3. Landmark Detection and Tracking

May 19, 2020

1 Project 3: Implement SLAM

1.1 Project Overview

In this project, you'll implement SLAM for robot that moves and senses in a 2 dimensional, grid world!

SLAM gives us a way to both localize a robot and build up a map of its environment as a robot moves and senses in real-time. This is an active area of research in the fields of robotics and autonomous systems. Since this localization and map-building relies on the visual sensing of landmarks, this is a computer vision problem.

Using what you've learned about robot motion, representations of uncertainty in motion and sensing, and localization techniques, you will be tasked with defining a function, `slam`, which takes in six parameters as input and returns the vector `mu`. > `mu` contains the (x,y) coordinate locations of the robot as it moves, and the positions of landmarks that it senses in the world

You can implement helper functions as you see fit, but your function must return `mu`. The vector, `mu`, should have (x, y) coordinates interlaced, for example, if there were 2 poses and 2 landmarks, `mu` will look like the following, where `P` is the robot position and `L` the landmark position:

```
mu =  matrix([[Px0],
              [Py0],
              [Px1],
              [Py1],
              [Lx0],
              [Ly0],
              [Lx1],
              [Ly1]])
```

You can see that `mu` holds the poses first `(x0, y0), (x1, y1), ...`, then the landmark locations at the end of the matrix; we consider a `nx1` matrix to be a vector.

1.2 Generating an environment

In a real SLAM problem, you may be given a map that contains information about landmark locations, and in this example, we will make our own data using the `make_data` function, which

generates a world grid with landmarks in it and then generates data by placing a robot in that world and moving and sensing over some numer of time steps. The `make_data` function relies on a correct implementation of robot move/sense functions, which, at this point, should be complete and in the `robot_class.py` file. The data is collected as an instantiated robot moves and senses in a world. Your SLAM function will take in this data as input. So, let's first create this data and explore how it represents the movement and sensor measurements that our robot takes.

1.3 Create the world

Use the code below to generate a world of a specified size with randomly generated landmark locations. You can change these parameters and see how your implementation of SLAM responds!

`data` holds the sensors measurements and motion of your robot over time. It stores the measurements as `data[i][0]` and the motion as `data[i][1]`.

Helper functions You will be working with the `robot` class that may look familiar from the first notebook,

In fact, in the `helpers.py` file, you can read the details of how data is made with the `make_data` function. It should look very similar to the robot move/sense cycle you've seen in the first notebook.

```
In [1]: import numpy as np
        from helpers import make_data

        # your implementation of slam should work with the following inputs
        # feel free to change these input values and see how it responds!

        # world parameters
        num_landmarks      = 5        # number of landmarks
        N                  = 20       # time steps
        world_size         = 100.0    # size of world (square)

        # robot parameters
        measurement_range  = 50.0     # range at which we can sense landmarks
        motion_noise       = 2.0      # noise in robot motion
        measurement_noise  = 2.0      # noise in the measurements
        distance           = 20.0     # distance by which robot (intends to) move each iteratati


        # make_data instantiates a robot, AND generates random landmarks for a given world size
        data = make_data(N, num_landmarks, world_size, measurement_range, motion_noise, measurem


Landmarks:  [[3, 73], [83, 23], [65, 50], [50, 4], [54, 57]]
Robot: [x=31.72668 y=35.19438]
```

1.3.1 A note on `make_data`

The function above, `make_data`, takes in so many world and robot motion/sensor parameters because it is responsible for: 1. Instantiating a robot (using the robot class) 2. Creating a grid world with landmarks in it

This function also prints out the true location of landmarks and the *final* robot location, which you should refer back to when you test your implementation of SLAM.

The `data` this returns is an array that holds information about **robot sensor measurements** and **robot motion** `(dx, dy)` that is collected over a number of time steps, `N`. You will have to use *only* these readings about motion and measurements to track a robot over time and find the determine the location of the landmarks using SLAM. We only print out the true landmark locations for comparison, later.

