Programming in Maple: Some Notes

Michael Monagan
Department of Mathematics
Simon Fraser University
Fall 2013.

These notes will work in any version of Maple from Maple 6.

Lists

The simplest data structure in Maple is a list. The elements of a list may be of any type. To create a list of values enclose them in square brackets [,]. Lists may be nested.

```
> E := []; # the empty list
```

$$E := [\,]$$

```
> L := [1,2,-3,4,1];
```

$$L := [1, 2, -3, 4, 1]$$

```
> M := [[1,2,3],[x-1,x^2-1,x^3-1]];
```

$$M := [[1, 2, 3], [x-1, x^2-1, x^3-1]]$$

To count the number of entries in a list use nops(L) command.

```
> nops(L);
```

$$5$$

To access the i'th element of a list (counting from 1) use a subscript.

```
> L[3];
```

$$-3$$

```
> M[2];
```

$$[x-1, x^2-1, x^3-1]$$

```
> M[2][2];
```

$$x^2-1$$

A negative subscript counts from the end. So here is the last element.

```
> L[-1];
```

$$1$$

Use the following to extract a sublist

```
> L[2..3];
```

$$[2, -3]$$

```
> L[2..-1];
```

$$[2, -3, 4, 1]$$

To append (prepend) elements to a list use the following.

```
> op(L);
```

$$1, 2, -3, 4, 1$$

```
> L := [op(L),5];
```

$$L := [1, 2, -3, 4, 1, 5]$$

To test if an element is in a list use

```
> member(2,L);
```

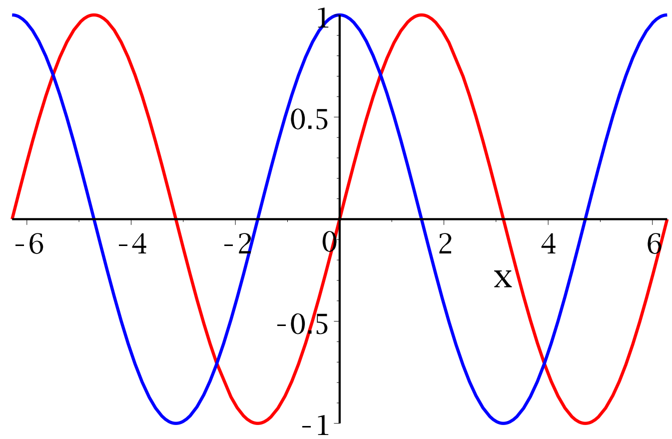
$$true$$

Lists are used by many Maple commands. For example, many of the graphing commands takes lists of points or lists of functions as inputs. For example, to plot sin(x) and cos(x) on the same graph do

```
> L := [sin(x),cos(x)];
```

$$L := [\sin(x), \cos(x)]$$

```
> plot( L, x=-2*Pi..2*Pi, color=[red,blue] );
```



Although you can assign to an entry of a list (as if it were an array) if the list has less than 100 elements, do not do this. It creates a copy of the entire list. So it's not efficient. Use Arrays .

```
> L[2] := 10;
```

$$L_2 := 10$$

```
> L;
```

$$[\sin(x), 10]$$

Sets

Maple also supports sets. Maple uses squiggley brackets { } for sets. For example {1,3,5}. Sets differ from lists in that the only one copy of each element is kept and the elements are sorted. Otherwise many of the commands that work for lists, such as subscripts, also work exactly the same way for sets. Here are some examples.

```
> S := {1,5,3,1};
```

$$S := \{1, 3, 5\}$$

```
> T := {2,3,4};
```

$$T := \{2, 3, 4\}$$

```
> S[2];
```

$$3$$

The number of elements of a set |S| is given by

```
> nops(S);
```

$$3$$

To test if an element x is in a set use member

```
> member(5,S);
```

$$true$$

The set operations union, intersection and set difference are

```
> S union T;
```

$$\{1, 2, 3, 4, 5\}$$

```
> S intersect T;
```

$$\{3\}$$

```
> S minus T;
```

$$\{1, 5\}$$

The empty set

```
> phi := {};
```

$$\phi := \{\ \}$$

```
> phi union S;
```

$$\{1, 3, 5\}$$

You can insert a new element in a set in two ways, either using union or, like lists, using op

```
> S union {9};
```

