CS1112 Fall 2010 Project 3 Part 2 due Thursday 10/14 at 11pm

(Part 1 appears in a separate document. Both parts have the same submission deadline.)

You must work either on your own or with one partner. You may discuss background issues and general solution strategies with others, but the project you submit must be the work of just you (and your partner). If you work with a partner, you and your partner must first register as a group in CMS and then submit your work as a group.

Objectives

Completing this project will solidify your understanding of user-defined functions and vectors. You will also do more graphics. In Part 2 you will develop code for the iRobot Create (a robot) using a simulator. This is an opportunity to see and appreciate the approximation and errors associated with real robotic control programs!

MATLAB Files and Simulator Toolbox

Download the file `p3part2.zip` from the Projects page on the course website. **The files contained in `p3part2.zip` must be in the Current Directory in** MATLAB.

This project uses the *iRobot Create Simulation Toolbox*, which is a set of MATLAB files developed in a joint project between CS1112 and MAE4180 Autonomous Mobile Robot. The code that runs on the simulator can run directly on the Create robot!

The Robot and an Example Control Program

The robot is round with a diameter of roughly 34cm. It has two motored wheels that we can control; a third castor wheel is near the front of the robot for balance. The robot has many sensors, but in this project we will make use of three kinds of sensors only: "bump sensors" which detect collision, "cliff sensors" for reading markings on the floor, and the "virtual wall sensor" for reading infrared signal. In the final problem we will make use of an "overhead localization system" for determining the robot's global position.

An example control program, `driveForwardUntilWall`, demonstrates two functions available in the simulator for robot movement and sensing. Here is how to run any control program in the simulator, assuming that the simulator code is accessible:

1. **Set the Current Directory to the folder that holds the control program file and map file.**
2. Type `SimulatorGUI` in the Command Window. The Simulator starts with a blue circle representing the robot in the center. The blue line indicates the heading of the robot. By default the robot is at (0,0) oriented towards the east.
3. Click the "Load Map" button. Select the map file in the dialog box that opens up. For this example the map is `squareEnclosure.txt`. Four walls (lines) forming a square room appears.
4. *Optional:* Change the position of the robot by clicking the "Set Position" button. Then make one click in the plot area to position the center of the robot and a second click to indicate the robot's orientation. For this example keep the robot inside the room facing but not touching a wall.
5. *Optional:* Under "Sensors" select the "Bump" checkbox to indicate that you wish to visualize the bump sensor (magenta when it is activated).
6. Click the "Start" button under "Autonomous" to select a control program to run. Select `driveForwardUntilWall.m`. The robot moves forward until it hits a wall. You will hear four beeps.

Observe that the "Manual Control" keys are dimmed (inactive) when autonomous code is running and light up once the control program ends. You can stop autonomous code execution by clicking the "Stop" key under "Autonomous."

Now read `driveForwardUntilWall.m` and note the following:

- The function parameter `serPort` is for communication with a *real* robot; in the simulator it is not meaningful. Nevertheless we must keep that parameter and use it when calling functions that run on the robot. *The code you develop in the simulator is the same code for controlling the real robot!*
- The `BumpsWheelDropsSensorsRoomba` function reads specific sensors on the robot and detects, among other things, bumps on the front half circle of the robot. We make use of only three of the six returned values: `BumpRight`, `BumpLeft`, and `BumpFront`. The robot actually has only two physical bumpers: if the right bumper is activated `BumpRight` is 1, if the left bumper is activated `BumpLeft` is 1, *if both left and right bumpers are activated it is treated as a front hit so* `BumpFront` *is 1 and* `BumpRight` *and* `BumpLeft` *are both 0.* The only argument needed in this function is `serPort`.
- The `SetDriveWheelsCreate` function allows us the set the velocity of the right and left wheels. Use three arguments: `serPort`, right wheel velocity, and left wheel velocity. The unit is meters/second and the range is -0.5 to 0.5. Negative velocity is backward.
- The `pause` halts the execution of code, not the robot. After setting the wheel velocities we pause code execution to let the robot move. If you remove the pause the code executes—calls functions on the robot—too fast and the program will be stuck.
- `StopCreate` is a *sub*function in `driveForwardUntilWall`. Look at the given code: it simply sets the wheel velocity to zero.
- `Signal` is a *sub*function in `driveForwardUntilWall`. We include it so that you hear an audible signal when the control program finishes execution.

