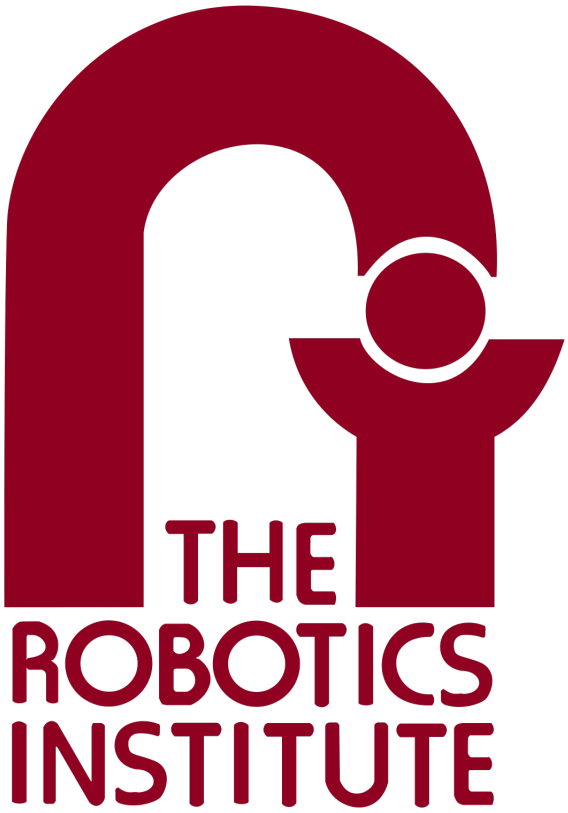
16-833A Robot Localisation and Mapping, Spring 2023
Homework 1: Robot Localization using Particle Filters

Sergi Widjaja, Sushanth Jayanth

Andrew ID: swidjaja, sushantj

April 7, 2023



Contents

1 Introduction to Particle Filter **1**

2 Approach **2**
2.1 Motion Model . . . 2
2.2 Understanding Log Files . . . 4
2.3 Understanding the Map Representation . . . 4
2.4 Sensor Model . . . 5
2.4.1 Ray Casting . . . 6
2.5 Importance Resampling . . . 7

3 Parameter Tuning **8**
3.1 Sensor Model . . . 8
3.2 Motion Model . . . 9

4 Performance and Improvements **10**
4.1 GPU Accelerated Ray Casting (Extra Credits) . . . 10
4.2 Pre-computed Ray-Casting . . . 11
4.2.1 Sampling Frequency . . . 12
4.3 Initialization of Particles . . . 13
4.3.1 Random Initialization . . . 13
4.3.2 Constrained Initialization . . . 14
4.4 Selective Sensor Model Integration . . . 14
4.5 Particle Quantity Decay . . . 15

5 Results **16**
5.0.1 Localisation Results Video Log . . . 16

1 Introduction to Particle Filter

Particle Filter works by predicting the new state of particles using a motion model and then correcting the prediction by sensing and resampling. From Bayes filter, we get the equation 1

$$p\left(x_{0:t} \mid z_{1:t}, u_{1:t}\right) = \eta p\left(z_t \mid x_t\right) p\left(x_t \mid x_{t-1}, u_t\right) p\left(x_{0:t-1} \mid z_{1:t-1}, u_{1:t-1}\right) \tag{1}$$

Where $p\left(x_{0:t} \mid z_{1:t}, u_{1:t}\right) = bel_{x_{0:t}}$

Although we don't know $bel_{x_{0:t}}$ and need to find it, we however do know the ratio:

$$\begin{aligned} w_t^{[m]} &= \frac{\text{target distribution}}{\text{proposal distribution}} \quad \begin{array}{l} \longleftarrow bel(x_{0:t}) \\ \longleftarrow p(x_t \mid x_{t-1}, u_t)\ bel(x_{0:t-1}) \end{array} \\ &= \frac{\eta\, p(z_t \mid x_t)\, p(x_t \mid x_{t-1}, u_t)\, p(x_{0:t-1} \mid z_{1:t-1}, u_{1:t-1})}{p(x_t \mid x_{t-1}, u_t)\, p(x_{0:t-1} \mid z_{0:t-1}, u_{0:t-1})} \\ &= \eta\, p(z_t \mid x_t) \end{aligned}$$

This ratio defined as w_t^m is the importance weight associated to each particle. This weight allows us to resample particles which are more closer to the proposal distribution and thereby are the correct particles which will help up localise the robot.

