Sasha Aastav Sen

PROJECT: Particle Filter (Python) Investigation

Problem Definition

In this project we will be concerned only about the pose estimate of a two wheel nonholonomic differential drive robot with input controls of forward velocity and angular velocity. The inputs are taken as constant positive values to result in the robot tracing a circular path (forward velocity = 2.0 m/s, angular velocity = 1.0 rad/s). The motion and measurement model are both noisey, where noise is modeled by a gaussian distribution. In a real world scenario a degree of uncertainty in this motion model is expected, as well as the effects of sensor noise on the measurement model. Thus, the motion model's covariance is taken as Q and the measurement model's covariance is taken as R (both kept constant). The values chosen for the Q and R matrix were chosen arbitrarily by trial and error and are illustrated below in Figure 1.

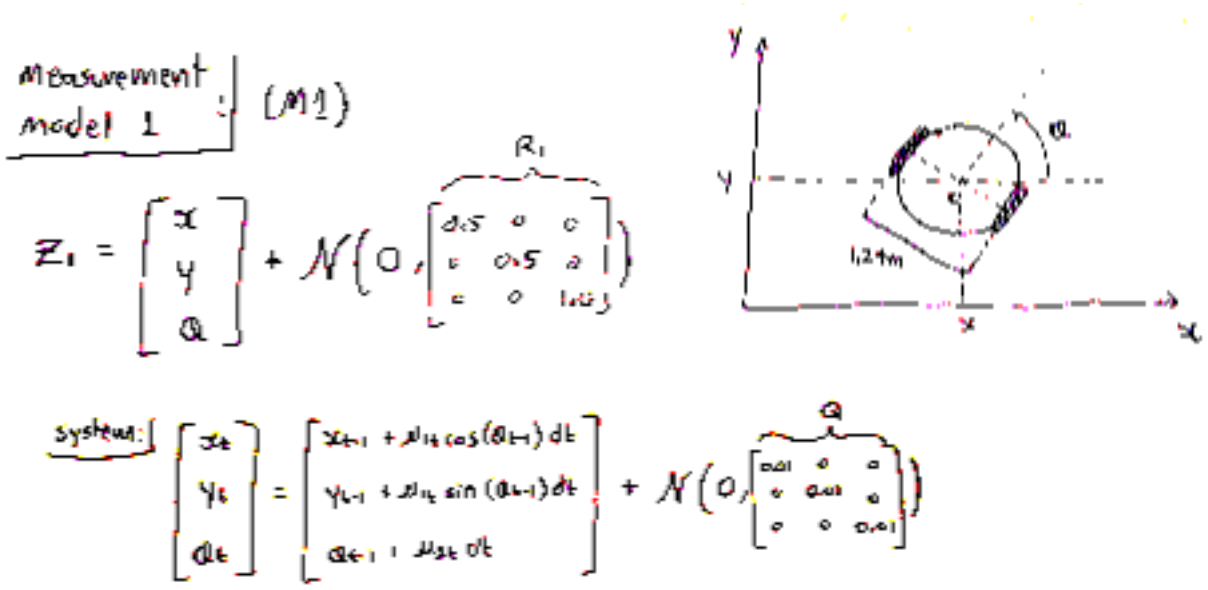


Figure 1: Motion & Measurement Model Illustration

The initial conditions are chosen arbitrarily, with the robot starting at the point (3.0, 2.0)=(x, y) with an initial yaw angle of 45 degrees counterclockwise from the horizontal. It is ensured that the circular path that the robot traces is fully enclosed within the particle filter map. The baseline of the robot is set to a constant value of 1.24m. The pose of the robot is retrieved with an indoor localization technique that returns a noisey x,y,theta state.

The following assumptions are made in this solution:

1) The distance and angle moved by the robot for each increment of the independent variable chosen (time in this case) must be infinitesimally small.
2) The robot is a non-holonomic robot with 2 inputs and 3 states. The states of focus in this solution are the x and y poses.
3) No slippage occurs between the robot's wheels and the ground.

4) The time-step (dt) is infinitesimally small such that the simulation approaches the real world scenario, ie) model is sufficiently discretized (by Euler Discretization).
5) Model uncertainty can be modeled by a gaussian distribution with a mean zero and covariance matrix Q (kept constant in this implementation).
6) Measurement noise can be modeled by a gaussian distribution with a mean zero and covariance matrix R (kept constant in this implementation).
7) Markov chain assumption, meaning that the next state is only dependent on the previous state and the input.

To tackle this localization problem a particle filter is used to estimate the pose (which throughout this paper will be used to refer to the x,y position) of the robot. In this problem we are writing a particle filter implementation from scratch and clarifying how the filter was implemented as well as investigating how changing the particle number parameter in the particle filter algorithm for the localization of the robot affects the final estimation accuracy and computational requirement.

General Motivation of use of Particle Filter

The particle filter algorithm is known to come with a high computational cost which is proportional to the resolution of the map and the number of particles (given constant simulation parameters) defined by the algorithm. The main motivation behind using particle filters is that they are tractable for high-dimensional problems whereas Kalman Filters are not. It also allows for robot localisation in problems with the following characteristics:

- **Multimodal:** Wish to track multiple objects simultaneously
- **Occlusions:** One object can hide another, resulting in one measurement for multiple objects
- **Nonlinear behavior:** Of the motion model, similar to the applicability of the Extended Kalman Filter vs. the Kalman Filter
- **Nonlinear measurements:** For example radar gives us the distance to an object Converting that to an (x,y,z) coordinate requires a square root, which is nonlinear
- **Non-Gaussian noise:** The random quantity does not have to satisfy any given distribution
- **Continuous:** the object's position and velocity (i.e. the state space) can smoothly vary over time
- **Multivariate:** we want to track several attributes, such as position, velocity, turn rates, etc
- **Unknown process model:** we may not know the process model of the system

Compared to the implementation of the Extended Kalman Filter to the same problem (Question 4 of Assignment 1), the Particle Filter algorithm is less sensitive to strong nonlinearities and non-Gaussian noise. However, the problem to be tackled in this project is kept simplistic and does not take full advantage of the versatility of the particle filter. This is

done as focus is dedicated to investigating the particle filter's naive implementation from scratch and further parameter (specifically the number of particles) optimization.

Particle Filter Implementation Details

An overview of the algorithm as well as how each step/ stage of the process is implemented is given below. The simulation is run for a total of 3 seconds with a time step of 0.1 seconds. Note that the specifics of this implementation are the focus rather than a general overview of the generic particle filter algorithm.

1. **Generate an initial distribution of particles:** Particles each have an x position, y position, heading angle (theta) and weight. The weight (probability) indicates how likely it matches the actual state of the system given a measurement. The initial particle distribution is taken as uniform to fit the resolution of the map (particle every dx = 0.1 and dy = 0.1). The weights of each initialized particle are set to unity (these will be changed upon measurement). Thus, the initial particle filter map is shown below (Figure 2):

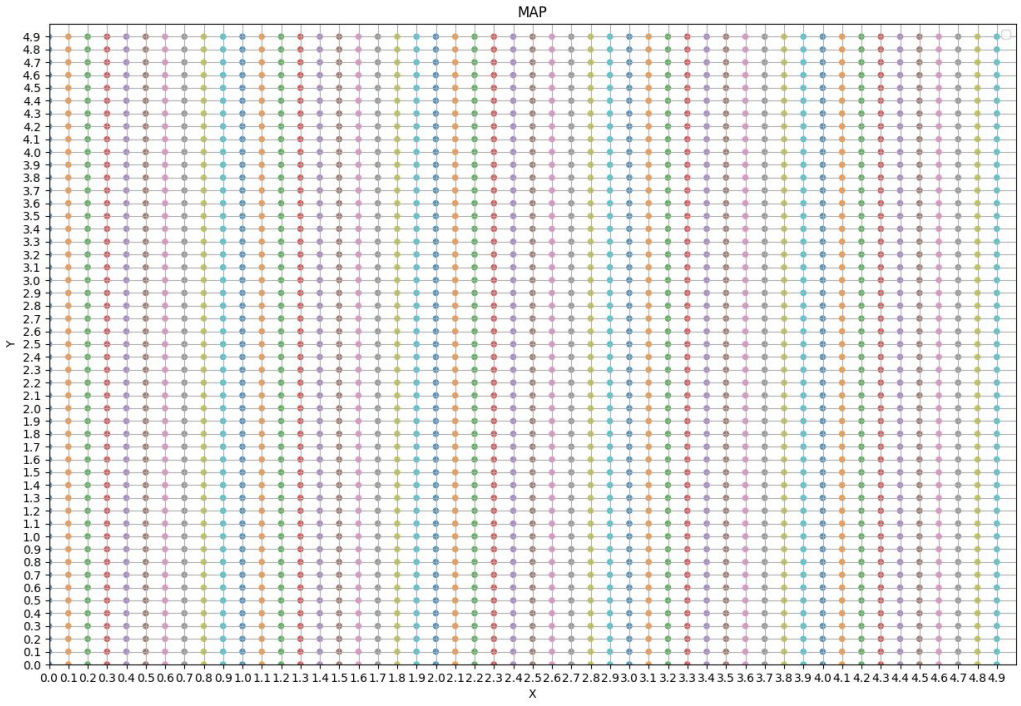


Figure 2: Initial Particle Filter Map

The above distribution of particles is only the case for the first initialization of the particle filter map. Following every state estimate at the end of each simulation time step all

particles in the map are (as above) initialized to a weight value of 1.0. Note that the other state variables for each particle are left unchanged.

