SEPTEMBER 17, 2021

ESP3201

PARTICLE FILTER MINI-ASSIGNMENT

GOH KHENG XI, JEVAN
A0199806L
NUS Engineering

Contents

1. Introduction

Particle filtering is a recursive state estimator using Bayesian logic [1]. It is commonly used in non-linear and non-gaussian system which provides advantages over Kalman filtering [1]. The state space is represented by numerous particles which individually represents the tracking target itself. As the target performed an action, the state space is updated and using Monte Carlo methods, the particles will be resampled and eventually converged to the true state [1].

1.1 Aim

A python skeleton code describing a particle filter algorithm for a two-wheeled robot is given. The robot's motion is described by r1, r2 and t, which corresponds to the left wheel rotation, right wheel rotation and forward translation respectively. The sensors onboard the robot can sense its range from each landmark.

The problem statement given is to complete the python code for both the eval_sensor_model() and the resample_particles() functions.

2. Code

The main.py file contains the main() function that gets called when the file is ran, which in turn calls out other functions. The particle_filter.py contains the functions for the particle filter which includes the 2 incomplete functions and the read_data.py contains the functions to parse the data files. The pos.dat file contains the initial ground truth state of the robot while the world.dat contains the information on the landmarks.

2.1 Particles

From the main.py, it is understood that each particle is a dictionary that contains the key 'x' and 'y' which corresponds to the x and y coordinates of the particles in the environment. Each particle also contains the key "theta" that represents the orientation of the robot. The values for each of these properties are initialised randomly (within the environment limits) for each particle and the number of particles is set to 1000 which is a hyperparameter that should be tuned when designing a particle filter.

2.2 Sensor

From the codes, the sensors reading is a dictionary that consist of 3 keys, "lm_ids", "ranges", "bearings" which represents the specific landmark, the ranges from each landmark and the bearings of each landmark respectively. However, the sensors appear to be reading only the robots' range from each landmark and not the bearings. Hence, it is implied that the range should be used as the basis of comparison to determine the importance weight of each particle.

2.3 Landmarks

The world.dat contains the information of 4 landmarks with ids, '1', '2', '3' and '4', which is shown on the first column. The x and y coordinates of each landmark are represented by the second and third column respectively.

3. Eval_sensor_model()

This function is responsible for calculating the likelihood of each particle being the ground truth target and hence, the weight or probability of the particle being chosen for resampling. The function takes in three parameters which are the sensor data, the particles' information, and the landmarks' information.

To calculate the likelihood, ranges of each particle from each landmark are calculated and then compared to the range measured by the onboard sensors (reason for using range as a basis for comparison is because that is the only data measured). Next the likelihood of each particle is calculated based on this error term and then normalised to obtain the importance weight of each particle.

3.1 Range

As the range of each particle from each landmark is not given directly, it must be calculated from the particle's current x and y coordinates and the landmark's x and y coordinates.

$$r_{i,j} = \sqrt{\left(X_i - x_j\right)^2 + \left(Y_j - y_i\right)^2} \qquad \text{(equation 1)}$$

$$i: particle$$

$$j: landmark$$

$$r: range$$

$$X: x - coordinate\ of\ particle$$

$$x: x - coordinate\ of\ landmark$$

$$Y: y - coordinate\ of\ particle$$

$$y: y - coordinate\ of\ particle$$

Equation 1 measures the Euclidean distance of the particle from each landmark to obtain the range.

3.2 Likelihood

The likelihood of each particle for each landmark is calculated based on the following equation:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(r-\mu)^2}{2\sigma^2}} \qquad \text{(equation 2)}$$

In equation 2, r represents the actual measurement which is the actual range of the particle from the landmark, μ represents the sensor measurement of the particle's range from the landmark and σ represents the measurement standard deviation which is given as 0.2 in the code (denoted by sigma_r in the code). However as this is the posterior estimate based on a single landmark and since the measurement of all the landmarks are simultaneous events, the likelihood of the specific particle is the multiplication of the posteriori estimates based on each landmark.

$$f(x)_i = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\left(x_{i,n} - \mu_{i,n}\right)^2}{2\sigma^2}} \qquad \text{(equation 3)}$$

Equation 3 provides the more complete formulation for the likelihood of each particle, where i represents the specific particle, n represents the specific landmark and N represents the total number of landmarks which is 4 in this problem statement.