The pseudocode for particle filter is shown below:

Algorithm 1 Particle Filter for Robot Localization

1: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \phi$
2: **for** $m = 1$ to M **do**
3: sample $x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]})$ ▷ Motion model
4: $w_t^{[m]} = p(z_t \mid x_t^{[m]})$ ▷ Sensor model
5: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \left\langle x_t^{[m]}, w_t^{[m]} \right\rangle$
6: **end for**
7: **for** $m = 1$ to M **do**
8: draw i with probability $\propto w_t^{[i]}$ ▷ Resampling
9: add $x_t^{[i]}$ to $\mathcal{X}_t$
10: **end for**
11: **return** $\mathcal{X}_t$

2 Approach

2.1 Motion Model

We began by using class notes and 1, Probabalistic Robotics. First we modelled the motion of the robot as an odometry model from Chapter 5. of (1).

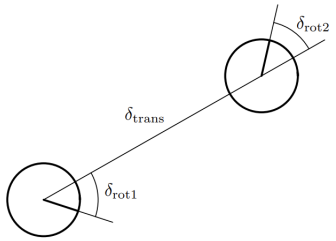


Figure 1: Odometry model: The robot motion in the time interval (t 1,t] is approximated by a rotation δ_{rot1}, followed by a translation δ_{trans} and a second rotation δ_{rot2}. The turns and translation are noisy.

The motion model was implemented with noise being added to rotation and translation updates

```
    def update(self, u_t0, u_t1, x_t0):
        """
        param[in] u_t0 : state odometry reading [x, y, theta] at time (t-1) [odometry_frame]
        param[in] u_t1 : state odometry reading [x, y, theta] at time t [odometry_frame]
        param[in] x_t0 : state belief [x, y, theta] at time (t-1) [world_frame]
        param[out] x_t1 : state belief [x, y, theta] at time t [world_frame]
        """

        y_bar_1 = u_t1[1]
        y_bar_0 = u_t0[1]
        x_bar_1 = u_t1[0]
        x_bar_0 = u_t0[0]
        yaw_bar_1 = u_t1[2]
        yaw_bar_0 = u_t0[2]

        x_0, y_0, yaw_0 = x_t0[0], x_t0[1], x_t0[2]

        delta_rot_1 = np.arctan2(y_bar_1 - y_bar_0, x_bar_1 - x_bar_0) - yaw_bar_0
        delta_trans = np.linalg.norm(np.array([x_bar_1 - x_bar_0, y_bar_1 - y_bar_0]), ord=2)
        delta_rot_2 = yaw_bar_1 - yaw_bar_0 - delta_rot_1

        delta_rot_1 = limit_angle(delta_rot_1)
        delta_rot_2 = limit_angle(delta_rot_2)

        # Adding noise to rotation and translation updates
        delta_rot_1_hat = delta_rot_1 - self.sample(self._alpha1 * delta_rot_1 ** 2 + self._alpha2
            * delta_trans ** 2)
        delta_trans_hat = delta_trans - self.sample(self._alpha3 * delta_trans ** 2 + self._alpha4
            * (delta_rot_1 ** 2 + delta_rot_2 ** 2))
        delta_rot_2_hat = delta_rot_2 - self.sample(self._alpha1 * delta_rot_2 ** 2 + self._alpha2
            * delta_trans ** 2)

        delta_rot_1_hat = limit_angle(delta_rot_1_hat)
        delta_rot_2_hat = limit_angle(delta_rot_2_hat)

        x_1 = x_0 + delta_trans_hat * np.cos(yaw_0 + delta_rot_1_hat)
        y_1 = y_0 + delta_trans_hat * np.sin(yaw_0 + delta_rot_1_hat)
        yaw_1 = limit_angle(yaw_0 + delta_rot_1_hat + delta_rot_2_hat)

        return np.array([x_1, y_1, yaw_1])
```

The above motion model was then used to verify approximate robot motion through dead-reckoning. This involved directly using the odometry updates from the log file to update the robot state without any additional noise input. This was plotted as shown in Fig. 2

