## Contents

1 Lab 1: CS 121 Computation Lab 1 Fall 2011 Directions and Problems ......................................................... 1
   1.1 Overview .................................................................................................................................................. 1
   1.2 Instructor-led introduction to Lab 1 (20-25 minutes) ............................................................................. 1
   1.3 Part 1 -- overview (20 minutes) ........................................................................................................... 1
   1.4 Part 1 -- problems ................................................................................................................................. 1
   1.5 Instructor-led introduction to Maple TA (20 minutes) ......................................................................... 4
      Notes on Maple TA ................................................................................................................................... 4
   1.6 Problems -- Part 2 (30 minutes) ........................................................................................................... 5
   1.7 Saving your work (5 minutes) .............................................................................................................. 5
   1.8 Final actions (End of class) .................................................................................................................. 6

2 Lab 2: Cs 121 Computation Lab 1 Fall 2011 Directions and Problems .......................................................... 7
   2.1 Lab 2 Overview ....................................................................................................................................... 7
      General directions for this lab .................................................................................................................... 7
   2.2 Instructor's demonstration of word processing, scripting ..................................................................... 7
   2.3 Part 1 ....................................................................................................................................................... 8
      Problem Description ................................................................................................................................. 8
   2.4 Part 2 ....................................................................................................................................................... 12
      Part 2 Description .................................................................................................................................... 12
   2.5 Final actions (end of class) .................................................................................................................. 13
   2.6 Concluding remarks ............................................................................................................................. 13

3 Lab 3: Cs 121 Computation Lab 1 Fall 2011 Directions and Problems .......................................................... 15
   3.1 Lab 3 Overview ....................................................................................................................................... 15
      Directions for this lab ................................................................................................................................ 15
   3.2 Instructor's demonstration ..................................................................................................................... 16
   3.3 Part 1 ....................................................................................................................................................... 16
   3.4 Part 2 ....................................................................................................................................................... 20
   3.5 Final actions (end of class) .................................................................................................................. 21
   3.6 Conclusion .............................................................................................................................................. 21

4 Lab 4: Cs 121 Computation Lab 1 Fall 2011 Directions and Problems .......................................................... 23
   4.1 Lab 4 Overview ....................................................................................................................................... 23
      Directions for this lab ................................................................................................................................ 23
   4.2 Instructor's demonstration of definition of functions, advanced plotting, and animation .................... 23
   4.3 Introduction to the "Human Cannonball" simulation ............................................................................. 23
   4.4 Problem 1 .............................................................................................................................................. 28
   4.5 Problem 2 .............................................................................................................................................. 28
   4.6 Problem 3 .............................................................................................................................................. 29
   4.7 Problem 4 .............................................................................................................................................. 31
   4.8 Final actions (end of class) .................................................................................................................. 32
   4.9 Conclusion ............................................................................................................................................. 32
   4.10 Acknowledgements ................................................................................................................................ 33
   4.11 Attachment: Lab4StarterScript ........................................................................................................... 33

5 .................................................................................................................................................................. 37
   5.1 Overview .............................................................................................................................................. 37
   5.2 A brief overview of autonomous car control ......................................................................................... 37
   5.3 Pre-lab preparation .................................................................................................................................. 38
   5.4 Part 0 -- Prelab prep ............................................................................................................................. 39
      Problem A - Learn how to enter a short script into a code edit region and execute it ............................. 39
List of Tables

Table 11.1: A table with three rows and two columns ............................................................. 93
Table 11.2: Sin Plotter ................................................................................................................ 94
Table 11.3: Sin Plotter ................................................................................................................ 95
1 Lab 1CS 121 Computation Lab I Fall 2011
Directions and Problems

1.1 Overview

This lab introduces the use of Maple, the primary computer language used for this course. You will learn how to do simple arithmetic calculations, as well as annotated plots. A ecology management problem is introduced that can be solved with the calculational facilities introduced. This interactive way of working with a computer is a skill that transfers to any number of similar technical computation systems, such as Mathematica, Matlab, or MathCAD.

This lab also introduces Maple TA, the primary homework/quiz/exam site for the course. You will log onto Maple TA with your personal account, and taking a practice quiz. Starting next week, there will be required and graded work on Maple TA for you to do.

1.2 Instructor-led introduction to Lab 1 (20-25 minutes)

The instructor will introduce themselves and present a brief overview of course, Maple, and the lab.

The lab staff will hand out verification sheets along with paper copies of these directions. In later weeks, these directions will be posted online and can be read from your lab computer. The verification sheets will still be passed out, to be the permanent record of your attendance and accomplishments during the lab.

1.3 Part 1 -- overview (20 minutes)

1. Sit down with your lab partner and if you haven't previously met, introduce yourself to them. Write both of your names down on the verification sheet in the space provided.

2. All of the partners should log onto a computer, following the demo given by the instructor in the introduction.

3. Do the calculations below. Everyone should try doing the computations on their own computer. To gain more confidence that you are getting the right answer, look at what your partners are getting. Get their help if they appear to be more successful than you. Sometimes just talking about what problems you are facing may produce useful insight towards overcoming them. If there is a problem that you can't collectively resolve, call the lab staff over and get some help.

4. You are to do all of the steps below. Some of the answers should be transcribed onto the verification sheet as indicated, for grading by the staff. Have a staff member come over to sign the verification sheet for part 1. Be prepared to show your work to the staff member, and to explain how you got your answers. This is also the opportunity to clear up any questions or uncertainties you may have even after doing the work.

5. When you complete part 1, get a staff member to verify your work before moving onto part 2.

1.4 Part 1 -- problems

1.a) Get Maple to calculate the sum of $2+2$. Presumably you will be able to tell whether or not you got the right answer pretty easily.

b) What is exact fraction you get from adding together $\frac{1}{2}$, $\frac{1}{3}$, and $\frac{1}{4}$? What about the sum of $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$, $\frac{3}{2}$, $\frac{4}{3}$ and $\frac{5}{4}$? Note that if you are doing a calculation that is highly similar to a previous one, cutting and pasting can save you some effort entering the second expression.

2. Use Maple to perform the following exact calculations. To enter $\pi$, you can select the letter from the Common Symbols palette on the left hand side of the Maple window (it's a few segments below the Expression palette). Note that Maple does not regard $\Pi$ as the same as $\pi$. To enter $e$, ...
the base of the natural logarithm, use the \( e^a \) from the expression palette, or the \( e \) from the "Common Symbols" palette. Typing "e" from the keyboard unfortunately does not produce the same result -- that kind of \( e \) Maple will regard as a symbol for an algebraic unknown like \( x \) or \( y \).

a) \[
\frac{1}{2} + \frac{1}{3} \cdot \frac{47}{42} \cdot \frac{2}{3 \cdot 5 \cdot 13}
\]

b) \[
sin \left( \frac{\pi}{3} \right)
\]

c) \[
\sqrt{\ln(e^{1024})} \quad (You \ should \ get \ 32.)
\]

d) \[
\sqrt{1 + \frac{2}{5} + \frac{3}{\left( \frac{15}{13} \right)}} \quad (You \ should \ get \ 2.)
\]

e) \[
\log_{55} \left( \sum_{i=0}^{10} i \right) \quad (You \ should \ get \ 1.)
\]

3. Powerball is a lottery that requires the player to choose six numbers. Here are the rules, as given by the Powerball site:

Powerball® is a combined large jackpot game and a cash game. Every Wednesday and Saturday night at 10:59 p.m. Eastern Time, we draw five white balls out of a drum with 59 balls and one red ball out of a drum with 39 red balls. The jackpot ... is won by matching all five white balls in any order and the red Powerball ... (http://www.powerball.com/powerball/pb_howtoplay.asp)

Determine the odds of winning the jackpot. In order to figure out the odds of winning at powerball, you need to multiply the odds of choosing the five numbers correctly, as well as choosing the powerball number correctly. For example, if the chances of picking the five numbers correctly were one in 10,000, and the chances of picking the powerball number one in 39, then the chances of doing both of them would be \( 10000 \cdot 39 = 390000 \).

You can use the "choose" function from the Maple expression palette.

\[
\binom{a}{b} \quad \text{means "the number of ways you can choose b things from a things". For example, if the lottery asked you to pick three numbers from the numbers from 1 to 6, the chances of winning would be 1 out of}
\]

\[
\binom{6}{3} = 20.
\]

4. Calculate \( 2^3^4 \). Note that \( (2^3)^4 = 8^4 = 4096 \). Why doesn't Maple give that as its answer?

5. Get Maple to reproduce this plot.

\[
\log_{10} \left( \left\| \sin \left( \frac{1}{x^2 + 1} \right) \right\| \right)
\]
6. You should use the right-click->plots->Plot Builder menu to specify things such as the plot range, the plot color, etc. Get Maple to reproduce this plot exactly, including the color, the line style, and dashes, the proper horizontal and vertical ranges and labels, and correct title and caption. In order to receive full credit, you will need to do everything letter-perfect.

\[(x - 1) \cdot (x - 2)^2 \cdot (x - 5) \cdot e^{\frac{x}{10}} \]


1.5 Instructor-led introduction to Maple TA (20 minutes)

1. The instructor will give a brief demo of how to use Maple TA, including how to log in, and how to take simple quizzes. (5 minutes)
2. Take Maple TA quiz 0. (10 minutes).

Notes on Maple TA

1. Maple TA is a quiz-administration system running separately from Blackboard Vista and Drexel One. Your userid should initially be your Drexel One userid (e.g. egk23) and the password should be your Drexel student ID number (e.g. 10096739). Let the staff know if you have trouble logging in. You can change your Maple TA password after you log in.
2. The address for Maple TA will be given in class. Links to it will also appear on the class web site www.cs.drexel.edu/cs121/Fall2011 as well as the class site on Blackboard Vista, under "Maple TA".
3. After logging onto Maple TA, you need to select the correct class, and then the correct test to take. This will change over the year as circumstances shift.
4. After you have finished answering all the questions, you should hit the "Grade" button so that your score is recorded. If you don't do this this, Maple TA will record your answers but you will receive no credit for your work because your recorded score will remain at 0. After you have gotten your work graded, then you can hit the "Save and Quit" button to exit the quiz.
5. If you encounter any technical difficulties, you should contact the course staff by visiting the Cyber Learning Center (University Crossings 147) or on-line in the Blackboard class discussion group. If you have questions about the grading of an Maple TA assignment, you should contact your section instructor (the person listed in the schedule of courses).
6. The quiz server will only handle 150 simultaneous users and will turn away the excess, so don't wait until the last moment to take the quiz. You will be given credit for only that part of the quiz that you finish before the deadline.
7. If there is a catastrophic system failure, the deadline will be adjusted. If this happens, an announcement will be made on Blackboard and the course website.

1.6 Problems -- Part 2 (30 minutes)

Complete part 1 problems if you haven't finished. Then work on part 2 of Lab. Get verification.

1. Find the exact solution to $3 \cdot x + 5 = 0$.

$$3 \cdot x + 5 = 0 \quad \Rightarrow \quad x = -\frac{5}{3}$$

2. Find the exact solution to $3 \cdot x^2 + 24 \cdot x + c^2 = 5$ (solve for $x$).

$$3 \cdot x^2 + 24 \cdot x + c^2 = 5 \quad \Rightarrow \quad \left[ x = -4 + \frac{\sqrt{159 - 3 \cdot c^2}}{3}, \quad x = -4 - \frac{\sqrt{159 - 3 \cdot c^2}}{3} \right]$$

3. Given your answer to 2, determine values of $c$ that make the solution for $x$ a real number, not a complex quantity. This means that the solution for $x$ won't involve any imaginary numbers. (Hint: experiment with using solve on an inequality. We haven't told you about this, but like many things in Maple, what should work often does. You can find inequality symbols under the Common Symbols palette. See if you can also figure out how to enter inequalities from the keyboard!)

$$159 - 3 \cdot c^2 \geq 0 \quad \Rightarrow \quad \left[ c \leq \sqrt{53}, \quad -\sqrt{53} \leq c \right]$$


A breeding group of 20 bighorn sheep is released in a protected area in Colorado. It is expected that with careful management the number of sheep, $N$, after $t$ years will be given by the formula:

$$N = \frac{220}{1 + 10 \cdot (0.83)^t}$$

and that the sheep population will be able to maintain itself without further supervision once the population reaches a size of 80.

(a) Graph $N$ versus $t$.
(b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work.
(c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of $t$.)

1.7 Saving your work (5 minutes)

1. The instructor will demo how to save a Maple worksheet file, and how to upload the work to Blackboard (occasionally required for some labs).

2. Save your work into a .mw file. The resulting file should show up on your Desktop, although it depends on your computer's notion of current working directory. If you have problems finding the file on the Desktop and your partners can't help you, call over a staff member.

After saving the file, upload a copy of the file to Blackboard so that you can refer to it later on. (Most public computers at Drexel automatically wipe out all files created during a student session after the student logs out.)
Ask a lab staff member for a demo of this if they haven't done it already.

You can also send yourself a copy via email as an attachment. This is good for those who want to remember how they did things, or wish to look at the worksheet again after lab. If you upload the file to Blackboard, you can download it to your home computer from there.

1.8 Final actions (End of class)

1. Before you leave, get the staff to grade, sign, and collect your verification sheet. You don't get credit for the lab unless they have a score recorded for you in a signed verification sheet that they have at the end of lab. You may leave the lab after you do this. You can get partial credit for the lab if a portion of your work is verified.

2. Final grades for the course will be curved if necessary, so don't fret excessively if you don't finish but it looks like others are in the same shape. However, you should try to learn the material you don't complete in lab so that you can pass the quizzes and be ready for the next lab. Computer work at this introductory level introduces a lot of ideas and concepts that appear pervasively in subsequent work. You'll probably see next time more of what you worked on this time, so you'll have another chance to practice and improve. The down side is that you can't ignore tough details and hope that future work won't depend on them in essential ways. Come with your questions to the Cyber Learning Center (University Crossings 147) next week during office hours and talk to the consultants there.
2 Lab 2 Cs 121 Computation Lab I Fall 2011

Directions and Problems

2.1 Lab 2 Overview

Overview

This lab practices the development of re-usable multi-step scripts to solve a problem. It assumes that you have read and understood the material in Chapters 2 and 3 of the course readings: "Getting started with Maple's Document Mode", and "Technical word processing".

Before beginning lab work, you will also see the instructor demonstrate the word processing features of Maple worksheets. You will also see a demonstration of how to assign names to parameters, and how to execute a whole script as a single block through Edit → Execute → Worksheet or Selection.

Part 1 of the lab has you apply a given script to solving several versions of a problem. You will find that constructing the first version of the script is the labor-intensive portion of the work. Solving the second and third versions of the problem should be very quick, since it involves only copying and a little editing of the numerical values used for the parameters of the problem.

Part 2 has you developing your own script and applying it. You must do three "original" things: a) figure out how to create a script that solves one version of the problem, b) identify the parameters of the problem, c) edit the script to use the parameters (if you haven't done so already), and d) apply your script to the other versions of the problem by cutting and pasting.

To develop the script for Part 2, you will need to come up with a recipe for how to get the solution as a sequence of Maple operations involving assignment, solving, plotting, and taking limits as well as work through the standard difficulties of getting the proper information and instructions into the computer. Script development is something where you should expect to succeed after trial, experimentation, and troubleshooting.

General directions for this lab

1. Form a lab team of two or three members. You should all sit on the same side of your work table. You need not use the same partner as last time. 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 new concepts and Maple features.

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

4. Work on part 1 (30 minutes), and part 2 (50 minutes). We would like to see everyone end up with individual copies of the solution scripts.

5. Once you have finished your work, save a copy of your work (email it to yourself, or something similar), and get the staff to sign off on the verification sheet. Having the work verified means that you can demonstrate to the staff how to solve the lab problems and can get the work done. In general, showing up at the start of lab with a completed copy of the lab work will not result in verification, since that doesn't demonstrate that your team can do the work.

2.2 Instructor's demonstration of word processing, scripting

Scripting and script-building

After this, the instructor will quickly review the work required in the scripting portion of the lab. It is expected that you will have read Chapters 3 through 5 of the course readings before coming to lab and are already familiar with the assignment operation (what you get through "assign to name" in the clickable menu, or by entering := from the keyboard), the concept of parameters in a script. If you've worked through the examples given in chapter 5 of a script and how to build one from a problem description, you should find the work in the lab straightforward.
### 2.3 Part 1

In this problem, we ask you to create a worksheet to solve a version of a falling body problem. The problem gives you a formula relating elapsed time to velocity. From this formula, you can calculate information such as the terminal velocity achieved by the body, and the amount of time it takes to achieve a certain percentage of terminal velocity. The Maple worksheet you will write will set up the formula and then perform the calculations needed to provide the desired information.

#### Problem Description

1. We want to solve three versions of a problem.

<table>
<thead>
<tr>
<th>Version 1</th>
</tr>
</thead>
</table>
| A package of medical supplies is dropped from a helicopter straight down by parachute into a remote area. The velocity $v$ (in feet per second) of the package $t$ seconds after it is released is given by the equation $v = 24.61 \left(1 - e^{-1.3t}\right)$.

(a) Graph $v$ versus $t$.

(b) What is the horizontal asymptote of the graph?

(c) How long does it take for the package to reach 98% of its terminal velocity?

<table>
<thead>
<tr>
<th>Version 2</th>
</tr>
</thead>
</table>
| A different package (with a different aerodynamic configuration) is dropped from a helicopter. The velocity $v$ (in feet per second) of the package $t$ seconds after it is released is given by the equation \[ v = 27.47 \left(1 - e^{-1.1t}\right) \],

(a) Graph $v$ versus $t$.

(b) What is the horizontal asymptote of the graph?

(c) How long does it take for the package to reach 87.5% of its terminal velocity?

<table>
<thead>
<tr>
<th>Version 3</th>
</tr>
</thead>
</table>
| A different package (with a different aerodynamic configuration) is dropped from a helicopter. The velocity $v$ (in feet per second) of the package $t$ seconds after it is released is given by the equation \[ v = 22.47 \left(1 - e^{-1.47t}\right) \],


(a) Graph $v$ versus $t$.

(b) What is the horizontal asymptote of the graph?

(c) How long does it take for the package to reach 47.3% of its terminal velocity?

Lab 2, Problem 2.1, Version 1 Solution

CS 121

-----

(Insert Group info, section info, date info here.)

-----

Version 1

A package of medical supplies is dropped from a helicopter straight down by parachute into a remote area. The velocity $v$ (in feet per second) of the package $t$ seconds after it is released is given by the equation $v = 24.61 \left(1 - e^{-1.3t}\right)$.

(a) Graph $v$ versus $t$.

(b) What is the horizontal asymptote of the graph?

