MATH549 Exercise Sheet 5

Deadline for submission: Monday 2nd November

1 Procedures

In this sheet, you'll start to learn how to *program* in Maple. While you can do a great deal within a Maple worksheet just by typing sequences of Maple's own commands, you can go even further by writing your own commands, or *procedures*. Please note that you will not get high marks for the Maple component of your Maple-LATEX project unless you write some relatively complicated procedures.

If you've never done any programming before, then you may find some of the ideas in this sheet difficult. Please feel free to come and see me in my office if you need extra help.

Once again, there are hints for many of the exercises on the module webpage: but you'll learn more if you try to do the exercises without looking at the hints.

1.1 First examples

You already saw a simple version of procedures in sheet 4, where you wrote your own *functions*, along the lines of

```
>f := x -> (x+1)^6 + (x-1)^6;
>H := (f,n) -> coeff(convert(series(f(x),x,n+1),polynom),x,n);
```

With procedures, you can do much more complicated things. (You can also make the same things look simpler: one reason the function H above is difficult to understand is that we had to put it all on a single line.) Let's start by writing the two functions above as procedures. (Remember to type SHIFT-ENTER between lines in a multi-line statement.)

```
>MyFirstProcedure := proc(x)
    (x+1)^6+(x-1)^6;
end proc;
>MyFirstProcedure(-1);
>MyFirstProcedure(a);
>a:=10;
>MyFirstProcedure(a);
```

```
>TaylorCoefficient := proc(f,n)
    # Calculates coefficient of x^n in Taylor series of f(x) about x=0
    local TS, TSTrunc;
    TS:=series(f(x),x,n+1); # Find Taylor series up to the term in x^n
    TSTrunc:=convert(TS,polynom); # Convert series to polynomial
    coeff(TSTrunc,x,n); # return the coefficient of x^n
    end proc;
>TaylorCoefficient(sin,4);
```

>c:=TaylorCoefficient(cos,4);

Note the following points about these procedures:

a) When you type MyFirstProcedure(-1);, the code within the procedure is executed after setting x to -1. x is called the *argument* of the procedure. Similarly, when you type TaylorCoefficient(cos,4);, the code within the procedure is executed after setting f to cos and n to 4. Thus the effect is that of executing the following lines:

>TS:=series(cos(x),x,5); # Find Taylor series up to the term in xⁿ >TSTrunc:=convert(TS,polynom); # Convert series to polynomial >coeff(TSTrunc,x,4); # return the coefficient of xⁿ

Actually, typing TaylorCoefficient(cos,4) is different from typing those lines in two ways. First, you don't see the output from the first two lines of code, only from the last one; and second, after you've run the procedure, Maple knows nothing about TS and TSTrunc (this is the effect of local TS, TSTrunc; see c) below).

- b) The last line before end proc; is what is *returned* by the procedure: that is, its *output*.
- c) The variables TS and TSTrunc are declared as local: this means that they are regarded as completely separate variables from any other variables that might be called TS or TSTrunc in your worksheet. This is very important, since someone might use your procedure TaylorCoefficient without having any idea that it uses a variable called TS internally, and they'd be very confused if their own variable TS started changing its value. To check you understand what local means, try the following:

```
>TS:=egg;
>TaylorCoefficient(cos,4);
>TS;
>TSTrunc;
```

If you don't declare your variables to be local then Maple will assume they are anyway, but will issue you with a warning. No one likes to be warned, so get into the habit of declaring variables local. In fact, you can declare variables global if you want: try changing the local to global in the definition of TaylorCoefficient(f,n), and then entering the four lines above to see the difference. It's rarely a good idea to have global variables in your procedures, though there are occasions when it can be useful (see Exercise 3.3 c)).

- d) Note the *comment lines* in the procedure TaylorCoefficient. Having a comment after each line of the procedure is excessive, but you should get into the habit of putting a comment at the top of each procedure which explains what it does, in enough detail that you'll understand it when you come back to your procedure in a year's time; and of putting occasional additional comments after lines which might be hard for the reader to understand, or to explain why you've done things a certain way.
- e) Try

```
>print(TaylorCoefficient);
```

This can be a useful way to remind yourself what your procedure does when it's several screens up in the worksheet (or in a different file: see Section 5). It can be interesting to do the same with Maple's own commands, though some of the most basic ones are *built in* (not written in Maple), and so can't be displayed. Try

>interface(verboseproc=2): # need to do this to display Maple's own commands
>print(numtheory:-tau); # this notation means you don't have to load numtheory
>print(eval);
>print(plot);

