

Notes on Computational Mathematics: Maple

Robert L. Higdon
Department of Mathematics
Oregon State University
Corvallis, Oregon 97331-4605

Revised May 1996

Introduction

These notes were originally developed for a course in computational mathematics given in the Department of Mathematics at Oregon State University. The goals of the course are to give an introduction to some standard mathematical software packages and to describe some mathematical topics that can be illuminated by computational examples and experimentation. The present notes were developed for the portion of the course that is concerned with Maple.

Maple is a software package that is devoted primarily to symbolic computation. It was developed at the University of Waterloo in Waterloo, Ontario. The present notes are based on Maple V Release 3. Much more extensive information about Maple can be found in the references *First Leaves: A Tutorial Introduction to Maple V*, *Maple V Language Reference Manual*, and *Maple V Library Reference Manual*, all by B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt. The publisher is Springer-Verlag. Also check the Web site <http://www.maplesoft.com>

The first section of these notes, *Maple fundamentals*, refers occasionally to the implementation of Maple on the network of Sun workstations in the Department of Mathematics at Oregon State University. These workstations operate under the Unix operating system and use the X Window System. Sun is a trademark of Sun Microsystems, Inc.; Unix is a trademark of American Telephone and Telegraph Company; X Window System is a trademark of M.I.T.

Maple fundamentals

1. Beginning and ending a Maple session
2. Getting help
3. Some syntax rules
4. Entering formulas
5. Obtaining hardcopy
6. Using packages

1. Beginning and ending a Maple session

One way to enter Maple is to type the name `maple` at the prompt from the operating system and then press the `return` key. To exit Maple, type `quit` at the Maple prompt `>`, and then press the `return` key.

If you are working in the X Window System, another way to enter Maple is to type `xmaple` at the Unix command prompt. The outline of a special Maple window will appear on the screen, with the mouse arrow at the upper left corner of the window. Move the window to the desired location, and then click a mouse button to activate the window. The window can be moved, resized, and iconified in the usual way. A scroll bar is located on the right side of the window. If you intend to use Maple to produce graphics, you should use `xmaple`. To exit Maple in this case, either type `quit` at the Maple prompt `>`, or use the mouse to select the item `File` at the top of the Maple window and then select the item `Exit` in the menu that pops up.

Occasionally you may enter a command and then decide that you want to stop execution of that command. For example, the command may be taking a long time to execute, and you might not want to wait that long. If you entered Maple by typing `maple`, type `ctrl c` to stop execution of a command; that is, hold down the `control` key and press `c`. If you are using `xmaple`, use the mouse to click on the box labelled `Interrupt` that is located near the top of the Maple window.

2. Getting help

Maple has a very extensive on-line help facility. If you are working in the X Window System and entered Maple by typing `xmaple`, the help facility can be accessed in a manner that is extremely convenient. Use the mouse to click on the item `Help` in the upper right corner of the Maple menu, and then select the item `Help Browser`. The outline of a new window will appear on the screen; move the window to the desired location and then press a mouse button to activate the window. Several subwindows will be contained in this new window. The left-most subwindow contains the names of some help categories. One of these categories is `Mathematics...`; to get information on this topic, move the mouse pointer to that name and press the left mouse button. A list of subtopics will then appear in the second subwindow. This list of subtopics may be too long to fit into the subwindow, so you may have to use the scroll bar on the right side of that subwindow. One of the subtopics is `Calculus...`; if you click on this name, a list of subtopics is listed in the third subwindow. One of the items in the third subwindow is `Differential Calculus...`; if

you click on this item, a list of items will appear in the fourth subwindow. One of those items is `diff`; if you click on that name and then click on the box labelled `Help` at the bottom of the help window, Maple will create a new window that contains information about the function `diff`. To see the entire contents of the file that is displayed, it will probably be necessary to use the scroll bar on the right side of that window. To close a help window, select the `File` menu at the top of the window and then select `Exit`.

In general, the help item for a function typically includes syntax rules, several examples, and references to related functions. The examples can be copied and pasted from the help window into the Maple window by using the mouse in the usual manner.

The help facility can also be accessed by typing commands at the Maple prompt `>`; if you are not using `xmaple`, this is the only way to gain access to the help facility. If you type `?` at the Maple prompt and then press the `return` key, you will receive a description of the help facility, including a list of the major help categories.

3. Some syntax rules

Every Maple command should end either with a semicolon `;` or a colon `:`. If a command ends with a semicolon, its output is displayed; if a command ends with a colon, the output is suppressed. Several commands can be typed on one line. After you type a line, press the `return` key to execute the command(s) on that line.

The operator `:=` is used to assign values to variables.

