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

# Assignment for Week 2

- Issued: Tuesday, Sept. 12
- Includes homework due before lab on Thursday, Sept. 14
- Also includes post-lab homework due at the beginning of class on Tuesday, Sept. 19

The objective for this week is to write some some simple Python procedures and to get comfortable doing simple data analysis and experimenting with your robot. One of the important messages here is that real-work measurements can be inconsistent and error-prone. Towards the end of the semester, you'll see how to deal with this in sophisticated ways. For now, you'll just observe the kinds of things that can go wrong. In the lab, you'll measure some properties of your robot's sonars – both as simulated and in real life. Then you'll write a simple program to make your robot chase you around.

The post-lab takes up a different theme: the consequences of different orders of growth in algorithms, where you'll experiment with different algorithms that test whether numbers are prime. The algorithms will have exponentially different orders of growth.

**Please read this entire handout before coming to lab.**

## 1. To do before lab on Sept. 14

### 1.1. Using the online tutor

Do the tutor problems that are due for September 14.

### 1.2 Keeping track of sonar readings

Here is the definition of a simple brain. Study the code in figure 1 to make sure you understand it. You can download this code from the class web site (linked from the calendar), or just type it into a file to get some practice with the editor.

Start up SoaR using the simulator and one of the built-in worlds, load the above code as the brain, step the robot several times and observe the results. Move the robot around by clicking and dragging with the mouse and observe how the results change.

Modify the brain to use a different sonar, and see whether the results make sense. (Remember to make SoaR "reload" the new brain.)

### 1.2 Calculate the arithmetic mean of recent readings.

Remember that the mean of $N$ numbers $x_1 \ldots x_N$ is

$$\overline{x} = \frac{\sum_{i=1}^{N} x_i}{N}$$

```
## This is a simple brain that at each step, prints out the last
## several values (in meters) of the readings from one sonar.

## Number of readings to keep
maxReadings=20

## Sonar to use -- the sonars are numbered from #0 (on the left side)
##                to #7 (on the right side)
sonarNum = 5

## Takes a list and a new value. Returns the list with the
## new value appended to the end, up to a given maximum length. That
## is, if the list is already at the maximum length, it drops the first
## element of the list before appending the new value.
## Note that with Python's indexing syntax list[1:] gives back a
## list containing all but the first element of the original list.

def maxExtend(list,new,max):
    if len(list) <  max:
        return list+[new]
    else:
        return list[1:]+[new]

## This is an example of the use of Python's "formatting strings".
## This one takes a number and formats it with at most two places
## before the decimal point and and three places after the decimal
## point.  We use this procedure to make the output more readable.
## Don't worry about Python's formatting string notation if it seems
## to obscure.

def format(num):
    return '%2.3f' % num


## Setup for the brain simply initializes the list of values
## The dot syntax, as in robot.readings, provides a way to name
## variables that exist in multiple functions.  We'll explain this
## in two weeks, when we talk about object-oriented programming.

def setup():
    robot.readings=[] # creates an empty list to hold the readings

## At each step, read the designated sonar, append it the list, and
## print the list of stored values
def step():
    newReading=sonarDistances()[sonarNum]
    robot.readings = maxExtend(robot.readings,newReading,maxReadings)
    print [format(x) for x in robot.readings]
```

Figure 1: Simple brain code for problem set 2

In the tutor problems, you wrote a procedure that computes the arithmetic mean (average) of a list of numbers.

Use this to modify your brain program so that rather than printing the past several values, it prints the mean of the values. Set the program up to print the mean of the last 20 values from one of the sonars. Start the robot running, and watch how the results change as you drag the robot around.

### 1.3 Standard deviation

Your brain now calculates the mean, or average, value of the most recent 20 readings of the sonar sensor. Now we want you to calculate a different statistic — the "standard deviation" of those readings. This is a measure of the "spread" or "width" of the distribution of values, and it's used all over statistics and data analysis.

When you take a test and the professors tell you that the class's score was "85 ± 13", the first number (85) is generally the mean, and the second number (13) is the standard deviation — again, a measure of how much spread there was around the mean.

Remember that the mean of $x_1 \dots x_N$ can be written as $\overline{x}$. The standard deviation is written $\sigma$ (that's a sigma) and is defined as:

$$\sigma(x_1 \dots x_N) = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

Implement a function, "stddev," that calculates the standard deviation of the elements in a list. Make as much use of your pre-existing "sum" and "mean" functions as you can. Don't duplicate work you've already done! Remember that one of the virtues of a good programmer is *laziness* — and especially, never wanting to do the same work twice. (*Hint*: You can calculate a square root by putting "`import math`" in your brain and using the "`math.sqrt()`" function.)

Then make a new brain that, at each step, prints the mean and standard deviation of the most recent 20 readings. Try moving the robot around by clicking and dragging and observe the values of the $\sigma$. Start the robot running, without moving it. What values of standard deviation do you see?