- f) You're not allowed to change the arguments of a procedure within the procedure.For example, within TaylorCoefficient, you aren't allowed to write n:=n+1;.(Nor is it necessary to declare the arguments to be local.)
- g) Resist the temptation to have your procedure print a friendly message like the coefficient in the Taylor series is 1/24. When you run Maple's own commands they don't do this: they say "1/24", plain and simple. The reason is that one day you'd like to use your useful procedure TaylorCoefficient inside another procedure: even something as simple as writing

```
TaylorCoefficient(cos,4)+TaylorCoefficient(exp,4);
```

is messed up if TaylorCoefficient insists on printing out stupid messages. Don't do it!

h) Personally, I think it's a good idea to have procedures with long and self-explanatory names (such as TaylorCoefficient) rather than short mysterious ones (such as H). The long name is certainly more boring to type, though. Remember that copy/paste is your friend. The CTRL-SPACE trick also works with your own procedure names: typing Tay followed by CTRL-SPACE is enough to bring up TaylorCoefficient.

1.2 Explicit returns

Often you know what value you want your procedure to return before you get to its last line. In such cases, you can use a **return** statement. Here are a couple of examples.

```
>StrangeFunction:=proc(x)
    # Returns the larger of x and 1/x
    if x>1/x then return x; end if;
    1/x;
 end proc;
>StrangeFunction(2);
>StrangeFunction(1/3);
>FindIndex:=proc(x,L)
   # Finds the index of first x in the list L, or returns 0 if none
    local i;
    for i from 1 to nops(L) do
       if L[i]=x then return i; end if;
    end do;
    0;
 end proc;
>FindIndex(egg,[grape,pea,egg,apple,egg,sandwich]);
>FindIndex(goat,[grape,pea,egg,apple,egg,sandwich]);
```

(If you try to plot a graph of StrangeFunction then you'll get an error message: see Section 4.)

1.3 Exercises

- a) Write a procedure WeirdFunction(x) which returns x^4 if $x \le 1$, x^3 if $1 \le x \le 2$, and $2x^2$ if $x \ge 2$. Test it on a few values of x.
- b) Write a procedure PrimeGap(n) which returns the first two consecutive primes which are at least n apart. So PrimeGap(100) should return 370261,370373, since these are the first two consecutive primes which are at least 100 apart. Include some examples with low n that you can check by hand.
- c) Write a procedure InterleaveLists(list1, list2) which returns the list [list1[1], list2[1], list1[2], list2[2], ...], provided that list1 and list2 have the same length. Include some examples.
- d) Write a procedure ChangeLast(list1, list2) which returns a list which is the same as list1, but with the last entry of list1 replaced by the last entry of list2. Include some examples.
- e) Write a procedure ElementaryMatrix(m,n,i,j) which returns an m by n matrix with zeroes in every entry except for the entry in the i^{th} row and j^{th} column, which should be 1. Include a couple of examples.

2 Error handling

2.1 Error returns

One day, someone will call one of your procedures with arguments that don't make sense. In such cases, you want to catch their mistake early and display a helpful error message so that they'll know what they've done wrong (don't take LATEX's error messages as your example...). You do this with the **error** statement, which prints out a message of your choice and aborts all running processes immediately. Consider **StrangeFunction**, for example. You don't want anyone to call this with x = 0 (since then 1/x doesn't make sense); and you also want to restrict x to be a real number, since if x is complex then it doesn't mean anything to ask whether or not x > 1/x. So you could write

```
>StrangeFunction:=proc(x)
    # Returns the larger of x and 1/x
    if not type(x, numeric) then
        error("x must be a real number");
    end if;
    if x=0 then
        error("x cannot be zero");
    end if;
    if x>1/x then return x; end if;
    1/x;
end proc;
```

numeric is just one of an enormous range of types you can test for: do ?type for a full list. Some other common types are: anything, boolean, complex, integer, list, negative, negint, nonnegative, nonnegint, nonposint, nonpositive, posint, positive, prime, rational, set, and sequential (which means *either* list *or* set).

Try out this revised version of the function with a selection of stupid inputs. Error handling in this way is very tedious, but it's an important part of good programming. **print** some of Maple's own commands again, and notice what a high proportion of them consists of error handling. *The people who use your procedure (apart from you of course) will be stupid, and anything they can do wrong they will.*

2.2 Restricting input type

Maple has another way of ensuring that the arguments you pass to procedures are of the right type. Instead of testing whether or not x is of numeric type inside the procedure, it's easier to do it like this:

```
>StrangeFunction:=proc(x::numeric)
# Returns the larger of x and 1/x
if x=0 then
error("x cannot be zero");
end if;
if x>1/x then return x; end if;
```

1/x; end proc;

See what happens when you try this revised version with some non-numeric input such as 1+I or x. You could also compare (using the help system) the behaviour of numeric with that of realcons (*real constant*).