At any stage in a Maple session, the double-quote symbol `"` refers to the output of the preceding command. The output of the command before that one can be referred to as `" "`, and the output three commands back can be referred to as `" " " "`.

Example 3.1. The following are some examples of Maple commands.

```
y := (x+1)^10;
expand(y);
factor(");
z := x^2 + 2*x - sin(x) + 1/x;
diff(z, x);
int(z, x);
w := exp(-x^2) / (x^4 + 1);
series(w, x=0, 10);
```

During a Maple session, you may want to remove a value that has been assigned to a variable. That is, you may want to “clear” a variable. For example, to remove a value assigned to the variable `x`, use `x := 'x';`.

Exponentiation can be denoted by either a hat `^` or a double asterisk `**`. For example, `x^2` and `x**2` are the same.

Within one input line, all characters which follow the symbol `#` are considered to be comments and are not executed.

4. Entering formulas

The following are several methods for making it more convenient to enter commands into a Maple session.

(1) If you are using `xmaple`, you can edit and re-execute a command that was entered previously. (This is useful when correcting typographical errors.) Just use the mouse to set the cursor at the desired location, insert or delete characters as needed, and press the `return` key.

(2) If you entered Maple by typing `maple` (not `xmaple`), it is possible to use the up-arrow key to recall previous lines on the current command line. The down-arrow key can be used to move back toward the most recent line. To edit a command line, use the left-arrow and right-arrow keys to move back and forth through a line, use the `delete` and/or `backspace` keys to delete text, and type at the cursor to insert text. This procedure is convenient if you need to correct a typographical error or execute a command that is similar to one used previously.

(3) If you are working in the X Window System, you could have one window open to Maple and another window open to an editing session on a file. Enter commands into that file, and then use the mouse to copy and paste commands into the Maple window. You can also copy and paste within the Maple window; for example, you can use the mouse to highlight a portion of a previous command and then use the `Copy` and `Paste` commands within the `Edit` menu at the top of the Maple window. After you copy and paste, you may need to press the `return` key in order to cause the command(s) to be executed; do this while the mouse pointer is in the Maple window.

(3) You can put a sequence of Maple commands into a file, and then read the file into a Maple session. For example, suppose that some Maple commands are contained in a file named `prog` in your current working directory. At the Maple prompt, type `read prog; .` If the filename contains unusual characters such as a period or a slash, the filename should be enclosed in backquotes `' '`. If you are using `xmaple`, an alternative is to select the item `File` at the top of the Maple window and then select the item `Include...` on the menu that pops up. Maple will respond by displaying a window that lists the files in your current working directory; you can then select the appropriate filename and click on the box labelled `Read`.

5. Obtaining hardcopy

It is often useful to print a hardcopy of your Maple session. However, during your session you may have made some errors and false starts, and it would be a good idea to delete these portions before printing. If you are using `xmaple`, this can be done easily. To select a portion of text to be deleted, use the mouse to move the arrow to one end of the portion of text, press the left button, drag to the other end, and then release the button. Then either press the `Delete` key, or select the item `Delete Selection` under the `Edit` menu at the top of the Maple window. If you entered Maple by typing `maple` instead of `xmaple`, you can write a transcript of your session into a file as described in (3) below and then edit the file before printing.

The following methods can be used to obtain hardcopy of your session.

(1) Suppose that you are using the X Window System and have entered Maple by typing `xmaple`. A plain text transcript of your session can be written to a file by the following procedure. Use the mouse to select the item `File` at the top of the Maple window, and then select the item `Export as Text...` on the menu that pops up. The system will then display a small window that shows the name of your current working

directory and its subdirectories. Use the mouse to click on the name of the directory where the transcript should be sent. The box entitled **Selection** will then contain the pathname of that directory; at the end of this pathname, insert the name of the destination file. Finally, click on the box labelled **Export** . The file can then be printed by using a suitable Unix command.

(2) It is possible to use `xmaple` to create a transcript in the form of a Postscript file. (“Postscript” is a graphics description language.) When such a file is printed, the hardcopy will have higher quality than a plain text transcript. To create a transcript in Postscript, select the item **File** at the top of the Maple window and then select the item **Print...** in the menu that pops up. In the new window that appears, you have two options. If you select the item **Printer Command** and then click on the button labelled **OK** , then the transcript is sent directly to the printer. On the other hand, if you want to send the transcript to a file, select the item **Output to File** , insert the filename into the box immediately to the right (the default name is `session.ps`), and then click on the box labelled **OK** . Under the present configuration of our system, a Postscript file can be viewed on the screen by using the command `ghostview` ; for example, to view a file named `session.ps` , execute the command `ghostview session.ps` at the Unix command prompt (not in the Maple window). You could also print the file by using the command `lpr` , e.g., `lpr session.ps` .

