1. Robot Moving and Sensing

May 19, 2020

1 Robot Class

In this project, we'll be localizing a robot in a 2D grid world. The basis for simultaneous localization and mapping (SLAM) is to gather information from a robot's sensors and motions over time, and then use information about measurements and motion to re-construct a map of the world.

1.0.1 Uncertainty

As you've learned, robot motion and sensors have some uncertainty associated with them. For example, imagine a car driving up hill and down hill; the speedometer reading will likely overestimate the speed of the car going up hill and underestimate the speed of the car going down hill because it cannot perfectly account for gravity. Similarly, we cannot perfectly predict the *motion* of a robot. A robot is likely to slightly overshoot or undershoot a target location.

In this notebook, we'll look at the `robot` class that is *partially* given to you for the upcoming SLAM notebook. First, we'll create a robot and move it around a 2D grid world. Then, **you'll be tasked with defining a `sense` function for this robot that allows it to sense landmarks in a given world**! It's important that you understand how this robot moves, senses, and how it keeps track of different landmarks that it sees in a 2D grid world, so that you can work with it's movement and sensor data.

Before we start analyzing robot motion, let's load in our resources and define the `robot` class. You can see that this class initializes the robot's position and adds measures of uncertainty for motion. You'll also see a `sense()` function which is not yet implemented, and you will learn more about that later in this notebook.

```
In [3]: # import some resources
        import numpy as np

        import matplotlib.pyplot as plt
        import random
        %matplotlib inline
```

```
In [4]: # the robot class
        class robot:
```

```
# --------
# init:
#   creates a robot with the specified parameters and initializes
#   the location (self.x, self.y) to the center of the world
#
def __init__(self, world_size = 100.0, measurement_range = 30.0,
             motion_noise = 1.0, measurement_noise = 1.0):
    self.measurement_noise = 0.0
    self.world_size = world_size
    self.measurement_range = measurement_range
    self.x = world_size / 2.0
    self.y = world_size / 2.0
    self.motion_noise = motion_noise
    self.measurement_noise = measurement_noise
    self.landmarks = []
    self.num_landmarks = 0


# returns a positive, random float
def rand(self):
    return random.random() * 2.0 - 1.0


# --------
# move: attempts to move robot by dx, dy. If outside world
#       boundary, then the move does nothing and instead returns failure
#
def move(self, dx, dy):

    x = self.x + dx + self.rand() * self.motion_noise
    y = self.y + dy + self.rand() * self.motion_noise

    if x < 0.0 or x > self.world_size or y < 0.0 or y > self.world_size:
        return False
    else:
        self.x = x
        self.y = y
        return True


# --------
# sense: returns x- and y- distances to landmarks within visibility range
#        because not all landmarks may be in this range, the list of measurements
#        is of variable length. Set measurement_range to -1 if you want all
#        landmarks to be visible at all times
#

## TODO: complete the sense function
```

```
def sense(self):
    ''' This function does not take in any parameters, instead it references interna
        (such as self.landamrks) to measure the distance between the robot and any l
        that the robot can see (that are within its measurement range).
        This function returns a list of landmark indices, and the measured distances
        between the robot's position and said landmarks.
        This function should account for measurement_noise and measurement_range.
        One item in the returned list should be in the form: [landmark_index, dx, dy
        '''

    measurements = []

    ## TODO: iterate through all of the landmarks in a world

    ## TODO: For each landmark
    ## 1. compute dx and dy, the distances between the robot and the landmark
    ## 2. account for measurement noise by *adding* a noise component to dx and dy
    ##    - The noise component should be a random value between [-1.0, 1.0)*measure
    ##    - Feel free to use the function self.rand() to help calculate this noise c
    ##    - It may help to reference the `move` function for noise calculation
    ## 3. If either of the distances, dx or dy, fall outside of the internal var, me
    ##    then we cannot record them; if they do fall in the range, then add them to
    ##    as list.append([index, dx, dy]), this format is important for data creatio
    for i, lm in enumerate(self.landmarks):
        dx, dy = lm[0] - self.x , lm[1] - self.y
        dx += self.rand() * self.measurement_noise
        dy += self.rand() * self.measurement_noise
        if (dx**2 + dy**2 <= self.measurement_range**2):
            measurements.append([i, dx, dy])

    ## TODO: return the final, complete list of measurements
    return measurements


    ## TODO: return the final, complete list of measurements
    return measurements


# --------
# make_landmarks:
# make random landmarks located in the world
#
def make_landmarks(self, num_landmarks):
    self.landmarks = []
    for i in range(num_landmarks):
        self.landmarks.append([round(random.random() * self.world_size),
                               round(random.random() * self.world_size)])
    self.num_landmarks = num_landmarks
```

```
        # called when print(robot) is called; prints the robot's location
        def __repr__(self):
            return 'Robot: [x=%.5f y=%.5f]'  % (self.x, self.y)
```

1.1 Define a world and a robot

Next, let's instantiate a robot object. As you can see in `__init__` above, the robot class takes in a number of parameters including a world size and some values that indicate the sensing and movement capabilities of the robot.