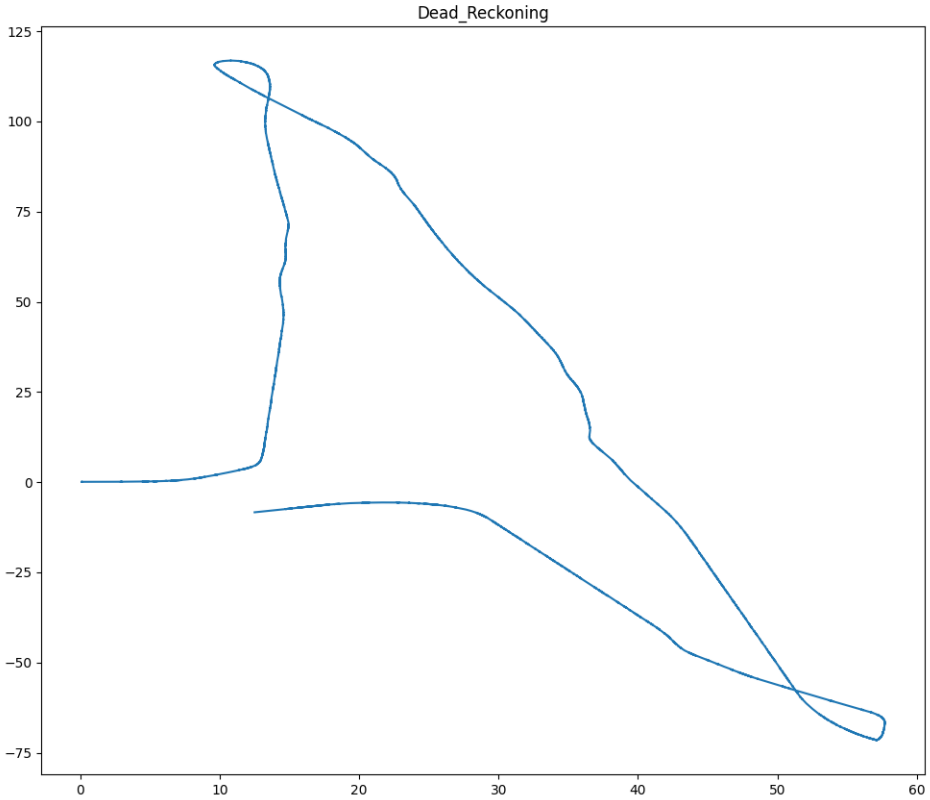


Figure 2: Robot tracking using dead reckoning

2.2 Understanding Log Files

While the codebase was sufficiently well detailed to test out the motion model, for further steps in the particle filter it was important to understand the nature of values in the log files - *robotdata1.log* as seen in line. 2 and line. 3

$$\mathrm{L} - 94.234001 - 139.953995 - 1.342158 - 88.567719... \tag{2}$$

$$\mathrm{O} - 94.234001 - 139.953995 - 1.3421580.025863 \tag{3}$$

The following points were noted:

- L = laser rangefinder reading (Entry type 1)
  - The first three numbers after L in line 2 are the estimated robot pose in odometry frame **x y theta**
  - The next three numbers are coordinates of laser rangefinder w.r.t robot's odometry frame **(xl, yl, thetal)**
  - The next **180 numbers are the rangefinder readings** (readings for 180 degrees captured by the rangefinder)
  - The last number of this entry is the **timestep**
- O = Odometry Reading **(x y theta)** (Entry type 2)

2.3 Understanding the Map Representation

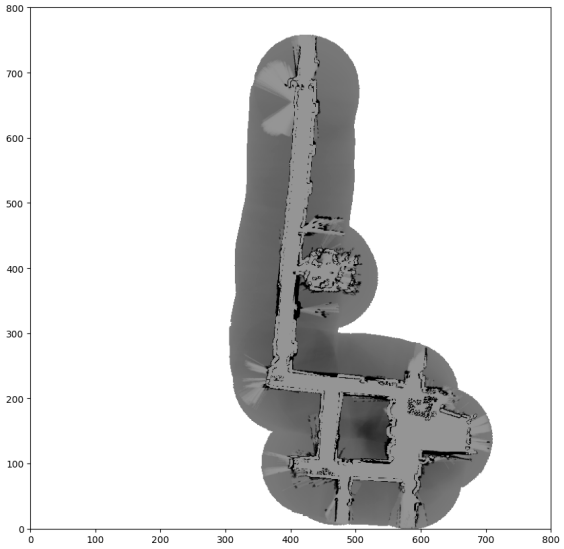


Figure 3: Given map of Wean Hall

The given map in *wean.dat* is an occupancy map which gives the probability for each cell being occupied (by an object such as a wall or human).