(3) Under the present configuration of our system, it is possible to write output to a file by using the `script` command. This is a Unix command, not a Maple command, and it can be used whenever you need to have a transcript of a terminal session. The following procedure can be used if Maple is entered by typing `maple` instead of `xmaple` . Suppose that you want to write Maple output to a file named `fname` ; this name could either be a simple filename or a pathname to a file in a different directory. Before you enter Maple, type `script fname` at the Unix command prompt. Everything that subsequently appears on that terminal or window is then written to the file `fname` ; if this file already exists, its previous contents will be overwritten. Enter Maple, do your work, and then exit from Maple. Then end the script session by typing `exit` . The script file `fname` will typically contain stray characters; to clean up the file, type `cleanscript fname` . The file `fname` will then contain a complete transcript of your session, including errors and false starts, so you will probably need to edit the file before printing.

A useful variation on the `script` command is the following. Suppose that a file `fname` already exists, and you want to append some output onto the end of that file and not overwrite what is already in the file. In that case, use the command `script -a fname` instead of `script fname` .

6. Using packages

As mentioned above, Maple has a number of “packages”, which are sets of functions that supplement the standard library functions. If you want to use a function in a package, you need to refer explicitly to that package. For example, if you want to use some functions in the `linalg` package, first type `with(linalg):` at the Maple prompt. This command has the effect of making all of the functions in the `linalg` package known to your current Maple session. If the statement were terminated with a semicolon instead of a colon, you would receive a list of all of the functions in the package. If the statement is terminated with a colon, the list is suppressed.

Manipulating numbers and expressions

1. Numbers
2. Factoring numbers and expressions
3. The functions `expand` and `combine`
4. The function `simplify`
5. The function `normal`
6. The functions `numer` and `denom`
7. The function `subs`
8. The functions `collect` and `coeff`
9. The function `convert`
10. Solving equations

1. Numbers

Maple can perform exact computations involving rational numbers and numerous special functions and constants. Some examples are the following.

```
72/5 + 13/15 - 2/3;
const := 7391073920/463792;
evalf(const);
evalf(Pi, 50);
sqrt(-8);
2 / sqrt(3);
sin(Pi/2); sin(Pi/3); sin(Pi/6);
exp(Pi * I);
```

The function `evalf` provides a floating-point approximation to a specified number. A second argument, if given, specifies the number of decimal digits to be shown in the result.

The built-in constants include `Pi`, `E`, and `I`.

2. Factoring numbers and expressions

The function `ifactor` can be used to factor integers and rational numbers. In the case of rational numbers, the numerator and denominator are factored. An optional second argument to the function `ifactor` can be given to specify the algorithm that is to be used; for details, see `?ifactor`. If the second argument to `ifactor` is the character string `'easy'`, then Maple will compute those factors that are easy to compute and will indicate the number of decimal digits in the remaining composite factor that was not factored. The function `isprime` tests for primality. A factored form can be multiplied out by using the function `expand`. Some examples are the following.

```
ifactor(482074307432);
expand("");
ifactor(32/81);
```

```

expand("");
isprime(1023);
ifactor(87940237840783297480924879237423903, 'easy');

```

The function `factor` can be used to factor polynomials and rational functions. Some examples are the following.

```

p := x^16 - 1;
factor(p);
q := (x^4 - 1) / (x^2 - 4);
factor(q);

```

In the examples just given, the expressions are factored over the field of rational numbers. That is, factors with rational coefficients are computed. However, it is possible to extend the field of rational numbers and then factor over the extended field.

Example 2.1. The rational factorization of the polynomial $p(x) = x^{16} - 1$ includes some factors of degree greater than 1. It is clear from the form of these factors that the polynomial could be factored further if some even roots of -1 were included in the field over which p is factored. The command `factor(p, I)`; will factor p over the field that is obtained by extending the field of rational numbers with the root $i = \sqrt{-1}$. To factor all the way to linear factors, use the following commands.

```

r := I^(1/4);
factor(p, r);

```

Example 2.2. It is possible to add more than one object to the field of rational numbers; just specify a set of objects. For example, the following commands produce a factorization of the polynomial $f(x) = x^2 + 8$ over the field $Q(i, \sqrt{2})$ which is obtained by extending the field of rational numbers with i and $\sqrt{2}$.

```

f := x^2 + 8;
factor(f);
factor(f, {I, sqrt(2)});

```

An alternative to the last line is the following pair of statements.

```

K := {I, sqrt(2)};
factor(f, K);

```

3. The functions `expand` and `combine`

If the function `expand` is applied to an expression involving products and sums, the effect is to apply the distributive law where applicable. If the function `expand` is applied to a rational expression, the expression is broken into a sum of terms. The function `expand` can also be applied to numerous special functions. For more information, type `?expand`. Some examples are the following.

```

p := (x-1)*(x+2)^5 + (y-1)^2;
expand(p);