2. **Predict the next state of the particles:** In this step we iterate through all the particles in the map and use the motion model (with noise) to predict the next pose of each particle. Using the motion model to predict the next state of a particle may result in some particles having a predicted state that lies outside the ranges of the particle filter map. In this implementation these particles are discarded. This is a reasonable thing to do as the map dimensions fully enclose the robot's simulated trajectory such that discarded particles (near the edges of the map) are irrelevant to the eventual state estimation. However, this can also be an indication that there are more particles on the map than necessary, leading to a higher variance in the pose estimation as well as an increased computational time (further investigation on this conducted later).
3. **Update particle weights using measurement:** Here we update the weighting of the particles based on the measurement. Particles that are closer to the state suggested by the measurement are given a higher weight than particles that are further. The weight is calculated using a probability density function to model the likelihood of the measured state (with noise) and the predicted measured state.

   In this implementation weights are calculated in this way for the x-position, y-position and theta heading angle, then summed to return a final single float value for the weight of the corresponding particle. All weights are then normalized by the maximum weight assigned to the particles (requires max value search) such that all weights now lie between 0 and 1 (projected as probabilities).

4. **Control the number of particles:** Before resampling a new distribution of particles, it is optional to limit the number of particles in the map to ensure that the number of particles does not 'blow up' due to new generated/ resampled particles in the following step. The number of particles was limited to 1000 (decided upon by trial and error). The particles removed are those with the lowest weight value (requires a min value search), as these do not represent the true state as well as particles with a higher weight. These improbable particles are only removed if the max number of particles is exceeded at this step.
5. **Resample from the particle map:** The goal of this step is to introduce particles that have a higher likelihood of representing the true state. This is done by generating a number of particles for each particle that has a weight greater than defined probability value.

   However, this method does not allow for the generation of a defined number of new particles. Rather, the number of generated particles is arbitrary. Furthermore, if existing particles on the map do not represent the true state suggested by the measurement (have a low weight) then no particles are generated. The two parameters that define this process are the particle_weight_boundary (=0.9) and the generation_number (=1). These quantities refer to the minimum weight value of a particle for it to be considered a

source particle (a particle from which more particles may be generated) and the number of particles to be generated from a given source particle in one step of the simulation respectively.

Generated particles are assigned a weight equal to that of the particle from which they originate (source particle). Generated particles are also placed in the vicinity of the source particle, with the deviation from the source particle modeled by a gaussian distribution with a zero mean and standard deviation equal to the grid resolution (=0.1). This value was chosen by trial and error. This ensures that generated particles are spread from the original source particles (in this case the spread is limited to immediate neighbouring cells).

6. **Compute a state estimate:** Finally, a weighted average (using the particle weights) is calculated for the x and y position separately to form a deterministic position estimation. Similarly, the weighted variance of the x and y coordinates of the particles are determined. This gives a measure of confidence of the position estimation, the lower the weighted variance of a pose, the higher the likelihood that the estimated pose resembles the true robot pose.

Results and Discussion

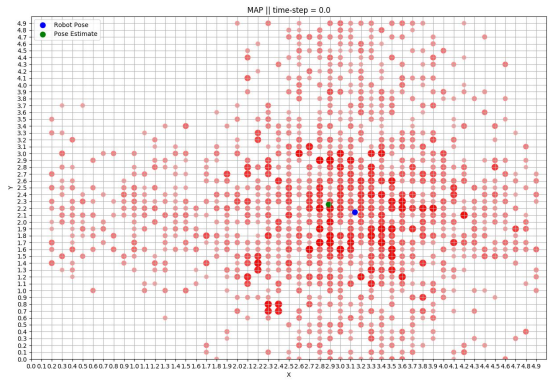
The accuracy of the particle filter algorithm largely depends on the algorithm's variable parameters (not inclusive of the motion or measurement model's defining parameters). These are the following:

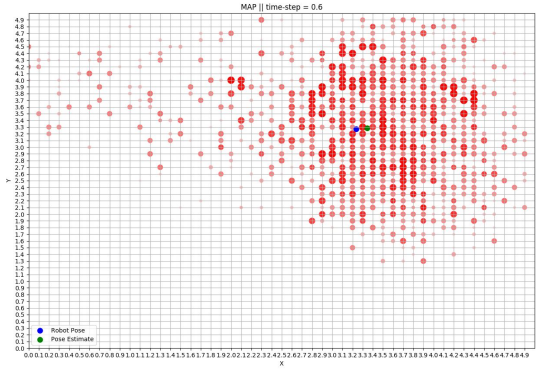
1. **Grid resolution/ map size:** The map size is taken as 50X50 (=2500 cells) with a grid resolution of 0.1 in dx and dy. These dimensions and resolution are kept constant.
2. **Number of particles:** This varies, however, limiting the number of particles when it becomes too large a quantity allows for faster computation. The maximum number of particles before new particle generation/ sampling is limited to 1000. Note that in the following discussions; 'Particle Number' is equivalent to 'Number of Particles'.
3. **Particle weight boundary:** Only particles that have a weight value that exceeds this (particles that better estimate the true state) will generate new particles. By trial and error, a value of 0.9 was chosen. This parameter value is kept constant.
4. **Particle generation number:** This corresponds to the number of particles a single particle is allowed to generate. This is set to a value of 1, and was chosen by trial and error. This value is kept constant.
5. **Generated particle spread:** Generated particles deviate from the source particle by a random distance modeled by a gaussian distribution with a zero mean and standard deviation equal to the grid resolution (=0.1). This measure of particle spread is kept constant for all generated particles.

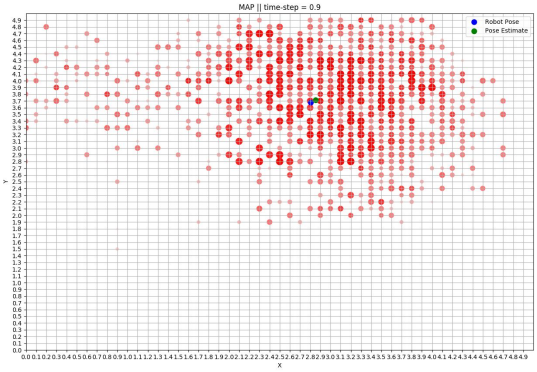
Running the simulation returns a particle filter map for each time step. In the map the resulting weighted average pose estimation and the actual robot pose are also displayed. In

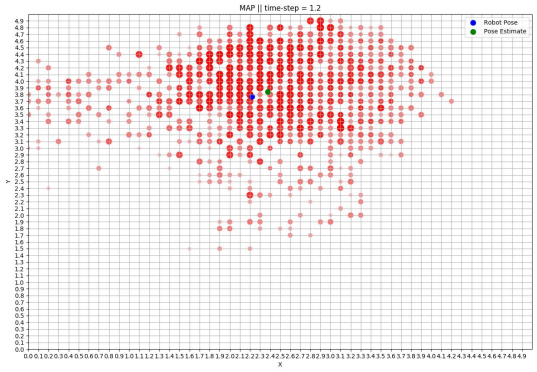
Table 1 we present the resulting particle filter map for only 11 timesteps shown with intervals of 0.3s. Note that the time interval at which each particle map was snapshotted is included in each map's title.