In `data` the measurement and motion data can be accessed from the first and second index in the columns of the data array. See the following code for an example, where `i` is the time step:

```
measurement = data[i][0]
motion = data[i][1]
```

```
In [2]: # print out some stats about the data
        time_step = 0

        print('Example measurements: \n', data[time_step][0])
        print('\n')
        print('Example motion: \n', data[time_step][1])
```

```
Example measurements:
 [[1, 31.402214718910848, -25.917660953032918], [2, 14.46652668657503, -0.20906931168967713], [3


Example motion:
 [14.048058996814028, -14.235590554031557]
```

Try changing the value of `time_step`, you should see that the list of measurements varies based on what in the world the robot sees after it moves. As you know from the first notebook, the robot can only sense so far and with a certain amount of accuracy in the measure of distance between its location and the location of landmarks. The motion of the robot always is a vector with two values: one for x and one for y displacement. This structure will be useful to keep in mind as you traverse this data in your implementation of slam.

1.4 Initialize Constraints

One of the most challenging tasks here will be to create and modify the constraint matrix and vector: omega and xi. In the second notebook, you saw an example of how omega and xi could hold all the values the define the relationships between robot poses `xi` and landmark positions `Li` in a 1D world, as seen below, where omega is the blue matrix and xi is the pink vector.

In *this* project, you are tasked with implementing constraints for a 2D world. We are referring to robot poses as `Px, Py` and landmark positions as `Lx, Ly`, and one way to approach this challenge is to add *both* x and y locations in the constraint matrices.

You may also choose to create two of each omega and xi (one for x and one for y positions).

1.4.1 TODO: Write a function that initializes omega and xi

Complete the function `initialize_constraints` so that it returns `omega` and `xi` constraints for the starting position of the robot. Any values that we do not yet know should be initialized with the value `0`. You may assume that our robot starts out in exactly the middle of the world with 100% confidence (no motion or measurement noise at this point). The inputs `N` time steps, `num_landmarks`, and `world_size` should give you all the information you need to construct intial constraints of the correct size and starting values.

Depending on your approach you may choose to return one omega and one xi that hold all (x,y) positions or* two of each (one for x values and one for y); choose whichever makes most sense to you!*

```
In [3]: def initialize_constraints(N, num_landmarks, world_size):
            ''' This function takes in a number of time steps N, number of landmarks, and a worl
                and returns initialized constraint matrices, omega and xi.'''

            ## Recommended: Define and store the size (rows/cols) of the constraint matrix in a

            ## TODO: Define the constraint matrix, Omega, with two initial "strength" values
            ## for the initial x, y location of our robot
            size = N+num_landmarks
            omega = np.zeros((size*2,size*2))

            omega[0][0] = omega[1][1] = 1

            ## TODO: Define the constraint *vector*, xi
            ## you can assume that the robot starts out in the middle of the world with 100% con
            xi = np.zeros((size*2))

            xi[0:2] = [world_size/2,world_size/2]

            return omega, xi
```

1.4.2 Test as you go

It's good practice to test out your code, as you go. Since `slam` relies on creating and updating constraint matrices, `omega` and `xi` to account for robot sensor measurements and motion, let's check that they initialize as expected for any given parameters.

Below, you'll find some test code that allows you to visualize the results of your function `initialize_constraints`. We are using the seaborn library for visualization.

Please change the test values of N, landmarks, and world_size and see the results. Be careful not to use these values as input into your final smal function.

