

MLC Lab Visit - Lab 01 - Introduction to Maple

Mth 355 (a.k.a. Mth 399) Jan 8, 2003 Maple 7

Revised Jan 10, 2003, Jan 28, 2003

Bent E. Petersen

There are 3 problems below. Problem solutions are due Jan 15, 2003. There's a lot to absorb. This initial selection of problems will take time to solve. Do not wait to the last minute! Email your solutions to me as Maple worksheet attachments. Your worksheet must execute correctly for full credit.

- Introduction

If you are viewing the MWS version of this document you will note all the Maple output has been removed. You will have to execute each command (by pressing Enter when the cursor is on the command line) to see the output. Take the opportunity to experiment! Change some things.

The static PDF version of this document shows all of the Maple output. It is useful to look at when Maple is not available.

Maple is a CAS, that is, a Computer Algebra System. It performs mathematical operations symbolically, but a large number of robust numerical routines are also built in. Maple can be used interactively as a rather fancy calculator, but it can also be used as a flexible programming language. In this introduction we will look both at interactive use and programming.

[>

- Getting Started

Scroll down to the restart command below. Position the cursor on the line containing the restart command and press Enter. Maple will execute the restart command and then position the cursor on the next command, skipping over all the intervening text. Now press Enter to execute the next command, etc., or be brave and edit it first. Experiment! Some of the commands depend on previous assignments, etc. If you skip around and something doesn't work you may just have to execute a few of the previous commands.

This entire document was written in Maple. The sample commands were selected to illustrate a few Maple features to get you started using Maple. To learn more you should make heavy use of the online help. The help is very good and usually includes a few examples.

Note each Maple command must be terminated by a colon or a semicolon (except help commands preceded by a question mark). The effect of the colon is to suppress output from the corresponding command, though the command is still carried out.

Maple commands are executed by pressing the Enter key when the mouse cursor (pointer, thumb) is anywhere in the line containing the commands. Note that Maple skips over the interpolated text comments (like this one). To execute the commands on this worksheet position the mouse cursor on the command line and press Enter. Edit the command first if you wish. Explore! Simply waiting for something to happen will not be productive.

You can spread a command over several lines by postponing the terminating colon or semicolon. You simply move to a new line by pressing Enter. Maple will chatter at you when you move to a new line in this manner if the previous command is unterminated. Ignore it, but keep in mind a command will not be executed before it is properly terminated.

You can also stack up several commands on one line by terminating them individually with colons or semicolons. All the commands on a line are executed when you press the Enter key (with the cursor anywhere on the line).

Here's a useful fact: You can open a new command line below the current one by pressing Ctrl-J, or above the current line, by pressing Ctrl-K. This is pretty useful when you realize you omitted something at a certain step.

[>

The Worksheet

When you are using Maple in a window environment it is possible to move around on the worksheet by left-clicking the mouse. As a result, commands may end up being executed in a nonlinear order. This can cause some confusion, since there is no visual clue. One way to fix a mess is to have Maple re-execute the whole worksheet (look on the Edit menu). This works best if old expressions are cleaned up first, so it is a good idea to start each worksheet with the command `restart`; You do not need to do so of course

[> `restart;`

To find out what `restart` does execute the command `?restart` - try it.

[>

Note that commands starting with `?` invoke the help system and do not need to be terminated with a semi-colon (but it doesn't hurt to terminate them).

[>

- Assignment and Ditto Operators

The assignment operator in maple is := (colon and equals sign juxtaposed). The equals sign by itself does not perform assignment.

Maple has two ditto operators, % and %%. The value of % is the previously evaluated expression, the value of %% is the one before that. Since the Worksheet commands may be executed in any order, the ditto operators can cause a lot of confusion. It is probably best to restrict them to the same line as the expressions they refer to.

[>

- Execution Time

There are times when we want to know how long it takes to carry out a given task. The time() function keeps track of the number of seconds that have passed since the beginning of the current Maple session. We can use it to record start and stop times. Here's an example:

```
> stime:=time(): 20000!: etime:=time()-stime;  
                               etime := 3.469
```

Let's execute the same command again:

```
> stime:=time(): 20000!: etime:=time()-stime;  
                               etime := 0.
```

Here we see an important property of Maple - it remembers and recalls previously computed expressions. You'll have to be careful if you intend to time calculations.

