Design Lab 3 *6.01 – Fall 2011*

All Carrot, No Stick

Goals: The overall goal of this lab is to build a robust capability for the robot to drive a route made up of linear path segments. We will do this in three stages:

- Implement a state machine that drives the robot in a straight line to a goal specified in the coordinate system of its odometry.
- Make the robot traverse a sequence of linear segments by cascading a machine that generates target points with the machine that drives to a target.
- Make the robot less of a danger to humanity by adding a 'reflex' that will make the robot stop when its path is blocked, and wait until it is unblocked, then resume its trajectory.

1 Materials

This lab should be done with a partner. Each partnership should have a lab laptop or a personal laptop that reliably runs **soar**. Do `athrun 6.01 getFiles` to get the files for this lab, which will be in `Desktop/6.01/designLab03/`; they will provide you with these resources:

Resources:

- `moveBrainSkeleton.py`: main brain file, which imports the next two files.
- `ffSkeleton.py`: template for your implementation of a figure-following state machine.
- `dynamicMoveToPointSkeleton.py`: template for your implementation of a behavior that drives the robot to a specified point.
- `testFF.py`: procedure for testing figure follower in **idle**. Not used when the brain is run.
- `testMove.py`: procedure for testing move to point in **idle**. Not used when the brain is run.

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

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].

For this week, you should read all of **chapter 4 of the Course Notes**. But before starting this lab, please read: **Section 4.2.1** (Cascade combination), **Section 4.2.2** (Parallel combination), and **Section 4.2.6** (Switch and multiplex).

2 Driven

Objective: Make a robot brain that can follow a path that is composed of a sequence of straight line segments.

Let us explore how state machines can be composed, by creating a robot brain made of a state machine that is itself composed of simpler state machines, as shown in **Figure 1**.

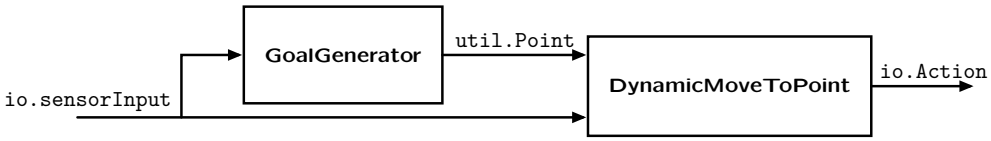


Figure 1 Architecture for robot brain (which is a state machine) constructed from two simpler state machines.

The two simpler state machines we shall construct are:

- A **GoalGenerator** is a state machine. On each step, the input is an instance of `io.SensorInput` (which contains the robot's sonar and odometry readings) and the output is an instance of `util.Point`, which represents the target that the robot should drive toward.
- A **DynamicMoveToPoint** state machine takes an input that is a tuple containing two items: the first is an instance of `util.Point` and the second is an instance of `io.SensorInput`. On each step, the state machine generates one instance of `io.Action`, which specifies a single step toward the specified `util.Point`.

Step 1. **Detailed guidance** : (Do not define any new classes for this part!)

Use `sm.Cascade`, `sm.Parallel`, `sm.Constant` and `sm.Wire` (which you can read about in the **online software documentation**, available from the *Reference* tab of the 6.01 home page, as well as in the readings) to construct a composite machine shown in **Figure 1**.

For now, make a state machine (which serves the role of a **GoalGenerator** in **Figure 1**) that always outputs the constant point `util.Point(1.0, 0.5)`. Also, create a **DynamicMoveToPoint** state machine (**Figure 1**) by making an instance of the `dynamicMoveToPointSkeleton.DynamicMoveToPoint` class.

Type your code for constructing this composite machine into `moveBrainSkeleton.py`, in place of `None` at the line

```
mySM = None
```

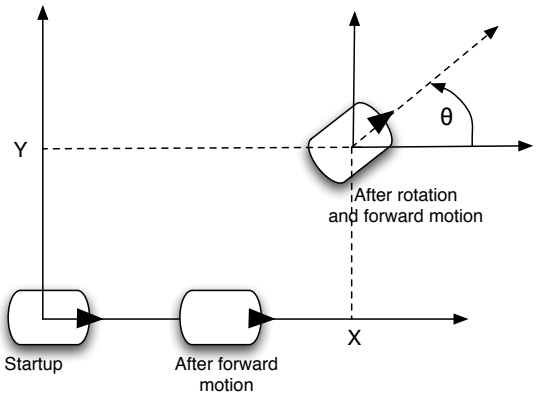
You can test your composite machine by running `moveBrainSkeleton.py` in **soar**. Select the `bigEmptyWorld.py` world. Each **soar** step should print a message (but there will be no motion of

the robot). In later steps, you will replace the code that generates these messages with code that moves the robot.