This code assumes that you have created one of each constraint: `omega` and `xi`, but you can change and add to this code, accordingly. The constraints should vary in size with the number of time steps and landmarks as these values affect the number of poses a robot will take `(Px0,Py0,...Pxn,Pyn)` and landmark locations `(Lx0,Ly0,...Lxn,Lyn)` whose relationships should be tracked in the constraint matrices. Recall that `omega` holds the weights of each variable and `xi` holds the value of the sum of these variables, as seen in Notebook 2. You'll need the

`world_size` to determine the starting pose of the robot in the world and fill in the initial values for `xi`.

```
In [4]: # import data viz resources
        import matplotlib.pyplot as plt
        from pandas import DataFrame
        import seaborn as sns
        %matplotlib inline
```

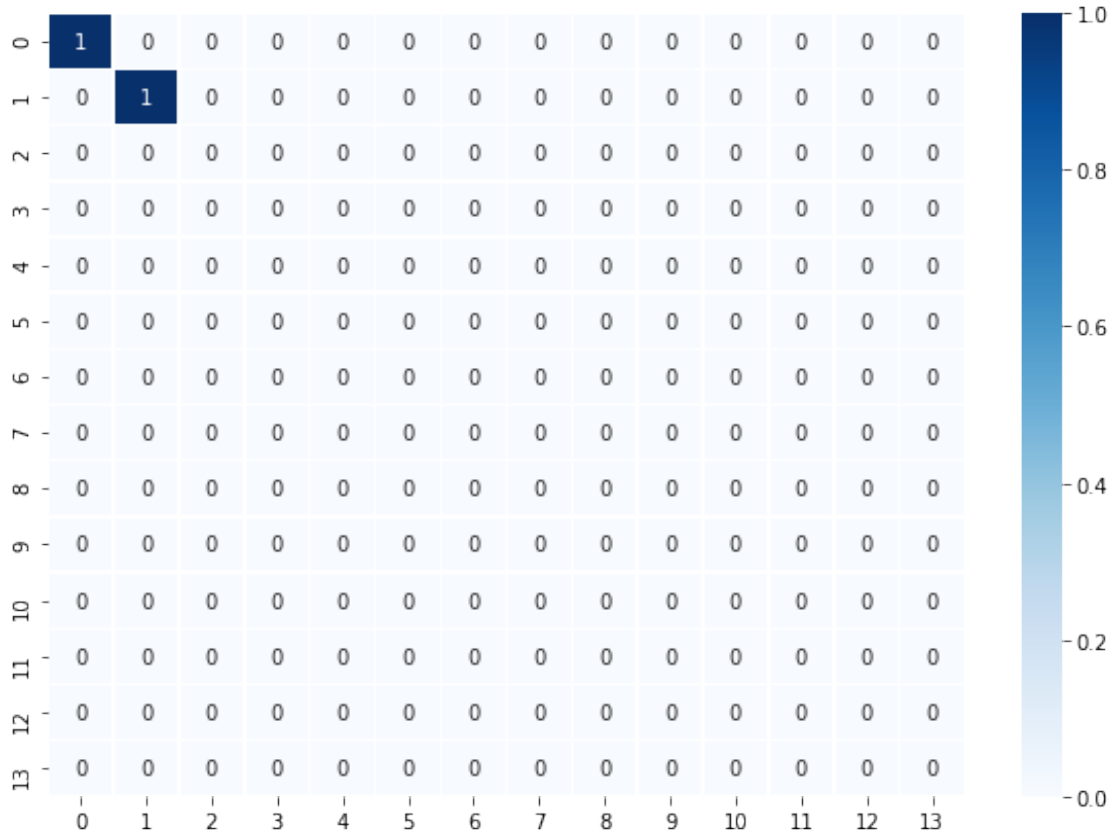
```
In [5]: # define a small N and world_size (small for ease of visualization)
        N_test = 5
        num_landmarks_test = 2
        small_world = 10

        # initialize the constraints
        initial_omega, initial_xi = initialize_constraints(N_test, num_landmarks_test, small_wor
```

```
In [6]: # define figure size
        plt.rcParams["figure.figsize"] = (10,7)

        # display omega
        sns.heatmap(DataFrame(initial_omega), cmap='Blues', annot=True, linewidths=.5)
```

