16-833: Robot Localization and Mapping, Spring 2022

Homework 1

Robot Localization using Particle Filters

Submitted by :

Aditya Ramakrishnan [aramakr2]

Chinmay Garg [chinmayg]

Taruna Sudhakar [tsudhaka]

Introduction

In this assignment, we are given a robot and are tasked with localizing the robot. The robot luckily has a map of the environment and noisy laser sensor and control data.

We will implement a 2-dimensional particle filter in python to better understand how the robot goes about localization.

The inputs to the particle filter are a map of Wean Hall (map_reader.py and the respective map ".c" and ".dat" files) and log files that define the robot odometry and laser sensor information.

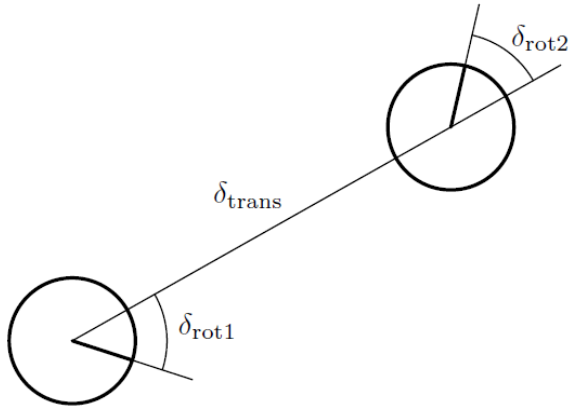
Our objective is to realize the particle filter algorithm by implementing the motion model sensor model and re-sampling logic.

In the context of localization, the particles are propagated according to the motion model. They are then weighted according to the likelihood of the observations in the sensor model. In a re-sampling step, new particles are drawn with a probability proportional to the likelihood of the observation.

1. Motion Model

We implement an 'Odometry-based Motion model' (Section 5.4, [1]) in **'motion_model.py'**. The "Predict" step in our algorithm uses the motion model to update the belief in the system state. We do this by considering a certain number of particles, where each particle represents a possible position for the robot.

For instance, consider we want the robot to both translate and rotate a certain way. We could translate and rotate each particle by this amount however, we would run into problems that arise due to the imperfections of the robot's control. Therefore we add noise to the particle's movements so as to have a reasonable chance of capturing the actual movement of the robot. It is implied that if the uncertainty in the system is not modeled properly, the particle filter's probability distribution of our belief in the robot's position will be incorrect.



The robot motion in the time interval (t − 1, t] is approximated by a rotation δ_{rot1}, followed by a translation δ_{trans} and a second rotation δ_{rot2}. The turns and translations are noisy.

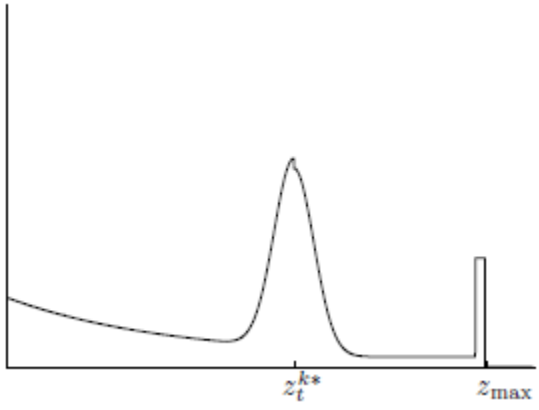
In our motion model, in the situation when there is no change in the control input (i.e. sensor measurements) from the logfile, no change is made via the motion model, consequently no noise is added to the particles as otherwise in a situation where the robot remains still for an extended period of time, it will reduce the belief of robot's position.

