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

Assignment 2

Issued: Tuesday, Sept. 11

This handout contains:

- Software Lab for Tuesday, September 11
- Pre-Lab exercises to do before Thursday, September 13 at 10AM; you can come and do them in lab on Wednesday, September 12. Don't forget that Part 2.2 of the on-line Tutor problems are also due before Thursday lab.
- Robot Lab for Thursday, September 13 (read the lab before coming to 34-501)
- Post-lab writeup and exercises due Tuesday, September 18 at 10AM. Don't forget that Part 2.3 and 2.4 of the on-line Tutor problems are also due before Tuesday lecture.

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, the post-class software lab, and the on-line problems, cover Python's support for *functional programming*. Thursday's lab applies higher-order procedures to the tasks of specifying robot *behaviors* with the aid of *utility functions*.

Make sure you do:
`athrun 6.01 update`
so that you can get the `Desktop/6.01/lab2` directory which has the files mentioned in this handout.

Tuesday Software Lab: Higher-order procedures

At the end of this lab, go to the on-line Tutor at `http://sicp.csail.mit.edu/6.01/fall07`, choose PS2, and paste all your code, including your test cases, into the box provided in the "Software Lab" problem.

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 procedures as inputs and/or returns procedures as results. We will be working with a simple model of probability spaces, one that we will use later in the course.

Probability

Probability theory is a calculus that allows us to assign numerical assessments of uncertainty to possible events, and then do calculations with the numerical assessments in a way that preserves their meaning. (A similar system that you might be more familiar with is algebra: you start with some facts that you know, and the axioms of algebra allow certain manipulations that will preserve truth.)

We'll just consider worlds that can be represented by the values of a small number of discrete variables. We'll let U be the *universe* (or sample space), which is a set of *atomic events*. An atomic event is just an outcome or a way the world could be. The universe associated with rolling a normal die once can be written as

$$U = \{1, 2, 3, 4, 5, 6\} \ .$$

The universe associated with rolling two dice, or one die twice, can be written as

$$U = \{(1,1), (1,2), ..., (1,6), ..., (6,5), (6,6)\} \ .$$

Atomic events could indicate which room a robot is in and whether the battery is charged, or any description of the world under consideration, but we are assuming that there are a finite number of possible combinations of values, so that the universe is finite.

An *event* is a subset of U. For example the set of atomic events in which the single die roll result is greater than 3 is an event. A probability *measure* P is a mapping from events to numbers that satisfy the following axioms:

$$\begin{aligned} P(U) &= 1 \\ P(\{\}) &= 0 \\ P(A \cup B) &= P(A) + P(B) - P(A \cap B) \end{aligned}$$

Or, in English:

- The probability that something will happen is 1.
- The probability that nothing will happen is 0.
- The probability that an atomic event in the set A or an atomic event in the set B will happen is the probability that an atomic event of A will happen plus the probability that an atomic event of B will happen, minus the probability that an atomic event that is in both A and B will happen (because those events effectively got counted twice in the sum of $P(A)$ and $P(B)$).

Armed with these axioms, we are prepared to do anything that can be done with discrete probability!

For example, consider the universe associated with rolling two fair dice (shown above); each of the atomic events is equiprobable. So, each event has probability 1/36.

- The probability of rolling a pair of twos is: $P(Die1 = 2 \cap Die2 = 2) = 1/36$
- The probability that the first die is a 2 is: $P(Die1 = 2) = 1/6$, the sum of 6 atomic events.
- The probability of one of the dice coming up 2 is

  $$P(Die1 = 2 \cup Die2 = 2) = P(Die1 = 2) + P(Die2 = 2) - P(Die1 \cap Die2) = 1/6 + 1/6 - 1/36 \ ,$$

  note that the $(2, 2)$ atomic event is included in both events "Die1 = 2" and "Die2 = 2".