- Map has a resolution of 10cm i.e. each adjacent cell is 10cm apart in cartesian coordinates
- In cartesian coordinates the map has a size of 8000cm x 8000cm
- In occupancy map coordinates the map has a size of 800cells x 800 cells

Understanding the above conversions was important for the sensor model and ray-casting operations which are discussed next.a

2.4 Sensor Model

In this section, we will derive our Sensor Model $p\left(z_t \mid x_t, M\right)$ formulation.

Using a parameter set that we will tune based on prior knowledge of the sensor, we build a probability density function by combining the probability density functions that are dictated by different scenarios during data acquisition.

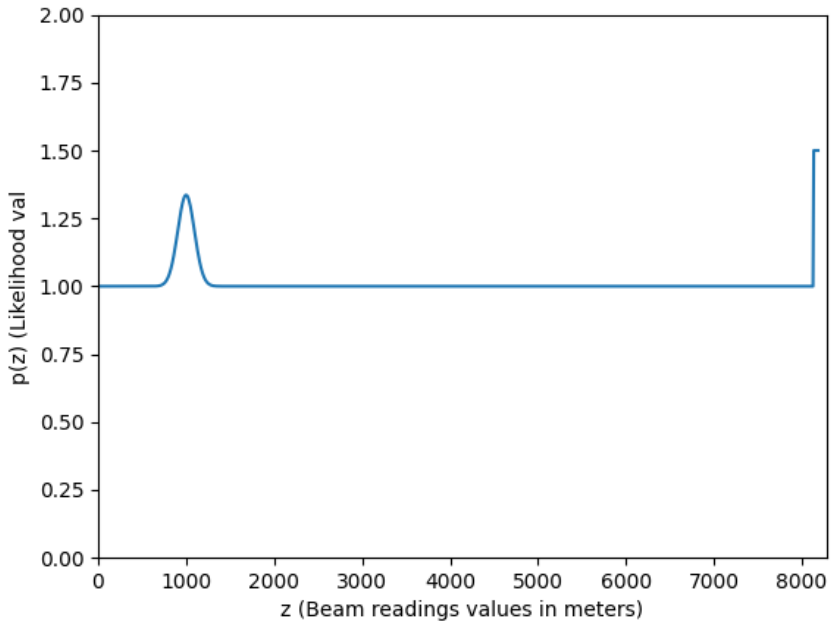


Figure 4: Probability density function

The following are phenomenons during data acquisition that dictates the shape of the probability distribution:

- Sensor readings matching the ground truth - This is represented by a Gaussian distribution with the location of the peak as the ground truth.
- Sensor beams hitting unexpected dynamic objects - This is represented by an exponentially decaying distribution.
- Missing obstacles altogether - This is represented by a Dirac delta function at the maximum possible distance reading.
- Random systematic noise - This is represented by a uniform distribution spanning across zero to maximum range.

Next, for every particle, we apply ray-casting on the map to find z_t^* which will serve as the locations of the peak of the probability density function found in Fig. 4 across beams. Ultimately, we arrive at NXB probability density functions, where N is the number of particles and B is the number of beams.

To obtain our importance weights for our importance sampling for every iteration. We then query the values of the probability density functions based on our current sensor readings coming from the log file. This mechanism is shown in Fig. 5.

Each particle will have 180 true rangefinder readings with an option to subsample. At this point, our implementation differs than the recommended implementation in the reference textbook, where instead of taking the *product* of individual probability values coming from each beams, we also tried the *sum* of the individual probability values. This is done to achieve a slower rate of convergence.