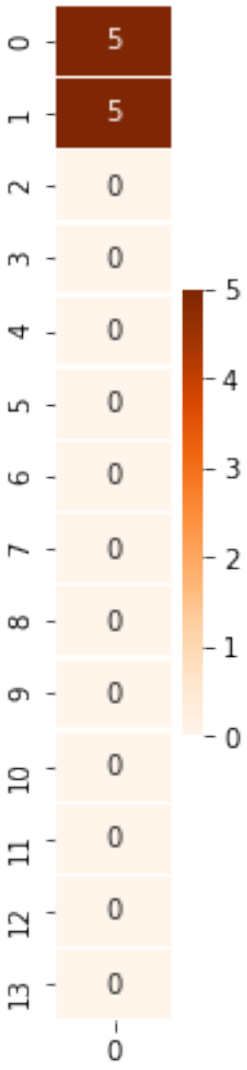
```
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb9eb185a90>
```



In [7]:
```
# define  figure size
plt.rcParams["figure.figsize"] = (1,7)

# display xi
sns.heatmap(DataFrame(initial_xi), cmap='Oranges', annot=True, linewidths=.5)
```

Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb9e8b34518>



1.5 SLAM inputs

In addition to `data`, your slam function will also take in: * N - The number of time steps that a robot will be moving and sensing * num_landmarks - The number of landmarks in the world * world_size - The size (w/h) of your world * motion_noise - The noise associated with motion; the update confidence for motion should be `1.0/motion_noise` * measurement_noise - The noise associated with measurement/sensing; the update weight for measurement should be `1.0/measurement_noise`

A note on noise Recall that `omega` holds the relative "strengths" or weights for each position variable, and you can update these weights by accessing the correct index in omega `omega[row][col]` and *adding/subtracting* `1.0/noise` where `noise` is measurement or motion noise. `Xi` holds actual position values, and so to update `xi` you'll do a similar addition process only using the actual value of a motion or measurement. So for a vector index `xi[row][0]` you will end up adding/subtracting one measurement or motion divided by their respective `noise`.

1.5.1 TODO: Implement Graph SLAM

Follow the TODO's below to help you complete this slam implementation (these TODO's are in the recommended order), then test out your implementation!

Updating with motion and measurements With a 2D omega and xi structure as shown above (in earlier cells), you'll have to be mindful about how you update the values in these constraint matrices to account for motion and measurement constraints in the x and y directions. Recall that the solution to these matrices (which holds all values for robot poses `P` and landmark locations `L`) is the vector, `mu`, which can be computed at the end of the construction of omega and xi as the inverse of omega times xi: $\mu = \Omega^{-1}\xi$

You may also choose to return the values of `omega` and `xi` if you want to visualize their final state!

