1 Programming in Maple

Make sure Maple has been configured properly before trying any of the examples below. Configuration instructions are in the module handbook.

1.1 Getting start

- A Maple program is a sequence of statements, each of which tells Maple to perform some action(s). For example typing

  ```
  restart
  ```

  and pressing `return` causes Maple to clear any currently stored data. It is a good idea to issue a `restart` command immediately before starting a new problem (including at the beginning of the Maple worksheet).

- Pressing `shift+return` moves the cursor down one line without executing. This is useful when writing complicated statements.

- The result of a statement that ends with a colon is not displayed. Otherwise, results will be displayed if they do not generate an excessive amount of output (e.g. a huge matrix). This behaviour can be altered by changing the `printlevel` and `rtablesize` parameters (see §1.12 and §1.9, respectively).

- The black square brackets to the left of the window indicate the scope of **execution groups**. If two statements are inside the same execution group, then they need to be separated by a colon or a semicolon.

- Maple is **case sensitive**; for example `int` and `Int` are different commands.

- Maple has a comprehensive online help system. To obtain information about a command, enter a question mark, followed by the command, e.g. `?evalf`. It is usually easiest to scroll down to the examples at the foot of the help page.

- Maple very rarely crashes, but this does sometimes happen. When you start a new worksheet, save it immediately after putting `restart` on the first line. By default, a saved worksheet will save again automatically every three minutes. Do not turn off autosave.

- If a calculation is taking too long, you can try to stop it by pressing the interrupt button (!) in the toolbar. This may take a few seconds to work, but it may not work at all. This is one reason why you should never deactivate autosave.

1.2 Exact vs app s

Unless told to do tries to produce exact results, which can be problematic. Supp termine whether the inequality

$$\int_0^1 \frac{x}{1+x^5}\,\mathrm{d}x > 0.4$$

holds. The statement

```
int( x / ( 1 + x^5 ) , x = 0..1 )
```


tells Maple to evaluate the integral, but the result is rather complicated, and doesn't really help. The `evalf` command tells Maple to calculate an approximate numerical value, so the result of


```
evalf( Int( x / ( 1 + x^5 ) , x = 0..1 ) )
```

is much easier to interpret. Complicated exact results can cause Maple to slow down and appear to freeze if they are reused in later calculations.

Use `evalf` to avoid complicated exact results.

Remark: `evalf( int( ... ) )` with a lower case 'i' tells Maple to evaluate the integral exactly (if it can) and then convert the result to a decimal. See problem 2.1.

1.3 Comments

Maple ignores material from a `#` symbol until the end of the line. This is used to insert explanatory notes for statements whose effect is not obvious.

```
# Create a 2 x 2 matrix
Matrix( 2 )
```

If a comment is inserted after a statement, then a colon or semicolon must be included in between.

```
a := 1 ;  # Would generate an error without the semicolon
```

1.4 Variables

Variables are used to store data. To assign a value to a variable, use the **assignment operator** :=.

```
a := 27 :
b := 4  :
c := a + b :
c
```

Assignments are **not** equations. The expression on the right is computed, then the result is stored in the variable on the left. This means the right-hand side can reference the variable that is about to be assigned.


```
a := 27 :
a := a + 1 :
a
```


$$28$$

Maple allows sequences of assignments to be made in the same statement.

```
a , b := 15 , 21 :
a
```


$$15$$

```
b
```


$$21$$

This provides a very useful shortcut for swapping variable values.

```
a := 15 :
b := -6 :
a , b := b , a :
a
```

$$-6$$

```
b
```

$$15$$

Often, Maple does not need to distinguish between different types of variable such as integer or complex, but there are situations where this is important.

```
a := 1 :
b := 1.5 :
type( a , integer )
```

$$\mathtt{true}$$

```
type( b , integer )
                              false
```


Variables in Maple can possess more than one type.

```
type( a , inte
type( a , nume
type( a , comp
                              true
```

In this way, Maple differs from languages such as C and Fortran.

See sections 2.11 and 2.12 in Understanding Maple for more about variables.