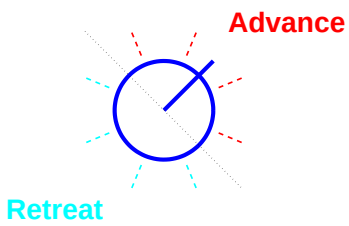
Now modify `driveForwardUntilWall.m`: Add `pause(2)` after the loop ends and before calling `StopCreate`. When you re-run the program (set the position of the robot in the simulator, click "Autonomous Start," and so on), you will notice that after the robot hits a wall, the robot *slips along the wall* before it stops and you hear the four beeps. This is because after hitting the wall, the wheels keep spinning for two seconds, pushing the robot against the wall. The simulator models wheel slips, and likely you will see them as well as other approximations and errors when you solve the next two problems.

Now you're ready to write your own control program!

2 The Randomly Wandering Robot

Complete the function `randomWalkRobot` to have our robot wander randomly inside a room until it hits a wall. Assume the room is completely walled and the robot starts inside the room not touching any wall. In each step of this random walk, the robot rotates a random angle value from its current heading and then drives forward for one second at .5 m/s.

The random angle is generated at each step as specified here. A rotation of $-\pi/2$ to $\pi/2$ is called an "advance angle" since the robot would then move forward relative to its previous position. A rotation of $\pi/2$ to $3\pi/2$ is called a "retreat angle." Generate a random angle such that it is w times as likely to get an advance angle than a retreat angle. *Note:* First choose whether it is an advance or a retreat angle. Then generate a uniformly random number within the advance or retreat range. Implement a function (not subfunction) to get this random angle:




```
function d = getRandAngle(w)
% d is a random angle in degrees. d is w times as likely to be an advance
%   angle than a retreat angle.  w is a positive integer.
% d is an advance angle if -90<d<90; d is a retreat angle if 90<d<270.
```

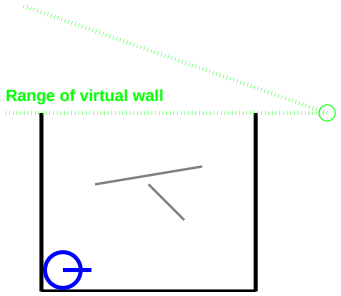
Take a look at the incomplete function `randomWalkRobot`. The weight `w` for advance versus retreat is set to be three, but make sure that your code works for different positive integer values of `w`. In `randomWalkRobot` you must call your function `getRandAngle`.

In addition to functions `BumpsWheelDropsSensorsRoomba` and `SetDriveWheelsCreate` demonstrated in the example above, you will need the function `turnAngle`. For example, the function call `turnAngle(serPort,.2,80)` turns the robot 80 degrees at .2 rad/s. The allowed angle range is -360 to 360 degrees and positive is counter clockwise. The allowed speed range is 0 to .2 rad/s. Each call to `turnAngle` produces a line of output under normal execution; don't worry about it.

Use the map file `squareEnclosure.txt` with this problem. (However the solution does not depend on the shape of the enclosure.) Submit your functions `randomWalkRobot` and `getRandAngle` on CMS.

3 The Reconnaissance Robot

Our intrepid robot now goes on a mission to discover the markings on the floor of a secret lab. The rectangular lab has three solid walls on the west, south, and east sides. The north side has a virtual wall, i.e., an infrared beam marks the northern extent of the lab. It is known that the lab floor is white and marked with dark lines, but near the walls and virtual wall, about a robot's width, there is no floor marking. In addition to the "bump sensors" the robot will employ its "cliff sensors," which measure the reflectivity of the floor, and a "virtual wall sensor." In order to map out the floor markings, the robot will also use a GPS-like system to obtain its location. Complete the control program `exploreRoom` to enable the robot to perform this recon mission.



3.1 Traversing the Floor

As usual, start by decomposing the problem! The first task is to travel over the entire floor area. Things to know/consider:

- The robot always begins in the southwest corner facing east. Use the map file `labSmall.txt` for program development and later test the program also on `lab.txt`.
- The only robot sensors to use for traveling are the bump sensors and virtual wall sensors. The statement `vws = VirtualWallSensorCreate(serPort)` will assign to variable `vws` the value 1 if a virtual wall is sensed and 0 otherwise. It's OK for the robot to be in the infrared range—it won't get fried and hopefully it is close to completing its mission by then.
- The only robot motion functions to use are those used in the previous problem: `SetDriveWheelsCreate` and `turnAngle`. What is your plan for moving over the entire floor? Can you decompose that plan into specific parts that are repeated in a systematic way?
- You probably want the robot to move at the maximum allowed speed, but to minimize hard bumps and wheel slips you might want to call the sensors frequently. Remember to insert a pause to "slow down" repeated function calls to the robot; otherwise your program will hang.