In [8]:
```
## TODO: Complete the code to implement SLAM

        ## slam takes in 6 arguments and returns mu,
        ## mu is the entire path traversed by a robot (all x,y poses) *and* all landmarks locati
        def slam(data, N, num_landmarks, world_size, motion_noise, measurement_noise):

            Wms = 1.0/measurement_noise
            Wmo = 1.0/motion_noise

            ## TODO: Use your initilization to create constraint matrices, omega and xi
            omega, xi = initialize_constraints(N, num_landmarks, world_size)

            ## TODO: Iterate through each time step in the data
            ## get all the motion and measurement data as you iterate
            for ts, sample in enumerate(data):
                measurements, motion = sample
                pose_i = 2*ts
```

```
        ## TODO: update the constraint matrix/vector to account for all *measurements*
        ## this should be a series of additions that take into account the measurement noise
            for m in measurements:
                lm_j, lm_x, lm_y = m[0], m[1], m[2]
                lm_j = 2*(N + lm_j)
                omega[pose_i]  [pose_i]     +=  Wms   # pose_i.x <-> pose_i.x
                omega[pose_i+1][pose_i+1]   +=  Wms   # pose_i.y <-> pose_i.y
                omega[pose_i]  [lm_j]       += -Wms   # pose_i.x <-> lm_j.x
                omega[pose_i+1][lm_j + 1]   += -Wms   # pose_i.y <-> lm_j.y
                omega[lm_j]    [pose_i]     += -Wms   # lm_j.x <-> pose_i.x
                omega[lm_j + 1][pose_i+1]   += -Wms   # lm_j.y <-> pose_i.y
                omega[lm_j]    [lm_j]       +=  Wms   # lm_j.x <-> lm_j.x
                omega[lm_j + 1][lm_j + 1]   +=  Wms   # lm_j.y <-> lm_j.y

                xi[pose_i]   += -lm_x*Wms
                xi[pose_i+1] += -lm_y*Wms
                xi[lm_j]     +=  lm_x*Wms
                xi[lm_j+1]   +=  lm_y*Wms

        ## TODO: update the constraint matrix/vector to account for all *motion* and motion
            omega[pose_i]  [pose_i]     +=  Wmo   # pose_i.x <-> pose_i.x
            omega[pose_i+1][pose_i+1]   +=  Wmo   # pose_i.y <-> pose_i.y
            omega[pose_i]  [pose_i+2]   += -Wmo   # pose_i.x <-> pose_i+1.x
            omega[pose_i+1][pose_i+3]   += -Wmo   # pose_i.y <-> pose_i+1.y
            omega[pose_i+2][pose_i]     += -Wmo   # pose_i+1.x <-> pose_i.x
            omega[pose_i+3][pose_i+1]   += -Wmo   # pose_i+1.y <-> pose_i.y
            omega[pose_i+2][pose_i+2]   +=  Wmo   # pose_i+1.x <-> pose_i+1.x
            omega[pose_i+3][pose_i+3]   +=  Wmo   # pose_i+1.y <-> pose_i+1.y

            xi[pose_i]   += -Wmo * motion[0]
            xi[pose_i+1] += -Wmo * motion[1]
            xi[pose_i+2] +=  Wmo * motion[0]
            xi[pose_i+3] +=  Wmo * motion[1]

        omega_inv = np.linalg.inv(omega)
        mu = np.dot(omega_inv, xi)

        return mu
```

1.6 Helper functions

To check that your implementation of SLAM works for various inputs, we have provided two helper functions that will help display the estimated pose and landmark locations that your function has produced. First, given a result `mu` and number of time steps, `N`, we define a function that extracts the poses and landmarks locations and returns those as their own, separate lists.

Then, we define a function that nicely print out these lists; both of these we will call, in the next step.

```
In [9]: # a helper function that creates a list of poses and of landmarks for ease of printing
        # this only works for the suggested constraint architecture of interlaced x,y poses
        def get_poses_landmarks(mu, N):
            # create a list of poses
            poses = []
            for i in range(N):
                poses.append((mu[2*i].item(), mu[2*i+1].item()))

            # create a list of landmarks
            landmarks = []
            for i in range(num_landmarks):
                landmarks.append((mu[2*(N+i)].item(), mu[2*(N+i)+1].item()))

            # return completed lists
            return poses, landmarks
```

```
In [10]: def print_all(poses, landmarks):
             print('\n')
             print('Estimated Poses:')
             for i in range(len(poses)):
                 print('['+', '.join('%.3f'%p for p in poses[i])+']')
             print('\n')
             print('Estimated Landmarks:')
             for i in range(len(landmarks)):
                 print('['+', '.join('%.3f'%l for l in landmarks[i])+']')
```

1.7 Run SLAM

Once you've completed your implementation of `slam`, see what `mu` it returns for different world sizes and different landmarks!

