MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Fall Semester, 2007

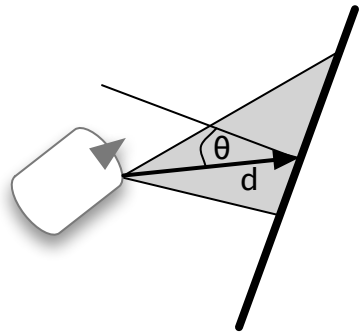
Assignment 14 (Part 2), Issued: Thursday, Dec. 6

Localization

Sensor model

One of the crucial components of the state estimation algorithm that we have been studying is the observation model. This model is supposed to tell us, for a given state, how likely the current observation is in that state, that is, $P(O|S)$. In the robot localization code, the robot state is represented as three indices into a three dimensional grid, representing quantized values of x, y and θ. For most of our examples, the grid has twenty values along each axis, so the indices go from 0 to 19.

We have produced data files that, for each entry in the state grid, store the distance d from each sonar sensor (along the ray emanating from the center of the sensor) to the nearest obstacle in the map. We've also stored the absolute value of the angle θ between the sonar beam (treated as an ideal line) and the normal to the surface; that is, the angle is 0 when the beam would be perpendicular to the surface and $\pi/2$ when the beam would be parallel to the surface, as shown in the figure below.



Note that this computation ignores a couple of important issues. First, it assumes that the robot is at the center of the grid cell, so these values are only really valid for that point in the cell (the ideal sonar readings would be somewhat different if the robot were not at the center of the cell; sometimes they can be very different, if a slight displacement means that the ideal sonar ray hits a different object that it did when the robot was at the center). Second, the actual sonar beam has a substantial "width" (shown in gray in the figure) which is being ignored in this model. The beam width means that sometimes the real reading is considerably shorter than the "ideal" one, because there is an object that actually closer, somewhere within the beam width. So these value should be taken with a grain of salt.

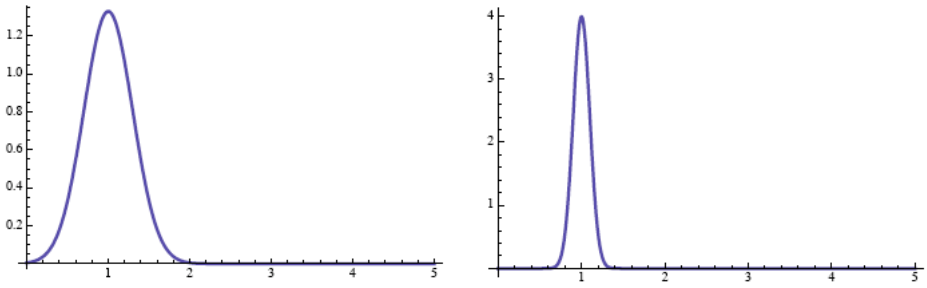
Given a robot state (a grid cell), we can look up the ideal distance from each sonar sensor to the nearest surface and the angle between the sonar beam and the surface. Let's focus on one sensor and ignore angle for now. Imagine that one of the sonar sensors has returned distance d and we

want to know how likely that reading is in state (ix, iy, it) – these are indices into the state grid. We can look up the ideal distance for that grid cell: $\hat{d}$. Now, we need to assign a probability $P(d|\hat{d})$. If we believed that there was no error in the sensor or in the map or anywhere else, we could say that the probability is 1 if $d = \hat{d}$ and 0 otherwise. But, this is not a very good model in practice.

We could instead assume that $\hat{d}$ is the mean value that we would expect to see but that there is some dispersion around that value. For example, we could assume that the probability of reading a particular value d from the sensor is distributed as a Gaussian whose mean is $\hat{d}$ and whose standard deviation is a known constant, σ, say 0.3. The Gaussian is defined as follows:

$$g(d, \hat{d}, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{(d-\hat{d})^2/2\sigma^2}$$

This is the familiar "bell curve", its maximum value is at the mean and it drops quickly so that about 95% of the area is within two standard deviations of the mean. The figure on the left shows a Gaussian with mean of 1 and sigma of 0.3; the one on the right has the same mean but with a sigma of 0.1.



The initial multiplier of the Gaussian distribution has been chosen so that the integral of the curve is exactly 1. Just as we want the elements of a discrete probability distribution to add to 1, we want continuous probability distributions to integrate to 1.

