

## Maple Lecture 15. Defining Mathematical Functions

We first consider how to turn formulas into functions. As many functions are not continuous and cannot be defined by one single formula, we will see how to create piecewise functions.

This lecture note corresponds to [1, Sections 8.1, 8.2, and 8.3]. Chapter 6 of [2] is recommended reading.

### 15.1 Mathematical Functions

Maple offers a vast library of mathematical functions. Here we see how to add our own functions.

```
[> expT := 30+40*exp(-0.2*t);          # cooling off
```

Suppose we wish to evaluate this expression for several values of  $t$ . There are three ways for doing this:

**1) Evaluation as a formula.** Suppose we wish to know the value at  $t = 1.5$ :

```
[> eva := subs(t=1.5,expT);
[> evalf(eva,30);
```

This is not such a good way to evaluate, although we may make it better with a macro:

```
[> macro(evatemp = evalf @ subs);
```

The `@` symbol stands for function composition (we will see more applications of this in lecture 17). Here it means: apply `evalf` after `subs`, i.e.: at the result of `subs`. (Note that the macro is an abbreviation mechanism like an alias, except that the alias also affects the output.)

We can use this macro then in a similar fashion as the substitute:

```
[> evatemp(t=1.5,expT); evatemp(t=2.3,expT);
```

**2) Evaluation with a procedure.** We can also use the formula in a procedure.

```
[> proctemp := proc(x)                # define function
>   description 'cooling temperature';
>   global expT;                    # expression is global
>   return evalf(subs(t=x,expT));
> end proc;
[> proctemp(2.4); proctemp(2);        # evaluate function
```

A shorter way to make a procedure from the formula is via

```
[> f := codegen[makeproc](expT,t);
```

Notice we must tell `makeproc` the name of the independent variable.

**3) Evaluation using `unapply`.** A fast and convenient way to turn a formula into a function goes like this:

```
[> funtemp := unapply(expT,t);
[> funtemp(1.5);                    # verify
```

It also works for functions in several variables:

```
[> monkey_saddle := x^3 - 3*x*y^2;    # polynomial in two variables
[> mky := unapply(monkey_saddle,x,y); # function of two variables
[> mky(2,3);                          # sanity check
```

To see why we call this polynomial a monkey saddle, execute the following:

```
[> plot3d(mk,-2..2,-2..2);
```

## 15.2 Arrow Operators

This is the fourth way to define functions from formulas. Arrow operators are close to the notation used in mathematics to denote functions.

```
[> arrtemp := t -> 30 + 40*exp(-0.2*t);
[> arrtemp(1.7);
```

Note that we have a problem if we wish to use the formula `expT`:

```
[> arrtemp2 := t -> expT;
[> arrtemp2(1.9);
```

This situation is particularly annoying if `expT` is so huge we have no courage to retype it. The solution is with `substitute`:

```
[> arrtemp3 := subs(body = expT, t -> body);
[> arrtemp3(2.1);
```

Also multiple arguments work with the arrow operator. Suppose we wish to give the accuracy, say we only want two decimal places instead of 10:

```
[> newtemp := subs(body = expT, (t,n) -> evalf(body,n));
[> newtemp(3,2); newtemp(3,3); newtemp(3,4);
```

With `subs` we have a very flexible mechanism to extend functions.

## 15.3 Piecewise Defined Functions

Not all functions are as smooth and continuous as the one we used as an example so far. For example, consider a step function.

```
[> step := proc(x)
>   description 'definition of a step function';
>   if x < 0 then
>     return 0
>   elif x < 1
>     return 1;
>   else
>     return 2;
>   end if
> end proc;
```

Let us check the function at some points:

```
[> step(0); step(1/2); step(1); step(1.5);
```

While the above definition worked fine for evaluation, we have a problem when it comes to differentiating :

```
[> diff(step,x);
```

But that is incorrect, as the derivative does not exist at 0 and 1.

```
[> int(step,x=0..3);
```

This does not make much sense either.

We better define the step function differently:

```
[> step2 := piecewise(x<0,0,x<1,1,2);      # same as procedure step
```

Now you see that we can differentiate and integrate:

```
[> diff(step2,x);
[> int(step2,x=0..3);
```

## 15.4 Assignments

1. Define a function `midpoint`, which returns the average of two arguments given on input. For example, `midpoint(2,3)` returns  $3/2$ , `midpoint(a,b)` returns  $a/2 + b/2$ .
2. For two positive natural numbers  $a$  and  $b$  we can define the least common multiple  $\text{lcm}(a, b)$  via their greatest common divisor  $\text{gcd}(a, b)$ , i.e.:  $\text{lcm}(a, b) = \frac{ab}{\text{gcd}(a,b)}$ . When  $a$  and/or  $b$  are zero,  $\text{lcm}(a, b) = 0$ .  
Use the arrow operator and piecewise to define the function `my_lcm` which returns the least common multiple of two natural numbers.  
To test your least common multiple, take random numbers and compare with Maple's `lcm` procedure. Also test the cases when zeroes are given on input.
3. Consider the general quadratic polynomial  $p = ax^2 + bx + c$ , with the coefficients as parameters. Suppose we wish to create a general quadratic function of  $x$ , for example `f(1)` returns  $a + b + c$ , `f(2)` returns  $4a + 2b + c$ , etc...  
Type `p := a*x^2 + b*x + c` and, without retyping the formula for  $p$ , create `f` in **four** different ways. Each time, test your function on some values for  $x$ .
4. The quadratic formula allows to write the roots of  $p = ax^2 + bx + c$  explicitly in terms of  $a$ ,  $b$ , and  $c$ .
  - (a) Use the output of the solve command (i.e., the output of `solve(p,x)`) to create a function which returns the first root, in function of the parameters  $a$ ,  $b$ , and  $c$ .
  - (b) From the output of solve, select the discriminant  $\sqrt{b^2 - 4ac}$  (use the `op` command) and create a function which returns the value of the discriminant for given values of  $a$ ,  $b$ , and  $c$ .
  - (c) The formula for the first root is only valid when  $a \neq 0$ . Use `piecewise` to create a function which tests on the case  $a = 0$  and which returns in this case  $\frac{-c}{b}$ , otherwise the function you created in (a) is invoked.
  - (d) Extend the function created in (c) to deal with the case when  $b = 0$ . In case  $b = 0$ , the function must return  $\infty$ , or the symbol `infinity`, the function created in (c) is called.
5. The exponential function  $e^x$  can be approximated by

$$\sum_{k=0}^N \frac{x^k}{k!}, \quad \text{for } N \text{ any number.}$$

Create a function `expfun` which has two arguments:  $N$  and  $x$ . Test how high  $N$  should be in `expfun(N,-0.1)` to let it coincide with the 10 decimal places you see in the output of `exp(-0.1)`.

6. The height of a polynomial with integer coefficients is defined as its largest coefficient. For example,

```
[> p := randpoly(x);
[> coeffs(p);
[> max(%);
```

returns the height of some random polynomial.

Use the commands `coeffs` and `max` to define a function `height` which returns the height of a polynomial. Test if your function also works for polynomials in several variables.

## References

- [1] A. Heck. *Introduction to Maple*. Springer-Verlag, third edition, 2003.
- [2] M.B. Monagan, K.O. Geddes, K.M. Heal, G. Labahn, S.M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 9 Introductory Programming Guide*. Maplesoft, 2003.