2. Sensor Model

The sensor model is implemented with the beam range finder model (Section 6.3, [1]) in **sensor_model.py**

For the ray casting, we implemented the vectorization along the rays for ray casting. The function iterates over the rays and calculates the distance of the obstacle for all the particles at the same time. The distances along each ray and each particle are stored in a matrix of dimension (no. of particles x no. of steps in the direction of ray). For this matrix, the probabilities at different steps are indexed from the provided occupancy map and if the probability is above a certain threshold it is considered to be occupied and assumed as a hit for the laser. The first index/ the lowest index from the matrix where the laser 'hit' an obstacle is used to infer the distance of the obstacle from the position of the robot. These distances are combined together to return the z* matrix for all the particles.

z* values for all the particles are then further used in the beam range finder model to find the probabilities in the pseudo-density probability distribution by comparing them with the measurements z from the sensor. This incorporates four types of measurement errors, small measurement noise, errors due to unexpected objects, errors due to failures to detect objects, and random unexplained noise which are modeled using different distributions such as gaussian, exponential and uniform. The weighted linear combination of these probabilities gives the 'pseudo-density' of a typical mixture distribution.

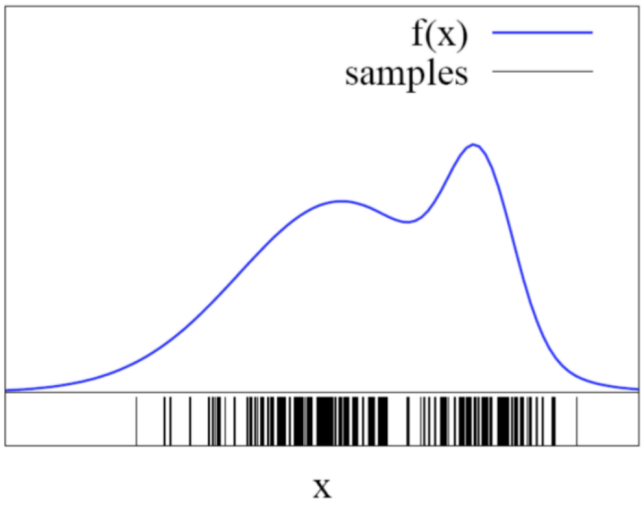


"Pseudo-density" of a typical mixture distribution $p(z^k_t \mid x_{t,m})$. [1]

The weighted linear combination is then normalized using log likelihoods and their sum is used as a factor against the number of beams to calculate weights for the particles. These weights are then further used in the resampling to find the best particles and discard the ones with a low likelihood.

3. Resampling - Low Variance

The Resampling method we followed was the Stochastic Universal Sampling method and it's implemented in the **'resampling.py'** file. We used this since it has low variance and it is faster. Once a single sample is randomly chosen, all other samples are deterministic. The goal of resampling is to increase the frequency of higher weighted samples in the sample set. By doing this, all the samples have equal weights after resampling however information about importance is not lost.



Samples have higher density where the probability is higher

```python
def low_variance_sampler(self, X_bar):

    X_bar_resampled = list()
    M = X_bar.shape[0]

    # * Normalization
    X_bar[:, 3] = X_bar[:, 3] / np.sum(X_bar[:, 3])

    # * Generate random uniform probability
    r = np.random.uniform(0, 1.0 / M)
    c = X_bar[0, 3]
    i = 0

    # * Resample particles based on weights
    for m in range(1, M + 1):
        u = r + (m - 1) / M
        while u > c:
            i = i + 1
            c = c + X_bar[i, 3]

        X_bar_resampled.append(X_bar[i])

    X_bar_resampled = np.array(X_bar_resampled)
    return X_bar_resampled
```

Low-Variance Resampling

Improving Efficiency

The following approaches were taken in order to improve the overall efficiency of our robot:

1. *Unoccupied space initialization of particles* : It is quite clearly observed that initializing the particles within the known free-space of the map significantly improves our accuracy of-course, but also reduces the time required to localize effectively, hence, this technique was deployed in our model.

