CIS 3415, Project-2 (**8-pt**)

1 Description

This project gets you using the bumper proxy and (if you want) handling odometry information from the Position2DProxy.

2 Starting with the simulator

1. This time, we will start with the Stage simulator.
2. Login to your computer. The user name is `student`, the password is `student`.
3. The first thing to do is to open a terminal window.
   - Just click on the  icon on the menu bar at the top of the screen.
   - You can also get hold of the terminal via:
     Applications > Accessories > Terminal
   - In fact, open two of these windows.
4. Start by getting to the right place in the file system. To do that type:
   `cd ~/Desktop/project2`
   in both terminal windows.
5. Now invoke the simulator. Do this by typing:
   `player world2.cfg`
   This should pop up a square window labelled **Player/Stage: ./world2.world** which contains a grey dot and a strangely shaped lump. This is the simulated world in which your robot will operate.
6. Remember that you compile the controller in `bumper.cc` using:
   `./build bumper`
   and run it using:
   `./bumper`
7. Do both these things.
8. Watch how the robot moves, and look at the values you get from the *odometry* — the robot's attempt to keep track of how far it has gone — these will show up in whatever window you run the controller in.

3 Using bumpers and odometry

1. You have a slightly different starting point from last time. Rather than `roomba-roam.cc` you have `bumper.cc`.
2. You can see the difference if you look at the code — this time the code is reading values from the Position2dProxy. This is the odometry.
3. Now, what I want you to do is to edit the original version of
   `bumper.cc`
   so that the robot makes a circuit of funny-shaped object in the middle of the world.

4. Start from the position in the unedited config file (which has the robot start moving horizontally from the bottom left corner).
5. Your solution should use a combination of odometry and the data from the bumpers to help the robot find its way around the obstacle.
6. See later in these notes to get a list of commands for the Position2dProxy and the BumperProxy.
7. When you are done, save your program as *(***your-names***)***-proj2-part1.cc** and make sure you put your name in the comments.
8. You'll need to submit this to me after you are done with the project.

4 Now the maze

1. Now edit the simulator setup.
2. Edit the file world2.world so that it uses the file maze.png rather than wall-world.png
3. The task is to have the robot move from its initial location, at the bottom left corner of the world to the top right corner.
4. This time you can use just odometry, just bumper data, or a combination of the two.
5. When you are done, save your program as *(***your-names***)***-proj2-part2.cc** and make sure you put your name in the comments.
6. You'll need to submit this to me after you are done with the project.

5 An almost-real maze

1. Once your program works in the simulator, you can try it out on the real robot — I will mark out the course on the floor (it won't be the same size as the simulated maze, but the topology will be the same), and we'll simulate the walls.
2. There's no need to save the program you end up with for this part.

6 Handling Proxies

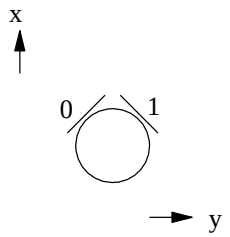
Here is a listing of the relevant commands for the proxies.

6.1 Bumper Proxy

As you can see from the code, it is possible to handle the bumper proxy as if it is an array. If the proxy is bp, then

bp[0] and bp[1]

return 0 if the bumper is not pressed, and 1 if it is. In this case we happen to know that the robot has two bumpers laid out like this:



More correctly, the proxy is handled using the following functions:

1. bp.GetCount(), which returns the number of bumpers;
2. bp.IsBumped(i), which reports 1 if bumper i is bumped and 0 otherwise; and
3. bp.IsAnyBumped(), which reports 1 if any bumper is bumped and 0 otherwise.

6.2 Position2d Proxy

The Position2dProxy has the following functions which are illustrated in bumper.cc:

1. pp.GetXPos(), which reports how far the robot estimates it has moved in the x direction;
2. pp.GetYPos(), which reports how far the robot estimates it has moved in the y direction; and
3. pp.GetYaw(), which reports how far the robot estimates it has rotated.
4. pp.SetSpeed(double aXSpeed, double aYSpeed, double aYawSpeed), which sets the speed of the robot.

It is also possible to ask the proxy how fast the robot is currently moving:

1. pp.GetXSpeed(), which reports how far the robot estimates it has moved in the x direction;
2. pp.GetYSpeed(), which reports how far the robot estimates it has moved in the y direction; and
3. pp.GetYawSpeed(), which reports how far the robot estimates it has moved in the y direction. We can

also change the odometry values:

1. pp.ResetOdometry(), which sets x, y and yaw to zero, so that you can count up to some desired value; and
2. pp.SetOdometry((double aX, double aY, double aYaw)), which sets the odometry readings to the specified values (it does not move the robot).