MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.081—Introduction to EECS I
Fall Semester, 2006

Assignment for Week 3

- Issued: Tuesday, Sept. 19
- Includes homework and preparation due before lab on Thursday, Sept. 21
- Also includes post-lab homework due at the beginning of class on Tuesday, Sept. 26

This week's assignment includes a substantial amount of code for you to work with. One of the goals of the class is not only to get you to write simple programs, but also to read and understand significant programs.

Please read this entire document before coming to lab on Thursday. As usual, there are homework problems to work on Wednesday night to help you prepare for lab and post-lab homework to be done before the next lecture.

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 than going forward 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. 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. In essence, a utility function provides a mathematical model for the intuitive notion of preferring one action over another. 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.

Assignment background:
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). 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 forward velocity, a turning velocity and a string that describes the action (for debugging):

```
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 a 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.

Utility functions and behaviors

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 one action or another 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):

```
# 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 case, the number will be 2.

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

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

Here's a more complicated primitive behavior that makes the robot attempt to avoid obstacles. This behavior does take account of the distances reported by the sonars and returns the appropriate utility function for those distances:

```
# 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
```

Neither of these primitive behaviors is any good if used alone. If only `wander` is used, 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. If only `avoid` is used, the robot will always choose `stop`![1]

Combining utilities

In order to produce a more useful behavior, we can combine utilities. Here's a means of combination that adds two utility functions: the value of the resulting utility function for an action is the sum of the individual utilities. Adding `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 the value range to 0 to 10, you could scale both returned values by 1/2 before combining (we will discuss scaling later), not that the range of values matters; only the relative values matter. Here is the combining function:

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

The procedure `addUf` is a *higher-order procedure*: it takes two procedures as input 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:[2]

[1] 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 highest utility action.

[2] 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.

```
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. Thus, you could write:

```
rangeValues = sonarDistances()
wander_utility = wander(rangeValues)
avoid_utility = avoid(rangeValues)
combined_utility = addUf(wander_utility, avoid_utility)
value = combined_utility(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, `wander_utility` and `avoid_utility`. Those two procedures are passed to `addUf`, which returns a new procedure (which we've assigned the name `combined_utility`) that takes in an action and returns the sum of the values obtained by calling `wander_utility(action)` and `avoid_utility(action)`. In this case, the action is `stop`, so value will be $0 + 10 = 10$.

Picking the action

Finally, our robot brain needs a `step` method that says what to do at each cycle in the robot control loop.

Our `step` reads the sonars and uses a behavior that is the sum of wandering and avoiding (with observed sonar readings).

```
def step():
   # 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)

   # for debugging it's best to read the sonar distances at a
   # well-defined point in the step look, rather than sprinkling
   # calls to sonarDistances() throughout the entire program
   rangeValues=sonarDistances()

   # 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 = wander(rangeValues)
   # u = addUf(wander(rangeValues), avoid(rangeValues))

   useUf(u)
```

We've defined two subprocedures to (hopefully) make the code more readable. The `bestAction` procedure applies the given utility function to all the available actions and picks the action with the largest utility.[3] The `useUf` procedure finds the best action for that utility function and does it.[4]

Notice that the `step` loop has the following form:

1. read the sensors
2. think
3. send a motor command

When you write step functions, we *strongly* recommend that you follow this format, rather than intermixing sensor reading and motion commands with your other processing. It will make your programs much easier to understand and debug.

[3] Observe that the code `values.index(max(values))` is essentially the `findMax` procedure you wrote for last week's lab. This time, we've used Python's builtin `index` operation, which finds the index of a given value in a specified list.

[4] 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).

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.

1. Homework in preparation for lab on Sept. 21

1.1. Exercises with the online tutor

Use the online tutor to complete the problems due for Sept. 21.

1.2. Programming with the simulator—Weighted sums of behaviors

Start by downloading the file `ps3-utility.py` from the course web site, and put it in SoaR/brains on your computer. This is the code explained above. Start up SoaR with the simulator and the LongHall world. Also set up `ps3-utility.py` for editing by loading it into Emacs or IDLE so you can easily read it, edit it, and then **reload brain** 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 (in the LongHall world) 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 with the mouse to an open space.

With the sum of the two utility functions, the robot's behavior is governed both by wandering and by avoiding. We can adjust the relative amount of wandering vs. avoiding by using a weighted sum of the two utilities. For example, we might want to generate a new 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 `s`, and returns a utility function whose value for any action is the `s` times the utility value returned by the function `u` for that action.

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

```
u = addUf(wander(), 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 on your Athena locker so that you can get it when you come into lab and load it on your robot's laptop.

2. To do in lab on Sep. 21

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

2.1 Quality metrics

In any engineering task, when you set out to do a job, you should have in mind what your measure of success is. We'll apply that principle to our behaviors, and their combination.

Let's say you work for a company that wants to buy a robot brain, and you are tasked with picking the best one to buy. There are 100 different companies selling robot brains, and each is slightly different in some way. One way to pick would be to try out each brain, stare at the robot and its behavior, and pick the one that seems to behave the best. But what if there are 1000 different programs? Or 10,000? You'd like an automatic method of determining how good each program is, so that you don't have to sit there and stare at each one yourself. Ideally, you'd like to set an automatic tester to run each brain in turn on the robot, and to rate that brain with a number that describes how well it does (also called a *quality metric*), according to some criteria. In our case, for instance, we might like a brain that avoids obstacles successfully but also wanders around covering lots of ground. So a brain that just goes forward and bumps straight into walls should be assigned a low quality value, as should a brain that doesn't hit anything but that spins in tight little circles. On the other hand, a brain that does a good job of wandering without hitting things should be assigned a high quality value.

The first step is to formulate the quality metric. One approach is that each behavior is meant to satisfy a constraint: that is, there is an all-or-nothing measure of the quality of the behavior. In that case, we might require that the *avoid* behavior never runs into anything and that the *wander* behavior never stops moving. What would it mean to combine these two goals? The new behavior would have to satisfy *both* of the criteria: that is, never run into anything and never stop.

For some simple behaviors and combinations like this, we can work with rigid constraints. More generally, we will have to compromise: we might seek to minimize the number of bumps into the wall or to maximize the amount of motion, without requiring absolutely strict compliance.

What would be a reasonable real-valued measure of the quality of the *avoid* behavior? Of the *wander* behavior? What about for the combined behavior? Some ideas for *avoid* might be the number of times the robot crashes into the wall per hour, or the number of times it gets within 10 inches of the wall (or it has a short sonar reading), or the length of time it runs before it crashes. For *wander*, it might be the robot's average absolute forward and/or rotational velocities. It also might be some sort of measurement of ground coverage, but let's put that off until later. For now, choose a metric that it can evaluate based on its immediate sensor readings or motor commands.

Are the quality metrics you came up with measurable *by* the robot? Often, they aren't: you might like the robot to minimize its distance to obstacles, for example, but the robot doesn't have direct access to that information.

Come up with a quality metric for the combined behavior that is measurable by the simulated robot. Change the robot brain so that it computes this measure and prints it out.

Now, we'll move to the real robot. First, think about whether the quantities that were measurable by the simulated robot are measurable by the real robot. In the real world, we might have to measure the quality of the robot's behavior using external means (a camera in the ceiling tracking its progress, for example). For this lab, we'll keep using metrics that can be computed by the robot as a function of its sensor readings. In so doing, we should be aware that we are not necessarily getting an accurate measurement of the quantities of interest.

Make any adjustments that seem necessary to your quality metric from the previous section, and try running it on the real robot. What do you find? If you have differences, how much is attributable to actual differences in behavior and how much to the fact that the metric is being computed differently (and is therefore, actually, a different metric)?

Checkpoint: 3:00 PM

- Explain your quality metric for the combined behavior
- Demonstrate that you can compute the quality and print it, with the simulator
- Demonstrate that you can compute the quality and print it, with the robot
- Discuss differences in results between simulator and robot

2.2 Improving behavior

Before lab you experimented informally with different methods of setting the weighting parameter to combine the `avoid` and `wander` utility functions. Now that we have a quality metric for the combined behavior, we can systematically try to set that parameter to maximize the quality of the behavior.

First in simulation, then in the robot, try to find a value for the weighting parameter that gives you the best performance. Come up with a plan for how to do experiments and set the parameter. You might start by running the robot several times with the same parameter value, and seeing what values of the quality metric you get back. How much variation is there? What should you do if there is a lot of variation?[5] This is an optimization problem over a one-dimensional real-valued space. It is made extra difficult in that each individual measurement of the function you are seeking to optimize is noisy. So it's a very hard problem.

You might find that, 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: 4:00 PM

- Explain what procedure you used to pick parameter values
- Show your data and the resulting choices

[5] The discipline of statistics is about, among other things, figuring out how many measurements you need to make of an uncertain quantity in order to make guarantees about it. We won't pursue that in detail here, but it's crucial to understand it eventually.

2.3 Covering ground

If the goal of avoid and wander is to move all the time and not run into things, then oscillating back and forth[6] is a perfectly adequate behavior. There's something sort of unsatisfying about that. Imagine we were making a robot to search for things, or vacuum, or mow the lawn. We'd like it to cover a lot of ground.

One simple approach to making the robot cover more ground is to add another behavior that encourages the robot to continue moving in a direction it has been moving in previously. Try this. You'll have to add some "state" variables that remember some number of previous actions. See the discussion in problem set 1 about how to do this. We'll discuss state a lot more in the next two weeks.

Does adding this behavior keep the robot from being aimless?

Develop a metric that can be measured by the robot for how much ground it covers, and see how well your new program works. Now, with three behaviors, you'll have two weights to tune. (Why two and not three?) You can do it informally this time, if you want to.

Think about additional strategies for covering more ground. If you have time, implement one.

Checkpoint: 4:50 PM

- Demonstrate your new added behavior
- Describe your new metric
- Explain another strategy for covering more ground

3. To do before Sep. 26

Work on the following problems after Thursday's lab and turn in the written parts at lecture on Sep. 26.

3.1. Exercises with the online tutor for Sep. 26

Use the online tutor to complete the problems due for Sep. 26.

3.2. Programming practice: Linear Difference Equations

The Fibonacci sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

is defined by the rule

$$\mathrm{Fib}_n = \mathrm{Fib}_{n-1} + \mathrm{Fib}_{n-2}$$

[6] This kind of behavior sometimes occurs in animals and in people; it's called *perseveration*.

with initial values $\mathrm{Fib}_0 = 0$ and $\mathrm{Fib}_1 = 1$. The Fibonacci relation is a special case of a *kth order linear difference equation*:

$$f[n] = a_1 f[n-1] + a_2 f[n-2] + \cdots + a_k f[n-k]$$

with initial values specified for $f[0]$ through $f[k-1]$.

In lecture, Hal talked about a procedure for computing Fibonacci numbers:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

(a) Write a similar (i.e., tree recursive process) procedure for computing the values of sequences specified by linear difference equations. How you choose to specify the equations and orders and initial conditions is up to you, but try to do something that is "elegant" (whatever you think that means). Use your procedure to compute the first few numbers in the sequence defined by

$$s[n] = 6s[n-1] - 11s[n-2] + 6s[n-3]$$

where $s[0] = 0$, $s[1]$=0, and $s[2] = 2$.

(b) The lecture also showed how one can eliminate the exponential time growth in the Fibonacci procedure by applying the technique of *memoization*:

```
def memoize(f):
    storedResults={}
    def doit(n):
        if storedResults.has_key(n):
            return storedResults[n]
        else:
            value = f(n)
            storedResults[n] = value
            return value
    return doit
```

and then defining

```
def fibComp(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

followed by evaluating [7]

```
fib = memoize(fibComp)
```

Memoize your procedure from part (a) in a similar manner. Demonstrate that it works and that the computations for large n are much faster than without memoization.

Note: You can check your answers by using the fact that $s[n] = 3^n - 2^{n+1} + 1$ (but don't just implement this formula instead of the recursive process). In a couple of weeks, we'll see how to solve difference equations to get closed-form formulas. This gives more insight into the structure of solutions to the equations than just getting the numerical answers.

3.3. Reflections on the lab: Continuous spaces of actions

So far, we've been considering 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 a 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

[7] It's best to run this command in the Python buffer, after defining `fibComp` and `memoize`. Otherwise, it's easy to get confused by the order in which Emacs sends expressions to the Python interpreter.

[8] At least, not in any technology known in 2006!

> 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.

This is an open-ended problem and you can take it as far as you wish.

To pass: At a minimum you should give new definitions for `avoid` and `wander`. For each behavior, sketch the rectangle of values (x, y), indicating the approximate value that the behavior should assign to that pair and write a couple of words explaining your choice. Remember that for `avoid`, at least, the value will depend on the sensor readings as well as on x and y (and you may want to make wander sensor-dependent as well: it's up to you).

Implement each behavior as a procedure that takes the list `[x,y]` as input and returns the values indicated in your sketch.

To get a good grade: You should also implement a simple maximization algorithm that scans with a coarse grid, and demonstrate that this program now works; i.e., it chooses the grid point with the maximum utility.

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.)

Exploration 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. If you manage to produce a sophisticated working program, that's terrific. But it's also OK to only write up your design for the new program, so long as your writing is clear and careful.

3.4 Post-lab Reflection

Think about what would be required to write a program to select the parameter that chooses the relative weights of behaviors automatically. A simple strategy would be to discretize the parameter space at fixed intervals, try each of those values some number of times, and pick the best. Can you think of more sophisticated strategies?

Of course, if you think about it, there are lots of other parameters hidden inside your behaviors. You could consider adjusting them, as well, in order to improve the overall score of the algorithm.

But the bigger the parameter space over which you are trying to optimize, the (much!) harder it becomes. Just think about the discrete sampling approach. If you tried 10 values for each parameter and had n parameters, how many values would you have to try in total? What would the Θ order of growth of this function be?

Exploration: Implement an algorithm, in the simulator, for automatically choosing a parameter value. One standard approach in the literature is *response surface methodology*.[9] [10]

What to turn in

For this homework assignment, in addition to the tutor problems, you should turn in

1. Your code and some test cases for computing solutions to difference equations.
2. The definitions for `wander` and `avoid` in a continuous spaces of action. They don't need to run as code; describe the idea, giving an explanation in English of what they're trying to do.
3. Anything you'd like to include on automatic parameter tuning. (Not required.)

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.[11]

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 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.

[9] Here is an introduction: `http://www.mne.psu.edu/me82/Learning/RSM/rsm.html`.

[10] Another, more sophisticated method is described here (the citations in this paper might also be useful): *Memory-Based Stochastic Optimization* Andrew Moore and Jeff Schneider, *Neural Information Processing Systems 8*, 1996. You can find it online (and stand on the shoulders of giants!) using `http://scholar.google.com`.

[11] 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.

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[12]

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

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

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.

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, parameter passing, and so on—can be used directly to support means of combination and abstraction.
- *Higher-order procedures* can express computations in terms of common general patterns, like memoization.
- 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 preferences. Given the mathematical model, we can apply general techniques such as *optimization.*
- 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.