MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Assignment for Week 1, Issued: Tuesday, Feb. 5

There will be three handouts issued in 6.01 each week:

1. *Weekly assignment:* This includes work to do in the software lab and the design lab, as well as a two-part homework—one part due before the design lab, one part due before the following week's lecture.
2. *Weekly class notes:* There is no course textbook. Instead, we'll distribute notes that cover the course material as we progress though it week by week.
3. *Weekly lecture handout:* This is to help you follow along in lecture and to review afterwards.

All three handouts will be distributed in lecture and also available online from the 6.01 web site `mit.edu/6.01`, linked from the course calendar. You should bring all three handouts with you to the software and design labs.

Overview of this week's work

Each weekly assignment has several parts. Here's an outline of the work for this week:

. . . to do in software lab

In software lab, you'll work individually with the help of an LA. You can use your laptop if you want to, or you can use one of our machines.

1. You'll write some simple procedures that incorporate examples of polynomial evaluation presented in lecture. At the end of the lab period, you'll need to submit your code, together with test cases that demonstrate that it works, by pasting it into the box provided in the "Software Lab" problem in the on-line tutor.
2. There's also a *software lab exploration* that you can work on if you finish the mandatory part of the lab. Any individual week's exploration is optional, but you'll need to do some of the explorations during the semester to get a high grade. See the course grading policy for details.

. . . homework to do before the start of design lab

1. Read the class notes and review the lecture handout.
2. Do the first half of the on-line tutor problems for week 1 (the part that is due before design lab).
3. Read the entire description of the design lab, so that you'll will be ready to work on it when you get to lab.

. . . to do in design lab

Each design lab will begin with a *nanoquiz.* The quiz may include material from the class notes, from the on-line tutor problems due that day, or from the design lab description. The quiz will be given in the first 15 minutes of lab, and if you miss it, we will not give you time to make it up before starting the lab, so please don't be late.

After the quiz, you'll work with a partner. This week's lab will focus on getting acquainted with the robot (both physical and simulated) and writing some simple *robot brains.* Work through the lab with your partner, answer the questions, and demonstrate the results to your LA.

. . . homework due before the start of your software lab, on Feb 12 or 13

1. This part of the homework will include some additional problems to do using the on-line tutor.
2. There is also a writeup where you provide code and written answers. These should be printed out and handed in at the beginning of your software lab.

   Important: Please read the "guidelines for written homework" available on the course web site. Homework that does not follow these guidelines *will not be graded* and you will get no credit for it.
3. Similar to the software lab, there are *design lab explorations* to do (optionally) to be eligible for a top grade.

. . . Optional exploration: Due at the start of your section, on Feb 19 or 20

Software Lab for Week 1

Getting started

For this software lab, you can use one of the lab laptops, an Athena terminal, or your own laptop. But don't use your own laptop unless you've previously loaded the IDLE and Python and gotten it running: setting up your machine isn't a good use of lab time. You can get help with software installation from the course staff after you complete the lab.

On lab laptops or Athena terminals, you should start by executing

```
athrun 6.01 update
```

in a Unix terminal window. This creates the `Desktop/6.01/lab1` directory that will contain the code files mentioned in this handout. These are no files for today's software lab, but after today—including this week's design lab—there will always be code to load.

If you are running from an Athena terminal, you should *also* execute

```
add -f 6.01
```

to get the appropriate versions of Idle and Python.

If you are using your own laptop, you'll generally download the lab code files from the course Web site, linked from the Calendar page. There's nothing to download for today's software lab.

Starting and using IDLE

IDLE is an environment for editing and running Python programs. To start it you can type `idle` into a terminal window. There may also be a launcher you can click, depending on how your machine is set up.

The Python Shell Starting IDLE brings up a window called "Python Shell." The shell acts a bit like a calculator. You type expressions and press enter, and Python evaluates them and prints the result. So, if you type

```
>>> 4 + 4
```

and press enter, Python will print `8`. You do *not* have to explicitly type "print": Python evaluates what you type and prints the value automatically. Play with the shell a little.

Editor window You'll want to use the shell window to test things by typing short expressions, but you won't use the shell window to write your programs. If you want to start defining procedures, you should open a new *editor window* (choose `New Window` from the `File` menu) and write your definitions there. To practice, open an editor window and type in these definitions:

```
a = 'hi'
b = 7
def f(x):
    return x + 1
```

Now save the contents in a file and choose `Run Module` from IDLE's `Run` menu. If there was something obviously syntactically wrong with your file (parentheses not closed, or wrong indentation, for example), IDLE will tell you about it and highlight the problematic point in your file. Otherwise, the shell will print something like

```
>>> ======================= RESTART ========================
>>>
```

Now you can use the shell to evaluate expressions incorporating definitions in your file.

- Use the Python shell to evaluate `f(f(f(b)))`.
- What happens if you evaluate `f(a)`?

When you do today's lab exercises, use the shell for experimentation and write new procedure definitions in your file. Whenever you change your file, you'll need to save it and do `Run Module` again.

Some programming: Evaluating polynomials

Using your editor window—*not* the shell—type in one of the polynomial evaluation procedures shown in lecture. You can use any one you like.

Test the procedure by evaluating $x^4 + 2x^3 + 3x^2 + 4x + 5$ at $x = 10$, as well as some other examples, for which you know the anwer.

Creating your own procedures: Finding roots of polynomials

Good general-purpose polynomial root finders are very tricky to write. But there's an easy special case where you can find a single real root of a polynomial.

Here's the idea: Suppose you have a polynomial p together with two interval endpoints a and b where $p(a) < 0$ and $p(b) > 0$. Then p must have at least one real root on the interval between a and b, because polynomials are continuous functions.

To find the root, compute m, the average of a and b, and check whether $p(m)$ is greater than or less than 0. If $p(m) < 0$, then p must have a root between m and b. So continue searching for a root on the interval from m to b. Conversely, if $p(m) > 0$, then p must have a root on the interval from a to m, so continue searching on that interval.

Repeat this process over and over. At each step, the interval becomes half as large. You can stop when you've isolated the root to an interval whose size is less than some specified error bound e (e.g., $e = 10^{-5}$).

Your job is to define a procedure called `halfInterval` that implements this method. The procedure should take four arguments: a polynomial p specified by a list of coefficients, interval endpoints a and b where $p(a) < 0 < p(b)$, and an error bound e. The `halfInterval` procedure should return a list of two elements: the approximate root together with the value of p at that point. The code should work both for $a < b$ and $a > b$. You'll need to use the polynomial evaluation procedure you wrote above.

There are many ways to write this program. Make sure you have a plan before you dive in writing code. To help in debugging and to see the progress of your algorithm, you might want to print `a` and `b` on each iteration.

Question 1: Implement `halfInterval`.

Question 2: Test your code by finding a root of $x^2 - x - 1$ on the interval from 0 to 2, and a root of $x^4 - 10x^3 + 2x^2 + x + 1$ on the interval from 1 to 10.

Improving the code

Does your `halfInterval` procedure do any error checking? What happens if you give it inputs a and b where:

- $p(a)$ and $p(b)$ have the same sign
- $p(a) > 0$ and $p(b) < 0$?

Question 3: Write a procedure called `polyRoot` that uses `halfInterval` after first checking that the input conditions are OK. If $p(a)$ and $p(b)$ have the same sign, then your `polyRoot` procedure should print a complaint and stop. If $p(a) > 0$ and $p(b) < 0$, your procedure should call `halfInterval` with the arguments appropriately modified so that you'll get the right answer.

Question 4: Think of some test cases that exercise the new capabilities of this procedure and make sure your implementation works correctly.

At the end of this lab, go to the on-line tutor at `http://sicp.csail.mit.edu/6.01`, choose the problem set for this week, and paste in your definitions of `halfInterval` and `polyRoot` and any auxiliary procedures you wrote into the box provided in the "Software Lab" problem. Be sure to include some test cases to show that these work.

Week 1 Software Lab Explorations: Sparse Polynomials

This exploration is worth 2 exploration points. Together with the exploration from the design lab, this constitutes one 10-point exploration assignment.

The Explorations section is not required. However to get an A in 6.01, you need to do some of the explorations (see the grading policy).

In lecture and in lab, we represented polynomials as lists of coefficients. That's a fine representation for polynomials where most of the coefficients are non-zero, but it's wasteful for something like $x^{85}+2x^{37}-4x^{20}+3$, because it leads to long coefficient lists that are mostly zero. A more appropriate representation for such a *sparse polynomial* is as a list of pairs, where each pair specifies a non-zero term: the exponent of the term and the coefficient of that term.

Exploration 1: Fill out the details in this representation and show how to generalize Horner's method to evaluate sparse polynomials.

Exploration 2: You can also do one of the other polynomial evaluation methods (one can be done very beautifully in a single line with a list comprehension) if you wish, but Horner's is the most interesting.

Exploration 3: Turn in your code and some tests, and *also* discuss the efficiency of this procedure. What can you say about the number of multiplications required to evaluate sparse polynomials with various degrees and numbers of terms?

Homework due before design lab

The following homework is due before your section's design lab:

1. Log in to the online tutor and do problems that are assigned. This is the first half of this week's tutor problems. The rest are due before next week's lecture.
2. Read the course notes for week 1 and review the lecture notes.
3. Finally, read through the entire description of the design lab so that you'll be prepared to work on it. Note the questions you'll need to answer in class. Pay special attention to the section on robot behaviors and to the code presented in that section and in the problems listed before Checkpoint 3. There *will* be a question about this in this week's nanoquiz.

Design Lab for Week 1

After you complete the nanoquiz, find your partner (partners will be assigned) and a lab laptop and a Pioneer robot. Start by executing:

```
athrun 6.01 update
```

in a Unix terminal window to generate the `Desktop/6.01/lab1` directory, which has the files mentioned in this handout.

This lab is organized into several sections, each of which contains some checkpoint questions. Be sure to show your work to a staff member at the end of each checkpoint and be ready to explain your answers.

Before you leave lab today, make sure that both you and your partner have copies of the code that you wrote for the lab. You'll need it for homework and for next week's lab.

SoaR Basics

SoaR is the software interface to the Pioneer robots that we will be using throughout the course. It provides a robot simulator as well as an interface to the real robots. [1]

SoaR

Using the simulator: Start SoaR by typing

```
soar
```

in a Unix terminal window (or by pressing the SoaR icon if there is one). When the SoaR window comes up, choose the simulator by pressing *Simulator* (slider icon) and pick one of the supplied worlds (for instance, `tutorial.py`). To drive the robot around using the joystick, press *Joystick* (joystick icon) and then *Start* (right triangle icon). Click and drag the vector in the resulting Joystick window to control the robot. Figure out what the direction of the vector represents.

Now try controlling the simulated robot with a program. Select *Brain* (light-bulb icon) and choose `testBrain.py` from the `ps1` folder. Then hit *Start*. The robot should zoom around the simulated world, avoiding obstacles.

Using the real robot: Now try the real robot. Connect the robot to the laptop using the serial cable, and turn the robot on. To drive the robot around using the joystick press *Pioneer* (gear icon) on the SoaR window, and *Joystick* (joystick icon) and then *Start* (right triangle icon). Click and drag in the resulting Joystick window to control the robot.

Our Pioneer robots have *sonar* sensors for measuring the distance to obstacles in different directions and *odometry* sensors for measuring how far the robot has moved. It's easy to understand what these sensors would do in an idealized setting like the simulator, but their actual behavior on the real robot is more complicated.

[1] For homework, you'll be using your own laptop or Athena. Use a lab laptop for today's design lab because you'll be working with the robot that connects to the lab computer's serial port.

The goals for your work in this lab are: to become familiar with reading the robot's sensors, issuing motor commands to get the robot to move and building "behaviors" that map sensor readings to motor commands. We also would like for you to come to appreciate that real-world sensors and effectors don't usually behave in the idealized way you might wish they would.

Brains

SoaR interacts with the robot (real or simulated) via a *brain program*.

A SoaR brain has the following structure:

```
def setup():
    print "Starting"
def step():
    print "Hello robot!"
```

The `setup` procedure is called once, when the brain is started. The the `step` procedure is called over and over, several times per second. The above brain, if run, will start by printing `Starting`, and then just keep printing `Hello robot!` in the SoaR message window until you stop it.

Use IDLE to open an editor window and type in the above definitions. When you are done you should save the file as `reallysimplebrain.py` in your own `ps1` folder so you'll have access to it later. To use it, you'll have to navigate to that folder when you click on the Brain button.

To run this brain, open the simulator in SoaR and pick the `tutorial.py` world again. Now choose *Brain* (light-bulb icon) and select `reallysimplebrain.py`, and finally hit *Start*. As you can see, this brain is deadly boring. Don't worry if you click the stop button and it keeps printing for a while. All that text has been stored up somewhere waiting to be printed.

In general, you define a SoaR brain in a file and load it into SoaR. If you need to edit the brain file, you can edit in the editor and then tell SoaR to reload the brain by pressing the "reload brain" button.[2]

Sonar

The robot has eight ultrasonic transducers (familiarly known as sonars). They send out pulses of sound and listen for reflections. The raw sensors return the time-of-flight values, which is how long it took the sound to bounce back. The robot's processor converts these into distance estimates.

For the sonars on the simulator, you'll see lines protruding from the robot at various points: these represent the signal paths of the sonar sensors. These simulated sensors follow an ideal model, in that they always measure the distance (with a small amount of added random noise) down that ray to the first surface they contact.

The command `sonarDistances()`, when executed in a brain, returns a list of eight numbers—the robot's current sonar readings, measured in meters. Try saving the following code (as say, `"simplebrain.py"`) and running it on the simulator:

```
def step():
    print "sonarDistances:", sonarDistances()
```

[2]If you use IDLE as the editor, remember that you *don't* run the brain with "run module" from the run menu. You can, however, check the syntax of your code with "check module," which could be a useful thing to do before loading it into SoaR.

This brain prints the current sonar readings on every step. The printout is pretty ugly, though. Here's a modification that will print beautifully.[3]

```
def prettyList(nums):
    return ["%5.2f" % n for n in nums]

def step():
    print "sonarDistances:", prettyList(sonarDistances())
```

In the simulator, you can pick up and drag the robot with the mouse while the brain is running and watch how the readings change. Which position in the list of values corresponds to which sensor?

Now try the same thing with the real robot, putting things (and also putting yourself) in front of different sensors and seeing how the values change. You should find that the results, unlike for the simulator, are far from ideal. Think of the transducer as emitting a cone-shaped beam of sound waves, which are reflected off surfaces back to the detector. The physical details of the shape of the cone, and the way the reflections work, can have a big effect. Perform experiments and take notes, to explore the following issues:

Question 5: What happens with things very close to the sonar?

Question 6: How far away can you get reliable distance readings? What happens when the closest thing is farther away than that?

Question 7: How do the results vary as a function of the angle between the transducer and the surface that it is pointed toward? Does this behavior depend on the material of the surface? Try bubble wrap versus smooth foam core: Are the results different for the two materials? What do you think is going on?

Checkpoint 1: To be completed by 30 minutes after the beginning of lab

Show your results for questions 5–7 to your LA and be ready to explain them.

Odometry

The robot has *shaft encoders* on each of its drive wheels that count (fractions of) rotations of the wheels. The robot processor uses these encoder counts to update an estimated *pose* (a term that means both position and orientation) of the robot in some global reference frame. You can access the robot's pose with the procedure `pose` of no arguments, which returns a tuple of three values containing the robot's x position, y position, and orientation in radians. This information is generally unreliable since the wheels slip on the floor, so counting wheel revolutions is not perfect.

Figure 1 shows the reference frame in which the robot's pose is reported. When the robot is turned on, the frame is initialized to have its origin at the robot's center and the positive x axis pointing out the front of the robot. You can now think of that frame as being painted on the ground; as you move or rotate the robot, it keeps reporting its pose in that original frame.

[3]Section 7.1 of the Python Tutorial, "Fancier Output Formatting" explains what's going on here.

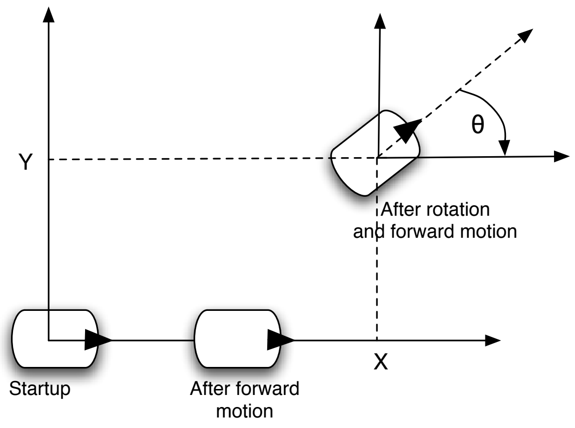

Figure 1: Global odometry reference frame

Robot motion and the transducer programming discipline

You can move turn on the robot's motors using `motorOutput(`*transVel*`,`*rotVel*`)`, where *transVel* (in meters/sec) is the (translational) velocity of the robot and *rotVel* (in radians/sec) is a rotational velocity. Experiment with this procedure a bit by adding it to a brain, to get a feeling for how it works. Be careful not to crash the robot into anything—a velocity of 1 is *fast*, so use something a lot less. `motorOutput(0,0)` will ask the robot to stop, but the robot may continue to move a little bit after you issue the command, due to inertia and command processing delays

The `step` procedure could be any Python program, but in this course we'll adhere to the discipline that a well-formed brain should execute *exactly* one call to `motorOutput` on every execution of `step`.

While the `step` procedure code might include multiple calls to `motorOutput`, embedded in `if` statements, only one of these calls should be executed each time the code runs. The idea is that the brain will decide, on each step, how fast the robot should be moving. Similarly, a well-formed `step` procedure should read the sensors with `sonarDistances` at most once on each round, typically at the beginning.

Adhering to this discipline is an example of the *transducer approach* to designing systems that interact with the environment, as described in the class notes.

Simple Brains

Write some simple brains. Try them on the simulator and then on the real robot.

Question 8: Write a brain that makes move the robot forward.

Question 9: Write a brain that makes the robot rotate.

Question 10: Write a brain that prints the robot's pose inside the `step` procedure while moving forward slowly. What is the robot's pose when your program starts? Does it get reset when the brain is stopped and restarted? Does it get reset when the brain is reloaded? Try it on the simulator and on the real robot; does it behave the same way?

Checkpoint 2: To be completed by 1 hour after beginning of lab

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

Robot behaviors

Now that we've looked at the basic robot operations, let's move to programming the robot to exhibit some specified behaviors. While we could install arbitrary Python programs as brains, we'll begin in this lab to design behaviors in a deliberate way. Next week, we'll continue in this direction, building up complexity as an illustration of the "primitives-combination-abstraction-patterns" design perspective that is a key idea of 6.01.

Here's the basic model we'll work with:

- A robot `action`, is represented as a list of two elements: the first is a name for the action, and the second is itself a list of two elements—a translational velocity and a rotational velocity. We can use the translational and rotational velocities as inputs to `motorOutput` to make the robot move.
- *Sensor data* is a list of two elements: the observed sonar readings (itself a list of readings from the individual sonars) and the robot's pose (a triple of `x`, `y`, and angle).
- A *behavior* is represented as a procedure that takes sensor data and returns an action.

Given this model, we'll adopt the convention of using brains whose `step` simply calls some behavior procedure on the sensor data to obtain an action, and then does the action by printing the action name and sending the velocity commands to the motor:

```
def step():
    doAction(behavior(collectSensors()))

def collectSensors():
    return [sonarDistances(), pose()]

def doAction(action):
    [actionName, [transVel, rotVel]] = action
    print "Action=", actionName
    motorOutput(transVel, rotVel)
```

We'll constrain things even further (this week) by using only the following robot actions (with speed=0.25):

```
stop = ["stop", [0,0]]
go = ["go", [speed,0]]          # transVel = speed (in meters/s)
left = ["left", [0,speed]]      # rotVel = speed (in radians/s)
right = ["right", [0,-speed]]
```

Thus, the behavior procedures will return one of these four actions, based on the sensor readings. (Next week we'll extend this framework to create more complex behaviors.) For example, a very simple behavior that makes the robot always go forward, would be

```
def forward(sensors):
   return go
```

Defining behaviors

You'll find the above code in the file `behaviorBrain.py` (in your `Desktop/6.01/ps1` folder), along with a couple of other useful procedures: one that computes the distance between two poses and one that returns the difference between two angles, which is nontrivial because angles are measured modulo 2π.