(c) How long does it take for the package to reach 98% of its terminal velocity?

Solution

Define the basic relationship between time and velocity. We use parameters $a$ and $b$ to represent the coefficients in the equation.

\[
\begin{align*}
24.61 & \quad \text{assign to a name} \\
& \quad a \quad 24.61 \\
-1.3 & \quad \text{assign to a name} \\
& \quad b \quad -1.3
\end{align*}
\]
We use the parameter $p$ to represent the percentage of terminal velocity that we want to hit.

\[ p = 0.98 \]

Enter the expression for velocity and assign it the name $velocity$

\[ a \cdot (1 - e^{b \cdot t}) \]

Plot this expression to better understand it. We do this by entering the name of the expression for velocity, and then right-clicking on the result to order up a plot. Generate the plot through right-click → **Plots** → **Plot builder** so that you specify the axes labels, plot range, etc. After the plot has been generated you can change/fix any settings by right-clicking on the plot and operating the pop-up menu, which works similar to but slightly differently from the Plot Builder.

\[ velocity = 24.61 - 24.61 e^{-1.3 t} \] (2.4)
The horizontal asymptote is the limit as $t$ goes to infinity of the right hand side of the equation. Don't worry too much if the mathematical notation for getting the asymptote seems unfamiliar; the important thing to note is that Maple can figure it out if you learn how to fill in the "lim" template from the Expression Palette.

$$\lim_{t \to \infty} \text{velocity}$$

\[24.61000000\] \hspace{1cm} (2.5)

assign to a name

$$\text{terminalVelocity}$$ \hspace{1cm} (2.6)

\[24.11780000\] \hspace{1cm} (2.7)

assign to a name

$$\text{fractionTerminalVelocity}$$ \hspace{1cm} (2.8)

Set up the target equation that equates the fractional velocity to the velocity expression, and solve it numerically. The latter "solve" is done using the right-click → Solve → Numerically Solve.

$$\text{fractionTerminalVelocity} = \text{velocity}$$

\[24.11780000 = 24.61 - 24.61 e^{-1.3 t}\] \hspace{1cm} (2.9)

solve

\[3.009248466\] \hspace{1cm} (2.10)

This value (2.10) is how many seconds it takes the falling body to attain $p \cdot 100 = 98.00$% of terminal velocity.

b) Once you have the script for Version 1 working, save the worksheet from part a) as yourNameLab2Part1-1.mw (recall from last week that this is by doing Save As from the File menu). Then save another copy of the work under a different name, by saving the worksheet as yourNameLab2Part1-2.mw. This allows you to do new work while retaining a copy of the original work.

Now change the parameter values in your worksheet to configure the script to solve version 2 of the problem. Edit the other textual information in the worksheet to reflect the second version.
Execute the new version. Check that it solves version 2 of the problem (how will you do that?). Save this second version in yourNameLab2Part1-2.mw.

In order to get credit for this part, you have to be able to show to the graders that all you did to the Version 1 script from part a) was to edit the value of the parameters, and then executed the whole worksheet. You shouldn't need to retype any of the formulas, the `solve` or `plot` commands, etc.

c) Send your version of the result for part b) to one of your partners. They in turn should send you a copy of their file. Since this part depends on all team members being at more or less the same place, team members should help each other out to get synchronized at this point.

Once you receive your partner's worksheet, open it up on your machine and edit it so that it will solve Version 3 of the problem. Highlight the sections that you changed or fixed in red. You may need to fix the commentary as well as the code if you find spelling or grammatical mistakes, inaccuracies, or unfinished sections.

Save this script as yourNameLab2Part1-3.mw. You can mail back the altered version to the partner you got Part1-2 from, so that they can see what changes you felt you needed to make.

2.4 Part 2

Part 2 Description


The normalized amplitude $A$, of the vibration of a door panel of an automobile is found to be

$$A = \frac{1}{\sqrt{\left(1 - \left(\frac{\omega}{\Omega_f}\right)^2\right)^2 + c^2 \cdot \left(\frac{\omega}{\Omega_f}\right)^2}}$$

where $c$ is a measured constant that depends on the car, $\omega$ is the number of revolutions per second of the motor, and $\Omega_f$ is the measured frequency of vibration of the door panel in cycles per second.

Consider the following three versions of the problem.

**Version 1**

We find that for a 2009 Camaro (yellow, of course), $c = 0.15$, and $\Omega_f=20$ Hz.

(a) Display a reasonable graph of engine speed (in "rpm", or revolutions per minute) versus amplitude.
(b) Calculate the speed of the engine (in rpm) for which the normalized amplitude is 2.

**Version 2**

We find that for a 2003 Mini Cooper, $c = 0.18$, and $\Omega_f=25$ Hz.

(a) Display a reasonable graph of engine speed (in revolutions per minute) versus amplitude.
(b) Calculate the speed of the engine for which the normalized amplitude is 2.7.
Version 3

We find that for a 1974 Mercury Marquis (black), $c = 0.11$, and $\Omega_f = 15$ Hz.

(a) Display a reasonable graph of engine speed (in revolutions per minute) versus amplitude.
(b) Calculate the speed of the engine for which the normalized amplitude is 1.5.

In a fresh document enter a script similar in style to that of Problem 1, to solve Version 1 of this problem. Be sure to include your name and the names of your other group members in the worksheet that you create.

Use the := operation to handle assignment of parameters in this part. You may use the "assign to a name" operation from the clickable menu to do assignment to non-parametric results, though.

One thing that you will have to think through before you start typing and clicking is how to handle the requirement that the information you is given and wanted in revolutions per minute when the formula you are given is using revolutions per second.

Another thing you will need to work on is how to establish the plotting ranges. This will not be done automatically for you since the software is not sophisticated to know what portion of the graph you'd find interesting to look at. (In other words, they haven't invented mind-reading computers yet.) We suggest experimenting with ranges until you find something satisfactory.

Save your Version 1 as myNameLab2Part2-1.mw.

Make a copy of your working script in myNameLab2Part2-2.mw and edit it to handle Version 2 of the problem. You should find that the work involved to convert the script to handle Version 2 of the problem is by editing the values of the parameters. Execute the script and check that it solves Version 2 correctly.

When you have Version 2 working, send a copy of that file to one of your lab partners, and get a copy of their version of the script from them in return. Save this as myNameLab2Part2-3.mw, and edit it to handle Version 3. Highlight the sections that you changed or fixed in red. You may need to fix the commentary as well as the code if you find spelling or grammatical mistakes, inaccuracies, or unfinished sections. You can mail back the altered version to the partner you got Part2-2 from, so that they can see what changes you felt you needed to make.

We are told that the normal operation of car engines leads them to operate in the 1500-2000 rpm range while cruising. Do you think that the drivers of these three cars will be satisfied with the vibration properties of their cars?

Do you know what TV shows or movies were these cars seen in? (No Maple, doesn't have a button which will answer that.)

2.5 Final actions (end of class)

Upload all of your work to Blackboard, or email copies to yourself and/or your partners. Before you leave, make sure that the staff has signed and accepted the verification sheet for your group so that your work is properly credited.

2.6 Concluding remarks

In this lab, you have learned how to create scripts that combine commentary and a sequence of computational actions. Scripting allows you to easily solve second and third versions of a problem once you have done the hard work of creating the script by studying how to the first version of the problem.

Because programming is a relatively expensive activity in terms of time, reusing someone else's work is normal activity in computing. Scripts need to be written in a way that makes it easy for someone else to use it and to modify it in modest ways. Of course there will be other times where you'll have the responsibility of creating something completely on your own instead of reusing someone else's work. Just keep in mind that you're writing not only for yourself but potentially for others.
3 Lab 3 Cs 121 Computation Lab I Fall 2011
Directions and Problems

3.1 Lab 3 Overview

Overview

This lab provides more practice with scripts. It assumes that you have read and understood the material in Chapters 4 through 7 of the course readings: "Assignment", "Building Scripts", "More Sophisticated Scripting" and "Using and Defining Functions".

This time you develop scripts using the "textual style" for operation entry where everything is specified from the keyboard, rather than the "calculator style" where expressions are created by filling in slots from Palette options and operations are selected by the mouse. Being fluent in this style helps prepare the way for doing "real programming" where you specify a block of operations to be done all together in a batch. The jump to textual entry of an entire Maple operation takes some getting used to, but the greater power of expression allows you to do even more sophisticated technical problems.

Maple extends the style of mathematical functions -- f(x), g(3,5), etc. -- to specify not only mathematics, but also computations. Thus solve, and plot are written textually as functions solve(....), and plot(....). Becoming proficient in reading and writing computer instructions in this style is a major step in learning how to program that transfers across the many different programming languages that use this style.

To add to the functional frame of mind, we also explore how you can create and name your own functions in Maple.

This lab has also introduces you to non-numeric data structures: lists and strings. Having strings allows you to compute with text rather than just with numbers. Lists are a way of collecting and accessing aggregations of results.

Before beginning lab work, you will also see the instructor demonstrate how to use these elements. This should be a recapitulation and time to clarify your own explorations with these features before lab.

In Part 1 of the lab, you apply a given script (written in textual form) to solving various versions of a problem. You then are presented with a variation of the problem and asked to figure out how to modify the script to solve the variant. This will require a bit of original thought -- it can't be handled just by changing parameter values.

In Part 2, you will study how to do data fitting -- find a formula that provides a mathematical description of measurements collected. Once you have the formula, you can use it to answer additional questions and make predictions about the situation that was measured. This part also requires the use of textual operations, as data fitting is not available under the clickable menu.

Directions for this lab

1. Find a lab partner. You need not use the same partner as last time. 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 new concepts and Maple features.

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

4. Work on part 1 (30 minutes), and part 2 (40 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.
3.2 Instructor's demonstration

Textual entry of Maple commands, lists, sets, and plots

The instructor will demonstrate:

How to invoke `solve` and `plot` with textual entry of the function, including the use of lists and strings. "Command completion" (entering `solve` and then hitting the escape key) will be demonstrated as a typing shortcut. Ways of troubleshooting your way out of problems with textual entry will also be discussed. How to use `convert` to do unit conversion for numbers or formulas.

Use of functions and defining your own functions

The instructor will review the available functions in Maple, where to read more about them, and how to define your own. Use of a user-defined function in a script will also be shown.

Data-fitting

The instructor will demonstrate the data fitting facilities in Maple. Part 2 is basically "read the on-line documentation and experiment with the examples until you get them to work".

3.3 Part 1

You can estimate in a reasonably accurate way the shutter speed of a camera by taking a picture of a moving object. If the movement of the object is known, then the amount of blur is related to the shutter speed.

In this picture, we see a circular bicycle wheel rotating. The spokes of the wheel blur and sweep out an angle we can measure from the photograph.

![Image of bicycle wheel with angle measurement](image)

Figure 3: Photo showing motion blur cause by slow shutter speed

\( \alpha \) is the angle swept out by the spoke while the shutter is open.

We look at one of several photographs we have taken with various cameras and bicycles, and come up with the following measurements:

Case A

Wheel diameter \( d = 26 \) in

Bicycle forward speed \( v = 15 \) mph

Angle of rotation \( \theta \): 10.2 degrees
We want to calculate the shutter speed \( s \), which we expect to be a time in seconds, e.g. \( s = .017 \) means that the shutter is open for .017 seconds (17 milliseconds). The terminology of "shutter speed" does not refer to the velocity of the shutter as it moves when taking a snapshot, but rather to the length of time the shutter lets light into the camera.

1. Work out the math steps that calculate \( s \) for case A.

It may not be immediately clear to you how to do this. Consider the following possible calculations. For those that you find useful, use Maple to do scratch calculations. Remember to assign useful results a name through assignment. For example, for part d) below, you might do \( c := \text{Pi} \* d \).

a) If \( v = 15 \) miles per hour, how long (in hours) would the bicycle take to travel 5 miles, if it were rolling down a road at that speed? What about 1 mile? What's the formula for the time to travel \( m \) miles at that speed? Is it \( m \cdot \frac{15}{15} \), \( \frac{m}{15} \), or \( \frac{15}{m} \)?

b) If you knew the velocity in inches per second was \( v\text{ips} \), how long (in seconds) would the bicycle take to travel 100 inches? What's the formula for the time to travel \( i \) inches at that speed? What unit of time is the travel time measured in if you compute using \( v\text{ips} \).

c) How can you calculate \( v\text{ips} \) from \( v \) using the Maple \text{convert} operation? Your instructor should have demonstrated this during their overview talk.

d) What is the circumference of the bicycle wheel? Recall how to enter \( \pi \) : from the Greek or Common Symbols Palettes, or textually, \( \text{Pi} \). Note that \( \text{pi} \) does not refer to the mathematical constant. Maple pays attention to whether letters are upper- or lower-case. It has to be \( \text{Pi} \).

You will surely have to use this quantity in your solution. Call it \( c \).

e) How far does the bicycle travel if it were rolling for one wheel revolution?

f) How long does it take, in seconds, for the bicycle to travel that distance?

g) What fraction \( f \) of a whole revolution is the blur angle \( \theta \)?

h) How long does it take for the bicycle to travel the distance which is that fraction of a revolution?

i) How is \( s \) related to the answer you got for h)?

Write down on the whiteboard your steps in calculating the answer. You don't need to use all of a) through h), but some of them will probably be part of what you write down. Be prepared to provide the staff with justification that this answer is correct.

Keep it real -- what justification would you be using to convince your boss that the answer was right if you had no other group around to compare answers to? "Because Maple did it" doesn't provide good justification, since it only does what you tell it to, and you might have told it to do the wrong thing.

2. If you have not already done so, get all the steps into a Maple worksheet, and have it calculate the answer for Case A. What is your answer for \( s \)? Write your answer down on the whiteboard. Provide the staff with justification that this answer is correct.

3. In a fresh worksheet, organize your ideas to create a script that can do the calculation for any given diameter, speed, and angle. Put the script into standard form with the parameters first. Only the "starting information" \( d,v, \) and \( \theta \) are parameters here.

Your worksheet should have this general form;

<table>
<thead>
<tr>
<th>CS 121 Lab 3 Shutter speed script</th>
</tr>
</thead>
<tbody>
<tr>
<td>By .....</td>
</tr>
<tr>
<td>Section .....</td>
</tr>
<tr>
<td>Instructor: .....</td>
</tr>
</tbody>
</table>

Unassign any existing assignments in the session.
**Begin parameters**

Unassign any existing assignments in the session.

Velocity (in miles per hour)

\[ v := \ldots \]

Wheel diameter (in inches)

\[ d := \ldots \]

Blur angle (in degrees)

\[ \theta := \ldots \]

**End parameters**

Compute circumference from diameter

\[ c := \ldots \]

Compute velocity in inches per second

\[ \text{vips} := \ldots \]
etc. etc.

Compute shutter speed from .... This is our answer

s := ..... 

End Script

For the last step, add a little post-calculation tidying up: use the right-clickable menu on the result for s, and apply numeric formatted (fixed) on the last result, so that it displays an appropriate number of significant digits. From Engr 101 considerations, you should be able to decide on your own how many digits that should be.

Double check that you using the correct script format, and have provided enough information for the description of each step. Your grade for this part of the lab is based in part on how closely you follow the intent to have enough information for others to understand who did the work and how you did it, and that the script is organized so that anyone will be able to use it to calculate a different version of the problem.

We expect that what you will have is a script with the values of the parameters set to calculate the solution for Case A. Check that it’s the same as your computed before in step 2. Save the worksheet as YourNameLab3-A.

4.

Swap copies of your worksheet with one of your lab partners or some one nearby. Edit it appropriately and save it as YourNameLab3-B.mw. Edit and re-execute the worksheet to handle the following case:

**Case B**

Wheel diameter d = 25.5 in

Bicycle forward speed v = 18 mph

Angle of rotation θ: 12 degrees

Assuming that the author wrote the worksheet well, it should take you about 30 seconds to do this after you open up the worksheet. Save this as YourNameLab3-B.

5. In this step, you don't need to build a script, but you do need to do the work in a fresh worksheet. **Do New-> Document Mode and then save the blank worksheet as MyLab3-C.mw**

If you have been following the derivation of s closely, you can realize that you can build a computation that combines all the operations. We're going to build a function that encapsulates all the operations. We will name the function `shutterSpeed` (rather than "f" or "g", as a math text book might. It will take three arguments values for v, d, and θ. The result of the function will be the shutter speed, in seconds, as the scripts did.

Function definitions were discussed in chapter 7 of the chapter readings, as well as the subject of a quizlet question or two.

In a fresh worksheet, type in this function definition:

```
shutterSpeed := ( v, d, θ ) → evalf((d · π/ convert(v, units, miles/hour, inches/second)) * θ/360)
```
(As mentioned in chapter 9 of the course readings, \texttt{evalf}, is the textual way of doing approximation. By default you get a ten digit approximation.)

a) Compute \texttt{shutterSpeed(15, 26, 10.2)} and compare your result to your script's operation for case A. They should be the same.

b) Do the same for Case B.

c) If you enter the symbol \texttt{v1} instead of a number for \texttt{v}, \texttt{shutterSpeed(v1, 25, 10.2)} returns a formula involving \texttt{v1} rather than a number. Try it. This is the formula for shutter speed for an arbitrary velocity \texttt{v1}, a wheel diameter of 25 inches, and a blur angle of 10.2 degrees. If you make all of the arguments to \texttt{shutterSpeed} symbolic, e.g. \texttt{shutterSpeed(v1, d1, a1)}, what does the result that you get back mean?

d) Save your work for this part.

### 3.4 Part 2

Sometimes rather than plotting points of a function, we are given data points taken from measurements and want to find a function that would produce them. One additional issue is that the data is typically precisely accurate, there is experimental error in making the measurements. So we are satisfied if the function we derive is "reasonably close" to the data points rather than passing exactly through them. This is called the \textit{data fitting problem}.

Typically rather than searching through all possible functions to find the best fit, we look for good candidates from a particular class of functions. One class are the \textit{linear functions}: all functions \(g(x) = a \cdot x + b\) for some values \(a\) and \(b\). The data fitting problem becomes that of finding good values of \(a\) and \(b\).

There are several techniques for doing data fitting. One of the more popular is called \textit{least squares data fitting}.

The data points \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\) can of course be split into two separate lists of values, \(x\) written as \([x_1, \ldots, x_n]\) and \(y\) written as \([y_1, \ldots, y_n]\).