1.5 Conditional statements

A conditional (`if`) statement causes Maple to test a condition, then carry out different operations depending on whether the condition is true or false. The simplest conditional statement instructs Maple to perform a set of actions only if a condition is true.


```
if condition then
  statement[s]
end if
```


Note that this is a single statement (which contains other statements). The line breaks are produced by pressing `shift+return`, and the whole structure must be executed together.

Any statement that evaluates to `true` or `false` can be used in an `if` statement.

```
cold_today := true :

if cold_today then
  "Wear your hat!" :
end if
                    "Wear your hat!"
```

In cases where a conditional contains a sequence of statements (as opposed to a single statement), these must be separated by colons or semicolons. Otherwise Maple has no way to know where one statement ends and the next begins.

```
a := 1 :
b := a :

increase_vars := true :

if increase_va
  a := a + 1 :
  b := b + 2 :
end if
```


colon on this line is crucial!

Boolean operators including `and`, `not` and `or` can be used to construct more complex conditions.


```
if a > 0 and frac( a ) = 0 then
  "a is a positive integer." :
end if
```


Here, the message is displayed if a is a positive integer. Note the `frac` command, which returns the fractional part of a number (i.e. the digits after the decimal point). Using `else` instructs Maple to take different actions (rather than none at all) if the condition is false.


```
if condition then
  statement[s]
else
  alternative statement[s]
end if
```


Finally, we can check additional conditions if the first turns out to be false.

```
if condition then

  statement[s] # Execute if condition is true

elif second condition then

  alternative statement[s] # Execute if first condition is false,
                           # but second condition is true.
else
  more alternative statement[s] # Execute if all above conditions
                                # are false.
end if
```

In the next example, the message `"a is negative"` will only be printed if both conditions ($a > 0$ and $a = 0$ are false).


```
if a > 0 then
  "a is positive"
elif a = 0 then
  "a is zero"
else
  "a is negative"
end if
```

See section 7.1 in Understanding Maple for more about conditional statements.

1.6 do loops

A do loop causes Maple to repeatedly execute the same statements. The next example causes Maple to display the approximate value of π five times.


```
from 1 to 5 do
  evalf( Pi ) :
end do
```



```
                    3.141592654
                    3.141592654
                    3.141592654
                    3.141592654
                    3.141592654
```

Often we need to use the step number itself during the iteration. This is achieved using a loop with an index variable.

```
for index from start to finish do
  statement[s]
end do
```

Here, *index*, *start*, and *finish* are integers. Initially, *index* is set to equal *start*. After each iteration, *index* is increased by 1. If the result exceeds finish, the loop terminates. Otherwise another iteration is performed. If *start* exceeds *finish*, the statements inside the do loop are not executed at all. We can compute $10!$ as follows.

```
p := 1 :

for j from 2 to 10 do
  p := p * j :  # Updates the value of p
end do :

p
```


As another example, we can evaluate the sum $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ as follows.

```
n := 100 : # (Or any other natural number)
s := 0   :

for j from 1 to n do
  s := s + evalf( 1 / j ) :
end do :

s
```


$$5.187377520$$

Increments other than 1 are possible.

```
for index from start by increment to finish do
  statement[s]
end do
```


If *increment* is negative, the loop will terminate when *index* reaches a value that is smaller than *finish*, and will not execute at all if *finish* exceeds *start*.

```
for j from 5 by -1 to 1 do
  j ;
end do
```

$$5$$
$$4$$
$$3$$
$$2$$
$$1$$

To increment a variable through noninteger values, calculate a step size before the loop starts. In the next example, x varies from 0 to 3 in steps of size $3/7$.

```
nsteps := 7 :  # Number of steps
dx     := evalf( 3 / nsteps ) :  # Step size
```

```
for j from 0 to nsteps do
  x := j * dx
end do
```


$$x := 0.$$
$$0.4285714286$$
$$0.8571428572$$
$$= 1.285714286$$
$$= 1.714285714$$
$$= 2.142857143$$
$$x := 2.571428572$$
$$x := 3.000000000$$