2.1 Odometry

Before we implement the module that controls the robot, we have to understand coordinate frames and how the robot reports its current location.

The robot has *shaft encoders* on each of its drive wheels that count (fractions of) rotations of the wheels. The robot processor uses these encoder counts to update an estimated *pose* (a term that means both position and orientation) of the robot in a global reference frame. This figure shows the reference frame in which the **robot's odometry** is reported.



When the robot is turned on, the frame is initialized to have its origin at the robot's center and the positive x axis pointing out the front of the robot. You can now think of that frame as being painted on the ground; as you move or rotate the robot, it keeps reporting its pose in that original frame. The pose has x, y, and θ components: x and y are its location, in meters, and θ is its rotation to the left, in radians.

Note that the pose reading doesn't change if you pick up the real robot and move it without the wheels turning. It only knows that it has moved because of the turning of the wheels. And remember that the odometry is far from perfect: the wheels slip and the error (especially rotational error) compounds over time.

2.2 Utilities

We will represent robot poses using the `util.Pose` class, which you can read about in the **online software documentation**. The **`util` module** also defines a class `util.Point` for representing points in 2D space. Assume that `p0 = util.Point(x0,y0)` and `p1 = util.Point(x1,y1)`, then

- `p0.angleTo(p1)` returns the angle (in radians) between the x-axis and the vector from `p0` to `p1`,
- `p0.distance(p1)` returns the Euclidean distance between `p0` and `p1`, and
- `p0.isNear(p1, distEps)` returns a boolean equal to `p0.distance(p1)<distEps`.

Assume that `pose0 = util.Pose(x0,y0,theta0)`, then

- `pose0.point()` returns a point equal to `util.Point(x0,y0)`.

These procedures may also be useful:

- `util.nearAngle(angle1, angle2, angleEps)`,
- `util.fixAnglePlusMinusPi(angle)`, and
- `util.clip(value, minValue, maxValue)`.

For reference, you should also consult the *Lab Infrastructure Guide*. It's a good place to go to remind yourself about how robots and brains work.

Warning! Be careful with angle arithmetic. In particular, notice that $\pi - \epsilon$ (say 179°) is very close to $-\pi + \epsilon$ (say $-179°$), but subtracting the numbers yields $2\pi - 2\epsilon$ (say 358°)! Use `fixAnglePlusMinusPi` to convert angles near 2π (360°) to equivalent angles near 0. Use `nearAngle` to compare angles.

All of our angle-manipulation procedures represent angles in **radians.**

2.3 Driving to a point

Objective: Implement the **DynamicMoveToPoint** state machine, which directs the robot to perform an action to reach a goal Point.

Detailed guidance :

Remember that the input to the **DynamicMoveToPoint** state machine is a tuple `(goalPoint, sensors)`, where

- `goalPoint` is an instance of `util.Point`. This point specifies where the robot should drive (in the coordinate frame of the odometry).
- `sensors` is an instance of `io.SensorInput`. You can get the current odometry value with `sensors.odometry`, which is an instance of `util.Pose`.

The output of the **DynamicMoveToPoint** state machine should be an instance of `io.Action` that specifies the action to be taken by the robot during its next step. For example, to create an action with forward velocity 0.2 m/s and rotational velocity 0.1 radians/s, the output should be `io.Action(fvel=0.2, rvel=0.1)`.

The robot should follow a (nearly) straight line from its current position to its goal point.

Step 2. Determine the action that should be output by the **DynamicMoveToPoint** state machine for the following input conditions. (Answers can simple commands such as 'move forward' or 'rotate left').

Current robot pose	goalPoint	Action
$(0.0, 0.0, 0.0)$	$(1.0, 0.5)$	
$(0.0, 0.0, \pi/2)$	$(1.0, 0.5)$	
$(0.0, 0.0, \tan^{-1} 0.5)$	$(1.0, 0.5)$	
$(1.0001, 0.4999, 0.0)$	$(1.0, 0.5)$	

Think of a strategy for implementing the **DynamicMoveToPoint** state machine. Explain how that strategy will drive the robot to the goal point. **The output of your state machine should depend only on the input; it doesn't need to use the state.**

Checkoff 1. **Wk.3.2.1**: Explain your strategy for implementing this behavior and your answers to the questions above to a staff member.

Step 3. Write code to implement your strategy in the file `dynamicMoveToPointSkeleton.py`.

