parametric form as a list of the x, y and z coordinates of points on the surface expressed as functions of two parameters.

Unlike two-dimensional plots, Maple does not use adaptive plotting when drawing surfaces in three dimensions. By default, it calculates the function values over a 25-by-25 grid in the xy-plane. If the function varies rapidly, this may not give a good representation of the surface and it will appear jagged and uneven when Maple displays it. To increase the resolution of the grid, the user can supply an additional parameter in the form grid=[M,N] where M and N are positive integers which indicate the number of points on the grid in the x and y directions respectively. The Bessel function example above can be re-displayed on a finer grid with a command such as

```
plot3d(BesselJ(0,x)*BesselJ(1,y), x=-5..5, y=-5..5, grid=[50,50]);
```

but when using larger grids, Maple will also take longer to calculate the coordinates of the points on the surface. In addition, some smaller computers such as PC-compatibles may be unable to display Maple's three-dimensional graphics when the user has specified a very large grid.

The plots package extends the capabilities of Maple's three-dimensional plotting system by introducing several alternative ways of specifying surfaces. These include sphereplot, cylinderplot and tubeplot.

The sphereplot procedure is the three-dimensional analogue of polarplot. It allows the user to define a surface in spherical coordinates, in the form

$$r = f(\theta, \phi).$$

The simplest example is that of a sphere, which can be drawn with the command

```
sphereplot(1,theta=-Pi..Pi,phi=0..Pi);
```

If the range of the angular coordinates is chosen differently, we can draw portions of a sphere. For example,

```
sphereplot(1,theta=-Pi..Pi,phi=0..Pi/2);
```

draws only the *upper hemisphere* i.e. that part of the sphere with $z > 0$.

An ellipsoid can be drawn by specifying the appropriate radial distance as a function of the angular coordinates. For example,

```
ellipsoid := sqrt( ((0.4*cos(theta)^2 + 0.6*sin(theta)^2)*sin(phi)^2
    + 0.8*cos(phi)^2):

sphereplot(ellipsoid, theta=-Pi..Pi, phi=0..Pi);
```

Surfaces can be defined in cylindrical polar coordinates using `cylinderplot`. In its simplest form, this requires an expression giving the radial coordinate r as a function of the azimuthal angle θ and the height z above the xy-plane. The following example draws a cone.

```
cylinderplot(20-z, theta=-Pi..Pi, z=0..20);
```

whilst this example draws a cylinder whose cross section is not circular and whose radius varies periodically:

```
cylinderplot((2+cos(theta))*sin(z/2),theta=-Pi..Pi,z=0..20);
```

The `tubeplot` function allows the user to draw surfaces which are tubes surrounding a specified curve in three-dimensional space. The user defines the curve representing the axis of the tube in parametric form, as a list of three functions of a single parameter. The user can also, optionally, define the radius of the tube at each point as another function of the parameter. The following example draws a helical tube whose cross-sectional radius is 0.2 and whose axis lies on a cylinder of radius 3:

```
tubeplot([3*cos(x), 3*sin(x), x], x=0..10, radius=0.2);
```

The next example draws a torus whose cross-sectional radius is a periodic function of the azimuthal angle:

```
tubeplot([10*cos(x), 10*sin(x), 0], x=-Pi..Pi, radius=1+0.5*sin(3*x));
```

By analogy with the `grid=[M,N]` option of the surface-plotting routines, the user can specify the number of points on the axis of the tube using the `numpoints` option and the number of points around the axis using the `tubepoints` option. The default values of these options are 50 and 10 respectively. For example, to plot the helical cylinder using 100 points along the axis curve and 20 points around the axis, we could write

```
tubeplot([3*cos(x), 3*sin(x), x], x=0..10, radius=2, numpoints=100,
     tubepoints=20);
```

9 Further Information

9.1 The Help System

Maple has an extensive on-line help facility which can provide information on Maple syntax, datatypes and library functions. For advice on how to use the help system, enter

```
help();
```

whilst the command

```
help(index);
```

lists the categories in which more detailed help is available. These include `library`, `libmisc` and `datatypes`. Thus

```
help(index,library);
```

lists the names of all of the Maple standard library functions. These include `expand` and `simplify`. Thus the command

```
help(expand);
```

displays a complete description of the `expand` function as it appears in the *Maple Library Reference Manual*. It will also contain some examples of the `expand` functions.