$$\{1, 3, 5, 9\}$$

```
> {op(S),9};
```

$$\{1, 3, 5, 9\}$$

The elements of a set may be of any type. Here is a set of equations

```
> eqns := { x+2*y=1, 3*x-z=2, x+y+z=0 };
```

$$eqns := \{x + 2\,y = 1, 3\,x - z = 2, x + y + z = 0\}$$

Many Maple commands take sets of objects as input. For example, the solve command takes a set (or list) of equations as input and a set of unknowns to solve for and outputs the solution as a set (list).

```
> sol := solve( eqns, {x,y,z} );
```

$$sol := \left\{x = \frac{3}{7}, y = \frac{2}{7}, z = -\frac{5}{7}\right\}$$

If statements

```
> restart;
```

To execute a command in Maple conditionally use the if command which has either of the following forms

if <condition> **then** <statements> **else** <statements> **fi**

or just

if <condition> **then** <statements> **fi**

For example,

```
> x := 2;
```

$$x := 2$$

```
> if x>1 then print(good); else print(bad); fi;
```

$$good$$

The if statement can be nested. For example

```
> if x>1 then if x>2 then print("x > 2"); else print("x > 1");
  fi; else print("x < 2"); fi;
```

$$\text{"x > 1"}$$

Although you can put it all on one line like that it's best to split it accross multiple lines. Use SHIFT-ENTER to get a new line.

```
> if x>1 then
     if x>2 then print("x > 2");
     else print("x > 1");
     fi;
  else
     print("x < 2");
  fi;
```

$$\text{"x > 1"}$$

The boolean operators in Maple are and, or, and not. The relational operators in Maple are =, >, <, >=, <=, and <> for not equals.

```
> x := 2;
  if x >= 1 and x <= 3 then x := x+1; fi;
```

$$x := 2$$

$$x := 3$$

Loops.

```
> restart;
```

To execute one or more statementsin a loop use the **for** command. It has the following form

for <variable> **from** <start> **to** <end> **do** <statements> **od**

```
> for i from 1 to 5 do i^2; od;
```

$$1$$
$$4$$
$$9$$
$$16$$
$$25$$

```
> for i from 1 to 5 do i; isprime(i); od;
```

$$1$$
$$\mathit{false}$$
$$2$$
$$\mathit{true}$$
$$3$$
$$\mathit{true}$$
$$4$$
$$\mathit{false}$$
$$5$$
$$\mathit{true}$$

To execute some statements while a condition is true use the while loop. It has the syntax

while <condition> **do** <statements> **od**

```
> i := 1;
  while i <= 5 do i^2; i := i+1; od;
```

$$i := 1$$
$$1$$
$$i := 2$$
$$4$$
$$i := 3$$
$$9$$
$$i := 4$$

$$16$$

$$i := 5$$

$$25$$

$$i := 6$$

There is quite a bit of output there. You can see each assignment and each square displayed. To suppress the output of a loop (or any Maple statement) use : instead of ; But then we won't see any output. We can override the : by using print(i^2) to see the squares like this. Notice I put a : on the i := 1: as well.

```
> i := 1:
  while i<=5 do print(i^2); i := i+1; od:
```

$$1$$

$$4$$

$$9$$

$$16$$

$$25$$

As a second example, we find the first prime bigger than 2. We only consider the odd integers.

```
> p := 33;
  while not isprime(p) do p := p+2; od:
```

$$p := 33$$

```
> p;
```

$$37$$

In a Maple for loop, you can count by a different value using the **by** clause. The general form is this

for <variable> **from** <start> **by** <increment> **to** <end> **do** <statements> **od**