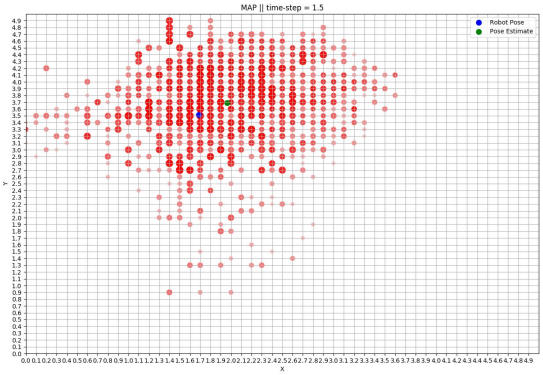
Table 1: Simulation Progression (0.3s intervals) (Particle Number = 1000)

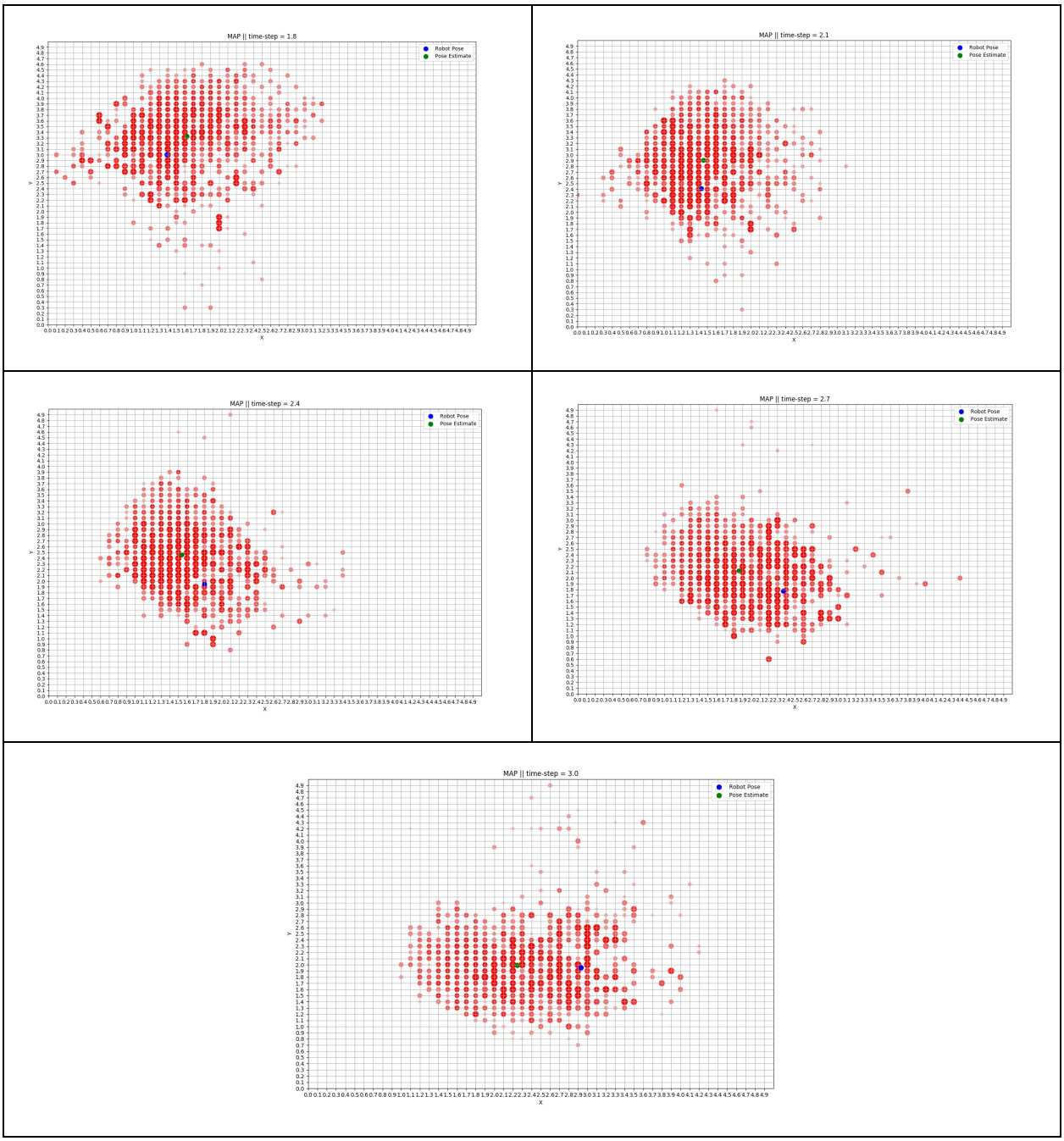












As can be seen from the above progression of the particle filter algorithm, the particles do in fact appear to condense about the true robot pose in attempt to track it. However, it can be seen that the spread of the particles suggest a low confidence in the estimated pose (even in the later timesteps) and as time progresses in the simulation the estimated pose lags further behind the true robot pose. This results in the final actual and estimated pose paths illustrated below.

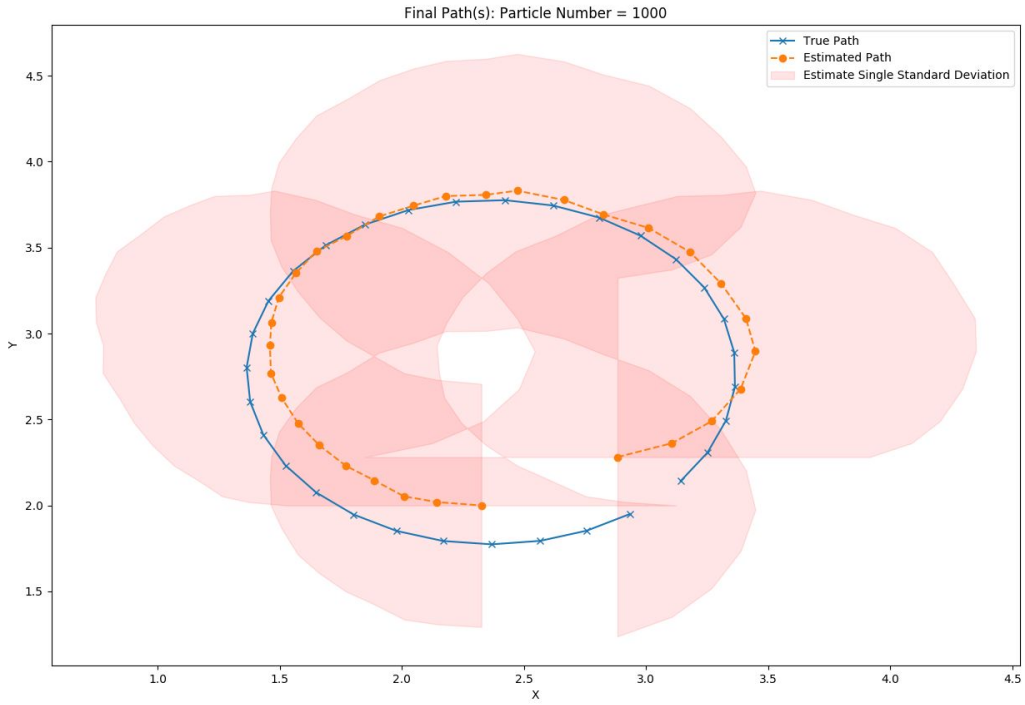


Figure 3: Final Path(s): Particle Number = 1000

As can be seen from the localization plot, the tracking capability of the particle filter appears to be limited. One reason for the high error in the particle filter state estimate is the limited resolution of the map (50x50) used in order to save on computational time. Although the majority of the true pose trajectory is within a single standard deviation from the estimated trajectory the high variance of the estimations suggests a low confidence in the pose estimations. The high variance is due to the large number of particles with a large spread. However, reducing the number of particles or the generation of new particles/ spread of newly generated particles could result in an inability to track the true state. This may result in a diverging estimate and in a worst case scenario; no final state estimate due to the absence of particles from the final map.

In order to evaluate the overall point to point tracking accuracy of the algorithm, a sum of the absolute differences between the state estimate poses and their respective true robot poses (that they attempt to track). Thus, a higher accuracy would correspond to a lower value of the sum. We refer to this as the trajectory tracking error.

$$error = \sum_{traj.} \left[\left| x_{estimate} - x_{true} \right| + \left| y_{estimate} - y_{true} \right| \right]$$

We will now proceed to conduct further investigation on the number of particles parameter in the following section.

Investigative Algorithm Adjustments

In this section analysis of the effects of altering the 'number of particles' parameter is conducted in order to estimate the optimal number of particles as well as investigate the results when non-optimally setting this parameter. With a decrease in the number of particles parameter there is a trade off in terms of decreased estimate variance and computational cost, however, this may result in less accurate pose estimates (the opposite is also true). These factors are evaluated in the following manner (the following applies to a single run of a simulation):

- **Measure of overall estimate variance:** Taken as an average of x and y pose variances (separately) for the simulated trajectory
- **Measure of computational requirement:** Represented by the total time taken to run the algorithm for a simulated trajectory
- **Error of pose estimates:** Overall error of tracking of the true robot poses by the estimated poses quantified by the trajectory tracking error (previously defined)

Running the algorithm for different values of the number of particles parameter results in the following.

