6 Lab 2CS 122 Computation Lab II Winter 2011

Directions and Problems

6.1 Overview

There are two parts to this lab. Part 1 has you creating a series of user-defined functions to draw things. User-defined functions provide a way to provide short-cuts to typing in Maple commands that can be customized to the particular use by giving different values for the parameters. Daisy-chaining function together also provides another way to develop code incrementally. Pieces can be tested and debugged individually and then connected together, like Lego.

Part 2 is our first experience with iteration -- getting a computation of many steps performed by getting the computer to repeat a short segment of code. Rather than being strictly repetitive, the code is written so that it does something different each repetition, leading to an interesting cumulative result. The code you will be adding to uses for and while to control the repetition.

6.2 Pre-lab preparation

1. Read chapters 12 and 13 (new material). Review prior material (chapters 1-11) as necessary.

2. Take the pre-lab quizlet 2 at the CS 122 Maple TA web site. The deadline for doing quizlet 1 will be 7:30am the Monday that lab week starts. Note: this is different, and not as generous, as the deadline for quizlet 1.

3. If you are feeling venturesome, try Part 0 of this lab on your own. It will save you time in the lab itself. Perform the training tasks of Part 0 of this lab on your own before the lab. You will be asked to demonstrate at least one of these tasks for the staff, for credit, at the start of the lab.

6.3 Directions for this lab

1. Form a lab group of two or three people. You are not allowed to work on your own without obtaining prior permission from the lab instructor. Group members should introduce themselves to each other if they haven't already met.

2. Listen to the instructor's overview of this week's lab. (15 minutes)

3. Get your copy of the verification sheet and check out which parts of the work will be verified.

4. Work on Part 0 (15 minutes). This is a warm-up for the real lab work, to have you try out the following concepts: XXX.

5. Work on Part 1 (30 minutes). We would like to see everyone end up with individual copies of the solution scripts. However, it may be more efficient this time to work as a pair at a single computer, and distribute the scripts to both partners via email or other file exchange at the end. Your work should use only textual versions of Maple operations. Avoid using the clickable interface to perform calculations. You will not receive credit for answers that invoke operations such as solve through the clickable interface. However, you may use the clickable interface for entering expressions (e.g. square roots) or symbolic constants (e.g. π).

6.4 Problems -- Part 0 (15 minutes)

Your instructor will select some of these exercises for you to demonstrate to the staff for credit. You can prepare for this by practicing ahead of time. You can prepare for this by practicing ahead of time. Time permitting, some of the work will also be demoed at the beginning of the lab.

A note about debugging syntax errors in code edit regions: messages that begin with Error, ... often leave the cursor blinking in the code edit region at the point where the error was detected. Before you move the cursor by clicking on the worksheet, scroll so that you can see the cursor in the code edit region. This can often speed up the search for a mistake such as missing comma or parenthesis, or a wrong symbol (:= instead of :=, fot instead of for). The cursor position isn't always a reliable indicator of where the mistakes are (particularly if you made several before you tried to execute the region) but it's the best place to start looking.

Problem 0.1

In this part, we practice more with the definition and use of user-defined functions as a way of creating parameterized segments of code that can be reused. The extra work involved in function creation is outweighed by the benefit in the ease of re-use. Parameterization, which allows substitution of new data into the code before it is executed, is key to making functions a kind of re-use that is value.

We build a function that displays two vertical lines \( n \) units apart. The location of the first line is defined by two parameters \( x \) and \( y \). The first line runs from \((x,0)\) up to \((x,y)\). The second vertical line runs from \((x+n,0)\) to \((x+n, y)\).

We can define each line using the line operation of the plottools package. To draw both lines at once, we use the display operation of the plots package. Rather than use the full name plottools[line] or plots[display], we first do with(plots): and with(plottools):.

(a) In a code edit region, type in this script and execute it. You will have to resize the region to be 800 x 400 pixels in order to see the code properly. You should see the two lines.

```maple
# define and execute a user defined function
# that draws 2 vertical lines on the same graph that are "n" units apart.

# 1st, bring in the necessary Maple plotting modules
with(plots); # for basic plotting and display
with(plottools);# for line plotting

# Now define the function
# It will need an x and y value for the initial line and a value for n - the gap
# between the 2 lines
# Note - with plots explicitly defined, it is not necessary to use plots[display] form

Draw2lines:= (x,y,n) -> display([line([x,0],[x,y]), line([x+n,0],[x+n,y])],color=red);

# Finally, call the function.
# this call should draw 2 vertical lines at x = 3 and x = 7 (4 units apart) with
# heights (y) = 5 units

Draw2lines(3,5,4);
```
(b) Change the function definition so that it draws the lines in blue instead of red. Draw vertical lines that are six units high where the bottom of the first line is at (2,0) and the bottom of the second line is at (7,0).

