Laboratory Exercises for Scripting and Programming for Modeling and Simulation

Copyright © Bruce Char
2008-10
Laboratory Exercises for Scripting and Programming for Modeling and Simulation
Contents

Acknowledgments ................................................................................................................................. vii

1 Lab 1 CS 121 Computation Lab I Fall 2010 Directions and Problems .......................................................... 1
  1.1 Overview ............................................................................................................................................. 1
  1.2 Introduction to Lab 1 (20-25 minutes) ................................................................................................. 1
  1.3 Problems -- Part 1 (20 minutes) ......................................................................................................... 1
  1.4 Maple TA (20 minutes) ...................................................................................................................... 4
    Notes on Maple TA ............................................................................................................................... 4
  1.5 Problems -- Part 2 (30 minutes) ....................................................................................................... 4
  1.6 Saving your work (5 minutes) .......................................................................................................... 5
  1.7 Final actions (End of class) ............................................................................................................ 5

2 Lab 2 Cs 121 Computation Lab I Fall 2010 Directions and Problems .......................................................... 7
  2.1 Lab 2 Overview ................................................................................................................................... 7
    Overview ............................................................................................................................................. 7
  2.2 Instructor's demonstration of word processing, scripting ................................................................. 8
  2.3 Part 1 ................................................................................................................................................... 8
    Problem Description ............................................................................................................................. 8
  2.4 Part 2 ................................................................................................................................................ 11
    Part 2 Description .............................................................................................................................. 11
  2.5 Final actions (end of class) ............................................................................................................. 12
  2.6 Concluding remarks ....................................................................................................................... 13

3 Lab 3 Cs 121 Computation Lab I Fall 2010 Directions and Problems .......................................................... 15
  3.1 Lab 3 Overview ................................................................................................................................ 15
    Overview ........................................................................................................................................... 15
    Directions for this lab ........................................................................................................................ 15
  3.2 Instructor's demonstration .............................................................................................................. 16
  3.3 Part 1 ............................................................................................................................................... 16
    Part 1 Description ............................................................................................................................. 16
    Part 1, problem 3 Answer ................................................................................................................... 18
    Part 1, problem 5 Answer .................................................................................................................... 20
    Part 1, problem 6 Answer .................................................................................................................... 23
  3.4 Part 2 ............................................................................................................................................... 27
    Part 2 Description ............................................................................................................................. 27
    Problem 2 Answer ............................................................................................................................. 29
  3.5 Attachment: starter script for Part 1 ............................................................................................... 32
  3.6 Final actions (end of class) ............................................................................................................ 35
  3.7 Conclusion ....................................................................................................................................... 35

4 Lab 4 Cs 121 Computation Lab I Fall 2010 Directions and Problems .......................................................... 37
  4.1 Lab 4 Overview ................................................................................................................................ 37
    Overview ........................................................................................................................................... 37
    To prepare for this lab beforehand .................................................................................................... 37
    Directions for this lab ........................................................................................................................ 37
  4.2 Instructor's demonstration of definition of functions, advanced plotting, and animation ................. 38
  4.3 Introduction to the "Human Cannonball" simulation ....................................................................... 38
  4.4 Problem 1 ......................................................................................................................................... 43
  4.5 Problem 2 ......................................................................................................................................... 43
  4.6 Problem 3 ......................................................................................................................................... 43
  4.7 Problem 4 ......................................................................................................................................... 45
  4.8 Final actions (end of class) ............................................................................................................ 46
  4.9 Conclusion ...................................................................................................................................... 47
  4.10 Acknowledgements ..................................................................................................................... 47
Acknowledgments

To our colleagues and families, who supported us in trail-breaking.

To our students, who learn how to work with the new and different.
1 Lab 1CS 121 Computation Lab IFall 2010 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 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 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 on-line 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 Problems -- Part 1 (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.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. |
| 1.b) What is exact fraction you get from adding together \( \frac{1}{2} \cdot \frac{1}{3} \), and \( \frac{1}{4} \) ? What about the sum of \( \frac{1}{2} \cdot \frac{1}{3} \), \( \frac{1}{4} \), \( \frac{3}{2} \), \( \frac{4}{3} \) 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 \( e^x \) 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 \( y \). |
\[
\frac{1}{2} + \frac{1}{3} \cdot \frac{47}{42} - 2 \cdot 3 \cdot 5 \cdot 13
\]

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

c) \(\sqrt{\ln(e^8)}\) (You should get 8.)

d) \(\sqrt{1 + \frac{2}{5} + \frac{3}{15} + \frac{1}{13}}\) (You should get 2.)

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

3. A state lottery allows you to pick six numbers from the numbers from 1 to 52 to win. Maple exact arithmetic to calculate the exact odds of winning. This can be done by using the "choose" function from the expression palette: \(\binom{\text{a}}{\text{b}}\) means "the number of ways you can choose \(\text{b}\) things from \(\text{a}\) 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.

\[
\log_{10}\left(\left|\sin\left(\frac{1}{x^2 + 1}\right)\right|\right) \\
\rightarrow
\]
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 type, the proper horizontal and vertical ranges and labels, and correct title and caption.

\[(x - 1)\cdot(x - 2)^2\cdot(x - 5)\cdot e^{-\frac{x}{10}} \rightarrow \]
1.4 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). Note that the password is probably Drexel One password. 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/Fall2009 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. Usually your choices will be limited, but the choices may change during the term depending on the need.

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.

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. An announcement will be made on Blackboard and the course website.

1.5 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 \( 3 \cdot x + 5 = 0 \).

2. Find the exact solution to

\[ 3 \cdot x^2 + 24 \cdot x + c^2 = 5 \]

(solve for \( x \)).

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 an also figure out how to enter inequalities from the keyboard!)


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:
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 \( t \)).

### 1.6 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.

### 1.7 Final actions (End of class)

1. Before you leave, get the staff to grade, sign, and collect the 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. The plus side is that 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 they won't matter much. If you don't get it all and it still appears mysterious after you review things on your own after class, 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 2010 Directions and Problems

2.1 Lab 2 Overview

Overview

This lab practices the development of re-usable multi-step scripts to solve a problem.

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? |
a) Do File -> New -> Document to get a fresh blank Maple worksheet. At the top of the document, insert the names of your group members, your lab section, your lab instructor's name, and the date/time. Then enter the following sequence of commands to solve version 1 of the problem. This portion of the work is "type it in and make sure that you get the same effects as the demo example shows".

For verification on this part, you should be able to identify the parameters of the problem and explain why they, and not other variables in the script are parameters.

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.

\[
24.61 \quad \text{assign to a name} \quad a \quad 24.61
\]
\[
-1.3 \quad \text{assign to a name} \quad b \quad -1.3
\]

We use the parameter $p$ to represent the percentage of terminal velocity that we want to hit.

\[
.98 \quad \text{assign to a name} \quad p \quad 0.98
\]

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

\[
a \cdot \left( 1 - e^{b \cdot t} \right) \quad \text{assign to a name} \quad velocity
\]

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.

\[
velocity
\]
\[
24.61 - 24.61 e^{-1.3t}
\]

(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.

\[
\lim_{t \to \infty} \text{velocity} = 24.6100000 \quad \text{(2.5)}
\]

assign to a name

\[
\text{terminalVelocity} = 24.11780000 \quad \text{(2.6)}
\]

\[
p=\text{terminalVelocity} = 24.11780000 \quad \text{(2.7)}
\]

assign to a name

\[
\text{fractionTerminalVelocity} = 3.009248466 \quad \text{(2.8)}
\]

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.

\[
\text{fractionTerminalVelocity} = \text{velocity}
\]

\[
24.11780000 = 24.61 - 24.61 e^{-1.3 t} \quad \text{(2.9)}
\]

solve

\[
3.009248466 \quad \text{(2.10)}
\]

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

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

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.

<table>
<thead>
<tr>
<th>Version 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>We find that for a 2009 Camaro (yellow, of course), $c = 0.15$, and $\Omega_f = 20$ Hz.</td>
</tr>
</tbody>
</table>

(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.

<table>
<thead>
<tr>
<th>Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>We find that for a 2003 Mini Cooper, $c = 0.18$, and $\Omega_f = 25$ Hz.</td>
</tr>
</tbody>
</table>
(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 $\Omega_f = 15 \text{ Hz}.$

(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 are given is using 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 2010 Directions and Problems

3.1 Lab 3 Overview

Overview

Before you come to lab, it is expected that you will have read chapters 6-8 of the readings, read these Lab directions, and taken the pre-lab quizlet on Maple TA.

This lab practices more with the development of scripts. 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.

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(...).

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

In addition to providing practice with the textual/functional style of programming, this lab has you use additional parts of Maple: lists and strings.

The jump to textual entry of an entire Maple operation takes some getting used to, but the greater flexibility and power of expression is needed to do sophisticated technical problems.

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.

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

Part 1 Description

From notes on Time Constants, ENGR 101 Fall 2009 (week 3)

A capacitor connected to a battery charges according to the following formula

\[ V(t) = V_i + (V_{max} - V_i)(1 - e^{-k\cdot t}) \]

where \( V_i \) is the starting voltage, \( V_{max} \) is the final voltage, and \( k \) is the reciprocal of the time constant \( \left( k = \frac{1}{\tau} \right) \). The voltage is measured in volts, the time in seconds.

Here is a problem that can be solved by a Maple script:

<table>
<thead>
<tr>
<th>Problem A, Version 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>For a particular capacitor/battery set up, we find</td>
</tr>
<tr>
<td>( V_i = 63, \ V_{max} = 266, \ k = 0.09 ).</td>
</tr>
<tr>
<td>(a) What is the value of ( \tau ), the time constant?</td>
</tr>
<tr>
<td>(b) What is the voltage after 20 seconds?</td>
</tr>
<tr>
<td>(c) What is the voltage after 40 seconds?</td>
</tr>
<tr>
<td>(d) How long does it take the voltage to reach 256 volts? Plot the voltage curve and a horizontal line whose intersection gives a visualization of when this happens</td>
</tr>
<tr>
<td>(e) What percentage of the difference in voltage has the capacitor been charged to after ( \tau ) seconds?</td>
</tr>
</tbody>
</table>
Directions, Part 1

1. Download the file Lab3Part1Script.mw and open it.

2. Do Edit->Execute->Worksheet and execute the worksheet. Verify that it's working correctly by comparing the output to what you believe is the correct answer.

3. Modify the worksheet to run the following version of the problem:

   **Problem A, Version 2**
   
   For a particular capacitor/battery set up, we find
   
   \[ \text{Vi}=37, \text{Va}=251, k=0.0075. \]
   
   (a) What is the value of \( \tau \), the time constant?
   
   (b) What is the voltage after 60 seconds?
   
   (c) What is the voltage after 120 seconds?
   
   (d) How long does it take the voltage to reach 220 volts? Plot the voltage curve and a horizontal line whose intersection gives a visualization of when this happens
   
   (e) What percentage of the difference in voltage has the capacitor been charged to after \( \tau \) seconds?

After you make your changes to the worksheet's parameters, you can do Edit->Execute->Worksheet to run the whole thing again. Verify that the worksheet is still working plausibly.

Save version 2 as YourNameLab3-A2.mw.

4. Here is a similar but different problem. Huddle with your group and figure out how to solve the problem. Everyone in the group should then modify their previous script to solve this new problem. You will need to do more than change the parameter values, you will have to change the instructions that occur in the script. The modifications revolve around figuring out how you find \( k \). That in turn will cause you to change which parameters the script has. Because of the great similarity between the problems, it's easier to edit a copy of your original script than to type things in all over again.

This happens all the time in programming -- once you have something successful, you run into variations that require reprogramming rather than just re-execution of a fixed script.

   **Problem B, Version 1**
   
   For a particular capacitor/battery set up, we find
   
   \[ \text{Vi}=37, \text{Vmax}=251. \]
   
   (a) We find that after 25 seconds, the voltage has risen from its initial value to 72.0 volts. What is the value of \( k \)?
   
   (b) What is the value of \( \tau \), the time constant?
   
   (c) What is the voltage after 40 seconds?
   
   (d) What is the voltage after 60 seconds?
   
   (e) How long does it take the voltage to reach 220 volts? Plot the voltage curve and a horizontal line whose intersection gives a visualization of when this happens
   
   (f) What percentage of the difference in voltage has the capacitor been charged to after \( \tau \) seconds?