[>

- Sets, Intersection, Union, Difference

Here's a simple way to define a (finite) set in Maple:

```
> A1:={3,1,3,13,5,36,7,11,6,9,16,17};  
      A1 := {1, 3, 5, 6, 7, 9, 11, 13, 16, 17, 36}
```

Notice we do not have to worry about repetitions since Maple discards repeated elements

automatically.

The sequence command `seq()` can be useful for creating sets:

```
> A2:={ seq(k^2, k=2..9) };
                A2 := { 4, 9, 16, 25, 36, 49, 64, 81 }
> A3:={ seq(ithprime(k), k=1..10) };
                A3 := { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 }
```

Note the Maple convention for a *range* `k=1..10`, in the above two constructions.

The `intersect` statement computes the intersection of sets:

```
> A1 intersect A2;
                { 9, 16, 36 }
> A4:=A1 intersect A2 intersect A3;
                A4 := { }
```

Note the notation `{ }` for the empty set.

Multiple intersections can be calculated in a more convenient manner by the `intersect` function call.

```
> `intersect`(A1,A2,A3);
                { }
```

The quote-marks here are back-ticks. They are needed because *intersect* is a Maple keyword. If the tick-marks are absent Maple will try to evaluate *intersect* immediately as above rather than as a function call.

I find the back-ticks hard to see. Perhaps you do too We can use assignment to give the `intersect` function a new name (for example, `mint` for multiple intersection):

```
> mint:='intersect':
> mint(A1,A2);
                { 9, 16, 36 }
> mint(A1,A2,A3);
                { }
```

Naturally unions behave similarly:

```
> A1 union A2;
                { 1, 3, 4, 5, 6, 7, 9, 11, 13, 16, 17, 25, 36, 49, 64, 81 }
```

```
> A1 union A2 union A3;
      { 1, 2, 3, 4, 5, 6, 7, 9, 11, 13, 16, 17, 19, 23, 25, 29, 36, 49, 64, 81 }
```

Again we avoid using 'union' for multiple unions by assigning a new name, muni().

```
> muni:='union':
> muni(A1,A2,A3);
      { 1, 2, 3, 4, 5, 6, 7, 9, 11, 13, 16, 17, 19, 23, 25, 29, 36, 49, 64, 81 }
```

Note the functions mint() and muni() are defined just in this worksheet. They do not become part of Maple and will not be available in other worksheets unless you define them again (except in the case of worksheets sharing a work space).

The difference of two sets may be calculated by using the minus command (see also the section on symmetric difference below):

```
> A5:={seq(k,k=1..12)};
      A5 := { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 }
> A6:={seq(2*k,k=1..8)};
      A6 := { 2, 4, 6, 8, 10, 12, 14, 16 }
> A7:=A5 minus A6;
      A7 := { 1, 3, 5, 7, 9, 11 }
> A8:=A6 minus A5;
      A8 := { 14, 16 }
>
```

Functions

There are several ways to define a function in Maple. One way is by the "arrow notation." In this formalism a function f is defined by describing how, given x , to compute $f(x)$. The notation easily extends to several variables:

```
> f1:=x->exp(sin(x));
      f1 := x → esin(x)
> f1(u); f1(Pi/2);
      esin(u)
      e
```

Note pi means the Greek letter π whereas Pi means the number usually symbolized by the Greek letter π . The number Pi is manipulated symbolically. That is, a value is not assigned to expressions

involving Pi unless it can be done exactly, or an estimate is specified.

```
> sin(Pi/3);
```

$$\frac{1}{2}\sqrt{3}$$

```
> sin(Pi/7); evalf(%);
```

$$\sin\left(\frac{1}{7}\pi\right)$$

.4338837393

The evalf() function evaluates an expression to floating point at the default precision (set by Digits) or prescribed. For example,

```
> evalf(Pi,40);
```

3.141592653589793238462643383279502884197

Try evalf(Pi,2000);

```
>
```

- Symmetric Difference of Sets

The symmetric difference of two sets is the union of the complement of either one in the other. It is a measure of how much two sets differ. Here is a function which computes the symmetric difference of two sets.

```
> symdiff:=(x,y)->(x minus y) union (y minus x);
```

symdiff := (x, y) → (x minus y) union (y minus x)