Since the likelihood represents the Bayesian probability that the particle is the target given the sensor measurements, the likelihood for each particle calculated in equation 3 needs to be normalised to obtain the importance weight, such that $0 < w_i < 1$.

$$w_i = \frac{f(x)_i}{\sum_{k=1}^{K=1000} f(x)_k} \qquad \text{(equation 4)}$$

In equation 4, the importance weight of a particle is normalised by dividing the likelihood of that particle by the sum of the likelihood of each particle, and then stored into a numpy array and returned by the function.

4. Resample_particles()

This function is responsible for the resampling of the particles based on their importance weight. It takes in two parameters which are the particles' information and their respective importance weight obtained by eval_sensor_model() (section 3) to resample the particles using stochastic universal sampling.

4.1 Stochastic universal sampling

Stochastic universal sampling is a sampling method that places discretise a line or wheel into portions representing each particle with the portion size proportional to their respective weight. Pointers are then placed at fixed intervals among the dataset and then choosing the samples where the pointers are pointing at, favouring samples with higher weight. [2]

4.2 Coding the resampling

To simulate the stochastic universal sampling method, each particle was coded as a python list of size 2 with the first element being the lower bound of its portion's location and the second element being the upper bound of its portion's location. Next a random floating point number between the range of 0 and 1/N was generated using numpy.random.uniform(), where N represents the current total number of particles. This is to generate the location of the first pointer and as I loop through the dataset, if the pointer points in between any particle's lower and upper bound, the particle will be chosen for the next sample, and the pointer is increased by 1/N. The process repeats until the pointer exceeds the value of 1 and this simulates the placing the placing of N pointers around the "data wheel".

5. Results

The programme is ran with a few changes in addition to the completed functions. Firstly, the number of iterations is set to 200 and using the sleep() function from the time module in python, the iterations and performance of the particle filter can be studied more closely.