Also handy is that you can exit a loop using the **break** command. Here we find the first prime bigger than 31 using a **for** loop.

```
> for p from 33 by 2 do
      print(testing(p));
      if isprime(p) then break; fi;
  od:
  p;
```

$$testing(33)$$

$$testing(35)$$

$$testing(37)$$

$$37$$

Notice that I didn't specify a **to** clause . All the clauses are optional. The defaults are

from 1
to infinity
by 1

As a final example here is a loop that generates some polynomials

```
> for n to 6 do n = factor(x^n-1) od;
```

$$1 = x - 1$$

$$2 = (x - 1)(x + 1)$$

$$3 = (x - 1)(x^2 + x + 1)$$

$$4 = (x - 1)(x + 1)(x^2 + 1)$$

$$5 = (x - 1)(x^4 + x^3 + x^2 + x + 1)$$

$$6 = (x - 1)(x + 1)(x^2 + x + 1)(x^2 - x + 1)$$

Three other useful looping constructs are the **map** command and the **seq** command and the **add** command. The examples show what the commands do.

```
> L := [1,2,3,4,5];
```

$$L := [1, 2, 3, 4, 5]$$

```
> map( f, L );
```

$$[f(1), f(2), f(3), f(4), f(5)]$$

```
> map( isprime, L );
```

$$[\mathit{false}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{true}]$$

```
> seq( i^2, i=1..5 );
```

$$1, 4, 9, 16, 25$$

```
> seq( L[i], i=1..nops(L) );
```

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

```
> seq( isprime(L[i]), i=1..nops(L) );
```

$$\mathit{false}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{true}$$

```
> seq( L[i]*x^(i-1), i=1..nops(L) );
```

$$1, 2\,x, 3\,x^2, 4\,x^3, 5\,x^4$$

```
> L := [seq( n^2, n=L )];
```

$$L := [1, 4, 9, 16, 25]$$

```
> add( f(i), i=1..5 );
```

$$f(1) + f(2) + f(3) + f(4) + f(5)$$

```
> add( i^2, i=1..5 );
```

$$55$$

```
> add( x[i], i=0..5 );
```

$$x_0 + x_1 + x_2 + x_3 + x_4 + x_5$$

```
> add( x^i, i=0..5 );
```

$$1 + x + x^2 + x^3 + x^4 + x^5$$

Read the help files for these commands, they are very handy.

```
> ?map
> ?seq
> ?add
```

Maple Functions and Procedures

Maple has a special syntax for inputting a simple function like $f(x) = x^2 + 1$. You may input using the arrow notation in Maple, as follows

```
> f := x -> x^2+1;
```

$$f := x \to x^2 + 1$$

Now you can apply the function to values in the usual notation

```
> f(2);
```

$$5$$

```
> f(0.5);
```

$$1.25$$

```
> f(z);
```

$$z^2 + 1$$

A procedure in Maple takes the form

proc(p1, p2, ...)
local l1, l2, ... ;
global g, g2, ... ;
statement1;
statement2;
....
statementn;
end proc

There may be zero or more parameters, one or more locals, one or more globals and one or more statements in the procedure body.
The local and global statements are optional. Variables in the procedure body that are not explicitly declared as parameters, locals, or globals are declared to be local automatically if assigned to, otherwise they are global. The value returned by the procedure is the value of *statementn*, the last statement in the body of the procedure or the value of an explicit return statement. Type declarations for parameters and local variables need not be explicitly given. Some examples will

help.

```
> f := proc(x) y := x^2; y+1; end proc;
Warning, `y` is implicitly declared local to procedure `f`
```

$$f := \mathbf{proc}(\{x\})\ \ \mathbf{local}\ y;\ y := x\text{^}2;\ y+1\ \ \mathbf{end\ proc}$$

```
> f(2);
```

$$5$$

```
> f(z);
```

$$z^2+1$$

Notice that Maple made the variable y local for us. To avoid the warning, we should declare it local ourselves. Also, you can break a procedure over more than one line - and you should unless it is a simple function.

```
> f := proc(x)
  local y;
      y := x^2;
      y+1;
  end proc;
```

$$f := \mathbf{proc}(\{x\})\ \ \mathbf{local}\ y;\ y := x\text{^}2;\ y+1\ \ \mathbf{end\ proc}$$