In the next example, we define a small 10x10 square world, a measurement range that is half that of the world and small values for motion and measurement noise. These values will typically be about 10 times larger, but we ust want to demonstrate this behavior on a small scale. You are also free to change these values and note what happens as your robot moves!

```
In [5]: world_size         = 10.0    # size of world (square)
        measurement_range  = 5.0     # range at which we can sense landmarks
        motion_noise       = 0.2      # noise in robot motion
        measurement_noise  = 0.2      # noise in the measurements

        # instantiate a robot, r
        r = robot(world_size, measurement_range, motion_noise, measurement_noise)

        # print out the location of r
        print(r)
```

```
Robot: [x=5.00000 y=5.00000]
```

1.2 Visualizing the World

In the given example, we can see/print out that the robot is in the middle of the 10x10 world at (x, y) = (5.0, 5.0), which is exactly what we expect!

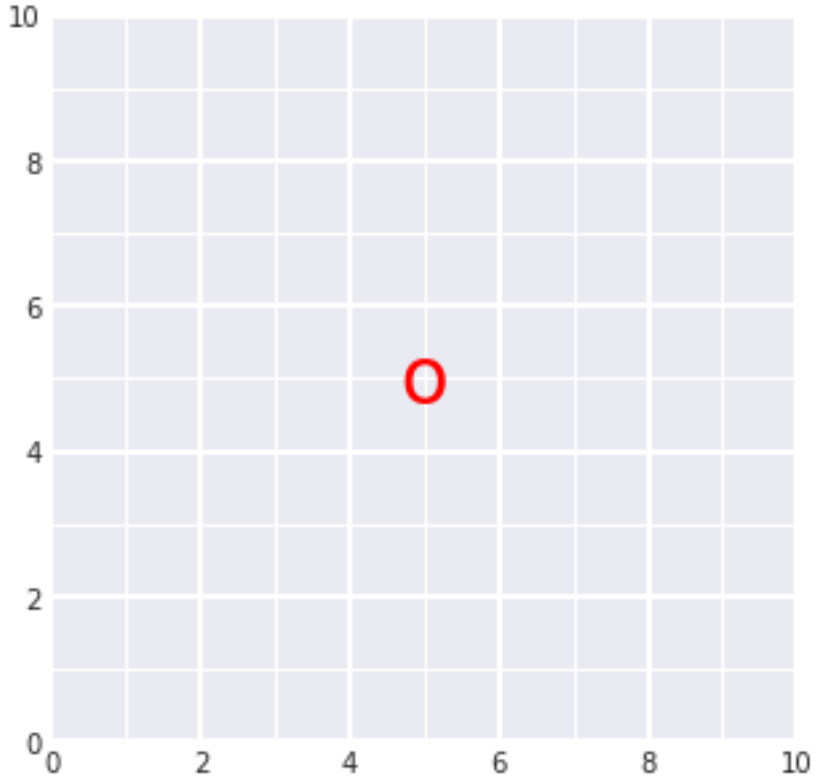
However, it's kind of hard to imagine this robot in the center of a world, without visualizing the grid itself, and so in the next cell we provide a helper visualization function, `display_world`, that will display a grid world in a plot and draw a red `o` at the location of our robot, `r`. The details of how this function wors can be found in the `helpers.py` file in the home directory; you do not have to change anything in this `helpers.py` file.

```
In [6]: # import helper function
        from helpers import display_world

        # define figure size
        plt.rcParams["figure.figsize"] = (5,5)

        # call display_world and display the robot in it's grid world
        print(r)
        display_world(int(world_size), [r.x, r.y])
```

```
Robot: [x=5.00000 y=5.00000]
```



1.3 Movement

Now you can really picture where the robot is in the world! Next, let's call the robot's `move` function. We'll ask it to move some distance `(dx, dy)` and we'll see that this motion is not perfect by the placement of our robot `o` and by the printed out position of `r`.

Try changing the values of `dx` and `dy` and/or running this cell multiple times; see how the robot moves and how the uncertainty in robot motion accumulates over multiple movements.