- First, test your code in **idle**, as follows. Open the file `dynamicMoveToPointSkeleton.py`, comment out
  ```
  from soar.io import io
  ```
  and uncomment
  ```
  import lib601.io as io
  ```
- Now, test the code in **idle**, by running the `testMove` procedure from `testMove.py`.

Check Yourself 1. Be sure you understand what the answers to the test cases in this file ought to be, and that your code is generating them correctly.

- After your code works in **idle**, go back to **soar**'s version of `io.py` by editing `dynamicMoveToPointSkeleton.py` and commenting out
  ```
  import lib601.io as io
  ```
  and uncommenting
  ```
  from soar.io import io
  ```
- Now, test it using **soar**, by running `moveBrainSkeleton.py` in the simulated world `bigEmptyWorld.py`. **If you get an error message saying 'Not connected to soar.', you need to change the `io` import statement as described above.** Note that if you drag the robot around with your mouse, the robot won't know that it has been moved.

For help in debugging, find the line `verbose = False` in the brain and change it to `verbose = True`; that will cause **soar** to print out a lot of information on each step of the state machine that controls the brain.

Check Yourself 2. The robot always starts with odometry reading $(0, 0, 0)$, so it ought to move to somewhere close to the point $(1.0, 0.5)$ and stop.

3 Hip to be Square

Objective: Make the robot move in a square. Accomplish this (and other cool robot moves) by defining a new state machine class `FollowFigure`, which serves the role of being a **GoalGenerator**.

When we initialize the `FollowFigure` state machine, we will give it a list of *way points*, which are points that define a sequence of linear segments that the robot should traverse. The job of `FollowFigure` is to take instances of `io.SensorInput` as input and generate instances of `util.Point` as output; it should start out by generating the first point in the input sequence as output, and do that until the robot's actual pose (found as `sensorInput.odometry`) is near that point; once the robot has gotten near the target point, the machine should switch to generating the next target point as output, etc. Even after the robot gets near the final point, the machine should just continue to generate that point as output.

So, for example, if we were to use the `FollowFigure` instance below as our target generator, it should cause the robot to move in a square.

```
squarePoints = [util.Point(0.5, 0.5), util.Point(0.0, 1.0),
                util.Point(-0.5, 0.5), util.Point(0.0, 0.0)]
FollowFigure(squarePoints)
```

If the robot is trying to follow the square above, and the `FollowFigure` state machine starts in its start state and sees odometry poses as input in the following sequence, what should its next state and output be on each step?

Current robot pose	State	Target point
$(0.0, 0.0, 0.0)$		
$(0.0, 1.0, 0.0)$		
$(0.499, 0.501, 2.0)$		
$(2.0, 3.0, 4.0)$		

Detailed guidance :

Step 4. Define your `FollowFigure` state machine class in `ffSkeleton.py`. Test it in **idle** by running the `testFF` procedure in `testFF.py`, which contains the test cases shown above.

Step 5. Substitute an instance of the `FollowFigure` class for the **GoalGenerator** into the overall control architecture in `moveBrainSkeleton.py`, and debug it in `bigEmptyWorld.py`. Instead of going in a square, you could have your robot do a cool dance!

> *Checkoff 2.* **Wk.3.2.2**: Show the slime trail resulting from the simulated robot moving in a square or other interesting figure to a staff member. Explain why it has the shape it does. Take a screenshot of the slime trail and save it for your interview. Mail the code and slime trail you have so far, to your partner.

4 Avoiding Pedestrians

> **Objective:** Your simulated robot, as it is currently constructed, tries to follow the specified figure and completely ignores its sonar sensors. We would like to define a new behavior that tries to follow the figure, but that monitors the front sonars and if any of them gets a reading less than 0.3, stops until the obstacle disappears, and then continues to follow the figure.

Detailed guidance :

Step 6. Use the `sm.Switch` state-machine combinator to make a robot that stops for pedestrians. Read about `Switch` in **Section 4.2.6** of the readings. Remember that the action `io.Action()` will cause the robot to stop. This should take *a very small amount* of additional code. *Do not change your definition of* `DynamicMoveToPoint`.

You can test this brain in simulation by dragging the robot around; its odometry reading won't reflect that it has been dragged (it is as if you picked the robot up and 'kidnapped it') so if you move it in front of a wall it should stop; and if you drag it away it should start to move again.

> *Checkoff 3.* **Wk.3.2.3**: Demonstrate your safe figure-follower to a staff member. Mail your code to your partner.

Optional: Do it on a real robot. Try not to run over a staff member.

5 I am a ballerina

Replace `squarePoints` in your figure follower with `secretDance`, to have your robot gracefully dance the pattern of a secret message. It may take a while...

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011