Test your code to make sure that the robot covers the floor. You are now working with a robot (although in a simulator) so keep in mind that there are errors and approximations everywhere. The robot wheels will slip sometimes; the right-angle turns are not always so "right-angled." Some redundancy (overlap) is good, so don't spend a lot of time worrying about things like exactly how far (what fraction of a second) a move needs to be. We can't get perfection, but we can get a program that works.

3.2 Mapping the Markings

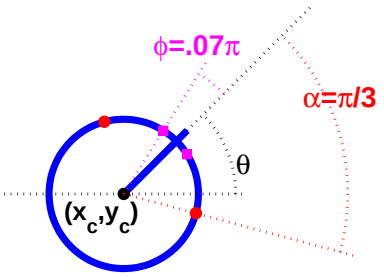
The robot will produce a map of the markings at the end of its travel. As it travels, it needs to check the reflectivity of the floor under its cliff sensors. Dark colors have low reflectivity and it is known that most white floors have a reflectivity of above 20 while dark paint has a reflectivity around 3. Here's the general

plan: when the robot detects a point that it decides is dark, it calls its localization system to get its current coordinates and records them. At the end of its travels it makes a plot of the recorded coordinates.

But it's not quite that simple... The localization system gives the coordinates of the center of the robot,(x_c, y_c), and the robot's orientation, θ. The four cliff sensors are symmetric about the robots orientation line: the front-left and front-right sensors are $.07\pi$ from the orientation line; the left and right sensors are $\pi/3$ from the orientation line. So if, for example, the front-left cliff sensor detects a dark spot, the coordinates *of the dark spot* are

$$(x_c + r \cdot \cos(\theta + \phi) \ , \ y_c + r \cdot \sin(\theta + \phi))$$

where r is the radius of the robot, 0.17m.



The statement `rf = CliffFrontLeftSignalStrengthRoomba(serPort)` assigns to `rf` the reflectivity of the floor under the front-left cliff sensor; it is a value in the range [0,100]. The three functions for the other three cliff sensors have these names and can be called in the same way:

```
CliffFrontRightSignalStrengthRoomba
CliffLeftSignalStrengthRoomba
CliffRightSignalStrengthRoomba
```

To get the current location and heading of the robot, call the function `OverheadLocalizationCreate`:

```
[xc,yc,theta]= OverheadLocalizationCreate(serPort)
```

The returned values `xc` and `yc` are the x- and y-coordinates of the center of the robot; `theta` is the heading of the robot in radians from the positive x-axis.

You will implement the *sub*function `ReadFloor` to call the four cliff sensor functions and return the coordinates of any dark spots found:

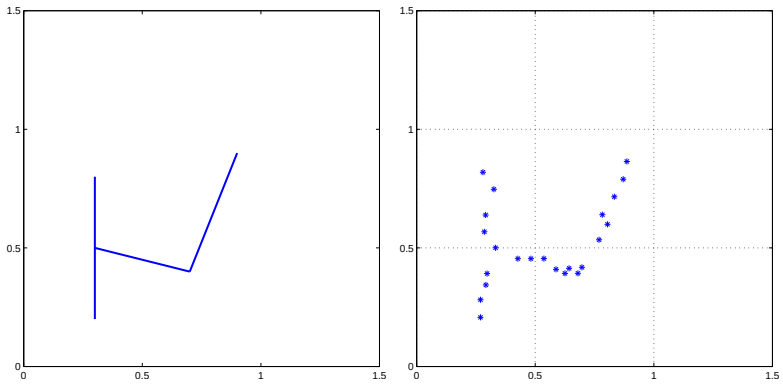
```
function [x, y]= ReadFloor(serPort)
% Return the coordinates of any dark spots detected by cliff sensors.
% x and y are vectors that store the coordinates of dark spots.  If there
% are no dark spots, x=[] and y=[].  The length of x (and y) is the number
% of dark spots detected.
% serPort is the serial port number (for controlling the actual robot).
```

`ReadFloor` is called by the main function `exploreRoom` every time that the robot needs to take a reading of the floor on which it stands. If there is no dark spot, the returned vectors are empty, i.e., they are vectors of length 0. If there is one dark spot, then the returned vectors are of length 1. Since there are only four sensors the maximum length of the returned vectors is 4.

After the robot stops moving, the program should issue an audible signal—call subfunction `Signal`. Then the program draws a plot of the coordinates of the dark spot:

```
figure(1)              % start a figure window and number it window 1
plot(xdark,ydark,'*')  % xdark, ydark are vectors storing the x, y coords of the dark spots
axis equal
axis([0 1.5 0 1.5])
grid on                % show grid lines on the plot
```

Below are figures of the actual marking in the map `labSmall.txt` and an example of a robot-generated map of the marking. As one would expect, the result is not a perfect match! (But it's reasonably close.)



Submit your file `exploreRoom.m` on CMS.