1.7.1 What to Expect

The `data` that is generated is random, but you did specify the number, `N`, or time steps that the robot was expected to move and the `num_landmarks` in the world (which your implementation of `slam` should see and estimate a position for. Your robot should also start with an estimated pose in the very center of your square world, whose size is defined by `world_size`.

With these values in mind, you should expect to see a result that displays two lists: 1. **Estimated poses**, a list of (x, y) pairs that is exactly `N` in length since this is how many motions your robot has taken. The very first pose should be the center of your world, i.e. `[50.000, 50.000]` for a world that is 100.0 in square size. 2. **Estimated landmarks**, a list of landmark positions (x, y) that is exactly `num_landmarks` in length.

Landmark Locations If you refer back to the printout of *exact* landmark locations when this data was created, you should see values that are very similar to those coordinates, but not quite (since `slam` must account for noise in motion and measurement).

```
In [11]: # call your implementation of slam, passing in the necessary parameters
         mu = slam(data, N, num_landmarks, world_size, motion_noise, measurement_noise)
```

```
# print out the resulting landmarks and poses
if(mu is not None):
    # get the lists of poses and landmarks
    # and print them out
    poses, landmarks = get_poses_landmarks(mu, N)
    print_all(poses, landmarks)
```

```
Estimated Poses:
[50.000, 50.000]
[62.366, 33.866]
[75.146, 17.389]
[88.150, 2.259]
[75.359, 17.996]
[62.154, 32.769]
[49.015, 47.956]
[38.252, 64.000]
[24.715, 79.298]
[12.159, 94.523]
[31.471, 95.554]
[51.443, 96.186]
[70.927, 95.060]
[90.054, 94.218]
[83.950, 75.128]
[78.877, 57.832]
[71.952, 38.242]
[65.232, 20.339]
[48.211, 27.443]
[29.390, 34.209]


Estimated Landmarks:
[3.228, 72.369]
[82.564, 22.949]
[64.203, 49.804]
[49.102, 3.740]
[53.477, 56.504]
```

1.8 Visualize the constructed world

Finally, using the `display_world` code from the `helpers.py` file (which was also used in the first notebook), we can actually visualize what you have coded with `slam`: the final position of the robot and the positon of landmarks, created from only motion and measurement data!

Note that these should be very similar to the printed *true* landmark locations and final pose from our call to `make_data` early in this notebook.