If you know the name of a standard or miscellaneous library function and want to know how to use it, you can invoke the help system to get that information directly. For example, to get a description of the miscellaneous library function `laplace`, enter

```
help(laplace);
```

The help system can also provide descriptions of functions that are members of a package. The command

```
help(packages);
```

lists all of the packages available in the current version of Maple. These include `numtheory` and `linalg`. To list the names of the functions in the linear-algebra package, enter

```
help(index,linalg);
```

and to get a description of the `linalg` function `inverse`, enter:

```
help(linalg,inverse);
```

Maple also allows the user to request help by typing a question mark (`?`) followed by the topic upon which help in needed, as shown in the next example.

```
?intro
```

will provide an introduction to the Maple system. Notice that the statement is *not* terminated by a semicolon.

To display a description of Hermite polynomials, the following command is used

```
?orthopoly[H]
```

or its equivalent

```
?orthopoly,H
```

When Maple fails to find the information on the keyword specified, it displays a list of keywords that matches the first three characters of the given keyword. This feature can be used to display a list of names starting with a given letter or string of letters and is useful if you cannot remember the full name of the function on which you need help. For example,

```
    ?L

Try one of the following topics:
                          {orthopoly[L], numtheory[L]}

    ?l
Try one of the following topics:
{libname, lcoeff, ldegree, length, lprint, list, ln, lasterror, leadterm,

    lexorder, lessthanequal, linalg, lcm, lhs, log, L, liesymm, logic,

    linemeet, linsolve, laguerre, listlist, laplacian, lastname, labeling,

    lastpart, lccutr2p, leastsqrs, lessequal, legendre, Lrank, local, laurent,

    lccutc, listsum, lccutr, ln03, leadmon, lnGAMMA, left, leftbox, libmisc,

    logical, loop, line, LaTeX, Limit, leftsum, lambda, Laplace, Latin,

    linear, Lie, lib, linearregression, lattice}

    ?la
Try one of the following topics:
{lasterror, laguerre, laplacian, lastname, labeling, lastpart, laurent, LaTeX,

    lambda, Laplace, Latin, lattice}
```

9.2 Learning by examples

The Maple help system provides the user with an extensive range of examples covering most of Maple's capabilities. Should this not be enough, the user has access to the `example` command that will illustrate the use of Maple functions and datatypes. For instance, to generate an example of a procedure, the following command is used:

```
example(procedure);

                  proc(y) A(y)*B(y)*cos(y) end
```

In the case of a function, the following commands would be used

```
example(function);

                      f(ln(y + 8/5), 5)

example(evalb);

                         evalb(x = x)
```

The `example` command can also be used to generate samples of Maple datatypes as seen in the following examples:

```
example(lhs);

                     lhs(tan(ln(d)) <= 3/4)

example(set);

                        {6, A, c, 1/7}

example({name,algebraic}=[set,list]);

       g(sin(((exp(n) + 8) a + 1/5) m y ln(a)))  + B = [{4}, [B, 8, 1, C]]

example({name,algebraic}=[set,list]);

                   B[7, 2, 4] = [{2, 1/14}, [10]]
```

The last two cases show that the `example` procedure generates random examples of the specified data type.

9.3 The Maple Language and Library Reference Manuals

Maple is supplied with a comprehensive set of manuals.

First Leaves: a Tutorial Introduction to Maple V provides an introduction similar to the guide that you are now reading, with many more examples of the ways in which Maple can be used. In particular, it includes many samples of Maple's graphical output.

The *Maple Language Reference Manual* is a complete guide to the Maple language and its datatypes. This is an invaluable reference for anyone who plans to write their own procedures.

The *Maple Library Reference Manual* contains a detailed description of every function in the Maple library, including all of the packages. However, this information is also provided by Maple's on-line help system, which is often more convenient to use than a printed manual.

9.4 Distributors of Maple world-wide

Maple is a product of Waterloo Maple Inc.

Waterloo Maple Inc.
450 Phillip Street
Waterloo, Ontario
CANADA, N2L 5J2

Telephone: (519) 747-2373

FAX: (519) 747-5284

E-mail: *info_web@maplesoft.com*

WWW: `http://www.maplesoft.com/`

List of distributors world-wide: `http://www.maplesoft.com/Company/OrderInfo.html`