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 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. Different behaviors will have different utility functions. A behavior where the robot tries to avoid obstacles might value turning more than going forward, while a wandering behavior might value going forward more than turning. With the utility function paradigm, a robot following a 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.

Debuggable Brains

This week we’re going to start using an extension of the usual PyRo brain class, called C1Brain. In addition to doing everything a regular brain does, it sets things up to make it easier to debug your robot programs.

Debugging is a deductive process, which applies when you have a program (or a circuit or a mechanical contraption) that doesn’t do what you want it to do. You systematically try to figure out what’s going wrong, localizing the problem to smaller and smaller parts of the code, until you can see exactly what’s wrong. This is a really important skill to master, but most people find it difficult until they’ve had practice at it.
Simple computer programs, all by themselves, are deterministic.¹ That means that you can run them again and again, and they will always go through the same steps and get the same answer. This makes debugging reasonably straightforward, because you know that once you see how one section of the program is behaving on a particular input, you can rely on that, and then go looking at a different part. However, when you change the input to a program, that might cause it to do all sorts of different things.

Running a robot is fundamentally non-deterministic. Each time you run the robot, it gets a slightly (or vastly) different stream of sensor readings, and hence generates a different stream of motor commands. This makes debugging extra hard, because you can’t even hold the input constant as you search through the program for the bug. To help with this, we’ve designed an infrastructure that allows you to:

- **Log** the stream of sensor readings your robot gets while it’s running
- **Replay** those readings into your program; that is, you can run your robot program as if it were running on the robot, but instead of getting sensor readings from the actual sensors, it will get them from the log.

This way, you can debug your program in replay mode, which will make it deterministic. You can run it again and again, printing and looking at different parts of it, and it will always behave the same way. Of course, as soon as you want to change the way your program generates motor commands, you should run it on the robot and gather a new log. Because it doesn’t make sense to actually move the robot when it is essentially hallucinating its input, the motor commands are ignored when you are in replay mode (think of this as a robot dreams!).

Another reason why logging is a good idea is that printing is slow. If you print out a lot of stuff while your program is running, it will either (or both) slow down the rate at which the step function is called, which will change the dynamics of the interaction between the robot and the world, or cause the stuff being printed on the output window to get behind, and not reflect what is currently happening on the robot (which is very confusing). So we recommend that you not print things out when the robot is running. Instead, gather a log, and then print as much as you want while running with the log.

The simples brain program looks essentially the same as before:

```python
from C1Brain import C1Brain

class SomeBrain(C1Brain):
    def setup(self):
        print "All systems are go!"

    def step(self):
        self.readSensors()
        self.motorOutput(0,0)

    def INIT(engine):
        return SomeBrain('SomeBrain',engine)

¹Provided they don’t use random numbers, or keyboard inputs or other external data.
Note that the class C1Brain replaces Brain from the basic version we introduced last week. The first thing you should do inside your step function is make a call to `self.readSensors()`, and the last thing is to make a call to `self.motorOutput(L,R)`, with some computing in between. (Our simplest program doesn’t do any computing in between and just sends 0 turning to the motors.) The overall `step` should have a single call to `readSensors` (at the beginning) and a single call to `motorOutput` (at the end).

The procedure `self.readSensors()` grabs a snapshot of readings from the sonar array and the odometry for later examination. You can access the sonar measurements from the list `self.sonarDistances`, with units of meters by default. The expression `self.sonarDistances[n]` is like `self.robot.range[n].distance(unit="METERS")`; but the former represents a single fixed set of readings taken at the time `self.readSensors()` was called, while the latter changes even within the same call to the `step` function. Similarly, you can access the odometry readings in the list `self.odometry`, which contains three values representing the $x$, $y$, and $\theta$ of the robot in the global frame.

The procedure `self.motorOutput(T,R)` sets the translation and rotation speeds, in the same way as does `self.robot.move(T,R)`, but it also supports logging and lets you set some operating parameters. For instance, in the setup routine, you can optionally set the following variables to different values:

- `self.maxTransSpeed`: maximum allowed translation speed, as a fraction of what the robot is capable of; if you send a command `self.motorOutput(T,R)` with $|T| > self.maxTransSpeed$, it will be thresholded. Must have $0 \leq self.maxTransSpeed \leq 1$. The default is 0.4.

- `self.maxRotSpeed`: Same thing for rotation speed. The default value is 1.

- `self.logfile`: A string that specifies the name of the file to use as the log. The default is "logfile".

- `self.readLog`: If 0, `readSensors` gets values from the robot, and `motorOutput` sends the robot the commands you pass it. If 1, `readSensors` gets values from the file, and `motorOutput` sends the robot commands from the file—that is, it replays exactly what happened the last time you ran the program with `writeLog=1`. Use this with print statements and `Step` rather than `Run` to debug. The default is 0.

- `self.writeLog`: If 0, no effect. If 1, writes to `self.logfile` all sensor readings and motor commands. `readLog` and `writeLog` cannot both be set to 1. The default is 0.

Here is a brain that writes a log.

```python
from C1Brain import C1Brain

class SomeBrain(C1Brain):
    def setup(self):
        self.logfile = "mylogfile.log"
        self.writeLog = 1
    def step(self):
        self.readSensors()
        self.motorOutput(0,0)
    def INIT(engine):
        return SomeBrain('SomeBrain',engine)
```
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:

```python
def stop(self): self.motorOutput(0,0)
def go(self): self.motorOutput(1,0)
def left(self): self.motorOutput(0,1)
def right(self): self.MotorOutput(0,-1)
```

Now we’ll define a list of available actions. Note that this is a list of Python procedures. Since procedures are first-class objects in Python, we can include them as elements in lists and other data structures.

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

We’ll also include a way to produce a string that identifies the action for use in printing messages. Here we’ve used the `func_name` attribute, that Python provides for procedures:

```python
def actionString(action):
    return action.func_name
```

Behaviors and utility functions

A behavior is represented in our Python implementation as a procedure that returns a utility function. The utility function is itself a procedure that takes an action as input and returns a number that indicates how much that behavior “prefers” that action. The procedure returns 0 if the input is something it doesn’t recognize.

Here is a primitive behavior that just has the robot wander around. It prefers going forward to turning left or right. Given our representation of behaviors, `wander` is a procedure of no arguments, which returns the corresponding utility function procedure.

```python
# Primitive wandering behavior
def wander():
    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 a more complicated primitive behavior, where the robot attempts to avoid obstacles. The avoid procedure takes as input a list of the distances reported by the sonars and returns the 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 fine, 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. For wander, the robot will always choose go, which will pin it up against a wall with its wheels turning. And with avoid, the robot will always choose stop!\(^2\)

**Combining utilities**

In order to get something better, 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:

```python
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:\(^3\)

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

**Picking the action**

Finally, we interface behaviors to the Pyrobot system by defining a robot brain. For Pyrobot, the brain needs a `step` method that tells the robot what 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).

\(^2\)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 below about how `bestAction` chooses the highest utility action.

\(^3\)In Python, `lambda` can be used only for a single expression. So 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. The seems (to Lisp programmers) to be an arbitrary restriction imposed by Python.
class SimpleBrain(C1Brain):
    def step(self):
        # Do an indicated action
        def doAction(action):
            if action in allActions:
                action(self.robot)
            else:
                print "error, unknown action", action

        # 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)

        self.readSensors()
        # Here we'll use the sum of wandering and avoiding. Later you can
        # change u to include other behaviors that you define
        u = addUf(wander(), avoid(self.sonarDistances))
        useUf(u)

We’ve defined three subprocedures to (hopefully) make the code more readable. The `doAction` procedure simply invokes an action by calling that procedure with the brain’s robot. 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.4

1. Homework in preparation for lab on Feb. 16

This section includes problems to complete with the online tutor. These are due before class on Thursday, Feb. 16. It also includes a Pyro programming problem to do with the robot simulator, as preparation for class. You should complete this before Thursday’s lab, and mail them to yourself or put them on your Athena account so you’ll have access to them in lab.

---

4If 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. And 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).
1.0 Read this whole document!

Please read the entire section on the lab before coming to lab.

1.1. Exercises with the online tutor for Feb. 16

Use the online tutor to complete the problems due for Feb. 16.

1.2. Programming with the simulator—Weighted sums of behaviors

Start by downloading the files C1Brain.py and utility.py from the course web site. Be sure to put them in the same directory (it doesn’t matter which one) on your computer. This is the code explained above. Start up Pyro with the Pyrobot simulator and either the Tutorial world or the LongHall world—whichever you like. Use the Pyrobot60000 robot and load utility.py as the brain. Also set up utility.py for editing by loading it into Emacs so you can easily read it, edit it, and then reload brain in Pyro to try your edits.

Run the robot. The utility function here, as explained above, is the sum of the utilities for wander and avoid. You should find that it works pretty well, although (in the LongHall world) you might see the robot get stuck when it runs into a cul de sac. 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 of u for that action.

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

$$u = \text{addUf}(\text{wander}(), \text{scaleUf}(\text{avoid}(\text{rangeValues}),2))$$

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 account so that you can get it when you come into lab and put it on your robot’s laptop.

2. To do in lab on Feb. 16

Today’s lab (and every Thursday lab) will begin with a mini-quiz that is based on the homework due for today, and on material covered in previous weeks.

After the quiz, you should work as part of a group with a single computer and robot.
2.0 Getting started

Start by loading your scaled sum of behaviors program onto your robot’s laptop. If you put your code on your athena account in the directory `dir` and file `myfile.py` then you can do:

```
[robot@eecs]$$ scp username@athena.dialup.mit.edu:dir/myfile.py myfile.py
```

Now, run your code on the robot and see how it behaves differently from the simulator. While you’re running it, gather a log. Replay your program using the log. Print out the sonar values or the actions taken.

Checkpoint: 1:45 PM

- Show your code running from a log that you gathered on the robot.

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.

The first step is to formulate quality metrics. 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 doesn’t require the robot to have any state, 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 Feb. 21

Work on the following problems after Thursday’s lab and turn in the written parts by Feb. 21 at Hal Abelson’s office (32-392) by 5PM. Remember that there is no lecture on Feb. 21 (MIT virtual Monday). Class on Thursday Feb 23 will begin with a special lecture at 1PM in room 34-401A, before we all go to the lab.

3.1. Exercises with the online tutor for Feb. 21

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

3.2. Programming practice: Linear Difference Equations

The Fibonacci sequence

\[
0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots
\]

\(^6\)This kind of behavior sometimes occurs in animals and in people; it’s called *perseveration.*
is defined by the rule

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

with initial values $\text{Fib}_0 = 0$ and $\text{Fib}_1 = 1$. The Fibonacci relation is a special case of an $k$th 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:

```python
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 different 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 apply memoization to eliminate the exponential time growth in the Fibonacci procedure:

```python
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
```

followed by evaluating

```python
fib=memoize(fib)
```

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.

---

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

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 $\text{motorOutput}(x, y)$ with $-L \leq x \leq L$ and $-R \leq x \leq R$, where $L$ is $\text{maxTransSpeed}$ and $R$ is $\text{maxRotSpeed}$. This suggests extending our finite set of actions to allow any $\text{motorOutput}(x, y)$ with $x$ and $y$ in the designated range. This is a continuous space of actions: an action for any point in the rectangle $\{(x, y) \mid -L \leq x \leq L, -R \leq y \leq R\}$.

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) We’ll need to change the representation for primitive actions. The program currently uses a list of procedures of one argument (the robot). In the new implementation, we’ll have an action be represented as a list $[x, y]$ of the $x$ and $y$ arguments to $\text{motorOutput}$. The procedure $\text{useUf}$ in the robot brain’s $\text{step}$ could then be

```python
def useUf(u):
    [x, y] = bestAction(u)
    print "Best xy: ", [x, y]
    self.robot.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—$\text{addUf}$, $\text{scaleUf}$ and any others you’ve defined—don’t need to change at all.

(c) The big change in the program will be in $\text{bestAction}$. 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 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.

\(^8\) At least, not in any technology known in 2006!
What to turn in

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 better 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.

To get a top grade: You should also do some research on maximization algorithms. You can write up a paragraph or two on what you find—you need not write any code.

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, 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 \(\Theta\) 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.\(^9\)\(^10\)

---

\(^9\)Here is an introduction: http://www.mne.psu.edu/me82/Learning/RSN/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.
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.\footnote{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.}

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.

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

- 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

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.

- 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.

---

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