Using a floating point (decimal) index can lead to a disastrous situation in which one too few steps is performed because the last value for x exceeds the upper limit due to rounding error.


```
dx := evalf( 3 / nsteps ) :  # Step size

for x from 0 to 3 by dx do
  x ;
end do
```


$$x := 0.$$
$$x := 0.4285714286$$
$$x := 0.8571428572$$
$$x := 1.285714286$$
$$x := 1.714285714$$
$$x := 2.142857144$$
$$x := 2.571428573$$


```
x
```

$$3.000000002$$

Never use a decimal as a do loop index.

Remark: In Maple, it is possible to use an exact fraction as a loop index (e.g. try the above example without `evalf`). However, very few other programming languages allow this, so we will always use integers.

1.7 Break statements

Sometimes it is not possible to predict how many iterations will be needed for a particular task. For these cases we can use a do loop with no final value for the index (or no index at all). In these cases the loop must be terminated by a break statement.

```
do

   statement[s]

   if condition then
     break :
   end if :

end do
```


Given X, we can find the smallest natural number n such that $n! > X$ as follows.

```
X := 100.0 : # (Or any other number)
f := 1 :

for n from 1 do

   f := f * n :

   if f > X then
     break :
   end if :

end do :

n
```

5

This structure is only appropriate when we are absolutely certain that the condition for the break statement will be met at some point, otherwise Maple will enter an infinite loop. If we can't be certain, we can impose a (large) maximum number of iterations, and check whether the loop has been terminated by the break statement (if this is not the case, the increment will be increased one last time before the loop terminates).

```
max_its := 1000000 : # (Or some other large number.)
for j from 1 by 1 to max_its do
  statement[s]
  if condition
    break :
  end if :
end do :
if j > max_its then
  error "Loop did not break" :
end if :
```


Note the `error` command, which reports that something has gone wrong, and stops execution.

See section 7.2 in Understanding Maple for more about do loops.

1.8 Summing series

Approximately summing a convergent infinite series is a very common application of a do loop with a break statement.

Example

Consider the sum

$$S = \sum_{j=1}^{\infty} \frac{\ln(1+j)}{(2+j)^8}.$$

We can use a basic do loop to calculate partial (i.e. finite) sums.

```
S := 0 :
for j from 1 to 100 do
  S := S + evalf( ln( 1 + j ) / ( 2 + j )^8 ) :
end do :
S
```

0.0001274382682

However, it is difficult to determine the accuracy of this result. Rather than choosing an arbitrary upper limit, we can keep summing until the terms in the series are small.

```
S       := 0 :
epsilon := 0.5
for j from 1 d

  t := evalf( ln( 1 + j ) / ( 2 + j )^8 ) :

  if t < epsilon* S then
    break :
  end if :

  S := S + t :

end do :
S
```


0.0001274382682

Remarks:

- Note that we compare the term t to the current partial sum S, multiplied by ϵ, which is small. That is, we stop summing when the next term is small relative to the result. See section 2 for more about relative errors.
- The significance of the value 0.5×10^{-10} is also discussed in section 2.
- Usually we have to take absolute values (i.e. compare $|t|$ to $\epsilon|S|$), but in this example all terms are positive.
- When we stop the summation, we discard the **tail** of the series, not just one term. In other words, if we use

$$S_N = \sum_{j=1}^{N} t_j,$$

as an approximation to

$$S = \sum_{j=1}^{\infty} t_j,$$

then we are assuming that the magnitude of the tail

$$S - S_N = \sum_{j=N+1}^{\infty} t_j$$

is small. Hov ng that one term is small does not guarantee this. The pr valid if

$$|t_{j+1}| < |t_j| \quad \text{for} \quad j = N, N+1, \ldots$$

and in practice one can get away with $|t_{j+1}| < 0.9|t_j|$ (see problem 1.5).

- If the convergence of the series relies on a power of the index in the denominator then it may be difficult to obtain accurate results if the power is too low (here it's 8, so there is no such issue).