Load this code into an editor window and edit it to do the following problems. You should try them on the simulator and on the real robot and show them to your LA to get checked off. You'll also need to write them up for homework.

Question 11: Write a behavior that moves the robot forward until it is 30 cm from an obstacle in front of it.

Question 12: Define a new action called `back` that makes the robot back up. Now augment your behavior in question 11 so the robot backs up if something gets closer than 30 cm.

Question 13: Implement a behavior called `seek` that will try to drive the robot toward some specified target pose. If the robot is not (approximately) pointed towards the target location, then try to orient the robot towards the target location. If it's oriented toward the target but not at the (approximate) target location, then drive forward. If it's close enough to the target location but not pointed in the correct orientation as specified in the goal pose, then rotate to reach that (approximate) orientation and then stop.

Hint 1: You can't assume that the robot will necessarily start at the origin of the coordinate system.

Hint 2: You'll need to compute the angle that the robot should turn in order to face the goal. Here you can make good use of the `angleDiff` procedure provided in the code for this lab, and also Python's builtin `math.atan2` procedure, which takes two inputs `y` and `x` and returns the arc tangent of `y/x`.

Question 14: How well would your robot behaviors have worked if the step procedure were called once per second, rather than 5 times per second? What if it were called once per minute? How accurate would your behavior from question 11 be if the robot moving very quickly?

Checkpoint 3: To be completed by the end of lab

Demonstrate your behaviors running on the real robots to your LA. Be ready to explain how they work and why you designed them the way you did. Don't forget to save your code before leaving the lab.

Homework due before lecture on Feb. 12

1. Log in to the online tutor and complete the second half of this week's tutor problems.
2. Write up and hand in the answers to exercises 11 through 14.

   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. We're interested in an explanation of your thinking, as well as the answer.

   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 **only** your own code, with test examples showing how it runs, and explanations of what you wrote and why.

   Also, as mentioned at the beginning of this handout, see the "guidelines for written homework" available on the course web site. Homework that does not follow these guidelines *will not be graded* and you will get no credit for it.
3. Hand in the code you wrote for the software lab, together with test cases and results demonstrating that it works.
4. If you're planning to use your own computer for homework, this would be a good time to make sure that all the 6.01 software is installed on it, and that you can run SoaR with the simulator.

Concepts covered in this design lab

Here are the important points covered in this lab:

- You got experience with a *transducer* organization for building embedded programs, programs that interact in real-time with a world.
- You began your experience with a real robot system and began to understand that real sensors and real devices can be quite different from the simple models that we often have of them. We will need to (a) build better models and (b) devise strategies for control that can cope with this less-than-ideal behavior.
- You started robot programming in a structured way—with explicit actions, sensor values, and behaviors—rather than just mashing everything together in arbitrary piles of code.
- You also got some programming practice with Python.

Explorations: The Potential Field Approach

The Explorations section is not required. However to get an A in 6.01, you need to do some of the explorations (see the grading policy).

This exploration is worth 8 exploration points. Together with the exploration from the software lab, this constitutes one 10-point exploration assignment.

You should have found that your program for moving to a goal pose is of limited utility in the presence of obstacles. That is, that your robot could find an obstacle directly between itself and the goal that it is trying to reach, and then be stuck forever.

One popular method for avoiding obstacles while moving towards a goal is called the *potential field* method. It's based on a simple electrostatics analogy in which the robot is treated as a negatively charged particle, the goal as a positively charged particle and obstacles are negatively charged particles. Therefore, the robot is attracted towards the goal and repelled by the obstacles.

We can define an actual potential field (a scalar valued function of the robot position $p = [x, y]$) by defining an attractive potential for the goal (p_g):

$$U_g(p) = \frac{1}{2} K_{goal} \mathrm{dist}(p, p_g)^2$$

where $\mathrm{dist}(p, q)$ is the usual Euclidean distance. Similarly, we can define a repulsive obstacle potential as

$$U_o(p) = -\frac{1}{2} K_{obs} \mathrm{dist}(p, p_o)^2$$

The total potential is obtained by adding the attractive potential with the repulsive potential of each obstacle.