2. *Vectorization of Motion Model* : This was achieved by passing all particles as a vector to `x_t1` (where the number of rows equal the number of particles selected)

```
# * Sample for state xt
x_t1 = np.zeros_like(x_t0)
x_t1[:, 0] = x + hd_trans * np.cos(theta + hd_rot_1)  # x
x_t1[:, 1] = y + hd_trans * np.sin(theta + hd_rot_1)  # y
x_t1[:, 2] = theta + hd_rot_1 + hd_rot_2  # theta

return x_t1
```

Sensor Model vectorization - beam_range_finder

3. *Vectorization in Ray-Casting* : Here we merely pass `x_t1` generated in the motion model which is a matrix containing the update pose for each and every particle. We then run through all particles using numpy functions such as `np.tile`. We also eliminate the need for "for loops", when out of bounds values are observed by employing `np.clip`.

4. *Increasing Step-size of laser measurements in Ray-Casting* : improves the computational ticostme by iterating over fewer steps in a single laser beam. It should be noted that this comes at the cost of accuracy, where the beam may skip a cell in the occupancy grid if the step-size is too large.
   Vectorization of Probability Distribution Function : we have attempted to vectorize the probability distribution function by passing in the vectorized `z_star` from our previous step, ray-casting. This in turn returns a probability distribution function with `z_star` of the dimensions *[number of particles] x [number of laser beams]*.

5. *Increasing the Sub-Sampling* : similar to increasing the step-size, improves the computational time by iterating over fewer laser rays. It should be noted that this comes at the cost of accuracy, which is not a major concern within the scope of this

assignment.

6. *Increasing the range of max_laser_range* : decreases the overall computational cost and speeds up the correction of the entire model, however, our laser range realistically has limitations, hence this point is merely for our knowledge.

```python
def __ray_cast(self, x_t1):
    m = x_t1.shape[0]
    z_star = np.zeros((m, self.n_beams))

    x = x_t1[:, 0]
    y = x_t1[:, 1]
    theta = x_t1[:, 2]

    theta_l = self.__wrap_to_pi(theta - (np.pi / 2))
    x_l = x + self.gap * np.cos(theta)
    y_l = y + self.gap * np.sin(theta)

    d_arr = np.arange(1, self.max_laser_range / self.step_size + 1) * self.step_size
    d_mat = np.tile(d_arr, (m, 1))

    x_l = np.tile(x_l, (d_arr.shape[0], 1)).T
    y_l = np.tile(y_l, (d_arr.shape[0], 1)).T

    angles = np.arange(np.pi / 2, -np.pi / 2, -np.pi / self.n_beams)
    # angles = np.arange(0, np.pi, np.pi / self.n_beams)
    for ray, angle in enumerate(angles):
        # print(f"\n____________________ Beam - {ray} @ {angle * 180 / np.pi} ____________________\n")
        # * Projecting steps on X and Y axes
        angles_x = np.cos(theta + angle)
        angles_y = np.sin(theta + angle)
        x_s = x_l + d_mat * angles_x[:, None]
        y_s = y_l + d_mat * angles_y[:, None]

        # * Clipping distances if beyond map
        x_s = np.clip(x_s, 0, self.map_x)
        y_s = np.clip(y_s, 0, self.map_y)

        # * Getting indexes from distances (0 - 799 that's why -1)
        x_s_idx = np.round(x_s / self.res - 1).astype(int)
        y_s_idx = np.round(y_s / self.res - 1).astype(int)

        # * Probabilites from Occupancy Map
        probs = self.occupancy_map[x_s_idx, y_s_idx]

        # * Keep probabilites > Threshold and get index of first 'True'
        mask = probs > self._min_probability
        indexes = np.where(mask.any(axis=1), mask.argmax(axis=1), -1)

        # * Update Z-star
        z_star[:, ray] = d_arr[indexes]

    return z_star
```

