Departments of Civil Engineering and Mathematics CE 109: Computing for Engineering *Mathematica* Session 3: Building your own functions

In this course, you'll have the chance to compare three quite different forms of computer software: the *Mathematica* system, a spreadsheet application, and the FORTRAN programming language. Spreadsheets are excellent for manipulating data, drawing graphs, doing statistics *etc*. (and they tie-in very closely with word processor applications, which is useful for preparing reports). FORTRAN is the dominant programming language for scientific computing, as it has been since the 1960's, and still the standard way for doing "number crunching"; but writing and running FORTRAN programs tends to be a very abstract process compared with the way you can directly get to grips with data in a spreadsheet.

In some ways, *Mathematica*'s capabilities are a combination of those of spreadsheets and FORTRAN: it has a very good user interface, it's wonderful for making graphs (and movies), *and* it has a programming language every bit as good as FORTRAN's; what you lose, in general, compared with FORTRAN programs is speed of execution. Remember, though, that *Mathematica* can do something which neither of the other two can do at all: it can do mathematical calculations in symbolic form.

We haven't said too much about symbolic calculations in these sessions, partly because *Mathematica* tends to *just do* the kind of calculations you're familiar with, which isn't very enlightening. However, we hope that as you study more advanced mathematical techniques you'll come back to *Mathematica* to explore what else it can do.

This session is concerned with building new *Mathematica* functions, which is the first and fundamental step on the way to being a *Mathematica* programmer.

1. Defining simple functions

You have already seen some simple function definitions in Session 2; for example this function for wave motion on a string:

wavel[x_, t_]:= $Sin[\pi x] Cos[\pi t]$

From the moment you evaluate this piece of code until the end of your session — or until the system crashes, whichever is sooner! — the function wavel can be used in combination with all *Mathematica*'s built-in functions and commands.

Some points to notice about the definition of wave1. First, it begins with a *lower-case* w, even though *Mathematica*'s built-in functions all begin with capitals. Well, actually, precisely *because* they do. The last thing you want is to define a new function and accidentally give it a name that's already been used by *Mathematica*, and if you stick to lower-case letters you avoid that.

Secondly notice that the combined symbol := ("colon equals") has been used, where one might have expected the simple = sign. The difference between = and := in *Mathematica* is quite subtle: for the moment it's worth learning the rule of thumb that := is almost always best for defining **functions**, whereas = is almost always best when assigning values to **variables** in commands like z = 3.

Finally, notice the use of the underscore symbol, _, on the left-hand side. This is essential, and you'll see why in a minute.

(i) Define a function quad by typing

quad[x_] := $x^2 - 4$

Test your definition by typing stuff like

```
quad[3]
quad[x]
quad[x + 2]
quad[y]
Plot[quad[x],{x, -3, 3}]
Expand[quad[x + 2]]
D[quad[x], x]
Integrate[quad[t], t]
```

etc.

Now clear your definition by typing

```
Clear[quad]
```

and try the whole thing again, but this time without the underscore: that is, starting with

quad[x] := $x^2 - 4$

What goes wrong? How would you describe the role of the underscore symbol?

(ii) Write some Mathematica code to define a function fib in such a way that fib[n] is given by the formula

$$\frac{1}{\sqrt{5}} \left(\alpha^{n+1} - \beta^{n+1} \right)$$

where

$$\alpha = \frac{1+\sqrt{5}}{2}$$
 and $\beta = \frac{1-\sqrt{5}}{2}$.

(Use Expand to make sure Mathematica multiplies out the brackets.)

Examine the sequence fib[0], fib[1], fib[2], fib[3],.... What's going on here?

2. Functions beyond numbers

When we speak of a *function* in mathematics, we usually (though not invariably) mean a rule linking *numbers* to other *numbers*. In *Mathematica*, though, and in computing generally, the term **function** is used to describe anything with a collection of inputs and outputs, whatever form those inputs and outputs take.