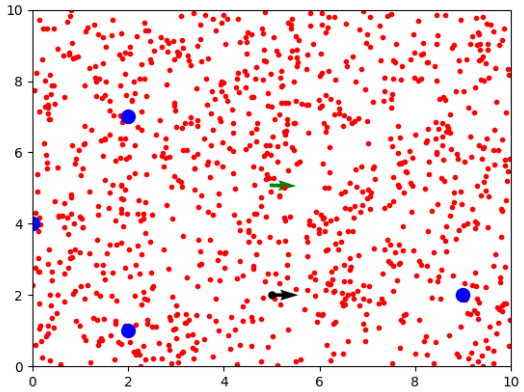

Figure 1. 0th iteration

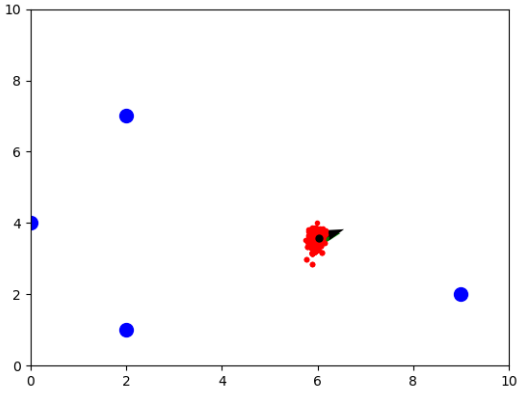

Figure 2. 50th iteration

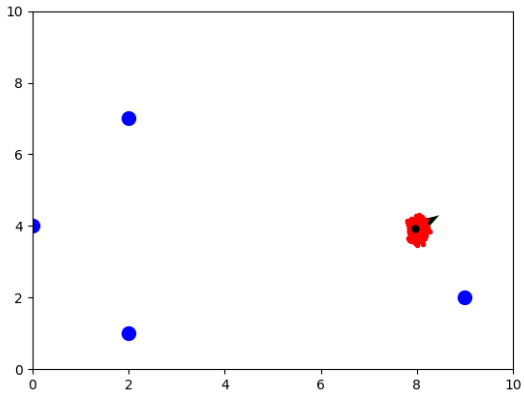

Figure 4. 100th iteration

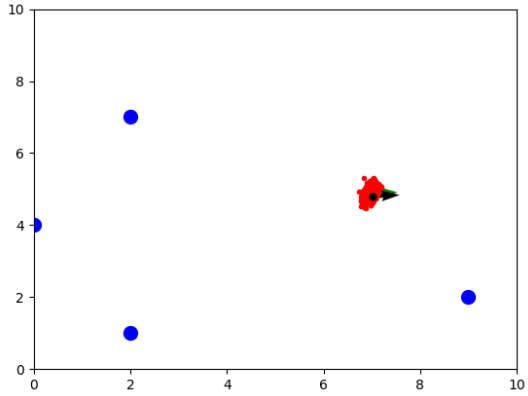

Figure 3. 150th iteration

Figure 1, 2, 3 and 4 shows the results of the particle filter where the black arrow represents the actual position and orientation of the robot, and the green position represents the position and orientation of the robot estimated by the particle filter.

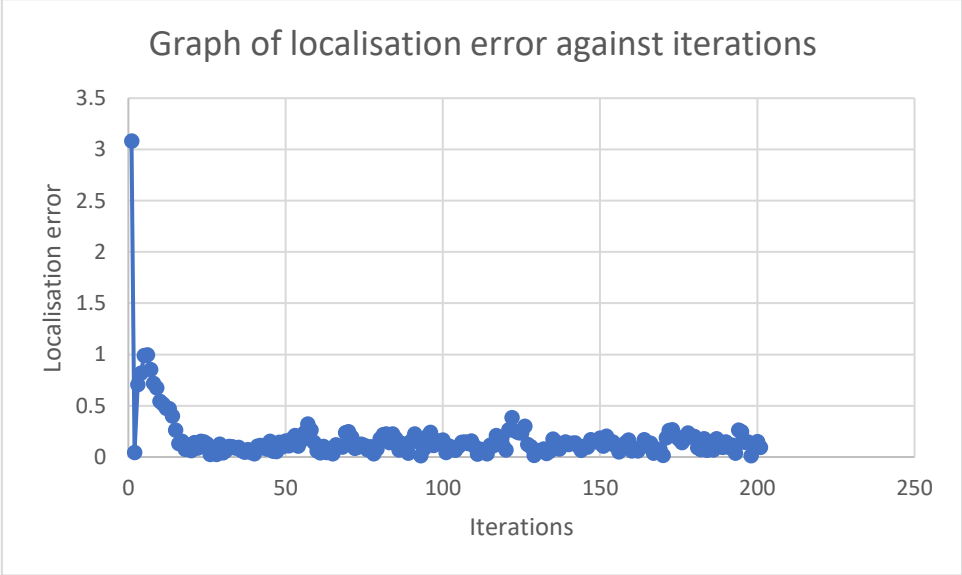

Figure 5. Graph of localisation error against iterations

The localisation error is recorded and plotted on a graph in figure 5. It shows a fast convergence and a low variance in the localisation error from approximately the 50th iteration onwards.

6. Tuning

Parameters of the particle filter were varied and adjusted in order to observe the difference in the particle filter's performance. This section describes the observations and conclusions derived from varying the number of particles, the number of landmarks and the measurement noises.

6.1 Varying number of particles

The number of particles is varied from 140 to 10 000 to observe its impact on the localisation error convergence.

Figure 6. Convergence results with varying number of particles

The convergence results in figure 6 shows that although the convergence is generally faster as the number of particles increases, the convergence speed has diminished returns as the number of particles increases. There seems to be a boundary where the increase in the number of particles sees the highest increase in convergence speed and this boundary occurs around between the 160 and 180 number of particles.

The results are expected due to the “bagging” (bootstrap aggregating) effect of the particle filter; as the number of particles increase, the impact of the outlying particles is reduced to provide a more accurate estimation of the target. However, the aggregating effect is reduced as the number of particles increases.

6.2 Varying number of landmarks

The number of landmarks is also varied from 2 to 40 landmarks and the results are recorded in figure 7 below.

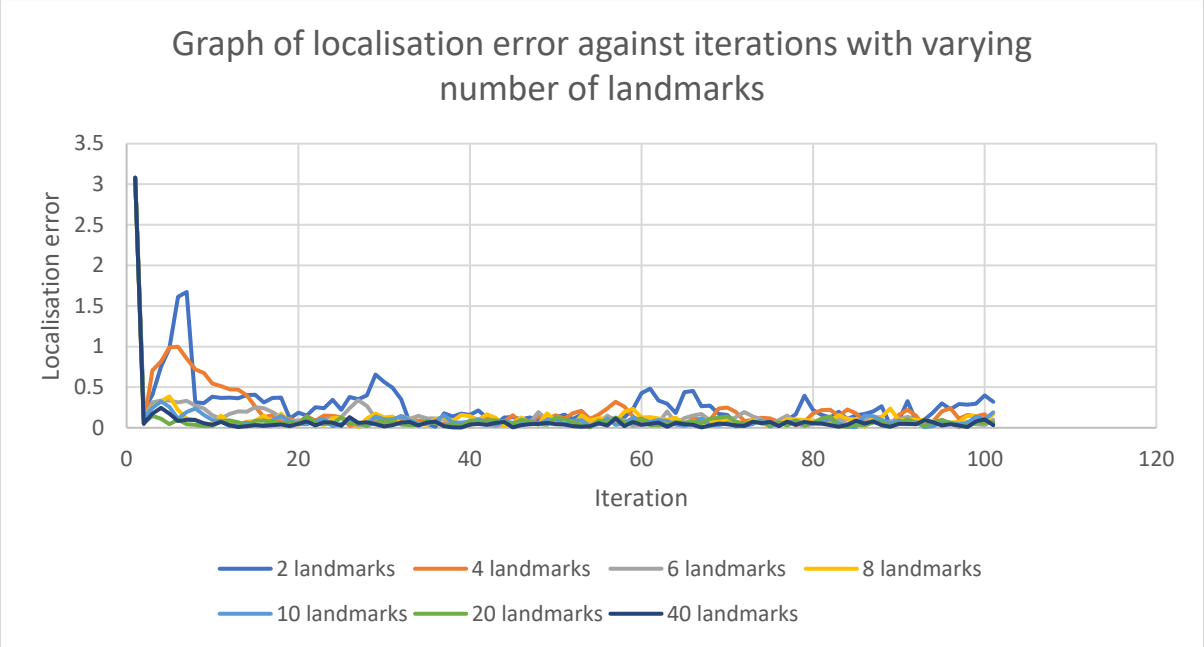

Figure 7. Convergence results with varying number of landmarks

The results in figure 7 shows that the convergence of the particle filter is generally faster as the number of landmarks increases. This is because an increase in the number of landmarks increases the amount of sensor data and hence, reduces the impacts of sensor noise. However, the increase in convergence speed also diminishes as the number of landmarks increases due to the reducing aggregating effect.

6.3 Varying measurement noise

The measurement noise used in the calculation of the likelihood function is also varied to observe its effects on the particle filter.

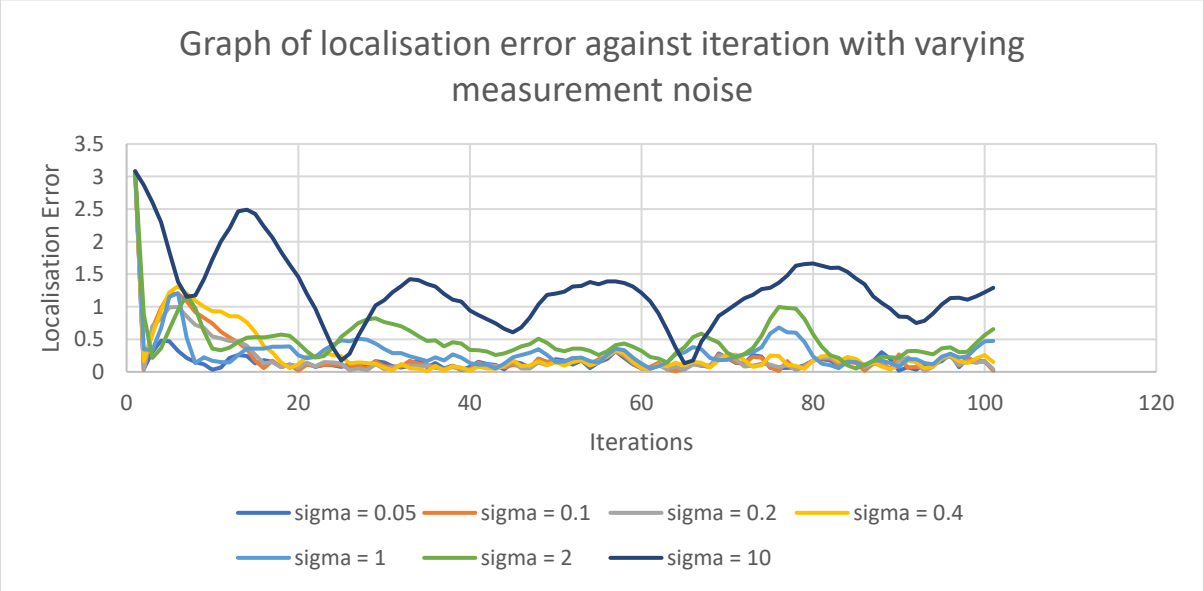

Figure 8. Convergence results with varying measurement noise

The results are shown in figure 8 where in general, the particle filter converges slower as the measurement noise increases and strays away from the actual sensor noise.

7. References

[1] S. Srinivasan, “Particle Filter : A hero in the world of Non-Linearity and Non-Gaussian,” Towards Data Science, 14 August 2019. [Online]. Available: https://towardsdatascience.com/particle-filter-a-hero-in-the-world-of-non-linearity-and-non-gaussian-6d8947f4a3dc. [Accessed 25 September 2021].

[2] Asti Dwi Irfianti, Retantyo Wardoyo, Sri Hartati and EndangSulistyoningsih, “DETERMINATION OF SELECTION METHOD IN GENETIC,” in *EDP Sciences*, 2016.

Appendix A

Code for eval_sensor_model() function

```
def eval_sensor_model(sensor_data, particles, landmarks):
    # Computes the observation likelihood of all particles, given the
    # particle and landmark positions and sensor measurements
    #
    # The employed sensor model is range only.

    sigma_r = 0.2

    #measured landmark ids and ranges
    ids = sensor_data['id']
    ranges = sensor_data['range']



    weights = []


    '''your code here'''
    for particle in particles: # for each particle
        weight = 1 # this weight have to multiply by each weight for each land
mark
        for i in range(len(landmarks)): # for each landmark
            particle_range = math.sqrt((particle['x']-
landmarks[i+1][0])**2 + (particle['y'] - landmarks[i+1][1])**2) #range of the
particle from the landmark
            ''' start of likelihood function '''
            exponential_numerator = (-
1) * ((sensor_data["range"][i] - particle_range)**2)
            exponential_denominator = 2 * ( sigma_r ** 2)
            weight_denom = math.sqrt(2 * math.pi * (sigma_r ** 2))
```

```
            weight *= math.exp(exponential_numerator/exponential_denominator)
/ weight_denom # multiplies the weight of previous landmarks
            ''' end of likelihood function'''

        weights.append(weight)

    weights = np.array(weights) # changing eights to numpy array


    '''***        ***'''


    #normalize weights
    normalizer = sum(weights)
    weights = weights / normalizer

    return weights
```

Appendix B

Code for resample_particles()

```
def resample_particles(particles, weights):
    # Returns a new set of particles obtained by performing
    # stochastic universal sampling, according to the particle weights.

    new_particles = []

    '''your code here'''

    wheel = []
    wheel.append([0, weights[0]])


    for i in range(len(weights)-1):
        wheel.append([wheel[i][1], wheel[i][1]+weights[i+1]])

    interval = 1/len(weights)
    current_pointer = np.random.uniform(0, 1/len(weights))


    it = 0
    while(it < len(weights)):
        if current_pointer > 1:
            break
        elif current_pointer >= wheel[it][0] and current_pointer < wheel[it][1
]:
            new_particles.append(particles[it])
            current_pointer += interval
```

```
        else:
            it += 1



    '''***        ***'''


    return new_particles
```