**Problem 0.2**

We practice writing code similar to that found in Part 2 of the lab.

(a) First, initialize two tables, use a loop to put items in them, and then convert them to a list. The variables xtabList and ytabList will have the items you put into the tables, in indexed order.
#initialize xtab and ytab to be (initially empty) tables
xtab := table(); # declare the Maple table for storing the x coordinate values
ytab := table(); # table for y coordinate values

#Define a parameter that describes how many items should be in the table
numpts := 10;

#Run a repetitive loop that puts 1 into xtab[1], 2*i into ytab[1], 2 into xtab[2],
# 2*i+1 into ytab[2].... 10 into xtab[10], 2*i+10 into ytab[10]
for i from 1 to numpts do
  xtab[i] := i;
  ytab[i] := 2*i;
end do;

#Convert the tables into lists in ascending order of index.
xtabList := convert(xtab,list);
ytabList := convert(ytab,list);

Sample output for (a)

```
10
1
2
3
4
5
6
7
8
9
10
12
14
15
16
18
19
20

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

(b) Next, put a colon on the table initialization statements, the initialization of numpts, and the "end do" to suppress output from execution of these statements, leaving only the table conversion. Note that putting a colon at the end of a loop suppresses the output (except for prints or printf) from everything going on inside the loop.

Sample output for b)

```
10
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

(c) Now add code to the code edit region, to draw a graph with a title and labels, using the values in xtabList as the x coordinates, and ytabList as y coordinates.

Test code for (c)
#graph points generated by a for loop
#initialize xtab and ytab to be (initially empty) tables
xtab := table(): # declare the Maple table for storing the X coordinate values
ytab := table(): # table for Y coordinate values

#Define a parameter that describes how many items should be in the table
numpts := 10;

#Run a repetitive loop that puts 1 into xtab[1], 2*1 into ytab[1], 2 into xtab[2],
# 2*2 into ytab[2]..... 10 into xtab[10], 2*10 into ytab[10]
for i from 1 to numpts do
    xtab[i] := i;
    ytab[i] := 2*i;
end do:

#Convert the tables into lists in ascending order of index.
xtabList := convert(xtab, list);
ytabList := convert(ytab, list);

#graph points

t := "Graph of y = 2 * x"; # title for graph
L := ["X axis", "Y axis"]; # labels for plot

plot(xtabList, ytabList, title=t, labels=L);

Sample output for (c)
6.5 Problems -- Part 1 (45 minutes)

Problem 1

A function for drawing a red box

We are going to develop a function that draws a box of any specified size and location. We do so incrementally rather than doing all the coding at once, so we can write and then test one piece at a time. This is an example of the incremental code development techniques that the course teaches.

a) Open Lab2StarterPart1-1.mw, which should be contained among the downloadable files for this lab. It contains a code region that contains part of the code for drawing a box whose right hand corner is at (0,0). Read about the line function in the plottools package to figure out how this function works. Then complete the code region to draw a box that looks like this.

Once you understand the basic actions that draw a box, we can move onto incorporating this into a function.

b) After you have been successful at drawing the square, add where indicated the definition a function called drawBoxA. After it is defined, drawBoxA can be invoked to draw a red box whose height and length are specified as numbers provided as arguments to the function. Here's a brief recapitulation of how to design and implement a function:

1. First describe and name the inputs (also called parameters) to the function. You can give them arbitrary names, but typically names suggestive of the purpose of the inputs is chosen.

   For example, for drawBoxA, there should be two inputs. We could name them height and width, or h and w, depending on our bent. These names do not have to do anything with any variables that we are using in our session. They are names of placeholders. For example, in mathematical functions, they may describe a function as "f(x)", but there is no expectation that the only way to use the function is to use x. You could talk about f(5) or f(a), or f(y+1). The initial "x" is just the placeholder name you are giving for first input to f.