Sensor Model vectorization - Ray Casting

```python
def beam_range_finder_model(self, z_t1, x_t1, odometry_laser):
    """
    param[in] z_t1_arr : laser range readings [array of 180 values] at time t
    param[in] x_t1 : particle state belief [x, y, theta] at time t [world_frame] [M particles x 3]
    param[out] prob_zt1 : likelihood of a range scan zt1 at time t
    """
    z_samples = z_t1[:: self._subsampling].copy()  # (180 / sub-sample) x 1
    z_star = self.__ray_cast(x_t1)  # M x (180 / sub-sample)

    prob_zt1 = np.ones((x_t1.shape[0], 1))
    M = x_t1.shape[0]
    for p in range(M):
        prob_dist = self.__get_prob(z_star[p], z_samples)
        prob_zt1[p] = np.sum(np.where(prob_dist != 0.0, np.log(prob_dist), 0.0))
        prob_zt1[p] = self.n_beams / np.abs(prob_zt1[p])

    return prob_zt1, z_star
```

Sensor Model vectorization - beam_range_finder

```python
def __get_prob(self, z_star, z_samples):
    # z_samples = np.linspace(0, self._max_range, 1000)
    # z_star = np.ones_like(z_samples) * 500

    mask_hit = (z_samples <= self._max_range) & (z_samples >= 0)
    mask_short = (z_samples < z_star) & (z_samples >= 0)
    mask_max = z_samples == self._max_range
    mask_rand = (z_samples < self._max_range) & (z_samples >= 0)

    # * Hit
    p_hit = np.exp((-1 / 2) * ((z_samples - z_star) ** 2) / (self._sigma_hit**2))
    p_hit = p_hit / np.sqrt(2 * np.pi * self._sigma_hit**2)
    p_hit = p_hit * mask_hit
    # print(p_hit)

    # * Short
    eta = 1.0 / (1 - np.exp(-self._lambda_short * z_star))
    # eta = 1.0
    p_short = eta * self._lambda_short * np.exp(-self._lambda_short * z_samples)
    # print(eta, p_short)
    p_short = p_short * mask_short
    # print(p_short)

    # * Max
    p_max = np.ones_like(z_samples)
    p_max = p_max * mask_max
    # print(p_max)

    # * Rand
    p_rand = np.ones_like(z_samples) / self._max_range
    p_rand = p_rand * mask_rand
```

Sensor Model vectorization - Probability Distribution Function

Tuning

The Alpha parameters within the motion model describe the overall accuracy of the motion model itself and infers the rover dependency on the sensor model. It is similar to assigning a cost function to both the sensor and motion models.

Tuning these parameters tells the robot intrinsically, which model to trust more than the other. Furthermore, the alpha values contribute to the noise introduced directly within the system, and the larger the alpha values, the more profound the noise added to the system.

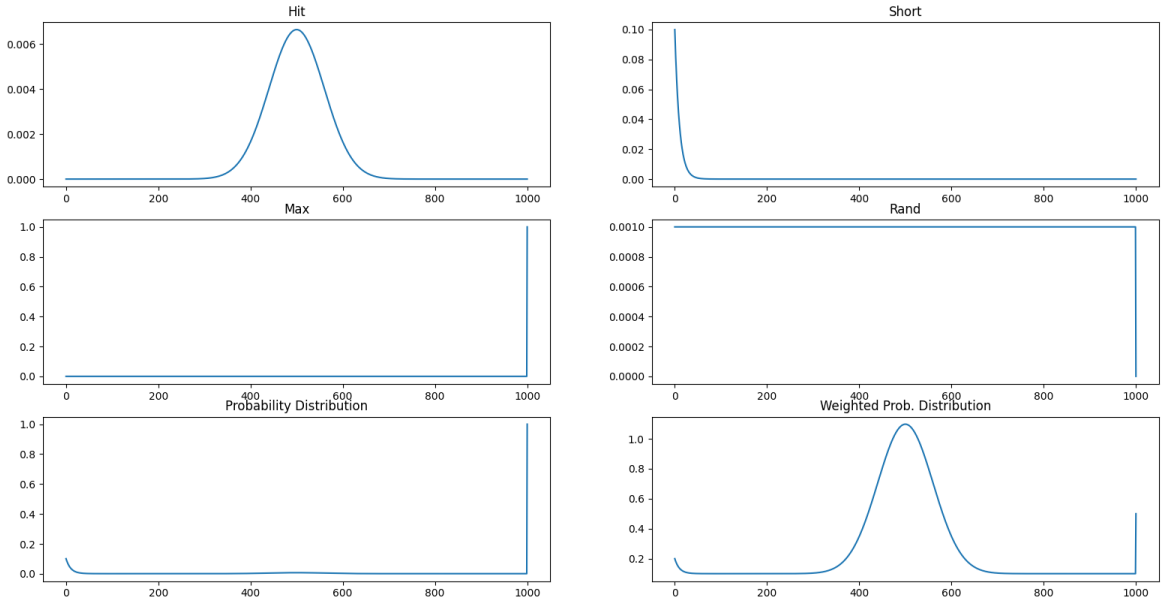
In our program, we observed that decreasing the values of Alpha's improved the convergence of our particle cluster as each iteration would introduce a small amount of noise. However, in order to simulate real conditions, we did play around with the Alpha values in an attempt to observe the impact on our system.