Note that the values of a Gaussian with small variance (the smaller the variance, the narrower and higher the peak) at a single point may be greater than 1, so it's clear the values of the Gaussian at a point are not probabilities; they are called *densities*. When we say that the probability of a sonar reading is distributed as a Gaussian what we mean that the probability that a value lies within some small range $[d - \delta, d + \delta]$ is the area under the curve at that range. For small δ, this is approximately the density at d times a small constant (2δ). So, we will play fast and loose and simply use the density values themselves (ignoring the scaling factor) and rely on the normalization step in the state update to get everything into the right range.

The net result is that we can imagine constructing a simple sensor model for a single sonar as follows:

```
def obsProb(obsDist, (trueDist, angle)):
    return gaussian(obsDist, trueDist, sonarSigma)
```

The call to `gaussian` computes the Gaussian density at `obsDist` for a Gaussian whose mean is `trueDist` and whose standard deviation is `sonarSigma`. So, we see that the probability is maximal when `obsDist == trueDist` and drops rapidly as the difference increases. In this model, there is no dependence on the angle between the beam and the surface—this is almost true for surfaces covered in bubble-wrap and wildly untrue for anything else (why do you think this is?).

Note that `obsProb` computes $P(O|S)$ and it must be the case that if we integrate the resulting densities over all possible observations (for a given state), then the integral must be 1. Since we're returning a gaussian density, this property holds and we're on solid ground.

There are many situations when the process generating the observations has more than one significantly likely result (mode). In that case, a single Gaussian distribution is not a good model. Consider, for example, the behavior of the real sonars. If something is too far away (or the angle too steep) then the sensor will not detect a return "echo" and simply return some large value determined by the settings in the robot firmware—call it `maxDist` (2.465 on some of our robots and 5.0 in others). It's clear that if we were to use a single Gaussian to build our observation model, we would have a very bad model; it would assign almost 0 probability to any of the large sonar values. In fact, those readings are "expected" in the case where the `trueDist` is larger than the effective range of the sensors.

We could partition the range of `trueDist` into a region where our simple model above holds and a region where the maximum reading is the expected value.

```
def obsProb(obsDist, (trueDist, angle)):
    if trueDist <= 1.5:
       return gaussian(obsDist, trueDist, sonarSigma)
    else:
       return gaussian(obsDist, maxDist, sonarSigma)
```

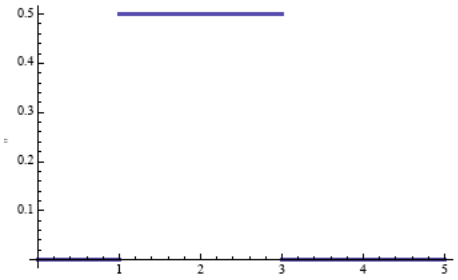
Note that as long as each distribution integrates to 1 over all the observations, this is fine, since $P(O|S)$ only needs to integrate to 1 for each state and each state will have exactly one of these legal distributions apply.

However, in some cases, we have states where we could get a sonar reflection or not, meaning that we could get a "good" reading or a "bad" one (the maximum value). Imagine that we think that (based on the `trueDist`) we expect a 30% chance of a good reading and 70% chance of a bad one. We can produce a *mixture distribution* as a weighted combination of two other distributions, for example,

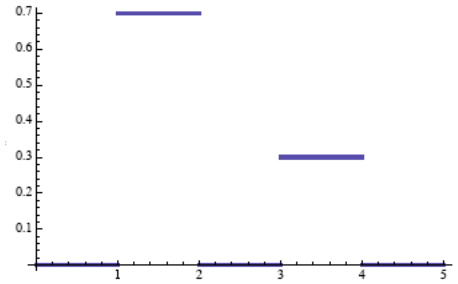
```
  def obsProb(obsDist, (trueDist, angle)):
    prob = 0.3
    return prob*gaussian(obsDist, trueDist, sonarSigma)+\
           (1-prob)*gaussian(obsDist, maxDist, sonarSigma)
```

This mixture distribution is a legal distribution and integrates to 1 (over the observations, i.e. all possible values of `obsDist`). In practice, we could make the value of `prob` be a function of the `trueDist` and/or the `angle`. Here is a mix of two Gaussians, one with mean 1 and sigma of 0.3 (weight = 0.8) and the other with mean 4 and sigma of 0.2 (weight = 0.2).

