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

**Work for Week 2**

Issued: Tuesday, Feb. 13

This handout contains:

- Software Lab for Tuesday, February 13
- Pre-Lab exercises due Thursday, February 15 at 2PM; you should come and do them in lab on Wednesday, February 14.
- Robot Lab for Thursday, February 15 (read this before coming to Lab)
- Post-lab exercises due *Thursday* Feb. 22. There is no lecture on Tuesday, Feb. 20 (MIT virtual Monday).

# Higher-order procedures; Behaviors and utility functions

This week's work gives you practice with *higher-order procedures*, i.e., procedures that manipulate other procedures. Tuesday's lecture and post-class lab cover Python's support for *functional programming*, including *list comprehension*. Wednesday's homework and Thursday's lab apply higher-order procedures to the tasks of specifying robot *behaviors* with the aid of *utility functions*.

## Tuesday Software Lab: Higher-order procedures

This session gives you practice with Python's basic tools for functional programming: higher-order procedures and list comprehension.

A higher-order procedure is a procedure that takes takes procedures as inputs and/or returns procedures as results. The basic example we'll work with now is a procedure called `plot` which takes a function `f` as input and graphs it.

Start by loading the file `feb-13-class.py`. It begins with the following code:

```
# import the graphing package from SoaR
import SoaR
from SoaR.Util.GraphingWindow import GraphingWindow

# import sine and cosine
import Numeric
from Numeric import sin
from Numeric import cos

# import the version of addition for use with reduce
import operator
from operator import add as add

# set up a variable for the plotting window
plotWindow = None
```

```
# plot a given function f
def plot(f):
    global plotWindow
    if plotWindow: plotWindow.close()
    xmax=10
    xvals = [i/100. for i in range(0,int(xmax*100))]
    yvals = [f(x) for x in xvals]
    ymax = reduce(max,[abs(y) for y in yvals])
    graphmax = 1.1 * ymax
    plotWindow = GraphingWindow(500, 500, 0, xmax, -graphmax, graphmax, "Plot")
    plotWindow.graphContinuous(f)
    plotWindow.helpIdle()
```

The `plot` procedure opens a graphing window and plots `f` in the interval from 0 to 10. The system then waits (hangs) until you close the plotting window with the mouse.[1] Note that the input `f` to `plot` is a procedure.

**Question** 1. What do you expect `plot(lambda x: x)` to produce? Try it and see.

**Question** 2. Make a plot of the sine function. Do you have to write `plot(lambda x: sin(x))` or can you simply write `plot(sin)`? Try it and see. Make sure to ask if the result isn't what you expect.

Observe how `plot` uses list comprehension and `reduce` with `max` to compute the ranges for the axes. If you've programmed before, you probably expected that this would be done using some kind of loop. One of the hallmarks of functional style is to use functions operating on sets of elements, rather than writing explicit loops with `for` or `while`.

Now let's plot something more interesting—superpositions of sine waves of increasing frequency:

$$\sin x + \frac{1}{2}\sin 2x + \frac{1}{3}\sin 3x + \frac{1}{4}\sin 4x + \cdots$$

out to more and more terms.

Let's call $saw(n)$ the function whose value at $x$ is the sum of the first $n-1$ terms of this sequence. Then we can compute *saw* as

```
def saw(n):
    return lambda x: reduce(add,
                            [1.0/j * sin(j*x) for j in range(1,n)])
```

(This definition is included in the file you loaded.) Here `saw` is a procedure that takes $n$ as input and returns a function of $x$, which we can pass to `plot`. Observe again how we've used list comprehension and `reduce` with `add` rather than write an explicit loop to sum the terms.

**Question** 3. Plot $saw(n)$ for various values of $n$, starting with $n = 2$, which is just a sine wave (do you see why?) up to large values of $n$, like 100. It should become obvious why we've named the procedure *saw*.

[1] We're embarrassed about setting things up this way. It's a kludge to get around a bad interaction between the SoaR robot system and Python's Idle editor.

As you'll learn later (in 18.03 and/or 6.003), any periodic function can be expressed as a *Fourier series*, i.e., a superposition of sines and cosines of the form

$$a_1 \sin x + b_1 \cos x + a_2 \sin 2x + b_2 \cos 2x + \cdots + a_k \sin kx + b_k \cos kx + \cdots$$

**Question** 4. Define and plot the function $squareWave(n)$ that produces partial sums of the Fourier expansion of a square wave:

$$\sin x + \frac{1}{3}\sin 3x + \frac{1}{5}\sin 5x + \frac{1}{7}\sin 7x + \cdots$$

Do this with list comprehension, using a construct of the form:
`[<expression in j> for j in <range> if <condition>]`

**Question** 5. Do the same thing for a triangle wave, whose Fourier expansion is

$$\cos x + \frac{1}{9}\cos 3x + \frac{1}{25}\cos 5x + \frac{1}{49}\cos 7x + \cdots$$

Now let's rewrite these programs using an "even more functional" style, similar to what you'll do in lab this week.

The following procedure `addf` (included in the file you loaded) takes as inputs two functions $f$ and $g$ and returns a function whose value at any $x$ is $f(x) + g(x)$:

```
def addf(f,g):
    return lambda x: f(x)+g(x)
```

The important thing to observe is that the inputs to `addf`, as well as the result, are *functions*, i.e., `addf` transforms functions to functions.

