Chapter 2

Understanding some Basics of *Maple*

2.1 *Maple* Basics and Elementary Programming

2.1.1 Labels, strings, numbers and constants

Strings are some of the most simple objects in Maple. A string has to be enclosed in double quotation marks.

```
> S:= " This is a string";
> S;
> whattype(S);
```

A **name** is usually just a symbol. In its simplest form a name is a letter followed by zero or more letters, digits, and underscores with lower and upper case letters distinct. Maple has a number of reserved names, like Pi and I, which cannot be used to assign another meaning (than the one defined by the system). A name usually has an expression as its value.

```
> AAbbccc4 := 21;
> diff:=3;
```

We got an error message here since `diff` is a reserved name for the differentiation procedure. Names can include empty spaces, in which case the whole name has to be included in backquotes.

```
> `This is a name`:=2;
```

The command **whattype** tells us the ***Maple*** type of an object. The command `whattype(S)` produced as its output *string*. However, `S` itself is not a string: it is a symbol - the name of a string. The ***value*** of `S` is a string. This example illustrates ***Maple***'s way of dealing with names. Whenever ***Maple*** sees a name, it evaluates it fully. So to ***Maple*** the command `whattype(S)` is the same as the command `whattype("This is a string")`. Therefore the result is *string*.

```
> whattype("This is a string");
```

It is possible to tell Maple not to evaluate all the way by inclosing the symbol in quotes:

```
> whattype(S);whattype('S');
```

Now, this evaluation of names is ***sometimes confusing and may lead to errors.***

```
> b:=1;a:=b;
> b:=2;
> a;
```

In the above we first assigned the value 1 for the symbol ***b***. Then we told Maple that ***a*** = ***b***. At this point ***Maple*** gave the value to the symbol ***a*** that it got by evaluating ***b*** fully. Compare this to the following:

```
> B:=1;A:='B';
> B:=2;'A'=A;
```

The quotes around ***B*** in the assignment `A:='B'` had the effect that the symbol ***A*** was given the value ***B*** rather than the value of ***B***.

You can instruct Maple to clear any previous assignments made to the symbol ***A*** by the command `A:='A'`.

```
> A:='A';
> A;
```

2.1.2 Expressions

Maple represents mathematical expressions as *expression trees*. To find out what the internal representation of a particular expression is, one uses the command `dismantle`. To use it, we first have to load the corresponding library.

```
> ### WARNING: persistent store makes one-argument readlib obsolete
> readlib(dismantle):
> f:=x^2+sin(x^2+1);
> dismantle(f);
```

This internal representation can be pictured as a directed acyclic graph.

There are a number of tools to manipulate expressions.

First one can find out the number of operations in any given expression by the command `nops()`:

```
> nops(f);
```

To find out what these operations are, use the command `op()`. These can be used recursively to go down the directed acyclic graph of an expression.

```
> op(1,f);op(2,f);
> op(0,f);
> nops(op(2,f));
> nops(op(1,op(2,f)));
```

One can change expressions by substituting elements with the command `subsop()`. However, the value of the original expression is not changed unless you explicitly change it.

```
> subsop(1=sin(y),f);
> f;
```

A set in Maple is a sequence enclosed in the brackets {}. Perhaps the most common application of the command `nops()` is to find out the number of elements in a set or a list. For example:

```
> A:={a,b,alpha,beta,1,2};
> nops(A);
```

2.1.3 Sequences, sets, lists and matrices

A sequence is of the following form

```
> Sequence:=expression[1],expression[2],expression[3];
```

A sequence can contain more than three terms. You access elements of a sequence as follows:

```
> Sequence[2];
```

To list all elements of the above sequence, do

```
> for i to 3 do Sequence[i]; od;
```

If you do not know the number of elements of a sequence, you can find out that number by the `nops()` command.

However,

```
> nops(Sequence);
```

does not work because all elements of the sequence in question become arguments of the command `nops()`. You can get around this by making a list out of the sequence:

```
> nops([Sequence]);
```

Then you can write

```
> for i to nops([Sequence]) do Sequence[i]; od;
```

to list the elements of the sequence.

A set is a sequence enclosed in the curly parenthesis {}. For example:

```
> Set := {Sequence};
> whattype(Sequence);whattype(Set);
```

The difference between a ***set*** and a ***list*** is that the members of the list are **ordered** while the members of a set do not need to be ordered:

```
> {alpha,beta,7,6};
> [alpha,beta,7,6];
```

Maple can also deal with tables. For instance:

```
> Table:=table([(R,1)=A1,(R,2)=B1,(P,1)=2]);
```

You can quickly access the data stored in the table. Here dimensions can be arbitrary.

```
> Table[P,1];
```

Tables (and also arrays defined later) have special evaluation rules. To access a table, you must evaluate it.

```
> op(0,Table);op(0,eval(Table));
```

An array is a table with specified dimensions with each dimension integer range.

```
> MyArray:=array(symmetric, 1..3,1..3,[(1,1)=1,(1,2)=2,(1,3)=3]);
```

As before, you can access the elements of an array as follows

```
> MyArray[2,1];
```

Array is always rectangular, while a table need not be such.

```
> whattype(MyArray);whattype(eval(MyArray));
```

To define matrices, one must first load the linear algebra library.

```
> with(linalg):
> A:=matrix(2,2,[[alpha,beta],[gamma,delta]]);
> a:='a':b:='b':
> B:=matrix(2,2,[[a,b],[c,d]]);
```

matrix operations are now (after loading linalg package) known to ***Maple***.

```
> AB:=A&*B;
> evalm(AB);
```

The expression &*() is the generic identity matrix. Observe that &*() takes its dimension from the dimensions of the surrounding matrices (if any).

```
> evalm(A&*&*());
```

Recall that, in general, matrix multiplication is not commutative.

```
> evalm(A&*B);evalm(B&*A);
```

are usually different matrices.

