Foundations of Robotics, Fall 2017

Coding Homework 4: Uncertainty

(CS 4750: 50 points, CS 5750: 60 points)

Due at 3:35pm, Tues, 5 Dec

Objective:

Demonstrate understanding of methods of handling uncertainty in robotics applications.

Instructions:

This assignment can be discussed as a group of arbitrary size. You may work with up to one partner on this assignment, but each student is responsible for writing and submitting a separate solution, and **no student should see another student's solution**. Include in a comment at the top of each file your name and the names of all classmates you discussed any part of the homework with.

Use the simulator tool to run the provided code for the assignment. **For this assignment, you will always need to run `./simulator-tool run hw4 main` to run the framework code.** We suggest using the timeout option (see Piazza post @424) so that the simulator runs for long enough. Also note that **the simulation will now auto-play**, so you will not need to press play on the simulator after running the main framework code.

If you encounter difficulty using the simulator tool, please contact the course staff. Note that you can also run `./simulator-tool --help` to get a listing of commands and `./simulator-tool <command> --help` to get more information on the usage of a specific command.

Assignment:

Implement code to solve the following problems, using the provided simulator framework. Start by making a `catkin` workspace for your code. Create a new ROS package for this project in the workspace (call it "hw4"); write your code for each problem in a corresponding file in the package (named according to the convention "pX.py", where X is the problem number).

Submission Format:

Compress your `hw4` package into a .tar.gz archive with the command `tar -czvf solutions.tar.gz hw4`. Double-check that your submission is a single, correctly-structured ROS package and that it builds. **We reserve the right to penalize for incorrectly structured submissions.**

NumPy:

You should use NumPy for this assignment. NumPy is a Python package that allows you to perform matrix operations conveniently and efficiently. If you haven't used it before, please see Appendix A for a quick guide to help get you started.

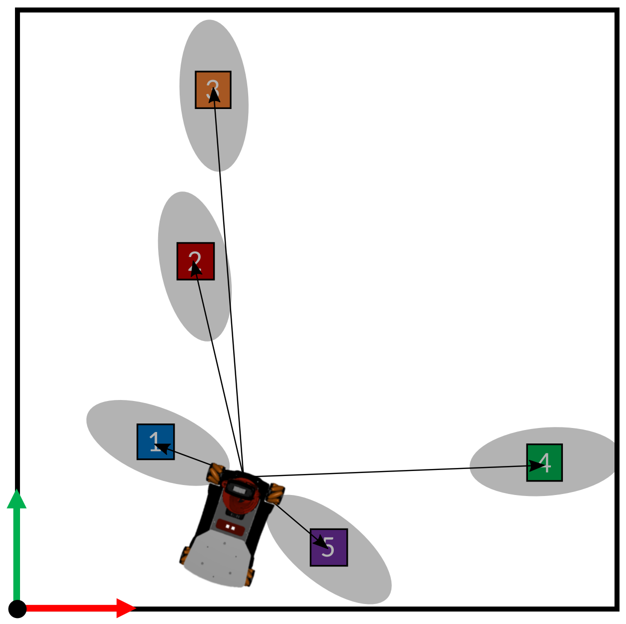

Figure 1: The youBot sensing the blocks with error. The ellipse around each block represents the region in which 95% of position measurements for that block will lie.

Background (read this first!)

In this assignment, we explore the application of explosive ordnance disposal (EOD), in which robots are used to detect, defuse, or detonate bombs. This project is based on the same structure as Coding Project 3. In this project, we have reason to believe that one of the blocks conceals a bomb, whereas the other four blocks are harmless. The sensor technology for detecting and localizing a bomb is unreliable, and it may result in false positives (detecting bombs that are not there) or false negatives (failing to detect bombs that are there). You must use techniques from probability and statistics to localize the blocks and correctly ascertain which one conceals the bomb.

1 Locating the Blocks (30 points)

To be able to do anything about a bomb in the set of blocks, you must be able to locate the blocks. Unfortunately, you no longer have an oracle that gives you the exact positions of the blocks; rather, you must rely on the youBot's noisy sensor for measuring block positions. When queried for the position of a specific block, the position sensor service (`/vrep/youbot/position_sensor`, which has type `foundations_hw4/GetPosition.srv`) will respond with (1) a point drawn from a multivariate Gaussian distribution whose mean is the true location of the block, and (2) the covariance matrix of the distribution. For a covariance matrix

$$C = \begin{bmatrix} C_{XX} & C_{XY} \\ C_{YX} & C_{YY} \end{bmatrix},$$

the service response will include the following array in its `cov` field: $\begin{bmatrix} C_{XX} & C_{XY} & C_{YX} & C_{YY} \end{bmatrix}$. As illustrated by Figure 1, the larger axis of variance of the distribution is aligned with the vector between the robot's position and the block's position. The reason for this is that the sensors are fairly accurate in bearing (i.e. direction to the block) but not very accurate in range (i.e. distance to the block). Consequently, a more accurate estimate of the block positions can be obtained by repeatedly taking measurements from different locations in the workspace.