Similarly, we have `scalef`, which takes a function $f$ and a number $s$ and return the function whose value at $x$ is $s \times f(x)$, and `expandf`, which produces the function whose value at $x$ is $f(s \times x)$.

Convince yourself that an equivalent way to write the `saw` procedure is

```
def saw2(n):
    return reduce(addf,
                  [scalef(expandf(sin,j),1.0/j) for j in range(1,n)])
```

and make some plots. The definitions of `addf`, `scalef`, `expandf`, and `saw2` are included in the file you loaded.

**Question** 6. Write similar definitions for square waves and triangle waves and check the results by plotting them.

For the rest of this period, play around with these procedures as you have time, doing things that interest you. Here are some ideas to try. You needn't do all of them.

**Question** 7. Change the square, triangle, and sawtooth wave definitions to use cosines in place of sines and vice versa, and see what kind curves this produces. Or try some other Fourier series.

**Question** 8. Generalize the `plot` procedure so that you can specify the range for $x$.

**Question** 9. Change the plot procedure to use the procedure `plotWindow.graphDiscrete` rather than `plotWindow.graphContinuous`. This is the sort of plotting we'll be doing later in the semester when we study discrete-time systems and difference equations. The plot here plots the function only at *integer* values in the range. If you use range 0 to 10 as given the the original definition of `plot` you'll see only 10 points plotted. If you want a denser plot, you'll have to change the scale.

## Homework for Wednesday and preparation for Thursday lab

Here's what you need to do for Thursday

1. First do the *on-line tutor problems* that are due for February 15.
2. Then read the lab background for the lab and do the preparation. Note that this includes writing some code and trying it with the simulator.
3. Finally, read through the entire description of Thursday's lab so that you'll be prepared to work on it on Thursday afternoon.

The lab background includes a large amount of code for you to read and work with. This may seem overwhelming at first. One of the goals of EECS1 is get you comfortable with reading and understanding moderately long programs. The key to this skill is to learn to distinguish the parts of the code that you need to understand in detail from the parts that you can simply skim.

### Lab background: Behaviors and utility functions

How does a robot faced with a choice of several actions decide which one to do next? This week's topic deals with *utility functions* as a method for choosing. Our implementation of utility functions is based on *higher-order procedures*, and it illustrates the general perspective this course takes on engineering complex systems:

> Start with *primitive elements*, build more complex elements through *means of combination*, and use *means of abstraction* to capture common patterns of use.

We'll be writing programs to give our robots *behaviors*, i.e., purposeful ways of interacting with the world, for example, avoiding obstacles or just wandering around. A behavior is exhibited by a sequence of actions, where an *action* is some some simple thing a robot might do, like go forward or turn. The essence of the behavior lies in the choice, based on the current circumstances, of which action to do next.

One way to choose actions is to associate to any behavior a *utility function*. The utility function takes an action as input and returns a number that represents how much that action is "worth" to the behavior. Or, in other words, it is a measure of how much a particular behavior prefers that a particular action be taken in the current situation. Different behaviors will have different utility functions associated with them. A behavior where the robot tries to avoid obstacles might value turning more highly than going forward, especially in the presence of obstacles, while a wandering behavior might value going forward more than turning in general and not care about obstacles at all. In essence, a utility function provides a mathematical model for the intuitive notion of preferring one action over another.

With the utility function paradigm, a robot following a single behavior decides which action to do next by applying the utility function to each action and picking the action with the largest utility. As we will soon see, utility functions can be used in combining multiple behaviors, so that a robot could, for instance, wander in general but also avoid obstacles that it detects.

The utility function paradigm is an instance of the *transducer* model of programming: the robt first observes the world, then decides what to do—in this case, by computing utilities—and then acts.

## Implementing behaviors and utility functions

In this week's assignment, we'll create some primitive behaviors and combine them to produce compound behaviors. In our Python implementation, we'll represent each behavior as a procedure (namely, the procedure that computes the behavior's utility function, given the current sensory input). As a consequence of this representation choice, Python's mechanisms for manipulating procedures as first-class objects (e.g., naming procedures, passing procedures as arguments to procedures, returning procedures as values of procedures) will be available to us as means of combination and abstraction for manipulating behaviors.

## Actions

The code you'll be working with begins by defining some basic actions. Each action is represented as a list of three things: a string that describes the action (for debugging), a forward velocity, and a turning velocity.

```
stop = ["stop", 0, 0]
go = ["go", 1, 0]
left = ["left", 0, 1]
right = ["right", 0, -1]
```

This choice of a list of three elements establishes a *data structure* for representing actions. We'll also define corresponding *selector* procedures to extract the various pieces from our data structure:

```
def actionForward(action):
   return action[1]

def actionTurn(action):
   return action[2]

def actionString(action):
   return action[0]
```

Next we'll define the list of available actions for the behaviors to work with.

```
allActions = [stop, go, left, right]
```

Given an action, if we actually want the robot to *do* the action, we use the following `doAction` procedure, which executes the action if the action is in the list of available actions and prints an error message otherwise:

```
def doAction(action):
    if action in allActions:
        motorOutput(actionForward(action), actionTurn(action))
    else:
        print "error, unknown action", actionString(action)
```

For instance, to make the robot go left we can execute:

```
doAction(left)
```

## Modeling and abstraction

Our implementation of actions is simple example of the general idea of *modeling with data abstraction*. Namely, suppose we're faced with the task of building a program for working in some application domain. In this case, our application domain involves moving the robot around. We use the things providing by our programming language to represent things in the domain. In this case, we've used lists to represent actions. Next, we define *operations* that work in terms of those representations. In this case, there's only one operation: the operation of doing an action, and we implement that with `doAction`.