The values we have chosen for our alpha parameters are shown below. These values were selected as they optimized the localizability of the robot in different locations (from different robodata.log files)

α_1	0.001
α_2	0.001
α_3	0.001
α_4	0.001

For tuning the weights for the probabilities, p_{hit}, p_{short}, p_{max} and p_{rand}, for a fixed z*, we plotted the individual probability distributions and also the combined mixture distribution over a range of z measurements. We tuned the weights, z_{hit}, z_{short}, z_{max} and z_{rand} to make the mixture distribution similar to a typical 'pseudo-density' plot in the Beam Range Finder model.

z_{hit}	150
z_{short}	2
z_{max}	0.5
z_{rand}	100



Probability distribution plots for different z-measurements at z = 500 cm*

Robustness and Repeatability

The code we have developed is intended to be modular and computationally efficient. We are able to reproduce the localization over multiple trials with a good level of accuracy. Our vectorization is modular as well, and can be applied to any number of particles, beams or sub-samples.

However, we do face challenges in fine-tuning our model. Our current understanding of the tuning parameters have enabled us to achieve a certain level of accuracy. Further refinement could improve our results, primarily with respect to the rate of convergence and issues arising from particle divergence as well.

Future Scope

It should be noted however that, although we have a moderately robust model, there is still room for improvement. With respect to resampling, the problem of particle depletion and diversity of particles appears to have an adverse impact on particle filter based localization. This is an issue we have yet to tackle.

Moreover, to address the situations where the robot is 'kidnapped' or an edge case can be implemented on top of the existing code to make the localization model more robust for the robot, and make it usable for any general environment. Further, making the code more optimum and efficient to run for online implementation would be a key improvement, by implementing adaptive particles as well as adding 3D Matrix factorizations to perform the operations faster/ implement the existing codebase in C++.

Results and Links

The results of our particle filter algorithm can be seen below. In almost all the logs the particle managed to converge fairly quickly, albeit with some issues due to noise. The vectorization definitely helped make the algorithm much faster and allowed the entire log file to run within 25 minutes for the largest file with 500 particles:

- The visualization (video) for ***robotdata1.log***:

  https://drive.google.com/file/d/1dskyLirxFkb_99X5pf_Zs_-5133JimIt/view?usp=sharing

- The visualization (video) for ***robotdata4.log***:

  https://drive.google.com/file/d/1swU6M3NsR8Dm4nUuhQlyHBS58UnoKdA9/view?usp=sharing

- The visualization (video) for ***robotdata5.log***:

  https://drive.google.com/file/d/1WVTRZLDnuNqMheJ3yJJOXB9NFXePevR6/view?usp=sharing

The videos were made with OpenCV video writer at 30 frames per second (with an exception for the robotdata2.log file owing to the large number of entries, it was generated at 60 frames per second).

Other videos and visualizations can be found in the folder:

https://drive.google.com/drive/folders/1Tr-2dn_crvm4LEgMf91J_jyJj9Uiw_eZ?usp=sharing

References

[1] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. Probabilistic robotics. MIT press, 2005.