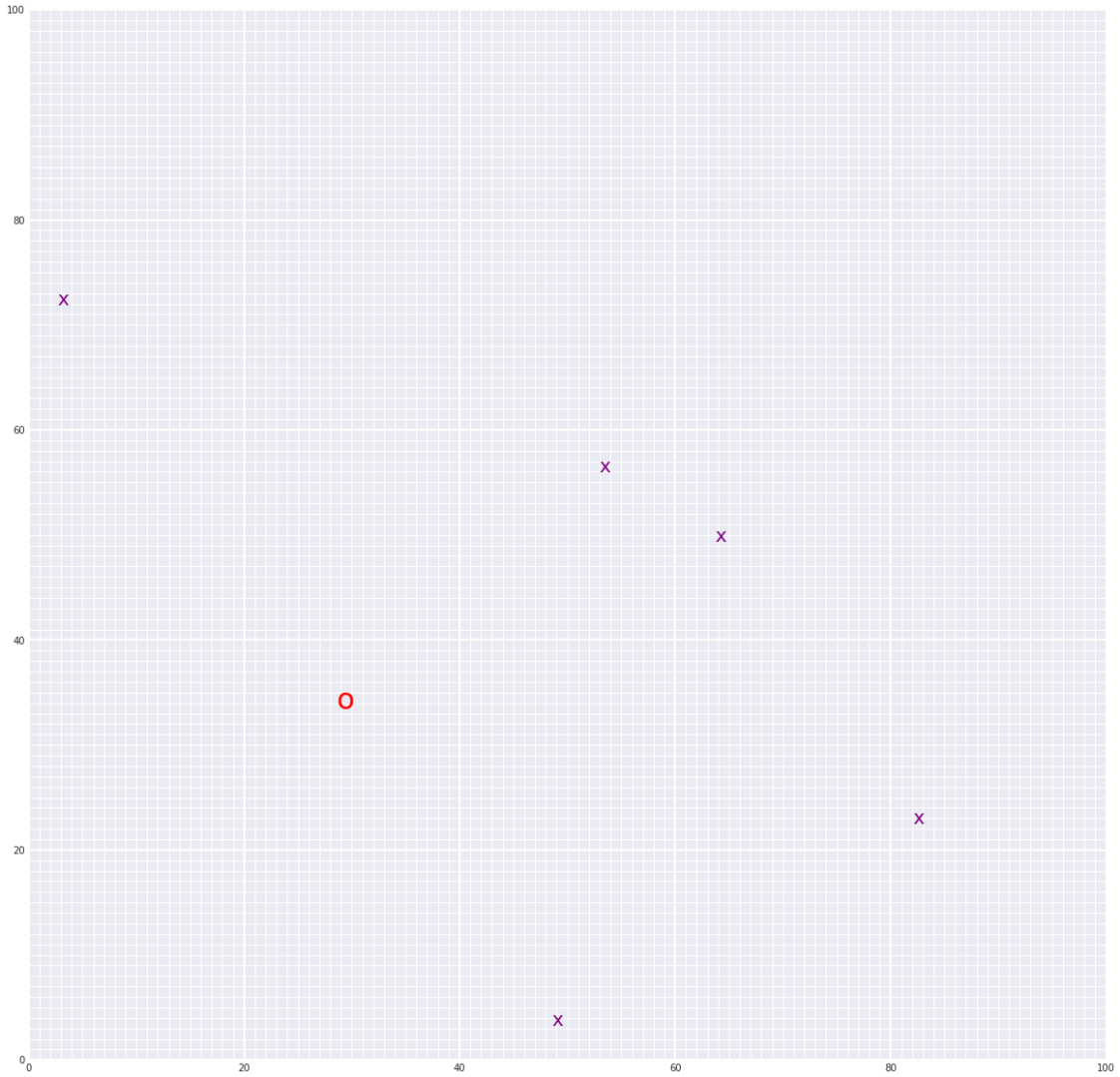
In [12]:
```
# import the helper function
from helpers import display_world

# Display the final world!

# define figure size
plt.rcParams["figure.figsize"] = (20,20)

# check if poses has been created
if 'poses' in locals():
    # print out the last pose
    print('Last pose: ', poses[-1])
    # display the last position of the robot *and* the landmark positions
    display_world(int(world_size), poses[-1], landmarks)
```

```
Last pose:  (29.390259380721254, 34.208715506709424)
```



1.8.1 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: True Pose = (x=31.73 y=35.19) and Final Pose = (x=29.39, y=34.20). The difference = (x=2.34 , y=0.99) . This difference is because of the uncertainty involved in motion measurement

1.9 Testing

To confirm that your slam code works before submitting your project, it is suggested that you run it on some test data and cases. A few such cases have been provided for you, in the cells below. When you are ready, uncomment the test cases in the next cells (there are two test cases, total); your output should be **close-to or exactly** identical to the given results. If there are minor discrepancies it could be a matter of floating point accuracy or in the calculation of the inverse matrix.

1.9.1 Submit your project

If you pass these tests, it is a good indication that your project will pass all the specifications in the project rubric. Follow the submission instructions to officially submit!