1.9 Arrays

Often it is convenient to store data using indexed variables $A_1, A_2, A_3, \ldots$ rather than variables $A, B, C, \ldots$ etc. This can be achieved by using an **array**, which is similar to a vector or matrix, but can have up to 63 dimensions.

```
A := Array( 1..5 ) ;  # 1D array
```

$$[0\ 0\ 0\ 0\ 0]$$

```
B := Array( 1..2 , 1..3 ) ;  # 2D array
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Note the upper case 'A', in `Array`, which is important. The `array` command (with a lower case 'a') has been deprecated, and should not be used in new worksheets.

By default, the indices for an array start from 1, but you can choose other values, for example

```
A := Array( 0..6 )
```

creates an array with seven entries, numbered from 0 to 6. Note that this has a strange effect on the display; see §1.9.4 for more details.

Arrays can also be generated from lists of initial values.

```
A := Array( [ 7 , 9 , 6 , 5 ] )
```

$$[7\ 9\ 6\ 5]$$

The individual elements in an array are accessed using an index in square brackets.

```
A[2]
```

$$9$$

```
A[3] := 27 :
A[4] := -1 :
A
```

$$[7\ 9\ 27\ -1]$$

Attempting to access an element that is out of range results in an error.

```
A[6]
```

```
Error, Array index out of range
```


More than one element can be accessed using a range.

```
A[1..3]
```


$$[7\ 9\ 27]$$

```
A[1..]
```


$$[7, 9\ 27\ -1]$$


```
A[..3]
```

$$[7\ 9\ 27]$$

A sequence of indices (e.g. `1,2` or `4,1,7`) is used to access the entries in an array with more than one dimension.

```
M := Array( 1..2 , 1..2 )
```

$$M := \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

```
M[1,1] := 5 :
M[1,2] := 3 :
M
```

$$\begin{bmatrix} 5 & 3 \\ 0 & 0 \end{bmatrix}$$

	Dimensions	Lower bound	Notes
Array	arbitrary	arbitrary	
list	1	1	Changing entries is inefficient.
Matrix	2	1	Required by some linear algebra functions.
Vector	1	1	Required by some linear algebra functions.

Rectangular data structures in Maple.

1.9.1 Other rectangular data structures

Aside from arrays, Maple also has lists, vectors and matrices. The differences between these are summarised in table 1. It is sometimes necessary to convert arrays to vectors or matrices, even though the conversions don't really do anything.


```
A := Array( [ [ 1 , 2 ] , [ 3 , -1 ] ] )
```

$$A := \begin{bmatrix} 1 & 2 \\ 3 & -1 \end{bmatrix}$$


```
with( LinearAlgebra ):
MatrixInverse( A )
Error (in MatrixInverse) ...
```



```
MatrixInverse( convert( A , Matrix ) ) ;
                    # or MatrixInverse( Matrix( A ) )
```

$$A := \begin{bmatrix} \frac{1}{7} & \frac{2}{7} \\ \frac{3}{7} & -\frac{1}{7} \end{bmatrix}$$

Bizarrely, some other linear algebra functions, such as `Trace` and `Determinant` work with both arrays and matrices.

1.9.2 Bounds

The `lowerbound` and `upperbound` functions can be used to enquire about the limits for the indices, so you don't need to keep track of them manually. If A is a one-dimensional structure (list, vector, or 1D array) then

```
upperbound( A )
```

returns the upper bound for the index. For an array with more than one dimension or a matrix,


```
upperbound( A , n )
```

returns the upper b dimension. Alternatively,

```
[ upperbound(
```

returns a list in w ement is the upper bound of A in the nth dimension (it's pr void using this version). In all of the above examples, upperbo ced with `lowerbound`, but remember that the lower bound for a vector, matrix or list is always 1.


```
A := Array( -1..1, 1..2 ) :
lowerbound( A , 1 )
```

-1


```
upperbound( A , 2 )
```

2


```
[ lowerbound( A ) ]
```


$[-1, 1]$

```
[ upperbound( A ) ]
```


$[1, 2]$

1.9.3 Example

We can store the first N Fibonacci numbers as follows.