One more important idea is *conditional probability*, where we ask the probability of some event E_1, assuming that some other event E_2 is true; we do this by restricting our attention to the part of the sample space (universe) in E_2. The conditional probability is the amount of the sample space that is both in E_1 and E_2, divided by the amount in E_2:

$$P(E_1|E_2) = \frac{P(E_1 \cap E_2)}{P(E_2)} \ .$$

That is, the fraction of the E_2 probability that is also in E_1.

The expression $P(E_1|E_2)$ is read as: the conditional probability of E_1 **given** E_2 or, more loosely, the probability of E_1 given E_2. The expression $P(E_1 \cap E_2)$ is read as: the probability of E_1 **and** E_2.

Manipulating Probability Spaces

We're going to build Python models of discrete *probability spaces*, that is, sample spaces with corresponding probability measures. Concretely, we'll represent a probability space as a list of atomic events, each with its assigned probability. Each element of the probability space, which we will call a *sample* (sometimes also known as an outcome) will be a list with two elements: a probability and the value that defines an atomic event. For example, to represent the probability space for a standard, six-sided die we can have:

```
dieSpace = [ [1.0/6, 1], [1.0/6, 2], [1.0/6, 3],
             [1.0/6, 4], [1.0/6, 5], [1.0/6, 6] ]
```

The atomic events don't have to be equiprobable. We can also model a loaded die:

```
loadedDieSpace = [ [1.0/10, 1], [1.0/10, 2], [1.0/10, 3],
                   [1.0/10, 4], [1.0/10, 5], [1.0/2, 6] ]
```

It is essential that the probabilities of all of the samples add up to 1.0, as required by the axioms.

The value used to define an atomic sample could be more complicated, e.g. another list. For example, if we wanted to represent the probability space for the roll of two fair dice, we could have:

```
diceSpace = [ [1.0/36, [1, 1]], [1.0/36, [1, 2]], [1.0/36, [1, 3]],
              ..., [1.0/36, [6, 6]] ]
```

Think about what the probability space for two loaded dice would be.

We can start our coding by defining the *selectors* for a single sample to let us refer to the components of a single sample more mnemonically and to let other code become independent of our implementation choice for samples, that is, that they are lists. The procedure `sampleP` takes a single sample and returns the probability of that sample:

```
def sampleP(s): return s[0]
```

The procedure `sampleV` takes a single sample and returns the value associated with that sample:

```
def sampleV(s): return s[1]
```

Note that it does not make any sense to apply these procedures to the sample space as a whole.

An event is a subset of the probability space, for example, that the die roll is greater than 4. That is, an event is the subset of the samples that satisfy some condition. The probability of an event is simply the sum of the probabilities of the atomic samples that satisfy the event's condition.

You can find some useful definitions in the file `prob.py` from the Week 2 code download. In particular, there are definitions for the probability spaces for one and two fair dice.

Question 1. Write a short Python function called `eventCreate` that takes two arguments: a test (a function) and a probability space (a list). The test is a function on a single **sample** in the probability space and returns a boolean. An example of a test, which tests that the first die in a two-dice sample space has value 3, is:

```
def dice1Is3 (sample): return sampleV(sample)[0] == 3
```

`eventCreate` returns a list, which contains the subset of the probability space that satisfies the test. You should use a list comprehension to implement `eventCreate`.

Question 2. Write a short Python function called `prob` that computes the probability of an event. Its single input is a list, typically produced by a call to `eventCreate`; its output is a number.

Question 3. Write a short Python function called `cprob` that computes the conditional probability of a first event argument given a second event argument. Both event arguments are lists. Assume that the events are both subsets of the same probability space, since otherwise this does not make sense.

Question 4. Test out your functions on the dice examples. Concretely: for each of these programs, come up with two test cases. Compute the answers by hand. Then show that your Python functions gets the same answers. Check robustness by trying some extreme cases, such as events that are empty. Some example tests: the probability that the sum of two dice is 5, the probability that the sum of two dice is 5 and the max value is 3, the conditional probability that the sum of two dice is 5, given that the max value is 3 (why the difference between these last two?).