To combine the series of measurements that you take of each block, you will implement a **Kalman filter** (see Section 7.8 of the course notes). Part of the problem involves specifying the matrices of the Kalman filter. To determine what matrices to use, it helps to think of A_t as the *state-transition model*, B_t as the *control-input model*, and C_t as the *measurement model.* Recall also that R_t is the covariance matrix associated with the state transition noise and that Q_t is the covariance matrix associated with the measurement noise. Given this information, consider the environment: Do the blocks move? Can you take actions to change their state, assuming that you never run into a block? What information about the covariance is available to you via the position sensor? Use the answers to these questions to select the matrices for your Kalman filter implementation.

The service that you implement for this problem (call it `p1`) should compute one iteration of the Kalman filter algorithm for each of the five blocks when called (i.e. update the state for each block once) and should not move the robot. It should have type `foundations_hw4/KalmanUpdate`. Note that the response has two fields: an `estimates` array that should contain the current belief of the position of each block, and a `confidence` array that should contain the current variance for each belief. Each element of the `confidence` array is a `geometry_msgs/Point` object; the `x` component of this object should contain the variance in the `x` component of the estimate, and the `y` component of the object should contain the variance in the `y` component of the estimate.

To test your service, replace our implementation of `p1` with yours in `bomb.launch`, launch the file with `roslaunch`, and check your accuracy via the visualization window as the robot drives around and updates its beliefs. If you want, you can also modify `bomb.launch` to run your own driving controller to see if you can make your measurements converge faster. You will be evaluated on this problem by comparison between our implementation (provided for reference as the `p1` binary) and yours. Specifically, we will look at the error between our service and yours for the same set of parameters and sample points.

2 Locating the Bomb (30 points)

Now that you can locate the blocks, you need to identify which block contains the bomb without risking an explosion. Since the youBot's bomb sensor is also noisy, taking only one reading will likely not be sufficient to confidently select a block. In this problem, you will use **Bayesian inference** to combine your individual beliefs about the blocks in the workspace into a probability distribution that represents your belief about the location of the bomb (see Figure 2).

Let x_R be the position of the robot, and let x_i be the location of the i-th block. Define B_i to be a Bernoulli random variable such that

$$B_i = \begin{cases} 1 & \text{if the } i\text{-th block contains the bomb,} \\ 0 & \text{if the } i\text{-th block does not contain the bomb.} \end{cases} \tag{1}$$

Finally, define Z_i to be the boolean measurement of whether the i-th block is a bomb. The accuracy of the measurement is negatively affected by a larger distance between the robot and the block. To use the youBot's bomb sensor on a specific block, call the service `/vrep/youbot/bomb_sensor`, which will return Z_i as well as the distance from the youBot to the block (for the service definition,

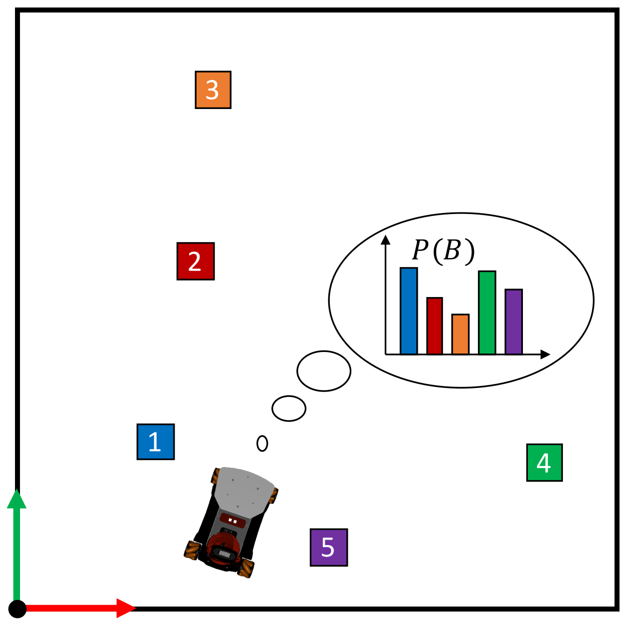

Figure 2: The youBot's belief about the location of the bomb is represented as a probability distribution over all possible bomb locations.

see `foundations_hw4/BombTest.srv`). You may assume that consecutive sensor readings for a given block are conditionally independent.

After obtaining a measurement Z_i for the i-th block from a distance of $d_i = ||x_i - x_R||$, you can use Bayes' rule to model the belief about whether the block contains a bomb as

$$bel(B_i) = \mathrm{P}(B_i|Z_i, d_i) = \frac{\mathrm{P}(Z_i|B_i, d_i)\mathrm{P}(B_i|d_i)}{\mathrm{P}(Z_i|d_i)}. \tag{2}$$

The event that the i-th block contains the bomb is independent of the distance from which a sensor reading occurs and therefore $\mathrm{P}(B_i|d_i) = \mathrm{P}(B_i)$. Thus, we may rewrite (2) as

$$bel(B_i) = \frac{\mathrm{P}(Z_i|B_i, d_i)\mathrm{P}(B_i)}{\mathrm{P}(Z_i|d_i)}, \tag{3}$$

where $\mathrm{P}(B_i)$ is the prior probability of of the i-th block containing the bomb. As far as you know, before taking any readings, each block has an equal probability of concealing the bomb. Use this information to to define your initial prior probability distribution.