```

```

q := (x-1) / ( (x-2)*(x+3) );
expand(q);
expand( sin(2*x) );
expand( tan(x + y) );
z := ln(x * y^2 / (a+b) );
expand(z);
expand( exp(x + y) );
expand( BesselJ(2, x) );

```

For many special mathematical functions, the Maple function `combine` performs operations that are essentially the inverses of the operations performed by the function `expand`.

```

z := ln(x * y^2 / (a+b) );
expand(z);
combine(" , ln);
combine( 2*sin(x)*cos(x), trig );
expand("");
w := exp(x) * exp(y) / exp(2*x);
combine(w, exp);
z := sin(x)^9; combine(z, trig);
z := sin(x)^4 + cos(x)^4; combine(z, trig);

```

In the next-to-last line, the result is a linear combination of $\sin x$, $\sin 3x$, $\sin 5x$, $\sin 7x$, and $\sin 9x$. In the last line, the result is $3/4 + (\cos 4x)/4$.

4. The function `simplify`

The function `simplify` will apply various simplification procedures to an expression. Some examples are the following.

```

q := x/(x^2 - 1) + x^2 / (x+1);
simplify(q);
t := sin(x)^2 + cos(x)^2;
simplify(t);
z := ln(x * y^2 / (a+b) );
simplify(z);

```

5. The function `normal`

The function `normal` will convert a rational function into a ratio of relatively prime polynomials having integer coefficients. Suppose that `q` is a rational expression. If `normal(q)` is executed, the numerator and denominator are products of polynomials. If `normal(q, expanded)` is executed, the numerator and denominator are expanded polynomials. For more information, type `?normal`. Some examples are the following.

```

q := x/(x^2 - 1) + x^2 / (x+1);
normal(q);
normal(q, expanded);

```

```

r := 1/a + 1/b + (a+b)^(-1);
normal(r);
normal(r, expanded);

```

6. The functions `numer` and `denom`

The functions `numer` and `denom` yield the numerator and denominator, respectively, of a rational expression. The expression is first converted into a single fraction, and the numerator or denominator of the result is then selected.

Example 6.1. The following statements define a rational function and then select the numerator and denominator of a combined form of that function.

```

q := x / (x^2 - 1) + x^2 / (x+1);
numer(q); denom(q);

```

The results of the commands `numer(q)`; and `denom(q)`; have a common factor. This could be avoided by executing the command `q := normal(q)`; before using the functions `numer` and `denom`.

7. The function `subs`

The function `subs` can be used to substitute values of independent variables into an expression. More generally, it can be used to substitute subexpressions into an expression.

Example 7.1. The command `subs(x=0, y=1, x^2 - y^2)`; evaluates the expression $x^2 - y^2$ for $(x, y) = (0, 1)$.

Example 7.2. In the following, it is clear that $z = 0$ when $x = 0$. However, when one looks at the form of w that is created by the third statement, it is not immediately obvious that $w = 0$ when $x = 0$. The `subs` function shows that this is the case.

```

z := sin(x)^20;
subs(x=0, z); evalf("");
w := combine(z, trig);
subs(x=0, w); evalf("");

```

Example 7.3. In the following sequence of commands, the relation $\sin 2x = 2 \sin x \cos x$ is substituted into the expression $(\sin 2x)^5$. The final result is $32 \sin^5 x \cos^5 x$. In Maple, a single equals sign `=` defines an equation, in the logical sense; `=` is not an assignment operator.

```

z := sin(2*x)^5;
sin(2*x) = 2 * sin(x) * cos(x);
subs("", z);

```

8. The functions `collect` and `coeff`

Suppose that `p` is a polynomial involving the quantity `expr`. The command `collect(p, expr)` re-arranges the expression so as to collect together all terms having the same power of `expr`. The command `coeff(p, expr)` produces the coefficient

of `expr` in `p`, and the command `coeff(p, expr, n)` produces the coefficient of expr^n in `p`. The operations `coeff(p, x, n)` and `coeff(p, x^n)` are equivalent. For more information, type `?collect` and `?coeff`.