5. Save this script as YourNameLab3B-1.mw. Lab partners should exchange their scripts and move onto the next problem.
6. Check out your partner's script by using it to solve the problem below. Verify that they've done an adequate job of parameterizing by observing that all that is needed to solve B version 2 is to change values of some of the parameters and re-execute the script.

**Problem B, Version 2**

For a particular capacitor/battery set up, we find

\[ V_i = 30, \quad V_{\text{max}} = 274. \]

(a) We find that after 30 seconds, the voltage has risen from its initial value to 101.0 volts. What is the value of \( k \)?

(b) What is the value of \( \tau \), the time constant?

(c) What is the voltage after 40 seconds?

(d) What is the voltage after 60 seconds?

(e) How long does it take the voltage to reach 220 volts? Plot the voltage curve and a horizontal line whose intersection gives a visualization of when this happens.

(f) What percentage of the difference in voltage has the capacitor been charged to after \( \tau \) seconds?

7. Save this version of the script as YourNameLab3B-2.mw. Mail all your scripts to yourself for future use.

**Part 1, problem 3 Answer**

\[ V_i = 37, \quad V_{\text{max}} = 251, \quad k = 0.0075 \]

**Start of parameters**

\[ V_i := 37 \]

37

(3.1)

\[ V_{\text{max}} := 251; \]

251

(3.2)

\[ k := 0.0075 \]

0.0075

(3.3)

\[ t1 := 40 \]

40

(3.4)

\[ t2 := 60 \]

60

(3.5)

\[ v_{\text{Target}} := 220 \]

220

(3.6)
End of parameters

Define a function that describes the voltage. This is not a parameter because it doesn't change between versions of the problem.

\[ V := t \rightarrow ( Vi + (V_{\text{max}} - Vi) \cdot (1 - \exp(-kt))) \]

\[ t \rightarrow Vi + (V_{\text{max}} - Vi) \left( 1 - e^{-kt} \right) \quad (3.7) \]

Test the function at \( t=0 \) -- should be initial voltage.

\[ V(0) \]

\[ 37. \quad (3.8) \]

According to the notes, the time constant is the reciprocal of \( k \).

\[ \tau := \frac{1}{k} \]

\[ 133.3333333 \quad (3.9) \]

Since we have defined \( V \), all we have to do is evaluate \( V \) at \( t=20 \) and \( t=40 \).

\[ V(t1) \]

\[ 92.46490077 \quad (3.10) \]

\[ V(t2) \]

\[ 114.5475756 \quad (3.11) \]

To find how long it takes to reach the target voltage, we evaluate \( V \) at the symbol \( t \), and equate it to the target voltage. Solving that equation for \( t \) will give us the answer.

\[ \text{eqn} := \text{vTarget} = V(t) \]

\[ 220 = 251 - 214 e^{-0.0075t} \quad (3.12) \]

The target voltage is reached in \( \text{solve(eqn, t)} = 257.5985081 \) seconds.

To do a plot which gives the same kind of answer, plot \( V \) and a horizontal line (a "constant function") whose value is the target voltage. The intersection of the two lines indicates where the voltage reaches the target. For the range, we pick ten times the time
constant, which should be enough to see the exponential curve go into its "almost flat" phase. After all, when \( t \) is equal to ten times the time constant, \( V(t) \) should achieve \( 1 - (1 - 0.632)^{10} \approx 0.9999544511 \) of the final voltage \( V_{\text{max}} \).

\[
\text{plot}(\{V(t), v\text{Target}, t = 0..10\cdot\text{tau}, labels = ["time (in seconds)", "voltage"])
\]

\[
v_{\text{diff}} := V_{\text{max}} - V_i
\]

\[
v_{\text{gain}} := V(t) - V_i
\]

\[
\text{percentGain} := \frac{v_{\text{gain}}}{v_{\text{diff}}}
\]

End of script

Part 1, problem 5 Answer

We modify the script so that rather than having a parameter \( k \), it has two parameters "firstTime" and "firstVoltage". They are used to set up an equation that we can solve for \( k \).

\( V_i = 37, V_{\text{max}} = 251 \)

\texttt{restart}
Start of parameters

\[ Vi := 37 \]

\[ V_{\text{max}} := 251; \]

\[ \text{firstTime} := 25 \]

\[ \text{firstVoltage} := 72.0 \]

\[ t1 := 40 \]

\[ t2 := 60 \]

\[ v_{\text{Target}} := 220 \]

End of parameters

Define a function that describes the voltage. This is not a parameter because it doesn't change between versions of the problem.

\[ V := t \rightarrow ( Vi + (V_{\text{max}} - Vi) \cdot (1 - \exp(-kt))) \]

\[ t \rightarrow Vi + (V_{\text{max}} - Vi) \left(1 - e^{-kt}\right) \]  \hspace{1cm} (3.23)

Test the function at \( t=0 \) -- should be initial voltage.

\[ V(0) \]

\[ 37 \]  \hspace{1cm} (3.24)

\[ k_{\text{Equation}} := \text{firstVoltage} = V(\text{firstTime}) \]

\[ 72.0 = 251 - 214 e^{-25 \, k} \]  \hspace{1cm} (3.25)

\[ k := \text{solve}(k_{\text{Equation}}, k) \]

\[ 0.007143608367 \]  \hspace{1cm} (3.26)
According to the notes, the time constant is the reciprocal of $k$.

\[ \tau := \frac{1}{k} \]

139.9852776  

(3.27)

Since we have defined $V$, all we have to do is evaluate $V$ at $t=20$ and $t=40$.

\[ V(t1) = 90.18869157 \]  

(3.28)

\[ V(t2) = 111.5983251 \]  

(3.29)

To find how long it takes to reach the target voltage, we evaluate $V$ at the symbol $t$, and equate it to the target voltage. Solving that equation for $t$ will give us the answer.

\[ eqn := v_{\text{Target}} = V(t) \]

\[ 220 = 251 - 214 e^{-0.007143608367 t} \]  

(3.30)

The target voltage is reached in \( solve(eqn, t) = 270.4499899 \) seconds.

To do a plot which gives the same kind of answer, plot $V$ and a horizontal line (a "constant function") whose value is the target voltage. The intersection of the two lines indicates where the voltage reaches the target. For the range, we pick ten times the time...
constant, which should be enough to see the exponential curve go into its "almost flat" phase. After all, when \( \tau \) is equal to ten times the time constant, \( V(t) \) should achieve \( 1 - (1 - 0.632)^{10} = 0.9999544511 \) of the final voltage \( V_{\text{max}} \).

\[
plot([V(t), v\text{Target}], t = 0..10*\tau, labels = ["time (in seconds)", "voltage"])
\]

\[
v_{\text{diff}} := V_{\text{max}} - V_i
\]

\[
214
\] (3.31)

\[v_{\text{gain}} := V(\tau) - V_i
\]

\[
135.2737996
\] (3.32)

\[\text{percentGain := } \frac{v_{\text{gain}}}{v_{\text{diff}}}
\]

\[
0.6321205589
\] (3.33)

\textbf{End of script}

\textbf{Part 1, problem 6 Answer}

\( V_i = 30, V_{\text{max}} = 274. \)

\textit{restart}
Start of parameters

\[ V_i := 30 \]  
\[ V_{\text{max}} := 274; \]  
\[ \text{firstTime} := 30 \]  
\[ \text{firstVoltage} := 101.0 \]  
\[ t_1 := 40 \]  
\[ t_2 := 60 \]  
\[ v_{\text{Target}} := 220 \]

End of parameters

Define a function that describes the voltage. This is not a parameter because it doesn't change between versions of the problem.

\[ V := t \rightarrow ( V_i + (V_{\text{max}} - V_i) \cdot (1 - \exp(-k \cdot t))) \]

\[ t \rightarrow V_i + (V_{\text{max}} - V_i) \left(1 - e^{-k \cdot t}\right) \]

Test the function at \( t=0 \) -- should be initial voltage.

\[ V(0) \]

\[ 30 \]

\[ k_{\text{Equation}} := \text{firstVoltage} = V(\text{firstTime}) \]

\[ 101.0 = 274 - 244 e^{-30 \cdot k} \]

\[ k := \text{solve}(k_{\text{Equation}}, k) \]

\[ 0.01146255436 \]
According to the notes, the time constant is the reciprocal of $k$.

$$\tau := \frac{1}{k}$$

87.24058954 \hspace{1cm} (3.45)

Since we have defined $V$, all we have to do is evaluate $V$ at $t=20$ and $t=40$.

$$V(t_1)$$

119.7359027 \hspace{1cm} (3.46)

$$V(t_2)$$

151.3401639 \hspace{1cm} (3.47)

To find how long it takes to reach the target voltage, we evaluate $V$ at the symbol $t$, and equate it to the target voltage. Solving that equation for $t$ will give us the answer.

$$eqn := V_{\text{target}} = V(t)$$

$$220 = 274 - 244 e^{-0.01146255436 t}$$ \hspace{1cm} (3.48)

The target voltage is reached in $solve(eqn, t) = 131.5748769$ seconds.

To do a plot which gives the same kind of answer, plot $V$ and a horizontal line (a "constant function") whose value is the target voltage. The intersection of the two lines indicates where the voltage reaches the target. For the range, we pick ten times the time
constant, which should be enough to see the exponential curve go into its "almost flat" phase. After all, when t is equal to ten times
the time constant, V(t) should achieve \(1 - (1 - .632)^{10} = 0.9999544511\) of the final voltage \(V_{\text{max}}\).

\[
\text{plot}([V(t), vTarget], t = 0 .. 10 \cdot \text{tau}, \text{labels} = ["time (in seconds)", "voltage"])
\]

\[
v_{\text{diff}} := V_{\text{max}} - Vi
\]

\[
244
\] (3.49)

\(v_{\text{gain}}\) is the difference between the starting voltage and the voltage at \(t=\text{tau}\)

\[
v_{\text{gain}} := V(\text{tau}) - Vi
\]

\[
154.2374163
\] (3.50)

\(\text{percentGain}\) is the percentage gain -- a decimal number between 0 and 1.

\[
\text{percentGain} := \frac{v_{\text{gain}}}{v_{\text{diff}}}
\]

\[
0.6321205586
\] (3.51)
3.4 Part 2

Part 2 Description

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 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 linear functions: all functions \( g(x) = ax + b \) 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 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 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 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 \( ax + b \) 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: \( pData := [134.2, 142.5, 155.0, 159.8, 171.1, 184.2] \). Similarly, create a second list and assign it to \( tData \).

3. Produce a point plot with Maple using the techniques discussed in chapter 5 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?

Problem 2 Answer

\[
pData := [134.2, 142.5, 155.0, 159.8, 171.1, 184.2];
\]

\[
[134.2, 142.5, 155.0, 159.8, 171.1, 184.2] \quad (3.52)
\]

\[
tData := [0, 20.1, 39.8, 60.0, 79.9, 100.3];
\]

\[
[0, 20.1, 39.8, 60.0, 79.9, 100.3] \quad (3.53)
\]

\[
plot(tData, pData, color = "blue", style = point)
\]
with(CurveFitting):

LeastSquares([[0, 1], [1, 2], [2, 3], [3, 10]], v)

\[-\frac{1}{5} + \frac{14}{5} v\]  \hspace{1cm} (3.54)

LeastSquares(tData, pData, v)

\[133.5000490 + 0.4858370741 v\]  \hspace{1cm} (3.55)

formula := LeastSquares(tData, pData, t)

\[133.5000490 + 0.4858370741 t\]  \hspace{1cm} (3.56)

plot(formula, t = 0..100)
To figure out the pressure when $t=120$, we evaluate the formula: $eval(formula, t = 120) = 191.8004979$.

To estimate absolute zero, we set up the equation $0=formula$ and solve for $t$:

\[ temp := 0 \]

\[ 0 \]

\[ eqn := temp = formula \]

\[ 0 = 133.5000490 + 0.4858370741 \cdot t \]

\[ solve(eqn, t) \]

\[ -274.7835769 \]

The actual answer is -273.15 degrees Celsius. Given the imprecision of our experimental measurements, our estimate is not that much off.
3.5 Attachment: starter script for Part 1

Script for Lab 3, Part 1

CS 121 Computation Lab I

Fall 2010

This script was run by: (fill in here).