```
N    := 20 : # (Or any other natural number)
f    := Array( 1..N ) :  # Note the capital 'A'.
f[1] := 1 :
f[2] := 1 :

for j from 3 by 1 to N do
  f[j] := f[j-2] + f[j-1] :
end do :

f[19]
```

4181

1.9.4 Displaying arrays

There are three things that can prevent Maple from displaying an array in a convenient form.

(i) *The array ha* *dimensions.*

There is noth bout this, since the screen has only two.

(ii) *One or more* *es not start from 1.*

In this case, convert the array to a matrix or vector to display it.

```
A := Array( -2 .. 2 ) :
A[1] := 57 :
A[2] := Pi :
Vector( A )
```


$$\begin{bmatrix} 0 & 0 & 0 & 57 & \pi \end{bmatrix}$$

```
B := Array( -1 .. 1, -1 .. 1 ) :
Matrix( B )
```


$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

(iii) *The array is too large.*

The display will truncate arrays (and also matrices and vectors) whose size in any dimension is greater than the parameter `rtablesize`, the default value of which is 10. You can change `rtablesize` using the `interface` command.

```
interface( rtablesize = 20 ) :
```

1.9.5 Copying arrays

Assigning an array to a new name does not create a copy. Instead it leads to a confusing situation in which the same data is accessible through more than one name.

```
A := Array( [ 1 , 2 ] ) :
B := A
```

$$B := \begin{bmatrix} 1 & 2 \end{bmatrix}$$

```
B[2] := -27 :
B
```

$$\begin{bmatrix} 1 & -27 \end{bmatrix}$$

```
A
```

$$\begin{bmatrix} 1 & -27 \end{bmatrix}$$

To make a copy, us mand.

```
A := Array( [ 1 , 2 ] ) :
B := copy( A )
```

$$B := \begin{bmatrix} 1 & 2 \end{bmatrix}$$

```
B[2] := -27 :
B
```

$$\begin{bmatrix} 1 & -27 \end{bmatrix}$$

```
A
```

$$\begin{bmatrix} 1 & 2 \end{bmatrix}$$

See section 7.5 in Understanding Maple for more about arrays.

1.10 Tables

Under some circumstances, we need to store a sequence of values without knowing in advance how many there are. A table can be used for this purpose. Unlike an array there is no need to declare bounds for a table; we can just keep adding entries as necessary.

```
t := table() :
t[1] := 5  :
t[2] := 72 :
t[3] := 44 :
```

Unfortunately, there is no shorthand way to retrieve multiple values from a table (`t[1..3]` won't work). Instead, we have to use the `seq` command for this.

```
seq( t[j] , j = 1 .. 3 )
```

$$5, 72, 44$$

Often the most useful thing to do is extract the entries and put them in an array.

```
B := Array( [ seq( t[j] , j = 1 .. 3 ) ] )
```

$$= [5,\ 72,\ 44]$$

Note that using in rackets without stating the type of object you want will produce

```
A[1] := 55 :
A[2] := 68 :
Describe( A )
```

```
A::table = table([(1)=55,(2)=68])
```


Expecting an array in such circumstances is a common mistake.


```
restart :
A[1] := 55 :
A[2] := 68 :
upperbound( A, 1 ) :
```



```
Error, invalid input: ...
```


Don't forget to declare arrays with the `Array` command.

1.10.1 Example

Suppose we wish to store a sequence of random numbers each of which lies between 0 and 1, stopping the first time we encounter a value greater than 0.95. There is no way to determine the length of the sequence in advance, so we can't use an array. Instead, we use a table.

```
# Initialise random number generator
randomize() :

# Create a random number generator function
f := rand( 0.0 .. 1.0 ) :
t := table() :

for j from 1 do

  #Get a random number, add it to the table
  t[j] := f() :
```

```
  if t[j] > 0.5 then
    break :
  end if :

end do :

seq( t[p] , p = 1 .. 3 );
```


$$0.8802371600, 0.9546663364$$

1.11 Procedures

Simple mathematical functions can be defined using arrow notation (- followed by >).