Here's the important part: Once we've defined the operations, we *stop thinking about* how the elements are represented, and think only in terms of the operations. So in this case, when we use the command

```
doAction(left)
```

we think "do the action `left`" and we don't think "call `motorOutput` with the second element of the list and the third element of the list." That's a lower level of detail.

It's important to separate different levels of detail as you design programs and read programs others have written. Trying to think about the different levels all at once will almost certainly lead to confusion.

The ability to look at some aspect of a system while suppressing details about other aspects of the system—treating them as *black boxes*—is critical to the ability to handle complexity. We'll see this idea repeatedly throughout the semester. We'll meet it again more formally next week with the idea of *abstract data types*, and we'll also see it when we study electrical circuits and signal-flow models.

### Behaviors are procedures that return utility functions

A utility function, as we noted at the outset, is a function that takes an action as input and returns a number. For example:

```
def f(action):
   if action == go: return 2
   else: return 0
```

is a utility function that returns 2 if the action is `go` and returns 0 for any other action.

A behavior is represented in our system as a procedure that returns a utility function. The idea is that the utility function, for any action, returns a number that indicates how much that behavior "prefers" that action. In our implementation, we'll use numbers ranging from 0 to 10: a 10 means maximum preference, while 0 means that the behavior has no preference at all for the action.

The input to the behavior procedure is a list of readings reported by the sonars, which we'll call `rangeValues`, since in general the behavior's preference for an action will depend on the sensor readings.

Here is a simple primitive behavior that makes the robot wander around (stupidly). It ignores the sonar readings and simply prefers going forward (10) to turning left or right (2), and never wants to stop (0):[2]

```
# Primitive wandering behavior
def wander(rangeValues):
   def uf(action):
      if action == stop: return 0
      elif action == go: return 10
      elif action == left: return 2
      elif action == right: return 2
      else: return 0
   return uf
```

Here's an example of how this behavior procedure might be used:

```
wanderUtility = wander(rangeValues)
utilityOfLeft = wanderUtility(left)
```

In this code snippet, the call to `wander` returns a utility function, to which we have assigned the name `wanderUtility`. This utility function takes an action (e.g., `left`) as argument and returns a number. In this for `left`, the number will be 2. If we'd instead asked about `go`, the utility would be 10.

Another way to write this, without defining the intermediate `wanderUtility` would be

```
wanderUtility = wander(rangeValues)(left)
```

[2] We have to use an internal definition here, rather than `lambda`, because of Python's restriction that a `lambda` can be only a single expession.

### The avoid behavior

Here's a more complex primitive behavior that makes the robot attempt to avoid obstacles. Unlike wandering, avoiding does use the distances reported by the sonars in computing the utility of an action. For example, going forward will have a larger utility when there is more free space in front of the robot.

The `avoid` procedure takes the sonar distances as input and return the corresponding utility function:

```
# Primitive behavior for avoiding obstacles
# rangeValues is a list of the distances reported by the sonars
def avoid(rangeValues):
   # establish a minimum and maximum distance
   mindist = 0.2
   maxdist = 1.2

   # clip a given value between mindist and maxdist
   def clip(value): return max(mindist, min(value, maxdist))

   # Stopping is always good at avoiding, so give it a utility of 10
   stopU = 10

   # The utility of going forward is greater with greater free
   # space in front of the robot.  To compute the utility we read the front
   # sonars and find the minimum distance to a perceived object.
   # We clip the shortest observed distance, and scale the result
   # between 0 and 10

   minFrontDist =  min(rangeValues[2:6])
   goU = (clip(minFrontDist) - mindist) * 10

   # For turning, it's always good to turn, but bias the turn in favor
   # of the free direction.  In fact, the robot can sometimes get stuck when it
   # tries to turn in place, because it isn't circular and the back
   # hits an obstacle as it swings around.  Think about ways to fix
   # this bug.

   minLeftDist = min(rangeValues[0:3])
   minRightDist = min(rangeValues[5:8])
   closerToLeft = minLeftDist < minRightDist
   if closerToLeft:
      leftU = 5
      rightU = 10
   else:
      leftU = 10
      rightU = 5

   # Construct the utility function and return it
   def uf(action):
      if action == stop: return stopU
      elif action == go: return goU
      elif action == left: return leftU
      elif action == right: return rightU
      else: return 0
   return uf
```

One thing to realize is that neither the avoid behavior nor the wander behavior is any good if used alone. With only `wander`, the robot will always choose `go` (since it has a utility of 10), which will pin it up against a wall with its wheels turning. With only `avoid`, the robot will always choose `stop`![3]

## Combining behaviors

In order to produce something more useful, we can combine behaviors. For example, given two behaviors with utility functions $u_1$ and $u_2$, we could consider the behavior whose utility for any action $a$ was $u_1(a) + u_2(a)$. Adding the utilities for `wander` and `avoid` essentially produces a new utility function, now with a value range from 0 to 20, that takes in an action and reports how good that action is for both wandering in general and avoiding obstacles at the same time, given the current sensor readings. If you wanted to return a value in the range to 0 to 10, you could scale both returned values by 1/2 before combining (we will discuss scaling later).