Some examples are the following.

```
w := expand( (1 + x + y)^4 );
collect(w, x);
collect(w, y);
coeff(w, x, 2);
coeff(w, x^2);
z := combine( sin(x)^20, trig );
coeff(z, cos(6*x) );
```

9. The function `convert`

The function `convert` can be used to convert expressions from one form into another. The list of possibilities is extensive; see `?convert`. Examples include converting a rational expression into a sum of partial fractions (the `parfrac` option), converting between degrees and radians, converting numbers to different bases, and converting English units to metric units. Further information can be obtained from the help menu; for example, to learn about conversion to metric units, type `?convert[metric]`. The following are some examples of conversion to metric.

```
convert(furlongs, metric);
convert(65*MPH, metric);
convert(35*MPG, metric, US);
convert(acres, metric);
sqrt("); convert(yards, metric); ""/";
```

10. Solving equations

The function `solve` can be used to seek exact (symbolic) solutions to an algebraic equation or system of algebraic equations. For example, the command `solve(x^8 - 1 = 0, x)`; finds the exact solutions to the equation $x^8 - 1 = 0$.

Example 10.1. In the following, the first command defines a set of equations to be solved. The second command computes exact solutions to the simultaneous system of equations.

```
eqn := {x^2 + y^2 = 2, x*y = 1};
soln := solve(eqn, {x, y});
```

The output of the second line is `soln := {y = 1, x = 1}, {y = -1, x = -1}`. It is then possible to refer to parts of this result and substitute those parts into other expressions. For example, `soln[1]`; yields `{y = 1, x = 1}`, and `soln[1][2]`; yields `x = 1`. If you then want to substitute the result `{y = 1, x = 1}` into the expression $x + 2y$, for example, use `subs(soln[1], x + 2*y)`;

Several other related functions are the following.

`fsolve` Compute floating-point (approximate) solutions.
`isolve` Find integer solutions.
`msolve` Solve over modular integers.
`dsolve` Find symbolic or numerical solutions to ordinary differential equations.

For more information, consult the help facility, e.g., type `?fsolve` . Information about numerical solutions to ordinary differential equations can be obtained by typing `?dsolve[numeric]` .

For systems of linear equations, you can use the `linalg` package. For a list of functions in that package, type `?linalg` . The function `linsolve` within that package can be used to find solutions to linear systems; for more information, type `?linsolve` or `?linalg[linsolve]` . The function `leastsqrs` in the `linalg` package can be used to find least-squares solutions to over-determined linear systems.

Linear algebra

1. Constructing matrices
2. Constructing vectors
3. Matrix operations
4. Linear systems, rank, inverse, and related topics
5. Eigenvalues and eigenvectors
6. Jordan canonical form

1. Constructing matrices

The function `matrix` in the `linalg` package can be used to create matrices. To load the `linalg` package, type `with(linalg):` at the Maple prompt. An alternative is to use the function `array`, which is one of the standard library functions; for more information, type `?array`. To see a listing of the functions in the `linalg` package, type `?linalg`.

A matrix can be created by typing all of its entries explicitly. For matrices having special structures, other possibilities are available. The entries of a matrix can be numbers or symbols.

In the following examples, it is assumed that the `linalg` package has already been loaded.

Example 1.1. Here, a matrix is created by typing all of its entries explicitly. The following two commands are equivalent.

```
A1 := matrix( [ [1,x], [b,16] ] );  
A2 := matrix(2, 2, [1, x, b, 16]);
```

In the first case, the matrix is defined as a list of rows, each of which is defined by a list. (In Maple, “lists” are enclosed in square brackets.) In the second case, the first two arguments to the function `matrix` give the dimensions of the matrix, and the entries of the matrix are then given in a single list.

If you want to view a matrix, use the `print` command. For example, to view the matrices `A1` and `A2` created above, use the commands `print(A1);` and `print(A2);`, respectively.

Example 1.2. The function `band` can be used to create banded matrices, namely, matrices that are constant along diagonals. (Such matrices are called Toeplitz matrices, and the Maple function `toeplitz` can be used in a similar manner for the special case where the matrix is symmetric.)

The following statements produce a 6×6 matrix that has 2’s along the main diagonal, -1 ’s immediately above and below the main diagonal, and zeros elsewhere.

```
n := 6;  
A := band([-1, 2, -1], n);
```

The statement `A := band([-1, 2, -1], 6);` would have the same effect. The version with the symbol `n` would be useful when writing general programs.

In general, the first argument given to the function `band` is a list having an odd number of elements. The middle element gives the value of the entries along the main diagonal. The elements to the left of the middle define the subdiagonal, and elements to the right of the middle define the superdiagonal. If the list is too short to cover the entire matrix, then the remaining entries of the matrix are set equal to zero.

The command `tri := n -> band([-1, 2, -1], n);` defines a function `tri` such that `tri(n)` is the $n \times n$ matrix having the form described above.