```
f := x -> 2 * x :

f( 1 );
```


$$2$$

```
f( k );
```


$$2k$$

```
g := ( x , y ) -> sin( x + y ) :

g( 3.1 , 1.2 );
```


$$-0.9161659367$$

Note that multiple variables before the arrow must be enclosed in brackets.

A **procedure** is similar to to a function, but it can use any Maple code to obtain its results (including do loops and conditional statements).

1.11.1 Example

This simple procedure implements the function $f(x) = 2x$.

```
f := proc( x )

  return 2 * x :

end proc :
```

```
f( -27 )
```

-54

```
f( a )
```

1.11.2 Local and [illegible]s

Procedures can ha[illegible] internal variables, which cannot be accessed from elsewhere in the worksheet. These are called **local** variables. Variables elsewhere in the worksheet with the same name are separate entities.

```
my_proc := proc()
  local a :
  a := 1 :
  return :
end proc :
a := Pi :
my_proc() :  # Doesn't change a
a
```

π

Here, the value of a is not changed, even though a local variable called a is used inside the procedure. The idea behind this is that the inner workings of procedures can be 'hidden' from the rest of the worksheet. It means we don't need to keep track of which names are used inside procedures, which would be very difficult (any annoying) in large projects.

Variables that are not local are **global**. These can be accessed from anywhere in the worksheet. If we change `local` to `global` in the above example and execute it again, the value of a outside the procedure is now set to 1. This type of side effect can easily lead to errors, especially in large projects where it is difficult to keep track of which names have been used in which places. In general, a procedure should not access or (worse) change global data unless this is **absolutely necessary**.

1.11.3 Parameter sequences and return statements

We will use the convention that the parameters in brackets after `proc` are the input for the procedure, and that the results are specified by a return statement. We won't try to cl eters from within the procedure (though this is possible in some t example shows a procedure that solves the quadratic equation

$$+ bx + c = 0.$$

The coefficients a, input for the procedure and the two roots r_1 and r_2 are the output.


```
solve_quadratic := proc( a , b , c )
#Solve the equation a * x^2 + b * x + c = 0

  local d , r1 , r2 :

  d  := sqrt( b^2 - 4 * a * c ) :

  r1 := evalf( ( -b + d ) / ( 2 * a ) ) :
  r2 := evalf( ( -b - d ) / ( 2 * a ) ) :

  return r1 , r2 :

end proc :

# Test
solve_quadratic( 1 , -1 -6 )
                    3, -2
```


Omitting the return statement causes the procedure to return the result of the last calculation. This is an unhelpful feature which can easily lead to mistakes (here only r_2 would be returned if the return statement was missing). As a general rule, it is best to always include a return statement, even in cases where this is not strictly necessary. Note that any number of return statements can occur anywhere in a procedure. If processing reaches a return statement, the procedure is terminated immediately. This is useful for dealing with cases where certain situations should trigger an immediate return, but processing should otherwise continue.

In most cases, procedures are not executed directly by a human — they are executed automatically by other parts of the worksheet. Therefore it is important to check that the input is valid. For the quadratic solver, we could check that a, b and c are numbers by changing the first line to the following.

```
solve_quadratic := proc( a :: numeric , b :: numeric ,
                         c :: numeric ) :
```


We can also perform checks inside the procedure; thus we might ensure that $a \neq 0$ by inserting the following before the local declarations.

```
if a = 0 then
  error "Leading coefficient should not be zero." :
end if :
```

With these modifications, the statements

```
solve_quadratic( Dr , Ian , Thompson ) :
solve_quadratic( 1, 2, 1 ) :
```

result in an error. Maple has many different types that can be used with `::`. Execute `?type` and scroll down for a list. The most useful for checking procedure input are `boolean` (true or false), `integer`, `posint` (positive integer), `nonnegint` (nonnegative integer), `numeric` (real) and `procedure`.

1.11.4 Writing a procedure

Before starting to write a program, break the problem up into parts, and ask yourself whether it makes sense to write each part as a separate procedure. Follow these steps to start writing a procedure.

(i) Think of a sensible name, and start with the `proc` and `end proc` statements.