This next example searches a list L for the value x. It outputs the position of the first occurrence of x in L and 0 otherwise. The example also uses an explicit return. When **return x** is executed, Maple immediately returns the value of *x* as the result of the procedure. I've also used a Maple comment. Anything following the # character on a line is treated as a comment and ignored by Maple.

```
> position := proc(x,L) local i;
     for i from 1 to nops(L) do
         if L[i]=x then return i fi;
     od;
     0; # meaning x is not in the list
  end proc;
```

$position := \mathbf{proc}(\{x, L\})$

$\quad \mathbf{local}\ i;$

$\quad \mathbf{for}\ i\ \mathbf{to}\ nops(L)\ \ \mathbf{do}\ \ \mathbf{if}\ \ L[i] = x\ \ \mathbf{then}\ \ \mathbf{return}\ \ i\ \ \mathbf{end\ if}\ \ \mathbf{end\ do};\ 0$

$\mathbf{end\ proc}$

```
> position(x,[u,v,w,x,y,z]);
```

$$4$$

```
> position(y,[u,v,w]);
```

$$0$$

This next example is a Maple procedure which returns the next prime bigger than the input. I am also telling Maple that the input parameter n must be an integer. If it's not, an error will be generated. See **?type** for a list of other allowable types.

```
> NextPrime := proc(n::integer)
```

```
  local x;
        x := n+1;
        while not isprime(x) do x := x+1; od;
        x;
  end proc;
```

$$NextPrime := \mathbf{proc}(\{n::integer\})$$

$$\mathbf{local}\ x;$$

$$x := n+1;\ \mathbf{while\ not}\ isprime(x)\ \mathbf{do}\ x := x+1\ \mathbf{end\ do};\ x$$

$$\mathbf{end\ proc}$$

```
> NextPrime(2);
```

$$3$$

```
> NextPrime(1000);
```

$$1009$$

```
> NextPrime(2/3);
Error, invalid input: NextPrime expects its 1st argument, n, to be of type
integer, but received 2/3
```

Now I'm going to redo this example. The first difference is that I'm going to use a : on the end proc: to suprress the output. The second difference is that I'm going to count by 2 (because that's more efficient). So I need to start with the first odd number bigger than *n*.

```
> NextPrime := proc(n::integer)
  local x;
        if irem(n,2)=0 then x := n+1; else x := n+2; fi;
        while not isprime(x) do x := x+2; od;
        x;
  end proc:
> NextPrime(1000);
```

$$1009$$

There is one major difference between Maple and most other programming languages like C and Java. The parameters to a procedure cannot be used like local variables. You cannot assign to parameters. If you try to, you will get an error. Let's redo the NextPrime example where we simply add 1 or 2 to *n* to make it the next odd number then use *n* in the procedure instead of the local variable *x.* You may have wondered why I did that.

```
> NextPrime := proc(n::integer)
        if irem(n,2)=0 then n := n+1; else n := n+2; fi;
        while not isprime(x) do n := n+2; od;
        n;
  end proc:
> NextPrime(1000);
Error, (in NextPrime) illegal use of a formal parameter
```

The error occurs because when Maple executes *n := n+1* it substitutes the parameter 1000 for n and tries to execute 1000 := 1000+1 which doesn't make any sense. Well, that's the way Maple does it. So we need to use a local variable *x* like I did.

Procedures may be nested.
Procedures may be returned and passed freely as parameters.
The simplest debugging tool is to insert print statements in the procedure. For example

```
> NextPrime := proc(n::integer)
  local x;
        if irem(n,2)=0 then x := n+1; else x := n+2; fi;
        while not isprime(x) do print(x); x := x+2; od;
        x;
  end proc:
> NextPrime(1000);
```

$$1001$$

$$1003$$

$$1005$$

$$1007$$

$$1009$$

The next simplest debugging tool is the trace command. All assigment statements are displayed.

```
> trace(NextPrime);
```

$$NextPrime$$

```
> NextPrime(1000);
{--> enter NextPrime, args = 1000
```

$$x := 1001$$

$$1001$$

$$x := 1003$$

$$1003$$

$$x := 1005$$

$$1005$$

$$x := 1007$$

$$1007$$

$$x := 1009$$

$$1009$$

```
<-- exit NextPrime (now at top level) = 1009}
```

$$1009$$

The printf command can be used to print more detailed information in a controlled format. It works just like the printf command in the C language. The main difference is the %a option for printing algebraic objects like polynomials.

