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

Assignment for Week 4

- Issued: Tuesday, September 26
- Includes homework and preparation due before class on Thursday, Sept. 28
- Also includes post-lab homework due before class on Tuesday, Oct. 3
- Readings:
  - study the code for this assignment (see appendix)
  - Read *How to Think Like a Computer Scientist: Learning with Python*, Chapters 12 - 16

Virtual Sensors

Now that we have had experience with measuring the performance of the sonar sensors and with using them to control the robot, many of us are wishing for better sensors. In this lab, we will try to make sensing more robust by building *virtual sensors.* A virtual sensor is a software module that gathers a variety of sensor data and processes it in an attempt to get more robust measurements. We call it a virtual sensor because the program that uses it can treat it as if it were a single real sensor value.

Because we want to hide the details of the processing that occurs inside a virtual sensor from the program that uses the sensor, and because the virtual sensor may need to aggregate real sensor readings taken at different time steps, it is appropriate to use software objects to represent virtual sensors.

So, we need to define a class of virtual sensors. In particular, we need to specify its *interface*: that is, the set of abstract operations we can do on a virtual sensor, trying to hide all of the unnecessary detail from the client. In some programming languages, like Java, the specification of the interface to a class is very rigid and must be done completely in advance. In Python, things are much more flexible, but it is still a good idea to design before you code.

What are the most basic things we'll need to do with a virtual sensor?

- create it
- update it
- get a virtual sensor reading
- ask it if its current reading is valid

Here is the definition of a virtual sensor class that actually just reads the sonar sensors directly. It remembers and then returns the most recent reading for the associated sensor index.

External interface:

self.*update* (sonarDistances, robotPose)	Updates internal state of the sensor. Pose is optional
self.*isValid* ()	Returns True if the value is good
self.*read* (robotPose)	Return virtual sonar reading. Pose is nnecessary for this simple case, but u

```
class virtualSonar:

    def __init__(self, sensorIndex, window = None):
        || Remember what sensor we are tracking
        self.index = sensorIndex
        || A place to store the value
        self.value = 0.0
        self.valid = False

    || Update our internal state. Pass in all the sonarDistances because if we are averaging over
    || space we might need to look at more than one value.
    def update(self, sonarDistances, robotPose = None):
        self.value = sonarDistances[self.index]
        self.valid = True

    def read(self, robotPose = None):
        return self.value

    def isValid(self):
        return self.valid
```

To make a more sophisticated sensor, we might need to store more data and process it in some smarter way.

The Trouble with Sonars

By now you've probably discovered that the sonar array is kind of unreliable. Individual readings can be completely inaccurate, the sonar beam can bounce off smooth surfaces and never return to the device, objects too close are invisible, and so on.

Two ways to improve on those shortcomings are to take advantage of *space* and of *time*.

Space In a given situation, one sonar may not return useful information, but the one next to it may. That's part of the advantage of having an array of sonars instead of just one: spatial redundancy. For instance, if the robot is traveling along a smooth wall, some sonars may not see anything because the beam angle to the wall is too great, but at least one sonar (hopefully) gets readings indicating the wall's presence.

What you can do, then, is improve the estimate of the distance to an object (or the detection of that object!) by taking the minimum reading over a set of sensors. In fact, that is what the `wander` and `avoid` behaviors that we worked with last week do, without explicitly constructing a virtual sensor.

Time A single reading from a given sensor might be flaky. However, the readings taken just beforehand and just afterwards might be fine. If you make use of the information in a series of readings from a sensor rather than just one, you can extract a signal from the noise that may obscure it.

To start with, you might just take the minimum or the average of a set of recent readings from a single sensor. If the robot is stationary, this is just aggregating over time; if the robot is moving, you're actually changing both time and space. The hope is that if some weird condition in the world is causing you to get bad readings, that by changing something, they might get better.

A more sophisticated approach, which we'll pursue in the post-lab work, is to collect data from a single sensor as the robot drives along a wall, and fit those data to a line, which represents the robot's belief about the wall's position. If you get a single reading at some point that doesn't match up, maybe that data point was bad and can be discarded. If you start getting enough data points like that, that are consistent with a different line, maybe the robot has reached a place where the wall really is in a different position.