The force defined by a potential field is the negative of the gradient of the field. For these simple quadratic potentials, the forces are linear (spring-like forces). The attraction towards the goal is:

$$F_g(p) = -K_{goal}(p - p_g) = K_{goal}(p_g - p)$$

and the obstacle repulsive forces are:

$$F_o(p) = K_{obs}(p - p_o) = -K_{obs}(p_o - p)$$

So, we can compute a net "force" on the robot by adding the attractive force and the repulsive forces from each obstacle. (If the discussion of potentials doesn't make sense to you, don't worry; you can just operate with the two equations above, describing forces exerted on the robot by the goal and by the obstacles).

This then leaves us with a few important questions:

- How do we relate these imaginary forces to actual robot commands? We could try to use them to set the robot acceleration, in the spirit of $F = ma$, but that's dangerous, since it doesn't guarantee that we will not collide with an obstacle. We could use them to set the robot's velocity. Or, we could use them simply to define a desired direction of motion, that is, ignore the magnitude of the forces and just look at the direction.

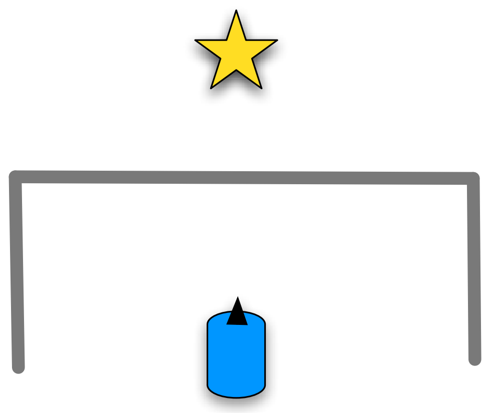

Figure 2: Robot in a cul-de-sac.

- Whatever way we interpret the forces, we must remember that our robot is not a point: it has a direction and can only move in the direction that it is pointing (or rotate). So, we can treat the net force as a desired direction of motion, rotate to that direction and then once that's done, advance in that direction (just as we did in our *seek* behavior above). In fact, you might be able to use your seek behavior as a component of your solution.

There are a number of additional practical considerations in implementing potential fields:

- Our discussion above was limited to particles. You will need to figure out the relatively straightforward generalization to circular robot and obstacles.
- We do not want the influence of obstacles to extend arbitrarily far; when moving in one side of the room, the obstacles on the other side are not relevant. Pick an influence radius and limit the repulsive forces to act in that radius.
- We want to limit the velocity of the robot within safe limits.
- We have to pick reasonable $\mathsf{K}_{\mathsf{goal}}$ and $\mathsf{K}_{\mathsf{obs}}$ and influence radius parameters.

Your implementation needs to deal with these issues.

For your exploration, hand in a write up that addresses these questions:

Exploration 4: Implement a potential field method in the SoaR simulator using the same behavior framework and the same actions that we have been using. Treat the robot and the obstacles as circles of known radius and assume you have a list of obstacle locations (centers of circles) and radii.

Exploration 5: Test your implementation initially on the `BoxWorld.py` world in the `ps1` distribution. You can open that file in an editor to see how worlds are defined and to see the location and dimension of the obstacle. Create some other worlds and test your code. Be sure that you include your test worlds (including obstacle and goal placement) in your write up. Describe your experience with your implementation.

Exploration 6: If a robot is sitting in a cul-de-sac, as shown in figure 2, what will happen if you use the potential field method? Can you think of extensions to the potential field method that might enable it to get out of a cul-de-sac. Hint: what if we can remember places that we have visited? You don't need to write code for this.

Exploration 7: Your implementation assumes that we know a map in absolute coordinates and the location of the robot in absolute coordinates. Describe how we might modify this approach assuming that we knew the goal in the robot's coordinates and had no map, only the sonars. You don't need to write code for this.

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

```
def cheatPose():
    return app.soar.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, it 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.)