UDACITY

‹ Return to Classroom

Landmark Detection & Tracking (SLAM)

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

CONGRATULATIONS. Your project passed all the requirements in the rubric.
Thanks for your time and efforts. Keep working hard and stay motivated. Good luck for your next projects.

I liked the way you programmed the SLAM algorithm and your programming experiments.

If this is your final project, congratulations again! You are about to graduate.
Go to the main page of this nanodegree and press the GRADUATE button.

Interesting application: SLAM is an amazing algorithm, which can be extended to 3D and 4D (with real time).
For example, Boston Dynamics used point clouds in 3D, sensed by their robots, to simultaneously localize and map 3D environments, in real time:
https://www.youtube.com/watch?v=Ve9kWX_KXus&t=65

To further investigate on SLAM algorithms, you can read the book of Prof. Sebastian Thrun:
http://www.probabilistic-robotics.org/

This video could tell you where SLAM and LIDAR technologies are heading.
New LiDAR: Driverless cars are about to get a whole lot better at seeing the world | WIRED Originals
https://www.youtube.com/watch?v=ibIzthKKZWY

`robot_class.py`: Implementation of `sense`

Implement the `sense` function to complete the robot class found in the `robot_class.py` file. This implementation should account for a given amount of `measurement_noise` and the `measurement_range` of the robot. This function should return a list of values that reflect the measured distance (dx, dy) between the robot's position and any landmarks it sees. One item in the returned list has the format: `[landmark_index, dx, dy]`.

Very good. **Your implementation of `sense` is correct.**

The condition to add `[landmark_index, dx, dy]` is something like:
`if abs(dx) < self.measurement_range and abs(dy) < self.measurement_range:`
This condition guarantees that measurements are considered only if they are within the measurement range, making the simulation more realistic and limited.
In other words, cheating is avoided with this condition.

Notebook 3: Implementation of `initialize_constraints`

Initialize the array `omega` and vector `xi` such that any unknown values are `0` the size of these should vary with the given `world_size`, `num_landmarks`, and time step, `N`, parameters.

The constraints are correctly initialized.
The dimensions of `omega` and `xi` vary with the given parameters: `world_size`, `num_landmarks`, and time step `N`.

```
def initialize_constraints(N, num_landmarks, world_size):
    ''' This function takes in a number of time steps N, number of landmarks, and a
 world_size,
        and returns initialized constraint matrices, omega and xi.'''

    ## Recommended: Define and store the size (rows/cols) of the constraint matrix i
n a variable
    cm_size = 2*(N+num_landmarks)   # time steps N is same as number of poses includ
ing initial position

    ## TODO: Define the constraint matrix, Omega, with two initial "strength" values
    ## for the initial x, y location of our robot
    omega = np.zeros((cm_size, cm_size))

    # initial pos = 0, x0 index is 0, y0 index is 1 (x0 index + 1)
    pos_index=0
    omega[pos_index,pos_index] = 1   #x0
    omega[pos_index+1, pos_index+1] = 1   #y0

    ## TODO: Define the constraint *vector*, xi
    ## you can assume that the robot starts out in the middle of the world with 100%

confidence
    xi = np.zeros(cm_size)
    xi[pos_index] = world_size / 2.0   #starting x pos
    xi[pos_index+1] = world_size / 2.0   #starting y pos

    return omega, xi
```

Notebook 3: Implementation of `slam`

The values in the constraint matrices should be affected by sensor measurements *and* these updates should account for uncertainty in sensing.

Your implementation is correct.
The values in the constraint matrices are affected by sensor measurements and these updates account for uncertainty in sensing.
Congratulations.

EXCELLENT implementation. Your abstract function is very useful to make your code more concise and less error-prone.

```
def updateOmegaXi(omega, xi, pos1_index, pos2_index, value, noise):
    omega[pos1_index, pos1_index] += 1/noise
    omega[pos1_index, pos2_index] += -1/noise
    xi[pos1_index] += -value/noise

    omega[pos2_index, pos1_index] += -1/noise
    omega[pos2_index, pos2_index] += 1/noise
    xi[pos2_index] += value/noise

    return omega, xi
```

The values in the constraint matrices should be affected by motion `(dx, dy)` *and* these updates should account for uncertainty in motion.

Your implementation is correct.
The values in the constraint matrices are affected by sensor measurements and these updates account for uncertainty in motion.
Congratulations.

EXCELLENT implementation. Your abstract function is very useful to make your code more concise and less error-prone.

The values in `mu` will be the x, y positions of the robot over time and the estimated locations of landmarks in the world. `mu` is calculated with the constraint matrices `omega^(-1)*xi`.

`mu` is computed in the correct way: `omega^(-1)*xi`
`mu` is the x, y positions of the robot over time and the estimated locations of landmarks in the world.

```
    mu = np.linalg.inv(omega) @ xi

    return mu, omega, xi
```

Compare the `slam`-estimated and *true* final pose of the robot; answer why these values might be different.

Very good answers and very good results. **You passed this part of the rubric.**
You actually made the experiment of varying the variable `N`. **KUDOS.**

Question: How far away is your final pose (as estimated by slam) compared to the true final pose? Why do you think these poses are different?
You can find the true value of the final pose in one of the first cells where make_data was called. You may also want to look at the true landmark locations and compare them to those that were estimated by slam. Ask yourself: what do you think would happen if we moved and sensed more (increased N)? Or if we had lower/higher noise parameters.

Answer: The true final pose is Robot: [x=69.16726 y=29.07029]. The final pose as estimated by SLAM is Last pose: (70.25390232177422, 28.356078522900077). As you can see, they are quite close. The small difference is due to sensor and motion noise. If we reduced the noise values to zero, the two positions would match 100%.

I tried another experiment (see above cell) where I increased the N to 600. True pose was Robot: [x=96.47294 y=15.83472] and estimated final pose was [96.741, 12.290]. While the x value is close, you can clearly see that the y value is way off. This is because each movement introduces additional errors in the position estimates due to the noise from the sensors and motion. The estimates for consequent poses gets worse with each step.

There are two provided test_data cases, test your implementation of slam on them and see if the result matches.

Your results and the results of the 2 testcases are similar,
which guarantees your implementation of SLAM is correct. **Congratulations.**

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review

START