2. Next describe the result or output of the function based on the inputs.
For \texttt{drawBoxA}, you could say that this function should produce as a result "a plot structure that is a box whose left bottom corner is at (0,0) and has width w and height h".

3. The next step is to write the code that produces the result, using the names for the placeholders you have decided on. The coding of the function definition in Maple is always given in the general form:

\texttt{function name := ( argument, argument , ... argument) -> expression involving the argument.}

For the case of \texttt{drawBoxA} the function definition could look like this:

\texttt{drawBoxA := (height, length) -> plots[display]( [ line( ...), line( ...), line( ...), line( ...) ])}

The expression to the left of the arrow does create a plot structure, using display's ability to take four plot structures (the lines) in a list and draw them all together. We expect to see the parameters \texttt{height} and \texttt{length} to show up in the expressions for the individual lines. If we chose different names for the parameters:

\texttt{drawBoxA := (h,l) -> display([ line( ...), line( ...), line( ...), line( ...) ])}

then we would expect the innards of the line expressions to include mention of \texttt{h} and \texttt{l} in the appropriate way.

Either way should define the \textit{same function that does exactly the same thing when it is used}. The function definition does not make any displaying occur right away. That only happens when the function is \textit{invoked} by writing an expression that provides \textit{actual values} for the arguments. For example, if we had already entered the code to define \texttt{drawBoxA}, then on a later line we could enter

\texttt{drawBoxA( 5, 6)}

and we would expect a plot structure to be created that was that of a red box with height 5 and length 6.

After you have defined \texttt{drawBoxA}, uncomment the first test of \texttt{drawBoxA} at the bottom of the code region and re-execute the region. In addition to the original box, you should now see another, smaller, box drawn.

c) Uncomment more of the tests of \texttt{drawBoxA} and see that they also work as they should. If not, then fix your problems.

d) Create another function \texttt{drawBoxB} that takes five arguments -- the length, width, the \((x,y)\) coordinates of the bottom left hand corner and a string that describes the color (see Maple's on-line help for \texttt{colornames} for a complete list). Uncomment the first test for \texttt{drawBoxB} and get it to work, then run the rest of the tests.
e) Once you have convinced yourself that your definition of \texttt{drawBoxB} works, write more code that uses \texttt{drawBoxB} and \texttt{display} to draw the following pictures. Using \texttt{drawBoxB} to draw all the boxes should be a lot more convenient than copying a lot of code that includes multiple lines.

\textbf{Art project 1}

\begin{center}
\includegraphics[width=0.5\textwidth]{art_project_1.png}
\end{center}

\textbf{Art project 2}

\begin{center}
\includegraphics[width=0.5\textwidth]{art_project_2.png}
\end{center}

\textbf{Art project 3}

\begin{center}
\includegraphics[width=0.5\textwidth]{art_project_3.png}
\end{center}
In the next lab we'll take this a little further by simulating a particle bouncing around in the box.

6.6 Problems -- Part 2 (45 minutes)

A chemical reaction involves four chemicals, A, X, Y, and B. B is the product, A is an initial "ingredient", X is a catalyst, and Y is an intermediate result. The reaction rates are moles/second.

<table>
<thead>
<tr>
<th>Reaction step</th>
<th>Reaction</th>
<th>Contribution to reaction</th>
</tr>
</thead>
</table>
| 1             | \( A + X \rightarrow 2X \) | \[
\frac{d[A]}{dt} = -k_1 \cdot [A] \cdot [X]
\]
\[
\frac{d[X]}{dt} = k_1 \cdot [A] \cdot [X]
\]
| 2             | \( X + Y \rightarrow 2Y \) | \[
\frac{d[X]}{dt} = -k_2 \cdot [X] \cdot [Y]
\]
\[
\frac{d[Y]}{dt} = k_2 \cdot [X] \cdot [Y]
\]
| 3             | \( Y \rightarrow B \) | \[
\frac{d[Y]}{dt} = -k_3 \cdot [Y]
\]
\[
\frac{d[B]}{dt} = k_3 \cdot [Y]
\]

We can approximate what happens in a process driven by this reaction through a computer script. To set things up, we do initialization that

a) Defines initial concentrations of the four chemicals.
b) Gives values for the constants \( k_1 \), \( k_2 \), and \( k_3 \). Typically, we would find values for the constants by looking them up in a handbook or by determining it through experimentation and observation in the Chem lab.

The simulation would then establish four variables \( A, X, Y, \) and \( B \) that are initialized to the initial concentrations. Then it would conduct a loop

of \( n \) time steps. Each time step would establish the most recent values of the four concentrations as an add-on to the previous values, using the rules:

new \( A = \) previous \( A - k_1 \ast \) previous \( A \ast \) previous \( X \)

new \( X = \) previous \( X + k_1 \ast \) previous \( A \ast \) previous \( X \ast \) previous \( Y \)

new \( Y = \) previous \( Y + k_2 \ast \) previous \( X \ast \) previous \( Y \ast \) previous \( Y \)

new \( B = \) previous \( B + k_3 \ast \) previous \( Y \)

Note that we need eight variables \( A, \) new\( A, X, \) new\( X, Y, \) new\( Y, B, \) and new\( B \) because we need the previous values around while we do a full round of computations of the new values. Once we have completed the computations, we can transfer the new values back into \( A, \) \( X, \) \( Y, \) and \( B. \) This sets things up for the next repetition of the for computation.

**Problem 2.1**

Open *Problem2-IStarter.mw*. Execute the script.

Complete the script so that it successfully updates \( A, X, Y, \) and \( B \) and plots the concentration of \( A. \) Once you have things working, change the number of time steps so that \( n = 50 \) rather than 10. You should see something like this:
**Problem 2.2**

Extend the script so that it simultaneously plots the concentrations of A, X, Y, and B. Modify Fplot again so that it allows you to vary colors, symbols and a legend. Figure out how to use display to combine together the plots of the individual chemicals. A good multiplot clearly differentiates between the different chemicals by color and symbol choice. In addition, there is also a legend that describes which chemical each symbol is describing. The symbols are large enough so that you can tell the differences in their shapes. Here is a fragment of a suggested way to draw the combined information, although you are free to design other ways that also present the information well.

<table>
<thead>
<tr>
<th>Fragment of a multi-part graph for the chemical reactions</th>
</tr>
</thead>
</table>

This a multi-plot of several plots multiplotted with display. Each plot allowed specification of the data, the color of the plot, the symbol used for the points (e.g. diamond, cross, etc.), and the name used in the legend appearing beneath the plot (e.g. "A", "B", "X"). You can look up how these things work in the Maple documentation for plot options. The multi-plot display combines these plots together as the list which is the first argument to `display`. Additional arguments to `display` include the graph title and the axes labels. `scaling=constrained` or `scaling=unconstrained` can also be given as optional arguments to `display`.

Answer the following questions: Describe what happens to the four chemical products. Which chemical eventually has the highest concentration? How many simulation steps does it take for "equilibrium" to be reached? How much time (in seconds) does the simulation predict this will take to happen?

**6.7 Final actions (end of class)**

Upload copies all of your work to Blackboard, or email copies to yourself and/or your partners. Be sure to get credit for doing this on the verification sheet before you leave. If you cannot complete the work in the lab period, talk to the instructor before you leave about whether you can get credit for anything beyond what you finished.
6.8 Summary and conclusion

In this lab, you practiced more with user-defined functions. Use of functions is separated into two parts: defining the function, and using (invoking) it. The function definition sets up a series of parameterized series of actions to perform. Invoking the function allows you to perform those actions on whatever values or structures are presented to the function when it is invoked. This allows you to reuse the coding (the function definition) and have different results by invoking the function with different values for the parameters.

Secondly, you had your first experience in using and writing code that performs repetition through "for". We used this idea to construct a time-step simulation, which calculates how quantities in a system change over time given the basic mathematical relationships of the system. A key idea is that certain variables are reused and changed during the repetition, even as other structures (tables) record all the values.

You have had practice working with the execution trace, and troubleshooting error messages. You have learned about printing using print, and suppressing portions of the execution trace by using a colon (:) at the end of a statement.

Thirdly, you have modified the Blammo trajectory-calculating code to reflect a different mathematical model, which takes into account air resistance. While the overall architecture of the resulting code is very similar to the original Blammo, the results are noticeably different. If you were responsible for the safety of a real human cannonball, you might prefer the more complicated model because if you found it to produce more accurate results. Working on the modified Blammo code gave you additional experience with software development in code edit regions.