An alternative representation for events

Let's investigate an alternate representation for events. Instead of computing the subset of the probability space when we create the event list, we'll be lazy and simply remember the test and

the space. So an event will be represented as a list of a test function and a probability space. The following functions generate the new representation of an event and provide implementation-independent access to the event components:

```
def eventCreateI (test, space): return [test, space]

def eventTest(event): return event[0]

def eventSpace(event): return event[1]
```

Question 5. Write a short Python function called `probI` that computes the probability of an event in this new representation.

Question 6. Write a short Python function called `eventAnd` that computes a new event from two other events which share the same probability space, by building a new test function that checks that both tests are satisfied. Make sure that it returns an event in the new representation.

Question 7. Write a short Python function called `cprobI` that computes the conditional probability of a first event argument given a second event argument. You should be able to do this mostly by using the `eventAnd` and `probI` functions.

Question 8. Test these programs on the test cases you used before.

Question 9. What are the advantages and disadvantages of these two representations for events?

If you have time, read a description of the Monty Hall problem in `http://courses.csail.mit.edu/6.042/fall06/lec19.pdf`. Write a procedure that generates the probability space and computes the probability of winning if you switch your initial choice of door. There is a problem on the on-line Tutor on Monty Hall (due next Tuesday - Sept 18), so be sure to save your work.

Go to the on-line Tutor at `http://sicp.csail.mit.edu/6.01/fall07`, choose PS2, and paste all your code, including your test cases, into the box provided in the "Software Lab" problem. Do this even if you have not finished everything. Your completed answers to these questions are to be handed in with the rest of your writeup for the Thursday lab.

Homework and preparation for Thursday lab

Here's what you need to do before the start of lab on Thursday:

1. Read the lecture notes.
2. Do the on-line Tutor problems for week 2 that are due on Thursday (Part 2.2).
3. Read the background for the lab and do the preparation questions (Questions 10 – 13). Note that this includes writing some code and trying it with the simulator.
4. Finally, read through the entire description of Thursday's lab so that you'll be prepared to work on it on Thursday.

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*, which are 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 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 with each 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 associated utility functions. 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, then selecting and executing 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. Having each behavior produce a utility function, rather than a single action, enables us to make trade-offs between their preferences, perhaps selecting an action that is the second choice of both behaviors, but the most satisfactory compromise.

The utility function paradigm is an instance of the *transducer* model of programming: the robot 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 from 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 two things: a string that describes the action (for debugging), and a list of arguments for `motorOutput`: a forward velocity and a turning velocity.

```
speed = 0.1
stop = ["stop", [0, 0]]
go = ["go", [speed, 0]]          # forward velocity = speed (in meters/s)
left = ["left", [0, speed]]      # rotational velocity = speed (in radians/s)
right = ["right", [0, -speed]]
```

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

```
def actionArgs(action):
   return action[1]
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(actionArgs(action)[0], actionArgs(action)[1])
    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 structures provided 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 when we use the command

```
doAction(left)
```

we think "do the action `left`" and we don't think "call `motorOutput` with the elements of the second element of the action 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. Why use a function and not simply a list of utility values for each action? If we have only four actions, then a list makes good sense. However, what if we had a very large range of actions, for example, any combination of forward and rotational velocity? A utility function would still make sense but a list would not; see the section on "Continuous spaces of actions" at the end of this handout.

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 can be any sensory data available (such as the sonar range values or the robot's pose), as well as parameters that modify the behavior (perhaps specifying whether it should move fast or slow).

Here is a simple primitive behavior that makes the robot wander around (stupidly). It ignores the pose values and sonar readings and simply prefers going forward (utility 10) to turning left or right (utility 2), and never wants to stop (utility 0):[1]

```
# Primitive wandering behavior
def wander(poseValues, 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
```

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

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

```
wanderUtility = wander(poseValues, 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 example, for action `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

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

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. It roughly tries to move or turn the robot so that it doesn't get too close to any obstacles.