Example 1.3. In the following sequence, the first command creates the 6×6 identity matrix, and the second command defines a function `id` so that `id(n)` is the $n \times n$ identity matrix.

```
A := band([1], 6);
id := n -> band([1], n);
```

Example 1.4. It is possible to define an arbitrary function of the row and column indices, and then use that function to define a matrix. For example, the (i, j) 'th entry of the Hilbert matrix is $1/(i + j - 1)$. The 6×6 Hilbert matrix can then be defined by the following commands.

```
h := (i,j) -> 1 / (i+j-1);
matrix(6, 6, h);
```

The following statements define a function `hilb` such that `hilb(n)` is the $n \times n$ Hilbert matrix.

```
h := (i,j) -> 1 / (i+j-1);
hilb := n -> matrix(n, n, h);
```

Actually, the Hilbert matrix is already built into Maple; the $n \times n$ Hilbert matrix can be obtained by typing `hilbert(n);` .

Example 1.5. Suppose that you want to create a matrix in which most of the entries have the same value. You could start with a matrix in which all entries have the same value, and then modify a few entries with explicit assignments. For example, if you want to create the tridiagonal matrix described in Example 1.2, you could start with a matrix of zeros and then modify the entries on and adjacent to the diagonal. This is accomplished by the following statements. For more information about `do` loops and `while` loops, type `?do` or `?while` .

```
n := 6;
A := matrix(n, n, 0);
for i from 1 to n do
    A[i, i] := 2;
od;
for i from 1 to n-1 do
```

```

        A[i, i+1] := -1; A[i+1, i] := -1;
    od;
    print(A);

```

Example 1.6. Random matrices can be generated with the `randmatrix` function. For example, the command `randmatrix(4, 3);` generates a 4×3 matrix of random integers in the range `-99..99`. The command `randmatrix(4,4, entries = rand(-9999..9999));` generates a 4×4 matrix of four-digit random integers. The command `randmatrix(3,3, symmetric);` generates a symmetric 3×3 matrix of random 2-digit integers, and `randmatrix(6,6, sparse);` generates a random 6×6 sparse matrix. A sparse matrix is a matrix in which most of the entries are zero.

The Maple function `rand` generates random integers, and the function `randpoly` generates random polynomials with integer coefficients. For more information, consult the help facility.

2. Constructing vectors

The Maple function `vector` can be used to construct vectors. This function is in the `linalg` package, and it is used in a manner similar to the function `matrix`. An alternative is to use the function `matrix` to create matrices having one row or column. To view the contents of a vector that has already been created, use the `print` command, e.g., `print(v);`.

In the following examples, it is assumed that the `linalg` package has already been loaded.

Example 2.1. A vector can be created by giving an explicit list of components. An example is given by the command `v := vector([1, 4, 9, 16]);`. When this vector is displayed by Maple, the entries are given in a row; however, Maple interprets this vector internally as a *column* vector. The vector `transpose(v)` is interpreted as a row vector.

Example 2.2. The command `v := matrix(4, 1, [1, 4, 9, 16]);` creates a 4×1 matrix having the indicated components. When this vector is displayed by Maple, the entries are displayed in a column, so it really does look like a column vector. The vector `transpose(v);` is interpreted internally by Maple as a row vector, and it is displayed as such. A row vector with the indicated entries can also be created with the command `w := matrix(1, 4, [1, 4, 9, 16]);`.

Example 2.3. It is possible to define an arbitrary function of the index, and then use that function to define a vector. For example, the following statements produce the same result as in Example 2.1.

```

f := i -> i^2;
v := vector(4, f);

```

The following statements produce the same column vector as in Example 2.2.

```

f := i -> i^2;
v := matrix(4, 1, f);

```

Example 2.4. It is possible to create an empty vector, and then assign the elements explicitly. For example, the following commands produce the same result as in Example 2.1.

```
v := vector(4);
v[1] := 1; v[2] := 4; v[3] := 9; v[4] := 16;
```

Example 2.5. In the following sequence of commands, the first statement defines a function `zeros` such that `zeros(n)` is a vector consisting of n zeros. The second statement defines a function for creating a vector consisting of n ones.

```
zeros := n -> vector(n, 0);
ones := n -> vector(n, 1);
```

Example 2.6. The components of a vector can be symbols, as well as numbers. An example is the vector `v` that is created by the command `v := vector([1, x, x^2, x^3, x^4]);`. More generally, the following commands create a function `powers` such that `powers(n)` is the vector $(1, x, x^2, \dots, x^n)^T$.

```
f := i -> x^(i-1);
powers := n -> vector(n+1, f);
```

3. Matrix operations

Additions and multiplications involving matrices and scalars can be performed in either of two ways.

One way is to apply the function `evalm` to the expression. (The function `evalm` produces evaluation over the matrices.) In this case, non-commutative matrix multiplication is denoted by the symbol `&*`, and scalar multiplication is denoted by `*`. If `A` is a square matrix and `c` is a scalar, then an expression `A + c` is interpreted as $A + cI$, where I is the identity matrix having the same dimensions as A .