Here's a few examples:

```
> symdiff(A1,A2);
```

{ 1, 3, 4, 5, 6, 7, 11, 13, 17, 25, 49, 64, 81 }

```
> A9:={2,3,4,5,6};
```

A9 := { 2, 3, 4, 5, 6 }

```
> A10:={1,2,3,4,5};
```

A10 := { 1, 2, 3, 4, 5 }

```
> symdiff(A9,A10);
```

{ 1, 6 }

```
>
```

- Cardinality of a Set

The number of elements in a set can be found by using the *number of operands*, `nops()`, function.

```
> nops(A1);  
11  
> nops({ });  
0  
> nops(A10);  
5  
>
```

- Elements of a Set

Recall the set `A3`

```
> A3 := { seq(ithprime(k), k=1..10) };  
A3 := { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 }
```

The elements of a set are its operands in Maple-speak. The operands are returned by the `op()` function.

```
> op(A3);  
2, 3, 5, 7, 11, 13, 17, 19, 23, 29
```

This may not look useful but we will see an application in the next section.

If we just want to test for membership we can use the `member()` function:

```
> member(7, A3);  
true  
> member(9, A3);  
false  
>
```

- Lists

A list is an ordered sequence of expressions enclosed in square brackets.

```
> L1 := [3, 7, 2, 1, 4, 9, 3, 1, 7];  
L1 := [3, 7, 2, 1, 4, 9, 3, 1, 7]
```

The sort function sorts the elements of a list L into ascending order if we have a comparison between elements of the list (or specify one - check help).

```
> sort(L1);
```

```
[1, 1, 2, 3, 3, 4, 7, 7, 9]
```

The op() function returns the list of elements in a list in the order specified in the list.

```
> op(L1);
```

```
3, 7, 2, 1, 4, 9, 3, 1, 7
```

If we surround the returned list of elements by [...] we, of course, get the original list back.

```
> is(L1=[op(L1)]);
```

```
true
```

Note this useful test for equality.

If however we surround the returned list of elements by { ... } we obtain the set of list elements, that is, duplicates are removed.

```
> {op(L1)};
```

```
{ 1, 2, 3, 4, 7, 9 }
```

Using these ideas we can construct a function which removes duplicate elements in a list.

```
> remdup:=x->[op({op(x)})];
```

```
remdup := x → [op({op(x)})]
```

Let's test it:

```
> remdup(L1);
```

```
[1, 2, 3, 4, 7, 9]
```

It works! Notice however that the original order of L1 is not preserved. If you want to keep the order in some sense, for example, you want to remove all but the first (or last) occurrence of each element and otherwise keep the elements in the original order, you'll have to work harder to come up with a solution (see below).

A very useful construction is the process for selecting sublists:

```
> L1[2..6];  
[7, 2, 1, 4, 9]
```

Of course, we can also use subscripting to extract individual entries from a list:

```
> L1[2]; L1[3];  
7  
2  
>
```

- Subsets of a Set

Recall the set A3

```
> A3:= { seq(ithprime(k), k=1..10) };  
A3 := {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

The subset statement is pretty simple:

```
> {13,11,2,3} subset A3;  
true  
> {13,11,4,3} subset A3;  
false
```

You can also use a function call, though you have to escape the reserved word subset with back-ticks.

```
> `subset`({13,11,2,3},A3);  
true  
>
```

- Maple Procedures

Maple procedures are similar to Maple functions, but much more complicated processing can be specified in a procedure. The definition of a procedure follows the pattern:

```
name := proc( args )  
  local variable list  
  ... commands ...  
end;
```

Note inside a procedure `nargs` tells you how many arguments were passed to the procedure and `args` is the list of arguments. This is very handy. The arguments do not have to be declared in the procedure definition, so it becomes possible to write procedures which take a variable number of arguments. You can also test the type of the arguments and so write procedures that are polymorphic, that is, change their behavior depending on the type of data received.

The procedure returns the last value evaluated in the body of the procedure. Sometimes it is necessary or just clearer to specify the returned value explicitly by using the `return` statement. The local variables are variables used only in the procedure. They do not conflict with variables of the same name existing outside of the procedure.

Once a procedure has been defined (that is, its definition has been executed) it can be used the same way as any other Maple procedure.

Here is a procedure, `remldupe()`, which removes duplicate entries from a list. It does it in such a way that only the first occurrence of each distinct entry is retained. Thus the retained entries are in the same order as the corresponding entries were in `L`. The `reml` part of the name is supposed to suggest "remove later duplicates."