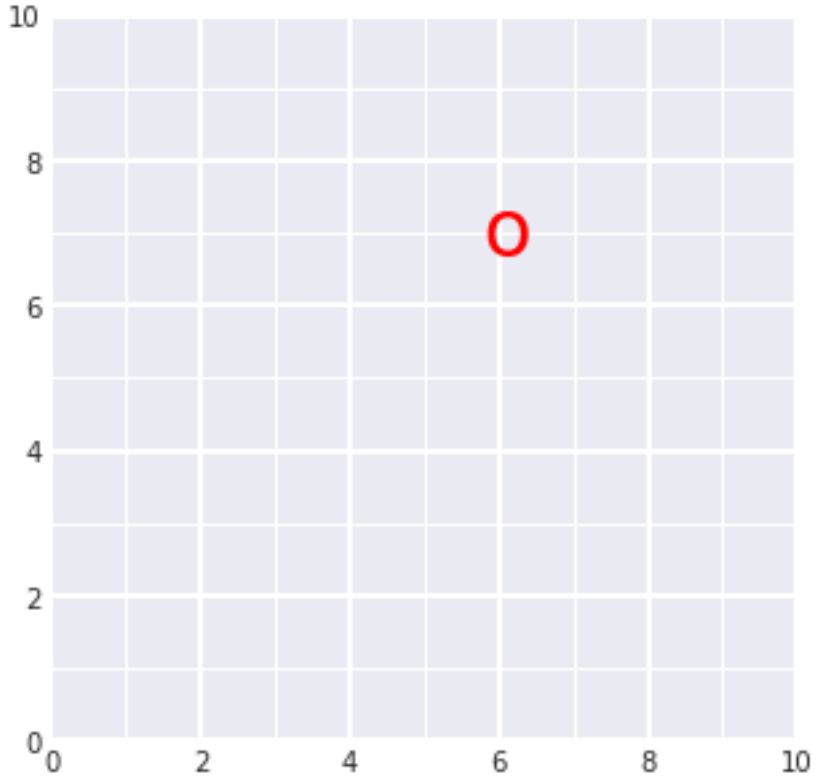
For a `dx` = 1, does the robot move *exactly* one spot to the right? What about `dx` = -1? What happens if you try to move the robot past the boundaries of the world?

```
In [7]: # choose values of dx and dy (negative works, too)
        dx = 1
        dy = 2
        r.move(dx, dy)

        # print out the exact location
        print(r)
```

```
        # display the world after movement, not that this is the same call as before
        # the robot tracks its own movement
        display_world(int(world_size), [r.x, r.y])
```

```
Robot: [x=6.09270 y=7.00651]
```



1.4 Landmarks

Next, let's create landmarks, which are measurable features in the map. You can think of landmarks as things like notable buildings, or something smaller such as a tree, rock, or other feature.

The robot class has a function `make_landmarks` which randomly generates locations for the number of specified landmarks. Try changing `num_landmarks` or running this cell multiple times to see where these landmarks appear. We have to pass these locations as a third argument to the `display_world` function and the list of landmark locations is accessed similar to how we find the robot position `r.landmarks`.

Each landmark is displayed as a purple `x` in the grid world, and we also print out the exact `[x, y]` locations of these landmarks at the end of this cell.