Another way to perform additions and multiplications involving matrices and scalars is to apply specific Maple functions; the function `add` is used to add two matrices, the function `multiply` is used to multiply matrices, and the function `scalarmul` is used to multiply a matrix by a scalar.

In the following examples, it is assumed that the `linalg` package has already been loaded.

Example 3.1. Suppose that 4×4 matrices `A` and `B` and a four-component vector `v` are defined as follows.

```
A := band([-1, 3, 1], 4);
B := band([2], 4);
v := vector(4, 1);
```

For each of the following lines, all commands within that line give the same result.

```
evalm(A + B); add(A,B);
evalm(A &* B); multiply(A, B);
```

```

evalm(A &* v); multiply(A, v);
evalm(A &* A &* A &* A); evalm(A^4); multiply(A,A,A,A);
evalm(3*A); scalarmul(A, 3);

```

Example 3.2. The function `evalm` is convenient for complicated expressions. For example, in the following sequence of commands, the first statement defines a symbolic expression involving the symbol `A`; the second statement then evaluates that expression for the current value of `A`, provided that `A` is a square matrix. An analogous expression involving the functions `add`, `multiply`, and `scalarmul` would be more cumbersome.

```

p := A^4 - 3*A^2 + A + 2;
evalm(p);

```

Example 3.3. Suppose that a vector `v` is defined by the statement `v := vector([1, 4, 9, 16])`; , and suppose that a vector `w` is then defined by `w := transpose(v)`; . Maple interprets the vector `v` internally as a column vector, so `w` is interpreted as a row vector. The command `evalm(v &* w)`; then produces a 4×4 matrix, as expected, and the command `evalm(w &* v)`; produces the dot product of `v` and `w`.

It was mentioned earlier that non-commutative matrix multiplication is denoted by the symbol `&*`, not `*`. In the present example, suppose that one were to make a syntax error by executing the commands `evalm(w*v)`; and `evalm(v*w)`; instead of the ones described in the preceding paragraph. The results of the commands `evalm(w*v)`; and `evalm(v*w)`; are both 1×1 matrices (i.e., scalars), so one of the results is not what the user intended.

4. Linear systems, rank, inverse, and related topics

The following functions are included in the `linalg` package. For more information on these commands, consult the help facility. In the examples given in this section, it is assumed that the `linalg` package has already been loaded.

<code>linsolve</code>	Solve a linear system.
<code>leastsqrs</code>	Find the least-squares solution to an overdetermined linear system.
<code>cond</code>	Condition number.
<code>gausselim</code>	Perform Gaussian elimination on a matrix.
<code>rref</code>	Reduced row-echelon form.
<code>gaussjord</code>	Same as <code>rref</code> .
<code>inverse</code>	Inverse of a matrix.
<code>augment</code>	Create an augmented matrix.
<code>det</code>	Determinant of a matrix.
<code>rank</code>	Rank of a matrix.
<code>colspace</code>	Compute a basis for the column space.

<code>rowSpace</code>	Compute a basis for the row space.
<code>range</code>	Same as <code>colSpace</code> .
<code>nullSpace</code>	Compute a basis for the null space.
<code>kernel</code>	Same as <code>nullSpace</code> .

Example 4.1. The following commands produce the solution of a linear system of the form $Ax = b$. The coefficient matrix is a tridiagonal matrix with 3's along the main diagonal, 1's immediately above the main diagonal, and -1 's immediately below the main diagonal.

```
A := band([-1, 3, 1], 4);
b := vector( [4, 3, 3, 2] );
x := linsolve(A, b);
```

Example 4.2. Solve a linear system in which the coefficient matrix and right side contain symbols, not just numbers. The first line of the following sequence removes any pre-existing values for the symbols `x` and `y` .

```
x := 'x'; y := 'y';
A := matrix( [ [1, x], [x^2, y] ] );
b := vector( [x*y, 2] );
linsolve(A, b);
```

Example 4.3. Find the determinant and inverse of a random matrix.

```
A := randmatrix(3, 3);
det(A);
inverse(A);
```

Example 4.4. Find the inverse of a random matrix by using the method taught in linear algebra courses: augment with the identity matrix, and then reduce to row-echelon form.

```
A := randmatrix(3,3);
id := band([1], 3);
C := augment(A, id);
C := rref(C);
inv := submatrix(C, 1..3, 4..6);
```

The last statement extracts the submatrix consisting of rows 1 through 3 and columns 4 through 6. This submatrix is the inverse of the original matrix A .