Table 2: Effects of Changing the Number of Particles Parameter

NUMBER OF PARTICLES (#)	**COMP. TIME (*s*) [4 d.p]**	**ERROR (m) [4 d.p]**	**AVG. X-VAR. (m^2) [4 d.p]**	**AVG. Y-VAR. (m^2) [4 d.p]**
50	18.6692	0.7334	0.3522	0.4305
100	23.7581	0.3119	0.3903	0.3759
500	60.1955	0.5185	0.5355	0.5538
1000	103.2571	0.6573	0.7426	0.6680
1500	159.3831	0.7367	0.7988	0.7223
2000	226.4930	0.7528	0.8721	0.7999

Which are represented by the following trajectories (with the particle number = 1000 case above in Figure 3):

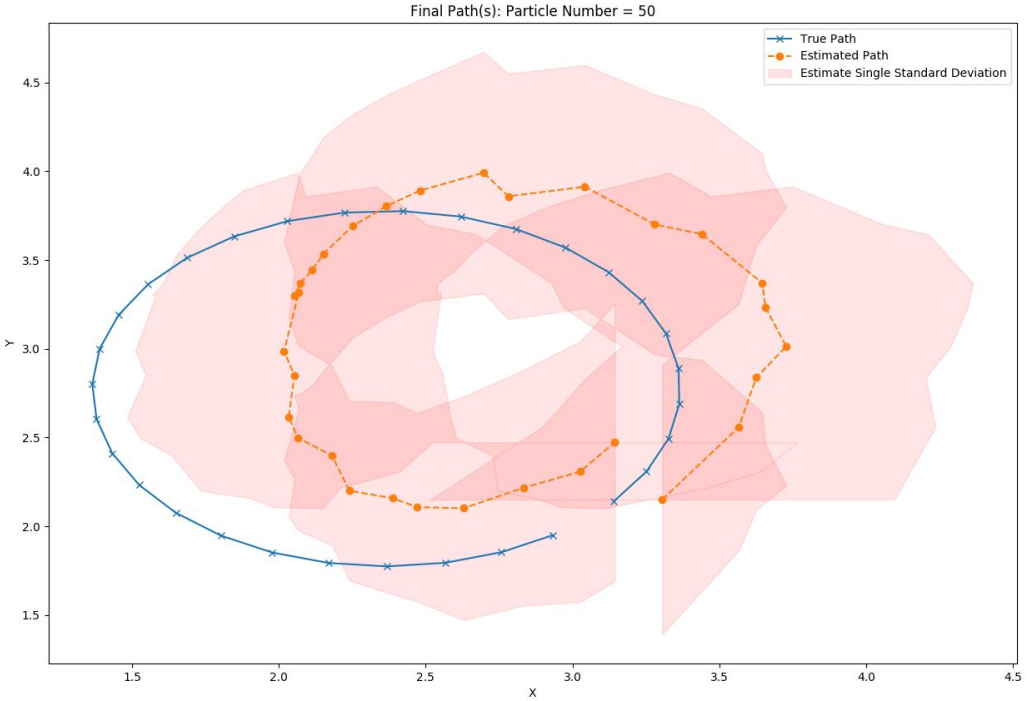


Figure 4: Final Path(s): Particle Number = 50