## 2. To do in lab on Sept. 14

Start the lab by loading the brain code you created for homework, and verify that it's still running. Then try the same thing with the real robot.

When you shift from the simulator to the real robot, you should keep two things in mind. One, we do not necessarily get a new sonar reading on each step of computation! For this lab, you should "import time" at the beginning of your brain, and run "time.sleep(0.1)" to pause (aka "wait" or "sleep" or "do nothing") for a tenth of a second at the beginning of each step. This will make sure you get a new sonar reading on each step.

Two, the sonars in the real robot behave differently from the ones in the simulator. Your program should ignore all values greater than two meters as outliers—don't include them in your calculations of the mean and stddev. Just pretend they were never observed and go on to the next step.

### 2.1 Real-life sonars

After making the above two modifications to your brain, produce measurements and a plot of the mean and standard deviation of the actual robot sonars bouncing off a book that's 20 cm away, 40 cm away, 80 cm away, and 160 cm away. How do these measurements differ from your results in the simulator?

The standard deviation is also sometimes called the *precision* or the *statistical error* of the measurement. Try varying the angle of the book and seeing if the precision of the sensors increases or decreases. Also jostle and spin the book around and see what standard deviations the robot measures.

### Checkpoint (should be completed by 2:15 p.m.)

- Produce a plot of the mean and precision of the sonar in the simulator and in real-life, for an distance 20 cm, 40 cm, 80 cm, and 160 cm from the sonar.
- Make measurements of the standard deviation of the sonar measurements as you spin a book around at some distance from the sonar.
- Tell us what happens when you vary the number of recent readings used to calculate the precision of the sonar.

### 2.2 Make the robot chase you

Now we're going to start using the sonars and the motors. Again, remember that we want you to write many small, simple functions — not abstruse monstrosities that are impossible to figure out. The goal for this section is to make the robot chase your hand around.

1. Write a `minval` function that takes a list and returns the smallest element. Use some simple test cases to verify that it produces the correct answer. (*Note:* Please do not use Python's built-in "`min`" function.)
2. Write a `find` function that takes a number and a list and returns the *index* of the number in the list. For example, if the function's arguments are the number −6 and the list `[5, 9, -6, 7]`, the `find` function should return "2", since the number −6 appears at the third spot in the list. Remember that indices conventionally start from zero. Use some simple test cases to make sure that `find` produces the correct answer. (What should happen if you give it a number that doesn't appear in the list?)
3. Use the `minval` function and the `find` function to implement a new function, `minindex`, which returns the index of the smallest number in a list.
4. Write a robot brain that uses `minval` applied to the sonar distances to find the closest obstacle (for example, your hand). Make the robot always turn toward the closest obstacle it sees, or if that obstacle is between the middle two sonars, the robot should move forwards. *Hint*: You should deal with the case where all sonars report distances greater than two meters.

### Checkpoint (should be completed by 3:30 p.m.)

Demonstrate that your robot successfully chases your hand around.

### 2.3 Do something creative!

For the next section, we want you to do something creative with your robot that involves the sonars and the motors. Teach it to do some new tricks. One idea is to implement a "secret code" so that waving your hand in various different ways makes the robot exhibit different behaviors (you can use the `pose` function to retrieve the robot's odometry). Be imaginative!

### Checkpoint (should be completed by 4:30 p.m.)

Write down a description of your new robot's behaviors. Then demonstrate your new well-trained robot and a few of its tricks. These should involve the sonars and the motors, and display good programming practices.

## 3. To do before Sept. 19

Work on the following problem after Thursday's lab. Turn in the written parts at the beginning of lecture on Sept. 19.

### 3.1. Reflection on lab results

Ask them to write a page or two of well-formed English describing what you did in the final part of the lab:

- What did you try to get the robot to do?
- What code did you write in order to do it?
- Explain the code
- How well did it work? How did you test it?
- How might you have improved things if you had more time?

### 3.2. Exercises with the online tutor for Sept. 19

Use the online tutor to complete the problems due for Sept. 19.

## 3.3. More Python programming practice: finding prime numbers

One of the on-line tutor exercises you did asked you to implement a procedure `findDivisor` that finds the smallest divisor (greater than 1) of a given integer. Having `findDivisor` provides an immediate way to check whether a number is prime:

```
def primeTest(n):
    return n==findDivisor(n)
```

**3.3.1. Searching for primes** Using `findDivisor` and `primeTest`, write a procedure `searchForPrimes` that, when called with an odd integer `n`, tests the primality of consecutive odd integers starting with `n` and prints the primes it finds. (Obviously, there is no point checking whether even integers are prime.) The procedure should keep running until you interrupt it by typing ctrl-C twice. Use your procedure to find the first few primes larger than $10^k$ for $k = 8, 9, 10, 11$.

