MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.081—Introduction to EECS I
Spring Semester, 2007

Part of the work for Week 4

Issued: Tuesday, Feb. 27

This handout contains:

- Robot Lab for Tuesday, February 27
- **Some** of the post-lab exercises, due Tuesday March 6
- There are no exercises due on March 1

Sequential robot behaviors

This week, we'll use the tools of object-oriented programming to develop a new system for defining and combining robot behaviors, as well as to explore multiple different ways of computing solutions to a very important class of problems.

Tuesday Robot Lab: Combinations of Terminating Behaviors

In lab 2, we explored a system for making primitive behaviors and combining them. In that case, our behaviors were all operating at once, and their outputs were expressed as utility functions. It's possible to do a lot of different things this way, but many people were frustrated by the inability to ask their robot to do things sequentially, like drive up to the wall and then turn left.

In this lab, we'll develop a new system of behavior definition and combination, that allows sequential combination.

We'll represent each behavior in our new system as a type of object called a `terminating behavior`. A terminating behavior must have three methods: `start`, `step`, and `done`. These methods have no arguments except `self`.

The `start` method will be called once when the behavior is supposed to begin running; it's an opportunity to remember some initial condition, like the current odometry values. The `step` method will be called repeatedly until the behavior is done. Although it has no arguments, it can call `pose()` or `sonarDistances()` to get sensory information, and can call `motorOutput` to cause the robot to move.

We haven't used the `pose` function much before. It returns a list of three numbers, `[x, y, theta]`, which can be interpreted as the robot's x, y position, and orientation in a global coordinate frame. The `theta` value ranges between 0 and 2π. What is the global coordinate frame, you might ask? It is defined to have the origin and zero orientation be wherever the robot is when the brain is reloaded.

Thus, in order to define a new primitive terminating behavior object, say, `GoForwardALittleBit` you could write code like this:

```
class GoForwardALittleBit:
    def __init__ (...): ...

    def start(self): ...

    def step(self): ...

    def done(self): ...
```

Here is the code for a brain creates a new instance of this behavior class, and then uses that behavior to select actions until its `done` method returns true, at which point the robot stops moving (though the brain continues to run).

```
def setup():
    robot.behavior = GoForwardALittleBit()
    robot.behavior.start()

def step():
    if robot.behavior.done():
        motorOutput(0.0,0.0)
    else:
        robot.behavior.step()
```

If you define a new terminating behavior class, like `DoTheMacarena`, then you'd have `DoTheMacarena()` in the code above in place of `GoForwardALittleBit()`.

Notice that the brain and the behavior each have their own `step` method, and that these are different. This is an example of polymorphism, implemented with the aid of object-oriented programming. In this implementation, the brain's `step` method calls the behavior's `step` method as long as the behavior is not `done`.[1]

Something to watch out for is this: the `setup` method of the brain, and therefore the `start` method of your behavior are called when the brain is *loaded*. Then, when you click `start`, the `step` method of the brain is called repeatedly. If you're using the simulator, and want to reposition the robot by dragging it, do this *before* you reload the brain. If you do it between reloading and hitting `start`, the pose readings will be all wrong.

Primitive terminating behaviors

Implement the following primitive terminating behaviors and test them by themselves using the brain code above.

When you're testing terminating behaviors, you'll need to reload the brain inside SoaR once a behavior has terminated. Why? Because the program has to be run again from the beginning; if it just continues from where it was, the behavior might still think it's done.

[1]Are you wondering why we refer to the brain's step as a *method*, when you don't see any class definition here? There really is a `Brain` class with `step` and `setup` methods, but the class definition is buried inside the implementation of SoaR. There's also a `robot` object that the brain uses in order to store attributes. That's what let's the `setup` procedure store something as `robot.behavior`, which can be referenced by the brain's `step`. You can use `robot` for storing anything that you like, and it will be local to the particular instance of the brain that SoaR is running.