2.3 Returning FAIL

Sometimes you want a program to end in a special way, to indicate something has gone wrong, without going so far as to say that that something is an **error**. Consider the procedure **FindIndex** in Section 1.2, for example. If I ask if a certain object belongs to a list, and it doesn't, that isn't really an error: nevertheless, there has to be some exceptional way of reporting the fact. (The original version of **FindIndex** returned 0, which isn't ideal.) The solution in such cases is to return **FAIL**, a special value which shows that the procedure couldn't complete normally.

```
>FindIndex:=proc(x::anything, L::list)
    # Finds the index of first x in L
    local i;
    for i from 1 to nops(L) do
        if L[i]=x then return i; end if;
    end do;
    return FAIL;
end proc;
```

The advantage of doing this rather than saying error(''value not found''); is that returning FAIL aborts FindIndex, but having an error statement aborts *everything*. For example, I might have a whole bunch of values of x that I want to search for in the list L. If I find a particular value of x then I do something, if I don't find it then I do nothing:

```
>for x in BigSetOfValues do
    i:= FindIndex(x,L);
    if i <> FAIL then
        DoSomething(x,i);
    end if;
end do:
```

If FindIndex had an error return when x wasn't found, then the first such x would cause the whole loop to terminate.

2.4 Debugging

The other sort of error, alas all too common, is one made by the programmer rather than the user. Suppose you write a procedure, test it, and it doesn't work. Perhaps it gives the wrong answers, or perhaps Maple gives an obscure error message when you try to run it. In such cases you need to *debug* your program.

The only advice I'm going to give here is that putting **print** statements at strategic points in your procedure can be a very useful debugging tool. For instance, suppose

that you get errors from the procedure FindIndex above (I don't think you do, but just suppose). A sensible thing to do would be to add a line as follows:

```
>FindIndex:=proc(x::anything, L::list)
# Finds the index of first x in L, or returns 0 if none
local i;
for i from 1 to nops(L) do
    print(i,L[i],x); # NEW LINE
    if L[i]=x then return i; end if;
end do;
return FAIL;
end proc;
```

Then you can trace exactly what's happening up to the point that the program goes wrong.

Maple has much more powerful debugging tools than the humble print statement: if you're interested, look up debugger in the online help.

2.5 Exercises

- a) Go back to the Exercises in Section 1.3, and amend each of your procedures so that they handle every error that you can possibly imagine a user making. Do this with all your procedures in future, too ...
- b) Write a procedure RemoveFirst(x::anything, L::list) which returns L with the first occurence of x removed, if there is such an occurence, and otherwise returns FAIL. Include some examples.

3 Recursion

3.1 First examples

3.1.1 Factorials

Recursion, where your procedure calls itself, can be a very useful technique for certain problems. The classic example is calculating factorials (of course, Maple has its own factorial command: this example is just for illustration).

Here's how you might write a factorial procedure non-recursively:

```
>factorial1:=proc(n::posint)
    # Calculates n!
    local i, result;
    result:=1;
    for i from 2 to n do
        result:=result*i;
    end do;
    result;
    end proc;
>factorial1(5);
>factorial1(1000);
```

Type this in and make sure you understand how it works. (In fact the final line **result**; isn't necessary, but it makes it clearer what value the procedure returns.)

Here's a recursive version:

```
>factorial2:=proc(n::posint)
    # Calculates n!
    if n<=2 then return n; end if;
    n*factorial2(n-1);
end proc;
>factorial2(5);
>factorial2(1000);
```

How does this work? Suppose we call factorial2(5). Since 5 > 2, the procedure returns 5*factorial2(4). The call to factorial2(4) returns 4*factorial2(3), so we now have 5*(4*factorial2(3)). After the call to factorial2(3), this becomes 5*(4*(3*factorial2(2))). Now $2 \le 2$, so factorial2(2) just returns 2, and we get the final answer 5*(4*(3*2)), which, of course, is 5!.

You can see the sequence of procedure calls in Maple itself if you add the line option trace; after the comment line # Calculates n!. Keep the value of n small when you're using option trace;!