```
my_proc  := proc()
end proc :
```

By putting the `end proc` in now, rather than waiting until the procedure is finished, we keep the code in an executable state, so we can test parts of it along the way.

(ii) Insert a comment to briefly describe what the procedure does.

```
my_proc := proc()
#Does something really clever


end proc :
```

(iii) Determine the input that the procedure needs, and use the type operator `::` to prevent acceptance of invalid data types.

```
my_proc := proc( n :: integer , m :: integer )
#Does something really clever

end proc :
```

(iv) Use conditio[illegible]nd the `error` command to perform any other checks on th[illegible] raise exceptions if anything is wrong.

```
my_proc :=[illegible]integer , m :: integer )
#Does something really clever

  if n = m then
    error "Doesn't work if n = m" :
  end if :

end proc :
```

(v) Only now should you begin the hard work of coding the actual machinery of the procedure.

```
my_proc := proc( n :: integer , m :: integer )
#Does something really clever

  local j , p :

  if n = m then
    error "Doesn't work if n = m" :
  end if :

  for j from 1 by 1 to n do
    for p from 1 by 1 to m do

      statement[s]

    end do :
  end do :

  return whatever :

end proc :
```

1.11.5 Example: summing a Taylor series

Consider the function defined by the series

$$= \sum_{j=0}^{\infty} \frac{(-x)^j}{(j+\pi)\, j!}$$

We can sum this se as from section 1.8. There is only one input to the procedure, whi value x, so we start with the following.

```
f := proc( x :: numeric )
#Sums the series (-x)^j / ( ( j + Pi ) * j! ) , j = 0, 1, ...

end proc :
```


There are no other checks to do on the input. We will need local variables j, t and s to represent the summation index, the current term and the partial sum, respectively.


```
f := proc( x :: numeric )
#Sums the series (-x)^j / ( ( j + Pi ) * j! ) , j = 0, 1, ...

  local j , s , t :

  for j from 0 do

  end do :

end proc :
```


If we calculate each term from scratch, the procedure will be as follows.

```
f := proc( x :: numeric )
#Sums the series (-x)^j / ( ( j + Pi ) * j! ) , j = 0, 1, ...

  local j , s , t :

  s := 0 :

  for j from 0 do

    t := evalf( ( -x )^j / ( ( j + Pi ) * j! ) ) :

    if abs( t ) < 0.5 * 10^(-10) * abs( s ) then
```

```
    break :
  end if :
  s := s + t :
end do :
return s :
end proc :
```


However, we can increase efficiency by noting that if

$$r_j = (-x)^j/j!$$

then

$$r_{j+1} = -xr_j/(j+1),$$

so part of the summand can be calculated recursively.


```
f := proc( x :: numeric )
#Sums the series (-x)^j / ( ( j + Pi ) * j! ) , j = 0, 1, ...
```



```
  local s , j , t , r :

  s := 0 :
  r := 1 :
```



```
  for j from 0 do

    t := evalf( r / ( j + Pi ) ) :

    if abs( t ) < 0.5 * 10^(-10) * abs( s ) then
      break :
    end if :

    s := s + t :
    r := -x * r / ( j + 1 ) :

  end do :

  return s :
```

```
end proc :

#Test
f( 1.2 )

f( -4 )

f( 0 )
```


Remarks:

- One should always test a function of this type at $x = 0$. In some problems one should treat this as a special case, i.e. use something like