```
> remldup:=proc(L)
> local LL,n,k,x;
> n:=nops(L);    # number of entries in L
> if n<=1 then
>   return L;    # if L has 0 or 1 entries we are done
> end if;
> #
> LL:=[L[1]];    # add the first entry of L to LL
> for k from 2 to n do
>   if `member`(L[k],L[1..k-1]) then
>     ;          # do not add any entries which come earlier
>   else        # (so in effect remove later duplicates)
>     LL:=[op(LL),L[k]];
>   end if;
> end do;
> LL:
> end;
```

Note above the use of the Maple comment character `#`. Everything following `#` to the end of the current line is ignored by Maple.

Let's test `remldup` on our list `L1`:

```
> L1; remldup(L1);
```

```
[3, 7, 2, 1, 4, 9, 3, 1, 7]
```

```
[3, 7, 2, 1, 4, 9]
```

Here's another example:

```
> L2:= [3,1,3,3,3,3,4,3,4,3,3,2,5];
```

```
L2 := [3, 1, 3, 3, 3, 3, 4, 3, 4, 3, 3, 2, 5]
```

```
> remldup(L2);
```

```
[3, 1, 4, 2, 5]
```

Note above the construction `[op(L), x]` to add the entry `x` at the end of the list `L`.

Perhaps we'd rather retain the last occurrence of each entry in the list. In that case this last string should yield `[1,4,3,2,5]`.

A simple way to do this is to reverse the list, use the procedure `remldup()` above, and then reverse the result. We can use the `Reverse()` function from the `ListTools` library:

```
> rev:=ListTools[Reverse]:
```

```
> rev( (remldup(rev(L2))) );
```

```
[1, 4, 3, 2, 5]
```

Using `ListTools[Reverse]()` is probably overkill. If we are careful we can achieve the reversal by using negative indexing.

```
> seq(L2[-k],k=1..nops(L2));
```

```
5, 2, 3, 3, 4, 3, 4, 3, 3, 3, 3, 1, 3
```

Perhaps a better approach is to write a procedure `remfdup()`, along the lines above, to remove the first occurrences of each duplicated entry, and to retain only the last.

```
>
```

- Cartesian Product of Sets

The Cartesian product $X \times Y$ of two sets X and Y is the set of ordered pairs $[x,y]$ where x is an element of X and y is an element of Y . The Cartesian product of an ordered set of sets is defined similarly. Can you write a definition?

Now ordered pairs, ordered triples, and so on, are simply lists, so we construct the Cartesian product in Maple in terms of lists. Here is a recursive Maple procedure to compute the Cartesian product of any number of sets (`mcarp` is motivated by multiple Cartesian product):

```
> mcarp:=proc() local Z,k,x,y;
```

```
> option remember;
```

```

>   if nargs=0 then
>     Z:={};
>   elif nargs=1 then
>     Z:=args[1];
>   else Z:={};
>   for x in mcarp( seq(args[k], k=1..nargs-1) ) do # note
recursion
>     for y in args[nargs] do
>       Z:= Z union {[op(x),y]};
>     od;
>   od;
> fi;
> return Z;
> end:

```

Here nargs is the number of variables (in our case sets) passed to the procedure on a function call. Let's test it:

```

> B1:={1,2,3}; B2:={b,a}; B3:={a,c,b}; B4:={}; B5:={3,6,7,8,4};
      B1 := { 1, 2, 3 }
      B2 := { a, b }
      B3 := { c, a, b }
      B4 := { }
      B5 := { 3, 4, 6, 7, 8 }
> mcarp(B1,B2,B3);
{ [1, b, b], [1, b, a], [1, b, c], [3, b, b], [3, b, a], [3, b, c], [2, b, a], [2, b, c], [2, a, b],
  [2, a, a], [2, a, c], [3, a, b], [3, a, a], [3, a, c], [1, a, b], [1, a, a], [1, a, c], [2, b, b] }
> mcarp(B1,B2,B3,B4);
      { }
> mcarp(B1,B2,B3,B5);
{ [3, a, b, 7], [3, a, b, 6], [2, a, c, 4], [3, a, b, 8], [1, a, b, 3], [3, a, b, 4], [2, a, c, 8],
  [3, a, b, 3], [3, a, c, 4], [2, a, c, 6], [2, a, c, 7], [2, b, b, 4], [1, a, c, 8], [2, b, b, 3],
  [2, b, b, 6], [1, a, c, 7], [1, a, c, 4], [1, a, a, 7], [1, a, a, 8], [1, a, c, 6], [1, a, a, 6],
  [1, a, a, 4], [1, a, b, 8], [1, a, a, 3], [1, a, b, 4], [1, a, c, 3], [3, a, c, 7], [3, a, c, 8],
  [3, a, c, 6], [3, a, c, 3], [3, a, a, 7], [3, a, a, 8], [1, a, b, 6], [1, a, b, 7], [3, a, a, 6],
  [3, a, a, 3], [3, a, a, 4], [2, b, b, 7], [2, b, b, 8], [1, b, b, 8], [1, b, b, 7], [1, b, b, 6],
  [1, b, b, 4], [1, b, b, 3], [1, b, a, 7], [1, b, a, 6], [1, b, a, 4], [1, b, a, 3], [1, b, c, 4],
  [1, b, c, 3], [1, b, a, 8], [3, b, b, 6], [3, b, b, 4], [3, b, b, 3], [1, b, c, 8], [1, b, c, 7],
  [1, b, c, 6], [3, b, a, 4], [3, b, a, 3], [3, b, b, 8], [3, b, b, 7], [3, b, a, 7], [3, b, a, 6],
  [3, b, a, 8], [3, b, c, 8], [3, b, c, 7], [3, b, c, 6], [3, b, c, 4], [3, b, c, 3], [2, b, a, 4],

```

```
[2, b, a, 3], [2, b, a, 7], [2, b, a, 6], [2, b, c, 6], [2, b, c, 4], [2, b, c, 3], [2, a, a, 4],
[2, a, a, 3], [2, a, b, 8], [2, a, b, 7], [2, a, b, 6], [2, a, b, 4], [2, a, b, 3], [2, b, c, 8],
[2, b, c, 7], [2, a, a, 6], [2, a, a, 8], [2, a, a, 7], [2, b, a, 8], [2, a, c, 3] }
```

The "option remember" sometimes speeds up recursive routines at the expense of using more RAM.

>

- Problems

Problem 1.

Write a procedure `remfdup()` which takes as input any list `L` and returns a list formed from `L` by retaining only the last occurrence of each entry in `L`. Order must be preserved.

For clarification note

```
remldup([1,2,3,2]); yields [1,2,3]
```

```
remfdup([1,2,3,2]); yields [1,3,2]
```

Problem 2.

Write a non-recursive procedure to compute the Cartesian product of any finite number of finite sets.

Note you will have to loop over the arguments. If you find this problem too difficult then, for partial credit, write a routine to compute the Cartesian product of any 2 finite sets. Indicate how you might use your routine (without rewriting it) for the case of 3 sets.

The Maple function call `irem(m,n)` returns the integer remainder when `m` is divided by `n`. Here are some examples:

```
> irem(142856,7);
```

```
0
```

```
> ifactor(142856);
```

```
(2)3 (7) (2551)
```

Sure enough 7 is a factor of 142856.

```
> irem(9937,11);
```

```
4
```

Problem 3.

Write a procedure `g(a,b)` which returns the *list* consisting of all integers from `a` to `b` which are divisible by 3 and by 7, but are not divisible by 5.

Your code should probably start something like

```
g:=proc(a,b)
local A,B,...;
if nargs<>2 then
  return FAIL;
fi;
A:=ceil(a); B:=floor(b);
.....
end;
```

As test `g(41, 327)`; should return

```
[42, 63, 84, 126, 147, 168, 189, 231, 252, 273, 294]
```

```
[ >
```

```
[ >
```

```
[ >
```