5. Eigenvalues and eigenvectors

Eigenvalues and eigenvectors can be computed with the functions `eigenvals` , `eigenvecs` , and `Eigenvals` , which are included in the `linalg` package. In the examples given in this section, it is assumed that this package has already been loaded.

If A is a square matrix that contains no floating-point numbers, then the command `eigenvals(A)`; causes Maple to try to find the exact (symbolic) eigenvalues by finding exact roots of the characteristic polynomial. If A contains at least one floating-point number but no symbols, then a numerical algorithm is used to compute approximations to the eigenvalues. The command `eigenvects(A)`; operates similarly, except that eigenvectors are also computed; for more information, consult the help facility.

The function `Eigenvals` uses the QR method to compute numerical approximations to eigenvalues. The function `Eigenvals` is “inert”; this means that the output from this function must be explicitly evaluated with the `evalf` function. If A is a square matrix of numbers, then the command `evalf(Eigenvals(A))`; produces numerical approximations to the eigenvalues. The command `evalf(Eigenvals(A, vect))`; returns the eigenvalues, and it also places corresponding eigenvectors in the matrix `vect`.

Example 5.1. Compute the eigenvalues of a symbolic matrix.

```
A := matrix( [ [1,x], [x^2,1] ] );
eigenvals(A);
```

Example 5.2. Define a function `tri` so that `tri(n)` is the $n \times n$ tridiagonal matrix having 2's on the main diagonal, -1 's immediately above and below the main diagonal, and zeros elsewhere. Then compute eigenvalues for various n . In each of the last three lines displayed below, the first command produces exact (symbolic) formulas for the eigenvalues, and the second statement then produces numerical approximations to those symbolic expressions. The symbolic expressions are messy when $n = 6$, but when $n = 7$ they are fairly simple.

```
tri := n -> band([-1, 2, -1], n);
v4 := eigenvals( tri(4) ); evalf("");
v6 := eigenvals( tri(6) ); evalf("");
v7 := eigenvals( tri(7) ); evalf("");
```

Example 5.3. Use the function `Eigenvals` to compute numerical approximations to the eigenvalues of the matrices in Example 5.2.

```
tri := n -> band([-1, 2, -1], n);
v4 := evalf( Eigenvals(tri(4)) );
v6 := evalf( Eigenvals(tri(6)) );
v7 := evalf( Eigenvals(tri(7)) );
```

Example 5.4. Use the function `Eigenvals` to compute numerical approximations to eigenvalues and eigenvectors of `tri(4)`. The columns of the matrix `vect` are approximations to eigenvectors. The matrix product `inverse(vect) &* A &* vect` is then diagonal, except for the effects of roundoff error.

```
A := tri(4);
evalf( Eigenvals(A, vect) ); print(vect);
evalm( inverse(vect) &* A &* vect );
```

6. Jordan canonical form

The function `jordan` in the `linalg` package produces the Jordan canonical form of a matrix. This function can be applied to matrices consisting of exact numbers and/or symbols, but it will not accept a matrix that contains any floating-point numbers.

The latter point might be explained by the following. The Jordan form involves the eigenvalues of the matrix. Suppose that a matrix consists entirely of numbers (i.e., no symbols), and suppose that finite-precision computations are used to obtain the eigenvalues. The results will be inexact, due to roundoff error, and the computed eigenvalues can then be regarded as the eigenvalues of a perturbed matrix. But if a matrix with multiple eigenvalues is perturbed, the multiple eigenvalues can become distinct, and in that case the perturbed matrix is diagonalizable. In a finite-precision environment, it is therefore difficult to detect whether such a matrix has a non-diagonal Jordan form or is diagonalizable with eigenvalues that are nearly equal. This may be why Maple apparently tries to stick with exact (symbolic) computations which it computes a Jordan canonical form.

The function `JordanBlock` produces a Jordan block. In particular, the command `JordanBlock(expr, n);` produces an $n \times n$ matrix having the value `expr` along the main diagonal, 1's immediately above the diagonal, and zeros elsewhere.

Example 6.1. Produce a Jordan matrix, apply a similarity transformation to disguise it, and then compute a Jordan form. The function `diag` produces a block-diagonal matrix having the diagonal blocks specified. The command `jordan(A);` produces the Jordan form. The command `jordan(A, 'P');` produces the Jordan form and returns a transformation matrix in the matrix `P`. The transformation has the property that PAP^{-1} is a Jordan matrix. In this example, it is assumed that the `linalg` package has already been loaded.

```
B1 := JordanBlock(1, 4);
B2 := JordanBlock(-2, 4);
J := diag(B1, B2);
Q := band([-1,2,-1], 8);
A := evalm(inverse(Q) &* J &* Q);
jordan(A);
jordan(A, 'P');
print(P);
evalm( P &* A &* inverse(P) );
```