2.1.4 Procedures

So far we have dealt with single ***Maple*** commands. One can write procedures, which are sequences of ***Maple*** commands. Consider, for example, the **Fibonacci numbers**. They are defined recursively as follows

```
> F[0]=0;F[1]=1;F[n]:=F[n-1]+F[n-2];
```

The computation of these numbers can be coded as follows.

```
> Fibonacci := proc(n::nonnegint)
> if n<2 then
> n;
> else
> Fibonacci(n-1)+Fibonacci(n-2);
> fi;
> end;
> Fibonacci(3);
```

```
> Fibonacci(2);
```

Here is a list of some Fibonacci numbers

```
> seq(Fibonacci(i), i=1..20);
```

This computation took some time. We can time the execution of commands and procedures by the **time()** command.

```
> time(Fibonacci(25));
```

The above procedure can be made much more effective by using the **remember** -option. That tells Maple to keep a list of the previous calculations so that they need not be repreated.

```
> FFibonacci := proc(n::nonnegint)
> option remember;
> if n< 2 then
> n;
> else
> FFibonacci(n-1)+FFibonacci(n-2);
> fi;
> end;
> time(FFibonacci(25));
```

2.2 Numerical Operations and their Kin

2.2.1 Numerical Calculations

The basic arithmetic operations of addition, subtraction, multiplication and division are performed in the usual way with Maple. They are are explicit. This means that **2x is different from 2*x. Whenever possible, Maple gives an exact answer and reduces fractions.**

- "a plus b" is entered as `a+b`;
- "a minus b" is entered as `a-b`;
- "a times b" is entered as `a*b`; and
- "a divided by b" is entered as `a/b`. Generally, when a and b are both numbers, a/b is given as a reduced fraction.

Consider the following calculations:

```
> 125+654;
> 6658124-55482234;
> -554*995844215;
> 22361*5545552*994587441;
> 467/35;
> 12315/45;
```

Now, what happens when we do not tell Maple the order of operations? Which operations does Maple perform first?

```
> 3+8*4+16/2;
> ((3+8)*4+16)/2;
> 3+(8*4+16)/2;
```

Exponentiation, "a raised to the power b," is entered as `a^b`. The term $a^{(\frac{n}{m})}$ is entered as `a^(n/m)`. The square root can be called with the command sqrt(a). Usually, these results are returned unevaluated. Sometimes, you can use the command `simplify` to reduce the expression to a nicer, or more expected form. If you need an approximation to the value, use `evalf`.

```
> (-5)^35;
> (-5)^(1/7);
```

When we use evalf to get a numerical approximation, note that Maple returns the "principal root," which is an imaginary number.

```
> evalf((-5)^(1/7));
```

What is the principal root? For any complex number, remember that from DeMoivre's Theorem we have $z^n = r^n\, e^{(i\, n\, \theta)}$ and $z^n = r^n\, (\cos(n\, \theta) + i \sin(n\, \theta))$ so it will follow that for roots $z^{(\frac{1}{n})} = r^{(\frac{1}{n})}\, (\cos(\frac{\theta + 2\, k\, \pi}{n}) + \mathrm{i}(\sin(\frac{\theta + 2\, k\, \pi}{n})))$ and the principal root has -p < q < p and *k=0*. Since for -5, the reference angle is π, then for the seventh root our angle is $\frac{\pi}{7}$.

```
> 8^(2/3);
> simplify(%);
```

Now, we have seen that for odd roots of real numbers, Maple will return a complex number. This is not what we would often expect. For example,

```
> (-27/64)^(2/3);
> simplify(%);
```

This is NOT what I expected. I expected, and wanted, a REAL number. I was expecting the cube root of (-27/64) squared, which should be real! In order to get this result, we need to use the function `surd`. For x real and negative and for n odd, `surd(x^n,n)` returns x. Thus `surd(-8,3)` returns the cube root of -8, or -2.

```
> surd(-27/64,3);
> surd(-27/64,3)^2;
```

2.2.2 Known Constants

```
> constants;
```

Note that Maple has the values built in for π, Euler's constant γ, and a few others. It also has other constants that we may access, such as

```
> exp(1);
> evalf(exp(1),25);
```

Note that e just returns e, it is not stored in exactly the same manner.

```
> e;
> sqrt(-1);
> I^2;
> (1-I)^8;
> (2-3*I)/(3+4*I);
```

2.2.3 Built-in Functions

Maple has a lot of built-in functions - more than we would use in a normal undergraduate course. These include the exponential function `exp(x)`; the natural logarithm `ln(x)`; the absolute value function, `abs(x)`; the trigonometric functions `sin(x)`, `cos(x)`, `tan(x)`, `cot(x)`, `sec(x)` and `csc(x)`; and the inverse trigonometric functions `arcsin(x)`, `arccos(x)`, `arctan(x)`, `arccot(x)`, `arcsec(x)`, and `arccsc(x)`. You can get a complete list of built-in functions by issuing the command `?inifcn`

```
> ?inifcn
> evalf(exp(-8));
```

The absolute value function:

```
> abs(-8);
> abs(34);
> abs(4-4*I);
```

The logarithmic and exponential functions:

```
> ln(exp(1));
> ln(exp(3));
> exp(ln(Pi));
> plot([ln(x),exp(x),x],x=-2..2,-2..2,color=black);
```

The trigonometric functions

```
> cos(Pi/3);
> sin(Pi/6);
> sin(Pi/8);
> tan(3*Pi/4);
> cos(Pi/8);
```

```
> convert(cos(Pi/8),radical);
> plot([cos(x),sec(x)],x=0..2*Pi,-Pi..Pi,color=black);
```