Finally, to achieve numerical stability related to floating point overflow and underflow. In all of our experiments where we are taking the product of the probability, we take the *natural logarithm* of the probability values to operate in a much stable log space, taking the sum and taking the exponent ultimately to get the product of the probability values.

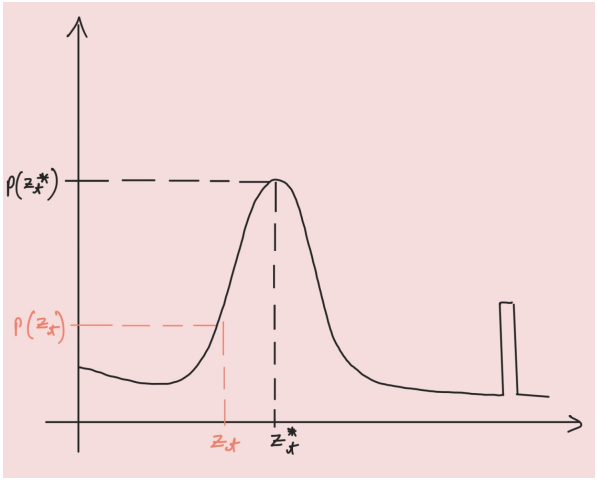


Figure 5: Obtaining importance weight of each particle

2.4.1 Ray Casting

To obtain z_t^*, a laser is simulated to be starting from the known (estimated) position of the particle in the occupancy map. Given the specification of the sensor placement relative to the robot. We displace the center of the ray casting operation to be 25 cm forward of the robot. Multiple rays are spread out in a 180 degree field-of-view based on the robot's heading. The distance value from each ray is noted and that serves value for the distribution $p(z_t^*)$.

This is a tedious process and was **vectorized** to allow for faster compute time. More importantly, when dealing with a large number of particles, vectorized implementation was necessary to give results in reasonable time. The vectorized implementation is shown below:

```
def ray_casting_vectorized_centimeters(self, X_t1):
    num_particles = len(X_t1)
    num_beams = 180 // self._discretization
    X_body, Y_body, Yaw = X_t1[:, 0], X_t1[:, 1], X_t1[:, 2]
    X_laser = X_body + 25 * np.cos(Yaw)
    Y_laser = Y_body + 25 * np.sin(Yaw)

    X_laser = np.repeat(X_laser.reshape(-1, 1), num_beams, axis=1) # m, 180
    Y_laser = np.repeat(Y_laser.reshape(-1, 1), num_beams, axis=1) # m, 180

    angles = np.array(list(range(-90, 90, self._discretization)))
    assert len(angles) == num_beams

    beam_hit_length = np.ones_like(X_laser) * self._max_range
    beam_step = self._occupancy_map_resolution_centimeters_per_pixel // 2.1
    np.arange(0, self._max_range, beam_step)
    for ray_length in np.arange(0, self._max_range, beam_step):
        # The beams start from the RHS of the robot, the yaw angle is measured from the heading
            of the robot.
        # Hence the minus 90 degrees.
        X_beams = X_laser + np.cos(np.radians(angles) + np.repeat(Yaw.reshape(-1, 1), 180 //
            self._discretization, axis=1)) * ray_length
        Y_beams = Y_laser + np.sin(np.radians(angles) + np.repeat(Yaw.reshape(-1, 1), 180 //
            self._discretization, axis=1)) * ray_length

        X_beams_pixels = np.round(X_beams / 10).astype(int)
        Y_beams_pixels = np.round(Y_beams / 10).astype(int)

        X_beams_pixels = np.clip(X_beams_pixels, 0, 799)
        Y_beams_pixels = np.clip(Y_beams_pixels, 0, 799)

        occupancy_vals = self._occupancy_map[Y_beams_pixels, X_beams_pixels]
        # occupancy_vals = self._occupancy_map[X, Y]
```

```
            beam_hit_length = np.minimum(
                beam_hit_length,
                np.where(occupancy_vals > self._occupancy_map_confidence_threshold, ray_length,
                    self._max_range)
            )

        Z_star_t_arr = beam_hit_length
        return Z_star_t_arr
```

