Overview

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

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

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

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

A brief overview of autonomous car control

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

The DARPA challenge spurred significant progress. Google continued this work, unveiling its self-navigating car in 2010. There are You Tube videos (e.g. http://www.youtube.com/watch?v=d0Ny-uztjS8&noredirect=1, http://www.youtube.com/watch?v=-nYhKD8leAg&feature=related) of the Google car driving in traffic.

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

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

**Autonomous vehicles**
Scenes from the DARPA autonomous vehicle race in 2007

**Autonomous Driving**

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

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

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

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

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

The Google self-driving Prius
Pre-lab preparation
1. Reading: chapters 10 and 11. Review older chapters 1-9 and labs from cs 121 as needed.
2. Take the pre-lab quizlet 1 at the CS 122 Maple TA web site. You should do quizlet 1 before lab to be prepared for the lab activities.

Part 0 -- Prelab prep

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

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

Do the following:

1. Open a Maple worksheet
2. Create a code edit region by

   Insert -> Code Edit Region

Checkpoint: What you should have should look like this:

```
.
.
.
```

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

Checkpoint: What you should have should look like this:
4. Enter some code (enter this code exactly)

#This is a proc to compute the hypotenuse of a triangle with sides s1 and s2.

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

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

Checkpoint: What you should have should look like this:
            Notice the elements of a proc (read over comments)
Hyp := proc(s1, s2)    # Proc header - proc name (Hyp) and parameters (s1, s2)
    local hypotenuse;   # This variable is used in the proc, but not passed in
    hypotenuse := sqrt(s1^2 + s2^2);  
    return hypotenuse;    # The value in this variable is returned at the conclusion of the proc
end proc;

# Now, execute this proc
Hyp(7, 24);          # Use the proc's name to execute it and pass in values for s1 and s2 respectively.

5. Execute the code. You can do this (assuming that the flashing cursor is located in the code edit region box, by typing command-E (control-E on Windows), or by clicking on the "!" icon in the Maple toolbar.

Checkpoint: What you should have should look like this:

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

# Now, execute this proc
Hyp(7, 24);
local hypotenuse;
    hypotenuse := sqrt(s1^2 + s2^2); return hypotenuse
end proc

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

    Hyp := proc(s1,s2)

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

Problem A deliverables and outcomes

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

a) How do you create a code edit region in a worksheet?
b) How do you adjust the width and height of the text box region? What are reasonable settings for a region so that you can get work done?
c) How do you "iconify" the region? What happens when you click on the icon? How do you "un-iconify" a region?
d) How do you execute all of the lines in a code edit region? What is the keyboard shortcut for doing so?
e) What is an execution trace and what kinds of information can you get from it?

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

1. Create a code edit region and re-size it as you proceed so that the entire segment of code is visible in the CER window.

2. Enter and execute the following code segment. Be sure to carefully review the code along with the associated comments:
# this code adds a list of numeric elements

AddList := [1, 2, 3, 4, 5];
ListSize := nops(AddList);  # nops is a Maple function to determine the number of elements in a list

ListSum := 0;  # keep an accumulated value of the sum as new elements are added in
count := 0;   # keep track of the number of elements already added to the sum

while (count < ListSize) do
    count := count + 1;  # increment the counter here so that the correct element is accessed next
    ListSum := ListSum + AddList[count];
end do;  # end of while loop - all elements accumulated

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

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

   a. What is the sum of the elements as produced by the program?
   b. What happens when we change the semi-colon after the "end do" statement to a colon?
   c. What is the effect of initializing the count to 1 instead of 0?
   d. If the count was initialized to 1, what else in the program would need to change in order to obtain the correct answer?

   Show the modified code and result to the grader.

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

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

CarSimulator.hdb
CarSimulator.mla

CS122Lab1StarterB1.mw
CS122Lab1StarterB2.mw

CS122Lab1Problem4-Tutorial.mw

CS122Lab1Problem1.mw
CS122Lab1Problem3.mw
CS122Lab1Problem4.mw
CS122Lab1Problem5.mw
When working on car simulator problems, you should begin work by double clicking on one of the lab .mw files to start up the simulator. This will have the effect of causing Maple to automatically look in the right place for the CarSimulator library files. If you have trouble doing this, ask for help.

Let's see a quick demo of the simulator.

(a) Open and execute the worksheet named CS122Lab1StarterB1.mw. Within the worksheet, you should see lines of code that:

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

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

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

Problem C deliverables and outcomes

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

a) What is the Maple name of the package for the Car Simulator?
b) What does the command with(CarSimulator): do?
c) What is the color of a "wall" square? What is the color of a target square?
d) What does direction of the arrow drawn at each step for the car indicate?
e) Describe what you would see if the car was originally located at (3,2) and was told to move 3 squares backwards. How could you tell whether the car was moving forwards or backwards?
f) What is the name of the operation in the car simulator package that
   - produces the animation of the car moving around?
   - runs the simulation of the car with a control program?
g) What is the purpose of the argument given to the run operation?
Lab Exercises

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

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

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

Problem 2 -- Understand more about the CarSimulator package

The CarSimulator package includes some help pages on the various features of the package. Activate Maple on-line Help. In the "Table of Contents" pane on the left hand side you should see a folder for the CarSimulator. (Note: CarSimulator help should be available after you have done Part 0, because some of the actions in Part 0 tell Maple to look in the current directory for help. If not, open again one of the Tutorial worksheets from Part 0 and then activate Maple help again.)
The documentation for this was written by course staff and included in the CarSimulator.hdb file that you downloaded with the other lab files.

Use this Help segment as you encounter car simulation features which need explanation or clarification.

\section*{Problem 3 -- Write a program to get the car to move in a Square}

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

This problem just has the car moving in a square whose side is 6 units. Note that movement begins Eastward from the starting position.
Problem 4 -- Write a program to get the car to move under a wall, detect a gap and proceed towards a target

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

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

(i) Describe, in your own words, what actions are being repeated by the while.
(ii) Describe in your own words, what actions are necessary as initialization before the repetition of the while gets started
(iii) Describe in your own words, the condition checked by the while loop. When will the loop repeat? When will the loop stop?

Write these descriptions on the whiteboard.

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

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

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

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

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

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

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

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

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

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

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