### Problem C, Version 1

(From Anton, Calculus 8th ed., p. 1007)

If a gas is cooled with its volume held constant, then it follows from the \textit{ideal gas law} in physics that its pressure drops proportionally to the drop in temperature. The temperature, that, in theory, corresponds to a pressure of zero is called \textbf{absolute zero}. Suppose that an experiment produces the following data for pressure \(P\) versus temperature \(T\) with the volume held constants:

<table>
<thead>
<tr>
<th>&quot;P (kilopascals)&quot;</th>
<th>134.2</th>
<th>142.5</th>
<th>155.0</th>
<th>159.8</th>
<th>171.1</th>
<th>184.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;T (deg Celsius)&quot;</td>
<td>0</td>
<td>20.1</td>
<td>39.8</td>
<td>60.0</td>
<td>79.9</td>
<td>100.3</td>
</tr>
</tbody>
</table>

We want to find values \(a\) and \(b\) so that the line described by \(a \cdot t + b\) does a good job of representing the data. Then we will use the formula we get for \(P\) to answer some questions.

1. Create a fresh Maple session through File->New->Document mode.

2. Enter two lists. Call the first list pData and assign it the numbers found in the first row of the above table: \(\texttt{pData := [134.2, 142.5, 155.0, 159.8, 171.1, 184.2]}\). Similarly, create a second list and assign it to tData.

3. Produce a point plot with Maple using the techniques discussed in chapter 6 of the course readings. Make the plot blue.

4. Look up the data fitting facility in maple by starting up Maple help and looking up "least squares". Find the examples given in the documentation page on CurveFitting[LeastSquares] and see one that will help you do data fitting using pData, and tData. Produce a formula for the line.
Notes:

(a) You will have to experiment in order to get things to work. Start by copying and pasting the instructions from the examples and getting them to work as advertised in your own worksheet. Then try substituting pData and tData for the values in the example.

(b) The "with(CurveFitting):" operation needs to be done before you can do any of the other lines in the examples.

(c) You don't have to use "v" as the variable in the curve fitting formula. It makes more sense to use "T".

(d) In the work to come, it helps to give the formula produced by the curve fitting a name, through assignment.

5. Plot the line you got from (d). Make the line blue.

6. Here's a trick to do a quick multi-plot that's not documented in the chapter readings. (a) copy the point plot to the bottom of the worksheet. (b) copy the formula plot. (c) click on the copy of the point plot. (d) right-click and select "paste". It works for a one-of plot but it doesn't lend itself to scripting very easily.

The combined plot should show the line passing close by most of the data points. If it doesn't this is an indication that something is wrong.

7. Once you have gotten a least squares formula, answer the following questions:

(a) Based on the formula, get Maple to estimate the pressure when the temperature is 120 degrees Celsius by evaluating the expression at T=120. (The eval operation is handy here).

(b) Produce an estimate for absolute zero (where pressure is zero) by solving an equation involving this formula. What is your estimate? Look up the actual value of absolute zero on the Internet and compare it with your estimate from this "virtual experiment". Include your calculated answer in a textual explanation of what you are doing, similar to the way that the target voltage was mentioned in the script for Part 1.

8. Save your worksheet for part 2 as Lab3Part2Solution.mw and mail copies of it to yourself and your lab partner. Be sure to put the names of your team on the worksheet for easy identification.

9. Re do this problem with Tools->Assistants->Curve Fitting. You will want to select "least squares" as the technique for fitting, not splines or interpolation. Be prepared to show your worksheet and the solution with the assistant to the staff for grading. Which way was easier for you to do?

3.5 Final actions (end of class)

Email (or put on a flash drive) copies of your work to yourself and/or your partner so that you have it for future reference and use.

3.6 Conclusion

In this lab, you have gotten further practice at creating scripts in Maple. We have expanded the repertoire of objects to include lists, which allow us to maintain aggregations of values in an easy-to-access fashion, and character strings, which allow computations with textual information. We have done a little data fitting, producing a formula describing a curve that approximately describes a collection of numerical measurements. We have practiced with point plotting, and "multi-plotting" (where multiple curves appear together on a single set of axes). In preparation for programming, we have begun to practice with entry of operations using text rather than mouse clicks and menu items.
4 Lab 4 Cs 121 Computation Lab I Fall 2011
Directions and Problems

4.1 Lab 4 Overview

Overview

The lab explores the problems of finding artillery trajectories that satisfy certain requirements and constraints. It assumes that you have read and understood the material in Chapters 8 and 9 of the course readings: "Programming with Functions" and "Visualization, Modelling and Simulation".

These computations use a mathematical model that describe how key values and properties change over time. It uses more sophisticated plotting and animation more rapidly achieve understanding essential to design verification or modification. As you will see, animation can lead to more rapid understanding compared to large tables of numbers or pages of graphs.

Before beginning lab work, you will see the instructor demonstrate how to use function definition, display, and animate which are key operations in the day's lab. You will be asked to write scripts using only textual versions of Maple operations. You will not receive credit for answers that use the clickable interface for operations.

To prepare for this lab beforehand

1. Read Chapters 8 and 9 of the course readings. One way to better understand the material is to learn how to reproduce the examples in the text on your own computer.

2. Study these lab directions and the lecture notes for this lab, posted on the course web site.

3. Take the pre-lab quizlet.

Directions for this lab

1. Find a lab partner. You need not use the same partner as last time. 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 new concepts and Maple features.

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

4. Work on the four problems of the lab (80 minutes). You should create a separate worksheet for each problem, named as described in the directions below. Your work should use only textual versions of Maple operations. You will not receive credit for answers that invoke operations through the clickable interface. You will not receive credit for solutions that are crammed together in a single worksheet.

4.2 Instructor's demonstration of definition of functions, advanced plotting, and animation

The instructor will demonstrate: function definition, function daisy chaining, display, paramplot, and animate.

4.3 Introduction to the "Human Cannonball" simulation

The following problem comes from the book Calculus: Early Transcendentals, 7th edition, by Howard Anton, Irl Bivens, and Stephen Davis, pages 462-465 (module created by John Rickert and Howard Anton): "Blammo the Human Cannonball will be fired from a cannon and hopes to
In this problem you will compute the equations of motion for Blammo traveling in the plane and use these equations to simulate the motion of Blammo flying towards the net. The equations of motion in the plane are similar to those that were derived and used in first tutorial; however, in this case you must track both the $x$ and $y$ coordinates of the object. Prior to shooting Blammo from the cannon, you will have to specify the angle of elevation of the cannon and the initial speed of Blammo exiting the cannon. Based on these parameters, and the distance between the cannon and the net, you need to determine whether the Blammo hits the net or not. We will initially assume that there is no resistance from the air as Blammo travels and that the only force acting on Blammo is gravity, which only affects the $y$ coordinate.

Consider an elevation angle of $\alpha$ degrees and an initial speed of $V_0$. In the triangle below, the cannon is located at point A, the angle of elevation is the angle $CAB$ and the length of the side $AC$ is equal to the initial speed $V_0$. The initial velocity in the $x$ direction is the length of the side $AB$ and is equal to $V_{0x} = V_0 \cos(\alpha)$. (Remember, SOHCAHTOA?, $\cos \alpha = \frac{\text{adjacent}}{\text{hypotenuse}}$) The initial velocity in the $y$ direction is the length of the side $BC$ and is equal to $V_{0y} = V_0 \sin(\alpha)$. 
The position of the cannonball is given by \( (x(t), y(t)) \), which provides the coordinates of the cannonball at time \( t \). The equations of motion, as derived in most elementary physics texts, can be found to be

\[
x(t) = V_{0x} t \quad \text{and} \quad y(t) = y_0 + V_{0y} t - \frac{1}{2} gt^2
\]

where \( g = 32 \, \frac{ft}{sec^2} = 9.8 \, \frac{m}{sec^2} \), depending on the units used. \( y_0 \) is the initial position of the object (if we are launching from the ground \( y_0 = 0 \)). We will use these equations for the following problem.

Let's work in the English FPS (foot/pound/second) system of units.

If we shoot Blammo off at 100 feet per second at an angle of 45 degrees, we find that \( v0 := 100 \)

\( 100 \) \hspace{1cm} (4.1)

We can develop a plot, using the paramplot feature of Maple's plots package:

\[
\alpha := \text{convert}(45 \cdot \text{degrees}, \text{radians})
\]

\( \frac{1}{4} \pi \) \hspace{1cm} (4.2)

\( v0x := \cos(\alpha) \cdot v0 \)

\( 50 \sqrt{2} \) \hspace{1cm} (4.3)

\( v0y := \sin(\alpha) \cdot v0 \)

\( 50 \sqrt{2} \) \hspace{1cm} (4.4)

\( g := 32 \)

\( 32 \) \hspace{1cm} (4.5)
We can see that after three seconds Blammo is still in mid-flight, having already reached his apex. We can solve an equation to find out at what times $t$ Blammo is on the ground.

$$\text{solve}(\text{ypos}(t) = 0, t)$$

$$0, \frac{25}{8} \sqrt{2}$$

Not surprisingly, one of the times is $t=0$ (the start). We can get the time we want by daisy-chaining `solve` (which gives a sequence of two roots) and the `max` function.
flightTime := max(solve(ypos(t) = 0, t))

\[ \frac{25}{8} \sqrt{2} \]  

(4.11)

We daisy-chain the function that calculates x position with the approximation function `evalf`. By default, `evalf` computes ten decimal digits accuracy.

\[ \text{distance} := \text{evalf}(\text{xpos}(\text{flightTime})) \]

(4.12)

Evidently Blammo travels 312.5 feet.

Now, if we plot from t=0 to flightTime, we should see the whole plot:

\[
\text{plot}([\text{xpos}(t), \text{ypos}(t), t = 0 .. \text{flightTime}], labels = ["feet", "feet"])
\]

If we want to produce an animation of Blammo flying through the air, we need to create a function that creates a plot with a shape located at (xpos(t), ypos(t)) for any given time t, and then gives it to the animate function for t=0..flightTime.

\[
\text{drawBlammo} := (t) \rightarrow \text{plot}([\text{xpos}(t), \text{ypos}(t)], style = \text{point}, color = "red") \\
\rightarrow \text{plot}([\text{xpos}(t), \text{ypos}(t)], style = \text{point}, color = "red")
\]

(4.13)
Maple has other useful functions to help us:

$maximize$ will find the largest value that a function attains. For example,

$$evalf(maximize(ypos(t)))$$

$$78.12500000$$

This means that Blammo reaches a maximum height of 78 1/8 feet. We could figure this out through calculus and/or remembering enough high school analytic geometry about parabolas, but Maple knows how to do these things without further programming on our part.

### 4.4 Problem 1

Suppose we shoot Blammo out of the cannon at an initial velocity of 110 feet per second, at an angle of 50 degrees. Read in the script Lab4StarterScript. Use the computational machinery to determine how high and how far Blammo travels. Play the animation to confirm that it is consistent with the numbers computed, as well as the parameter plot.

This is a "use the script and interpret the results" problem. You have to learn how to use the features in the script, but you don't have to modify any code or write new code.

### 4.5 Problem 2

This problem requires you to modify code. To cleanly separate your work from problem 1, save a copy of the starter script in a different file, YourNameLab4Problem2.mw, then begin the Problem 2 worksheet to solve this problem.

We want to have a different shape for Blammo. Consult the on-line documentation for plottools and look up the $disk$ and $pieslice$ functions. Choose one, and replace the point plot with a red or blue object of your choice. Assume that Blammo is roughly six feet tall.

To develop your code, modify the plotting instructions so that they produce a shape in the correct position rather than a point plot. Then change the function being given to animate to draw the shape rather than the point. Produce an animation for when Blammo is launched at a speed of 50 feet/second, at an angle of 35 degrees.
You may notice that the shape looks more squashed than it ought to be. This is because `animate` is not using the same scaling for the horizontal and vertical axes. To correct this, add the option `scaling=constrained`, as illustrated in the various examples in Chapter 9 of the readings.

Save your work as `YourNameLab4Problem2.mw`, to show to the grader.

This file should contain the solution to problem 2 ONLY. Having several solutions (possibly incorrect, and possibly interfering with each other) all in the same worksheet will just create more problems for you to have to solve.

## 4.6 Problem 3

Suppose we have a tent that's 200 feet long and 50 feet high. We buy a standard explosive charge from a manufacturer that will shoot Blammo out at a velocity of 82 feet per second. Use the computational machinery to determine what angles the gun may be set up to have Blammo safely fly through the air without running into the walls of the tent.

**Directions**

Open up the starter script again and modify it so that it eliminates the animation but adds bounding lines to the parameter plot, so that you can see whether Blammo's trajectory will exceed the boundaries of the tent.

To do this, create a function that uses `display` with some green dotted lines. plotting this will quickly establish whether Blammo exceeds the boundaries. You don't need to produce an animation.

For example,

```
with(plottools):

p1 := line([0, 50], [200, 50], linestyle = "dash", color = "green")
   CURVES([[0., 50.], [200., 50.]], COLOUR(RGB, 0., 1.00000000, 0.), LINESTYLE(3)) (4.15)

p2 := line([200, 0], [200, 50], color = "green", linestyle = "dash")
   CURVES([[200., 0.], [200., 50.]], COLOUR(RGB, 0., 1.00000000, 0.), LINESTYLE(3)) (4.16)
```
If we display them together we get a plot that looks like this:

\[ \text{with}(plots) : \]
\[ \text{display}([p1, p2]) \]

What you want to do is to generate the parameter plot and assign it to p3. Then doing \text{display}([p1, p2, p3]) should result in a picture that looks something like this.

\[ \text{display}([p1, p2, p3]) \]
This diagram shows quickly that the flight path goes quickly beyond the boundaries of the tent when we launch Blammo at 100 feet per second at 45 degrees. But you will have different results for 10 feet per second.

Use this as an idea to modify your existing scripts to handle this problem. When you find a range of angles that works, keep the execution from the largest angle that works. Save your work as YourNameLab3Problem3.mw.

4.7 Problem 4

Exchange the script from Problem 2 with one of your partners. You will now use their script to solve another problem.

We would like to add a net into our simulation, that Blammo can land in safely. Change the script from Problem 2 to add a new function drawNet:=(d, w) -> a green line centered at (d,0) with total width w. This means that the line extends from (d-w/2,0) to (d+w/2,0).

Modify your animation function so that every frame displays not only the position of Blammo but also the net. For example, here is a frame of an animation we created that has Blammo (a red disk), flying through the air towards the net.
Through trial and error, find angles and initial velocities that solve the following problems. Try to get Blammo's center point to land as close as you can conveniently arrange to the center of the net.

(a) **Distance to net = 100 feet. Size of net = 10 feet.** Try 70 feet/second and 30 degrees initially. Team members should try various values of $v_0$ and the angle to make Blammo land in the net. Be prepared to play the animation of the successful shot for the grader. Note that you can vary both the velocity and the angle, so it doesn't have to be just a patient variation of just velocity or just the angle to find a solution.

Once you find a solution, see if you can find another solution with an initial velocity 10% faster, and a different angle.

(b) **Distance to net = 200 feet. Size of net = 5 feet.** Record the velocity and angle that you found that worked.

(c) **Distance to net = 500 meters. Size of net = 3 meters.** Record how many tries it took you to find a solution that worked. In order to solve this problem, you will have to figure out how to handle metric values for the distances. However you find it, at the end of the script be sure to express the initial velocity in meters/second instead of feet per second.

### 4.8 Final actions (end of class)

Email copies to yourself and/or your partners. Be sure to get credit for doing this on the verification sheet before you leave.

### 4.9 Conclusion

In this lab, you have used a computerized mathematical model of artillery trajectories to solve problems having to do with a human cannonball. One of the advantages of using the model is that it's not necessary to spend as much money on test firings and replacement human cannonballs while you try to find what will work safely. You have extended your repertoire to include more elaborate user-defined functions that can be used to generate animated plots. You have learned how to plot objects besides lines and points. You have gotten more practice at using and modifying scripts.
4.10 Acknowledgements

This exercise was developed with the help of Dr. Jeremy Johnson, Dr. Fred Chapman, and Mr. Ryan Walls.

4.11 Attachment: Lab4StarterScript

CS 121
Lab 4
Starting Script for Blammo

This script takes as parameters the initial velocity v₀ of Blammo, and initial position (x₀,y₀). It shows a plot of Blammo's path through the air, and calculates the maximum horizontal and vertical distance achieved during his trip. It also produces a computer-generated animation of him flying through the air.

Start of Parameters

Starting velocity

\[ v₀ := 100 \]  

(4.17)

Starting positions

\[ x₀ := 0 \]  

(4.18)

\[ y₀ := 0 \]  

(4.19)

Firing angle of gun

\[ angle := 45 \]  

(4.20)

End of Parameters

Calculate the angle in radians

\[ \alpha := \text{convert}(angle \cdot \text{degrees}, \text{radians}) \]  

\[ \frac{1}{4} \pi \]  

(4.21)

Split the starting velocity into x- and y- components.

\[ v₀x := \cos(\alpha) \cdot v₀ \]  

\[ 50 \sqrt{2} \]  

(4.22)
Establish the gravitational constant (in English Units, its 32 feet per second, squared).

\[ g := 32 \]  

Define the functions for x and y position

\[ x_{\text{pos}} := (t) \rightarrow x_{0} + v_{0x} \cdot t \]

\[ y_{\text{pos}} := (t) \rightarrow y_{0} + v_{0y} \cdot t - \frac{1}{2} \cdot g \cdot t^{2} \]

Determine how long Blammo will be in the air (assuming he doesn't hit the wall of the circus tent, the man on the flying trapeze, etc.). We can get the time we want by daisy-chaining `solve` (which gives a sequence of two roots) and the `max` function.

\[ \text{flightTime} := \max(\text{solve}(y_{\text{pos}}(t) = 0, t)) \]

\[ \frac{25}{8} \sqrt{2} \]  

Calculate the horizontal distance Blammo travels. We daisy-chain the function that calculates x position with the approximation function `evalf`. By default, `evalf` computes ten decimal digits accuracy.

\[ \text{distanceTraveled} := \text{evalf}(x_{\text{pos}}(\text{flightTime})) \]

\[ 312.5000000 \]  

Determine the highest vertical position.

\[ \text{evalf}(\text{maximize}(y_{\text{pos}}(t))) \]

\[ 78.12500000 \]
Use a parameter plot to show the whole trajectory.

\[
\text{plot}([\text{ xpos}(t), \text{ ypos}(t), t = 0..\text{flightTime}], \text{ labels} = ["\text{feet}", "\text{feet}"])
\]

Create a function that draws Blammo at a moment of time, \( t \)

\[
\text{drawBlammo} := (t) \rightarrow \text{plot}([\text{ xpos}(t)], [\text{ ypos}(t)], \text{ style} = \text{ point, color} = "\text{red}")
\]

\[
t \rightarrow \text{plot}([\text{ xpos}(t)], [\text{ ypos}(t)], \text{ style} = \text{ plottools:-point, color} = "\text{red}")
\]  

Create an animation of Blammo flying through the air.
```python
with(plots):

animate(drawBlammo, [t], t = 0 ..flightTime)
```

End of script
5

5.1 Overview

At this point, we've written scripts to do mathematical calculations and visualizations -- plots and animations -- of them. Some of the calculations involved mathematical models of physical situations. We found that that parameterizing scripts gave them greater flexibility, making it easy to reuse them to solve variations of a basic situation.