This script solves the following problem:
A capacitor connected to a battery charges according to the following formula

\[ V(t) = V_i + (V_{\text{max}} - V_i) \left(1 - e^{-kt}\right) \]

where \( V_i \) is the starting voltage, \( V_{\text{max}} \) is the final voltage, and \( k \) is the reciprocal of the time constant ( \( k = \frac{1}{\tau} \)). The voltage is measured in volts, the time in seconds.

Problem A, Version 1

For a particular capacitor/battery setup, we find

\( V_i = 63, V_{\text{max}} = 266, k = 0.09. \)

(a) What is the value of \( \tau \), the time constant?

(b) What is the voltage after 20 seconds?

(c) What is the voltage after 40 seconds?
(d) How long does it take the voltage to reach 256 volts? Plot the voltage curve and a horizontal line whose intersection gives a visualization of when this happens.

(e) What percentage of the difference in voltage has the capacitor been charged to after \( \tau \) seconds?

**Start of parameters**

\[ Vi := 63 \]  
\[ V_{\text{max}} := 266; \]  
\[ k := 0.09 \]  
\[ t_1 := 20 \]  
\[ t_2 := 40 \]  
\[ v_{\text{Target}} := 256 \]

**End of parameters**

Define a function that describes the voltage. This is not a parameter because it doesn't change between versions of the problem.

\[ V := t \rightarrow (Vi + (V_{\text{max}} - Vi) \cdot (1 - \exp(-k \cdot t))) \]

\[ t \rightarrow Vi + (V_{\text{max}} - Vi) \left(1 - e^{-kt}\right) \]

Test the function at \( t=0 \) -- should be initial voltage.

\[ V(0) \]

\[ 63. \]

According to the notes, the time constant is the reciprocal of \( k \).

\[ \tau := \frac{1}{k} \]

\[ 11.11111111 \]
Since we have defined $V$, all we have to do is evaluate $V$ at $t=20$ and $t=40$.

$$V(t_1)$$

$$\begin{align*}
232.4443257
\end{align*}$$

(3.69)

$$V(t_2)$$

$$\begin{align*}
260.4532844
\end{align*}$$

(3.70)

To find how long it takes to reach the target voltage, we evaluate $V$ at the symbol $t$, and equate it to the target voltage. Solving that equation for $t$ will give us the answer.

$$eqn := v_{\text{Target}} = V(t)$$

$$\begin{align*}
256 = 266 - 203 e^{-0.09t}
\end{align*}$$

(3.71)

The target voltage is reached in $solve(eqn, t) = 33.45134318$ seconds.

To do a plot which gives the same kind of answer, plot $V$ and a horizontal line (a "constant function") whose value is the target voltage. The intersection of the two lines indicates where the voltage reaches the target. For the range, we pick ten times the time constant, which should be enough to see the exponential curve go into its "almost flat" phase. After all, when $t$ is equal to ten times the time constant, $V(t)$ should achieve $1 - (1 - 0.632)^{10} = 0.9999544511$ of the final voltage $V_a$.

$$plot(\{V(t), v_{\text{Target}}\}, t = 0..10 \cdot \text{tau}, labels = ["time (in seconds)", "voltage"]$$

![Plot of voltage over time]

$$v_{\text{diff}} := V_{\text{max}} - V_i$$

$$\begin{align*}
203
\end{align*}$$

(3.72)
vgain is the difference between the starting voltage and the voltage at $t=\tau$

$$vgain := V(tau) - Vi$$

$$128.3204734$$  \hfill (3.73)

percentGain is the percentage gain -- a decimal number between 0 and 1.

$$percentGain := \frac{vgain}{vdiff}$$

$$0.6321205586$$  \hfill (3.74)

End of script

3.6 Final actions (end of class)

Upload all of your work to Blackboard, or email copies to yourself and/or your partners.

3.7 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. 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 2010 Directions and Problems

4.1 Lab 4 Overview

Overview

The lab explores the problems of finding artillery trajectories that satisfy certain requirements and constraints. 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 land in a small net at the opposite end of the circus arena. Your job as Blammo's manager is to do the mathematical calculations that will allow Blammo to perform his death-defying act safely. The methods that you will use are from the field of ballistics (the study of projectile motion)."

![Human cannonball](http://www.phpsolvent.com/wordpress/?p=1263)

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

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

where \( g = 32 \frac{\text{ft}}{\text{sec}^2} = 9.8 \frac{\text{m}}{\text{sec}^2} \), 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

\[
v0 := 100
\]

\[
100
\]
We can develop a plot, using the `paramplot` feature of Maple's plots package:

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

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

(4.2)

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

\[
50 \sqrt{2}
\]  

(4.3)

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

\[
50 \sqrt{2}
\]  

(4.4)

\[
g := 32
\]

\[
32
\]  

(4.5)

\[
y0 := 0
\]

\[
0
\]  

(4.6)

\[
x0 := 0
\]

\[
0
\]  

(4.7)

\[
\text{xpos} := (t) \rightarrow x0 + v0x \cdot t
\]
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.

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

\[
0, \quad \frac{25}{8} \sqrt{2}
\]

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.

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

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

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

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

\[
312.5000000
\]

Evidently Blammo travels 312.5 feet.
Now, if we plot from \( t=0 \) to flightTime, we should see the whole plot:

\[
\text{plot([xpos(t), ypos(t), } t = 0 \text{..flightTime], labels = ['"feet", ',"feet"]])}
\]

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 \((\text{xpos}(t), \text{ypos}(t))\) for any given time \(t\), and then gives it to the animate function for \(t=0..\text{flightTime}\).

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

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

\(\text{with(plots)}:\)

\[
\text{animate(drawBlammo, [t, } t = 0 \text{..flightTime])}
\]

Maple has other useful functions to help us:

\(\text{maximize}\) will find the largest value that a function attains. For example,

\[
\text{evaf}('\text{maximize(ypos(t))})
\]

\[
78.12500000
\]

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,

\[
\text{with(plottools)} : \\
\]

\[
p1 := \text{line}([0, 50], [200, 50], \text{linestyle} = "\text{dash}", \text{color} = "\text{green}\") \\
\quad \text{CURVES}([[0., 50.], [200., 50.]], \text{COLOUR(RGB, 0., 1.000000000, 0.), LINESTYLE(3)}) \tag{4.15}
\]

\[
p2 := \text{line}([200, 0], [200, 50], \text{color} = "\text{green}\", \text{linestyle} = "\text{dash}\") \\
\quad \text{CURVES}([[200., 0.], [200., 50.]], \text{COLOUR(RGB, 0., 1.000000000, 0.), LINESTYLE(3)}) \tag{4.16}
\]
If we display them together we get a plot that looks like this:

```python
with(plots):

display([p1, p2])
```

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

```python
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.
5 Lab 1 CS 122 Computation Lab II Winter 2011 Directions and Problems

5.1 Lab 5 Overview

Overview

There are two parts to this lab. Part one has you developing a user-defined function from a "word specification" and applying it to solve a problem. The work involved includes determining how to translate a "human-friendly" description of a problem into the syntax of Maple, entering it into Maple, and troubleshooting what you enter and figuring out how to get it to work. This exercise makes the transition to the style more usual with programming work -- most of the directions are given in English, with the programmer handling the task of figuring out how to convert the intentions into the syntax of the programming language. The programmer is not left totally to their own devices -- usually there are some parts inherited from other workers and given as "use this". Other pieces can be lifted and edited from tutorial examples in a straightforward fashion. The problem-solving and mental English-to-programming language translation abilities are where most of the "value added" comes from.

Part two provides a guided experience in using Maple's code edit regions to easily segregate and execute user developed scripts.

Pre-lab preparation

1. Reading: review chapters 1-9 from cs 121. Read chapter 10 (new material).

2. Practice creating a code edit region (chapter 10), and replicate the operation of an example given in the readings, or one of your own devising.

3. Take the pre-lab quizlet 1 at the CS 122 Maple TA website. The deadline for doing quizlet 1 will be through the end of the first lab (week 2 of the quarter). This is an exception for the typical due date for quizlets. The pre-lab quizlets for Labs 2-4 will be available only in the week before the lab.

4. If you are feeling venturesome, try Part 0 of this lab on your own. It will save you time in the lab itself.

Directions for this lab

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

2. Listen to the instructor's overview of this week's 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 0 (15 minutes). This is a warm-up for the real lab work, to have you try out the following concepts: a) code edit regions and their execution, b) review of user-defined functions, c) review of combining multiple plots.

5. Work on Part 1 (20 minutes). We would like to see everyone end up with individual copies of the solution scripts. However, it may be more efficient this time to work as a pair at a single computer, and distribute the scripts to both partners via email or other file exchange at the end. Your work should use only textual versions of Maple operations. Avoid using the clickable interface to perform calculations. You will not receive credit for answers that invoke operations such as solve through the clickable interface. However, you may use the clickable interface for entering expressions (e.g. square roots) or symbolic constants (e.g. π).


7. Work on Part 3 (30 minutes).
5.2 Part 0 -- verify your understanding of basic features through experimentation

Code edit region warm up

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 – it contains syntax errors – we can practice troubleshooting code here)

   #This is a script to compute the hypotenuse of a triangle with sides s1 and s2.
   s1:=6;
   s2:=8;
   Hyp := SQRT((s1^2 – s2^2)

Checkpoint: What you should have should look like this:

5. Execute the faulty 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:
6. What we want in the code edit region is a script that computes the hypotenuse of a right triangle respectively. Edit the region to fix up the script so that it does this. What you were given in step 5 does not do this -- it has a number of errors that are there for you to find and fix. Some of the mistakes are syntax errors (missing parentheses, wrong characters), some are logic errors (e.g. the wrong formula is specified, or the wrong name used for a function). Re-execute the region until you see the following, which is the sign that the hypotenuse has been computed correctly.

In order to do this step, you will need to have a firm notion of what you are aiming for, since we're not showing you what changes to make. If you're uncertain and can't figure out it after a few minutes of experimentation, ask.

**Result of executing code region with all errors fixed.**

Like all scripts, it should be possible to get a different computation going by changing the assigned values of parameters s1 and s2, but not the computation line. For example, try editing the code edit region changing the first two lines to s1 := 9; and the second line to s2 := 13. You should then be able to re-execute the code region *without changing the third line* and see the result of the different computation:
Result of re-executing script with different parameter values.

\[
\begin{align*}
    s1 & := 9 \\
    s2 & := 12 \\
    Hyp & := 15
\end{align*}
\]

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

Result of collapsing code edit region

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

**User-defined function review warm up**

In this section, we review how to define a function and then invoke it. This was first discussed in Chapters 7 and 8, read previously.

In a code edit region, enter the following as a function definition. Executing the region should produce confirmation that the function has been defined.

Code region after entering a function definition

You should now be able to use the function by adding lines that invoke the function. For example, adding the following lines

Hypot(6,8); #should compute 10
Hypot(9,12); #should compute 15

and re-executing the code region should produce the following results:

<table>
<thead>
<tr>
<th>Result of defining a function and invoking it, in a code edit region</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Hypot := (leg1, leg2) -&gt; sqrt(leg1^2 + leg2^2);</code></td>
</tr>
<tr>
<td>Hypot(6, 8); #should produce 10</td>
</tr>
<tr>
<td>Hypot(9, 12); #should produce 16</td>
</tr>
</tbody>
</table>

Notice that the comments are helpful to the person reading the code, but do not cause any computational or output effects.

**Multiplotting warm up**

This is a review of the material in chapter 9, read previously. Recall that the general gist of creating a plot with multiple graphs in it is to first create plot structures with the individual plots, and then to call the `display` function of the `plots` package with the list of plot structures. The title and labels of the graph can be given as additional parameters to `display`.

1. In a code edit region, generate the a plot of a parabola by entering

   `plot(t^2, t=-10..10, color="blue");`

   and then executing the code region. You should see the plot below the code region:
2. Now edit the region so that rather than generating a picture, the plot is assigned to the variable \( P1 \). Re-execute the code region. You should now see the plot structure result in the execution trace.

**Parabolic plot generated by executing a code edit region**
3. Add onto the code edit region the instructions that will generate a point plot and assign it to the variable P2:

\[ T := \{-10, -5, 0, 5, 10\}; \]
\[ X := [90.3, 23, 0.1, 27, 101.2]; \]
\[ P2 := \text{plot}(T, X); \]

Re-execute the code region and you should see:

Parabolic plot generated by executing a code edit region

\[
P1 := \text{plot}(t^2, t=-10..10, \text{color}="\text{blue}"); \\
T := [-10, -5, 0, 5, 10]; \\
X := [90.3, 23, 0.1, 27, 101.2]; \\
P2 := \text{plot}(T, X); \\
\]

\[ PLOT(\ldots) \]
\[ [-10, -5, 0, 5, 10] \]
\[ [90.3, 23, 0.1, 27, 101.2] \]
\[ PLOT(\ldots) \]
4. Now display the two plots together by adding the instruction:

plots(display)([P1, P2]);

Re-executing the code edit region should show both plots together:

5. We can add a title and a label for the multi-plot by editing the code edit region by inserting the lines