3.1.2 Fibonacci numbers

A slightly more complicated example is given by a procedure to calculate *Fibonacci* numbers. Recall that the Fibonacci numbers F_n are defined by

$$F_0 = 0$$
, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ for $n \ge 2$.

(So $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, and so on.) Here's a recursive procedure for calculating Fibonacci numbers:

```
>FibonacciNumber:=proc(n::nonnegint)
    # Calculates nth Fibonacci number (with F_0=0, F_1=1)
    if n<=1 then return n; end if;
    return FibonacciNumber(n-1)+FibonacciNumber(n-2);
end proc;
>FibonacciNumber(7);
>FibonacciNumber(30);
```

You'll notice that FibonacciNumber(30); takes quite a long time to return its result: this procedure is impractical for even fairly small values of n. (On my laptop, FibonacciNumber(20) takes 0.015 seconds, FibonacciNumber(30) takes 2.7 seconds, and FibonacciNumber(35) takes 29.8 seconds. If you want to time this, or any other, command you can use time(FibonacciNumber(30));).

Why is it so inefficient? Let's trace a call to FibonacciNumber(5), as we did with factorial2(5) above. We have (abbreviating FibonacciNumber to Fib):

$$\begin{aligned} \texttt{Fib}(5) &= \texttt{Fib}(4) + \texttt{Fib}(3) \\ &= (\texttt{Fib}(3) + \texttt{Fib}(2)) + (\texttt{Fib}(2) + \texttt{Fib}(1)) \\ &= ((\texttt{Fib}(2) + \texttt{Fib}(1)) + (\texttt{Fib}(1) + \texttt{Fib}(0))) + ((\texttt{Fib}(1) + \texttt{Fib}(0)) + 1) \\ &= (((\texttt{Fib}(1) + \texttt{Fib}(0)) + 1) + (1 + 0) + ((1 + 0) + 1) \\ &= (((1 + 0) + 1) + (1 + 0) + ((1 + 0) + 1) \\ &= 5. \end{aligned}$$

That's 15 calls to FibonacciNumber altogether (try putting option trace in the procedure to see this). If you call FibonacciNumber(20) there's a total of 21891 calls.

3.2 Remember tables

This exponential growth of function calls is typical in recursive procedures which call themselves more than once. The solution is simple. Add the line option remember; in the same place that you would have added option trace;. Now you can happily calculate FibonacciNumber(10000);.

What this does is to associate a *remember table* to the procedure FibonacciNumber, which stores the value of FibonacciNumber(n) after calculating it. If you call FibonacciNumber again with the same value of n, it simply returns the value which it calculated before. So in the example above, it only calls FibonacciNumber once with each value of n between 0 and 5. What's more, suppose you've calculated FibonacciNumber(10000), and then ask Maple for FibonacciNumber(5000). Well, it's already had to work out FibonacciNumber(5000) in order to find FibonacciNumber(10000), so it just tells you the value it remembers. As you can see, the time saving can be immense.

Remember tables are very often useful with recursive procedures, but you can use them with any procedure which you think it's likely you'll be calling many times with the same arguments. Of course, the more times you call a procedure with different arguments the bigger the remember table gets, so you're paying for the increased speed by using up more of your computer's memory.

3.3 Exercises

a) The Fibonacci polynomials $F_n(x)$ are defined by

$$F_0(x) = 1$$
, $F_1(x) = x$, $F_n(x) = xF_{n-1}(x) + F_{n-2}(x)$ for $n \ge 2$

Write a procedure FibonacciPolynomial(n) which returns $F_n(x)$. Make sure it gives $F_6(x) = x^6 + 5x^4 + 6x^2 + 1$, not

$$x \left(x \left(x \left(x \left(x^{2}+1 \right) +x \right) +x^{2}+1 \right) +x \left(x^{2}+1 \right) +x \right) +x \left(x \left(x^{2}+1 \right) +x \right) +x^{2}+1.$$

What is the coefficient of x^{864} in $F_{1000}(x)$?

(Don't try to go much beyond F_{1000} if you don't want to hang your computer.)

b) The Hailstone iteration is defined as follows: given a positive integer n, repeatedly replace it either with n/2 (if it's even), or with 3n + 1 (if it's odd). Stop when you get to 1. (It has been checked that you will eventually reach 1 for all starting values of n up to about 10^{18} , but it is unknown whether this is the case for all starting values.) So, for example, if we take n = 11 we get the Hailstone sequence

11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.