This kind of addition is a *means of combination* for utility functions: given two utility functions, the result is a new utility function whose value is the sum of the original two. We can implement this as a procedure `addUf`:

```
def addUf(u1, u2):
   return lambda action: u1(action) + u2(action)
```

The procedure `addUf` is a *higher-order procedure*: it takes two procedures as inputs and returns a procedure as value. The value returned by `addUf` is itself a utility function (represented as a procedure). Here we've used `lambda` to create that procedure.

An equivalent way to have written this without `lambda` would be:[4]

```
def addUf(u1, u2):
   def uSum(action):
      return u1(action) + u2(action)
   return uSum
```

In either form of the definition, `addUf` takes in two utility functions and returns a new function (`uSum`) whose value on any action is the sum of the values of the two utility functions on that action.

Here's a step-by-step example that shows how the pieces fit together:

```
# read the sonars
rangeValues = sonarDistances()

# use those sonar values to get utlity functions for wandering and avoiding
wanderUtililty = wander(rangeValues)
avoidUtililty = avoid(rangeValues)
```

[3] You should convince yourself of this by tracing through the computation to see that the maximum value for `goU` will always be 10, which is the same as `stopU`, so that `stop` will always have a utility at least as large as `go`. Then note the comment in footnote 4 about how `bestAction` chooses the largest utility action.

[4] In Python, `lambda` can be used only for a single expression. Notice that you do *not* write `return` inside the `lambda` expression, although you *do* use `return` to return the lambda expression itself as the value produced by `addUf`. If the procedure to be returned by `addUf` had done something more complicated, e.g., using a conditional expression, then we could not generate it with `lambda` and would instead have to use the embedded subprocedure form. This seems (to Lisp programmers) to be an arbitrary restriction imposed by Python.

```
# add the resulting utility functions
combinedUtililty = addUf(wanderUtililty, avoidUtililty)

# evaluate the combination on the stop action
value = combinedUtililty(stop)
```

If you recall, the result returned by calling `wander(rangeValues)` and `avoid(rangeValues)` each are procedures that take in an action and output a number. Here we've assigned those resulting procedures new names, `wanderUtililty` and `avoidUtililty`. Those two procedures are passed to `addUf`, which returns a new procedure (which we've assigned the name `combinedUtililty`) that takes in an action and returns the sum of the values obtained by calling `wanderUtililty(action)` and `avoidUtililty(action)`. In this case, the action is `stop`, so value will be $0 + 10 = 10$, regardless of what the sonar readings were.

## Performing a behavior

Finally, here's the robot brain that performs a behavior, using a loop that runs over and over. Each time through the loop, the robot

1. Reads the sensors
2. Evaluates the behavior on those sensor readings to produce a utility function
3. Applies that utility function to each of the available actions
4. Picks the action with the highest utility and does it

Observe that this adheres to the general form for *transducer-style* programs described in the first lecture: (a) observe the state of the world, (b) think, (c) do an action; and repeat this sequence over and over.

Here's the `step` method for a robot brain that carries out this recipe:

```
def step():
   # read the sonars
   rangeValues=sonarDistances()

   # get the utility function
   u = wander(rangeValues)

   # after you verify that the program runs, comment out the
   # u = wander() line and uncomment the u=addUF(...) line
   # and see how the behavior changes
   # u = addUf(wander(rangeValues), avoid(rangeValues))

   # pick the best action and do it
   useUf(u)
```

The `step` procedures uses two subprocedures that we've defined to (hopefully) make the code more readable:

```
# Pick the best action for a utility function
def bestAction(u):
   values = [u(a) for a in allActions]
   maxValueIndex = values.index(max(values))
   return allActions[maxValueIndex]

# to use a utility function u, pick the best action for that
# utility function and do that action
def useUf(u):
   action = bestAction(u)
   # Print the name of the selected action procedure for debugging
   print "Best Action: ", actionString(action)
   doAction(action)
```

The `bestAction` procedure applies the given utility function to all the available actions and picks the action with the largest utility.[5] The `useUf` procedure finds the best action for that utility function and does it.[6]

We've set up the code so that initially, all the robot does is wander. Once you've verified that things are working, you can modify the commented line to change the behavior to be the sum of avoid and wander.

### Programming with the simulator: Weighted sums of behaviors

You should do this section *before* Thursday's lab, for example, on Wednesday evening when you can get help.

The code above is contained in the file `ps2-utility.py` from the course web site. Download it an put it in the SoaR brains folder on your computer. Start up Idle and SoaR, choosing the simulator and the LongHall world. Load `ps2-utility.py` as the brain. Also open `ps2-utility.py` in Idle for editing so that you can easily edit and save the code with Idle, and then reload the brain code in SoaR to try your edits.

Run the robot. The utility function is initially set simply to `wander`, which is pretty boring: the robot should quickly run into a wall and get stuck there.

Edit the code so that the utility is now the sum of the utilities for `wander` and `avoid`. You should find that this works pretty well, although you might see the robot get stuck when it runs into a *cul de sac* or gets stuck on a corner. You can unstick the robot by dragging it to an open space with the mouse.

[5] Observe how finding the element with the maximum value is implemented using list comprehension together with Python's builtin `index` operation, which finds the index of a given value in a specified list.

[6] If there are several actions that all have the same highest utility, `bestAction` will return the first of these actions. This is a consequence of using `index` to select the desired value from the list of all values. That's why the `avoid` behavior will always result in `stop`, even when `stop` and `go` both have the highest utility. In a different implementation, we might check to see whether more than one action has the same highest utility and choose at random from among them. It might be tempting to "debug" avoid's boring behavior by using this random selection approach, but that's not such a great idea, because it makes the program nondeterministic and consequently harder to debug. In general, simply adding randomness without trying to understand things better probably won't work very well—although there *are* some great applications of randomness developed in theoretical computer science in the "theory of randomized algorithms". In fact, there are cases in robot control where it's provably better to add randomness (getting out of metastable states, or "jittering" to do things like insert a key in a hole).

With the sum of the two utility functions, the robot's behavior is governed both by wandering and by avoiding. You can adjust the relative amount of wandering vs. avoiding by using a *weighted sum* of the two utilities. For example, you might want to use a utility function that combines the utilities `wander` and `avoid`, but where `avoid`'s utility is weighted twice as much as `wander`'s.

Define a new means of combination `scaleUf`, which takes a utility function `u` and a number `scale`, and returns a utility function whose value for any action is the `scale` times the utility value returned by the function `u` for that action. (There's a partial definition of `scaleUf` in the code file, which you should uncomment and complete.)

You can test your `scaleUf` by changing the behavior used in `step`:

```
u = addUf(wander(rangevalues), scaleUf(avoid(rangeValues),2))
```

Try different values of the scale factor to see how they compare. With more weight given to `avoid`, the robot is more conservative in avoiding obstacles, with the result that it may not see as much of the world as one might like. With more weight given to `wander`, the robot is more aggressive, covering more of the space, but risking getting stuck when it approaches to close to a wall or other obstacle.

| Mail your code to yourself or put it in your Athena locker so that you can get it when you come into lab and load it on your robot's laptop. |
|---|

## To do in lab

*Skim this section before coming to lab.*

Start by loading the problem set code, completed with your scaled sum of behaviors program, onto your robot's laptop. Check that the code still works on the simulator and then try it with the real robot. Send your robot on a walk around the lab or out into the corridor and see if it is doing anything that might sensibly be described as wandering and avoiding obstacles. Does the robot get stuck?

### New behaviors

Invent and implement a few new primitive behaviors. When you implement, remember that a behavior in our system is a procedure that takes the sonar readings in `rangeValues` and returns a utility function, which itself is a procedure that takes an action as input and returns a number.

This is really open-ended. You and your partner should do some deliberate planning, and check with your LA for programming advice before you get too deeply into implementation.

Here are some ideas, but feel free to make your own. It would be a good idea to do something simple first, before trying things that are more elaborate.

- *Scared:* Move away from things that you sense. You may want to add `back` as a new primitive action in order to implement this behavior.
- *Friendly:* Move towards things that you sense, but try not to actually run into them. (That would be aggressive, not friendly.)

- *Persistent:* Do the same thing you did last time.

  "Persistent" is a little tricky because our transducer model of programming doesn't have explicitly give a way to remember information from one round to the next. So you'll have to build that. Here's one way to go about it.

  1. Define a global variable called `actionLastRound`, which is the action that the robot actually did on the last round. Initialize it to some action of your choice, e.g., stop.[7]
  2. "Persistent" is a behavior whose utility function returns 10 if the action is `actionLastRound` and returns zero otherwise.
  3. Modify the brain step loop, so that at the very end of the loop, after it chooses the action to do, it saves the action as `actionLastRound`,

     Of course, if you simply run the persistent behavior by itself, the robot will perform the same action each time.

- *Onward:* Try to keep going in some unifrom direction, even if there are occasional turns to avoid obstacles. This is similar to persistent in that it requires creating some variables to keep track of the robot's state from one round to the next.

**Checkpoint: 3:00 PM**

- Describe two of your new primitive behaviors and demonstrate them both on the simulator and with the real robot.
- Demonstrate that your implementation of these behaviors is compatible with the adding and scaling means of combination. For example, if you've defined the scared behavior, you ought to be able to run the robot with

  ```
  u = addUf(wander(rangeValues), scared(rangeValues))
  ```

  as the utility function in the step loop, and similarly with scaling.

## Means of combination

Invent some new means of combination for utility functions to supplement adding and scaling. Remember that to work compatibly with our system, a means of combination must be a procedure that takes one or more utility functions and returns a utility function.

Here are some ideas:

- *Maximum:* Given two utility functions, return for any action the maximum value of the two utilities for that action.
- *Most eager:* Given two utility functions, pick the one that gives the highest weight to going forward. (Notice that the robot might not actually choose to go forward as a result, since some other action might still have a larger weight.)

[7] Remember that in order to change the value of a global variable from within a procedure, you need to declare it as `global`.

- *Convex combination:* Given two functions $u_1$ and $u_2$, and a number $p$ between 0 and 1, use the function whose value is
$$p \times u_1 + (1 - p) \times u_2$$
- *Random choice:* Given two utility functions, pick one at random. If you decide to implement this, you should use the Python `choice` procedure, which picks an element at random from a list. Import `choice` from the `random` module.
- *Directed choice:* This takes two utility functions and a *test* procedure (which might to a test that depends on the sensor values). It uses utility function 1 if the test is true, and utility function 2 otherwise. For example, the robot might act curious unless it is very close to something, in which case it should act scared.

With primitives and means of combination, you now have the beginnings of a vocabulary for assembling a wide variety of robot behaviors. For instance, you could put combine elements of wandering with persistence and eagerness (maybe with some appropriate scaling)

Make sure that you've constructed your means of combination so that they are *composable*, i.e., so that your compound behaviors can themselves be combined to form more complex behaviors. For example, you ought to be able to write things like

```
addUf(maximum(wander(rangeValues),persistent(rangeValues)),
      scaleUf(randomChoice(friendly(rangevalues), scared(rangeValues)),2.0))
```

although you probably want to define some procedures (means of abstraction) to help you create such combinations without having to type such complex expressions.

Now that you have primitives and means of combination, you have the beginnings of a vocabulary that you can use to assemble a wide collection of behaviors.

Try out some behaviors, both with the simulator and the real robot. You'll probably find that the robot won't always behave as you expect, often because the sensor readings can be so unreliable and *especially* if you use any randomness (which is not usually a good idea if you want to make something that is understandable and debuggable).

**Checkpoint: 3:45 PM**

- Demonstrate two means of combination and the resulting behaviors in your repertoire, both on the simulator and the real robot, illustrating various compositions. Describe some of the interesting things you've observed, both expected and unexpected.

## Quality metrics

By now you've played around with a lot of different behaviors, with a lot of choices in making combinations and varying scale factors and the like. You might be wondering whether there is some more deliberate way of finding good behaviors than just trying things at random.

A first step is to try to define what you mean by "good behavior" and express that as a rating, or a *quality metric* that rates how good the behavior is. Of course, the metric depends on what you'd like to accomplish: for example, you might want to keep from bumping into objects, or

maximize the distance the robot manages to go forward without turning, or some combination of these criteria.

Once you decide on a quality metric, you can run the robot for a while and measure its performance. Given the variability in sonar readings you might actually want to run the robot several times and get some average rating of the performance.

If your behavior depends on an adjustable parameter, you can see how the behavior quality varies as you change the parameter. You can even write a program that tries to optimize the behavior by automatically adjusting the parameter.

Carry out as much of this plan as you have time for. You'll need to decide on a quality metric and some range of behaviors to explore. Then you'll need to modify your program so that it computes this metric and records it. And you'll need a plan for adjusting the parameters.

Don't be surprised if, after doing a careful job of optimizing your quality metric, the robot's behavior looks "worse" than it did before. This is a case of needing to be careful what you wish for: it can sometimes be hard to intuit what the optimal behavior for a quality metric will look like. In a real application, you might decide that you need to change your metric.

**Checkpoint: 4:45 PM**

- Demonstrate your results to your LA. Even if you don't have all this working, you should be able to discuss your plan for an approach, including:
  - What's your metric and how do you change the program to keep track of it?
  - What's the behavior you're exploring and what are the tunable parameters?
  - What's your strategy for optimizing the quality by varying the parameters, and how do you modify the program to do this?
  - If you've manged to get this working, show your data and the resulting parameter choices.

## Post-Lab Exercises: Due before lecture on Feb. 22

There are more on-line tutor problems to do for this assignment. Start by completing them.

### Programming practice: Roots of polynomials

Last week, you write a program that solves quadratic equations. This week, you'll try Newton's method as a way of finding roots of arbitrary polynomials. This programming assignment provides more review of abstractions expressed as higher-order procedures.

The following code was presented in last Tuesday's lecture. You can retype it in, or load it from the file `ps2-poly.py`.

```
tolerance = 1.e-4
dx = 1.e-4

## test whether two values are close
```

```
def close(g1,g2):
    return abs(g1-g2)< tolerance

## compute a fixed point of the function f by
## by applying f over and over until the result
## doesn't change very much
## guess is the value that starts the interation
def fixedPoint(f,guess):
    next=f(guess)
    while not close(guess, next):
        guess=next
        next=f(next)
    return next

## given a function f, produce the function that
## approximates the derivtive of f
def deriv(f):
    return lambda x:(f(x+dx)-f(x))/dx

## find a zero of the function f using Newton's method
## (expressed as a fixed point)
def newtonsMethod(f,firstGuess):
    return fixedPoint(
        lambda x: x - f(x)/deriv(f)(x),
        firstGuess)
```

### Evaluating polynomials with Horner's Rule

We can represent a polynomial as a list of coefficients starting with the highest-order term. For example, the polynomial $x^4 - 7x^3 + 10x^2 - 4x + 6$ would be represented as the list `[1,-7,10,-4,6]`. One good way to evaluate a polynomial at a given point is to use *Horner's Rule*, which structures the computation of

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

as

$$(\cdots (a_n x + a_{n-1})x + \cdots + a_1)\, x + a_0$$

In other words, we start with $a_n$, multiply by $x$, add $a_{n-1}$, multiply by $x$, and so on, until we reach $a_0$. For example, we'd evaluate $8x^3 - 3x^2 + 4x + 1$ as

$$((8 \times x - 3) \times x + 4) \times x + 1$$

Horner's Rule can be expressed recursively:

To evaluate the polynomial represented by $[a_n, a_{n-1}, \ldots, a_1, a_0]$ at $x$

1. Evaluate the polynomial represented by $[a_n, a_{n-1}, \ldots, a_1]$ at $x$
2. Multiply the result by $x$ and add $a_0$.

The termination condition happens when we reach a list with only one coefficient $a_n$, in which case the result is just $a_n$.

**Question** 10. Define a procedure `hornersRule` that takes a polynomial $P$ represented as a list of coefficients, and a value $x$, and returns the value of $P(x)$. Demonstrate that your procedure works by exhibiting some test cases. (Note: If `k` is a list in Python, then `k[-1]` returns the last element and `k[:-1]` returns the list with the last element removed.)

**Question** 11. A more straightforward way to evaluate polynomials is to explicitly add up the terms $a_i x^i$. We can do this with list comprehension and `reduce`

```
reduce(add,[coeffs[i]*x**(k-i-1) for i in range(0,k)])
```

where `k` is the length of the list. Make sure you understand the reason for the $-1$ in the exponent. Use this to define another procedure to evaluate and verify that it works. Remember that `add` here should be the `add` imported from the `operator` module.

**Question** 12. Explain why Horner's Rule provides an advantage over the straightforward method in terms of number of arithmetic operations required. Of course, the computer's we're using are fast enough so that this isn't a significant advantage for most applications.

### Polynomial roots

You can now evaluate polynomials, but if you want to use this together the `newtonsMethod` procedure, you'll need the polynomial in the form of an actual procedure, not just a list of coefficients.

**Question** 13. Define a procedure `coeffsToPoly` that takes a list of coefficients representing a polynomial $P$ and returns the the procedure whose value at $x$ is $P(x)$. For example, `coeffsToPoly([1,2,3])` should be a procedure whose value at 10 is 123.

**Question** 14. Demonstrate how to use `coeffsToPoly` together with `newtonsMethod` to find roots of various polynomials. As a first test, find a root of $P(x) = x^5 + 2x^4 + 3x^2 + 4x - 5$. Evaluate the $P$ at the answer you get to demonstrate that this is indeed a root.

**Question** 15. What happens if you use Newton's method to try to find a root of $x^2+2$? Try this starting the iteration at 1, verify that it doesn't work, and explain why you wouldn't expect it to work. Show, on the other hand, that the method works if you start the iteration at $\sqrt{-1}$, written in Python as `0+1j`. A cute thing about our Newton's method procedure is that Python automatically does complex arithmetic on complex inputs, and our fixed points and derivative and Newton's procedures don't make any assumptions about the inputs being real numbers. Show that you can use your procedure to find a solution to the equation $x^5 = 1$ (other than $x = 1$).

**Question** 16. Use your procedure to find a root of $x^4 + 2x^3 + 3x^2 + 4x + 5$. Does it work if you start the iteration at 1? at $\sqrt{-1}$?

**Question** 17. Try to find roots of some polynomials of your choice, starting at various initial values. You should be able to find lots examples where it doesn't work: either the iteration doesn't converge and the program runs forever, or the computation blows up, or you get an answer that's simply wrong. Give some examples to illustrate things that work and that don't work. You might also consider the effect of changing the values of `dx` and `tolerance`.

The attraction of the code shown here is that it's simple. But really good polynomial root finders that work in almost all cases are complicate, delicate algorithms.

Write up your answers and turn them in at lecture on Thursday, Feb. 22. When you write up answers to programming problems in this course, *don't* just simply print out the code file and turn it in. Especially, don't turn in long sections of code that we've given you. Turn in your own code, with examples showing how it runs, and explanations of what you wrote and why.

## Concepts covered in this assignment

Here are the important points covered in this assignment:

- One general perspective on engineering complex systems is to start with *primitive elements*, build more complex elements through *means of combination*, and use *means of abstraction* to capture common patterns of use.
- As a programmer, you have a lot of powerful tools at hand for *modeling and representation*. To make effective use of these, it's important to use *abstractions*, so that you can avoid thinking about all details of a system at the same time.
- *Higher-order procedures* can be powerful tools in computer modeling because the computer language's capabilities for manipulating procedures—naming, functional composition, pa-

rameter passing, and so on—can be used directly to support means of combination and abstraction.

- Utility functions can be a powerful technique for organizing decision making. More generally, utilities illustrate the general approach of making a mathematical model that reflects choices and preferencs.
- Moving from simulation to the real world (e.g., the robot) isn't as straightforward as it might seem. Systems that work in the real world need to be tested and have their performance measured, in order to achieve good performance.

You also got more programming practice with Python.

## Reflections on the lab: Continuous spaces of actions

The following section is not required. It's something that you can work on optionally for your own interest, and that we'd be happy to discuss with you.

In the lab, we considered situations where the robot chooses from among a finite set of actions. And yet all the actions in our application are of the form `motorOutput(x,y)` with $-L \leq x \leq L$ and $-R \leq y \leq R$, where $L$ is `maxTransSpeed` and $R$ is `maxRotSpeed`. This suggests extending our finite set of actions to allow any `motorOutput(x,y)` with $x$ and $y$ in the designated range. This is a *continuous space of actions*: an action for any point in the rectangle $\{(x, y) | -L \leq x \leq L, -R \leq y \leq R\}$.

How could we rewrite the code you've been working with to deal with a continuous space of actions, while keeping the overall organization of the program intact, changing as little as possible?

Here's an outline of how we might proceed:

(a) Rather than having a finite list of actions, we'll now have one for every pair $(x, y)$. So we may as well change the representation of actions so that an action is just the list $[x, y]$. The procedure `useUf` in the robot brain's `step` could then be

```
def useUf(u):
    [x,y]=bestAction(u)
    print "Best xy: ", [x,y]
    motorOutput(x,y)
```

(b) A utility function will now be a represented as a procedure that takes as arguments a list $[x, y]$ and returns a number (the utility). The number will be 0 if the pair is outside the rectangle. Inside the rectangle, the number will depend on the behavior you want to express. For example, wandering could have non-zero utility for any $(x, y)$ while preferring actions with small turning speed, and not going too fast.

The means of combination—`addUf`, `scaleUf` and any others you've defined—don't need to change at all.

(c) The big change in the program will be in `bestAction`. With an infinite choice of actions, we can't just try all the pairs $(x, y)$ and pick the one with the highest utility.[8] What we have

[8] At least, not in any technology known in 2007!

here is a *two-dimensional maximization problem*: find the pair $(x, y)$ in the rectangle such that $u(x, y)$ is maximized, where $u$ is the utility function.

One way to find the maximum is to subdivide the rectangle by a grid (by subdividing the intervals for $x$ and $y$), evaluate the function at each grid point select the maximum. Of course, the more accurately you want to locate the maximum, the more finely you should lay down the grid, and the more computing you'll need to do in scanning all the grid points to locate the maximum. For this application, scanning a coarse grid might be good enough. But there are also sophisticated algorithms for finding maxima of functions of two (or $n$) variables, and you could take this problem as an opportunity to learn about some of them.

*For extra edification:* Do some research on maximization algorithms. For example, see if you can find anything on the web on that describes the Nelder and Mead's *downhill simplex method*. You can write up a paragraph or two on what you find. You need not write any code. (And you don't need to just quote the Wikipedia: we can read it, too.)

If you are very ambitious, you could try to implement one of these algorithms. There are some Python implementations of maximization code that you should be able to find on the Web (for instance: SciPy's optimize.fmin function (you have to install SciPy and import scipy.optimize separately from just scipy), that uses a downhill simplex algorithm), and we invite you to download one of them and see if you can get it running. Reading other people's code and trying to make it work is a good way to learn a computer language. You could easily spend days working on this problem. Don't. This is only part of the second week's homework, and there will be plenty more opportunities for open-ended work throughout the semester.

## Utility functions: Summary and perspective

This section is for reading only: there are no problems to do. Hopefully, it will help you look back on this week's assignment and provide a more general perspective.

### Utilities

The notion of utility is an important concept for measuring the quality of decision-making processes. A decision-making agent's utility for a state is a fundamental measure of how much that state is valued by the agent. Agents are said to be "rational" if they choose actions they believe will maximize utility.[9]

Any agent (e.g., a robot) can be viewed as having a set of possible actions that it can choose among. For now, let's think of our robot as having a discrete set of actions: stop, go, left, and right. Now, we can also think of the robot as having a set of (possibly competing) goals: to move around, to not run into the wall, to stay charged, to make its owners happy, etc. One way to articulate these goals is in terms of primitive utilities assigned to states of the world. With respect to the goal of not running into things, the state of being crashed into something has low utility, for example.

Although the basic notion of utility adheres to states of the world, we can also regard primitive actions as having utilities: the utility of an action, in the current state of the world, is the utility

[9] We assume for now that an agent knows everything about the state of the world, but we will return to examine the issue of beliefs later this semester.

of the state that would result from taking the action. So, when the robot is close to the wall, the utility of moving forward is low because the utility of the resulting (crashed) state is low. When the robot is far from the wall, the utility of moving forward might be high.

## Specifying behavior

The most straightforward way to specify the behavior of an agent is to directly write down a "behavior" or "policy", which is just a function that maps from percepts, or observed states of the world, to actions. A program that says "if there is an obstacle close in the front, then stop, otherwise move forward" is a behavior. Most programs are of this form.

On the other hand, for reasons of modularity, we would like to be able to write some sort of a program for moving around and another for avoiding obstacles, and combine them to get a program that moves around while avoiding obstacles. This can be hard to do if we specify behaviors directly: you might imagine a situation in which the "wander" behavior chooses to move forward and the "avoid" behavior chooses to stop. Then there's no clear way to combine these actions to do something reasonable. In reality, the wander behavior might have also been happy to turn left or right; and those actions might have been suitable for the avoid behavior; but we would have no way of knowing that.

So instead of programming behaviors directly, we'll specify utility functions for each "goal". The avoid behavior will say, for each primitive action, what the utility of the resulting state would be. This reveals more information than an arbitrary choice of a single action, and allows the combination of subgoals to be done much more intelligently.

In the language of higher-order functions, a utility function is a function that maps a state into a function that maps an action into a real value.

## Combining utility functions

If we have only a single goal, then we can, on each step, simply choose the action that maximizes the utility function for the current state. But what if we have multiple goals? It will very rarely be the case that there is a single action that maximizes both utility functions (in such a case, the action is said to *dominate* the other actions). So, we have to decide how to combine the functions. There is an elaborate theory of so-called "multi-attribute decision making" that deals with these kinds of situations. A classic example is a decision about where to situate a new airport. The "actions" are the possible airport sites. The component utility functions are to[10]

- minimize the costs to the federal government
- raise the capacity of airport facilities
- improve the safety of the system
- reduce noise levels
- reduce access time to users

[10]Example taken verbatim from *Decisions with Multiple Objectives*, Keeney and Raiffa, Cambridge University Press, 1993.

- minimize displacement of people for expansion
- improve regional developments (roads for instance)
- achieve political aims

In this problem set, we took a simple approach of maximizing a linear combination of the utility functions of the subgoals. So, we might give twice the weight to the avoid utilities as to the wander utilities, and choose the action that maximizes this weighted combination of utilities. Such a simple combination might not always be sufficiently nuanced, but it's a good way to start.