1. Homework in preparation for lab on Sept. 28

1.0 Read this whole document!

Please read the entire section, **including all the code**, on the lab before coming to lab.

1.1. Exercises with the online tutor

Use the online tutor to complete problems before lab; they will exercises some of the object-oriented programming concepts from lecture and ask you to build some of the classes we need for the lab.

In lab, you might find the methods for maintaining a *queue* in Python helpful; read section 5.1.2 of the Python Tutorial to learn about them in advance.

2. To do in lab

Download `ps4.zip` from the calendar web page.

2.1 Wall following

In the code for this week, you'll find `wallFollowBrain.py`, shown below. It makes use of the simplest virtual sonar sensor, and tries to drive the robot along, following the wall on its right at a fixed distance.

Get it running on your robot, and see how well it works.

Checkpoint: 1:45 PM

- Run the existing software
- Explain any problems it might have to a staff member

2.2 Better Virtual Sensors

One simple thing that might improve the sonar readings, if they occasionally get a bad value, is to base a virtual reading on a few recent readings. If your last three readings were 12, 10, 31, and 11, what would you guess is the distance to the relevant surface?

The program `virtualSonar.py` is in the zip file for this problem set; it's the very simple class definition shown above. Edit this file, and extend your virtual sonar to keep a small history of readings from that sensor and to return a virtual value based on those readings. You might want to use a queue for this; see section 5.1.2 of the Python Tutorial to learn about them.[1]

There are a couple of ways of taking historical averages. The simplest is to remember the last k values of the sensor, and then to compute an equally weighted average of those values. What kind of behavior would you get in the extreme of a large k?

One computational problem with having a large k is that it takes space to store the old values and time to recompute the new average. We can solve this problem by computing what's called a *moving average* or a *recursive estimate*.[2] Let s_t be the sensor reading at time t and a_t be our "average" value at time t.

Certain kinds of averages can be computed by remembering only a_t, and doing an update like this:

$$a_{t+1} = w_t s_t + (1 - w_t) a_t \quad .$$

Where did we get w_t? In the very simplest case, if $w_t = 1$, then we have the original strategy of just looking at the most recent measurement. But what happens if we let $w_t = .5$, for example? What happens if we let $w_t = 1/t$? Is there a way setting of w_t that captures the same behavior as averaging the last k readings?

Think of a way to evaluate whether your virtual sensor is performing well. Try using a different amount of history and/or a different processing method and compare their performance. Think about the advantages and disadvantages of using a long history of measurements. If you have extra time, try some other kinds of virtual sensor processing, possibly including aggregating measurements from multiple sensors.

[1] Extra credit (well, actually, just extra Spam) if you can give the last names of Eric, John, Michael, Terry, and Graham. It's not called "Python" for nothing...

[2] This use of the word "recursive" is somewhat different from our usage to describe functions that call themselves. Unfortunately, technical terms often get used to describe multiple, sometimes distantly related, concepts. In this case, the idea is that most of the work of the update was done previously, and we only need to do the last step.

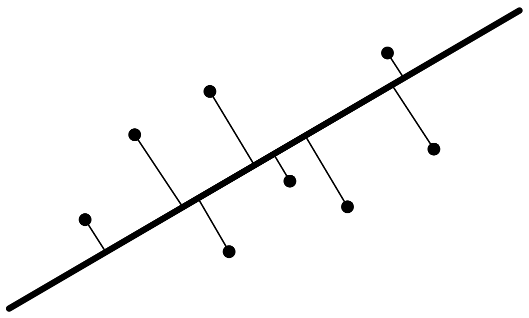

Figure 1: Perpendicular distance from a collection of points to a line.