The Hailstone length of n is the number of terms in its Hailstone sequence. For example, the Hailstone length of 11 is 15 (there are 15 terms in the above sequence), while the Hailstone length of 1 is 1, and the Hailstone length of 10 is 7 — its hailstone sequence is the tail of the one above,

```
10, 5, 16, 8, 4, 2, 1.
```

Write a procedure HailstoneLength(n) which returns the Hailstone length of n. (Do it recursively.) Which value of n between 1 and 100 has the largest hailstone length?

c) Enter the procedure FibonacciNumber as it first appeared in Section 3.1 (not using a remember table). Now write a procedure CountCalls(n) which returns the total number of times FibonacciNumber is called when you type FibonacciNumber(n). You are allowed to modify FibonacciNumber to include some lines which help with the counting.

Check that CountCalls(25) returns 242785.

4 Returning unevaluated

The technique in this section may seem esoteric, but it's one which is very often useful. Recall the procedure StrangeFunction(x) from Section 1.2.

```
>StrangeFunction:=proc(x::numeric)
    # Returns the larger of x and 1/x
    if x=0 then error("x cannot be zero"); end if;
    if x>1/x then return x; end if;
    1/x;
end proc;
```

Type restart; and re-enter this procedure. Then try typing StrangeFunction(x);. You get the error message

"StrangeFunction expects its 1st argument, x, to be of type numeric but received x".

That's because you've said that StrangeFunction needs to have *numeric* input, but you've given it the *symbol* x. If you delete the ::numeric and try again, you get the error message

"cannot determine if this expression is true or false: 1/x < x".

That's because StrangeFunction does different things depending on whether or not x>1/x, and since it doesn't know what x is it can't decide what to do.

You probably don't want to type StrangeFunction(x) on its own, but you may well want to do this:

```
>plot(StrangeFunction(x),x=0.5..2);
```

which gives exactly the same error message.

The solution is to tell the procedure **StrangeFunction** that, if it receives input which isn't just a number, then it should *return unevaluated*: that is, it should delay returning a result until it has an actual number to work with. Here's how you do it:

```
>StrangeFunction:=proc(x)
```

```
# Returns the larger of x and 1/x
if not type(x,numeric) then
    return 'StrangeFunction'(x);
end if;
if x=0 then error("x cannot be zero"); end if;
if x>1/x then return x; end if;
1/x;
end proc;
```

Modify StrangeFunction to look like this, and then try the two commands

```
>StrangeFunction(x); # Look, it returns without evaluating anything
>plot(StrangeFunction(x),x=0.5..2); # Hooray
```

You should look at return 'StrangeFunction'(x); as a magic invocation which tells the procedure to return unevaluated. You always use the same invocation, except that you should replace the x with whatever the arguments to the particular function are: for example,

```
>AnotherFunction:=proc(L,x,y,a)
```

return 'AnotherFunction'(L,x,y,a); # this line causes an unevaluated return

4.1 Exercises

a) Modify your procedure WeirdFunction(x) from Exercise 1.3a) so that you can work out $\sum_{n=0}^{10}$ WeirdFunction(n) by typing: sum(WeirdFunction(n),n=0..10);

5 Putting your procedures in separate files

Once you've written some useful procedures, you won't want them to sit in the middle of a Maple worksheet: you'll want to be able to use them in any worksheet, rather like you use Maple packages by typing with(packagename);. Here's how you do it:

- a) Pick one or more of the procedures in your worksheet that you'd like to be able to load into other worksheets.
- b) Copy them into a blank notepad file. You should copy each procedure from the first line ProcedureName:=proc... up until the last line end proc;, leaving out any of the > prompts.
- c) Save them with some filename such as firstprocs.txt in a sensible folder (e.g. M:\documents\Maple procedures).
- d) Start a new Maple worksheet and type read("M:\\documents\\Maple procedures\\firstprocs.txt"):.
- e) Check that you can use your procedures in this new worksheet.

If all of your procedures are in M:\documents\Maple procedures, then it can get tedious to keep typing this each time you read a new file of procedures. Instead, you can start each new worksheet with the line

```
>currentdir("M:\\documents\\Maple procedures");
```

Afterwards, you can just type read("firstprocs.txt"): to load up the procedures in a particular file. As an alternative to copying and pasting, you can save your procedures (called proc1, proc2 etc.) to firstprocs.txt (after the currentdir command) by typing save proc1,proc2,...,procn,"firstprocs.txt";

6 Submission

Send me an email (jonwoolf@liv.ac.uk), attaching your Maple worksheet yourname5 (please tidy it up first).