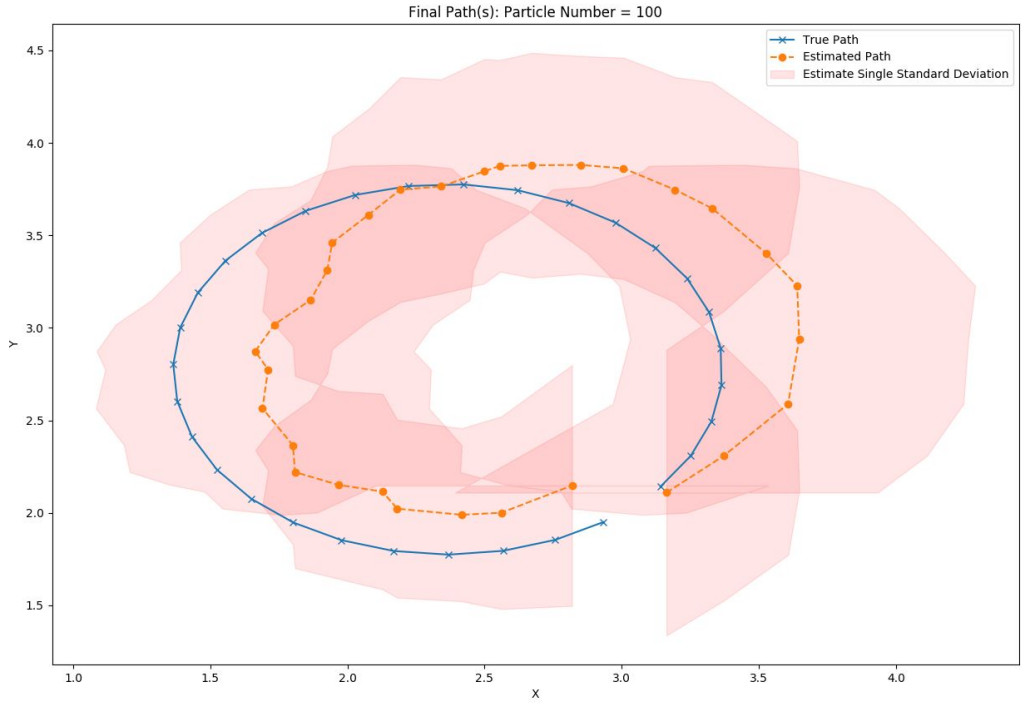


Figure 5: Final Path(s): Particle Number = 100

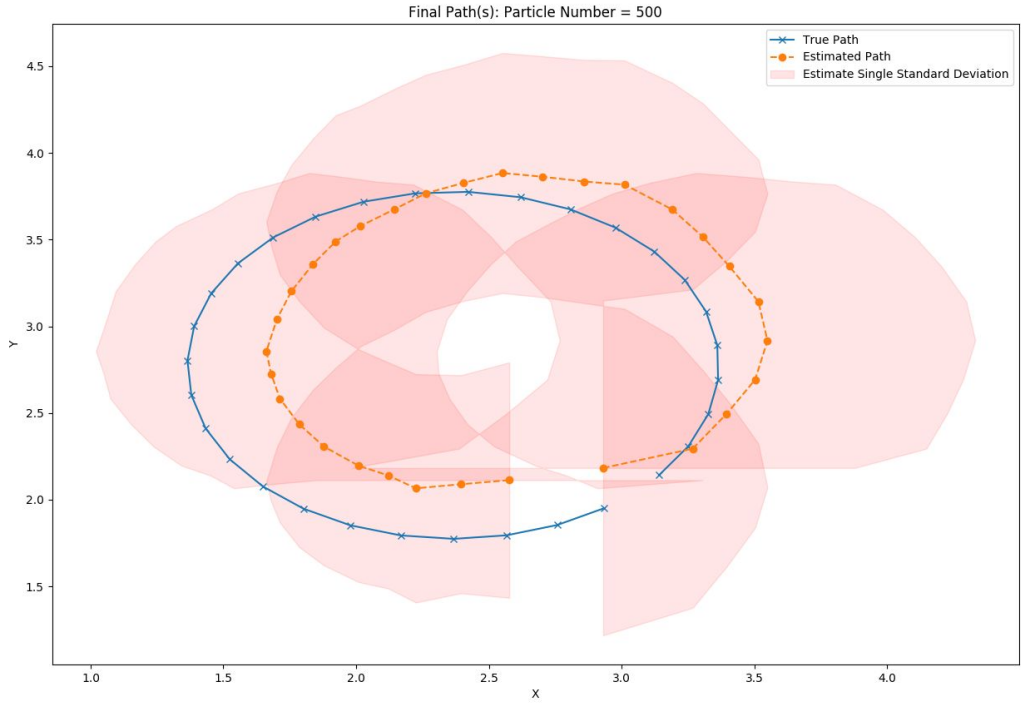


Figure 6: Final Path(s): Particle Number = 500

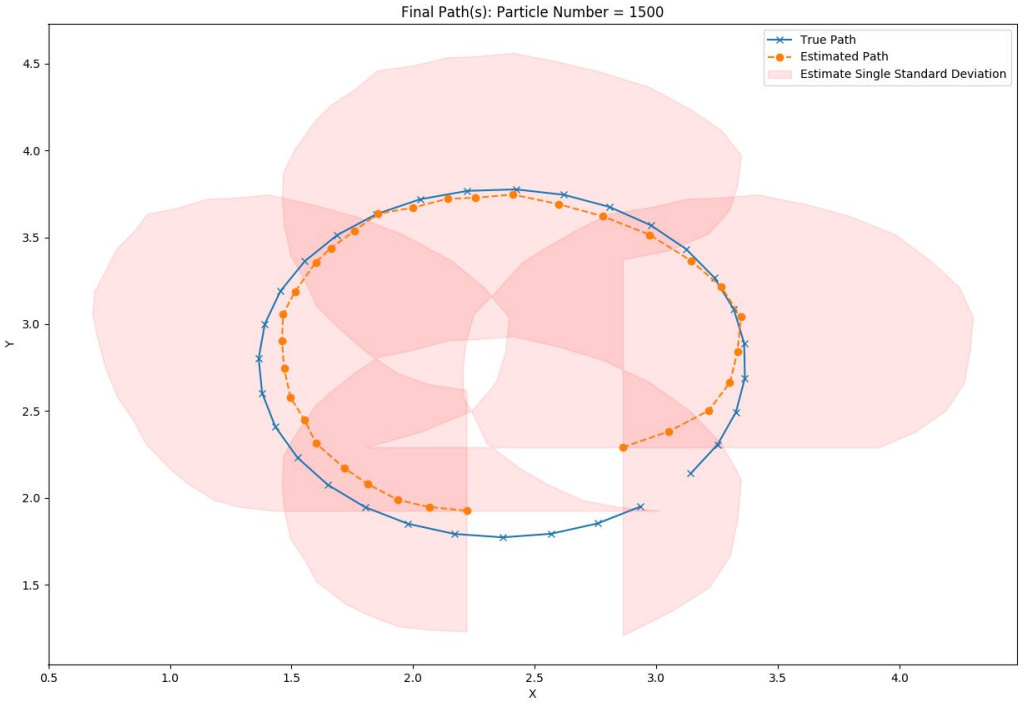


Figure 7: Final Path(s): Particle Number = 1500

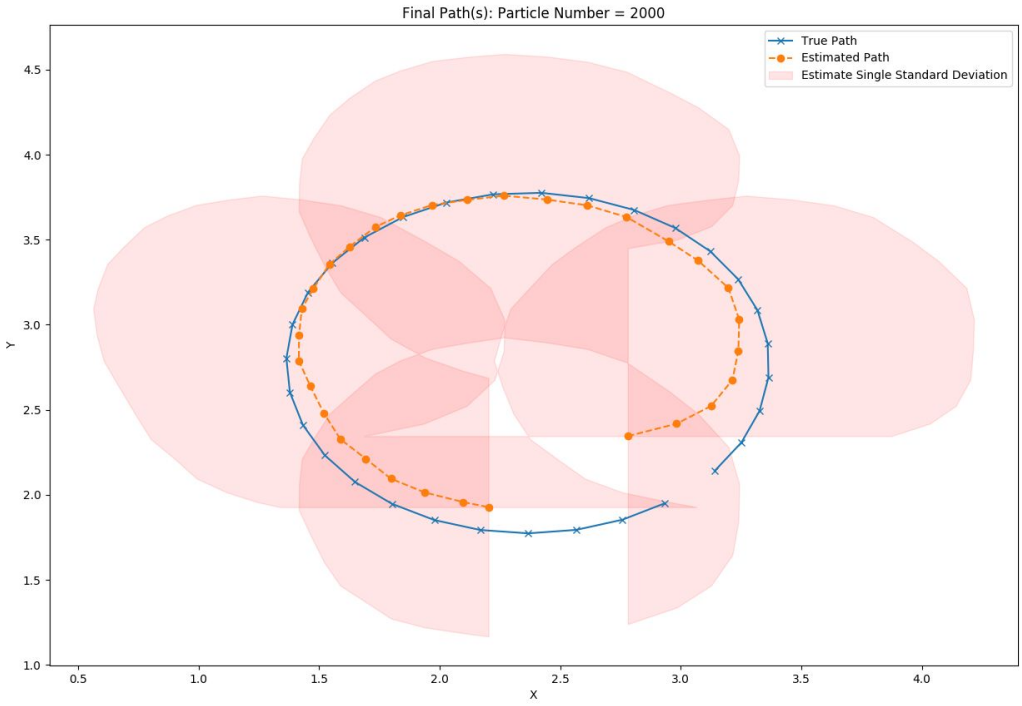


Figure 8: Final Path(s): Particle Number = 2000

To evaluate the ideal particle number for this problem; with all other algorithmic parameters kept constant, we can generate plots of our previously mentioned evaluative quantities (presented in Table 2).

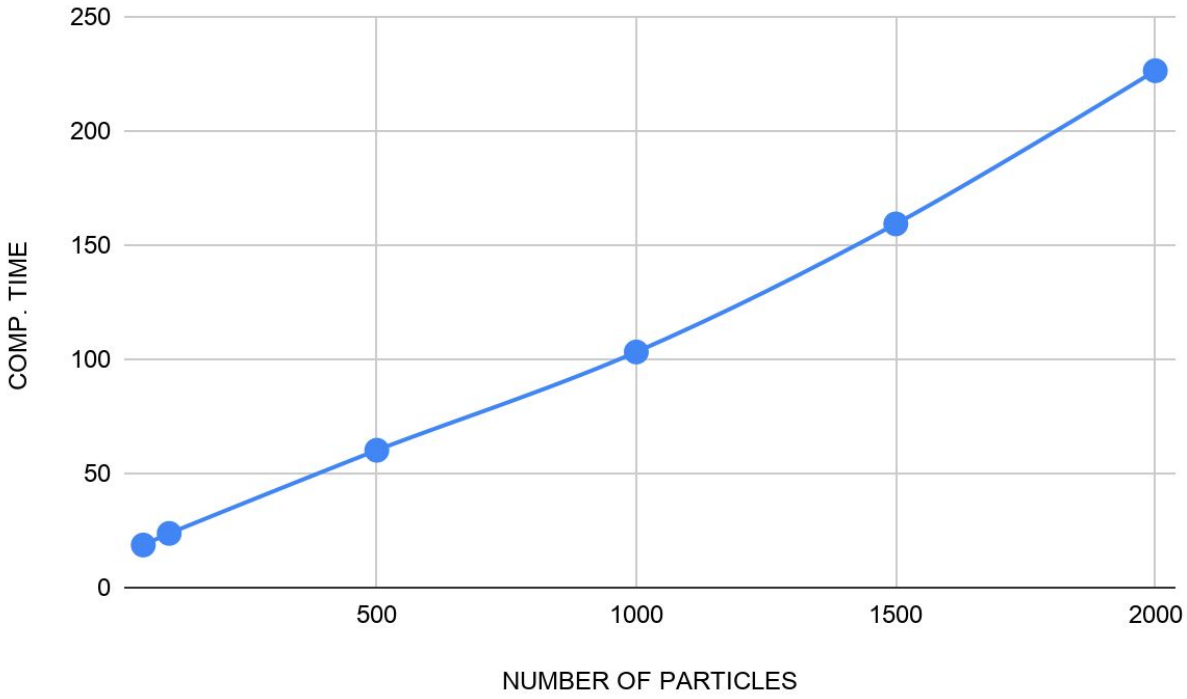


Figure 9: Time of Computation vs. Number of Particles

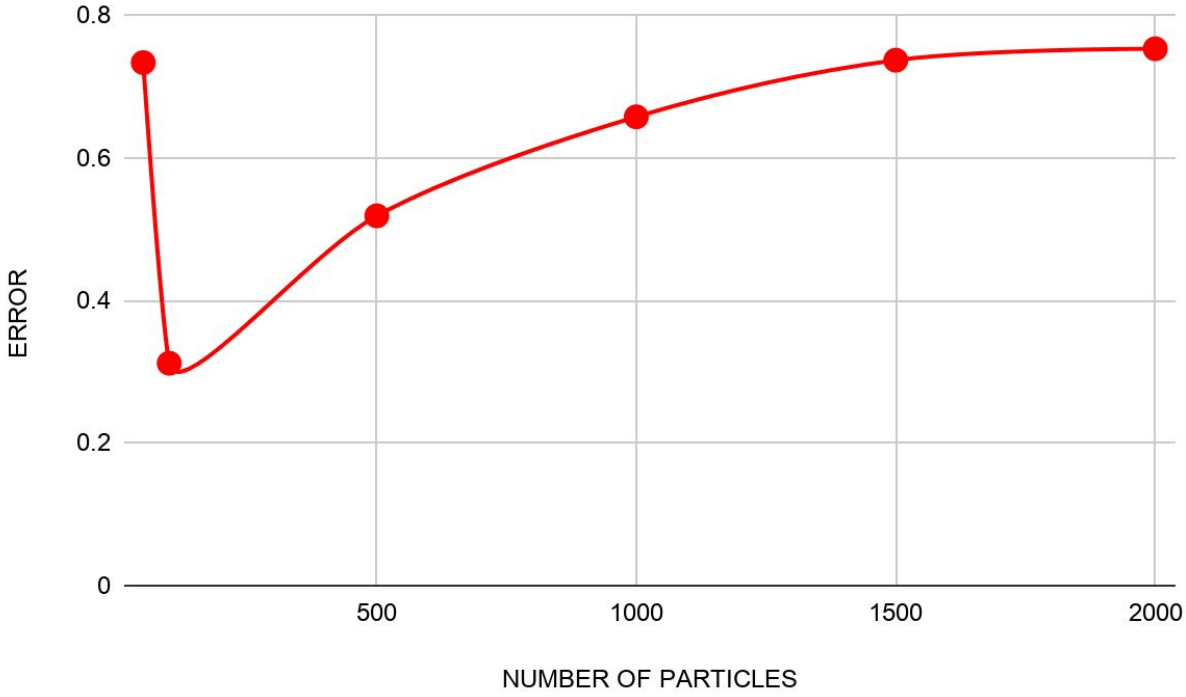


Figure 10: Trajectory Error vs. Number of Particles

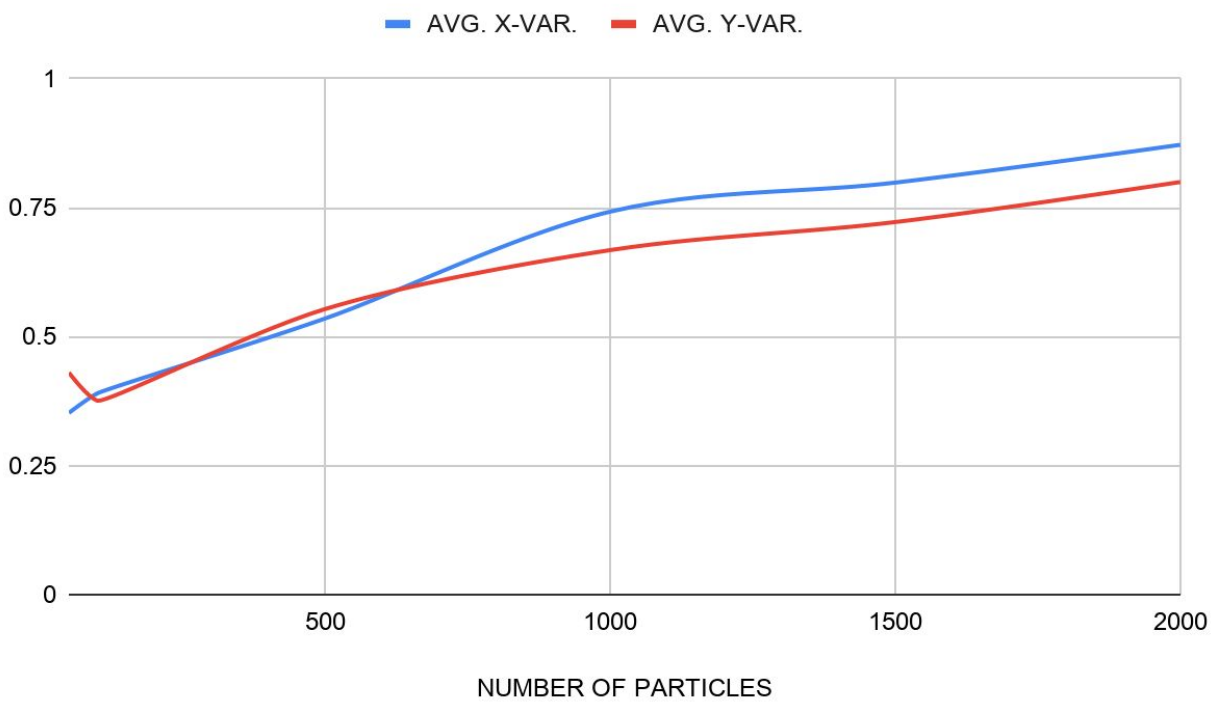


Figure 11: X/Y Pose Variance(s) vs. Number of Particles

As the goal of the particle filter is to track the true trajectory as accurately as possible, the settings that return the lowest trajectory error will be the most desired. As can be seen from the above plots this corresponds to a particle number of 100. Setting the particle number higher than this value results in an increase in the trajectory tracking error. This is primarily due to the 'lag' introduced by an excessive number of particles resulting in many highly weighted particles following the true robot pose while they are a source for further particle generation. This lag effect can be seen to become more prominent with a further increase in the particle number and for later pose estimates (increase in lag seen in Figures 6-8).

We also observe an increase in the pose variance with an increase in the number of particles. As our removal of particles involves removing particles with the lowest weight first until the desired particle number is reached, a larger number of particles entails that lower weight particles that are further from the true robot pose (suggested by the measurement) remain, thus, increasing the variance of the pose estimate.

Also note that as the number of particles increases, the time required for computation also increases at what can be observed on Figure 9 to be a near linear rate. Thus, the lowest number of particles possible is desirable. However, if the particle number is too low (such as the case of 50 particles) the tracking algorithm's pose estimations begin to diverge from the true robot pose (as can be observed in Figure 4). This is due to the fact that the resulting small

cluster of particles may lose the true robot pose in very few time steps or even a single time step due to the particle cluster's limited span on the map.

Conclusion

Particle filters are particularly useful when dealing with Bayesian problems with arbitrary probability distributions and a large number of states, and thus are a popular implementation of complex localization problems. They even allow for on-the-fly/ real time trade off between accuracy and computational complexity. However, naively applying particle filtering to fault diagnosis can lead to poor tracking of the most important states as the filter is (as proven here with the number of particles parameter) sensitive to changes in its defining parameters.

In this project the effect of controlling the number of particles on the map to a certain quantity was investigated and it was concluded that this parameter plays a large role in obtaining an optimal particle filter algorithm. However, this analysis was done while keeping other algorithmic parameters constant and the same type of analysis should be conducted on these parameters (outlined at the beginning of the Results and Discussion section) to understand the influence they have on the particle filters optimality.

Appendix (Code)

MTE544_PF_2.py

```
#import autograd.numpy as np
import numpy as np
import time as timer
import autograd
from autograd import jacobian
import matplotlib.pyplot as plt
import matplotlib
import _tkinter
import scipy
from scipy.stats import norm
matplotlib.rcParams['figure.figsize'] = (15.0, 10.0)

# on using autograd to find Jacobian:
# https://stackoverflow.com/questions/49553006/compute-the-jacobian-matrix-in-python
# ============================================================
t0 = timer.time()

# The inputs are custom
def u1_const (t):
   return 2.0

def u2_const (t):
   return 2.0

def u1_rand_sin (t):
   return np.random.normal(0, 0.1, 1)[0] + np.sin(t)

def u2_rand_sin (t):
   return np.random.normal(0, 0.1, 1)[0] + np.cos(t)

# The system is as follows
def x1_calc (x, t, input_func):
   return x[0] + input_func(t)*np.cos(x[2])*time_step

def x2_calc (x, t, input_func):
   return x[1] + input_func(t)*np.sin(x[2])*time_step

def x3_calc (x, t, input_func2):
   return x[2] + input_func2(t)*time_step

# ================== first measurement model....
def y1_calc (x):
   return x[0]

def y2_calc (x):
   return x[1]
```

```
def y3_calc (x):
   return x[2]

def y1_calc_noise (x):
   return x[0] + np.random.normal(0, R[0, 0], 1)[0]

def y2_calc_noise (x):
   return x[1] + np.random.normal(0, R[1, 1], 1)[0]

def y3_calc_noise (x):
   return x[2] + np.random.normal(0, R[2, 2], 1)[0]

# ================= second measurement model....
def y1_calc_m2 (x):
   return np.sqrt(np.square(x[0])+np.square(x[1]))

def y2_calc_m2 (x):
   return np.tan(x[2])

def y1_calc_noise_m2 (x):
   return np.sqrt(np.square(x[0])+np.square(x[1])) + np.random.normal(0, R_m2[0, 0], 1)[0]

def y2_calc_noise_m2 (x):
   return np.tan(x[2]) + np.random.normal(0, R_m2[1, 1], 1)[0]

# The conditions are
x_0 = 3.0
y_0 = 2.0
q_0 = 45.0*(np.pi/180)
robot_init_pos = np.array([x_0, y_0, q_0])

baseline = 1.24
time_step = 0.1

time = 0.0
time_array = np.array([0.0])
x1 = x_0
x2 = y_0
x3 = q_0
x_state = np.array([x1, x2, x3], dtype=float)
x_state_m2 = np.array([x1, x2, x3], dtype=float)
y1 = 0.0
y2 = 0.0
x_state_cov = np.zeros((3, 3))
x_state_cov_m2 = np.zeros((3, 3))
# save arrays for later plotting
predicted_states = np.array([0, 0, 0])
corrected_states = np.array([0, 0, 0])
predicted_states_m2 = np.array([0, 0, 0])
```

```
corrected_states_m2 = np.array([0, 0, 0])
# for uncertainty/ noise modelling
# R for measurement => GIVEN!
R = np.array([[0.5, 0.0, 0.0], [0.0, 0.5, 0.0], [0.0, 0.0, 1.0]])
R_m2 = np.array([[0.5, 0.0], [0.0, 1.0]])
# Q for model. 0.5, 0.5, 0.1
Q_const_m1 = 0.01  # originally 0.0133
Q_const_m2 = 0.01
Q = np.array([[Q_const_m1, 0.0, 0.0], [0.0, Q_const_m1, 0.0], [0.0, 0.0, Q_const_m1]])
Q_m2 = np.array([[Q_const_m2, 0.0, 0.0], [0.0, Q_const_m2, 0.0], [0.0, 0.0, Q_const_m2]])

# ------------------------------ Calculating the value of R/Q
rq_m1 = np.trace(R)/np.trace(Q)
rq_m2 = np.trace(R_m2)/np.trace(Q_m2)

# --------------------------------- Create a 2D map
pf_map = np.ones((50, 50)).astype(object)  # the prior belief (50X50)
# within each map grid cell, there is a list of particles
# each particle consists of an array (x, y, theta). The grid is split as follows
dx = 0.1
dy = 0.1
# we limit the number of particles (=1000)
NP = 2000
# the rate of particle generation (=1) allow us to keep the map populated
generation_number = 1
# only if a particles weight is above 0.9, then that particle will generate particles
particle_weight_boundary = 0.90
# first place a single particle in every map grid cell. in each cell we have a list of
tuples
counter = 0
for row_index, row in enumerate(pf_map):
   for col_index, lis in enumerate(row):
       pf_map[row_index, col_index] = [(col_index*dx, row_index*dy, 0.0, 1.0)]  # x, y,
theta, weight
       # pf_map[row_index, col_index].append((col_index*dx, row_index*dy, 45.0*(np.pi/180),
1.0))
       # pf_map[row_index, col_index].append((col_index * dx, row_index * dy, 90.0 * (np.pi
/ 180), 1.0))
       # pf_map[row_index, col_index].append((col_index * dx, row_index * dy, 135.0 * (np.pi
/ 180), 1.0))
       # pf_map[row_index, col_index].append((col_index * dx, row_index * dy, 180.0 * (np.pi
/ 180), 1.0))
       # pf_map[row_index, col_index].append((col_index * dx, row_index * dy, 225.0 * (np.pi
/ 180), 1.0))
       # pf_map[row_index, col_index].append((col_index * dx, row_index * dy, 270.0 * (np.pi
/ 180), 1.0))
       # pf_map[row_index, col_index].append((col_index * dx, row_index * dy, 315.0 * (np.pi
/ 180), 1.0))
       print("updated...")
```

```
# not let us plot the grid! (should be an even spread)
# fig = plt.figure()
# for row_index, row in enumerate(pf_map):
#     for col_index, lis in enumerate(row):
#         print("plotted: ", (row_index, col_index))
#         print("state/ lis: ", lis[0])
#         plt.scatter(lis[0][0], lis[0][1], alpha=0.8, edgecolors='none', s=30)
# plt.title('MAP')
# axes = plt.gca()
# axes.set_xticks(np.arange(0, pf_map.shape[0]*dx, dx))
# axes.set_yticks(np.arange(0, pf_map.shape[1]*dy, dy))
# axes.set_xlim([0.0, pf_map.shape[0]*dx])
# axes.set_ylim([0.0, pf_map.shape[1]*dy])
# plt.grid()
# plt.xlabel('X')
# plt.ylabel('Y')
# plt.legend()
# plt.show()

robot_pos = robot_init_pos
fig1 = plt.figure()
fig1.canvas.draw()
plt.show(block=False)

# store the paths....
robot_path_x = np.array([0.0])
robot_path_y = np.array([0.0])
estimate_path_x = np.array([0.0])
estimate_path_y = np.array([0.0])
estimate_var_x = np.array([0.0])
estimate_var_y = np.array([0.0])

loop_count = 0
while time <= 3:  # original = 3 seconds
   # ========================================================= Particle Filter
   # make all the weights of every particle = 1
   for row_index, row in enumerate(pf_map):
       for col_index, lis in enumerate(row):
           for elem_index, elem in enumerate(lis):
               lis[elem_index] = (elem[0], elem[1], elem[2], 1.0)
   # >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> PREDICTION
   # actual position of the robot?
   robot_temp_actual_pos = np.array([x1_calc(robot_pos, time, u1_const),
                                     x2_calc(robot_pos, time, u1_const),
                                     x3_calc(robot_pos, time, u2_const)])
   robot_pos = robot_temp_actual_pos
   robot_path_x = np.append(robot_path_x, robot_pos[0])
   robot_path_y = np.append(robot_path_y, robot_pos[1])
   # first step all particles forward using the motion model
   for row_index, row in enumerate(pf_map):
```

```
        for col_index, lis in enumerate(row):
            #if len(lis) == 0:
            #    break
            for elem_index, elem in enumerate(lis):
                print("list element as tuple: ", elem)
                print("old point: ", elem)
                particle_forward = np.array([[x1_calc(elem, time, u1_const) +
np.random.normal(0, Q[0, 0], 1)[0]],
                                             [x2_calc(elem, time, u1_const) + np.random.normal(0,
Q[1, 1], 1)[0]],
                                             [x3_calc(elem, time, u2_const) + np.random.normal(0,
Q[2, 2], 1)[0]]])
                print("delete element: ", lis[elem_index])
                del lis[elem_index]
                # now place the new position of the particle on the map
                x_temp = round(particle_forward[0][0], 1)
                x_index_temp = int(round(x_temp/dx, 0))
                #print("x index: ", x_index_temp)
                #print("x value: ", x_temp)
                y_temp = round(particle_forward[1][0], 1)
                y_index_temp = int(round(y_temp/dy, 0))
                theta_temp = particle_forward[2][0]
                #print("map shape: ", pf_map.shape)
                print("new point: ", (x_temp, y_temp, theta_temp, 1.0))
                # Now calculate the weight for the point
                state_temp_y = np.array([x_temp, y_temp, theta_temp])
                y_state_pred_noise = np.array([y1_calc_noise(robot_pos),
                                               y2_calc_noise(robot_pos),
                                               y3_calc_noise(robot_pos)])
                # now calculate the weight.....
                weights = norm.pdf(y_state_pred_noise, state_temp_y, R)
                weight = np.trace(weights)
                print("weight assigned: ", weight)
                # if we are still on the map, save point
                if y_index_temp < pf_map.shape[0] and x_index_temp < pf_map.shape[1]:
                    pf_map[y_index_temp, x_index_temp].append((x_temp, y_temp, theta_temp,
weight))
                    #pf_map[y_index_temp, x_index_temp] = [(x_temp, y_temp, theta_temp, 1.0)]
    # Now to go through the entire map and normalize the weights
    weight_max = 0.0
    for row_index, row in enumerate(pf_map):
        for col_index, lis in enumerate(row):
            for elem_index, elem in enumerate(lis):
                if elem[3] > weight_max:
                    weight_max = elem[3]
    print(">>>>> MAX OF WEIGHTS: ", weight_max)
    # then dividing all the weights by the max found weight
    for row_index, row in enumerate(pf_map):
        for col_index, lis in enumerate(row):
            for elem_index, elem in enumerate(lis):
```

```
                temp_weight = elem[3] / weight_max
                lis[elem_index] = (elem[0], elem[1], elem[2], temp_weight)
                print("final Normalized (by weight) element: ", lis[elem_index])

    # Now we ensure that we are below the max particle number before generating new particles
    particle_number = 0
    for row_index, row in enumerate(pf_map):
        for col_index, lis in enumerate(row):
            for elem_index, elem in enumerate(lis):
                particle_number = particle_number + 1
    print("The number of particles: ", particle_number)
    while particle_number > NP:  # delete low weight particles
        min_weight = 1.0
        for row_index, row in enumerate(pf_map):
            for col_index, lis in enumerate(row):
                for elem_index, elem in enumerate(lis):
                    if elem[3] < min_weight:
                        min_weight = elem[3]
                        min_particle = np.array([row_index, col_index, elem_index])
        print("element to delete: ",
pf_map[min_particle[0]][min_particle[1]][min_particle[2]])
        del pf_map[min_particle[0]][min_particle[1]][min_particle[2]]
        particle_number = particle_number - 1
    # Now we sample from this distribution to get new particles
    # if its probable we keep the particle, otherwise remove it
    #for row_index, row in enumerate(pf_map):
    #    for col_index, lis in enumerate(row):
    #        for elem_index, elem in enumerate(lis):
    #            if elem[3] > np.random.normal(0, 1, 1):
    #                print("keep particle, weight = ", elem[3])
    #            else:
    #                del lis[elem_index]

    # now generate particles from existing particles!
    # generate from the highest weighted particles first!
    # pf_map_fixed = pf_map.copy()  # for generating particles and avoiding generating from
generated particles
    # while particle_number < (NP + NP_2):
    #     max_weight = 0.0
    #     for row_index, row in enumerate(pf_map_fixed):
    #         for col_index, lis in enumerate(row):
    #             for elem_index, elem in enumerate(lis):
    #                 if elem[3] > max_weight:
    #                     max_weight = elem[3]
    #                     max_particle = np.array([elem[0], elem[1], elem[2]])
    #     # now generate particle around this particle...
    #     for i in range(0, generation_number):
    #         del_pos_x = np.random.normal(0, 0.1, 1)[0]
    #         del_pos_y = np.random.normal(0, 0.1, 1)[0]
    #
```

```
    #         x_temp = round(max_particle[0] + del_pos_x, 1)
    #         print("x_temp: ", x_temp)
    #         x_index_temp = int(round(x_temp / dx, 0))
    #         print("x_index_temp: ", x_index_temp)
    #         y_temp = round(max_particle[1] + del_pos_y, 1)
    #         print("y_temp: ", y_temp)
    #         y_index_temp = int(round(y_temp / dy, 0))
    #         print("y_index_temp: ", y_index_temp)
    #         theta_temp = max_particle[2]
    #         if 0 <= y_index_temp < pf_map.shape[0] and 0 <= x_index_temp < pf_map.shape[1]:
    #             pf_map[y_index_temp, x_index_temp].append((x_temp, y_temp, theta_temp,
1.0))
    #             particle_number = particle_number + 1
    #         else:
    #             print("generated particle OUT OF RANGE")
    pf_map_fixed = pf_map.copy()  # for generating particles and avoiding generating from
generated particles
    for row_index, row in enumerate(pf_map_fixed):
        for col_index, lis in enumerate(row):
            for elem_index, elem in enumerate(lis):
                if elem[3] > particle_weight_boundary:
                    print("Particle x,y: ", (elem[0], elem[1]))
                    # now generate particle around this particle...
                    for i in range(0, generation_number):
                        del_pos_x = np.random.normal(0, 0.1, 1)[0]
                        del_pos_y = np.random.normal(0, 0.1, 1)[0]

                        x_temp = round(elem[0] + del_pos_x, 1)
                        print("x_temp: ", x_temp)
                        x_index_temp = int(round(x_temp / dx, 0))
                        print("x_index_temp: ", x_index_temp)
                        y_temp = round(elem[1] + del_pos_y, 1)
                        print("y_temp: ", y_temp)
                        y_index_temp = int(round(y_temp / dy, 0))
                        print("y_index_temp: ", y_index_temp)
                        theta_temp = elem[2]
                        if 0 <= y_index_temp < pf_map.shape[0] and 0 <= x_index_temp <
pf_map.shape[1]:
                            # give the generated particle the same weight as the particle is
came from
                            pf_map[y_index_temp, x_index_temp].append((x_temp, y_temp,
theta_temp, elem[3]))
                            particle_number = particle_number + 1
                        else:
                            print("generated particle OUT OF RANGE")
    print("Final number of particles: ", particle_number)
    # now calculate a state estimate
    # do this be modelling a gaussian for the robot pose
    # ie) get a mean and variance
    x_sum = 0.0
```

```
    x_divider = 0.0
    y_sum = 0.0
    y_divider = 0.0
    x_vals = np.array([0.0])
    y_vals = np.array([0.0])
    weights_array = np.array([0.0])
    estimate_array = np.array([0.0, 0.0, 0.0, 0.0])
    particle_count = 0
    for row_index, row in enumerate(pf_map):
        for col_index, lis in enumerate(row):
            for elem_index, elem in enumerate(lis):
                # here the average/ mean (=estimated pose) is weighted
                x_sum = x_sum + elem[0]*elem[3]
                x_divider = x_divider + elem[3]
                y_sum = y_sum + elem[1]*elem[3]
                y_divider = y_divider + elem[3]
                x_vals = np.append(x_vals, elem[0])
                weights_array = np.append(weights_array, elem[3])
                y_vals = np.append(y_vals, elem[1])
                particle_count = particle_count + 1
    #x_mean = x_sum/particle_count
    x_mean = x_sum/x_divider
    #y_mean = y_sum/particle_count
    y_mean = y_sum/y_divider
    #x_var = np.var(x_vals[1:])
    x_var = np.sqrt(np.cov(x_vals[1:], aweights=weights_array[1:]))
    #y_var = np.var(y_vals[1:])
    y_var = np.sqrt(np.cov(y_vals[1:], aweights=weights_array[1:]))
    # we attempt to estimate the robot position only!
    state_estimate = np.array([x_mean, y_mean, x_var, y_var])
    #print("Thus the x/y mean and x/y variance (respectively): ", state_estimate)
    # save the estimated path.....
    estimate_path_x = np.append(estimate_path_x, state_estimate[0])
    estimate_path_y = np.append(estimate_path_y, state_estimate[1])
    estimate_var_x = np.append(estimate_var_x, state_estimate[2])
    estimate_var_y = np.append(estimate_var_y, state_estimate[3])

    error_sum = 0.0
    # Calculate the error in the position estimate
    error_sum = error_sum + abs(robot_pos[0] - state_estimate[0]) + abs(robot_pos[1] -
state_estimate[1])
    # now we can PLOT the new particle distribution map!
    print(">>>>>>>>>>>>>>>> completed update loop with time: ", round(time, 1))
    # fig1 = plt.figure()
    if (loop_count % 3) == 0 or loop_count == 0 or loop_count == 1:
        for row_index, row in enumerate(pf_map):
            for col_index, lis in enumerate(row):
                for elem_index, elem in enumerate(lis):
                    plt.scatter(elem[0], elem[1], alpha=1.0*elem[3], color='red',
edgecolors='none', s=10*(elem[3]*15))
```

```
        plt.scatter(robot_pos[0], robot_pos[1], alpha=1.0, color='blue', edgecolors='none',
s=90, label='Robot Pose')
        plt.scatter(state_estimate[0], state_estimate[1], alpha=1.0, color='green',
edgecolors='none', s=90, label='Pose Estimate')
        map_string = "MAP || time-step = {}".format(round(time, 1))
        plt.title(map_string)
        plt.legend()
        axes = plt.gca()
        axes.set_xticks(np.arange(0, pf_map.shape[0]*dx, dx))
        axes.set_yticks(np.arange(0, pf_map.shape[1]*dy, dy))
        axes.set_xlim([0.0, pf_map.shape[0]*dx])
        axes.set_ylim([0.0, pf_map.shape[1]*dy])
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.grid()
        fig1.canvas.draw()
        print(">>>>>>>>>>>>>>>> CUMULATIVE ESTIMATION ERROR = ", error_sum)
        timer.sleep(0.1)
        plt.clf()
        #plt.show()


    # increment time step...
    time = round(time + time_step, 1)
    time_array = np.append(time_array, np.array([time]))
    # increment the loop count (helps plot every 3 steps)
    loop_count = loop_count + 1
    print("=================================== TIME-STEP: ", time)
# ======================================================================
t1 = timer.time()
print("Total Time taken: ", (t1-t0))
print(">>>>>> TOTAL ESTIMATION ERROR: ", error_sum)
print(">>>>>>>>>>>>>>>>> SIMULATION COMPLETE")
print("Averaged x variance: ", np.mean(estimate_var_x[1:]))
print("Averaged y variance: ", np.mean(estimate_var_y[1:]))

print("..... now we can plot the entire paths...")
fig_final = plt.figure()
ax = plt.axes()
ax.plot(robot_path_x[1:], robot_path_y[1:], label='True Path', linestyle='-', marker='x')
ax.plot(estimate_path_x[1:], estimate_path_y[1:], label='Estimated Path', linestyle='--',
marker='o')
# plotting y variance
y = estimate_path_y[1:]
error = np.sqrt(estimate_var_y[1:])
plt.fill_between(estimate_path_x[1:], y-error, y+error, alpha=0.1, color='red',
label='Estimate Single Standard Deviation')
# plotting x variance
x = estimate_path_x[1:]
error2 = np.sqrt(estimate_var_x[1:])
```

```
plt.fill_betweenx(estimate_path_y[1:], x-error2, x+error2, alpha=0.1, color='red')
final_string = "Final Path(s): Particle Number = {}".format(NP)
plt.title(final_string)
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()

print("================== CODE COMPLETE ==================")
```