- `ForwardTB(d)` Define a class `ForwardTB`, with an `__init__` method that takes as an argument a distance d to move. Move forward a fixed distance, d, from the current position, along the current heading. *Caveat:* Remember that the robot might be moving at an arbitrary angle with respect to its global coordinate system.
- `TurnTB(a)` Class to turn a fixed angle a (in radians) from the current heading. *Caveat:* Remember that `pose()` returns an angle between 0 and 2π.
- `ForwardUntilBlockedTB()` Class to move forward until the robot is blocked in front.

Think carefully about an appropriate termination condition to use in each case. **Don't start implementing these until you've talked with your LA about your plan!**

Checkpoint: 3:00 PM

- Explain your strategy for implementing each of these classes to your LA.
- Demonstrate each behavior to your LA.

Means of combination

Now, we'd like to implement some means of combining these behaviors so we can make more complex overall behavioral structures. How can we make this happen? We'd like to define a new class, called `Sequence`, that takes as an initialization argument a list of terminating behavior objects, and executes them in order.

To do this, we have to keep track of which action we are currently executing, and keep doing its associated step function until it is done, and then start executing the next action. The code below contains all but the `step` and `done` methods for the class.

```
class Sequence:
    def __init__(self, actions):
        self.actions = actions

    def start(self):
        self.counter = 0
        self.actions[0].start()
```

Checkpoint: 3:30 PM

- Explain your strategy for implementing the `Sequence` class your LA.
- Write the `step` and `done` methods. You might find it easiest to start with `done`.

Hip to be Square

Now that we have primitives and a method for putting them together, we can do interesting things with the robots.

Question 1. What would happen if you executed this code?

```
def setup():
    f = ForwardTB(1.0)
    robot.behavior = Sequence([f, f])
    robot.behavior.start()
```

Use sequencing and your primitives to make the robot drive in a one meter square.

Checkpoint: 4:15 PM

- Explain the answer to the question above to your LA. It might be useful to draw a picture of the behavior instances and their state.
- Execute your program for driving in a square repeatedly and measure its accuracy.

Safety

Write a class `SafeTB` that will have a superclass that is a terminating behavior, and that overrides one of the superclass methods so that the robot will stop rather than run into an obstacle.

Checkpoint: 4:45 PM

- Demonstrate your safe driving by putting your robot too near a wall and asking it to drive in a one meter square.

Exploration

If you have time, pick one or more of these things to do:

- Put the robot inside a square "room" made of foam-core and bubble-wrap. Write a primitive behavior that moves forward until it is blocked in front. Use that to drive in a square, inside the room. Is it more accurate?

  Replace your move-forward behavior, with one that assumes the robot is roughly parallel to a wall, but that uses some of the sideways-looking sonar measurements to try to keep itself parallel (and possibly some fixed distance) from the wall. Does that make it more accurate?
- Use sequence to make your robot do something more interesting than drive in a square.

If you'd like to do extra work on a robot, just come into lab on Wednesday night, or during the TA's office hours on Sunday or Monday.

Post-lab questions

Please write up answers to these questions in coherent English sentences and paragraphs. There will be additional post-lab questions based on Thursday's software lab.

Question 2. We have now seen two different frameworks for making primitive robot behaviors and combining them. Consider a robot for operating in a household, doing chores. Give examples of situations in which each type of behavior combination would be appropriate.

Question 3. Does it make sense to use both strategies? If so, explain how they could be integrated in a sensible and useful way. If not, explain why not.

Question 4. How reliable was your program to drive in a square? What do you think was the main contributing source to the errors? What are some strategies for reducing the error?

Question 5. If we had asked you to write a program to drive in a square before you had read and done this lab, how would you have approached it? Do you think it would have been easier or harder?

Concepts covered in this lab

- Encapsulation of state into objects can provide useful abstraction.
- There are many different frameworks for abstraction and combination, and it's important to choose or design one suited to your problem.
- It can be hard to get repeatable behavior from a physical device.