In the first part of this lab, we introduce the code edit region of Maple that facilitates the construction and execution of textually-entered scripts. The Maple clickable interface that you learned originally is good for rapidly doing a "one of" calculation, as long as it involves a sequence of one-line calculations available from the clickable menu. However, when you get into more elaborate and length segments of code, it's more convenient to be able to deal with multi-line sequences of operations in one block of text as code edit regions allow.

Besides mathematical modeling, another important application of computer programming is control of devices. In this lab we introduce a Maple package that supports control of a simplified model of a car. Rather than controlling a physical car, the programming controls a "virtual car" whose movements are shown in an animation constructed when you run the simulation. The programming for the control is similar to the NXT robots used in ENGR 102 -- but you can certainly learn to control the car without any knowledge of what goes on in that course. The simulator world does not have the same difficulties with wheel slippage, sensor error, and motor imprecision that a real robot would have. This makes it easier to develop and debug program for more complicated situations. A control program for a real car in the same situation would be yet more complicated because it would also have to deal with the imperfections of real world robots. Building a more realistic simulation is feasible but we have left those details because getting good control of a car even in the idealized world of this simulator is hard enough for a short student lab.

This lab presents an experience with controller programming, using an API (application programming interface). It gives you a chance to practice your software development skills by controlling a car simulator that behaves suspiciously like a NXT robot. However, since it's a simulation, you don't have to spend a lot of time with mechanical breakdowns or resets.

5.2 A brief overview of autonomous car control

The Defense Advanced Research Projects Agency DARPA a few years ago sponsored a contest for "autonomous vehicles" -- cars that drive themselves. There were no humans inside the cars, but there is a "chase car" nearby with an emergency cut-off switch just in case. The driving range for the contests in successive years became increasingly more difficult, from open countryside, to urban traffic navigation. The winning team from Carnegie Mellon in 2007 won $2,000,000 for placing first. You can read about the 2007 contest at http://www.darpa.mil/GRANDCHALLENGE/.

The DARPA challenge spurred significant progress. Google continued this work, unveiling its self-navigating car in 2010. There are YouTube videos (e.g. http://www.youtube.com/watch?v=d0Ny-u2tjS8&noredirect=1, http://www.youtube.com/watch?v=-nYhKD8leAg&feature=related) of the Google car driving in traffic.

Since good driving depends on road, vehicle, and traffic conditions, it's clear that any autonomous vehicle could not function on a fixed strategy. In other words, it would have to carry on-board some computer programming that would look at and respond to variable conditions.

In this lab, we look at some aspects of the programming involved in a simple simulation of an autonomous vehicle. We introduce and use the concepts of procedure, Application Programming Interface (API), and control loop to guide the car. In the next lab, we will introduce the concept of conditional execution to extend our programming to handle more complicated scenarios.
Scenes from the DARPA autonomous vehicle race in 2007

**Autonomous Driving**

Google's modified Toyota Prius uses an array of sensors to navigate public roads without a human driver. Other components, not shown, include a GPS receiver and an inertial motion sensor.

- **LIDAR**
  A rotating sensor on the roof scans more than 200 feet in all directions to generate a precise three-dimensional map of the car's surroundings.

- **VIDEO CAMERA**
  A camera mounted near the rear-view mirror detects traffic lights and helps the car's onboard computers recognize moving obstacles like pedestrians and bicyclists.

- **RADAR**
  Four standard automotive radar sensors, three in front and one in the rear, help determine the positions of distant objects.

- **POSITION ESTIMATOR**
  A sensor mounted on the left rear wheel measures small movements made by the car and helps to accurately locate its position on the map.

Source: Google

The Google self-driving Prius

### 5.3 Pre-lab preparation

1. Reading: chapters 10 and 11. Review older chapters 1-9 and labs from cs 121 as needed.

2. Take the pre-lab quizlet 1 at the CS 122 Maple TA web site. You should do quizlet 1 before lab to be prepared for the lab activities.
5.4 Part 0 -- Prelab prep

Problem A - Learn how to enter a short script into a code edit region and execute it

To do this work, it will be helpful to have read "Code edit regions: executing a series of actions at once" in Chapter 10 of the course readings and have it handy as you do the work.

Do the following:

1. Open a Maple worksheet
2. Create a code edit region by
   Insert -> Code Edit Region

Checkpoint: What you should have should look like this:

3. Change size of region by
   Right click -> Component Properties -> change to 800 x 200 pixels

Checkpoint: What you should have should look like this:

4. Enter some code (enter this code exactly)
   #This is a proc to compute the hypotenuse of a triangle with sides s1 and s2.
   Hyp := proc(s1,s2)
   local hypotenuse;
   hypotenuse := sqrt(s1^2 + s2^2);
   return hypotenuse;
   end proc;
   # Now, execute this proc
   Hyp(7, 24);

Checkpoint: What you should have should look like this:

Notice the elements of a proc (read over comments)

Hyp := proc(s1,s2)  # Proc header - proc name (Hyp) and parameters (s1,s2)
local hypotenuse;  # This variable is used inn the proc, but not passed in
hypotenuse := sqrt(s1^2 + s2^2);
return hypotenuse;  # The value in this variable is returned at the
# conclusion of the proc
end proc;

# Now, execute this proc
Hyp(7, 24);  # Use the proc's name to execute it and pass in
# values for s1 and s2 respectively.
5. Execute the code. You can do this (assuming that the flashing cursor is located in the code edit region box, by typing command-E (control-E on Windows), or by clicking on the "!'" icon in the Maple toolbar.

Checkpoint: What you should have should look like this:

```maple
Hyp := proc(s1,s2)
    local hypotenuse;
    hypotenuse := sqrt(s1^2 + s2^2);
    return hypotenuse;
end proc;

# Now, execute this proc
Hyp(7, 24);
```

6. Collapse the code region by clicking on the region and then Right-click -> Collapse code region. After that, you should see:

```maple
Hyp := proc(s1,s2)
    local hypotenuse;
    hypotenuse := sqrt(s1^2 + s2^2);
    return hypotenuse;
end proc;

# Now, execute this proc
Hyp(7, 24);
```

Clicking on the code region icon will re-execute the script.

**Problem A deliverables and outcomes**

Have copies of your executed worksheet files ready to show and demo to the staff in order to get credit for doing this part of the pre-lab. Your team should be prepared to explain verbally or demonstrate some of the following things:

a) How do you create a code edit region in a worksheet?

b) How do you adjust the width and height of the text box region? What are reasonable settings for a region so that you can get work done?

c) How do you "iconify" the region? What happens when you click on the icon? How do you "un-iconify" a region?

d) How do you execute all of the lines in a code edit region? What is the keyboard shortcut for doing so?

e) What is an execution trace and what kinds of information can you get from it?

**Problem B - "while" loop introduction - using "while" loop and variables as counters**

1. Create a code edit region and re-size it as you proceed so that the entire segment of code is visible in the CER window.
2. Enter and execute the following code segment. Be sure to carefully review the code along with the associated comments:

```maple
# this code adds a list of numeric elements
AddList := [1, 2, 3, 4, 5];
ListSize := nops(AddList); # nops is a Maple function to determine the number of elements in a list
ListSum := 0; # keep an accumulated value of the sum as new elements are added in
count := 0; # keep track of the number of elements already added to the sum
while (count < ListSize) do
    count := count + 1; # increment the counter here so that the correct element is accessed next
    ListSum := ListSum + AddList[count];
end do; # end of while loop - all elements accumulated

print("Sum of list elements ":, ListSum);
```

3. **Problem B deliverables and outcomes**: Answer the following questions:

   a. What is the sum of the elements as produced by the program?

   b. What happens when we change the semi-colon after the "end do" statement to a colon?

   c. What is the effect of initializing the count to 1 instead of to 0?

   d. If the count was initialized to 1, what else in the program would need to change in order to obtain the correct answer?

Show the modified code and result to the grader.

**Problem C -- learn how to run a car simulator program in a code region and interpret the results**

First download and expand the zip file CS122Lab1.zip from the course web site.

CarSimulator.hdb

CarSimulator.mla

CS122Lab1StarterB1.mw
CS122Lab1StarterB2.mw
CS122Lab1Problem4-Tutorial.mw
CS122Lab1Problem1.mw
CS122Lab1Problem3.mw
CS122Lab1Problem4.mw
CS122Lab1Problem5.mw

When working on car simulator problems, you should begin work by double clicking on one of the lab .mw files to start up the simulator. This will have the effect of causing Maple to automatically look in the right place for the CarSimulator library files. If you have trouble doing this, ask for help.

Let's see a quick demo of the simulator.
(a) **Open and execute** the worksheet named CS122Lab1StarterB1.mw. Within the worksheet, you should see lines of code that:

1. Establish initial values for Maple variables used in the simulation. This initialization must be each time you want to use the simulator in order to set up the car "arena" and set the initial position of the car.
2. Setting up a target for the car to end up at (it's optional).
3. Establishing the limits of the "playing field" for the simulator. You can visualize this as a wall of specified dimensions that the car will run into if it goes too far.
4. Writing a Maple procedure that describes actions that the car will take. Note that the sample program, `run1a`, contains a few actions from the car simulator package *move*, and *turn*.

Note that the car is pinkish when it is running but turns gold when it finishes. This is what it does when it stops next to a target square (blue). The car should automatically stop when it is pointing at and next to a target.

(b) **Open and execute** the worksheet named CS122Lab1StarterB2.mw. This worksheet explores the state table, which explains the position and condition of the car at points of time.

**Problem C deliverables and outcomes**

Have copies of your executed worksheet files ready to show and demo to the staff in order to get credit for doing this part of the pre-lab. Your team should be prepared to explain verbally or demonstrate some of the following things:

a) What is the Maple name of the package for the Car Simulator?

b) What does the command `with(CarSimulator):` do?

c) What is the color of a "wall" square? What is the color of a target square?

d) What does direction of the arrow drawn at each step for the car indicate?

e) Describe what you would see if the car was originally located at (3,2) and was told to move 3 squares backwards. How could you tell whether the car was moving forwards or backwards?

f) What is the name of the operation in the car simulator package that
   - produces the animation of the car moving around?
   - runs the simulation of the car with a control program?

g) What is the purpose of the argument given to the `run` operation?

**5.5 Lab Exercises**

For these problems, you should continue to work with the unzipped files for this lab.

**Problem 1 -- Get the car to move in a variety of directions**

Open the worksheet named CS122Lab1Problem1.mw Solve the problem in it, which asks you to write a program to move in a variety of directions, as specified in the work sheet. To get points for this part, you should be able to demonstrate the program.

**Problem 2 -- Understand more about the CarSimulator package**

The CarSimulator package includes some help pages on the various features of the package. Activate Maple on-line Help. In the "Table of Contents" pane on the left hand side you should see a folder for the CarSimulator. (Note: CarSimulator help should be available after you have done Part 0, because some of the actions in Part 0 tell Maple to look in the current directory for help. If not, open again one of the Tutorial worksheets from Part 0 and then activate Maple help again.)

| CarSimulator documentation displayed in Maple on-line help |
Problem 3 -- Write a program to get the car to move in a Square

Solve the programming problem in CS122Lab1Problem3.mw. You should write a program that exactly duplicates the behavior of the sample animation.

This problem just has the car moving in a square whose side is 6 units. Note that movement begins Eastward from the starting position.

Problem 4 -- Write a program to get the car to move under a wall, detect a gap and proceed

towards a target
In this problem, we present another scenario. Your mission is to write a program that will negotiate any of the various versions of the scenario. In order to get credit, the same program should work for all of the scenarios. In other words, you should be able to copy and paste your solution for the first version into the code edit region that executes the second version, and it should work there without alteration.

(a) Open, inspect, and execute the scenario and code presented in CS122Lab1Problem4-Tutorial.mw. This shows the car moving forward until it bumps into the right wall, turns around and proceeds until it bumps in to the left wall. Notice that you can execute the same program in a scenario where the walls are in different locations and the car still turns/stops when it hits the right/left walls respectively. You should be able to answer these questions:

(i) Describe, in your own words, what actions are being repeated by the while.

(ii) Describe in your own words, what actions are necessary as initialization before the repetition of the while gets started.

(iii) Describe in your own words, the condition checked by the while loop. When will the loop repeat? When will the loop stop?

Write these descriptions on the whiteboard.

(b) Open and study the scenario in the code presented in CS122Lab1Problem4.mw. Complete the coding so that your control function can handle any of the sample scenarios.

Idea: move forward until you encounter a gap in the overhead barrier. Then turn right and move until you encounter a goal.

When you have solved the problem and can get the same code to solve both executions (with different wall lengths and target positions), have it graded by the staff.

Problem 5 -- Write a program to detect the nth gap in an overhead wall

Solve the following scenario. As with Problem 4, your mission is to write a program that will negotiate any of the various versions of the scenario. In order to get credit, the same program should work for all of the scenarios. In other words, you should be able to copy and paste your solution for the first version into the code edit region that executes the second version, and it should work there without alteration.

Open the code in CS122Lab2Problem5.mw, follow the instructions and solve the problem. You will notice that this problem is an extension of Problem 4, except that you need to find the nth (not 1st) gap in the wall before turning right and proceeding towards the target. This additional complexity causes you to use nested "while" loops (the outer loop moves you from gap to gap, while the inner loop moves you into and then across a specific gap. A counter variable will be needed to keep track of the number of gaps and special logic will need to be included before you proceed to the final target to turn the car around and move back to the nth gap (if you overshoot this gap).

5.6 Afterword: Comparison of NXT and Maple as programming languages for control

NXT for Lego robots and the CarSimulator package in Maple present different approaches to writing computer programs that control a device. Mindstorms' visual interface allows for "wired together" actions configured by point-and-click menu selection that move a robot. The Maple CarSimulator uses textual specification for actions and configuration. Repetition (handled by "for" in Maple) and conditional decision-making (handled by if or while) are present in both languages as a way of expressing control. A significant difference is that the car simulator takes a "move one unit at a time" approach to motion whereas control of a real robot typically engages motion or a motor continuously until another command disengages the action.

The textual interface tends to be preferred by programmers for larger control programs. It is usually faster to enter control programs (or to modify them) through textual entry than through clicking -- assuming that you are reasonably proficient at typing and meeting the syntactic exactitude required when communicating in a programming language. It also tends to be easier to do transformations of textual programs than visual programs. For example, if you decided that you wanted to modify a procedure so that all the turns to the right are replaced by turns to the left, you could find and modify all relevant instances of turn textual search and replacement. Doing a similar transformation in a visual program can often be much more laborious.

The Maple CarSimulator does not control a real robotic car. Nevertheless it and other software control simulators have their use in engineering applications because it's usually much faster to develop ideas and principles using simulator results, than it is to set up and test a physical device. For example, in a single lab period, you've been able to get in many more test runs with the simulator than were possible with running a physical
Even if the simulation is not perfectly realistic, there are typically many problems that can be worked out using them. Once the program is ported from the simulator world to a robot, more development will need to be done to adjust the simulator solution into a solution that works with the real robot. However, the overall development time should be less.

Textually-entered languages such as Java, C or Python can be used to program NXT robots. Maple could be as well. For the reasons we have mentioned, extended projects with robots such as solving and writing the solution to Sudoku puzzles (http://www.gizmowatch.com/entry/lego-mindstorms-nxt-robot-tailored-to-solve-sudoku-puzzles/) or manipulating and solving a Rubic's Cube (http://www.youtube.com/watch?v=3QOvEG27Gt4) would be more quickly developed in such languages rather than NXT visual language.
6
6

6.1 Overview

In this lab, we continue to expand our repertoire of program control by looking at what we can do with if and for. We've already seen the use of conditions for continuing or terminating repetition. if addresses situations which select between two possibilities, each of which is intended to be done only once. for addresses situations that are repetitions that uses counter (index) variables to help control repetition. for is like while, with the counter variable initialized and updated automatically. It can simplify certain situations where there the number of times a repetition should happen is known before the looping starts.

We will continue to utilize the car simulator API from Lab 1 to introduce these new control statements.

6.2 Pre-lab preparation

1. Reading: chapters 12 and 13. Review previous chapters as needed.

2. View the on-line video that demonstrates usage of if and for. You can find the video at the course web site among the course files for this lab. Before lab, imitate the examples given in the video, to see that you can enter code into a code edit region and execute it.

3. Take the pre-lab quizlet 2 at the CS 122 Maple TA web site. You should do quizlet 2 before its deadline to be prepared for the lab activities.

6.3 Part 0 -- Prelab prep

A. Introducing the "for" loop

Sometimes, we know exactly how many times a loop should be executed prior to running the loop. In this case, using a "for" loop control statement instead of the conditional "while" loop construct can be a more efficient programming approach. The following example illustrates the "while" and "for" loop alternatives for solving the same problem.

The code edit region below demonstrates this comparison for adding the numeric elements of a Maple list and reporting their sum:

```
# 1st, define the list of integers to add together
AddList:= [1, 2, 3, 4, 5];

# obtain the number of elements in this list using Maple's "nops" function
ListSize:= nops(AddList);

# Using a while loop, solve this problem
WhileListSum := 0;  # stores the cumulative sum of the elements
Count := 0;    # Current count of number of elements processed

while (Count < ListSize) do
  Count := Count + 1;
  WhileListSum := WhileListSum + AddList[Count];
end do:

print("Sum of list using while loop logic ", WhileListSum);
```
# Now solve the same problem using a "for" loop
# Note that a "for" loop based approach is appropriate since we know the
# number of list elements (= ListSize) prior to entering the loop

ForListSum := 0;
for count from 1 to ListSize do
    ForListSum := ForListSum + AddList[count];
end do:
print("Sum of list using for loop logic :",ForListSum);

[1, 2, 3, 4, 5]
5
0
0
"Sum of list using while loop logic :", 15
0
"Sum of list using for loop logic :", 15

Note that the "for" loop based solution does not require any manual tracking of the "count" of elements. This simplifies the logic for the solution algorithm.

B. Introducing the "if" statement

The "if" statement allows a program to selectively execute code, based on whether or not a condition is true (or false). The following pair of examples demonstrate 2 of the basic versions of the "if" statement construct.