2.5 Importance Resampling

To resample particle based on the importance weight a low-variance resampler was used as specified in **Table 4.4** of (1). However, since the low-variance sampler relies on probabilites (which are normalized between 0 and 1), an additional step of normalization was added as shown below:

```
    def low_variance_sampler(self, X_bar):
        """
        param[in] X_bar : [num_particles x 4] sized array containing [x, y, theta, wt] values for
            all particles
        param[out] X_bar_resampled : [num_particles x 4] sized array containing [x, y, theta, wt]
            values for resampled set of particles
        """

        M = len(X_bar)
        weights = X_bar[:, 3]
        # normalize the weights
        weights = weights / weights.sum()
        r = np.random.uniform(0, 1 / len(X_bar))
        c = weights[0]
        i = 0
        X_bar_resampled = []

        for m in range(1, M + 1):
            u = r + (m - 1) * (1 / M)
            while u > c:
                i = i + 1
                if i > len(weights):
                    break
                c = c + weights[i - 1]
            X_bar_resampled.append(X_bar[i - 1])
        X_bar_resampled = np.array(X_bar_resampled)

        return X_bar_resampled
```

3 Parameter Tuning

3.1 Sensor Model

In our initial experiments, we noticed that the particle set converges too quickly to an incorrect solution. We hypothesize that it is necessary to explore all possible solution spaces during the beginning of the integration steps, as there are similar features that can be encountered in the map. We then realize that this fast convergence is due to the importance weights across particles taking up values that are drastically different than each other (high variability). This way, during the resampling step, higher-weighted particles will dominate the particle set early in the integration steps.

In short, we want to ensure that after taking the product of probability across beams, the aggregate probability doesn't result in drastically low or high values. The key to that is by ensuring that $Z_{rand} \times p_{rand}$ results in a value of 1.

Given the following equations:

$$p_{rand}(z_t^k) = \begin{cases} \frac{1}{z_{max}} & \text{If } 0 < z_t^k < z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

$$\bar{p_{rand}}(z_t^k) = Z_{rand} \times p_{rand}(z_t^k) \tag{5}$$

We want to ensure that within the sensor's operating range, the values produced by the random measurement component of the sensor model arrive to a value of *1*. Therefore, the value of Z_{rand} is derived to be:

$$Z_{rand} = z_{max} \tag{6}$$

To reduce the speed of convergence, we then also tune the values of Z_{hit} to not also produce drastically high values if a correct correspondence is found. Therefore, we choose to experiment with a relatively low Z_{hit} values $Z_{hit} \in [0.05, 1]$. The same is done for the rest of the probability function components $Z_{short} \in [0.0005, 0.5]; Z_{max} \in [0.0005, 1]$, where we take up relatively low Z values for the particle filter to explore the possible solution space in the map.

Furthermore, to ensure a balanced convergence pace and appropriate exploration of the solution space, the width σ_{hit} of the peak $p(z_t^{k*})$ of the probability distribution function is chosen to be a relatively high value $\sigma_{hit} \in [80, 100]$.

After careful tuning using the belief product formulation, we came up with the idea of summing the belief across beams. By taking the sum instead of the product of the distributions, we will come up with a lower variability of importance weight values across the particles. This will result in a slower convergence rate across model integration steps.

Therefore, we need to devise a parameter set that balances this phenomenon. We achieve this by ensuring that the parameter set produces drastically different likelihood values between beams. For that reason, we arrived at the following value ranges for our sensor model parameters if we were to sum our belief across all beams.

- self._z_hit = 100
- self._z_short = 0.0001
- self._z_max = 1
- self._z_rand = 1
- self._sigma_hit = 10
- self._lambda_short = 0.12

3.2 Motion Model

Visualizing the dead-reckoning (Fig. 2) gave some insights into the nature of noise that can be expected in the motion model.

- All hallways are almost perpendicular and therefore the robot motion should also be mostly parallel to the x-axis and y-axis.
- Looking at dead-reckoning, the robot does not seem to rotate in multiples of 90 degrees, and therefore it was inferred that a white noise with larger variance must be added to every state update obtained in odometry frame.
- The noise in translation is difficult to gauge from looking at dead-reckoning. However, odometry readings are generally accurate in translation and we used that as an assumption.