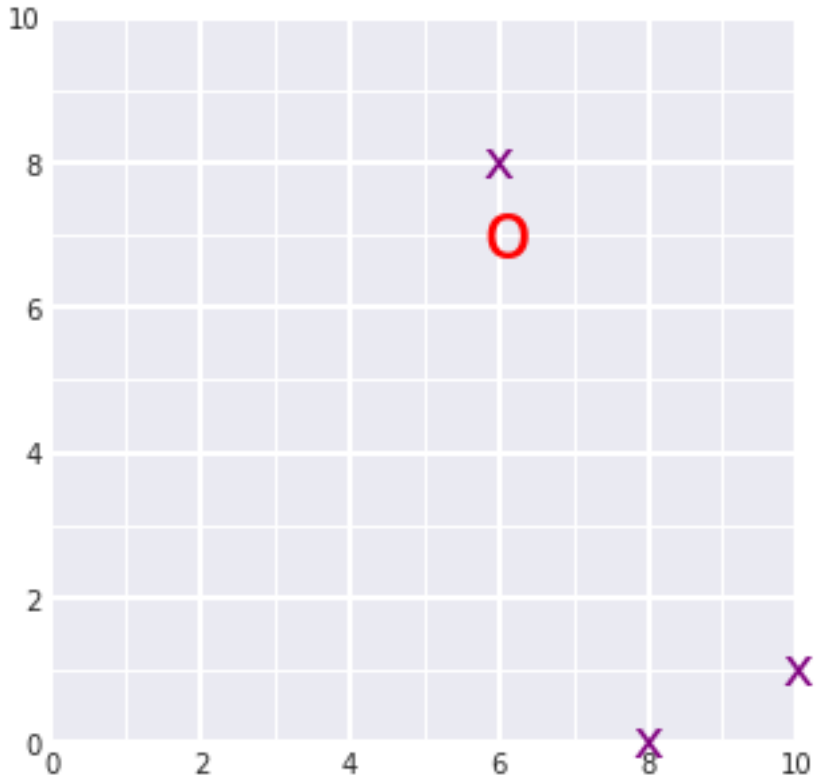
```
In [8]: # create any number of landmarks
        num_landmarks = 3
        r.make_landmarks(num_landmarks)
```

```
        # print out our robot's exact location
        print(r)

        # display the world including these landmarks
        display_world(int(world_size), [r.x, r.y], r.landmarks)

        # print the locations of the landmarks
        print('Landmark locations [x,y]: ', r.landmarks)
```

```
Robot: [x=6.09270 y=7.00651]
```




```
Landmark locations [x,y]:  [[10, 1], [6, 8], [8, 0]]
```

1.5 Sense

Once we have some landmarks to sense, we need to be able to tell our robot to *try* to sense how far they are away from it. It will be up t you to code the `sense` function in our robot class.

The `sense` function uses only internal class parameters and returns a list of the the measured/sensed x and y distances to the landmarks it senses within the specified `measurement_range`.

1.5.1 TODO: Implement the `sense` function

Follow the `##TODO's` in the class code above to complete the `sense` function for the robot class. Once you have tested out your code, please **copy your complete `sense` code to the `robot_class.py` file in the home directory**. By placing this complete code in the `robot_class` Python file, we will be able to refernce this class in a later notebook.

The measurements have the format, `[i, dx, dy]` where `i` is the landmark index (0, 1, 2, ...) and `dx` and `dy` are the measured distance between the robot's location (x, y) and the landmark's location (x, y). This distance will not be perfect since our sense function has some associated `measurement noise`.

In the example in the following cell, we have a given our robot a range of `5.0` so any landmarks that are within that range of our robot's location, should appear in a list of measurements. Not all landmarks are guaranteed to be in our visibility range, so this list will be variable in length.

*Note: the robot's location is often called the **pose** or `[Pxi, Pyi]` and the landmark locations are often written as `[Lxi, Lyi]`. You'll see this notation in the next notebook.*

```
In [9]: # try to sense any surrounding landmarks
        measurements = r.sense()

        # this will print out an empty list if `sense` has not been implemented
        print(measurements)
```

```
[[1, -0.08816124978456759, 1.141589824338623]]
```

Refer back to the grid map above. Do these measurements make sense to you? Are all the landmarks captured in this list (why/why not)?

1.6 Data

Putting it all together To perform SLAM, we'll collect a series of robot sensor measurements and motions, in that order, over a defined period of time. Then we'll use only this data to re-construct the map of the world with the robot and landmar locations. You can think of SLAM as peforming what we've done in this notebook, only backwards. Instead of defining a world and robot and creating movement and sensor data, it will be up to you to use movement and sensor measurements to reconstruct the world!

In the next notebook, you'll see this list of movements and measurements (which you'll use to re-construct the world) listed in a structure called `data`. This is an array that holds sensor measurements and movements in a specific order, which will be useful to call upon when you have to extract this data and form constraint matrices and vectors.

`data` is constructed over a series of time steps as follows:

```
In [10]: data = []
```

```
        # after a robot first senses, then moves (one time step)
        # that data is appended like so:
        data.append([measurements, [dx, dy]])

        # for our example movement and measurement
        print(data)
```

```
[[[[1, -0.08816124978456759, 1.141589824338623]], [1, 2]]]
```

```
In [11]: # in this example, we have only created one time step (0)
         time_step = 0

         # so you can access robot measurements:
         print('Measurements: ', data[time_step][0])

         # and its motion for a given time step:
         print('Motion: ', data[time_step][1])
```

```
Measurements:  [[1, -0.08816124978456759, 1.141589824338623]]
Motion:  [1, 2]
```

1.6.1 Final robot class

Before moving on to the last notebook in this series, please make sure that you have copied your final, completed `sense` function into the `robot_class.py` file in the home directory. We will be using this file in the final implementation of slam!