1. Simple "if" - action to be executed only if condition is "true" (no action if "false")

# From a list of people's ages, determine the number of people who are 21 +
AgeList := [25, 15, 88, 11, 96, 21, 30];
Count21Plus := 0;
for i from 1 to nops(AgeList) do
    if(AgeList[i] >= 21) then
        Count21Plus := Count21Plus + 1;
    end if;
end do:
print("Count of people over 21:", Count21Plus);
6.3 Part 0 -- Prelab prep

2. "if-else" construct - different action based on condition being "true" or "false"

```plaintext
# Print a list of GPAs and include the note "Dean's List" if greater than 3.5
GpaList:= [3.2, 4.0, 3.5];
for i from 1 to nops(GpaList) do
    grade := GpaList[i];
    if (grade <= 3.5) then
        print("GPA :", grade);
    else
        print("GPA :", grade, "  Deans List");
    end if;
end do:
```

C. Using "for" loops and "if" statements to simplify a Car Simulation example

In CS122 Lab 1 (problem 5), we created code to drive the car to a target placed above the nth gap in an overhead wall. Since we were confined to using "while" loops (we did not yet know about "for" loops and "if" statements), we were forced to construct a rather inelegant algorithm that might have looked something like this:

```plaintext
GapCount := 0;
# ngap is passed to the proc and represents the gap above which the # target resides
while (GapCount < ngap) do    # if you are at a gap which is not the nth gap
    while (isTouching(Pi/2, 'bt')) do    # you are moving towards a gap
```
move(1);
end do;

# at this point, you are at a gap
move(1);  # move 1 more block past the gap to the start of the next wall
GapCount := GapCount + 1;
end do;  # end of processing for current gap. Search for next one.

# At this point, we are 1 block beyond the nth gap. Must turn around and go
# back to the nth gap and re-orient the car.

turn(Pi);  # do a 180
move(1);  # move back to the nth gap
turn(Pi/2)  # make a left turn, so that car is now aimed toward the target

# move up until you reach the target from below

Please notice that this (inelegant) solution, constructed using only (nested) while loops and counters, requires us to include rather awkward logic at the end to correct the overshoot and re-position the car. Also, notice the need to manually manage a gap counter variable.

A more well constructed algorithm is shown below to accomplish the same navigation by introducing "for" and "if" control statements.

# ngap is passed to the proc and represents the gap above which the
# target resides

# since we know that we need to run following "while" loop code exactly ngap
# times, we can use a "for" loop to accomplish this

for GapCount from 1 to ngap do
    while (isTouching(Pi/2, 'bt')) do  # you are moving towards a gap
        move(1);
        end do;
    # at this point, you are at a gap
    if (GapCount < ngap) then  # use an "if" to avoid the extra move at the nth gap
        move(1);
    end if;
end do;  # end of processing for current gap. Search for next one.

# At this point, we are exactly at the nth gap. No need to turn around and go
# back to the nth gap and reorient the car.

```
turn(-Pi/2);  # make a right turn towards the target

# move up until you reach the target from below
```

Compare this version of the code to the version above.

a. Note the elimination of manually maintained counters. The "for" loop automatically handles this activity

b. Note also how the "if" statement selectively suppresses the move beyond the gap for the target gap, thus enabling the direct "right turn"

at the final gap towards the target.

You might want to run this version of the code using the files from Lab 1, problem 5 to confirm its correctness.

### 6.4 Lab Exercises

For these problems, you should continue to work with the unzipped files for this lab. The zip file is located in the lab materials section of the course web site along the Lab 2 description link.

**Problem 1 - Moving the car in a figure 8**

Problem: Open and study the scenario in the code presented in CS122Lab2Problem1.mw. With the car at this starting position, create code to have it move in a figure 8 consisting of 2 squares whose sides are both 5 units in length. Use nested "for" loops to trace the figure 8 path.

This exercise is similar to the week 1 engr102 robot lab assignment of tracing a variety of shapes. One difference is that the Car Simulator API is restricted to turns in increments of 90 degrees (cannot simulate a curved turn).

**Problem 2 - Moving to and turning at designated intersections**

Problem: Open and study the code and setup in CS122Lab2Problem2.mw. It shows a grid of streets and intersections with the car positioned at a starting point. Develop code to move the car forward to the 1st 4 way intersection, turn left and proceed to a target. The key concept to employ is to determine when the car is at a 4 way intersection. Using the "isTouching" function, the car will be at a 4 way intersection when this function returns "false" for all 4 angles (0, Pi/2, Pi and -Pi/2) when testing a specific car position.

**Problem 3 - Moving on a diagonal**

The Car Simulator API turns only at 90 degree angles (ie. it cannot execute a 45 or other non right angle turn). If a target object is diagonally across from the car, move the car in a "straight line" to the target.

Problem: Open the code in CS122Lab2Problem3.mw and develop code to diagonally move across the grid to the target.

This example is similar to the engr102 robot lab requirement of moving directly to the appropriate (nuclear or waste) dump site.

**Problem 4 - Moving around a barrier to a target placed above the barrier**

Problem: Open the code in CS122Lab2Problem4.mw. Move up until you touch a wall. Then move left around the wall and back to the middle above the wall. Finally, proceed to a target which is directly (vertically) above the original position of the car.
This problem mimics the engr102 robot lab exercise in which a touch sensor notices the barrier and then action is taken to travel around it and finally straighten out again to move forward to a target.
7 Lab 3CS 122 Computation Lab II Winter 2012 Directions and Problems

7.1 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. In this lab, we will be utilizing Maple's "proc" version of user defined functions as opposed to the single statement user defined function studied in CS121.

Part 2 practices with iteration (loops) in the context of a numerical computation. 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 to control the repetition. The loops occur within procedures whose parameters specify initial values for the loop as well as constants that occur within the computation. This allows us to conveniently re-run the computation with different values for the parameters.

7.2 Pre-lab preparation

1. Read chapters 13 and 14 (new material). Review prior material (chapters 1-12) as necessary.

2. Read the rest of these lab directions.

3. Take the pre-lab 3 quizlet at the CS 122 Maple TA web site. The deadline for doing the quizlet will be 8:00 am the Monday that lab week starts.

4. 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.

7.3 Directions for this lab

1. Form a lab group of two or three people. 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. (20 minutes)

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

4. The lab will start with the instructional staff going over part of Part 0 (15-20 minutes). This is a warm-up for the real lab work, to have you try out the following concepts: user-defined functions that draw, setting up and using tables, using a loop to assign values.

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 in a code edit region. Avoid using the clickable interface to perform calculations. However, you may use the clickable interface for entering expressions (e.g. square roots) or symbolic constants (e.g. π).

*** Note that the drawBoxB proc that you develop will be used in Lab 4.


7.4 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 whose re-use is of 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:= proc (x,y,n)
    display([line([x,0],[x,y]), line([x+n,0],[x+n,y])],color=red);
end proc;

# 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);
```
[animate, animate3d, animatecurve, arrow, changecoords, complexplot, complexplot3d, conformal, conformal3d, contourplot, contourplot3d, coordplot, coordplot3d, densityplot, display, dualaxisplot, fieldplot, fieldplot3d, gradplot, gradplot3d, implicitplot, implicitplot3d, inequal, interactive, interactiveparams, intersectplot, listcontplot, listcontplot3d, listdensityplot, listplot, listplot3d, loglogplot, logplot, matrixplot, multiple, odeplot, pareto, plotcompare, pointplot, pointplot3d, polarplot, polygonplot, polygonplot3d, polyhedra_supported, polyhedrplot, rootlocus, semilogplot, setcolors, setoptions, setoptions3d, spacecurve, sparsematrixplot, surfdata, textplot, textplot3d, tubeplot]

[annulus, arc, arrow, circle, cone, cuboid, curve, cutin, cutout, cylinder, disk, dodecahedron, ellipse, ellipticArc, getdata, hemisphere, hexahedron, homothety, hyperbola, icosahedron, line, octahedron, parallelepiped, pieslice, point, polygon, prism, project, rectangle, reflect, rotate, scale, sector, semitorus, sphere, stellate, tetrahedron, torus, transform, translate]

`proc(x, y, n)
plots:-display([plottools:-line([x, 0], [x, y]), plottools:-line([x + n, 0], [x + n, y])], color = red)
end proc`
(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). Remove the axis lines and labeling from the plot, so only the blue lines remain -- no numbers or black lines from the axes. Use on-line help for plot options to retrieve the information about what extra arguments you need to supply to the plot function to do this.

**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` (and `xtabList2` and `ytabList2`) will have the items you put into the tables, in indexed order.

<table>
<thead>
<tr>
<th>Test code + plotted output for tables and loops (a)</th>
</tr>
</thead>
</table>
| #Define a procedure to create two lists of data values that  
| #we will eventually want to plot.                  
| #We use tables to accumulate the                   
| #values in a loop, then convert the tables to lists.|
| FillTabs := proc(numpts)                           |
|   local xtab, ytab, i;                             |
|   #Initialize the tables to be empty               |
|   xtab := table();                                 |
|   ytab := table();                                 |
|   #Accumulate (i, 2*i) points in the x and y tables.|
|       for i from 1 to numpts do                    |
|         xtab[i] := i;                              |
|         ytab[i] := 2*i;                             |
|       end do;                                      |
|   #Return a sequence of the two tables converted to lists.|
|       return convert(xtab,list), convert(ytab,list);|
| end proc:                                         |
| #Invoke the procedure for various values of the parameter.  
| # Notice how the return result (a sequence of 2 lists) is stored in variables  
| # xtablist, ytabList below                          |
| xtabList, ytabList :=FillTabs(10); #Test1: produces two lists with 10 items |
| xtabList2, ytabList2 := FillTabs(15); #Test2: produces two lists with 15 items |
(b) We will practice looking at the execution trace generated by the loop inside the procedure.

**Sample output for b)**

```r
plot(xtabList2, ytabList2, style=point);

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

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```
\[\textit{FillTabs}\]

\{--> enter \textit{FillTabs}, args = 3\}

\texttt{table([ []])}

\texttt{table([ []])}

\begin{verbatim}
  1
  2
  2
  4
  3
  6
\end{verbatim}

\(---> \text{exit FillTabs (now at top level)} = \{1, 2, 3\}, \{2, 4, 6\}\)

\[\{1, 2, 3\}, \{2, 4, 6\}\] (7.3)

(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. Note that we are daisy-chaining together the \texttt{plot} and \textit{FillTabs} procedures

\begin{verbatim}
\textbf{Test code + sample output for (c)}

\texttt{untrace(FillTabs); \#Turn off execution trace}

\texttt{T := }"\text{Graph of } y = 2*x"; \#title for graph
\texttt{L := }["X axis", "Y axis"]; \#labels
\end{verbatim}
7.5 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 Lab3StarterPart1-1.mw, which should be contained among the downloadable files for this lab. It contains a code region that contains part of the code (sides 11 and 12) for drawing a box whose lower left hand corner is at (0,0). Read about the line function in the plottools package to figure out how this function works. Then complete (sides 13 and 14) the code region to draw a box that looks like this.

*** be sure to remove comments (#) from the code edit region as you activate those statements.
Once you understand the basic actions that draw a box, we can move onto incorporating this into a function (proc).

b) After you have been successful at drawing the square, add where indicated the definition a function called \textit{drawBoxA}. After it is defined, \textit{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 \textit{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 \textit{drawBoxA}, there should be two inputs. We could name them \textit{width} and \textit{height}, or \textit{w} and \textit{h}, 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 \textit{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 \textit{f}.

2. Next describe the result or output of the function based on the inputs.

For \textit{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 \textit{w} and height \textit{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:

\begin{verbatim}
function name := proc ( argument, argument , ... argument)
\end{verbatim}
return expression involving the arguments that computes the result;
end proc;

For the case of drawBoxA the function definition could look like this:

drawBoxA := proc (width, height)
  return plots\[display\]([[ line( ...), line( ...), line( ...), line( ....) ]]);
end proc;

The idea behind this definition is to use display's ability to combine several plot structures together into a single plot. In this case we combine four lines together to create a box. We're doing the "whole name" of display -- plots\[display\] because we're not counting on the user doing a "with(plots)" before they use drawBoxA.

In case we find that the return expression is too large to fit on one line conveniently, we can split it up over several lines. As a courtesy to the reader, we use indentation to make the structure of the expression more intelligible. Maple doesn't care about indentation in processing the definition, so we can use as much as we want to make the code more readable. So we could do something like this:

drawBoxA := proc (width, height)
  return plots\[display\]([[ line( ...), line( ...), line( ...), line( ....) ]]);
end proc;

Doing this kind of indentation also makes it easier to compare the different line subexpressions, which often allows us to find typos faster.

We expect to see the parameters width and height to show up in the expressions for the individual lines. If we chose different names for the parameters:

drawBoxA := proc (w, h)
  return plots\[display\]([[ line( ...), line( ...), line( ...), line( ....) ]]);
end proc;

then we would expect the innards of the line expressions to include mention of w and h in the appropriate way.

Either way should define the same function. Recall that defining the function does not make any displaying occur right away. That only happens when the function is invoked by writing an expression that provides actual values for the arguments. For example, if we had already entered the code to define drawBoxA, then on a later line we could enter

drawBoxA(5, 6)

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

After you have defined drawBoxA, uncomment the first two tests of drawBoxA 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 drawBoxA and see that they also work as the test commentary says they should. If not, then troubleshoot your problems.

d) Create another function drawBoxB that takes five arguments -- the width, height, the (x,y) coordinates of the bottom left hand corner and a string that describes the color (see Maple's on-line help for colornames for a complete list). See for example Test 3 in the worksheet, that says:
drawBoxB( 10, 10, 0, 0, "DarkMagenta"); #10 x 10 magneta box with bottom corner at (0,0).

Uncomment the first test for drawBoxB and get it to work, then run the rest of the tests.

*** Hint - once drawBoxA is working successfully, you could copy the drawBoxA code and:
1. add 3 new parameters (x, y offsets form origin and color) to the drawBoxB parameter list
2. add the offset variables for x and y coordinates to every point in the line plots

for example, if the point [ width, height ] is a point form drawBoxA and xlo and ylo are the names of the offset variables -->
change { width, height } to [ width + xlo, height + ylo ]

Apply this change even when a coordinate value is 0 eg. [ 0, 0 ] --> [ 0+xlo, 0+ylo ]
e) Once you have convinced yourself that your definition of `drawBoxB` works, write more code that uses `drawBoxB` and `display` to draw the following pictures. Using `drawBoxB` to draw all the boxes should be a lot more convenient than copying a lot of code that includes multiple lines.

**Art project 1**

```
```

**Art project 2**

```
```
In the next lab we’ll take this a little further by simulating a particle bouncing around in the box. Remember that you will need the drawBoxB proc as the starting point for Lab 4.

**7.6 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 \( A - k_1 \) * previous \( A \) * previous \( X \)

new \( X = \) previous \( X + k_1 \) * previous \( A \) * previous \( X \) - previous \( X \) * previous \( Y \)

new \( Y = \) previous \( Y + k_2 \) * previous \( X \) * previous \( Y \) - previous \( Y \) * previous \( X \) - previous \( Y \) * previous \( Y \)
new B = previous B + k3* previous Y

Note that we need eight variables: A, newA, X, newX, Y, newY, B, and newB 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 Lab3StarterPart2-1.mw. Execute the script.

*** Note that the starter script contains both the code for and call to proc chemSim1.

Complete the script so that it successfully updates A, X, Y, and B and plots the concentration of A over 5 time steps. In order to do this, you will have to translate the mathematical description for how to calculate the new values from the old into Maple syntax -- change equations to assignments, decide on the names of programming variables, etc. When you finish, you should see something like this:

![Graph of A](image)

Once you get things to work, adjust the number of time steps so that the reaction runs to equilibrium (ie. no further changes in concentration for any component in the reaction). In this case, you determine this most easily by experimentation with the simulation.

By reading the documentation associated with the simulation answer the question: how much time is needed to achieve equilibrium?

**Problem 2.2**

This section asks you to do extensive modifications to chemSim1 to build a new procedure chemSim2 that does a bit more.

1. In a new document and fresh code edit region, create a procedure chemSim2.

Hint: use the script results from Part 2.1 as the starting point for this part. Be sue to change the proc name from chemSim1 to chemSim2.
It should differ from chemSim1 in the following ways:

a) A0, B0, X0, and Y0 are now parameters to the procedure rather than local variables initialized inside the procedure. Your definition of chemSim2 should begin like this:

```
chemSim2 := proc(numTimeSteps, A0, B0, X0, Y0)
...
end proc;
```

You can't use A0 (or any other parameter that is now being passed into the proc) both as a parameter name and as a local variable. If A0 becomes a parameter, you have to delete both its initialization within the procedure and it's mention in the local variable list at the start of the procedure definition.

When you invoke the procedure, you'd do `chemSim2(5, 1000, 0, 10, 50)` to get a run similar to what you did with chemSim1 originally.

b) The procedure uses extra tables: xTab, yTab, and bTab, and accumulates the concentrations in a fashion similar to what was done for A.

c) The procedure returns lists of the time values and concentrations as five lists. Thus the return in chemSim2 will look like:

```
return convert(iTab,list), convert(aTab,list), convert(bTab,list), .... ;
```

Hint: Your call to chemSim2 could look like

```
IndexList, AL, BL, XL, YL := chemSim2(5, 1000, 0, 10, 50);
```

In this case, you would need to return these 5 lists in the same order they are listed in the assignment.

2. Create Fplot2, similar to Fplot but allowing additional parameters selecting a color, symbol, and legend. This will give additional style to our point plots, which we will need to display information about all four chemicals.

For example, to plot values in a list named xList, in blue with circles and the plot legend "chemical X", you would want to use Fplot2 like this:

```
Fplot2(timelist, xList,"Blue",circle, "chemical X").
```

If we wanted to display both the x and a values on the same plot we'd do (after a with(plots)):

```
display([Fplot2(iList, aList,"Green",diamond, "chemical A"),Fplot2(iList, xList,"Red",box, "chemical X")], title="A,X concentration versus time", labels=["time (tenths of a second)", "concentration (micromoles/L)"]);
```

This explanation only tells you how you want to use Fplot2 after you define it. Before you use it, you still need to provide the definition yourself.

3. After you've defined Fplot2, you can combine together the plots that produces into a single multiplot using `display` as we've done earlier with other kinds of plots. For example,

```
display( [ Fplot2(timelist, aList, "Red", box, "chemical A"), Fplot2(timelist, xList, "Blue", circle, "Chemical X")], title="plot of A and X");
```

We could give additional plot options arguments to `display` to have a title, axes labels, etc. for the multiplot. See on-line help on plot options to review how these parameters work with the `plot` function.

A good multiplot 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. The figure below indicates part of a multi-part graph which does all these things. You are free to design an alternative as long as it has the desirable properties mentioned: clear differentiation between chemicals through color, symbol, and legend, with a caption and labeled axes.

**Fragment of a multi-part graph for the chemical reactions**
Problem 2.3

We've been busy getting code to work properly. Now we use it to understand what the model, as implemented by the simulation program, predicts will happen.