```
if x = 0 then
  return evalf( 1 / Pi ) :
end if :
```


However, if $x = 0$ here the first term always evaluates to $1/\pi$ and then r is updated to 0, so the loop will break on the second term.

- Numerically evaluating a sum of the form

$$S = \sum_{j=0}^{\infty} c_j (x-a)^j$$

works best when $|x - a| < 1$, even if the series converges for all x. Many functions have alternative representations that can be used if the convergence of the Taylor series is too slow.

1.11.6 More examples

See the following resources on Canvas:

- Searching Arrays (video)
- Sorting Arrays (video)
- `search_array_and_sort_array.mw`
- Calculating Catalan numbers (video)
- `calculate_catalan.mw`

See chapter 8 of Understanding Maple for more about procedures.

1.12 Eliminating errors from a program

Programs are rarely written correctly at the first attempt. The process of removing errors is called **debugging**.

If Maple reports an ally due to a syntax problem such as a missing colon, semicolon or se are easy to find; the set of possible locations for the error can b by deactivating sections of code using # and executing again.

A more troublesome situation arises if a program runs but produces incorrect results. In this case, it usually helps to find out what is going on inside loops, procedures, etc. It may be possible to do this by increasing the `printlevel` parameter (default value 1). If a statement doesn't produce the output you expect, make sure it is not terminated with a colon, and then try increasing `printlevel` by 5, and executing again. Exactly how this parameter works is described on the relevant help page `?printlevel`; see also §7.3 of Understanding Maple. Remember:

- Colons and semicolons inside do loops and conditionals have no effect on output. Only the terminating character following the outermost `end do` or `end if` matters in this respect.
- Whether a procedure produces output depends on the terminating character after the procedure call. On the other hand, `end proc` should always be terminated by a colon; you won't gain anything useful by omitting this (or using a semicolon):

```
printlevel := 10 :
sin( 2.5 ) ;  # This is a procedure call.
```

Often, increasing `printlevel` will cause Maple to output too much material. An alternative is to display the values of relevant variables using the `print` command. For example

```
p := 1 :

for j from 2 to 10 do
  p := p * j :  #Updates the value of p
  print( j , p ) :
end do :
```

The integers j and p are output at each step, regardless of the current value of `printlevel`. The more sophisticated command `printf` allows you to specify details of the output format, such as the number of character columns to use and the number of decimal places to show.

```
p := 1 :

for j from 2 to 10 do
  p := p * j :  #Updates the value of p
  printf( "j = %0d\n" , j , p ) :
end do :
```


See `printf_demo` [illegible] of Understanding Maple for more details.

1.13 Style guidelines

The same program can be written in many different ways. Our main concerns are to maximise efficiency (primarily in processor time, but also in memory usage). However, it is also important to write programs that are logically structured and understandable, because this will help you to avoid making mistakes. In addition, parts of a program will often be useful again later, saving time and effort, but trying to reuse a badly written program long after its creation is difficult and error prone.

Format your code according to the following rules:

- Use explanatory comments (starting with a '#' symbol) for lines or blocks of code whose purpose is not immediately obvious. Maple will ignore these.
- Use extra space in your code; i.e.

  ```
  h  := 0.5*(a+b):
  fh := evalf( f( h ) ) :
  ```

  rather than

  ```
  h:=0.5*(a+b):
  fh:=evalf(f(h)):
  ```

  Large blocks of unspaced code can be difficult to read.
- Use blank lines to separate different tasks within a program.
- Indent code inside procedures, do loops and if statements by two spaces.

  ```
  my_proc := proc( n :: integer , m :: integer )
  #Does something really clever

    local j , p :

    for j from 1 by 1 to n do
      for p from 1 by 1 to m do
  ```

```
        statement[s]
    end do
end do ;

return

end proc
```


This makes it easier to see the overall structure of the program.

- Do not write long lines (more than about 100 characters) of code.
- No more than one assignment operator (`:=`) should appear on a line.
- The following elements should always appear on a line of their own:
  - the first line of a procedure (containing `proc`),
  - `end proc`,
  - `return` statement,
  - `for ... do`,
  - `end do`,
  - `if ... then`,
  - `end if`.
- Do not abbreviate `end if`, `end do` or `end proc` to `end`, because this makes it difficult to see what statement is being terminated. If you must use abbreviations, you can use `fi` and `od` in place of `end if` and `end if`, respectively.