Design Lab 2 *6.01 – Fall 2011*

Controlling Robots

Goals:

- Experiment with state machines controlling real machines
- Investigate real-world distance sensors on 6.01 robots: sonars
- Build and demonstrate a state machine to make the robot do a task: following a boundary

1 Materials

This lab should be done with a partner. Each partnership should have:

- A lab laptop.
- A robot, a (long, gray) serial cable, and a (short, blue) serial-to-USB adapter.
  - The serial cable is a long beige or gray cable. Most of the robots already have one attached.
  - *Warning: if your robot starts to go too fast or get away from you,* ***pick it up!!***
- A white foam-core board with bubble-wrap on one side.

Be sure to mail all of your code and data to your partner. You will both need to bring it with you to your first interview.

2 Simple Brains

A 'brain' is a Python program that specifies behavior for the robot. The process of constructing and running a brain is described in detail in the *Robot Infrastructure Guide*.

Objective: Build a state machine "brain" for controlling a robot, first in the *soar* simulator, then on a real robot.

- Run a simple brain in *soar*, and record the robot's path
- Modify the simple brain to make the robot rotate in place
- Run the brain on the **Pioneer robot platform**

Some of the software and design labs contain the command athrun 6.01 getFiles. Please disregard this instruction; the same files are available on the 6.01 OCW Scholar site as a .zip file, labeled Code for [Design or Software Lab number].

Resources:

- `~/Desktop/6.01/designLab02/smBrain.py`: a simple robot brain, which uses the 6.01 **state machine class sm**
- `tutorial.py`: a virtual world in soar

Detailed guidance : Do the following with a 6.01 lab laptop:

1. **Run a brain in the simulator.**
   - **a.** In the Terminal window, type `soar &`.
   - **b.** Click soar's **Simulator** button and double-click `tutorial.py`. This loads a specific virtual world into our robot simulator.
   - **c.** Click soar's **Brain** button, navigate to `Desktop/6.01/designLab02/smBrain.py`, and click Open. This loads a specific state machine definition into the robot simulator. That state machine describes the actions that the robot will take in response to sensed information about the virtual world surrounding it.
   - **d.** Click soar's **Start** button, and let the robot run for a little while.
   - **e.** Click soar's **Stop** button.
   - **f.** Notice the graph that was produced; it shows a 'slime trail' of the path that the robot followed while the brain was running. You can just close the window. (If you don't want the brain to produce a slime trail, you can set the `drawSlimeTrail` argument to the `RobotGraphics` constructor in the `smBrain.py` file to be `False`).
2. **Modify the brain and run it.**
   - **a.** In the Terminal window, type `idle &` to open up an Idle environment.
   - **b.** Click Idle's **File** menu, select **Open...**, navigate to `Desktop/6.01/designLab02/smBrain.py`, and click **Open**.
   - **c.** The state machine that controls the robot's actions is defined by the `MySMClass` definition. Think of this state machine as taking sensory data as input, and returning as output instructions to the robot on how to behave. The `io.Action` object returned as the output by the `getNextValues` method of the `MySmClass` tells the robot how to change its behavior, and has two attributes that are important to us:
     - ⋆ `fvel`: specifies the forward velocity of the robot (in meters per second)
     - ⋆ `rvel`: specifies the rotational velocity of the robot (in radians per second), where positive rotation is counterclockwise
   - **d.** Find the place where the velocities are set in the brain, and then modify it so that it makes the simulated robot rotate in place.
   - **e.** Save the file.
   - **f.** Go back to the soar window and click the **Reload Brain** button
   - **g.** Run the brain by clicking the **Start** and then the **Stop** buttons.
3. **Run it on the robot**
   - **a.** Connect the robot to your laptop, making sure the cable is tied around the handle in the back of the robot.
   - **b.** Power on the robot, with a switch on the side panel.

c. Click soar's Pioneer button, to select the robot. You should be able to hear the sonar sensors making a ticking noise.

d. One partner should be in charge of keeping the robot safe. Keep the cable from getting tangled in the robot's wheels. **If the robot starts to get away from you, pick it up, then, turn it off using the switch on the robot.**

e. Click soar's Start button.