But %a works for anything.   E.g.

```
> printf( "A polynomial %a\n", x^2-2*y*x );
A polynomial x^2-2*y*x
```

```
> NextPrime := proc(n::integer)
  local x;
        if irem(n,2)=0 then x := n+1; else x := n+2; fi;
        while not isprime(x) do
            printf("%a is not prime\n",x);
            x := x+2;
        od;
        x;
  end proc:
```

```
> NextPrime(1000);
1001 is not prime
1003 is not prime
1005 is not prime
1007 is not prime
```

$$1009$$

There is more. But this should be enough for the course. See **?proc** if you need more information or more tools.

Subscripted Names and Arrays

Variables may be subscripted. For example, here is a polynomial in x_1, x_2, x_3. You can assign to the subscripts.

```
> restart;
```

```
> f := 1-x[1]*x[2]*x[3];
> x[1] := 3;
```

$$f := 1 - x_1\,x_2\,x_3$$

$$x_1 := 3$$

```
> f;
```

$$1 - 3\,x_2\,x_3$$

There may be more than one subscript and the subscripts may be any value. Arrays are like arrays from computing science. Here is how to create a one-dimensional array A with values indexed from 1 to 5.

```
> A := Array(1..5);
```

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

By default, the entries in the array A are initialized to 0.

```
> A[1] := 3;
```

$$A_1 := 3$$

```
> A[1];
```

$$3$$

```
> for i from 2 to 5 do A[i] := 3*A[i-1] od;
```

$$A_2 := 9$$

$$A_3 := 27$$

$$A_4 := 81$$

$$A_5 := 243$$

Often you will want to convert an Array to a list or a list to an Array. Use the following. For Array to list use

```
> L := convert(A,list);
```

$$L := [3, 9, 27, 81, 243]$$

For list to Array use

```
> A := Array(1..5,L);
```

$$A := \begin{bmatrix} 3 & 9 & 27 & 81 & 243 \end{bmatrix}$$

Oh, they look the same. Let's check

```
> whattype(L);
```

$$list$$

```
> whattype(A);
```

$$Array$$

So what's the difference? In an Array you can change a value in constant time. So when we do

```
> A[3] := 10;
```

$$A_3 := 10$$

It doesn't matter how long the Array is, this takes a fixed amount of time. This is not the case for lists.

The last thing I want to mention is that you should not build up a list of items one at a time. For example, do not do this

```
> L := []:
  for i from 1 to 6 do
      L := [op(L),i^2];
  od;
```

$$L := [1]$$

$$L := [1, 4]$$

$$L := [1, 4, 9]$$

$$L := [1, 4, 9, 16]$$

$$L := [1, 4, 9, 16, 25]$$

$$L := [1, 4, 9, 16, 25, 36]$$

Why not? Because each time you add the next square to the list, Maple makes a copy of all the previous elements. So the amount of space that it uses is 1 + 2 + 3 + 4 + 5 + 6 words. If we keep doing this we will use a quadratic amount of space because the sum of the first n integers is $\frac{n\,(n+1)}{2}$. Instead, use an Array like this and then convert the Array to a list if you want a list.

```
> A := Array(1..6):
  for i from 1 to 6 do A[i] := i^2; od;
  L := convert(A,list);
```

$$A_1 := 1$$

$$A_2 := 4$$

$$A_3 := 9$$

$$A_4 := 16$$

$$A_5 := 25$$

$$A_6 := 36$$

$$L := [1, 4, 9, 16, 25, 36]$$

I'm going to time this (in CPU seconds) for the first n integers so you can see the difference.

```
> n := 5000;
```

$$n := 5000$$

```
> st := time():
  L := []:
  for i to n do L := [op(L),i^2] od:
  time()-st;
```

$$0.145$$

```
> st := time():
  A := Array(1..n):
  for i to n do A[i] := i^2 od:
  L := convert(A,list):
  time()-st;
```

$$0.008$$