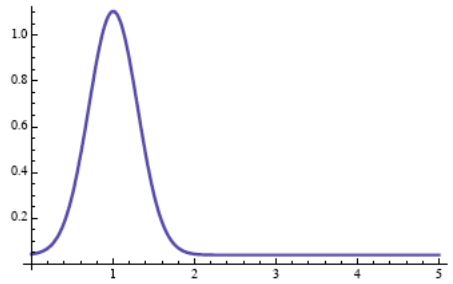
Instead of using the Gaussian distribution as above, which assumes the mean value has a high probability, we could instead use *uniform* distributions to model the sonar behavior. This assumes that we are equally likely to get a reading within some specified range of values. A uniform distribution is defined by a low value (L) and a high value (H). The probability of a value x under this distribution is $1/(H - L)$ for $L \leq x \leq H$ and 0 otherwise (see figure below).



This clearly integrates to 1 We can use such a distribution wherever we used a Gaussian above and we can even mix Gaussians and uniforms just as we mixed Gaussians. Below is a mix of two uniforms, one from 1 to 2 (weight = 0.7) and the other from 3 to 4 (weight = 0.3).



Below is a mix of a Gaussian with mean at 1 (weight = 0.8) and sigma of 0.3 with a uniform from 0 to 5 (weight = 0.2).



Note that this is a classic primitives, combinations and abstraction situation.

Up to now, we have considered only one sonar sensor but the robots have eight sensors. We will assume (as a computationally convenient approximation) that the sensor values are *independent* given the state and that, therefore, we can simply multiply the probabilities (densities) of the individual readings to obtain the probability of the combination of sonar values. If you look at the code, you'll notice that instead of multiplying the probabilities (which would produce very small numbers and lead to numerical underflow problems in our computations) we take the logarithms of the probabilities and add them.

Question 1. Run `dataBrain.py` to characterize the sonar sensors on your robot (with bubble-wrap walls), that is, you should understand the behavior of the sensors well enough to build a sensor model. `dataBrain.py` prints average (and max-min range) of the values of a sonar sensor (chosen by the value of `sonarI` which is set to 3 initially). The values are grouped into two bins by comparing them to a threshold (set to 2.4 by default). The bin with lower values is called the "inliers" and the other bin is called the "outliers". If you have time, you might want to characterize the angle dependence when not using bubble wrap.

Question 2. Build one or more observation model for a sensor; define an `obsProb` that assigns high probability to likely observations. The file `sonarModel.py` has the default definition for `obsProb`. If you have time, you might want to incorporate the angle dependence when not using bubble wrap.

Question 3. Test your models numerically for the types of readings that you saw in your data gathering. This is just to verify that the code does what you expect it to; we will carry out more extensive testing below. Show a situation that you found in your testing where your model is significantly better than the default model.

Checkpoint: Thursday 11:00 AM

- Implementation and testing of sensor models.

Quality of Localization

We will want to know how well our localization method is working, in order to evaluate how changes in various aspects of our model affect localization performance. So, first, we have to decide what would be a good measure of how well the localization is working, and then implement a method that computes it. Assuming we knew the true pose of the robot, we have to think of a way to measure the quality of the current belief state, given the true pose.

If `m` is set to be an instance of `GridStateEstimator` for the current map, you can find the logarithm of the probability associated with a real-valued pose by doing

```
m.getBeliefAtPose(pose)
```

Or, if you want to get the log probability associated with a set of grid indices, you can do

```
m.getBeliefAtIndices((2, 2, 19))
```

To convert these numbers to probability, use `math.exp`. If you want to use it, you'll have to put `import math` at the top of your file. You can also convert a set of grid indices into a pose using `m.indicesToPose`, or a pose into a set of grid indices using `m.poseToIndices`.

The robot's opinion of the grid square with the highest probability (i.e., the tuple `(ix, iy, ith)` with the highest probability of containing the robot) can be found with:

```
m.bestPoseIndices
```

Another instance variable that you might find useful is

```
m.kBestPoses
```

It returns a tuple of the `k` most likely poses, in the format `((lp1, (x1, y1, th1)), (lp2, (x2, y2, th2)), ...)`, where `lp1` is the log of the probability of the robot being in the pose with *indices* `(x1, y1, th1)`. By default, it keeps the 5 best poses, but you can change this value by changing the argument in the initializer for `GridStateEstimator`.

In order to do repeatable experiments, it will be useful to gather data logs from simulated or real robot runs. To gather a log of the sensor data that the robot gathers as it's moving around, add `writeLog(labDirectory+"robot.log")` to your brain. This will keep track of all the sensor and odometry readings that your robot got. Now, you can run your program again, but replace that line with `readLog(labDirectory+"robot.log")`; the robot won't move, and instead it will "hallucinate" the sensor readings that it had on the previous run whenever you call `sonarDistances()` or `pose()`. This allows you to sit quietly at your desk and print out values, or stare at the graphics windows to really understand what was going on, rather than trying to figure it out as you chase your robot around the lab. When you are replaying a log, you'll have to choose a simulated world, and the robot will run around, insensate, banging into things. Just don't watch.