For example, *Mathematica*'s Integrate facility is a function in this sense. It takes input in the form of an algebraic expression and a variable name, as in:

```
Integrate[2x Sin[x<sup>2</sup>], x]
```

and returns output in the form of another algebraic expression, as in:

 $-Cos[x^2]$

Solve is also a function. It takes input in the form of a set of equations and a set of variables to be solved for, as in:

Solve [$\{x + 2y = 5, x y = 2\}$, $\{x, y\}$] and outputs the solutions in a special format as *Mathematica* "rules", as in:

$$\{\{\mathbf{x} \rightarrow \mathbf{1}, \mathbf{y} \rightarrow \mathbf{2}\}, \{\mathbf{x} \rightarrow \mathbf{4}, \mathbf{y} \rightarrow \frac{1}{2}\}\}$$

By now it may have occurred to you that, since nearly every use of *Mathematica* involves inputs and outputs, nearly all of what *Mathematica* does can be described in terms of functions. In graphics, too, the appearance of a graph—e.g. the appearance of lines (thick/thin, solid/dashed) and colours—is often adjusted by way of functions. Recall the RGBColor command in Session 1, whose inputs are the (relative) magnitudes of red, green and blue, and whose output is, in effect, the resultant colour.

Since functions are so basic to *Mathematica*'s workings, the idea of function-building introduced in section 1 can be carried across to virtually all contexts in which you want to get *Mathematica* to do something new, whether or not you're working with numbers.

Two examples of this process. (1) the following function takes a mathematical expression as an input, and plots graphs of both the expression and its derivative:

```
funcWithDerivPlot[expr_, x_, xmin_, xmax_]:=
Plot[Evaluate[{expr, D[expr, x]}], {x, xmin, xmax},
PlotStyle->{Thickness[0.01], RGBColor[1,0,0]}]
```

(i) Test this function on some mathematical expressions. Adapt funcWithDerivPlot to define a new function which draws simultaneous graphs of the expression and its first and second derivatives.

(2) Suppose we wanted to build into *Mathematica* the ability to locate the stationary points of an expression. This process can be broken down into stages, along these lines:

(*a*) differentiate the expression;

(*b*) solve the equation "derivative = 0" to get the *x*-coordinates;

(c) substitute back into the expression to get the y-coordinates.

And the *Mathematica* function could then end up as:

```
statPoints[expr_, x_] :=
  (deriv = D[expr, x];
   sols = Solve[deriv == 0, x];
   {x, expr} /. sols)
```

There are two new bits of *Mathematica* syntax here. (1) the commands that make up the function must be grouped together in parentheses (...). (2) Each command, except the last, must end with a semicolon; thus the three individual commands are "strung together" to produce in effect a larger "compound command". You'll also notice that the first two commands don't output anything: another effect of a semi-colon on the end of a command is to "turn off" output.

(ii) Test the statPoints function on the expression $x^3 - 4x^2 - 3x - 5$. The inputs are an expression and a variable: how would you describe the output? Check whether your results fit the graph of the expression.

3. The Rainbow Bridge

Finally, another engineering activity, based on structural mechanics as in the previous sessions, but for the more sophisticated case of a trussed bridge structure. The idea here is not to make sense of the mathematics underlying the structural analysis (you will meet this in your studies of structures in due course), but to use two aspects of *Mathematica*—the ability to make "movies" (see Session 2) and the ability to define new functions—to explore some of the structural properties of simple trusses.

We have ourselves defined some new functions for this activity¹, which we've stored in a *Mathematica* notebook. You can get this via a Web browser from either of these sites:

http://metric.ma.ic.ac.uk/civ-eng/rainbow.nb
http://www.cv.ic.ac.uk/ce109/rainbow.nb

Save this notebook (either in your personal directory, or in C:\Temp), and open it in *Mathematica*. To set up all the functions defined in the notebook, you can simply go to the Kernel menu and select Evaluation \rightarrow Evaluate Notebook.

Now try:

bridgeMovie[10000, rainbowFun]

All being well, you should have got a 20-frame animation of a test load of 10000 N moving, quasi-statically, across a simple two-bay truss.