Through analysis and experimentation, answer the following questions:

1. What would you say in a report about what happens to the four chemical products with the given initial concentrations. How long does it take to achieve equilibrium for all four products (each time step = 0.1 seconds)? Which chemical(s) have non-zero concentrations in equilibrium? Which component(s) are a) consumed in the reaction, b) produced in the reaction and c) are relatively unchanged by the reaction?

2. Briefly experiment with the simulation by changing some of the initial concentrations (A0, B0, X0 and Y0). See if you can produce a plot that varies noticeably from the original. Think about why your concentration change(s) might have impacted the reaction in this manner.
7.7 Final actions (end of class)

Upload copies all of your work 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 about scheduling a visit to a completion lab.

7.8 Summary and conclusion

In this lab, you practiced more with user-defined procedures. Use of procedures (and/or user-defined functions) is separated into two parts: defining the procedure, and using (invoking) it. Invoking the function allows you to perform those actions for whatever parameter values you invoke the function with. This allows you to reuse the coding (the definition) by invoking the function with different values for the parameters.

We used this idea, along tables and lists to construct a time-step simulation. A key idea in the looping using in the simulation is that certain variables are reassigned values during the repetition based on their previous value, even as tables record all the values.

Code development always involve some missteps, so you have also had practice troubleshooting using information available with error messages and execution trace information available with trace.
8 Lab 4CS 122 Computation Lab II
Winter 2012

Directions and Problems

8.1 Overview

There are two parts to this lab which gives you more practice with conditional execution, where different statements are executed depending on what is true at any particular point in time. We used conditional execution in Lab 2, to control the car in the Car Simulator so that it would decide between alternative series of actions based on what it sensed at a particular location. while and for loops are also a kind of conditional execution, but the conditions in them only control loop repetition and are awkward to use for other kinds of conditional execution.

Part 1 asks you to build a simulation of a particle rolling around a box. Each time step of the simulation involves movement of the particle according to its present velocity and position. Occasionally the particle hits the wall of the box, which causes it to rebound, changing the direction and possibly the speed. if statements are used to handle the two cases of the time step -- what happens when the particle doesn't encounter a wall and so travels in a straight line, and what happens when it does hit a wall and rebounds off the wall at a different angle. A for loop is used to repeat the time step calculation for many time steps.

Part 2 asks you to build a simple simulation of a bouncing object from specifications and a modest code outline. You will be expected to supply the rest. A for-while loop is used to control the number of time steps so that the simulation stops either after a specified number of bounces, or if not much bouncing is happening, after a specified number of steps.

8.2 Pre-lab preparation

1. Read chapter 15 (new material). Review older chapters and labs as needed. To start, you can practice by copying the examples in these chapters and getting them to work in your own copy of Maple. How can you test yourself about whether you can modify them to do slightly different things that you choose on your own?

2. This lab expects you to reuse the drawBoxB code you wrote from Lab 3, so you should retrieve the saved copies of your code from this lab and have it ready to download for this one.

3. Take the pre-lab quizlet 4 at the CS 122 Maple TA web site. The deadline for doing quizlet 4 will be 8am the Monday that lab week starts. There is no make up quiz for this since it's about pre-lab preparation.

3. You can get a head start on the lab by trying the exercises in Part 0, below.

8.3 Problems -- Part 0 (15 minutes)

We'll practice coding with a loop that looks at elements of a list of grade point averages. Open CS122Lab4Starter0.mw. It contains a code edit region. Execute the region. You will see that it prints out a message for each gpa that corresponds to a Dean's List level grade.

1. Change the loop so that it prints four kinds of messages: "Dean's List", "satisfactory", "on probation" and "suspended". Set up the program so that it flags probation for those with a gpa of less than 2.0 but greater than or equal to 1.0, suspension occurs with a gpa of less than 1.0. Note that a "satisfactory" gpa is between 2.0 (inclusive) and 3.6 (exclusive).

In order to do this, you will have to define values for two more variables, satisfactoryLevel and probationLevel. Then, modify the if statement in the loop to include some elif clauses:

if (Dean's List) then
  print .......
elif (Satisfactory) then
  print ......
elif (Probation) then
  print ......

else # this bucket will capture suspension level gpa's
print ........
end if;

if ...>=deansLevel then... elif ... <=satisfa then ... elif .... <= probationLevel then .... else ... end if;

Your output should look something like this:

Start test 1
2.200000 is a Satisfactory gpa.
0.700000 is a Suspension Level gpa.
2.600000 is a Satisfactory gpa.
1.500000 is a Probation Level gpa.
2.900000 is a Satisfactory gpa.
2.600000 is a Satisfactory gpa.
3.200000 is a Satisfactory gpa.
1.100000 is a Probation Level gpa.
3.400000 is a Satisfactory gpa.
3.000000 is a Satisfactory gpa.
1.800000 is a Probation Level gpa.
1.800000 is a Probation Level gpa.
1.300000 is a Probation Level gpa.
2.800000 is a Satisfactory gpa.
3.800000 is a Dean's List gpa.
4.000000 is a Dean's List gpa.
1.300000 is a Probation Level gpa.
1.500000 is a Probation Level gpa.
1.300000 is a Probation Level gpa.
1.300000 is a Probation Level gpa.
1.900000 is a Probation Level gpa.
1.700000 is a Probation Level gpa.
0.100000 is a Suspension Level gpa.
3.200000 is a Satisfactory gpa.
2.500000 is a Satisfactory gpa.
1.100000 is a Probation Level gpa.
2.000000 is a Satisfactory gpa.
3.000000 is a Satisfactory gpa.
3.800000 is a Dean's List gpa.
2.400000 is a Satisfactory gpa.
There are 3 students on the Dean's List.
There are 13 students on the Satisfactory List.
There are 12 students on the Probation List.
There are 2 students on the Suspension List.

Start test 2
There are 0 students on the Dean's List.
There are 0 students on the Satisfactory List.
There are 0 students on the Probation List.
There are 0 students on the Suspension List.

Start test 3
2.900000 is a Satisfactory gpa.
2.600000 is a Satisfactory gpa.
3.500000 is a Satisfactory gpa.
2.900000 is a Satisfactory gpa.
3.650000 is a Dean's List gpa.
3.200000 is a Satisfactory gpa.
2.700000 is a Satisfactory gpa.
1.400000 is a Probation Level gpa.
2.800000 is a Satisfactory gpa.
3.800000 is a Dean's List gpa.
4.000000 is a Dean's List gpa.
1.300000 is a Probation Level gpa.
1.500000 is a Probation Level gpa.
1.300000 is a Probation Level gpa.
1.300000 is a Probation Level gpa.
1.300000 is a Probation Level gpa.
1.700000 is a Probation Level gpa.
0.100000 is a Suspension Level gpa.
3.200000 is a Satisfactory gpa.
2.500000 is a Satisfactory gpa.
1.100000 is a Probation Level gpa.
2.000000 is a Satisfactory gpa.
3.000000 is a Satisfactory gpa.
3.800000 is a Dean's List gpa.
2.400000 is a Satisfactory gpa.
There are 4 students on the Dean's List.
There are 12 students on the Satisfactory List.
There are 8 students on the Probation List.
There are 1 students on the Suspension List.

Even though there's quite a lot here to do, there's no reason to code it all at once before you start testing your code. Formulating a plan for proceeding incrementally will make clearer the path to success at the work. Here's what we suggest: First, add the "suspension" messages by adding the elif needed to catch them. Execute the code you have written so that you see both the original messages, plus the suspension messages. After you succeed, get the probation level and the satisfactory messages working in a similar way. Consult the examples in the course readings for how to modify the loop to count the number of Dean's List student, printing out a message with the value of the counter after the end of the loop. Add other counters for the other messages in a similar fashion. We aren't going to give you points for each step of this, but it's still a good way to feel like you're making progress.

If you are unclear about the details of the coding even with the plan, huddle with the other members of your team to get things straight. Someone in the group should review the course readings to find examples of loops with if-then statements that you can use as models.

**8.4 Problem 1 Overview (40 minutes)**

We develop a time-step simulation of a particle moving around a box. Unlike the work with the Human Cannonball in CS 121, we don't have a formula in \( t \) that tells us the position of the particle for any value of time \( t \). Rather, like the chemical reaction simulation of Lab 3, we use the location of the particle at time step \( i \) and the knowledge of the particle's current velocity, to calculate the location of the particle at time step \( i+1 \). We do this in a loop and accumulate in tables the data on the position as it changes over time. Rather than a plot of all the positions together, we construct frame by frame an animation that shows the particle moving.

**8.5 Problem 1.1**

**Use a previously developed old function** (drawBoxB) **to provide a missing ingredient for an animation of a moving particle simulation.**

Open the code edit region in the file CS122Lab4Starter1.mw. Read the code. At one point there is an empty spot for the drawBoxB function that you built in Lab 3. Retrieve a copy of your code from Lab 3 and place it into the code edit region where indicated. Execute the region. You will see an execution trace, and at the end an animation (movie). Run the animation and see what happens.

a) Alter the parameters so that the simulation runs for 50 time steps in a 10 x 5 green box. You should see the particle move a bit further.

c) What happens if you set the number of time steps to be 200?
***Note that this script issue is due to the fact that we have not yet included code for the bounce off of the Western Wall.

c) Save a copy of your for this problem as yourNameCS122Lab41-1Answer.mw, for example TamiTaylorCS122Lab41-1Answer.mw.

8.6 Problem 1.2

We are going to follow the "incremental development" approach. We want a simulation code that will handle the particle bouncing off a wall in any direction, but to get that to happen we proceed one step at a time. In this part, you have code that has a "bounce of eastern wall" working and you are trying to extend it to handle another case; make it handle a bounce off of both the eastern and western walls. At the particle velocity set in the script, this should happen no later than 150 time steps.

In order to handle the western bounce, before coding you should first work out and understand the math used in the formula being used to calculate the location of the particle after the bounce after the eastern wall. Here is an explanation of the eastern wall bounce:

If the particle is within the box at \((x, y)\) and would move to \((x + \Delta x, y)\), it will hit the wall if \(x < WID \leq x + \Delta x\). Since \((x + \Delta x, y)\) is outside the box, it would bounce at \((\text{width}, y)\). If a bounce occurs with "perfect rebound" as we will assume here, the \(x\)-velocity is reversed and becomes \(-\Delta x\). The \(y\) velocity stays the same (and would do so for any bounce into the eastern wall even if the \(y\) velocity were non-zero).
This means that the travel back from the wall during the time step will be the amount beyond (WID,y) that the particle would have traveled, but in the opposite direction. Thus during a time step that has a bounce off the eastern wall we would break things into three phases:

a) The particle travels between x and WID.
b) The particle hits the wall, causing the x velocity to reverse itself.
c) The particle bounces and travels a distance \((x + \Delta x - WID)\) more away from the wall, but in the western direction. This is the same amount as it would have traveled during the remainder of the time period if the wall had not been there.

Thus, the final location of the particle is \((WID - (x + \Delta x - WID), y)\) = \(\left(2 \cdot WID - x - \Delta x, y\right)\). That is, the x-coordinate of the location of the particle at the end of the time step when the bounce occurred is \(2 \cdot WID - x - \Delta x\).

**Figure 2 Particle rebound from eastern wall**

Eastern wall running from \((WID, 0)\) to \((WID, \text{LEN})\)

- Particle rebounds and travels back that distance, ending at \((WID - (x + \Delta x - WID), y)\) = \((2 \cdot WID - x - \Delta x, y)\)
- Starting point of particle is \((x, y)\).
- Particle hits wall at \((WID, y)\)
Your job is to figure out the analogous formula for the bounce off the western wall (writing on a whiteboard and getting your teammates to agree is a good idea here -- it may be hard to keep it all in your head). Then alter the script so that instead of

```plaintext
if (delx>0 and ptpos[1]>= WID) #bounce East
then
  xwallPos := WID;
  delx := -delx;
  dely := dely;
end if;
```

it has, as another case, the test the bounce into the western wall.

```plaintext
if (delx>0 and ptpos[1]>= WID) #bounce East
then
  xwallPos := WID;
  delx := -delx;
  dely := dely;
end if;
```

```plaintext
elif condition #bounce West
then
  updating xwallPos, ptpos, delx, dely for a western wall bounce
end if;
```

Set up a simulation run that draws a 5 x 2 red box and places the particle initially at the coordinate (4,2). Run the simulation for 150 time steps. You should see the particle bounce off of both walls.

Now create a new procedure that it takes additional parameters vx0 and vy0, with delx and dely being initialized to the values of these parameters. Call this procedure particle2.

Make the particle2 procedure pass two tests:

```plaintext
particle2( 5, 2, "Red", "Blue", 100, [4,1], 0.1, 0.0); #should head eastwards and bounce off of both walls
particle2( 1, 1, "Red", "Blue", 100, [.5,.6], -0.1, 0.0); #should head westwards and do multiple bounces.
```

If your procedure doesn't seem to be working, turn on tracing so that you get to see the step-by-step actions of the procedure.

Your code will be inspected for this problem. You will be expected to following standard indentation and commenting style for the code that you write.

When you have completed this problem, save your work as `yourNameCS122Lab4-I-2Answer.mw`.

### 8.7 Problem 1.3

The objective in this problem is to extend your answer in Problem 1.2 to handle north-south bounces as well. You should test this by setting

```plaintext
pos0 := [1,9];
```
8.8 Problem 2 Overview (35 minutes)

We launch a rubber ball up in the air, at a velocity of 100 m/sec and an angle of 45 degrees. Each time the ball hits the ground, it rebounds upwards with a velocity that is only a fraction of the downwards velocity. In addition, the horizontal motion slows down each time due to friction between the ball and the ground.

We want to create a time-step simulation similar to that of Problem 1. New x and y positions will be updated in a loop, and the results accumulated in tables. Our rules for time step motion will cause the particle to slow down when it hits the ground, but not while it's moving through the air. While the x velocity is constant between bounces, the y velocity will be constantly changing due to gravity. Nevertheless, the overall structure of the program will look similar to that of part 1.

**Problem 2.1**

Recall that the Blammo trajectory model in CS 121 used the following formulae for position and velocity:

\[ \text{velx}(t) = v0x \]

\[ \text{vely}(t) = v0y - g \cdot t \]

\[ \text{xpos}(t) = x0 + t \cdot \text{velx}(t) \]

\[ \text{ypos}(t) = y0 + v0y \cdot t - \frac{g \cdot t^2}{2} \]

From this, we calculate

\[ \theta = \text{angle in radians} \]
\[ v_0x = v_0 \cdot \cos(\theta) \]

\[ v_0y = v_0 \cdot \sin(\theta) \]

We will update the movement of the ball in small time steps, in the way we did with Problem 1. This contrasts to what we did with Blammo in CS 121, where we had a single formula and just evaluated it for various values of \( t \). While having a single formula that describes everything is preferable, there are many practical situations where it isn't feasible to develop a single formula for the situation. Time step simulations are more work to program than formula evaluation a la Blammo, but they often can be made to work in even in situations where single formulas can't be found.

To get this simulation written, we again employ the incremental development methodology. In this part, we will just get a few time steps to work with this way of doing the simulation. In later parts of this problem, we will add bouncing and the collection of summary statistics.

The simulation revolves around the values of the following variables:

\[ dt \]: the amount of time between steps of the simulation. We will set it to .1 for this Part, although after you get the program to work you can change this value and see how that affects things. In a calculus mentality, making the value of \( dt \) smaller and smaller should produce a better and better approximation to real-life, where changes in velocity are instantaneous and constantly happening. In our simulation, the changes in velocity occur only every \( dt \) seconds.

We use the following variables to keep track of the position and velocity of the ball.

\( xp \): \( x \) position of ball at the current time

\( yp \): \( y \) position of ball at the current time

\( xv \): horizontal velocity of ball at the current time

\( yv \): vertical velocity of ball at the current time

Each time step, we do the following

#Store current values of \( xp \) and \( yp \) in the tables \( xpos[i] \) and \( ypos[i] \).

#Calculate new values for \( xp \), \( xv \), \( yp \), and \( yv \). Store these into \( newxp \), \( newxv \), \( newyp \), and \( newyv \) respectively.

These values can be calculated as

#new x position is \( xp + dt \times xv \) (present position plus the horizontal velocity times the amount of time of the time step)

\[ newxp := xp + dt \times xv; \]

#new y position is approximately \( yp + dt \times yv \)

\[ newyp := yp + dt \times yv; \]

#new x velocity is \( xv \) (no change unless there's a bounce)

\[ newxv := xv; \]

#new y velocity is \( yv - g \times dt \)

\[ newyv := yv - g \times dt; \]

Open up CS122Lab4Starter2.mw, which has most of this code already written. Fill in the parameters so that the simulation runs ten time steps, and then stops. You should see a plot that looks like the figure below. For this duration of time, the path looks almost like a straight line but you can tell that it's slightly curved. If we ran it for a longer period of time we'd see the parabolic behavior as with Blammo.

**Result of running Starter2 with parameters filled in to do ten time steps**
Once you have this working, make it run for 100 or 200 time steps. You should see the trajectory go back down, but keep on going even after it reaches the ground. In the next part, we will put more features into the simulation to get the bounces to happen.

Save a copy of your work for this part as yourNameLab4Answer2-1.mw

**Problem 2.2**

In this problem, we will detect the ball hitting the ground and calculate the rebound.

For a bouncing model, we will use two new parameters,

\[ R = \text{the coefficient of restitution} \] that describes the ratio of the rebound speed to the collision speed. We will take \( R=0.6 \).

\[ \eta = \text{the coefficient of friction}, \] that describes the ratio of the pre-impact horizontal speed, to the post-impact horizontal speed. Basically, the forward motion "erodes" a bit with each impact. We will take \( \eta =0.5 \). (The Greek letter \( \eta \) is pronounced "eta", by the way.)

Save a fresh copy of your work from 2.1 as yourNameLab4Part2-2.mw and then modify it as follows:

a) In the initialization section of the simulation, assign \( R \) and \( \eta \) the values described in the model description.

b) We will now install the bounce code, which will be contained within an if statement inside the loop, just as we did in Part 1.

Right after you compute newxp, newyp, newxv, and newyv in the simulation loop, check to see if \( newyp \) is at or below ground level. If it is, then do the following:

1. Change newyv to be \(-R*newyv\). This will cause the velocity to switch directions, but to be a factor of \( R \) less. Don't forget the minus sign or else the velocity will not reverse in direction!
2. Change newxv to be eta*newxv. This will cause the horizontal velocity to slow due to friction, but to continue in the same direction as before. There is no minus sign here.

3. Change newyp to be -newyp. In other words, the altitude achieved at the end of the time step is the same as the bubble would have traveled under ground level if there had been no rebound.

This is not exactly correct but should be close enough to produce realistic results with small values of dt. We see that this computational model diverges in two ways from "reality" -- the movement does not take into account air resistance, and introduces small errors at each time step because the steps are not infinitesimal though small. These are some of the causes why many computational models are only an approximation to the actual situation being modeled. Hopefully they are good approximations, but there isn't a claim that they will be identical to the results you'd get if you measured a bouncing ball experimentally.

When you run the simulation, you should now see a bounce occur. If you run it long enough, you should see several bounces.

Save a copy of your work as yourNameCS122Lab42-2Answer.mw

**Problem 2.3**

Now save a copy of your work as yourNameLab4Answer2-3.mw and make further changes:

a) Introduce an additional variable numBounces. Set it up so that it is initialized to zero and then incremented every time a bounce occurs. Print out the number of bounces after the simulation is over.

b) Modify your loop control so that the simulation runs until a specified number of bounces (say, 6) occurs regardless of how many time steps it takes. The grader will grade this segment by telling you how many bounces they want to see.

c) In addition to a plot, make a movie of the bouncing happening as in the original Blammo work. You should use display( listOfFrames, insequence=true,scaling=constrained) rather than the animate function to do this.

**Hint:** To capture each frame for the movie, use

frames[i] := plot([xp], [yp], style=point, color="Green");

in the loop as soon as xpos[i] and ypos[i] have been assigned at the start of the loop

**Hint:** Note that the loop header for running the 2.3 simulation can be written as

for i from 1 while numbounces < 6 do

where "for i from 1" ensures that an index for i will be available to use throughout the loop as table subscripts and

"while numbounces < 6" allows the loop to process until 6 bounces have taken place

Save your work as yourNameCS122Lab42-3Answer.mw

8.9 Final actions (end of class)

Be sure to get credit for your work verification sheet before you leave. In order to ensure proper credit, you should fill out the section number, time, and instructor name as well as your own names. If you cannot complete the work in the lab period, talk to the instructor how you can get credit for the unfinished work.

8.10 Summary and conclusion

In this lab, you practiced modifying and writing code that had loops and conditional execution in it: if...then...end if, if...then...else ... end if, and if...then...elif ... end if. You also saw how the use of parameters (LEN, WID, etc.) in the simulation made it easy to change the operating conditions of the simulation without having to change a lot of lines where these size parameters were mentioned.
You developed time step simulations where you had to provide more code than before. You should see in this simulation's loop, the use of at least two coding patterns mentioned in the section "Constructing iterations and the roles of variables" of Chapter 16 of the course readings: most recent value (new \(x\) and \(y\) position and velocity calculated from old position) and gathering a final result (storing plots in a table, which is eventually converted in a movie after the loop).
9 Lab 1 CS 123 Computation Lab III Spring 2012

Directions and Problems

9.1 Overview

In CS 121, we learned of basic operations that Maple (and many systems like it) provide: calculation, solution of equations, plotting, and data fitting. None of this really needed any college-level math. In this lab, you will learn how to use some of Maple's more advanced computational functions: \texttt{diff}, \texttt{int}, operations for optimization, piecewise expressions and spline data-fitting. This will allow you to use Maple to compute solutions to problems that benefit from college-level mathematics. Not much programming (procedure construction) is needed this week -- most of the work can be done interactively or with scripts in code edit regions. In Lab 2, we will return to programming again.

9.2 Pre-lab preparation

1. Reading: chapters 16 and 17. Review older chapters and labs as needed. If you are rusty on the math involved, you should refresh your familiarity with the math content (e.g. differentiation and integration) as needed.

2. Practice with the chapter 16 operations by trying to recreate the results given in the chapter readings in your own copy of Maple.

3. Practice creating piecewise expressions by entering the examples from Chapters 17 and performing calculus operations on them as given in the examples for that chapter.

4. Take the pre-lab quizlet 1 at the CS 123 Maple TA web site. You should do quizlet 1 before lab to be prepared for the lab activities.

9.3 Part 0

Download the worksheet CS123Lab1Part0Starter.mw. Fill it in with results and explanation where indicated. Use the word processing features of the Maple worksheet to include thoughtful and properly formatted explanations of your results, as directed.

9.4 Part 1

Download the worksheet CS123Lab1Part1Starter.mw. Using the word processing features of Maple worksheets, calculate the solution to a practical minimization problem, providing results and explanation where indicated.

9.5 Part 2

Download the worksheet CS123Lab1Part2Starter.mw. Using the word processing features of Maple worksheets, fill it in with results and explanation where indicated.

9.6 Part 3

Download the worksheet CS123Lab1Part3Starter.mw. Using the word processing features of Maple worksheets, fill it in with results and explanation where indicated.

9.7 Part 0 Answer

See CS123Lab3Part0Answer.mw.

9.8 Part 1 Answer

See CS123Lab3Part1Answer.mw
9.9 Part 2 Answer

See CS123Lab3Part2Answer.mw.

9.10 Part 3 Answer

See CS123Lab1Part3Answer.mw.

9.11 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.

9.12 Summary and conclusion

In this lab, you've gained experienced with a few more of the mathematical library functions available in Maple and similar systems. Like solve, they simplify problem solving because you need only input the problem description (whatever numbers, equations, formulas, etc. are necessary); the built-in functionality finds the results for you. Your programming skills are called into play when you build scripts using these operations as building blocks to calculate all the information you need to handle a situation. Some of the problems or the solution approaches don't make sense unless your mathematical knowledge covers the conceptual basis for what the functions know.

We have seen a new kind of expression, the *piecewise expression*, which is a mathematical cousin of the "if -then-else" statement. Piecewise expressions differ from if because it doesn't require the variables in it to have known values in order for Maple to do algebraic calculations with it. Maple's mathematical operations such as addition, solve, or differentiation work on piecewise expressions but do not always work with if.

Most computer systems, Maple included, contain thousands of built-in procedures who can be used by the programmer to speed the programming process. Experienced users learn how to use most procedures by being well-acquainted with the mathematics behind them, and then consulting on-line and tutorial documentation. Once they get the basic form and some examples, they write small experimental programs using the features until they are convinced they understand how to use them to attain their overall goal.
10 Lab 2CS 123 Computation Lab III
Spring 2012
Directions and Problems

10.1 Overview

Part 1 of this lab asks you to build a simulation procedure and supporting scripts from scratch. Fortunately, you've seen enough examples of this already so that it's a matter of figuring out how to modify the basic pattern for a time-step simulation, rather than having to figure out everything from a standing start. In order to do this, you will need to refamiliarize yourself with the time-step simulations that used loops in CS 121 and 122 -- the Chemical reaction and Predator-Prey simulations, as well as the moving particle and bouncing ball simulations.

You are writing your simulation as a Maple procedure, using the ideas from Chapter 11 of the course readings. Having the simulation in procedure form allows you to use it as a function. Having the simulation as a function makes it easier to run multiple variants of the simulation. Wanting to do that is typical for when you want to use the simulation to predict what will happen under multiple scenarios.

10.2 Pre-lab preparation

1. Review older chapters, labs, and quizzes as needed so that you are reacquainted with the prior time-step simulations, looping, conditional execution, and procedures.
2. Read the introductory directions of the lab. The Introduction section of this lab is much longer than usual because it explains the simulation scenario. Unlike prior work, where the formulas used in the time-step loop are just given to you, in this lab there is an explanation where and how the code is derived from the mathematical equations of the model.
3. Take the pre-lab quizlet 2 at the CS 123 Maple TA web site. You should do quizlet 2 before lab to be prepared for the first lab.
4. Practice building simple Maple procedures, such as the examples given in chapter 11. You should be able to enter such procedures and get them to execute. You should be able to invoke the functions you define and get them to return results.
5. If you're feeling adventuresome, you can practice building the procedures of Part 0 before you come to lab.

10.3 Part 0: Tour of a simulation with a graphical user interface

1. Download and open chemSimGUIWorksheet.mw. Open the first code edit icon and browse the code inside. It defines three procedures. chemSim2 is

the procedure that was developed in CS 122 that calculated the changing concentrations of four chemicals. It has as procedure parameters the quantities such as A0, B0, X0, Y0 (the initial concentrations of the four chemicals), N the number of time steps. It returns as a result a sequence of four plots, one for each chemical. The second, chemSim2B has a single parameter, A0, with fixed values for all of parameters of the chemSim2 simulation. After invoking chemSim2, this procedure produces a single plot that combines all four results returned from chemSim2.

In addition to the code in the code edit region, at the bottom of the worksheet are some GUI displays and controls.

2. Execute the entire worksheet by clicking on the !!! button at the top of the Maple toolbar. Not much will appear to happen, although there is more going on that appearances would indicate. Execution of the code edit region defines the procedures, but does not execute them. Because the procedure definitions end with colons, the execution trace does not list them, although it would had the definitions ended with colons. However, the chemSim2 and chemSim2B procedures have to be defined before theGUI can have anything to work with.

3. Operate the slider. It will update the plot window so that the results of the simulation with the initial concentration of A as set by the slider, and the fixed initial concentrations of B, X, and Y as indicated.

4. By operating the slider in a trial-and-error fashion, find the approximate value for the initial concentration of A such that the peak concentration of Y is around 700. You should be able to find it in a matter of seconds through GUI operation.
The moral of this story is that GUI controls make it much faster to enter parameters repeatedly than typing would. Having the plot window change in response to changes in the slider is another way to shorten the time it takes to try out and analyze a new parameter value.

So far, we have built simulations of physical phenomena where we have varied the initial conditions or other parameters. In Labs 2 and 3, we will build a simulation of a constructed environment -- an HVAC (Heating, Ventilation and Air Conditioning) system. The simulation models the heating and cooling of a room in a house. Relevant parameters (inputs) to the simulation include: the outdoor temperature, the initial indoor temperature, the temperature of the cold air coming from the air conditioner, and the amount of chilled air being pushed into the room by fans under control from a thermostat. Relevant information being computed by the simulation includes the indoor temperature as it changes over time.

In Lab 2, you build and test the basic procedure that runs the simulation. In Lab 3, we will extend the procedure so that it takes as an additional parameter another procedure that describes the thermostatic control we are interested in putting into the system.

Build the GUI controls and display, and then use it to explore some HVAC design situations. Design exploration is one of the reasons why technical workers typically need to run a simulation many times with possible parameter variation.

### 10.4 Introduction: A mathematical model of an HVAC system, and how use it to write a program simulating how air conditioning cools a house

We can model the heating and cooling of a house (called HVAC -- Heating, Ventilation, and Air Conditioning) through a simulation. Recall that in the bouncing ball simulation, we had four variables of interest, $x$ and $y$ position, and $x$ and $y$ velocity. However, the model had other symbols called parameters, which for the most part had fixed values during the entire simulation: $g$ the gravitational force constant, $R$ and $\eta$ the elastic and frictional force coefficients, etc.

The problem in this Lab is to model the cooling of an air conditioned house. The house under consideration has the following features:

1. It is rectangular, approximately 10 x 50 x 8 feet.
2. It has eight windows, each 2 x 4 feet.
3. It has an air conditioner, which blows cold air into the house.
4. There are two variables of interest:
   a) $T_z$ the air temperature inside the house (in °F), and
   b) $T_{ew}$ the temperature of the exterior wall(s).
5. Initially, the interior of the house is at the same temperature as the outside, before the AC is turned on. However, as the house cools off (if it does), then $T_z$ and $T_{ew}$ will decrease.
6. We consider both $T_z$ and $T_{ew}$ as functions of time, and so write them as $T_z(t)$ and $T_{ew}(t)$. The simulation's goal is to compute these temperatures as they change over time, and then produce plots and other information.
7. The house has additional sources of heat, which can be modeled as necessary. In this problem, we will assume that there are two computers with monitors turned on., Each computer system generates 400 BTUs/hour of heat. The model has provisions for other sources of heat but in this scenario we will assume that they are shut off.

---

**A somewhat incomplete diagram of the air conditioned house (courtesy of Drexel University AC 380 class, Spring 2009)**
Fundamental modeling relationships

The relationship between the rate of change of \( T_z \) with respect to time \( \frac{dT_z}{dt} \), and the interior air temperature and wall temperatures at that time can be modeled as:

\[
\rho_a V_z c_a \left( \frac{dT_z}{dt} \right) = U_{wi} A_{ew} (T_{ew} (t) - T_z (t)) + U_{win} A_{win} (T_0 - T_z (t)) + q_{heater} + q_{s-int} \tag{1}
\]

Here is a quick guide to all the symbols in this equation:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Nominal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>time</td>
<td>to be calculated (a variable)</td>
</tr>
<tr>
<td>( A_{ew} )</td>
<td>Exterior wall area ((ft^2))</td>
<td>(2 \cdot 8.50 + 2 \cdot 10 \cdot 50 + 2 \cdot 10 \cdot 8 = 1960)</td>
</tr>
<tr>
<td>( A_{win} )</td>
<td>Exterior window area ((ft^2))</td>
<td>32</td>
</tr>
<tr>
<td>( Q_{ea} )</td>
<td>Entering air flow volume (cubic feet per minute)</td>
<td>3000</td>
</tr>
<tr>
<td>( T_0 )</td>
<td>Outside temperature (in °F)</td>
<td>90</td>
</tr>
<tr>
<td>( T_{ea} )</td>
<td>Entering air temperature</td>
<td>65</td>
</tr>
<tr>
<td>( U_{wi} )</td>
<td>interior side U value for the exterior wall (\frac{BTU}{ft^2 \cdot °F \cdot hr})</td>
<td>.01</td>
</tr>
<tr>
<td>( V_z )</td>
<td>zone volume ((ft^3))</td>
<td>(10 \cdot 50 \cdot 8 = 4000)</td>
</tr>
<tr>
<td>( c_a )</td>
<td>Air specific heat (\frac{BTU}{lbm \cdot ft^3 \cdot °F})</td>
<td>0.24</td>
</tr>
</tbody>
</table>
Heat load from the heater (BTU/hr)  
Heat from solar and other internal thermal loads (BTU/hr)  
Air density (in \( \frac{lbm}{ft^3} \))  
Exterior wall temperature (in °F)  
Zone temperature (as a function of time in °F)  
Rate of change of zone temperature with respect to time.  
Window U value (\( \frac{lbm}{ft^2 \cdot °F \cdot hr} \)) (see http://www.crittall-windows.co.uk/content/3/96/what-is-a-window--u-factor-.html)  
\( \frac{d}{dt} T_z(t) \) (to be used in a formula)  
\( U_{win} \) (to be calculated)

The relationship between the rate of change of exterior wall temperature and the interior air temperature and wall temperatures at that time can be modeled as:

\[
\rho_{ew} L_{ew} c_{ew} \left( \frac{d}{dt} T_{ew}(t) \right) = U_{wi} \left( T_z(t) - T_{ew}(t) \right) + U_{wo} \left( T_0 - T_{ew}(t) \right) 
\]

A quick guide to the symbols in (2) that do not appear in (1) are:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Nominal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_{ew} )</td>
<td>Exterior wall thickness (ft)</td>
<td>1</td>
</tr>
<tr>
<td>( U_{wo} )</td>
<td>Exterior side U value for the exterior wall (( \frac{BTU}{ft^2} ))</td>
<td>0.81</td>
</tr>
<tr>
<td>( c_{ew} )</td>
<td>Exterior wall specific heat (( \frac{BTU}{lbm \cdot ft^2 \cdot °F} ))</td>
<td>0.21</td>
</tr>
<tr>
<td>( \rho_{ew} )</td>
<td>Exterior wall density (( \frac{lbm}{ft^3 \cdot °F \cdot hr} ))</td>
<td>143</td>
</tr>
</tbody>
</table>

(1) and (2) are not quite in the right form to build a simulation program of the type we are familiar with, but they are close. Recall that the chemical reaction equations have calculations for each step along the loop:

new value of \( A := \text{some expression involving old values of } A, B, X \) and \( Y; \)

...  

new value of \( Y := \text{some expression involving old values of } A, B, X \) and \( Y; \)

What we are looking for in this problem is to write a loop for \( i \) where successive values of \( T_z \) are stored in a table \( TZ: TZ[0], TZ[1], TZ[2], ... \) etc. Similarly, values of \( T_{ew} \) are stored at \( TEW[0], TEW[1], ... \)

Thus we are looking for a way of writing
TZ[i+1] := \text{some expression involving older values of TZ and TEW};

TEW[i+1] := \text{some expression involving older values of TZ and TEW}.

Recall from your study of derivatives in calculus that for a small value of \( h \) and a "reasonable" functions \( T_z \),

\[
\frac{d}{dt} T_z(t) \approx \frac{T_z(t + h) - T_z(t)}{h} . \tag{3}
\]

If we set \( dt = h \), the time interval between steps of the simulation, then

\[
T_z(t + h) = TZ[i + 1] \text{ and } T_z(t) = TZ[i] . \tag{4}
\]

Since TEW[i] is the supposed to be the value of \( T_{ew}(t) \), then substituting (3) and (4) back into (1) we get:

\[
\rho_a V_z c_a \left( \frac{TZ[i + 1] - TZ[i]}{dt} \right) \approx U_{wi} A_{ew} \left( TEW[i] - TZ[i] \right) + U_{win} A_{win} \left( T_0 - TZ[i] \right) + q_{heater} + q_{s - int} + \rho_a Q_{ea} c_a ( T_{ea} - TZ[i] ) \tag{5}
\]

If we solve (5) for \( TZ[i+1] \) we get

\[
TZ[i + 1] \approx \frac{1}{\rho_a V_z c_a} \left( \rho_a V_z c_a T_z i + q_{s - int} dt + U_{wi} A_{ew} dt T_{ew} i - U_{wi} A_{ew} dt T_z i \\
+ U_{win} A_{win} dt T_0 - U_{win} A_{win} dt T_z i + q_{heater} dt + \rho_a Q_{ea} c_a dt T_{ea} - \rho_a Q_{ea} c_a dt T_z i \right) \tag{6}
\]

If we enter (1) into Maple, we can get it to derive the right hand side of (6) with a little editing and use of \textit{solve} for \( TZ[i+1] \).

If we go through similar substitutions for \( \frac{d}{dt} T_{EW}(t) \) in (2), we get that

\[
TEW[i + 1] \approx \rho_{ew} L_{ew} c_{ew} T_{ew} i - U_{wi} dt T_{ew} i + U_{wi} dt T_z i + U_{wo} dt T_0 - U_{wo} dt T_{ew} i \tag{7}
\]

After doing such derivation in Maple, we could \textit{latex} the right hand sides of (6) and (7) to put the formula in the textual format that would be reasonable to include in Maple code:

\[
\]

\[
\]

A third year undergraduate taking a numerical analysis course might be expected to derive (8) from (1) and (2) on their own. We don't expect you to do that, but have included the derivation as a demonstration of how basic principles and physics and calculus determine the code that's in a simulation program.
You're going to use the formulas in (8) in the rest of the lab, to simulate the behavior of the air conditioned house under various conditions of heating and cooling.

### 10.5 Part 1

Retrieve Lab2Part1 Starter.mw. It contains a code window listing the parameters and their value assignments, as well as the code from (8).

Using simulations of the past as models, write a simulation that calculates the values of $T_z$ and $T_{ew}$ as they change over time. The values should be stored into two tables, TZ and TEW.

A plot of the air temperature and wall temperature changing over time should be produced as a result. You should plot both temperature curves together -- air temperature should be red, and wall temperature should be blue.

Unlike past simulation problems, we are not telling you how to do this except that you can use techniques similar to the simulations of past labs and quiz problems where time steps are involved. As a hint, you should use the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Design purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>totalTime</td>
<td>The number of minutes you want the simulation to run.</td>
</tr>
<tr>
<td>dt</td>
<td>The length of the time step, in minutes. .01 is a reasonable value to start with.</td>
</tr>
<tr>
<td>T0</td>
<td>The initial temperature in the room, in Fahrenheit. The model assumes that this is the same as the outdoors temperature. 90 degrees might be a reasonable value to start with.</td>
</tr>
<tr>
<td>Tea0</td>
<td>The initial temperature of the cool air being put out by the HVAC. This might be something in the range 50-65 degrees.</td>
</tr>
<tr>
<td>Vz0</td>
<td>How much air is in the room, in cubic feet. We are thinking of the room as being 8000 cubic feet.</td>
</tr>
<tr>
<td>Qea0</td>
<td>Initial airflow of cool air pushed into the room by the HVAC fans, in cubic feet. 3000 might be reasonable given Vz0 above.</td>
</tr>
</tbody>
</table>

Entering items into the tables Time, TZ, and TEW up in a loop will allow you to generate the needed plots. Remember that after accumulating results into tables, the tables have to be converted into lists in order to have plots work.

Test your program by having it compute only a few time steps at first. When you get the simulation producing reasonable results, try a longer run so that you can see "steady state" temperatures. You should see the simulation cool off the room. How cool and how fast will depend on parameter values for Qea, Tea and perhaps others.
How to tell whether your function is working properly.

a) There should be no obvious error messages. There should be no warning messages about undeclared local variables.

b) The results should look correct. Typically, you can do this by augmenting the program to print out additional output and checking that it is valid (or at least plausible). We're not giving you target output to hit because "in the real world" problems rather than program output, are what is handed to you.

3. Once your simulation is working, answer the following questions:

a) What is the steady state temperature for the exterior wall, and the air inside?

b) Discover two drastically different ways of setting Q[e] and T[e] to establish a steady state temperature of 78 degrees.

4. Save this worksheet as myNameCS123Lab1Part1.mw.

10.6 Part 2

We've seen that the model handles a fixed situation -- the HVAC is turned on and spews out air at a constant cool temperature. The HVAC system puts the same amount of cool air all the time. If we were to model a real HVAC system, we'd want the model to be able to reproduce situations where the amount of cold air varies. For example, a thermostatic control might increase or decrease the air flow depending on how close the room temperature was to a target temperature.

In order to handle this new demand, we do what programmers need to do in this kind of situation -- take a working program and get it to do something additional.

We will introduce two new procedures into the control situation, acState, and airFlowControl.

acState

acState is a function whose parameters (inputs) are the current room temperature, and one of the values low or high. The design of the acState function is to describe the thermostat's behavior: it says that the state of airflow should be high when the temperature goes beyond a trigger value, and says that it should be low when the temperature falls below another trigger value.

Here is a sample definition for acState.

#Thermostat control function definition. Trigger temperatures are built into the function
#definition.

acState := proc(temp, presentState)
local triggerTemp, shutoffTemp ;
#temperatures that trigger high speed fan and low speed fan
triggerTemp := 79;
shutoffTemp := 76;
if presentState=low and temp>=triggerTemp
then return high;
elif presentState=high and temp<=shutoffTemp
then return low;
else return presentState;
end if;
end;
This is probably more verbose than needed to control the computer, but it also serves as a way of clearly communicating the original programmer's intentions to anyone who would alter the code because they want to change the thermostatic behavior. So ultimately it may do a better job than terse, cryptic code that's harder to figure out.

**airFlowControl**

*airFlowControl* (or a similarly named function) has a procedure that takes three parameters (inputs): the current state of the HVAC fan (*on* or *off*), and the designer's intention for the amount of air flow when the HVAC fan state is low or high. An example definition might be:

```maple
#Air flow control function definition
airFlowControl0 := proc(state, lowFlow, highFlow)
    #Parameters to set "low" and "high" settings of fan.
    if state=high
        then return highFlow
    elif state=low
        then return lowFlow;
    else
        error "state input should be high or low, was", state;
    end if;
end;
```

The design of the other function *airFlowControl0* is to specify the amount of air (in cubic feet per minute) that the fan is should blow at the high or low level. To aid in "bug proofing" the simulation, the *airFlowControl0* function will generate an error message if the fan state is neither high nor low.

1. Open a blank worksheet and set up an initial code edit region. Copy and paste these two (acState and airFlowControl) proc definitions into it. Then add tests into the code edit region to see that the definitions as they should. For example, acState(75, high) should return low, acState(95, low) should return high, airFlowControl0(on, 2000, 3000) should return an error message, etc.

    Testing these functions now, when you first enter them, is an aspect to the "incremental development" approach to writing software that we have been following in this course. Trying to use them in a simulation before you know that they work will increase the complexity of the debugging task.

    Save this worksheet as *myNameCS123Lab2Part2.mw*.

2. The next step is to change the simulation model so that it incorporates a thermostat controlling the airflow. Retrieve the worksheet *myNameCS123Lab1Part1.mw*. Add in the two function definitions from 1) into the code edit region between the restart and the rest of the initialization assignments (ie. directly after the "restart" command). Resize the code edit region so that you can see all of the code without too much scrolling. Save this worksheet as *myNameCS123Lab2Part2.mw*.

    Make additional modifications to the simulation to incorporate the HVAC acState and airFlowControl procs to control air flow. Call this new version of the procedure *HVACSim2*.

    i) Incorporate a further (local) variable, presentState, which is initialized to low before the loop starts.

    ii) Introduce additional parameters *lf* and *hf* (into the proc parameter list) that define the levels of low and high airflow.

    iii) Remove the parameter Qe0 from the proc's parameter list. Add the parameter *airFlowProc*. The design intent is that *airFlowProc* should be the name of the procedure that defines the airflow control, e.g. *airFlowControl0*. 
iv) Introduce the additional parameter fanStateProc. The design intent is that fanStateProc should be the name of the procedure that defines the state transition between low and high, e.g. acState.

iv) Inside the loop, the first step (just after the for loop header) is to use acState to compute the new present State.

iv) Edit the formula for Tz[i+1] so that it uses airFlowProc(presentState, lf, hf) rather than the constant Q[ea] for the air flow (ie. replace Q[ea} in 2 places with airFlowProc(presentState, lf, hf) in the Tz equation).

Get HVACSim2 working. As before, it should avoid error or warning messages when defined or operating, and give correct results.

**** To test, the HVAC proc for HVACSim2 should pass in parameters for totalTime, dt, T0, Tea0, Vz0, lf, hf, airflowProc and fanStateProc

3. Design air temperature, airflow and thermostat trigger values so that the temperature declines and then stays between 79 and 76 degrees. Find results so that it takes about 5-10 minutes to get the house to the point where the AC is cycling on and off.

4. Save the final version of your work for Part 2. You will need this result for next week's work.

10.7 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.

10.8 Summary and conclusion

In this lab, we have gotten you to synthesize code in the fashion that usually occurs in programming. Rather than looking for a cookbook that lays out all the details of a solution, we rely on our experience to find a relevant pattern, and then do the thinking that adapts the pattern to the situation at hand:

a) The explanation for how to calculate "new values from previous values" involves use of a differential equation that comes from the underlying science (physics in this case). While numerical computation isn't up to "infinitesimals" that derivatives would seem to need, mathematics provides a way of coming up with a formula that is a reasonable approximation. We know that the approximation will probably not be good unless the time steps be fairly small, though.

b) The software pattern involves the use of Maple tables to store a collection of time values, and of temperature values over time

c) The software pattern uses a loop to compute "new values from previous values", store these values into a table, and possibly prints out computed values. The pattern also typically has a sequence of initialization actions (time and temperature at the initial time, setting up empty tables, etc.) that happen before the loop.

e) After the loop converts the information in the tables in list form, which can then be printed, plotted, or animated. Post-loop activities can also include printing out other summary information (totals, minima or maximum).

Following this pattern should allow you to write the code for a variety of simulations.

In Lab 2 we will turning the simulation into a procedure. We will exploit the easy-to-invite feature of procedures by then connecting the simulation to a graphical user interface. This will make it easy to invoke the function with different parameter values using much less keyboard action.

It is possible to completely describe some situations using differential equations -- equations involving derivatives, such as we found in expressions (1) and (2) of the introduction. For such situations, one can avoid writing as much code by calling a library differential equation solver instead. A numerical d.e. solver can be given the differential equations as input and produces the numerical solution values automatically, without the user having to write the looping code that produces the values. One would still have to write the initialization and plotting code, though.

Maple's numerical differential equation solver is called dsolve numeric. If you want to find out more about it, you can read the on-line documentation for it. A "famous" numerical differential equation solver in Matlab is rk45. The name of the Mathematica solver is NDSolve.
10.9 Acknowledgments

We are grateful to Professor Jin Wen of the Civil, Architectural and Environment Engineering Department of Drexel University and her AE 380 class for providing us with the technical information used in this lab's model of HVAC.
11 Lab 3CS 123 Computation Lab III
Spring 2012

Directions and Problems

11.1 Overview

There are two parts to this lab. Part 1 asks you to connect the HVAC procedure to some of the graphical user interface widgets which will make the simulation easier to use. This will make it easier to conduct computational experiments that involve the simulation multiple times with different values of the parameters. Part 2 asks you to create a new version of thermostat control with 3 (low, middle and high) settings.

11.2 Pre-lab preparation

1. Reading: review chapter 17 (procedures, their creation, and their use) and read chapter 18 (how to configure and program GUI widgets). Review older chapters and labs as needed. Parts 1 and 2 of this lab require you to use the HVAC simulation that you built in Lab 2, so you should put your working HVAC simulation script from Lab 2 in a place from which you can conveniently download them to your lab computer when you do the Lab.

2. Practice with the input and output widgets by getting the slider/plot example given in Chapter 18 to work in a fresh Maple worksheet.

3. Take the pre-lab quizlet 3 at the CS 123 Maple TA web site. You should do quizlet 3 before lab to be better prepared for the lab activities.

4. Time permitting, attempt the Part 0 exercises of the lab, before you attend lab. If you are able to complete them ahead of time, it will save you time during the lab period.

11.3 Problems (100 minutes)

Part 0

We practice with GUI construction. We will design a GUI that is a user interface for the plot of the sin function, using a slider to determine the period of the sin function. By dragging the slider, we will see how the sin function changes shape as we change the value of the period. This is like the GUI given as an example in Chapter 18 of the readings. Because of the similarity, it is something that you should be able to do on your own through reading and experimentation before lab.

We will need four GUI elements to do this:

- A slider
- A text Area, which will show in text the numerical setting of the slider. This can be helpful feedback to the GUI and the GUI developer, to see what the slider is doing.
- A plot area -- a box into which the plot will appear
- A "Draw plot" button. Clicking on this button will redraw the plot after the slider is moved.

a) Create GUI layout. In a new worksheet, create a table with 3 rows and 2 columns, using the Insert->Table from Maple's menu. Your worksheet should look like this. You should be able to tell right away if you've succeeded in creating something that looks like this:

Table 11.1: A table with three rows and two columns

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We will now place the GUI components into this table.

b) **Placing GUI components in the layout.** Open the Components Palette (located on the left hand side of the Maple window, underneath the palettes for math expressions and symbols). Drag the appropriate components into the worksheet. Add a little text to the left of the text area labelling it "Frequency". After doing this, your worksheet should look like this:

**Table 11.2: Sin Plotter**

<table>
<thead>
<tr>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slider</td>
</tr>
<tr>
<td>Text Area</td>
</tr>
<tr>
<td>Plot Area</td>
</tr>
<tr>
<td>Button</td>
</tr>
</tbody>
</table>

The next step is to configure the components.

c) **Configuring the widgets.** Configure each component by right clicking on the component, then selecting Component Properties:

<table>
<thead>
<tr>
<th>Component</th>
<th>Configuration Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slider</td>
<td>name: plotSlider</td>
</tr>
<tr>
<td></td>
<td>range: 0 to 10</td>
</tr>
<tr>
<td></td>
<td>ticks: major=5, minor=1</td>
</tr>
<tr>
<td></td>
<td>show axis labels</td>
</tr>
<tr>
<td></td>
<td>update continually while dragging</td>
</tr>
<tr>
<td>Text Area</td>
<td>name: kText</td>
</tr>
<tr>
<td></td>
<td>number of visible rows = 1</td>
</tr>
<tr>
<td></td>
<td>not editable (slider will determine the k value)</td>
</tr>
<tr>
<td>Plot Area</td>
<td>name = sinPlotter</td>
</tr>
<tr>
<td>Button</td>
<td>Caption = &quot;DrawPlot&quot;</td>
</tr>
</tbody>
</table>
After configuring, your GUI layout should look like this:

Table 11.3: Sin Plotter

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Embedded Plot Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

While the GUI looks right, we haven't set up the computations triggered when the user touches the slider or clicks on the button.

d) Slider Programming

We now program the slider by right clicking on the slider -> Component Properties -> Edit - Action when Contents Change

When the programming window appears, just before the "end use" statement at the end of the region enter the programming `Do (%kText = %plotSlider);` in the programming window just before the end use; `Do` (with a capital "D") rather than `do` is the correct thing to enter here -- this is a different kind of programming than the kind we were doing with for/while loops.

Slider Programming window before programming added
use DocumentTools in

use DocumentTools in
# Enter Maple commands to be executed when the specified
# action is carried out on the component.
# Use:
#   # Do( %component_name );
# and
#   # Do( %component_name = value );
# to set and get properties of the component.
# You can also use arbitrary expressions
# involving components, e.g.:
#   # Do( %target = %input1 + 2*%input2 );
# Note the %-prefix to each component name.
# See ?CustomizingComponents for more information.

end use;

Slider Programming window after programming added

Do ( %kText = %plotSlider);
When you wiggle the slider around, you should see the text box change its value.

**e) Button programming**

In a similar fashion, edit the programming box for the button so that it includes the programming line:

\[
\text{Do( } \%\text{sinPlotter} = \text{plot(} \sin(\%\text{plotSlider}*x), x=0..10, \text{color=red})\text{);}
\]

This causes the plot window to compute a new sin plot, using the current value of the slider as the period.

After you add this programming, pressing the button should cause the plot window to display a plot with the specified period.

**Part 1**

Import a copy of the worksheet you had developed for the HVACSim2 version of the simulation of Lab2. Name it CS123Lab3Prob1-1.mw. We will now add a GUI onto this.

*** Run this version of the HVACSim2 solution from Lab 2 in order to confirm that you have a properly working copy of the lab 2 solution.

Now, create sliders, text boxes, buttons, and a plot area resembling the figure below. Configure and program them so that pressing the button runs the HVAC1 simulation on the specified fan flows.

**Sample GUI interface to HVAC simulation**
The user is supposed to set the sliders, then press the Draw Plot button. This will cause the simulation to run and display the results in the plot area.

Once you have your simulation working, determine the following:

Find fan settings so that the HVAC system is 1) always on high, 2) always on low once it attains the low state, 3) cycles between low and high approximately every 4 minutes, 4) cycles between low and high approximately every eight minutes.

**Part 2**

Save a copy of your work for Part 1 as `myNameCS123Lab3-Part2-1.mw`. 
1. We want to set Save your result again as myNameCS123Lab3-Part2-2.mw. Change the simulation as follows:

Create new thermostat functions `airFlowControl1` and `acState1`. This models a controller that has three fan states (low, middle, high) instead of just two.

Define these procedures as follows:

`acState1` sets the fan speed to low at 74 degrees or less. It sets the fan speed to high at 80 degrees or above. When the state is low and the temperature is between 76 and 80, the fan state is set to middle. There are a number of other cases to consider, we let you think this through, using designer's choice where that makes sense.

`airFlowControl1` takes the same three parameters as the original `airflowControl0`. If the value of the state parameter is middle, then the procedure returns a value between lf and hf.

*** Note that if the proc returns the exact average of lf and hf for the middle state flow, it is likely that this calculated flow will be too great to keep the temperature from rising to the triggertemp (= 80). Therefore, you might want to return a flow value that is close to the low flow (lf) than high (hf).

a) Devise tests for your new thermostat control procedures. Show the code edit region with the thermostat procedures should also contain the definitions of the original `airflowControl0` and `acState0`.

b) Create a test of the HVAC simulation running with the new thermostat. To do this, you only need to change the line invoking HVACSim2, to mention the names of the new thermostat control functions. To protect yourself, initially run a test that takes 10 or fewer time steps. Once it starts working, you can extend the operation period.

c) Change your GUI to have two buttons: one that plots the results of running the new thermostat, and another one that plots the old. Demonstrate the operation of both.

| GUI display results of plot with new thermostat |
Low fan (in cfm) 3900

High fan (in cfm) 8500

New thermostat

Original thermostat

time (in minutes) versus zone temperature (in degrees F)
11.4 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.

11.5 Summary and conclusion

In this lab, you have learned how to take a script developed interactively and turn it into a procedure with a name, HVACSIm2. Running the simulation with different values of the parameters becomes just entering a line with a different function call. The procedure result was not a number, or even a collection of numbers, but a plot structure.

This particular design for getting information into and out of the procedure lent itself to use with GUI widgets. The GUI interface we programmed allowed us to rapidly search through a collection of parameter values to see the variety of possible HVAC operation results that are possible.

The various ways of interacting with a Maple computation -- 2D input and output in a worksheet, 1D input and output in a code edit region, and through a GUI all have their strengths and drawbacks. The simple computations we did in CS121 are quickly understood and grasped through the basic worksheet, but it's harder to handle repeated execution with that style. Code edit regions make it convenient to write down scripts and procedures that have more complicated sequences of operations, including if and while/for statements. GUI interfaces invoke procedures that were first developed using code edit regions, but they can speed entry of parameter values, and have the potential of speeding comprehension of results by the use of graphical display and organization.
Index