We have gathered two logs with two different robots, they are available in the `lab14` folder. `robot_2.465.log` and `robot_5.0.log` each contain a log for the robot moving in the `boxWorld`. We have also generated an *approximately correct* set of pose indices in the file `robotTruePoses.py`, which is imported by `WanderEstBrain.py`. This file defines two lists of pose indices `robot_2465` and `robot_50`. The version of `WanderEstBrain.py` we have given you sets `robot_true_poses` at the top of the file to one of these and it accesses it inside `beliefUpdateEveryN`. You should modify the code so that you compute a quality metric at every update step.

When changing between logs, three things have to be kept consistent:

- **The value of `robot_poses` at the top of `WanderEstBrain.py`.**
- **The name of the log file in the `readLog` command in `WanderEstBrain.py`.**
- **The value of `maxReading` set in sonarModel.py.**

So, `robot_poses` = `robot_2465` goes with `robot_2.465.log` and with `maxReading = 2.465`. On the other hand, `robot_poses` = `robot_50` goes with `robot_5.0.log` and with `maxReading = 5.0`.

We have added some printing to the brain. On each cycle, it will print something like the following:

```
Best pose: (-1.1206372327277521, (14, 4, 0))
```

This says that the most likely grid indices for the robot are $(14, 4, 0)$, and that the log of the probability that the robot is at that pose is about -1.12.

```
o= 2.465 t= 2.26500004698 a= 0.000203673205103 p= 0.01
o= 2.465 t= 1.38869419849 a= 1.13 p= 0.0118228301207
o= 0.741 t= 0.734210468952 a= 0.698 p= 1.14669435613
o= 0.925 t= 0.55649087171 a= 0.262 p= 0.544695259091
o= 0.725 t= 0.55649087171 a= 0.262 p= 0.98105579419
o= 0.686 t= 0.734210468952 a= 0.698 p= 1.13239855779
o= 0.736 t= 0.508617647354 a= 0.440796326795 p= 0.863118387399
o= 0.533 t= 0.435000009023 a= 0.000203673205103 p= 1.08791100707
```

This says, for each of the 8 sonar readings (in order from left to right as the robot faces forward), the observed reading, the "true" reading (that is, the ideal reading that a robot in the center of

the grid square associated with true pose would get), the angle of the true sonar reading with the surface it is bouncing off of, and the density of the observation given the ideal reading and the angle.

```
P(o | truePose) =  -9.47127052638
```

This is just the sum of the logs of the densities of each of the observations.

```
true indices= (15, 4, 0)
```

Finally, this is the "true" pose.

You can use this listing to look for situations where your model is assigning a very low density to a reading; try to understand whether that truly is an unlikely reading or whether you need to change your model.

Question 4. Think of two ways to measure how well the localization is working, and augment `WanderEstBrain.py` to compute one of these measures and print it out. Talk over your planned measure with your LA.

Question 5. Implement your measure and arrange it so that your brain prints it out every time the belief state is updated. You might want to print average value of your measure over time, as well. In order to compare localization performance using different sensor models, you should try different methods on each log file, first using the default sensor model and then your improved models. Do you see a noticeable improvement from using your sensor models?

Question 6. If you see some common situations leading to very low probability for the observations, consider extending your model so that those situations are assigned higher probability (since they are obviously more probable than you thought).

Checkpoint: Thursday 12:00 Noon

- Discuss your proposed metrics with your LA.
- Implementation of a metric for localization performance.
- Testing of sensor models.

On the robot

Using a robot and the available playpens (we'll have one corresponding to `boxWorld`, which can be converted to `emptySquare` by removing the box), test the localization code using your sensor model (use `XYThEstBrain.py`). Keep some track of the time it takes to reach the destination (say in number of brain steps). Compare to the behavior with the default sensor model (we will have that posted in the lab).

Question 7. Does changing the sensor model make the estimation work better? How much better?

Checkpoint: Thursday 1:00 PM

- Tests on the robot, with improved sensor model.

Explorations

Here are some ideas for extensions:

- Can you look at the whole belief state and come up with a way of avoiding situations when the robot thinks it has reached the goal state but it really hasn't.
- See if you can do localization in an environment with no bubble wrap on the walls.