Checkpoint: 3:00 PM

- Code for a virtual sensor that aggregates over time
- Answers to the questions about recursive estimation: What happens if we let $w_t = .5$, for example? What happens if we let $w_t = 1/t$? Is there a way setting of w_t that captures the same behavior as averaging the last k readings?
- Results of evaluating at least two versions of your virtual sensor

2.4 Finding Walls

One way to be more sure of your sensor readings is to find that they are consistent with some hypothesis you have about what the external world is really like. So, for instance, if you get a bunch of sonar readings that all line up in a line, then you might be well justified in believing they were all reflected from a wall or other linear surface in the world. It is very unlikely that you would have gotten that pattern of readings accidentally (though we've seen it happen when a person walks along right next to a robot!).

Line Fitting

Given a set of data, expressed as a set of $\langle x_i, y_i \rangle$ coordinates, we can try to find the equation of a line that minimizes the *sum squared error*. That is, the line that minimizes the sum, over all the points, of the square of the perpendicular distance from each point to the line. Figure 1 illustrates this idea.[3]

A line can be expressed using three parameters,[4] a, b, and c, as the set of all points satisfying the equation

$$ax + by + c = 0 \ .$$

[3]This is also known as "least-squares" line fitting. In many cases, rather than minimizing perpendicular distance, the squared vertical distance from points to the line is minimized instead.

[4]You are probably used to a parameterization of lines with $y = mx + b$, which has only two parameters. So why do we use three? Because if the line is vertical, it can't be represented in using $y = mx + b$ because the coefficient of y needs to be 0. In order to avoid having special cases, we use all three parameters. But in many cases you can just set one of them to 1; or some people like to normalize the coefficients so that $\sqrt{a^2 + b^2} = 1$, which makes testing the distance to the line more efficient.

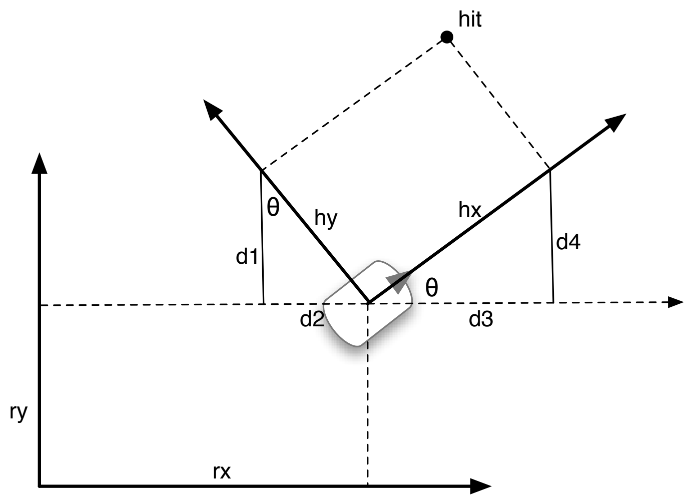

Figure 2: Transforming a sonar hit into the global coordinate frame. The coordinates of the hit in the robot's frame are $\langle hx, hy \rangle$. The pose of the robot in the global frame is $\langle rx, ry, \theta \rangle$. What is the location of the hit in the global frame?

For example, a 45-degree line through zero satisfies $x + y = 0$, and a vertical line passing through $y = 0, x = 1$ satisfies $x - 1 = 0$. The signed distance from a point $\langle x, y \rangle$ to a line $\langle a, b, c \rangle$ is

$$d(\langle x, y \rangle, \langle a, b, c \rangle) = \frac{ax + by + c}{\sqrt{a^2 + b^2}} \quad .$$

The sign encodes which side of the line the point is on. Convince yourself that if you plug a point, $\langle x^*, y^* \rangle$, that is on the line into this equation, the distance is zero.

So, the sum squared "error" of a line, over a whole data set, is

$$sse(a, b, c) = \sum_i d(\langle x_i, y_i \rangle, \langle a, b, c \rangle)^2 \quad .$$

Given a set of n sonar hits, then, our job is to find values of a, b, and c that minimize *sse*. This "best line" will be our best guess for a wall that the sensors are seeing (if indeed, they are seeing wall).

We could try to find a, b, and c by putting a grid on the three-dimensional parameter space or doing a fancier search technique of the kind we thought about last week. But luckily for us, it turns out that there is an analytic, closed-form solution to this particular minimization problem. Given basic calculus and some algebraic wrestling, it's not too hard to derive the solution. We'll ask you to do some of this work on the post-lab problems.

Finding a wall

To find a wall, you need to look for patterns of lines in your sonar data. The following steps will help you write a wall finder.

Question: Sonar hits are returned in "robotcentric" coordinates (that is, a system that is centered on the current robot pose, with the x-axis pointing out the robot's nose). So, each subsequent hit, if the robot is moving, is reported in a different system. In order to fit a line to such a set of points, it's crucial to have them all in the same system: the global coordinate system in which the odometry is reported is a handy system for this. Edit the file `poseClassIncomplete.py` to finish the method `transform(self, p)`; then save the file out as `poseClass.py`. If `p` is a point, defined in the local frame of the pose, the method returns the transformed version of point in the global frame. You might find figure 2 helpful. Start by figuring out the distances $d_1, \ldots, d_4$ in terms of the hit location in the local frame.

Question: Once you have fit a line to a set of points, you'll have to decide whether the result is good (that is, the line is a good description of the points). Edit the file `lineFitterIncomplete.py` to finish the method `maxError(self)`; then save the file out as `lineFitter.py`. The method should return the largest perpendicular distance from the set of points to the best-fit line. You can use this later, to decide whether a line fit is good.

Checkpoint: 3:45 PM

- `transform` method
- `maxError` method

Now, you'll need to substantially extend the code in in `lineTrackerShell.py`, and save it out as `lineTracker.py`.

It should store up some sonar readings and try to find a line in them. Some things to watch out for are:

- Lots of readings very close together can sometimes lead to bogus fits; it might be good to throw away readings that are very close to ones you already have.
- Use your `maxError` method to to keep from delivering bogus line hypotheses.
- Try to keep your line current by adding and deleting new points as the robot moves.
- You might want to just throw out very long readings.
- Make it throw out an old hypothesis if all the new points are being rejected, and start over.

At a minimum, make it reinitialize the line fit if all the new points it is seeing are incompatible with the current hypothesis.

Debug it in the simulator using `wallFollowBrain.py` (but while debugging you might want to set the gain to 0, so the robot will just go straight). You'll have to uncomment the line that uses the `lineSensor` instead of the `virtualSonar` sensor. The `lineSensor` asks the `lineTracker` to fit a line to the sonar readings, and if it finds a good one, it returns the distance to the best-fit from the sonar sensor, rather than the actual sonar reading. This ought to mean that your robot can ignore pieces of junk against the walls of a hallway, for example. This also creates a window, which you can use for debugging. We found it useful to draw the sonar hits in the window (in the global frame) to debug our transformation code. Then, you can experiment with drawing them in different colors depending on whether or not you think they fit into your current line. And use the `drawLine` procedure to draw your line hypotheses. It will be really hard to debug if you don't do this. (If you don't like to have your window so cluttered, you can save the result of drawing a line like this:

```
self.windowLine = window.drawLine(self.line)
```

And then delete it it on the next step with something like this:

```
window.delete(self.windowLine)
```

Checkpoint: 4:45 PM

- Demonstrate your line tracking in simulation.
- Demonstrate it on your robot. Does it make the robot behave more robustly?

3. To do before October 3

3.1. Exercises with the online tutor

Use the online tutor to complete the problems due for October 3

3.2. Fitting lines

Let's work on finding the equation of a line that minimizes the sum-squared perpendicular error of a set of points to the line. You remember how to do minimization, right? You take the derivative, set it to zero, and solve. Let's start doing that.

Question 1a: What are the values of

$$\begin{array}{ll} \frac{\partial sse}{\partial a} & \\ \frac{\partial sse}{\partial b} & \text{and} \\ \frac{\partial sse}{\partial c} & ? \end{array}$$

If we set those expressions to zero, we have a system of three equations in three unknowns to solve. The algebra is kind of hairy, so we we'll only ask you to work a little bit of it out.

The *centroid* of the data is a point whose coordinates are the average of the x coordinates, $\frac{1}{n}\sum_i x_i$, and the average of the y coordinates, $\frac{1}{n}\sum_i y_i$. We sometimes call these quantities the *means* of the dimensions, and write them $\bar{x}$ and $\bar{y}$.

Question 1b: Now, if we look at the result of setting $\partial sse/\partial c = 0$, we can solve for c in terms of $\bar{x}$ and $\bar{y}$. What result do you get? What relationship do you see between the centroid and the line?

Rather than asking you to continue solving the equations, we'll present the answer here, so you can use the result in questions 2–4.

Let's start by naming some useful sums over the data:[5]

$$\begin{aligned} s_x &= \sum_i x_i \\ s_y &= \sum_i y_i \\ s_{xx} &= \sum_i x_i^2 \\ s_{yy} &= \sum_i y_i^2 \\ s_{xy} &= \sum_i x_i y_i \end{aligned}$$

So now the means are defined as $\bar{x} = s_x/n$ and $\bar{y} = s_y/n$. Now, we can define the *sample variances*:[6]

$$\begin{aligned} \sigma_x^2 &= \frac{s_{xx}}{n} - \frac{s_x^2}{n^2} \\ \sigma_y^2 &= \frac{s_{yy}}{n} - \frac{s_y^2}{n^2} \end{aligned}$$

and the covariance:

$$\sigma_{xy}^2 = \frac{s_{xy}}{n} - \frac{s_x s_y}{n^2}$$

The variances, intuitively, describe how spread out the data is in each dimension; the covariance is a measure of how "rotated" the cloud of data points is (it has value 1 if the axis along with the points are most spread out is at a 45-degree angle, and -1 if it's at 135 degrees).

We'll define one more useful quantity:

$$h = \frac{\sigma_y^2 - \sigma_x^2}{\sigma_{xy}^2} \quad .$$

Given all this shorthand, we can tell you what the solution to the three equations is that you got from the partial derivatives. In all but some special cases, the solution can be written as:

$$\begin{aligned} a &= h \pm \frac{1}{2}\sqrt{h^2 + 2} \\ b &= -1 \\ c &= -a\bar{x} + \bar{y} \end{aligned}$$

Something is funny here: there are actually two answers given, because of the $\pm$. One is the answer we want and the other is a saddle point (it has zero derivative but is neither a minimum nor a maximum). The easiest way to decide which is which is to take both values for a (and the c values that are derived from them), and put them into the *sse* function to see which value minimizes squared error.

[5]People often call sums like these *statistics*: it turns out that all you need to remember about your data is these summary statistics, and you can add new points and still compute the best fit line.

[6]Some statisticians might complain that we're using a biased form of the estimator. They're right, but we don't want to get into it here and now, and it won't make much of a difference in our application.

We can get into trouble trying to use that solution when the covariance, σ_{xy}^2, is 0. This happens when the data are symmetric with respect to both the x and y axes. If this is true, we have to consider the situation a little further. If the variances are also equal, $\sigma_y^2 = \sigma_x^2$, then the points are in a completely 4-way rotationally symmetric arrangement. In this case, there is no reasonable answer to be given as to the best fit line (imagine data in a perfect circle or square). Otherwise, if $\sigma_x > \sigma_y$ then the data are more stretched along the x axis and the best fit is a horizontal line, with $a = 0$, $b = 1$, and $c = -\bar{y}$. Finally if $\sigma_x < \sigma_y$ then the best fit is a vertical line, with $a = 1$, $b = 0$, and $c = -\bar{x}$. Whew! It's a little messy, but math has really helped us with our job here.

Our Python code for fitting a line is in the appendix below, in `lineFitter.py`.

What to turn in

For questions 1 and 2, hand in a written solution that shows your answer and how you got it; include pictures of a couple of example cases that verify your solution is correct.

Concepts covered in this assignment

Here are the important points covered in this assignment:

- Object-oriented program can help abstract away from implementation details and encapsulate state in a real application.
- Sensor readings can sometimes be made more sensible when interpreted in the context of other readings rather than on their own.
- Analytic geometry is good for something.

You also got more programming practice with Python.

Appendix: All the code

point.py

Support the following interface to the outside world.

Point (x, y)	Creates a new point
self.*distToPoint* (p)	Returns the distance to point p
self.*isNear* (p)	Returns True if point p is within epsilon of self
self.*addPoint* (p)	Adds the point p, to self, changing self
self.*scalePoint* (a)	Multiplies both coordinates of self by a
self.*magnitude* ()	Returns the distance form self to the origin
self.*xytuple* ()	Returns a tuple with the x and y coords of self

```
import math
epsilon = 0.000001

class Point:
    x = 0.0
    y = 0.0

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distToPoint(self, p):
        return math.sqrt((p.x − self.x) ** 2 + (p.y − self.y) ** 2)

    def isNear(self, p):
        return self.distToPoint(p) < epsilon

    || Add this point to self, and modify self
    def addPoint(self, p):
        self.x+ = p.x
        self.y+ = p.y

    def scalePoint(self, a):
        self.x = self.x ∗ a
        self.y = self.y ∗ a

    def magnitude(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)
```

```
def xytuple(self):
    return (self.x, self.y)
```

line.py

Support the following interface to the outside world. Constructor can accept three different sets of arguments:

1. (a, b, c)
2. (m, b): slope and intercept
3. (p1, p2): two points

In any case, it should arrive at an internal representation of (a, b, c)

self.*distToPoint* (p) Returns the perpendicular distance from point p to the line

---

```
import math

class Line:
    def __init__(self, constructorType, args):
        if constructorType == "abc":
            self.initializeABC(args)
        elif constructorType == "mb":
            self.initializeMB(args)
        elif constructorType == "points":
            self.initializePoints(args)
        else:
            raise("Unknown constructor type")

    def initializeABC(self, (a, b, c)):
        self.a = a
        self.b = b
        self.c = c

    def initializeMB(self, (m, b)):
        self.a = m
        self.b = -1
        self.c = b

    def initializePoints(self, (p1, p2)):
        self.a = p2.y - p1.y
        self.b = p1.x - p2.x
        self.c = p2.x * p1.y - p1.x * p2.y
```

```
def distToPoint(self, p):
    return abs(self.a * p.x + self.b * p.y + self.c)/math.sqrt(self.a ** 2 + self.b ** 2)

|| The line drawer doesn't know about this class, so sometimes we have to make a tuple
def abcTuple(self):
    return (self.a, self.b, self.c)
```

poseClassIncomplete.py

Support the following interface to the outside world.

Pose ((*x*, *y*, theta), (xorigin, yorigin))	Constructs a pose from a tuple of values, subtracting off the origin point (but leaving theta alone)
self.*getPoint* ()	Returns a point with the *x* and *y* coordinates of the pose only
self.*transform* (*p*)	Returns a new point, transformed by pose

```
import math

import point
reload (point)
from point import Point

class Pose:

    def __init__ (self, (x, y, theta), (xOrigin, yOrigin, thetaOrigin))
        self.x = x − xorigin
        self.y = y − yorigin
        self.theta = theta

    def getPoint (self):
        return Point (self.x, self.y)

    || Returns a new point. New point is in the global frame, if it was originally local to the pose
    || frame
    def transform (self, p):
        || Your code here
```

lineFitterIncomplete.py

Support the following interface to the outside world.

LineFitter ()	Makes a new line fitter with no points
self.*clear* ()	Empties out the set of points to be fit
self.*mostRecentPoint* ()	Returns the point added most recently
self.*push* (*p*)	Adds a point p to the list
self.*pop* ()	Removes the oldest point from the list
self.line	The best-fit line to the current set of points
self.*maxError* ()	The maximum perpendicular distance of one of the set of points to the best fit line.

```
import math
import point
reload (point)
from point import *
import line
reload (line)
from line import *

class LineFitter:

    def __init__ (self):
        self.clear ()

    def clear (self):
        self.points = [ ]
        self.n = 0
        self.line = None

    def mostRecentPoint (self):
        return self.points[-1]

    || Keep the best fine line updated, whenever the list of points changes.
    def push (self, p):
        self.points.append (p)
        self.n+ = 1
        self.line = self.bestPerpFit ()

    def pop (self):
        self.points.pop (0)
        self.n- = 1
        self.line = self.bestPerpFit ()
```

```
def maxError(self):
    || Your code here

|| Sum squared distance from list of points to line
def sse(self, line):
    return sum([line.distToPoint(p) ** 2 for p in self.points])

def bestPerpFit(self):
    sum_x = 0.0
    sum_y = 0.0
    sum_xx = 0.0
    sum_yy = 0.0
    sum_xy = 0.0
    for p in self.points:
        sum_x = sum_x + p.x
        sum_y = sum_y + p.y
        sum_xx = sum_xx + p.x * p.x
        sum_yy = sum_yy + p.y * p.y
        sum_xy = sum_xy + p.x * p.y
    if self.n <= 1:
        || not enough points
        return None
    xmean = sum_x/self.n
    ymean = sum_y/self.n
    || should be divided by n**2, but it cancels
    cov = sum_xy * self.n - sum_x * sum_y
    xvar = self.n * sum_xx - sum_x ** 2
    yvar = self.n * sum_yy - sum_y ** 2
    numer = yvar - xvar
    if near(cov, 0.0) and near(xvar, yvar):
        || points are in a 4-way symmetric blob
        print "Error: no line fit"
        return None
    elif near(cov, 0.0) and yvar > xvar:
        || vertical line
        return Line("abc", (1, 0, -xmean))
    elif near(cov, 0.0) and xvar > yvar:
        || horizontal line
        return Line("abc", (0, 1, -ymean))
    else:
        h = numer/cov
        a1 = (h + math.sqrt(h ** 2 + 4))/2
        a2 = (h - math.sqrt(h ** 2 + 4))/2
        c1 = -a1 * xmean + ymean
        c2 = -a2 * xmean + ymean
        l1 = Line("abc", (a1, -1, c1))
        l2 = Line("abc", (a2, -1, c2))
        || Pick the value of (a,c) that minimizes squared error
        if self.sse(l1) < self.sse(l2):
            return l1
        else:
            return l2
```

Never test real values for equality!!

```
def near(v1, v2):
    return abs(v1 − v2) < .0000001
```

lineSensor.py

Support the following interface to the outside world.

LineSensor (sensorIndex, window)	Makes a new line sensor, taking readings from sonar number sensorIndex; using window for debugging output
self.*update* (distances, pose)	Given a vector of sonar readings and the current robot pose, update the state of the line sensor
self.*read* (pose)	Returns a virtual sonar reading, by "shooting" a ray out of the sensor and seeing how far it goes before intersecting the line
self.*isValid* ()	Returns True if the readings are likely to be valid

```
from soarmath import intersection
from simulator import SONAR_INFO

import line
reload (line)
from line import *
import poseClass
reload (poseClass)
from poseClass import *
import lineTracker
reload (lineTracker)
from lineTracker import LineTracker
import point
reload (point)
from point import *

maxReading = 10.0

sonarPoses = [Pose (p) for p in SONAR_INFO]

class lineSensor:

    def __init__ (self, sensorIndex, window):
        self.tracker = LineTracker (sensorIndex)
        self.sensorIndex = sensorIndex
        self.w = window
```

```
|| Update our internal state. Pass in all the sonarDistances because if we are averaging over
|| space we might need to look at more than one value.
def update (self, distances, pose):
    def getSonarHit (d, pose):
        return pose.transform (Point (d, 0))
    hits = map (getSonarHit, distances, sonarPoses)
    self.tracker.update (pose, hits, self.w)

|| Read a value
def read (self, robotPose):
    || This part is really gross. Need to find two points on this line for Mike's intersection
    || routine. Better to use something that intersects a line and a ray
    if self.tracker.currentLine == None:
        return maxReading
    else:
        (a, b, c) = self.tracker.currentLine.abcTuple ()
        if abs (b) > .00001:
            lineP1 = Point (100, −(a ∗ 100.0 + c)/b)
            lineP2 = Point (−100, (a ∗ 100.0 − c)/b)
        elif abs (a) > .00001:
            lineP1 = Point (−(b ∗ 100.0 − c)/a, 100)
            lineP2 = Point ((b ∗ 100.0 − c)/a, −100)
        else:
            return maxReading

        localSonarPose = sonarPoses[self.sensorIndex]
        globalSonarPoint = robotPose.transform (localSonarPose.getPoint ())
        globalSonarRayEnd = robotPose.transform (localSonarPose.transform (Point (3.0, 0.0)))
        sonarLine = Line ("points", (globalSonarPoint,
            globalSonarRayEnd))
        crossing = intersection ((globalSonarPoint.xytuple (),
            globalSonarRayEnd.xytuple ()),
            (lineP1.xytuple (), lineP2.xytuple ()))
        if crossing == False:
            return 10.0
        else:
            if self.w != None:
                self.w.drawPoint (crossing[0], crossing[1], color = "orange")
            return globalSonarPoint.distToPoint (Point (crossing[0], crossing[1]))

def isValid (self):
    return self.tracker.isLineValid
```

lineTrackerShell.py

Support the following interface to the outside world.

LineTracker (sensorIndex)	Creates a new line tracker using data from sensor with index sensorIndex
self.currentLine	Variable containing the current line hypothesis
self.isLineValid	Variable that's True when the line hypothesis is reasonable
self.*update* (robotPose, sonarHits, window)	Updates the hypothesis based on the robot's odometry, and a list of all the sonar hits in local coordinates window is used for debugging information

```
import math

import lineFitter
from lineFitter import *
reload(lineFitter)

class LineTracker:
    def __init__(self, sensorIndex):
        || Index of sensor whose readings we're tracking
        self.sensorIndex = sensorIndex
        || The line fitter
        self.lineFit = LineFitter()
        || Is the current line valid?
        self.isLineValid = False
        self.currentLine = None
        || Remember this so we can delete old line from drawing
        self.windowLine = 0

    || Update tracker with a new reading. Needs to know current pose of robot in global coordinates
    def update(self, pose, sonarHits, window):

        localHit = sonarHits[self.sensorIndex]
        || Location of reading in global coordinates
        globalHit = pose.transform(localHit)

        || Your code here. Remember to study the interface to LineFitter.
```

wallFollowBrain.py

```
import poseClass
reload (poseClass)
from poseClass import Pose

import lineSensor
reload (lineSensor)
from lineSensor import lineSensor

import virtualSonar
reload (virtualSonar)
from virtualSonar import virtualSonar

import DrawingWindow
from DrawingWindow import DrawingWindow
```

Follow the wall on the left or right. Do right, by default.

```
clearance = 0.4
gain = 0.2
def clip (v, vMin, vMax):
    return max (vMin, min (v, vMax))

def setup ():
    robot.w = DrawingWindow (600, 600, -1, 8, -4, 5)
    || robot.rightSensor = lineSensor(7, robot.w)
    robot.rightSensor = virtualSonar (7, robot.w)
    || Remember initial pose; we'll use it as the center of our coordinate system so that the graphics
    || comes out nicely
    robot.origin = pose ()

def step ():
    || Make a pose object representing where the robot is relative to the origin.
    robotPose = Pose (pose (), robot.origin)
    || Update the virtual sensor
    robot.rightSensor.update (sonarDistances (), robotPose)

    || Move the robot.
    if robot.rightSensor.isValid ():
        positionError = clearance - robot.rightSensor.read (robotPose)
    else:
        || Invalid reading; don't rotate for now
        positionError = 0.0
    motorOutput (0.1, clip (gain * positionError, -0.3, 0.3))
```