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

```
# Primitive behavior for avoiding obstacles
def avoid(poseValues, rangeValues):
    # Parameters for clip
    mindist = 0.3
    maxdist = 1.2
    # clip a given value between mindist and maxdist and scale from 0 to 1
    def clip(value, mindist, maxdist):
        return (max(mindist, min(value, maxdist)) - mindist)/(maxdist - mindist)

    # Stopping is not that useful.
    stopU = 0

    # Going forward away from obstacles, is useful when there are nearby
    # obstacles.  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 shortest observed
    # distance, and scale the result between 0 and 10
    minAnyDist = min(rangeValues)
    minFrontDist =  min(rangeValues[2:6])
    if minAnyDist <= maxdist/3:
        goU = clip(minFrontDist, mindist, maxdist) * 10
    else:
        goU = 0

    # 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.
    minLeftDist = min(rangeValues[0:3])
```

```
    minRightDist = min(rangeValues[5:8])
    closerToLeft = minLeftDist < minRightDist
    if closerToLeft:
        rightU = 10 - clip(minLeftDist,mindist, maxdist/3) * 10
        leftU = 0
    else:
        leftU = 10 - clip(minRightDist, mindist, maxdist/3) * 10
        rightU = 0

    # 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 move until it's not too close to any walls, and then just sit.

Note that the utility function returned by a call to `avoid` is a function of the rangeValues it was called for. On the next call to the brain's `step` we need to call `avoid` again with the new rangeValues to get a new utility function.

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:

```
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 sensors
rangeValues = sonarDistances()
poseValues = pose()
# use those sensor values to get utility functions for wandering and avoiding
wanderUf = wander(poseValues, rangeValues)
avoidUf = avoid(poseValues, rangeValues)
# add the resulting utility functions
combinedUf = addUf(wanderUf, avoidUf)
# evaluate the combination on the stop action
value = combinedUf(stop)
```

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

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()
    poseValues = pose()
    # get the utility function just for wandering
    u = wander(poseValues, rangeValues)
    # uncomment this line to combine wander and avoid
    # u = addUf(wander(poseValues, rangeValues), avoid(poseValues, 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)
    doAction(action)
```

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

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.

Weighted sums of behaviors: Due before lab on Sept. 13

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 `utilityBrain.py`. Start up Idle and SoaR choosing the simulator and the Tutorial world. Load `utilityBrain.py` as the brain. Also open `utilityBrain.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.

Question 10. Run the robot in the simulator. 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. Why?

Question 11. Edit the code for the `step` procedure 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 (in the simulator).

Question 12. With the sum of the two utility functions, the robot's behavior is governed both by wandering and by avoiding. One could 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(poseValues,rangeValues),
          scaleUf(avoid(poseValues,rangeValues),2))
```

Question 13. Try different values of the scale factor to see how they compare. What happens when more weight is given to `avoid`? How about when more weight is given to `wander`?

Make sure that your code is available on the Web so that you can get it when you come into lab and load it on your robot's laptop.
Demonstrate your code to your LA on the simulator.

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? Warning: robots in close proximity may interfere with each other's sonars.

New behaviors

Invent and implement one or more new primitive behaviors. When you implement your behavior(s), remember that a behavior in our system is a procedure that takes the pose values in `poseValues` and 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 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 variable called `robot.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`.[2]
  2. "Persistent" is a behavior whose utility function returns 10 if the action is the same as `robot.actionLastRound` and returns zero otherwise.
  3. Modify the brain's step operation, so that at the very end, after it chooses the action to do, it saves the action as `robot.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 uniform 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.
- *Move to goal:* This one is tricky. We'll use it in the next lab, though, so if you're feeling bold, give it a try! (If not, you can use our code later on.) If you give the behavior a target pose, you should be able to implement a behavior that will try to drive the robot toward that pose.

  To simplify debugging in the simulator, we recommend adding this definition to your file

  ```
  def cheatPose():
      return app().output.abspose.get()
  ```

  and using `cheatPose()` instead of `pose()` to determine where the robot is. If you do this, then when you drag the robot around the simulator window with the mouse, you will magically know the true pose. (On the real robot, of course, if you were to pick it up and put it down, it would have no idea that it had moved.)

[2] When we want a variable to maintain its value across successive calls to the `step` function, we need to prefix it with `robot.` . The reasons for this will become clearer next week.

Question 14. Describe two of your new primitive behaviors and demonstrate them both on the simulator and with the real robot. Be sure that you debug them on the simulator before attempting to use the robot.

Question 15. 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(poseValues,rangeValues), scared(poseValues,rangeValues))
```

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

Checkpoint: 11:30 AM

Demonstrate your programs running on the real robots to your LA. Be ready to explain how they work and why you made them the way you did.

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 total weight.)
- *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.
- *Conditional choice:* This takes two utility functions and a *test* procedure (which might do 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.

  One interesting application of this might be to implement the behavior of going to a goal, which can be thought of as taking place in phases.

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 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(poseValues,rangeValues),
              persistent(poseValues,rangeValues)),
      scaleUf(randomChoice(friendly(poseValues,rangevalues),
                           scared(poseValues,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.

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).

Question 16. Demonstrate two means of combination and the resulting behaviors in your repertoire, both on the simulator and the real robot, illustrating various compositions.

Question 17. Describe some of the interesting things you've observed, both expected and unexpected.

Checkpoint: 12:30 PM

Demonstrate your programs running on the real robots to your LA. Be ready to explain how they work and why you made them the way you did.

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.

- 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.
- 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.

Post-Lab Writeup and Exercises: Due before lecture on Sept. 18

All post-lab hand-ins should be written in clear English sentences and paragraphs. We expect at least a couple of sentences or a paragraph in answer to each of the numbered questions in this lab handout (including the software lab).

We also want the code you wrote. 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.

We also expect at least a couple of sentences or a paragraph in answer to the reflection questions below. We're in interested in an explanation of your thinking, as well as the answer.

Reflection on lab results

Question 18. Critique the use of utility functions as a mechanism for modularity and abstraction in defining behaviors. To what extent can the behaviors be designed independently?

Question 19. Can you suggest an alternative approach to defining behaviors for the robots that might have advantages? Compare and contrast your proposal and the utility-based approach we explored in this lab.

The following sections are not required. They're things that you can work on if you're interested and/or have extra time in lab. If you hand in a page or two describing some work you did on this, you are eligible for up to an extra half a point on the lab.

Exploration: 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 measure 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 (using the simulator). 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.

Here are some things to describe in your writeup:

- What's your metric and how did 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?
- Show your data and the resulting parameter choices.

Exploration: Continuous spaces of actions

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(fv,rv)` with $-L \leq fv \leq L$ and $-R \leq rv \leq R$, where L and R are some maximum translation and rotation speeds. This suggests extending our finite set of actions to allow any `motorOutput(fv,rv)` with fv and rv in

the designated range. This is a *continuous space of actions*: an action for any point in the rectangle $\{(fv, rv) | -L \leq fv \leq L, -R \leq rv \leq R\}$.

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 (fv, rv). So we may as well change the representation of actions so that an action is just the list $[fv, rv]$. The procedure `useUf` in the robot brain's `step` could then be

```
def useUf(u):
    [fv,rv]=bestAction(u)
    print "Best fv,rv: ", [fv,rv]
    motorOutput(fv,rv)
```

(b) A utility function will now be represented as a procedure that takes as arguments a list $[fv, rv]$ and returns a number (the utility). The number will be -1 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 (fv, rv) 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 (fv, rv) and pick the one with the highest utility. What we have here is a *two-dimensional maximization problem*: find the pair (fv, rv) in the rectangle such that $u(fv, rv)$ is maximized, where u is the utility function.

One way to find the maximum is to subdivide the rectangle into a grid (by subdividing the intervals for fv and rv), 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.

Here are some things to do:

- Define wander and avoid utility functions over the continuous space.
- Implement a grid-based `bestAction`.

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.