```
titleName := "2 graph plot example";
labelNames := ["time", "position"];
```

and editing the display line to include title and label arguments:

```
plots(display)([P1, P2], title = T, labels = L);
```

Re-executing the region should produce the following result:
6. Add an appropriate comment into the code edit region and collapse it so it looks like this:

```plaintext
P1 := plot(t^2, t=-10..10, color="blue");
T := [ -10, -5, 0, 5, 10];
X := [ 90.3, 23, 0.1, 27, 101.2];
P2 := plot(T,X);
titleName := "2 graph plot example";
labelNames := ["time", "position"];
plots[display](P1,P2,title=titleName, labels=labelNames);
```

Execution trace from step 6
Note that we have developed this script incrementally, adding a few lines at a time and then trying out what we added. Developing the entire script in this fashion allows us to spot and fix mistakes more easily, because we can concentrate our bug-fixing attention on the few lines that we've newly added, rather than the entirety of the script.
5.3 Part 1 -- function design and testing (20 minutes)

This is an exercise to practice your ability to design and implement a user-defined function from a specification.

The problem

Design and implement another user-defined function that, when given two plot structures \( p1 \) and \( p2 \), a string \( t \), and a list of strings \( labels \), displays the two plots together with the specified title \( t \), and labels. Give this function a mnemonic name (a name that is easy to remember and suggestive of its purpose) that suggests what it does.

For example, if you had a Danish boss and she told you to call your function \( toGraf \), then the following script would draw a single plot that displays two graphs together.

\[
\begin{align*}
\text{plotOne} & := \text{plot}(\sin(x), x=0..10); \\
\text{plotTwo} & := \text{plot}([1,2,3],[1,5,6], \text{color}="\text{DodgerBlue}");
\end{align*}
\]

\( \text{toGraf(plotOne, PlotTwo, "Two plots", ["time", "temperature (in Celsius)"])}; \)

Of course, if you were doing this for your own satisfaction, or to satisfy the graders for this course, you wouldn't use a Danish name (even if you understand Danish, but most of the people reading your code don't). In order to do Part 1, you should review the material on user-defined functions and the \textit{display} function in chapters 7 and 8 of the course readings from last term. You should discover how to get the \textit{display} function to display a title and labels through a combination of reading the on-line documentation and experimentation.

In order to demonstrate that your function works correctly, create tests for it. Each test should do the following:

a) create plot structures by assigning to variables such as \( p1 \) and \( p2 \) the results of plots or built-in functions such as \textit{line}, \textit{circle} or \textit{disk}. (Recall the material in Lab 4 CS 121 about the need to use \textit{with(plots)}: and \textit{with(plottools)}:

b) Assign a string to a variable.

c) Invoke your function using your results from a) and b) as arguments to the function. You should expect a picture as a result,. You should also know what picture to expect given that you created the function and all the inputs to it.

d) Once you've tested your function to your satisfaction, exchange your file with the function definition only with one of your lab partners. You should test their function with your test, and vice versa. If the tests uncover a problem, it should be fixed.

5.4 Part 2 -- working with code regions (30 minutes)

The Human Cannonball (Blammo) problem treated in Lab 4 of CS 121 used a mathematical model that did not take into account air resistance. We now will use a more realistic mathematical model that takes air resistance into account, and compare the results of the two models.

In the original model, the functions for \( x \) and \( y \) position were given by:

\[
\begin{align*}
\text{xpos} & := (t) \rightarrow x0 + v0 \cdot \cos(\theta) \cdot t \\
\text{ypos} & := (t) \rightarrow y0 + v0 \cdot \sin(\theta) \cdot t - \frac{1}{2} \cdot g \cdot t^2
\end{align*}
\]

where \( t \) is time (in seconds), \( v0 \) is the initial velocity, \( \theta \) the initial angle of the cannon (in radians), and \( x0, y0 \) the initial horizontal and vertical position. and \( g \) is the Earth’s gravitational constant (32 feet/ sec\(^2\), or 9.8 meters/ sec\(^2\) ).

(a) Download and open the worksheet CS122Lab1Starter.mw It contains a script for running a version of the original "no air resistance" model of Blammo's flight. Note that the script is an a code edit region and it has comments and print statements. Execute the script (see Chapter 10). Observe how the results are printed out both as a result of executing an instruction, or through a print/printf instruction.
(b) Edit the code region to rerun this starter script with an initial position of (0,0), θ = 35 degrees, and v0 = 40 meters/second. Your computation and results should be displayed in metric units, so you should figure out how to adjust the computation to do this. In the sample output below, we have suppressed most of the execution trace by changing the semi-colons to colons.

<table>
<thead>
<tr>
<th>Code region output with most output suppressed with colons, initial firing angle of 47 degrees and firing velocity of 50 meters per second.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ t \rightarrow v0xt ]</td>
</tr>
<tr>
<td>[ t \rightarrow y0 + v0yt - \frac{1}{2} gt^2 ]</td>
</tr>
<tr>
<td>47 degrees converts to 0.820305 radians.</td>
</tr>
<tr>
<td>With an initial velocity of 50.000000 meters/sec and firing angle of 0.820305 radians, Blammo lands 2.544806e+02 meters away, reaching a maximum height of 59.326756 meters.</td>
</tr>
</tbody>
</table>

If you run into trouble making these changes, part of your work will be to practice your trouble shooting skills. If there are Error message, see if they match of the ones discussed in Chapter 10 or earlier chapters. If they match, you should try to fix them yourself. If there are no error messages but the results are not correct, employ the "comment trick" explained in the troubleshooting sections of Chapter 9 of the readings to identify which line is the first one that causes problems. The lab staff will expect you to apply these measures and analyze the results as a prerequisite to discussing troubleshooting problems with you.

1. Execute the code region and look at the execution trace, note what problems there are. Find the first line of the trace in which you detect a problem. If you can determine for the output what is causing the problem, then fix the problem and do (a) again. If not, proceed to step (b).

2. Put a "#" in the beginning of the last line of the region. Execute the script. You should see one less line to the execution trace because the line has been turned into a comment and won't be executed.

Continue doing this, one line at a time, until the last line of the trace is the first line where you detected a problem in part a.

The line that you just commented out should have something going wrong in it -- perhaps an incorrect number, a missing - sign, or the wrong expression. Contemplate what you have found and see if you can figure out what is going wrong with it. Make the change, and execute the worksheet to see if that fixes it. Feel free to include other print or printf statements to help you confirm that the values of variables/results are what you think they are.
If you are not making progress in determining the cause of your problems after two minutes, then ask for help. This "think for a few minutes before seeking help" will boost a crucial skill in troubleshooting -- your ability to connect symptoms with causes, and your development of strategies that work for you in finding the cause of problems.

(c) In the example output, we have used colons to suppress much of the execution trace, leaving only the information we really want to see from the correct program. Try to do this yourself after you are certain that things are working correctly.

5.5 Part 3 -- A new model, comparing two models (30 minutes)

In a model that takes wind resistance into account, the functions for $x$ and $y$ position can be described by

Horizontal position is given by

$$x_0 = \frac{mv0x}{b} \left( 1 + \frac{-bt}{m} \right)$$

Vertical position is given by

$$y_0 = \left( -mg - \frac{b0y}{b^2} + \frac{b0x}{b^2} + \frac{bt}{2m} \right)$$

where in addition to the symbols described previously, we have $m$ the mass of Blammo (60 kg), and $b=10$, the constant due to wind resistance. "e" in the formula is not a symbol, but invoking the exponential function.

Note that the original model that we worked with in Part 1 and the new model are talking about the same situation -- shooting Blammo out of a cannon. However, the formulae in the new model are more complicated, indicating that they are handling details (the effect of wind resistance) that were ignored in the formulae for the first model. Since all models choose to leave some details of reality out of the mathematical description. Engineers and scientists often then to do computations comparing the results of the models to understand whether the extra mathematics is producing better or less accurate results, or whether the two produce results that are about the same. While we don't have a "human cannonball lab" to measure what actually happens when people are fired out of a cannon, we can compare the results of the two models through computation.

(a) First, let's get the new model working. Start with a copy of what you did in Part 2. Modify the code edit region so that it uses the new formulae, and plots the trajectory for the new model. Plots with wind resistance should look something like this:

Blammo's flight with wind resistance

![Blammo's flight with wind resistance](image-url)
(b) Take the code outline file CS122Lab1Starter2.mw. It contains an outline for a script that combines the computation of the trajectories under the original and wind resistance models, and compares the result. The net result will be a plot of both trajectories. Proceed to develop the outline incrementally, as described in Chapter 9 of the readings. If you don't proceed incrementally, and enter the code for the entire computation before testing any piece of it, you will spend a long time tracking down where the mistakes are occurring.

The final output of your script should be a plot that has both trajectories plotted together, similar to the following:

**Comparison of trajectories: launch angle = 45 degrees, v0 = 40.000000 meters/sec**

Notice unlike in Part 1, you do not have to develop the tests yourself, you are given something to compare your results to.

(c) Once you have your code working, solve the following problems: (1) Calculate the percentage difference between the distance traveled with and without taking wind resistance into account, for $v_0=45$, $m=60$, angle $=45$. (2) How does the percentage difference change if the velocity is increased to $v_0=50$? $v_0=60$? What is the trend? (3) Blammo goes on a diet and now weighs only 57 kg. Repeat the analysis. Would you say that this kind of weight loss does or does not have a significant effect on the outcome?
5.6 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.

If you do not complete the lab in this session, you may ask the instructor for an appointment to complete the work in a catch-up session next week. The catch-up session is by appointment only. You must receive the instructor's permission to attend this session.

5.7 Summary and conclusion

In this lab, you have done three things. First, you have created a function from a written non-mathematical specification. You have tested that function and demonstrated to your satisfaction that it works correctly.

Secondly, you have learned about entering a script into a "code edit region" and executing it. New features of code in regions is that you can include comments (by putting a # before the comment), and have to separate the end of a statement from the beginning of the next one with a semi-colon (;).

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

Thirdly, you have modified the Blammo trajectory-calculating code to reflect a different mathematical model, which takes into account air resistance. While the overall architecture of the resulting code is very similar to the original Blammo, the results are noticeably different. If you were responsible for the safety of a real human cannonball, you might prefer the more complicated model because if you found it to produce more accurate results. Working on the modified Blammo code gave you additional experience with software development in code edit regions.
6 Lab 2CS 122 Computation Lab IIWinter 2011Directions and Problems

6.1 Overview

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

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

6.2 Pre-lab preparation

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

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

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

6.3 Directions for this lab

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

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

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

4. Work on Part 0 (15 minutes). This is a warm-up for the real lab work, to have you try out the following concepts: 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. Avoid using the clickable interface to perform calculations. You will not receive credit for answers that invoke operations such as solve through the clickable interface. However, you may use the clickable interface for entering expressions (e.g. square roots) or symbolic constants (e.g. π).

6.4 Problems -- Part 0 (15 minutes)

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

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

Problem 0.1

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

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

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

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

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

# 1st, bring in the necessary Maple plotting modules

with(plots); # for basic plotting and display
with(plottools);# for line plotting

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

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

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

Draw2lines(3,5,4);
```
Problem 0.2

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

(a) First, initialize two tables, use a loop to put items in them, and then convert them to a list. The variables xtabList and ytabList will have the items you put into the tables, in indexed order.