**3.3.2. Timed prime search** The following procedure runs `primeTest` and prints the result, together with the number of seconds required to perform the test:

```
def timedPrimeTest(n):
    startTime=clock()
    result=primeTest(n)
    elapsedTime=clock()-startTime
    print "result: ", result, "elapsed: ", elapsedTime
```

The `clock` procedure here is supplied by Python. It returns the amount of time in seconds that the Python process has run (CPU time). In order to use `clock`, you should place the command

```
from time import clock
```

at the beginning of your code file.[1]

See how long it takes to check the primality of some of the primes you found in the previous exercise. Since `findDivisor` searches for values up to the square root of its input, you might expect the time to find primes to grow with order $\Theta(\sqrt{n})$. Do some experiments to check whether this seems to be true. Do they bear out the expectation of $\Theta(\sqrt{n})$ growth? Hint: Do a (hand) plot of the time versus the square root of the numbers being tested.

Would you expect the time to also grow as $\Theta(\sqrt{n})$ for testing arbitrary integers $n$, not necessarily prime? Why or why not?

**3.3.3. Fast primality checking** In the previous lecture, you saw a procedure that performs *modular exponentiation*, that is, given $b$, $e$, and $m$, it computes the remainder of $b^e$ modulo $m$:

[1]Here "time" is the name of Python's `time` module, and "clock" is the name of a procedure in that module.

```
def expmod(b,e,m):
    if e==0:
        return 1
    elif e % 2 == 1:
        return (b * expmod(b,e-1,m)) % m
    else:
        return square(expmod(b,e/2,m)) % m

def square(x):
    return x*x
```

Fast modular exponentiation provides a fast way to test whether numbers are prime, based on a result from number theory known as *Fermat's Little Theorem*:

> If $n$ is a prime number and $a$ is any positive integer less than $n$, the remainder modulo $n$ of $a^n$ is equal to $a$.

If $n$ is not prime, then in general, most of the numbers $a < n$ do not satisfy the above relation. This leads to the following algorithm for testing primality: Given a number $n$, pick a random number $a < n$ and compute the remainder of $a^n$ modulo $n$, and check whether that equals $a$. This is called the *Fermat test*.

If the Fermat test fails for a randomly picked $a$, then $n$ is not prime. If the test passes, then chances are good that $n$ is prime. Now pick another random number $a$ and apply the Fermat test again. If the second test also passes, then we can be even more confident that $n$ is prime. By trying more and more values of $a$, we can increase our confidence in the result.[2]

Write a procedure to implement the Fermat test, and then write a procedure `fastPrimeTest` that applies the test to a positive integer $n$. If $n$ fails the test, then it is not prime. Assume that if $n$ passes the Fermat test 5 times, then it is prime.[3]

Use your procedure to recheck some of the large primes you found in exercises 3.3.1.

**3.3.4. Fast prime search** Write a variant of `search-for-primes` from exercise 3.3.1 that uses `fastPrimeTest` and compare your results with the ones from that exercise. You should find that you can now quickly check numbers that are enormously beyond the range that was practical to check with the old algorithm. For example, what are the first few primes greater than $10^{100}$?

**Exploration: Timing the fast algorithm** What do you expect to be the order of growth of the fast prime checking algorithm? Test some large primes to see whether your expectation is true, and gather data that support—or do not support—your expectation.

[2]The Fermat test is not perfect: there are numbers $n$ that are not prime, and yet pass the Fermat test for every $a < n$. These are called Carmichael numbers. You might want to do a little exploring on the web to see what's known about Carmichael numbers. The first few of them are: $561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, \ldots$.

[3]To pick random integers use Python's `randint` procedure, which takes two integer arguments $p$ and $q$ and returns a random integer $a$ in the range $p \leq a \leq q$. You'll need to import `randint` from the `random` module by adding `from random import randint` at the top of your code file.

**3.3.5. Mersenne Primes** In 1644, the French mathematician Marin Mersenne published the claim that numbers of the form $2^p - 1$ are prime for the following values of $p$ and for no other $p$ less than 257.

2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257

It turns out that Mersenne missed a few values of $p$, and that some of the values in his list do not give primes.

(a) Which of Mersenne's values do not give primes?

(b) Prime numbers of the form $2^p - 1$ are known as *Mersenne primes*. Do some exploring on the Web to find the largest known Mersenne prime. When was it discovered?

### What to turn in

Write up your report from section 3.1, together with definitions of all the programs you wrote for section 3.3 and the answers to the questions in that section. Turn these in at the beginning of lecture on Sept. 19.

## Concepts covered in this assignment

Here are the important points covered in this assignment:

- Good programmers are lazy. They write small, simple procedures that do one thing well, and they use them over and over.
- You got a quick introduction to a lot of tools: Python, Emacs, SoaR, and the Pioneer robots.
- The robot gets information about the world from its sensors. But the sensors are inaccurate. Getting good estimates of what's really going on will require combining data from multiple sources (as we'll see later in the semester).
- Procedures generate computational processes in the computer, and different kinds of processes have different characteristic "shapes" and different orders of growth.