Next, by the law of total probability, we can rewrite the denominator of (3) as follows:

$$\mathrm{P}(Z_i|d_i) = \mathrm{P}(Z_i|B_i = 1, d_i)\mathrm{P}(B_i = 1) + \mathrm{P}(Z_i|B_i = 0, d_i)\mathrm{P}(B_i = 0). \tag{4}$$

In order to use (3) and (4) to infer the bomb location, you need a model of the bomb sensor. The following model incorporates the sensor's dependency on the distance to the measured block:

$$\mathrm{P}(Z_i = 1|B_i = 1, d_i) = \frac{4 - \tanh(3(d_i - 1.5)) - \tanh(3)}{4}, \tag{5}$$

$$\mathrm{P}(Z_i = 1|B_i = 0, d_i) = \frac{\tanh(3(d_i - 1.5)) + \tanh(3)}{4}. \tag{6}$$

Equation (5) represents the case where the i-th block contains the bomb and the sensor correctly detects a bomb in the i-th block. (To determine the probability where the i-th block contains the bomb and the sensor incorrectly does *not* detect a bomb, simply compute $1 - \mathrm{P}(Z_i = 1 | B_i = 1, d_i)$). As d_i decreases, the measurement Z_i will be 1 more consistently, indicating higher confidence that the block has the bomb. On the other hand, if d_i is large enough, the sensor reading of a block is no better than flipping a coin, and the probability approaches 0.5. Equation (6) follows a similar logic but for the case that the i-th block does not have the bomb. Try plotting both of these functions to better understand their behavior.

Once you have computed an updated belief about each block, you need to renormalize them, as

$$\mathrm{P}(B_i) = \frac{bel(B_i)}{\sum_{j=1}^{5} bel(B_j)}. \tag{7}$$

The service that you implement for this problem (call it `p2`) should return the distribution given by (7) after every call and should be of type `foundations_hw4/FindBomb` (check the service definition to see what data structure to use for the distribution). Similar to your `p1` service, your `p2` service should update the belief state once per call and should not move the robot. Like before, you may test your service by replacing our implementation of `p2` in `bomb.launch` with your implementation, launching the file with `roslaunch`, and checking your accuracy via the visualization window. Your solution will be evaluated on a fixed arrangement of blocks with our driving code. Specifically, we will run both your solution and our solution (provided for reference as the `p2` binary) for a fixed number of iterations and then score your service based on whether your distribution matches ours. We will repeat this process five times and then average the scores.

A NumPy Guide

Importing NumPy First, make sure to import NumPy: `import numpy as np`

Constructing an array A NumPy array can be any number of dimensions. To create a n-dimensional array, you can use `np.array()`. For example, `a_1D = np.array([1.,2.,3.])` creates a 1D array of floats, and `a_2D = np.array([[2,2],[0,1]])` creates a 2D array of ints where the first row is [2, 2] and the second row is [0, 1].

Special arrays NumPy includes some useful functions to create special arrays. One such function is `np.zeros(shape)`, which returns a n-dimensional array of the specified shape (a tuple specifying each of the n dimensions). For instance, `np.zeros((3,2))` returns a 3×2 array of 0s. Another useful function is `np.identity(n)`, which returns a $n \times n$ identity matrix.

Constructing a matrix A NumPy matrix is just a special 2D NumPy array. Use `np.matrix()` to create a matrix. For example, `m = np.matrix([[2,2],[0,1]])` creates a matrix with the same values as `1_2D` from above. However, while they may be very similar and you can do the same things with both, the matrix and array types are not exactly the same. Besides the fact that a matrix can only be 2D, an important difference is in matrix multiplication, as described next.

Matrix multiplication To perform matrix multiplication with the matrix type, you can simply use the * operator. For example, we can multiply the matrix `m` we defined above by doing `m*m`. If you're working with 2D matrices, the matrix type is convenient since matrix multiplication is very intuitive. **Be careful:** In this case, the * operator performs matrix multiplication like you might expect. However, if you use n-dimensional arrays instead of matrices, the * operator will multiply the arrays element-wise. You can do normal matrix multiplication with arrays by using NumPy's `dot()` function. For example, `a_2D.dot(a_2D)`, or alternatively `np.dot(a_2D,a_2D)`, will return the same matrix product as `m*m`, and these are NOT equal to `a_2D*a_2D`.

Inverse and transpose Getting the transpose of a matrix/array with NumPy is easy. You just need to use `.T` (e.g. `m.T` or `a_2D.T`). You can get the inverse of a square matrix/2D array using `np.linalg.inv()`. For example: `np.linalg.inv(m)`.

Indexing You can access specific parts of an array/matrix by indexing particular cells like you would expect (e.g. `m[1,0]` would return 0). You can also use slicing to get subsets of a matrix/array. For example, `m[:,1]` returns the second column of `m`, where the colon selects all of the rows and the 1 specifies the column we want.

For a more in-depth NumPy guide, you can read the quickstart tutorial here.