In [14]:
```
# Here is the data and estimated outputs for test case 1

test_data1 = [[[[1, 19.457599255548065, 23.8387362100849], [2, -13.195807561967236, 11.

##  Test Case 1
##
# Estimated Pose(s):
#     [50.000, 50.000]
#     [37.858, 33.921]
#     [25.905, 18.268]
#     [13.524, 2.224]
#     [27.912, 16.886]
#     [42.250, 30.994]
#     [55.992, 44.886]
#     [70.749, 59.867]
#     [85.371, 75.230]
#     [73.831, 92.354]
#     [53.406, 96.465]
#     [34.370, 100.134]
#     [48.346, 83.952]
#     [60.494, 68.338]
```

```
#     [73.648, 53.082]
#     [86.733, 38.197]
#     [79.983, 20.324]
#     [72.515, 2.837]
#     [54.993, 13.221]
#     [37.164, 22.283]


# Estimated Landmarks:
#     [82.679, 13.435]
#     [70.417, 74.203]
#     [36.688, 61.431]
#     [18.705, 66.136]
#     [20.437, 16.983]


### Uncommment the following three lines for test case 1 and compare the output to the v

mu_1 = slam(test_data1, 20, 5, 100.0, 2.0, 2.0)
poses, landmarks = get_poses_landmarks(mu_1, 20)
print_all(poses, landmarks)
```

```
Estimated Poses:
[50.000, 50.000]
[37.973, 33.652]
[26.185, 18.155]
[13.745, 2.116]
[28.097, 16.783]
[42.384, 30.902]
[55.831, 44.497]
[70.857, 59.699]
[85.697, 75.543]
[74.011, 92.434]
[53.544, 96.454]
[34.525, 100.080]
[48.623, 83.953]
[60.197, 68.107]
[73.778, 52.935]
[87.132, 38.538]
[80.303, 20.508]
[72.798, 2.945]
[55.245, 13.255]
[37.416, 22.317]


Estimated Landmarks:
```

```
[82.956, 13.539]
[70.495, 74.141]
[36.740, 61.281]
[18.698, 66.060]
[20.635, 16.875]
```

In [15]:
```
# Here is the data and estimated outputs for test case 2

test_data2 = [[[[0, 26.543274387283322, -6.262538160312672], [3, 9.937396825799755, -9.

##  Test Case 2
##
# Estimated Pose(s):
#     [50.000, 50.000]
#     [69.035, 45.061]
#     [87.655, 38.971]
#     [76.084, 55.541]
#     [64.283, 71.684]
#     [52.396, 87.887]
#     [44.674, 68.948]
#     [37.532, 49.680]
#     [31.392, 30.893]
#     [24.796, 12.012]
#     [33.641, 26.440]
#     [43.858, 43.560]
#     [54.735, 60.659]
#     [65.884, 77.791]
#     [77.413, 94.554]
#     [96.740, 98.020]
#     [76.149, 99.586]
#     [70.211, 80.580]
#     [64.130, 61.270]
#     [58.183, 42.175]


# Estimated Landmarks:
#     [76.777, 42.415]
#     [85.109, 76.850]
#     [13.687, 95.386]
#     [59.488, 39.149]
#     [69.283, 93.654]


### Uncomment the following three lines for test case 2 and compare to the values above

mu_2 = slam(test_data2, 20, 5, 100.0, 2.0, 2.0)
```

```
poses, landmarks = get_poses_landmarks(mu_2, 20)
print_all(poses, landmarks)
```

```
Estimated Poses:
[50.000, 50.000]
[69.181, 45.665]
[87.743, 39.703]
[76.270, 56.311]
[64.317, 72.176]
[52.257, 88.154]
[44.059, 69.401]
[37.002, 49.918]
[30.924, 30.955]
[23.508, 11.419]
[34.180, 27.133]
[44.155, 43.846]
[54.806, 60.920]
[65.698, 78.546]
[77.468, 95.626]
[96.802, 98.821]
[75.957, 99.971]
[70.200, 81.181]
[64.054, 61.723]
[58.107, 42.628]


Estimated Landmarks:
[76.779, 42.887]
[85.065, 77.438]
[13.548, 95.652]
[59.449, 39.595]
[69.263, 94.240]
```

In [ ]: