A Bayesian Approach to Plotting Noisy Sensor Data

Sophia Seitz and Claire Diehl

October 2014

1 The Question

The output of a sensor, in our case an infrared range finder, is often not as accurate as we would like it to be. In previous classes, we have tried averaging many points together, and some other methods to try to fix this problem, but we still wonder whether or not there is a better way, or whether we can do this in a Bayesian way.

We take inspiration from another Bayesian problem, one that uses altered GPS data to better characterize the actual position of the sensor. After solving this problem and working with several different LIDARs and dealing with this sometimes less than optimal sensor, we want to try a Bayesian update of the location of a perceived object.

2 Our Solution

Our system is a range finder mounted to a servo that sends range readings to an arduino. The arduino then communicates the data to python via the serial port. From there, we take the data and update our hypothesis about where there might be something and then finally, we plot a representation of where there are probably objects.

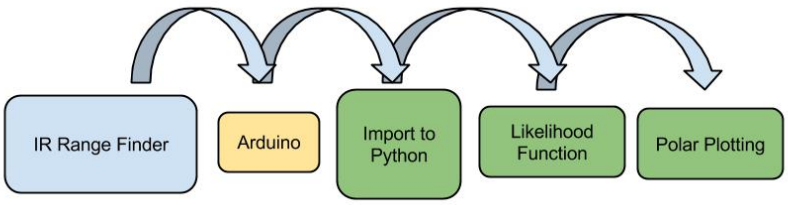

Figure 1: System Diagram

2.1 Sensor Setup

To get the actual sensor data, we mounted an infrared range finder to a servo. We then rotate the servo through a range of angles, collect the output of the range finder and the angle, and pass it to our python code running the Bayesian update. Our servo supposedly has a range of 90 degrees, but in the interest of not pushing the servo right to its limit, we sweep through a range of 0 to 85 degrees.

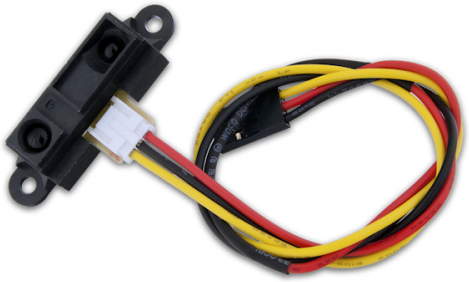

Figure 2: IR Range Finder similar to the one we used

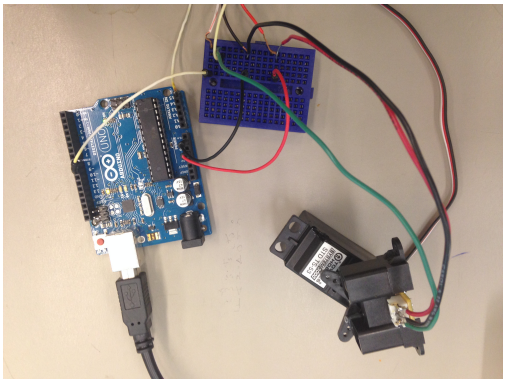

Figure 3: The testing system we used to collect data

2.2 From Sensor Data to Code

Before we can do any Bayesian, we have to learn a little about the error of our sensor. We characterize the error for the data by calculating the standard deviation and mean of the error at a series of distance points. The mean of the error as a function of distance is

$$mean = 0.0054x^2 - 0.4089x + 6.8076 \tag{1}$$

and the standard deviation of the error based on the distance from the sensor to an object is

$$standarddeviation = 0.1211x - 3.2346 \tag{2}$$

Here, we also assume that the error is normally distributed. This allows us to create a error function based on distance. We use this to calculate potential errors and likelihood of the data.
For this case study, we are representing all of our data and hypothesis as radius, angle pairs. It is also worth noting that we assume that there is no error in the angle of the sensor.

2.2.1 Hypotheses

Okay, enough about all the non-Bayesian parts of this project. In this case study, we have a separate suite of hypotheses for each angle. Within each suite our hypotheses correspond to the distance at which there might be an object. These hypotheses are defined in our code as

```
rs = np.linspace(minRange,maxRange,numPoints)
thetas = np.arange(0,maxAngle)
suites = []
for theta in thetas:
    suites.append(Lidar(rs))
```

Here Lidar is the class that represents our suite of hypotheses.

2.3 Updating our Hypotheses

2.3.1 Data for the Update

Back to non-Bayesian things for a second! The data we pass into our likelihood function is all the distances for one sweep of the sensor. This data is of the form of a large tuple containing (85) smaller tuples, where each of those consists of a measured distance and angle. As a note, the sensor does not respond well to distances that are outside its maximum range and often returns "Not a Number", or "NaN". When this happens, we simply set the measured distance to the maximum distance.

2.3.2 Updating Multiple Suites

Because we have a suite of hypotheses for each angle, we have to iterate through our data, and choose the appropriate angle suite to update for each piece of data. The code for doing this is

```
def UpdateSuites(suites, data):
    for measurement in data:
        r = measurement[0]
        theta = measurement[1]
        suite = suites[theta]
        suite.Update(r)
```

2.3.3 Likelihood Function

Our likelihood function is as follows. We use the error that we characterized above to compute the likelihood that we get a given measurement if we have an object at our hypothesis.

```
def Likelihood(self, data, hypo):
    #Computes the likelihood of the data under the hypothesis.
    #hypo: hypothetical distance
    #data: measured distance.

    error = data - hypo
    #Mean and STD calculated based on distance
    mean = 0.0054*hypo**2 - 0.4089*hypo + 6.8076
    std = 0.1211*hypo - 3.2346

    like = thinkbayes2.EvalNormalPdf(error, mean, std)
    if math.isnan(like):
        like = 5e-50

    return like
```

2.4 Plotting Things

After calculating likelihoods and updating our hypotheses, we would like to present our data in an understandable way. Funnily enough, we don't think thinkplot includes anything about plotting in polar coordinates so we made our own because polar plotting is the easiest method for us to communicate our data.

```
def plotPolar(suites):
    rsarray = np.linspace(minRange,maxRange,numPoints)
    thetasarray = np.arange(0,maxAngle)
    thetasradarray = np.radians(thetasarray)

    rs,thetasrads = np.meshgrid(thetasradarray, rsarray)
    likes = np.zeros((rsarray.size,thetasarray.size))

    for theta in thetasarray:
        currentSuite = suites[theta]
        for indexr,r in enumerate(rsarray):
            likes[indexr][theta] = currentSuite.Prob(r)

    fig, ax = plt.subplots(subplot_kw=dict(projection='polar'))
    ax.contourf(rs,thetasrads, likes)

    plt.show()
```

3 Our Results

To test how well our Bayesian system works, we scan a corner of the walls in an East Hall team room. Here is our testing setup.

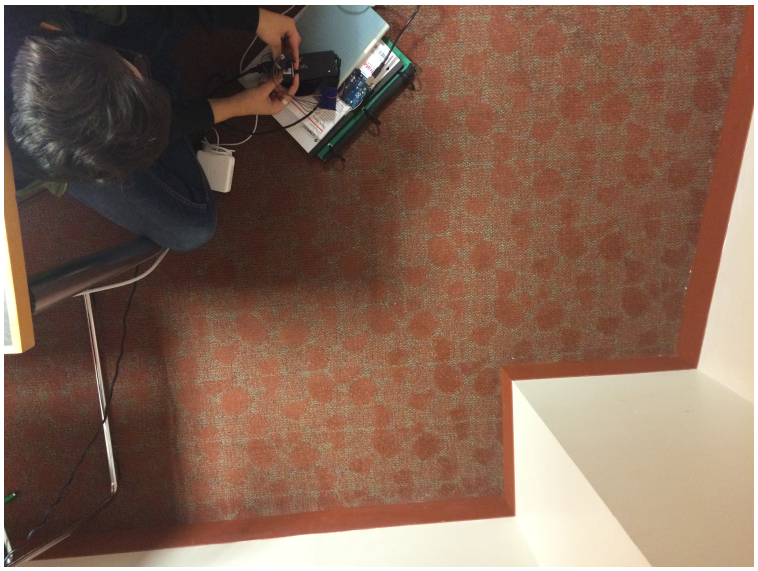

Figure 4: The testing setup

We scanned this area 5 times, and below are each of the outputs of our likelihood function.

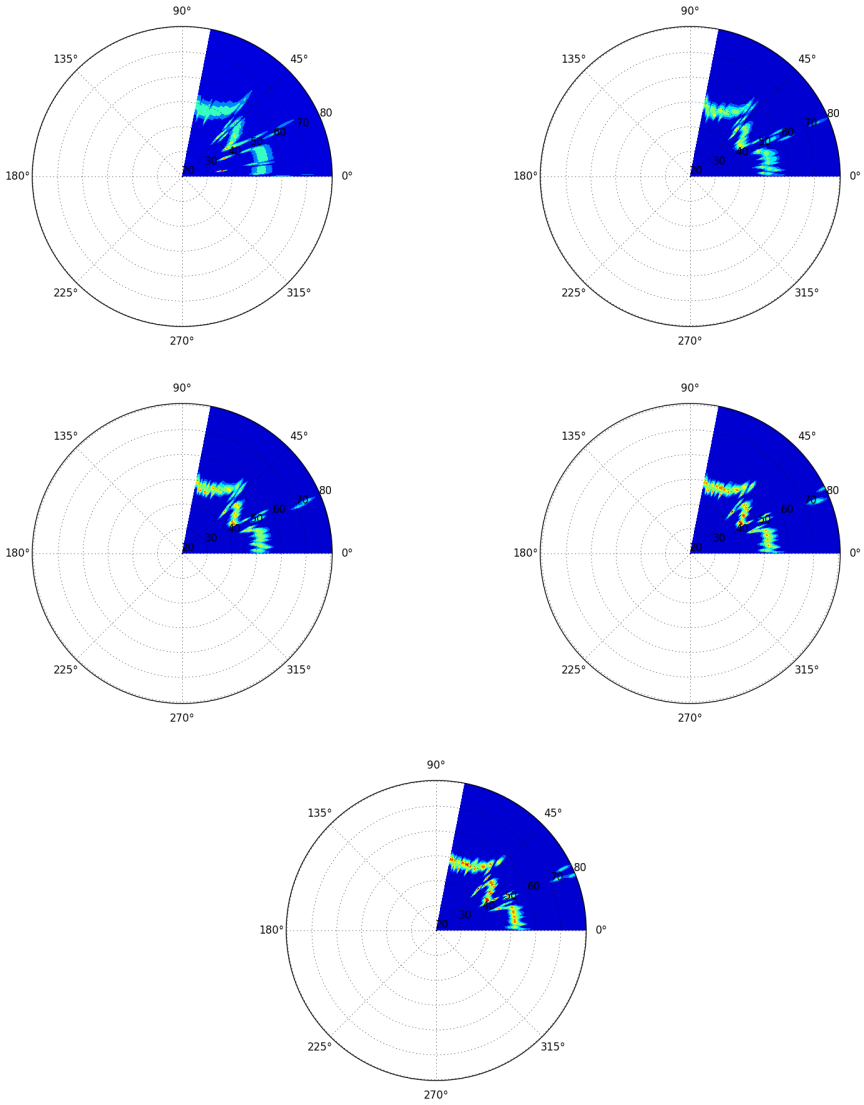

Figure 5: Our scans of a corner in an East Hall team room

Each time the general shape of the graph looks the same. Every time that we scan, we are more and more sure that there is a set of three corners as well as the precise location of these obstacles. There is still a little noise right around the sharp corners, but that's reasonable. Corners are hard.

4 Our Interpretation

Bayesian is the best!
Maybe. We do no compare this data to another method of calculating the probable location of the wall and are not completely convinced that this solution would best an average of multiple scans or even just the raw data. However, in a short number of scans we can relatively accurately map a difficult set of corners with

decent probability. This proves that Bayesian is, at very least, a usable solution to this question.

5 References

Our repo: CompBayesCaseStudy
`https://github.com/phiaseitz/CompBayesCaseStudy`

Reference for arduino control of IR sensor and motor, as well as communication with python: PoE Lab
`https://github.com/JPWS2013/PoE_F2013/tree/master/Lab_1`