3 Sonars

Objective: Investigate the behavior of the **sonar sensors**, and modify a robot brain to make a robot keep a certain distance from an obstacle. **Don't spend more than 10 or 15 minutes experimenting with the sonars. When you're done, ask a staff member for a checkoff.**

The `inp` argument to the `getNextValues` method of `MySMClass` is an instance of the `soar.io.SensorInput` class, which we have imported as `io.SensorInput`. It has two attributes, `odometry` and `sonars`. For this lab, we will just use the `sonars` attribute, which contains a list of 8 numbers representing readings from the robot's 8 sonar sensors, which give a distance reading in meters. The first reading in the list (index 0) is from the leftmost (from the robot's perspective) sensor; the reading from the rightmost sensor is the last one (index 7).

Detailed guidance :

- Modify the brain so that it sets both velocities to 0, and uncomment the line

  ```
  print inp.sonars[3]
  ```

  Reload the brain and run it. It will print the value of `inp.sonars[3]`, which is the reading from one of the forward-facing sonar sensors.
- From how far away can you get reliable distance readings? What happens when the closest thing is farther away than that?
- What happens with things very close to the sensor?
- Does changing the angle between the sonar transducer and the surface that it is pointed toward affect the readings? Does this behavior depend on the material of the surface? Try bubble wrap versus smooth foam core.
- Now, set the `sonarMonitor` argument to the `RobotGraphics` constructor to be `True`.

  Reload the brain and run it. This will bring up a window that shows all the sonar readings graphically. The length of the beam corresponds to the reading; red beams correspond to "no valid measurement". Test that all your sonars are working by blocking each one in turn. If you notice a problem with any of the sensors, talk to the staff.

Checkoff 1. **Wk.2.2.1**: Explain to a staff member the results of your experiments with the sonars. Demonstrate that you know your partner's name and email address.

Make the robot move forward to approximately 0.5 meters of an obstacle in front of it and keep it at that distance, even if the obstacle moves back and forth. Do this by editing the `getNextValues` method of `MySMClass`; there is no need to change any other part of the brain. Don't set the forward velocity higher than 0.3 (or lower than -0.3). Debug it in simulation, by clicking soar's **Simulator** button and choosing `tutorial.py`. Once it seems good, run it on a real robot, by choosing soar's **Pioneer** button.

Checkoff 2. **Wk.2.2.2**: Demonstrate your distance-keeping brain on a real robot to a staff member.

4 Following Boundaries

Objective: Our goal now is to build a state machine that controls the robot to do a more complicated task:

1. When there is nothing nearby, it should move straight forward.
2. As soon as it reaches an obstacle in front, it should follow the boundary of the obstacle, keeping the right side of the robot between 0.3 and 0.5 meters from the obstacle.

Draw a **state-transition diagram** that describes each distinct situation (state) during wall-following and what the desired output (action) and next state should be in response to the possible inputs (sonar readings) in that state. Start by considering the case of the robot moving straight ahead through empty space and then think about the input conditions that you encounter and the new states that result. Think carefully about what to do at both inside and outside corners. Remember that the robots rotate about their center points. Try to keep the number of states to a minimum.

Checkoff 3. **Wk.2.2.3**: Show your state-transition diagram to a staff member. Make clear what the conditions on state transitions are, and what actions are associated with each state.

Copy your current `smBrain.py` file to `boundaryBrain.py` (you can do this with **Save As** in idle), and modify it to implement the state machine defined by your diagram. Make sure that you define a `startState` attribute and a `getNextValues` method.

Try hard to keep your solution simple and general. Use good software practice: do not repeat code, use helper procedures with mnemonic names, try to use few arbitrary constants and give the ones you do use descriptive names.

To debug, add print statements that show the relevant inputs, the current state, the next state, and the output action.

Record a slime trail of the simulated robot following a sequence of walls; make sure that it can handle outside and inside corners. Going around very sharp corners or hairpin turns, such as the L in `tutorial.py`, is not required, but is extra cool.

Checkoff 4. **Wk.2.2.4**: Demonstrate your boundary follower to a staff member. Explain why it behaves the way it does. **Mail your code to both partners.**

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011