In each frame, the colour of each member indicates the absolute magnitude of the axial force it's experiencing. Those colours are determined by the second input to bridgeMovie: let's look at how the "colour function" rainbowFun is defined:

```
rainbowFun[force_]:=
  (scaledForce = Abs[force]/18700;
  Hue[0.8*scaledForce])
```

where Abs is the absolute value function and Hue is one of *Mathematica*'s built-in colour specification functions. For the given test load of 10 kN, the forces range in absolute magitude from 0 to just under 18.7 kN, and rainbowFun is *calibrated* to output colours in the range Hue[0] to Hue[0.8]. You can see what range of colours this corresponds to using the colorPalette function:

colorPalette[Hue, {0, 0.8, 0.1}]

Of course, the forces experienced in the members vary according to the magnitude of the test load. Another function outputs the force in each member for each frame of the movie:

bridgeForces[10000] // TableForm

TableForm lays out the numbers in columns: column 1 corresponds to the member labelled 1 in the movie, and so on. You can quickly see the maximum absolute force in the table of forces using:

Max[Abs[bridgeForces[10000]]]

(i) Do you think that rainbowFun is an informative colour function? (What information should be conveyed by the colour?) Experiment with variations on rainbowFun (either redefine it, or

¹ The code is not intended for inspection here, but please do so if you're interested.

adapt it under new names), and on the following alternatives (you'll need to type these in and <shift-return> to define them):

```
monochromeFun[force_]:=
  (scaledForce = Abs[force]/18700;
  GrayLevel[0.8-0.8*scaledForce])

blueFun[force_]:=
  (scaledForce = Abs[force]/18700;
  CMYKColor[0.2+0.8*scaledForce, 0.2+0.8*scaledForce, 0, 0])

roseFun[force_]:=
  (scaledForce = Abs[force]/18700;
  CMYKColor[0, 0.2+0.8*scaledForce, 0.2+0.8*scaledForce, 0]

You can use colorPalette to examine the colour ranges of Hue, GrayLevel and
CMYKColor, for example:
```

```
colorPalette[GrayLevel, {0,0.8,0.1}]
cmykfun[x_]:= CMYKColor[0, x, x, 0];
colorPalette[cmykfun, {0.2, 1.0, 0.1}]
```

The maximum allowed range for the arguments of GrayLevel and CMYKColor is 0 to 1.

(ii) (optional) Suppose that the maximum safe axial force (tensile or compressive) in any member is 35 kN. By running bridgeMovie for different inputs of the test load, get an estimate for the maximum safe load that can traverse the structure by devising a colour function that will signal when any member experiences an unsafe force. You may want to think about the idea of piecewise-defined functions—see the notes for Session 1.

(iii) (optional) Apply your colour functions to a consideration of the "safe load" problem (with the same maximum force per member) for a four-bay, 13-member truss, whose movie and forces are generated by the commands bridgeMovie2 and bridgeForces2.

Summary of the Rainbow Bridge functions

bridgeMovie: Creates a 20-frame animation of a test load traversing a two-bay truss. Examples: bridgeMovie[10000, rainbowFun], for a test load of 10000 N and "colour function" rainbowFun; bridgeMovie[10000, rainbowFun, 30], the same, but for a movie with 30 frames.

bridgeForces: Outputs a list of the axial forces in the members for each frame of bridgeMovie. Examples: bridgeForces[10000], for a test load of 10000 N; bridgeForces[10000, 30], the same, but for a 30-frame movie.

bridgeMovie2: Creates a 30-frame animation of a test load traversing a four-bay truss. Same arguments as for **bridgeMovie**.

bridgeForces2: Outputs a list of the axial forces in the members for each frame of **bridgeMovie2**. Same arguments as for **bridgeForces**.

colorPalette: Displays the range of colours output by a given *Mathematica* colour specification function. Example: **colorPalette[Hue, {0, 1, 0.1}]**, the colours output by the **Hue** function, for the input range 0, 0.1, 0.2, ..., 1. The other specification functions available are: **GrayLevel**, **CMYKColor** and **RGBColor**.

rainbowFun: A sample "colour function" for use with **bridgeMovie**. *Calibration* of colour functions is essential; **rainbowFun** is initially calibrated for forces ranging in magnitude up to 18.7 kN. You can recalibrate **rainbowFun** by overwriting the initial definition (or by defining a new function).