Test code for tables and loops (a)
# Initialize xtab and ytab to be (initially empty) tables
xtab := table(); # declare the Maple table for storing the x coordinate values
ytab := table(); # table for y coordinate values

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

# Run a repetitive loop that puts i into xtab[1], i^2 into ytab[1], 2 into xtab[2],
# 2*i into ytab[2]..... 10 into xtab[10], 2*i10 into ytab[10]
for i from 1 to numpts do
    xtab[i] := i;
    ytab[i] := i^2;
end do;

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

Sample output for (a)

```
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
</tr>
</tbody>
</table>

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

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

Sample output for (b)

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

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

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

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

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

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

# graph points

t := "Graph of y = 2 * x";  # title for graph
L := ["X axis", "Y axis"];  # labels for graphplot3d
plot(xtabList, ytabList, title=t, labels=L);

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

Problem 1

A function for drawing a red box

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

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

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

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

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

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

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

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

\[
\text{function name} := (\text{argument, argument, ... argument}) \rightarrow \text{expression involving the argument.}
\]

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

\[
drawBoxA := (\text{height, length}) \rightarrow \text{plots[display]}([\text{line(...), line(...), line(...), line(...)}])
\]

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

\[
drawBoxA := (h,l) \rightarrow \text{display([ line(...), line(...), line(...), line(...) ])}
\]

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

Either way should define the same function that does exactly the same thing when it is used. The function definition does not make any displaying occur right away. That only happens when the function is 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 height 5 and length 6.

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

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

d) Create another function `drawBoxB` that takes five arguments -- the length, width, the \((x,y)\) coordinates of the bottom left hand corner and a string that describes the color (see Maple's on-line help for `colornames` for a complete list). Uncomment the first test for `drawBoxB` and get it to work, then run the rest of the tests.
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

Art project 3
In the next lab we'll take this a little further by simulating a particle bouncing around in the box.

### 6.6 Problems -- Part 2 (45 minutes)

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

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

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

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

The simulation would then establish four variables $A$, $X$, $Y$, and $B$ that are initialized to the initial concentrations. Then it would conduct a loop of $n$ time steps. Each time step would establish the most recent values of the four concentrations as an add-on to the previous values, using the rules:

new $A = \text{previous } A - k_1 \times \text{previous } A \times \text{previous } X$

new $X = \text{previous } X + k_1 \times \text{previous } A \times \text{previous } X - k_2 \times \text{previous } X \times \text{previous } Y$

new $Y = \text{previous } Y + k_2 \times \text{previous } X \times \text{previous } Y - k_3 \times \text{previous } Y$

new $B = \text{previous } B + k_3 \times \text{previous } Y$

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

**Problem 2.1**

Open *Problem2-1Starter.mw*. Execute the script.

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

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

<table>
<thead>
<tr>
<th>Fragment of a multi-part graph for the chemical reactions</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Graph of multi-plot chemical reactions" /></td>
</tr>
</tbody>
</table>

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

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

6.7 Final actions (end of class)

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

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

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

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

Thirdly, you have modified the Blammo trajectory-calculating code to reflect a different mathematical model, which takes into account air resistance. While the overall architecture of the resulting code is very similar to the original Blammo, the results are noticeably different. If you were responsible for the safety of a real human cannonball, you might prefer the more complicated model because if you found it to produce more accurate results. Working on the modified Blammo code gave you additional experience with software development in code edit regions.
7 Lab 3CS 122 Computation Lab IIWinter 2011

Directions and Problems

7.1 Overview

This lab is about conditional execution, where different statements are executed depending on what is true at that particular point in the program's execution. Some of you who have taken Engineering 102 already have seen conditional execution in programming the robots, where the value of sensor input is used to conditionally cause the robot to turn one way or the other, or to stop and back up.

This lab revisits the trajectory calculations we did with Blammo, asking you to use a while loop to automatically find the angle to fire a projectile at in order to hit a target. This will give you practice at looping under conditional control (while). You are also asked to print out messages whenever a particular iteration of the loop encounters something of interest. This also requires conditional execution (if).

7.2 Pre-lab preparation

1. Read chapters 14 and 15 (new material). Review older chapters and labs as needed. Note that this lab expects you to remember your experience with the Blammo code last used in Lab 1/CS122 and Lab 4/CS121, and to have a copy handy so that you can reuse it. To start, you can practice by copying the examples in these chapters and getting them to work in your own copy of Maple. See whether you understand how to modify them to do slightly different things.
2. Take the pre-lab quizlet 3 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.

7.3 Directions for this lab

1. Form a lab group of two or three people. You are not allowed to work on your own without obtaining prior permission from the lab instructor. Group members should introduce themselves to each other if they haven't already met.
2. Listen to the instructor's overview of this week's lab. (15 minutes)
3. Get your copy of the verification sheet and check out which parts of the work will be verified.
4. Work on Part 0 (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 the four problems (80 minutes).
### 7.4 Part 0 (20 minutes) Self-guided demonstration of loops, if statements, and while

1. Rather than using `solve`, we can just try values and see if we can find any that work. Starting from the CS122Lab3Starter01.mw worksheet provided, complete the code region so that it looks and performs as below: a loop to print out values of \( \sin(\theta) \) and \( \cos(\theta) \). For which values of \( \theta \) does the code results say that sine and cosine functions are (approximately) equal?

```plaintext
#Find where sin() and cos() are equal
#Run this region to see the result in tabula format.

rad := 0:
sine := 0:
cosine := 0:
printf("%6s%10s%10s\n","Angle", "Sin", "Cos"); #Use formatted output with 6 columns, 10 columns, and 10 columns

for theta from 0 to 90 do
    rad := convert(theta, units, degree, radian):
sine := sin(rad):
cosine := cos(rad):
printf("%6d%10.4f%10.4f\n",theta, sine, cosine); #6 column integer, 10 columns with four decimal digits
end do:
```
<table>
<thead>
<tr>
<th>Angle</th>
<th>Sin</th>
<th>Cos</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>1</td>
<td>0.0175</td>
<td>0.9998</td>
</tr>
<tr>
<td>2</td>
<td>0.0349</td>
<td>0.9994</td>
</tr>
<tr>
<td>3</td>
<td>0.0523</td>
<td>0.9986</td>
</tr>
<tr>
<td>4</td>
<td>0.0698</td>
<td>0.9976</td>
</tr>
<tr>
<td>5</td>
<td>0.0872</td>
<td>0.9962</td>
</tr>
<tr>
<td>6</td>
<td>0.1045</td>
<td>0.9945</td>
</tr>
<tr>
<td>7</td>
<td>0.1219</td>
<td>0.9925</td>
</tr>
<tr>
<td>8</td>
<td>0.1392</td>
<td>0.9903</td>
</tr>
<tr>
<td>9</td>
<td>0.1564</td>
<td>0.9877</td>
</tr>
<tr>
<td>10</td>
<td>0.1736</td>
<td>0.9848</td>
</tr>
<tr>
<td>11</td>
<td>0.1908</td>
<td>0.9816</td>
</tr>
<tr>
<td>12</td>
<td>0.2079</td>
<td>0.9781</td>
</tr>
<tr>
<td>13</td>
<td>0.2250</td>
<td>0.9744</td>
</tr>
<tr>
<td>14</td>
<td>0.2419</td>
<td>0.9703</td>
</tr>
<tr>
<td>15</td>
<td>0.2588</td>
<td>0.9659</td>
</tr>
<tr>
<td>16</td>
<td>0.2756</td>
<td>0.9613</td>
</tr>
<tr>
<td>17</td>
<td>0.2924</td>
<td>0.9563</td>
</tr>
<tr>
<td>18</td>
<td>0.3090</td>
<td>0.9511</td>
</tr>
<tr>
<td>19</td>
<td>0.3256</td>
<td>0.9455</td>
</tr>
<tr>
<td>20</td>
<td>0.3420</td>
<td>0.9397</td>
</tr>
<tr>
<td>21</td>
<td>0.3584</td>
<td>0.9336</td>
</tr>
<tr>
<td>22</td>
<td>0.3746</td>
<td>0.9272</td>
</tr>
<tr>
<td>23</td>
<td>0.3907</td>
<td>0.9205</td>
</tr>
<tr>
<td>24</td>
<td>0.4067</td>
<td>0.9135</td>
</tr>
<tr>
<td>25</td>
<td>0.4226</td>
<td>0.9063</td>
</tr>
<tr>
<td>26</td>
<td>0.4384</td>
<td>0.8988</td>
</tr>
<tr>
<td>27</td>
<td>0.4540</td>
<td>0.8910</td>
</tr>
<tr>
<td>28</td>
<td>0.4695</td>
<td>0.8829</td>
</tr>
<tr>
<td>29</td>
<td>0.4848</td>
<td>0.8746</td>
</tr>
<tr>
<td>30</td>
<td>0.5000</td>
<td>0.8660</td>
</tr>
<tr>
<td>31</td>
<td>0.5150</td>
<td>0.8572</td>
</tr>
<tr>
<td>32</td>
<td>0.5299</td>
<td>0.8480</td>
</tr>
<tr>
<td>33</td>
<td>0.5446</td>
<td>0.8387</td>
</tr>
<tr>
<td>34</td>
<td>0.5592</td>
<td>0.8290</td>
</tr>
<tr>
<td>35</td>
<td>0.5736</td>
<td>0.8192</td>
</tr>
<tr>
<td>36</td>
<td>0.5878</td>
<td>0.8090</td>
</tr>
<tr>
<td>37</td>
<td>0.6018</td>
<td>0.7986</td>
</tr>
<tr>
<td>38</td>
<td>0.6157</td>
<td>0.7880</td>
</tr>
<tr>
<td>39</td>
<td>0.6293</td>
<td>0.7771</td>
</tr>
<tr>
<td>40</td>
<td>0.6428</td>
<td>0.7660</td>
</tr>
<tr>
<td>41</td>
<td>0.6561</td>
<td>0.7547</td>
</tr>
<tr>
<td>42</td>
<td>0.6691</td>
<td>0.7431</td>
</tr>
<tr>
<td>43</td>
<td>0.6820</td>
<td>0.7314</td>
</tr>
<tr>
<td>44</td>
<td>0.6947</td>
<td>0.7193</td>
</tr>
<tr>
<td>45</td>
<td>0.7071</td>
<td>0.7071</td>
</tr>
<tr>
<td>46</td>
<td>0.7193</td>
<td>0.6947</td>
</tr>
<tr>
<td>47</td>
<td>0.7314</td>
<td>0.6820</td>
</tr>
<tr>
<td>48</td>
<td>0.7431</td>
<td>0.6691</td>
</tr>
<tr>
<td>49</td>
<td>0.7547</td>
<td>0.6561</td>
</tr>
<tr>
<td>50</td>
<td>0.7660</td>
<td>0.6428</td>
</tr>
<tr>
<td>51</td>
<td>0.7771</td>
<td>0.6293</td>
</tr>
<tr>
<td>52</td>
<td>0.7880</td>
<td>0.6157</td>
</tr>
<tr>
<td>53</td>
<td>0.7986</td>
<td>0.6018</td>
</tr>
<tr>
<td>54</td>
<td>0.8090</td>
<td>0.5878</td>
</tr>
</tbody>
</table>
2. Now modify the code edit region so that it uses an if statement to only print the message when sin(theta) and cos(theta) are within a tolerance. You should see lines of output only for the lines where sin and cos agree. How many lines do you find?

```c
# Use sin() and cos() in a simple 'for' loop for angle from 0 to 90 degree, and use 'if' selection (conditional) statement to print out only the angle whose sine is equal to cosine.

rad := 0;
sine := 0;
cosine := 0;
tol := 1e-6: #tolerance of comparison

printf("%6s%10s%10s\n","Angle", "Sin", "Cos");

for theta from 0 to 90 by .01 do
    rad := convert(theta, units, degree, radian):
    sine := sin(rad):
    cosine := cos(rad):
    if (abs(evalf(sine) - evalf(cosine)) < tol) then
        printf("%6.3f\%10.4f\%10.4f\n",theta, sine, cosine);
    end if;
```
3. The problem with this kind of search is that even once a suitable value is found, the loop continues on to its limits. In order to stop it once it has found the goal of the search, modify your code edit region to use a **while** instead of a **for**.

```plaintext
#Use 'while' repetition statement to print out only the angles whose cosine is greater or equal to sine.
theta := 0:
rad := 0:
sine := 0:
cosine := 1:
printf("%6s%10s%10s\n","Angle", "Sin", "Cos");
while (cosine >= sine) do
    printf("%6.3f%10.4f%10.4f\n", theta, sine, cosine);
    theta := theta + 1;
    rad := convert(theta, units, degree, radian):
    sine := evalf(sin(rad)):
    cosine := evalf(cos(rad)):
end do:
```
<table>
<thead>
<tr>
<th>Angle</th>
<th>Sin</th>
<th>Cos</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td>0.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>1.000</td>
<td>0.0175</td>
<td>0.9998</td>
</tr>
<tr>
<td>2.000</td>
<td>0.0349</td>
<td>0.9994</td>
</tr>
<tr>
<td>3.000</td>
<td>0.0523</td>
<td>0.9986</td>
</tr>
<tr>
<td>4.000</td>
<td>0.0698</td>
<td>0.9976</td>
</tr>
<tr>
<td>5.000</td>
<td>0.0872</td>
<td>0.9962</td>
</tr>
<tr>
<td>6.000</td>
<td>0.1045</td>
<td>0.9945</td>
</tr>
<tr>
<td>7.000</td>
<td>0.1219</td>
<td>0.9925</td>
</tr>
<tr>
<td>8.000</td>
<td>0.1392</td>
<td>0.9903</td>
</tr>
<tr>
<td>9.000</td>
<td>0.1564</td>
<td>0.9877</td>
</tr>
<tr>
<td>10.000</td>
<td>0.1736</td>
<td>0.9848</td>
</tr>
<tr>
<td>11.000</td>
<td>0.1908</td>
<td>0.9816</td>
</tr>
<tr>
<td>12.000</td>
<td>0.2079</td>
<td>0.9781</td>
</tr>
<tr>
<td>13.000</td>
<td>0.2250</td>
<td>0.9744</td>
</tr>
<tr>
<td>14.000</td>
<td>0.2419</td>
<td>0.9703</td>
</tr>
<tr>
<td>15.000</td>
<td>0.2588</td>
<td>0.9659</td>
</tr>
<tr>
<td>16.000</td>
<td>0.2756</td>
<td>0.9613</td>
</tr>
<tr>
<td>17.000</td>
<td>0.2924</td>
<td>0.9563</td>
</tr>
<tr>
<td>18.000</td>
<td>0.3090</td>
<td>0.9511</td>
</tr>
<tr>
<td>19.000</td>
<td>0.3256</td>
<td>0.9455</td>
</tr>
<tr>
<td>20.000</td>
<td>0.3420</td>
<td>0.9397</td>
</tr>
<tr>
<td>21.000</td>
<td>0.3584</td>
<td>0.9336</td>
</tr>
<tr>
<td>22.000</td>
<td>0.3746</td>
<td>0.9272</td>
</tr>
<tr>
<td>23.000</td>
<td>0.3907</td>
<td>0.9205</td>
</tr>
<tr>
<td>24.000</td>
<td>0.4067</td>
<td>0.9135</td>
</tr>
<tr>
<td>25.000</td>
<td>0.4226</td>
<td>0.9063</td>
</tr>
<tr>
<td>26.000</td>
<td>0.4384</td>
<td>0.8998</td>
</tr>
<tr>
<td>27.000</td>
<td>0.4540</td>
<td>0.8910</td>
</tr>
<tr>
<td>28.000</td>
<td>0.4695</td>
<td>0.8829</td>
</tr>
<tr>
<td>29.000</td>
<td>0.4848</td>
<td>0.8746</td>
</tr>
<tr>
<td>30.000</td>
<td>0.5000</td>
<td>0.8660</td>
</tr>
<tr>
<td>31.000</td>
<td>0.5150</td>
<td>0.8572</td>
</tr>
<tr>
<td>32.000</td>
<td>0.5299</td>
<td>0.8480</td>
</tr>
<tr>
<td>33.000</td>
<td>0.5446</td>
<td>0.8387</td>
</tr>
<tr>
<td>34.000</td>
<td>0.5592</td>
<td>0.8290</td>
</tr>
<tr>
<td>35.000</td>
<td>0.5736</td>
<td>0.8192</td>
</tr>
<tr>
<td>36.000</td>
<td>0.5878</td>
<td>0.8090</td>
</tr>
<tr>
<td>37.000</td>
<td>0.6018</td>
<td>0.7986</td>
</tr>
<tr>
<td>38.000</td>
<td>0.6157</td>
<td>0.7880</td>
</tr>
<tr>
<td>39.000</td>
<td>0.6293</td>
<td>0.7771</td>
</tr>
<tr>
<td>40.000</td>
<td>0.6428</td>
<td>0.7660</td>
</tr>
<tr>
<td>41.000</td>
<td>0.6561</td>
<td>0.7547</td>
</tr>
<tr>
<td>42.000</td>
<td>0.6691</td>
<td>0.7431</td>
</tr>
<tr>
<td>43.000</td>
<td>0.6820</td>
<td>0.7314</td>
</tr>
<tr>
<td>44.000</td>
<td>0.6947</td>
<td>0.7193</td>
</tr>
<tr>
<td>45.000</td>
<td>0.7071</td>
<td>0.7071</td>
</tr>
</tbody>
</table>

4. Extend your code so that it produces an animation of the values of sin and cos for the various values of the angle $\theta$. Why does the last frame of the animation show $\sin(\theta) > \cos(\theta)$ if the while condition says to continue the repetition only while cosine $\geq$ sine?

```python
# Use 'while' repetition statement to print out only the angles whose cosine is greater or equal to sine.

theta := 0:
```
rad := 0:
sine := 0:
cosine := 1:

frames := table(): #An empty 'table' data structure to store frames of the animation
with(plots): #Enable use of display to produce animation
with(plottools): #Enable use of "point" procedure for producing a point.

printf("%6s%10s%10s\n","Angle", "Sin", "Cos");

while (cosine >= sine) do
    printf("%6.3f%10.4f%10.4f\n",theta, sine, cosine);
    theta := theta + 1;
    rad := convert(theta, units, degree, radian):
    sine := evalf(sin(rad)):
    cosine := evalf(cos(rad)):
    sinpos := point([theta, sine], color=red):
    cospos := point([theta, cosine], color=blue):
    frames[theta] := display([sinpos, cospos]):#Place into a table a plot of the sine and the cosine.
end do:

L := convert(frames,list):#This creates a list of plots in the same order.

display(L, insequence=true); #Display all the frames, as a movie.
<table>
<thead>
<tr>
<th>Angle</th>
<th>Sin</th>
<th>Cos</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td>0.000</td>
<td>1.000</td>
</tr>
<tr>
<td>1.000</td>
<td>0.0175</td>
<td>0.9998</td>
</tr>
<tr>
<td>2.000</td>
<td>0.0349</td>
<td>0.9994</td>
</tr>
<tr>
<td>3.000</td>
<td>0.0523</td>
<td>0.9986</td>
</tr>
<tr>
<td>4.000</td>
<td>0.0698</td>
<td>0.9976</td>
</tr>
<tr>
<td>5.000</td>
<td>0.0872</td>
<td>0.9962</td>
</tr>
<tr>
<td>6.000</td>
<td>0.1045</td>
<td>0.9945</td>
</tr>
<tr>
<td>7.000</td>
<td>0.1219</td>
<td>0.9925</td>
</tr>
<tr>
<td>8.000</td>
<td>0.1392</td>
<td>0.9903</td>
</tr>
<tr>
<td>9.000</td>
<td>0.1564</td>
<td>0.9877</td>
</tr>
<tr>
<td>10.000</td>
<td>0.1736</td>
<td>0.9848</td>
</tr>
<tr>
<td>11.000</td>
<td>0.1908</td>
<td>0.9816</td>
</tr>
<tr>
<td>12.000</td>
<td>0.2079</td>
<td>0.9781</td>
</tr>
<tr>
<td>13.000</td>
<td>0.2250</td>
<td>0.9744</td>
</tr>
<tr>
<td>14.000</td>
<td>0.2419</td>
<td>0.9703</td>
</tr>
<tr>
<td>15.000</td>
<td>0.2588</td>
<td>0.9659</td>
</tr>
<tr>
<td>16.000</td>
<td>0.2756</td>
<td>0.9613</td>
</tr>
<tr>
<td>17.000</td>
<td>0.2924</td>
<td>0.9563</td>
</tr>
<tr>
<td>18.000</td>
<td>0.3090</td>
<td>0.9511</td>
</tr>
<tr>
<td>19.000</td>
<td>0.3256</td>
<td>0.9455</td>
</tr>
<tr>
<td>20.000</td>
<td>0.3420</td>
<td>0.9397</td>
</tr>
<tr>
<td>21.000</td>
<td>0.3584</td>
<td>0.9336</td>
</tr>
<tr>
<td>22.000</td>
<td>0.3746</td>
<td>0.9272</td>
</tr>
<tr>
<td>23.000</td>
<td>0.3907</td>
<td>0.9205</td>
</tr>
<tr>
<td>24.000</td>
<td>0.4067</td>
<td>0.9135</td>
</tr>
<tr>
<td>25.000</td>
<td>0.4226</td>
<td>0.9063</td>
</tr>
<tr>
<td>26.000</td>
<td>0.4384</td>
<td>0.8998</td>
</tr>
<tr>
<td>27.000</td>
<td>0.4540</td>
<td>0.8910</td>
</tr>
<tr>
<td>28.000</td>
<td>0.4695</td>
<td>0.8829</td>
</tr>
<tr>
<td>29.000</td>
<td>0.4848</td>
<td>0.8746</td>
</tr>
<tr>
<td>30.000</td>
<td>0.5000</td>
<td>0.8660</td>
</tr>
<tr>
<td>31.000</td>
<td>0.5150</td>
<td>0.8572</td>
</tr>
<tr>
<td>32.000</td>
<td>0.5299</td>
<td>0.8480</td>
</tr>
<tr>
<td>33.000</td>
<td>0.5446</td>
<td>0.8387</td>
</tr>
<tr>
<td>34.000</td>
<td>0.5592</td>
<td>0.8290</td>
</tr>
<tr>
<td>35.000</td>
<td>0.5736</td>
<td>0.8192</td>
</tr>
<tr>
<td>36.000</td>
<td>0.5878</td>
<td>0.8090</td>
</tr>
<tr>
<td>37.000</td>
<td>0.6018</td>
<td>0.7986</td>
</tr>
<tr>
<td>38.000</td>
<td>0.6157</td>
<td>0.7880</td>
</tr>
<tr>
<td>39.000</td>
<td>0.6293</td>
<td>0.7771</td>
</tr>
<tr>
<td>40.000</td>
<td>0.6428</td>
<td>0.7660</td>
</tr>
<tr>
<td>41.000</td>
<td>0.6561</td>
<td>0.7547</td>
</tr>
<tr>
<td>42.000</td>
<td>0.6691</td>
<td>0.7431</td>
</tr>
<tr>
<td>43.000</td>
<td>0.6820</td>
<td>0.7314</td>
</tr>
<tr>
<td>44.000</td>
<td>0.6947</td>
<td>0.7193</td>
</tr>
<tr>
<td>45.000</td>
<td>0.7071</td>
<td>0.7071</td>
</tr>
</tbody>
</table>

Here are a few frames of the animation. Look carefully and you will see a little red and blue dot in each one, representing the value of sine and cosine at various angles.
5. In this part, we intentionally create and execute an "infinite loop", so that you will see what happens and learn how to bail out of it. Before you do this part, save your other work so that it isn't lost.

Open up CS122Lab3Starter02.mw and inspect the code in the code region. Why will this loop not stop?

Execute the code. Note the bottom left corner of the Maple worksheet window says "evaluating". Hit the red stop hand at the top of the Maple toolbar (to refresh your memory about where this is, see chapter 14 of the course readings.) How long does it take before the computation stops? How many times does the loop execute before it stops? The summation being performed is supposed to approximate $e^x$ for whatever value of x you're using.

Save your work so far.

Now uncomment the print statement in the loop and execute the code region again. What happens when you try to stop execution? How did you get yourself out of trouble?

For a value of x such as .01, eventually $\frac{x}{i!}$ will get very small, such as $10^{-6}$. Change tol to 10e-6 and rewrite the while condition as while term > tol. Rerunning the loop (without the print turned on) should allow you to observe the loop to stop all by itself. How many iterations did it take before it stopped?

7.5 Problems (80 minutes)

Recall in Lab 4, CS 121, we explored code that, given an firing angle and velocity, computed the trajectory and landing point of a human cannonball. In order to find the angle and velocity that would cause Blammo to land a specified distance away, we had to experimentally run the simulation script, trying different angles until we found one that worked.

In this part of the lab, we modify our trajectory code so that it can find the answer automatically by trying angles until it finds one that works. In particular we look at code that will answer this question:

Blimmo is shot out of his cannon at 40 m/sec. What angle should the cannon be aimed at in order for him to land approximately 60 meters away in a net that's 5 meters wide?
Problem 1

First, we'll practice with for loops. We assume that Blammo is launched at 40 meters per sec and use the wind resistance model to calculate where he lands. As before, we know that his mass is 60 kg and the wind resistance constant $b=10$.

What we want to do in this problem is to print a table of distances as we vary the angle from 30 degrees to 70 degrees. We want the script output to look like this:

Angle = 30 degrees, time = 3.703112 seconds, distance = 95.721287, max height = 16.710341
Angle = 31 degrees, time = 3.804913 seconds, distance = 96.609279, max height = 17.636593
Angle = 32 degrees, time = 3.905204 seconds, distance = 97.370883, max height = 18.573115
Angle = 33 degrees, time = 4.003975 seconds, distance = 98.008304, max height = 19.518697
Angle = 34 degrees, time = 4.101217 seconds, distance = 98.523747, max height = 20.472140
Angle = 35 degrees, time = 4.196919 seconds, distance = 98.919422, max height = 21.432256
Angle = 36 degrees, time = 4.291074 seconds, distance = 99.197540, max height = 22.397871
Angle = 37 degrees, time = 4.383671 seconds, distance = 99.360313, max height = 23.367824
Angle = 38 degrees, time = 4.474703 seconds, distance = 99.409952, max height = 24.340967
etc. etc.

Upload the trajectory code in CS122Lab3Starter.mw.

We intend to do produce the table by modifying the Blammo calculation code using the following code outline:

```plaintext
#initialize parameters for calculation, except for angle
for angle from 30 to 70 do
    #calculate distance
    #print a line of the table
end do;
```

Execute the code region, being careful that you are getting the expected results in the execution trace. You might do incremental development by first getting the "new" code to execute without giving errors, and then adding in the portion of the old code that does initialization, getting that to work, and then adding in the contents of the loop body.

Once you get the script to print out the table, you should be able to answer the question posed at the beginning of the problem.

Problem 2

Now rather than trying all the angles from 30 to 70, we'll try to find the smallest angle that will allow us to hit a target approximately 90 meters away. We develop another two parameters and change the looping control:

```plaintext
targetDistance := 90;
tol := 2.5; #Since Blammo is six feet tall, if he lands within 2.5 feet of the target he'll be okay.
... for angle from 30 to 75 while condition do ... end do;
```

You should invent a condition should stop the repetition when the distance computed is within tol meters of the targetDistance.

One issue that the condition will mention distanceWind, but before you execute the loop once, distanceWind will not have a value yet. The computer won't be able to decide if the condition is true or not and you will get an error Error, cannot determine if this expression is true or false: .....
You can fix this by initializing `distanceWind` with arbitrary value of infinity in the code before the loop. This will allow the loop to work correctly the first time. After that `distanceWind` will be given a properly computed value, so the loop will work correctly the second and subsequent times.

**Problem 3**

Once you have your code set up, answer the following question:

Suppose Blammo is being launched at a speed of 90 meters per second. We want to Blammo to land in a net 250 meters away. Find an angle that, taking wind resistance into account, will allow Blammo to hit the target. If there is more than one solution, find them all and decide which one is the best "show biz solution". You can do this by having an `if` statement look at the absolute value of the distance between the landing point and the target distance. If this is less than 5 meters, then the statement can print out a message with the angle, distance, and other information. If it is greater than 5 meters, then the `if` statement does nothing.

Be prepared to explain your reasoning. Feel free to alter your script so that it makes it easier to solve this problem.

**Problem 4**

Create a movie of the trajectories by drawing a plot each trip through the table, and putting it into a table, similar to how you did the point plot of the chemical simulation. At the end of the loop, convert the table into a list and use `display` to make it into a movie. Recall that if `L` contains a list of plots, then `plots[display](L, insequence=true);` will produce an animation rather than a multiplot.

We list a few of the frames below to give you an idea of what to expect,. The movie is more interesting to look at than the table of static plots, though.
7.6 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.

7.7 Summary and conclusion

In this lab, you practiced modifying and writing code that had conditional execution in it. `while (condition) do ... end do;` and `if ... then ... else ... endif`.

We used `while` to automatically try different firing angles until it found one that hit the target. Like many situations in `while` programming, the `condition` in the `while` is the logical opposite of what we're looking for -- we're looking for an angle that hits, so the `condition` is that the angle generates a miss. We used `if` to conditional print out messages.

In the moving particle simulation, we did different updates to the (x,y) location based on whether the particle hit a box wall during a time step. The alternatives for the updates were laid out in an `if ... then ... else ... endif`. The simulation was still controlled by a `for` repetition rather than a `while` because the number of steps to be taken could be determined in advance.

Conditional execution can be tricky, particularly when you have to write your own from scratch. Using `printf` to print out extra messages in the execution trace can help you more easily understand what is happening in a troubleshooting situation.
8 Lab 4CS 122 Computation Lab II Winter 2011 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 3, to control a loop's printing so that it only printed messages when it found projectile angles of interest, but did not issue messages otherwise. Some of you who have taken Engineering 102 already have seen conditional execution in programming the robots, where the value of sensor input is used to conditionally cause the robot to turn one way or the other, or to stop and back up.

Part 1 asks you to build a simulation of a particle rolling around a box. This gives you a further experience in conditional execution -- if statements. 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 16 (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. See whether you understand how to modify them to do slightly different things.

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

3. Take the pre-lab quizlet 4 at the CS 122 Maple TA website. 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 (20 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 three kinds of messages: Dean's List, passing, "on probation" and "suspended". Set up the program so that it flags probations for those with a gpa of less than 2.0 but greater than 1.0, suspension occurs with a gpa of less than 1.0. In order to do this, you will have to define values for two more variables, suspendedLevel and probationLevel. Then, modify the if statement in the loop to include some elif clauses: if ...>=deansLevel then.... elif ...<= suspended then .... elif .... <= probationLevel then .... else ... end if;
Your output should look something like this:

2.200000 is a satisfactory, non Dean's Level gpa.
0.700000 is a suspension Level gpa.
2.600000 is a satisfactory, non Dean's Level gpa.
1.500000 is a probation Level gpa.
2.900000 is a satisfactory, non Dean's Level gpa.
2.600000 is a satisfactory, non Dean's Level gpa.
3.200000 is a satisfactory, non Dean's Level gpa.
1.100000 is a probation Level gpa.
3.400000 is a satisfactory, non Dean's Level gpa.
3.000000 is a satisfactory, non Dean's Level 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, non Dean's Level 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.
1.700000 is a probation Level gpa.
0.100000 is a suspension Level gpa.
3.200000 is a satisfactory, non Dean's Level gpa.
2.500000 is a satisfactory, non Dean's Level gpa.
1.100000 is a probation Level gpa.
2.000000 is a satisfactory, non Dean's Level gpa.
3.000000 is a satisfactory, non Dean's Level gpa.
3.800000 is a Dean's List gpa.
2.400000 is a satisfactory, non Dean's Level gpa.

There are 3 students on the Dean's List.
There are 2 suspended students.
There are 12 students on probation.
There are 30 students total.

Even though there's quite a lot here to do that's need, 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 messages working. Finally, add the satisfactory messages so that the output looks exactly as above. 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 the course readings to find examples of loops with if-then statements that you can use as models.
8.4 Problems -- Part 1 (40 minutes)

In this part we are going to use while and for loops to run an simulation of a particle moving in a box. The approach taken is that we will give you model code that does one of the simple animations. You will then modify the code to do similar but different things.

Problem 1.1

Using old function to draw a box within a simulation code.

Open the file CS122Lab4Starter1.mw. Read the code. At one point there is an empty spot for the drawBoxB function that you built in Lab 2. Retrieve a copy of your code 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) Change the script so that it does not print during execution by putting a colon instead of a semi-colon at the end of the loop block.

b) Change the limit of the number of time steps to be 50 rather than 20. You should see the particle move a bit further. What happens if you set the number of time steps to be 200?

c) Save a copy of your for this problem as yourNameCS122Lab4Problem1-1.mw, for example TamiTaylorCS122Lab4Problem1-1.mw.

Problem 1.2

Your task in this problem is to extend the script to 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, you should 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 \((WID,y)\). 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).

| Figure 1 Particle hitting eastern wall |
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 \((WID - (x + \Delta x - WID), y)\) = \((2\cdot WID - x - \Delta x, y)\). That is, the \(x\)-coordinate of the location of the particle at the end of the time step when the bounce occurred is \(2\cdot WID - x - \Delta x\).
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

\[
\text{if (} \text{delx} > 0 \text{ and ptpos[1]} \geq \text{WID}) \# \text{bounce East} \\
\text{then} \\
xwallPos := \text{WID}; \\
\text{ptpos} := [2*xwallPos - ptpos[1] , ptpos[2]]; \\
delx := -delx; \\
dely := dely; \\
\text{end if;}
\]

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

\[
\text{if (} \text{delx} > 0 \text{ and ptpos[1]} \geq \text{WID}) \# \text{bounce East} \\
\text{then} \\
xwallPos := \text{WID}; \\
\text{ptpos} := [2*xwallPos - ptpos[1] , ptpos[2]]; \\
delx := -delx; 
\]
dely := dely;

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

To test this combined proposition, you can change the initial conditions in the script to test the western bounce first. Set

pos0 := [1,1];
delx := -0.1;
dely := 0.1;

and run for 50 time steps. You should see the particle move westwards and bounce off the western wall, then stop.

Once you have that working, set the script to run for 150 time steps. You should see the particle bounce off of both walls. If you want to play, set it for 250 time steps and you should see multiple bounces.

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.

We are following the "incremental development" approach, in that you have code that has a "bounce of eastern wall" working and you are trying to extend it to handle another case.

When you have completed this problem, save your work as yourNameCS122Lab4-1-2.mw.

**Problem 1.3**

To do this part, you will need the solution to Problem 1.2 from one of your lab partners. Get it (by swapping seats, or through file transfer) and open it up. 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

pos0 := [1,9];
delx := 0.0;
dely := 0.1;

and running the script for 150 time steps. To extend the script, you will have to figure out the bounce formulas for when the particle hits north or south. You can extend the script by using the following idea for a code outline for that portion of the code:

#check for bounce east/west and update ptpos, delx, dely if a bounce occurs

(include existing if statement)

#check for bounce north/south and update ptpos, delx, dely if a bounce occurs

(add an analogous if statement)

**Problem 1.4**

In the code developed so far, we first defined parameters LEN, WID, delx, and dely and assigned them values, and then referred to these values in the rest of the code through their symbolic names. We now reap an advantage of coding this way -- we can change the box size just by changing two lines -- the assignments to LEN and WID. The simulation will then work without further change for the new size box specified in that way.
Take your script from problem 1.2 and change the following lines:

```plaintext
pos0 := [8, 9]; # initial position of point.
ptpos := pos0; #initialize ptpos
delx := 0.1;
dely := 0.1;
```

(The second line doesn't change but we include it because the other lines do.).

Rerun the script and you should see the particle moving in a diagonal direction and bouncing off of walls.

```plaintext
WID := 2:
LEN := 10:
```

as well as

```plaintext
pos0 := [1,1]; # initial position of point.
ptpos := pos0; #initialize ptpos
delx := 0.2;
dely := 0.1;
```

Experiment with other values of the parameters and show the grader something else interesting of your own creation.

We can comment about why having parameters such as LEN and WID in the code saves programmers time in the long run. If we hadn't used parameters, then we could have skipped the lines of code assigning the parameters, e.g. `LEN := 10; WID := 10;` Wherever we had code that mentioned length, we would have to enter 10 directly rather than by mentioning the name LEN. This would be relatively straightforward, but consider what we would have to do if we wanted to then change the length of the box to 20 while keeping the width at 10. We would have to look at all the lines of code that had a "10" and figure out which ones had to do with the length and only change them. This would be much more work than what we did with this code -- just change a single assignment `LEN := 20;`.

### 8.5 Problems -- Part 2 (40 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 bubble 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 part 1. The x and y positions will be updated in a loop. Unlike the last Blammo calculation where we took wind resistance into account, our rules for time step motion will cause the particle to slow down only when it hits the ground, not while it's moving through the air. While the x velocity is constant (as it was in the particle-in-a-box simulation), the y velocity will be changing all the time due to the influence of gravity. Nevertheless, the overall structure of the program will look similar to that of part 1.

#### Problem 2.1

Recall that the artillery trajectory model in Lab 2 without air resistance, we used the following functions for position and velocity:

\[ v_{xy}(t) = \theta \]

\[ v_{yt}(t) = v_0 y - g \cdot t \]

\[ x_{pos}(t) = x_0 + t \cdot v_{xy}(t) \]

\[ y_{pos}(t) = y_0 + v_0 y \cdot t - \frac{g \cdot t^2}{2} \]

From this, we calculate

\[ \theta = \text{angle in radians} \]
v\theta_x = v\theta \cdot \cos(\theta)

v\theta_y = v\theta \cdot \sin(\theta)

We will update the movement of the ball in small time steps, in the way we did with Part 1. This contrasts to what we did with Blammo, where we had a formula for the entire movement and just evaluated it for various values of t.

First, 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 in 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 $xp + dt \cdot xv$ (present position plus the horizontal velocity times the amount of time of the time step)
  
  newxp := $xp + dt \cdot xv$;

- #new y position is approximately $yp + dt \cdot yv$
  
  newyp := $yp + dt \cdot yv$;

- #new x velocity is $xv$ (no change unless there's a bounce)
  
  newxv := $xv$;

- #new y velocity is $yv - g \cdot dt$
  
  newyv := $yv - g \cdot dt$;

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 yourNameLab4Part2-1.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 "nu", 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 of part 1 if you haven't already written this code.

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

change \( \text{newyv} \) to be \(-R\cdot\text{newyv}\). 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!

change \( \text{newxv} \) to be \( \eta\cdot\text{newxv} \). This will cause the horizontal velocity to slow due to fraction, but to continue in the same direction as before. There is no minus sign here.

change \( \text{newyp} \) to be \(-\text{newyp}\). 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 has 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 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.

**Problem 2.3**

Now save a copy of your work as `yourNameLabPart2-3.mw` and make further changes:

a) Introduce an accumulation variable totalTime. Initialize it to zero before the simulation loop, then add dt to it each trip through the loop. Use it to print out the total number of steps for the bouncing, along with the total horizontal distance traveled (approximately). The print statement needs to occur only after the loop is finished, as summary data, but when debugging your code you may want to put print statements inside the loop to check that the updates to the accumulation variable are happening correctly. You can then comment out or delete the debugging prints when you are satisfied that things are working.

b) 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.

c) 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.

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

Save your work.

**8.6 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.

**8.7 Summary and conclusion**

In this lab, you practiced modifying and writing code that had 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 a time step simulation 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 1CS 123 Computation Lab IIISpring 2011Directions and Problems

9.1 Overview

Part 1 of this lab asks you to build a simulation script 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 122 -- the Chemical reaction and Predator-Prey simulations, as well as the moving particle and bouncing ball simulations.

Part 2 asks you turn your simulation into a Maple procedure, using the new programming ideas given in Chapter 17 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.

9.2 Pre-lab preparation

1. Read chapter 17 of the course readings. Review older chapters, labs, and quizzes as needed so that you are reacquainted with the prior time-step simulations.
2. Read the rest of the 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 1 at the CS 123 Maple TA web site. You should do quizlet 1 before lab to be prepared for the first lab.
4. Practice building simple Maple procedures, such as the examples given in chapter 16. 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 adventurous, you can practice building the procedures of Part 0 before you come to lab.

9.3 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)

![Diagram of the air conditioned house]

**Fundamental modeling relationships**

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

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

(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>2.8·50 + 2.10·50 + 2.10·8 = 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 (in $ft^3$)</td>
<td>10·50·8 = 4000</td>
</tr>
<tr>
<td>$c_a$</td>
<td>Air specific heat (in $\frac{BTU}{lbm \cdot ft^3 \cdot °F}$)</td>
<td>0.24</td>
</tr>
</tbody>
</table>
The relationship between the rate of change of $T_{ew}$ and the interior air temperature and wall temperatures at that time can be modeled as:

$$\rho_{ew} L_{ew} c_{ew} \left( \frac{dT_{ew}(t)}{dt} \right) = U_{wi} (T_z(t) - T_{ew}(t)) + U_{wo} (T_0 - T_{ew}(t)) \quad (2)$$

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 ($\frac{BTU}{f^2 \cdot °F \cdot hr}$)</td>
<td>0.81</td>
</tr>
<tr>
<td>$c_{ew}$</td>
<td>Exterior wall specific heat ($\frac{BTU}{lbm \cdot f^2 \cdot °F}$)</td>
<td>21</td>
</tr>
<tr>
<td>$\rho_{ew}$</td>
<td>Exterior wall density ($\frac{lbm}{f^3 \cdot °F \cdot hr}$)</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 $A :=$ some expression involving old values of $A$, $B$, $X$ and $Y$;

... 

new value of $Y :=$ 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 $TEW[0]$, $TEW[1]$, etc.

Thus we are looking for a way of writing

$TZ[i+1] :=$ some expression involving older values of $TZ$ and $TEW$;

$TEW[i+1] :=$ 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 \( T_z \),

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

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

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

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

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

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

\[
TZ[i+1] \approx \frac{1}{\rho_a V_z c_a} \left( \rho_a V_z c_a T_z[i] + q_s - int + U_{wi} A_{ew} dt Tew[i] - U_{wi} A_{ew} dt Tz[i]ight.
\]

\[
\left. + U_{win} A_{win} dt T_0 - U_{win} A_{win} dt Tz[i] + q_{heater} dt + \rho_a Q_{ea} c_a dt T_{ea} - \rho_a Q_{ea} c_a dt Tz[i] \right)
\]

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 `solve` for TZ[i+1].

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

\[
T_{EW}[i+1] \approx \frac{\rho_{ew} l_{ew} c_{ew} Tew[i] - U_{wi} dt Tew[i] + U_{wi} dt Tz[i] + U_{wo} dt T_0 - U_{wo} dt Tew[i]}{\rho_{ew} l_{ew} c_{ew}} \quad (7)
\]

After doing such derivation in Maple, we could `lprint` 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:

\[
\]

\[
Tew[i+1] := (rho[ew]*l[ew]*c[ew]*Tew[i]-U[wi]*dt*Tew[i]+U[wi]*dt*Tz[i]+U[wo]*dt*T[0]-U[wo]*dt*Tew[i])/(rho[ew]*l[ew]*c[ew])
\]

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.
9.4 Part 0 -- problems

In this part, we practice building Maple procedures, which are explained in detail in Chapter 17 of the course readings. Maple procedures are an alternative way to the -> notation to defining a function in Maple. The procedure notation, while more verbose, is more powerful, in that the code in a procedure definition can include for/while loops or if statements.

1. First, some function definitions using the arrow (->) notation that we have used in CS 122. Recall that for the quadratic equation \( ax^2 + bx + c = 0 \), the two roots of the equation are \( \frac{-b + \sqrt{b^2 - 4ac}}{2a} \) and \( \frac{-b - \sqrt{b^2 - 4ac}}{2a} \).

(a) Define two functions \( X1 := (a,b,c) \rightarrow \) expression for the first root, and \( X2 := (a,b,c) \rightarrow \) expression for the second root.

(b) What are the two roots of \( x^2 - 6x + 10 = 0 \)? Compute the answer using the functions you defined in (a). Verify that the answers are correct.

(c) Similarly compute the two roots of \( x^2 + 6x + 10 = 0 \), and \( x^2 - 2x + 1 = 0 \). Confirm that your answers are correct.

2. Enter the following procedure definition to compute roots:

\begin{verbatim}
QuadForm:=proc(a,b,c)
    local d,X1,X2,numSolns;
    # compute the discriminent
d:=b*b-4.0*a*c;
    printf("a = %f, b = %f, c = %f: ",a,b,c);
    X1:=(-b+sqrt(d))/(2*a);
    X2:=(-b-sqrt(d))/(2*a);
    if(d<0) then
        # 2 imaginary solutions
        printf("%s", "both solutions are imaginary.");
        numSolns:=0;
    elif(d=0) then
        # 2 (duplicate) real solutions
        printf("%s", "both real solutions are same.");
        numSolns:=1;
    else
        # 2 distinct real solutions
        printf("%s", "2 distinct real solutions.");
        numSolns:=2;
    end if;
    return numSolns;
end proc;

# test the procedure. Should return the correct number of real solutions
QuadForm(1,6,10);
QuadForm(1,-2,1);
QuadForm(1,-6,8);
\end{verbatim}
Make up tests that provide convincing evidence that the procedure works as it should.

3. (a)

Download Lab1CS123Part0-3.mw. Identify the parts that: initialize tables, set the initial concentrations, and plot out the concentrations. Execute it.

(b) Note that in the original simulation, it takes about 10 time units for the concentration of A to be almost zero. We want the concentration of A to take half as long to decline to zero. The only variable under our control is the initial concentration of X. Approximately what should the concentration of X to do achieve that? How can you confirm this using the simulation?

9.5 Part 1

Retrieve Lab1Part1Starter.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 big 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>N</td>
<td>The number of time steps. You can calculate this from the first two parameters in the obvious way.</td>
</tr>
<tr>
<td>TZ</td>
<td>A table containing values of the zone temperature. TZ[0] is the initial temperature, TZ[1] the temperature after the first time step, etc.</td>
</tr>
<tr>
<td>TEW</td>
<td>A table containing values of the exterior wall temperature</td>
</tr>
<tr>
<td>Time</td>
<td>A table containing values of elapsed time</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.

After you have gotten your simulation to running for a few time steps, vary the necessary parameters to 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[ea] and T[ea] to establish a steady state temperature of 78 degrees.

Save this worksheet as myNameCS123Lab1Part1.mw. (You knew you were supposed to substitute your own name for myName didn't you?)
9.6 Part 2

a) Enter the following Maple function definitions into a fresh worksheet's code edit region:

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

acState := proc(temp, presentState)
    local triggerTemp, shutoffTemp;
    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;

# Air flow control function definition

airFlowControl0 := proc(state, lowFlow, highFlow)
    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 acState function is to make the state of airflow high when the temperature goes beyond a trigger value, and to turn it down to low when the temperature falls below another trigger value. 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.

After you have entered them, test these functions to make sure that they are operating correctly before you move onto the next part. A test should consist of invoking the function where you know in advance what the result should be. 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.

b) The next step is to change the simulation model so that it incorporates a thermostat controlling the airflow. Retrieve the worksheet `myNameCS121Lab1Part1.mw`. Add in the two function definitions from a) into the code edit region between the restart and the rest of the initialization assignments. Resize the code edit region so that you can see all of the code without too much scrolling. Save this worksheet as `myNameCS121Lab1Part2.mw`.

Make additional modifications to the simulation to incorporate the HVAC airflow.

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

ii) Introduce additional variables \( lf \) and \( hf \) that define the levels of low and high airflow. Initialize \( lf \) and \( hf \) before the time step loop so that they have sensible values, such as 2000 and 4000.
iii) Inside the loop, the first step is to use acState to compute the new present State.

iv) Edit the formula for Tz[i+1] so that it uses airFlowControl0(presentState, lf, hf) rather than the constant Q[ea] for the air flow.

Using the results for Part 1, 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.

c) If you verified that the simulation results were accurate to real-life, consider the things that the model would allow you to change by changing a parameter value. Which of those would you want to change to have a more satisfying conditioning system? In order to answer this question, you will have to relate the model to actual things and which of its properties would be satisfying to paying customers.

d) Save the final version of your work for Part 2 as myNameCS121LabPart2.mw.

9.7 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.

9.8 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-invoke 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.
9.9 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.