Based on the above cues, we tuned our motion model to be the following:

- self._alpha1 = 0.02 # associated with heading angle
- self._alpha2 = 0.02
- self._alpha3 = 0.0002 # associated with the wheel odometry
- self._alpha4 = 0.0002

4 Performance and Improvements

The vectorized version of the code running on *robotdata1.log* took X time to compute. The main bottleneck was found to be ray-casting (ADD SOME PERCENTAGE ON HOW LONG IT TAKES). However, for quick iteration cycles and parameter tuning, a faster method of ray-casting using GPU was implemented.

4.1 GPU Accelerated Ray Casting (Extra Credits)

```
def forward(self, X_t1):
    num_particles = len(X_t1)
    num_beams = 180 // self._discretization
    X_body, Y_body, Yaw = X_t1[:, 0], X_t1[:, 1], X_t1[:, 2]
    X_laser = X_body + 25 * torch.cos(Yaw)
    Y_laser = Y_body + 25 * torch.sin(Yaw)
    X_laser_pixels = X_laser / 10 # m, 1
    Y_laser_pixels = Y_laser / 10 # m, 1

    X_laser = torch.repeat_interleave(X_laser_pixels.reshape(-1, 1), num_beams, dim=1) # m, 180
    Y_laser = torch.repeat_interleave(Y_laser_pixels.reshape(-1, 1), num_beams, dim=1) # m, 180

    angles = torch.Tensor(list(range(-90, 90, self._discretization))).cuda()
    assert len(angles) == num_beams

    max_range_pixels = self._max_range / self._occupancy_map_resolution_centimeters_per_pixel
    beam_hit_length_pixels = torch.ones_like(X_laser).cuda() * max_range_pixels
    for ray_length in range(0, int(round(max_range_pixels) + 1)):
        # The beams start from the RHS of the robot, the yaw angle is measured from the heading
            of the robot.
        # Hence the minus 90 degrees.
        X_beams_pixels = X_laser + \
                         torch.cos(torch.deg2rad(angles) +
                             torch.repeat_interleave(Yaw.reshape(-1, 1), 180 //
                             self._discretization, dim=1)) * ray_length

        Y_beams_pixels = Y_laser + \
                         torch.sin(torch.deg2rad(angles) +
                             torch.repeat_interleave(Yaw.reshape(-1, 1), 180 //
                             self._discretization, dim=1)) * ray_length

        X_beams_pixels = torch.round(X_beams_pixels).int()
        Y_beams_pixels = torch.round(Y_beams_pixels).int()

        X_beams_pixels = torch.clip(X_beams_pixels, 0, 799)
        Y_beams_pixels = torch.clip(Y_beams_pixels, 0, 799)

        occupancy_vals = torch.Tensor(self._occupancy_map[Y_beams_pixels.cpu(),
            X_beams_pixels.cpu()]).cuda()

        beam_hit_length_pixels = torch.minimum(beam_hit_length_pixels,
            torch.where(occupancy_vals > self._occupancy_map_confidence_threshold, ray_length,
                max_range_pixels))

    Z_star_t_arr = beam_hit_length_pixels
    return Z_star_t_arr
```

The whole computation is wrapped inside a PyTorch module that handles most of the calculations in GPU. We found that applying this method increases the ray casting speed by approximately a factor of 3. As we will see further, however, that a method can be devised to avoid performing the ray casting operation online.

4.2 Pre-computed Ray-Casting

Even with GPU acceleration, we could not bring down the ray-casting to be fast enough to allow parameter tuning. Therefore a look-up table was made which pre-computes the ray-casting for every possible robot position on the map. Additionally, this look-up table was created at 2 times the resolution of the occupancy map. This is a sufficient condition to avoid aliasing of the raycasting computation.

```
MAP_SAMPLING_MULTIPLIER = 2
ORIGINAL_OCCUPANCY_MAP_RESOLUTION = 10
MAP_TO_CARTESIAN_MULTIPLIER = ORIGINAL_OCCUPANCY_MAP_RESOLUTION/MAP_SAMPLING_MULTIPLIER

def get_ray_cast_per_square(start_pos_x, grid_discretize, map_shape_axis_1):
print(f"computing grid cell {start_pos_x}")
# raycast_lookup defines the size of this slice of the graph on which we will do raycasting
# NOTE: RAYCAST_LOOKUP = 5x1600x360
# NOTE: 1600 = Map resolution in cartesian coordinates(8000) * 0.2
raycast_lookup = np.zeros((grid_discretize, map_shape_axis_1, 360), dtype=np.float16)

for xpos in range(raycast_lookup.shape[0]):
    for ypos in range(raycast_lookup.shape[1]):
        # use ray casting to find the range_finders simulated readings at this robot pose

        # to make use of vectorized ray casting, init some dummy particles
        # THE MAP_TO_CARTESIAN_MULTIPLIER = 5 at the moment
        dummy_particles = np.ones((1,3))
        dummy_particles[:,0] = (xpos + start_pos_x) * MAP_TO_CARTESIAN_MULTIPLIER
        dummy_particles[:,1] = (ypos) * MAP_TO_CARTESIAN_MULTIPLIER
        dummy_particles[:,2] = 0

        raycast_vals = sensor_model.ray_casting_vectorized_centimeters(dummy_particles)
        raycast_lookup[xpos][ypos] = raycast_vals[0,:]

name = os.path.join('./raycast_lookup', str(start_pos_x))
np.savez(name, arr=raycast_lookup)

if __name__ == '__main__':
    grid_discretize = 5

    # multiprocessing.set_start_method('forkserver', force=True)
    # TODO: Check which number of processes is fastest, not just max
    pool = multiprocessing.Pool(processes=12)

    # define resolution of the lookup table (keep min resolution = 2x that of occupancy map in
        cartesian)
    lookup_resolution = [occupancy_map.shape[0]*MAP_SAMPLING_MULTIPLIER,
        occupancy_map.shape[1]*MAP_SAMPLING_MULTIPLIER]

    print(f"running raycasting on map of resolution \
           {lookup_resolution} = 1/{MAP_TO_CARTESIAN_MULTIPLIER} * MAP_RES in
               cartesian(8000x8000)")

    items = [(xpos, grid_discretize, lookup_resolution[1]) for xpos in range(0,
        lookup_resolution[0], grid_discretize)]
    pool.starmap(get_ray_cast_per_square, items)
    print("done")
```

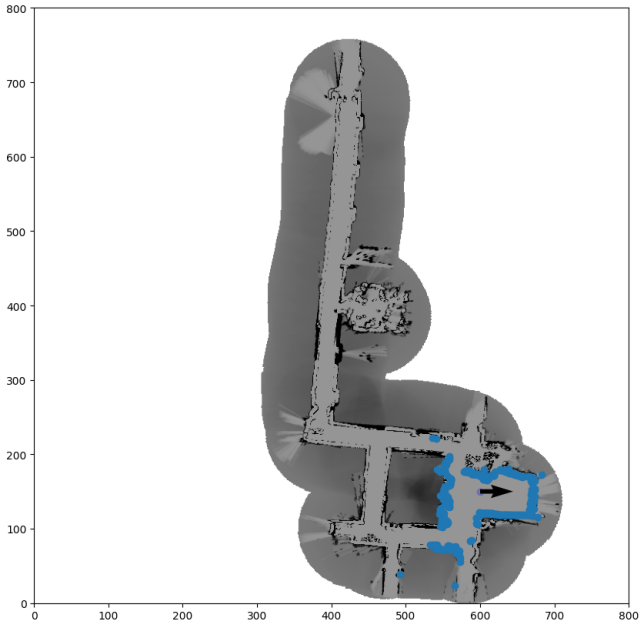


Figure 6: Results of ray-casting using look-up table

4.2.1 Sampling Frequency

From Shannon Nyquist theorem, it can be inferred that higher sampling frequency leads to better reproduction of the original object. In our case, ray-casting involves sampling the map in the direction of a ray until we hit an object. This is done in the following manner

- The original map is in cartesian coordinates (8000cm x 8000cm)
- Tracing a raw at each occupany map cell results in tracking the ray at 10cm resolution on the cartesian map (since the occupancy map has a resolution of 10cm)
- Instead, we could sample the ray every 5cm which equates to twice the resolution of the occupancy map

The benefits of a higher sampling rate is explained in Fig. 7. It shows that a wall can be estimated at a more accurate distance from the robot if the frequency of sampling is increased.

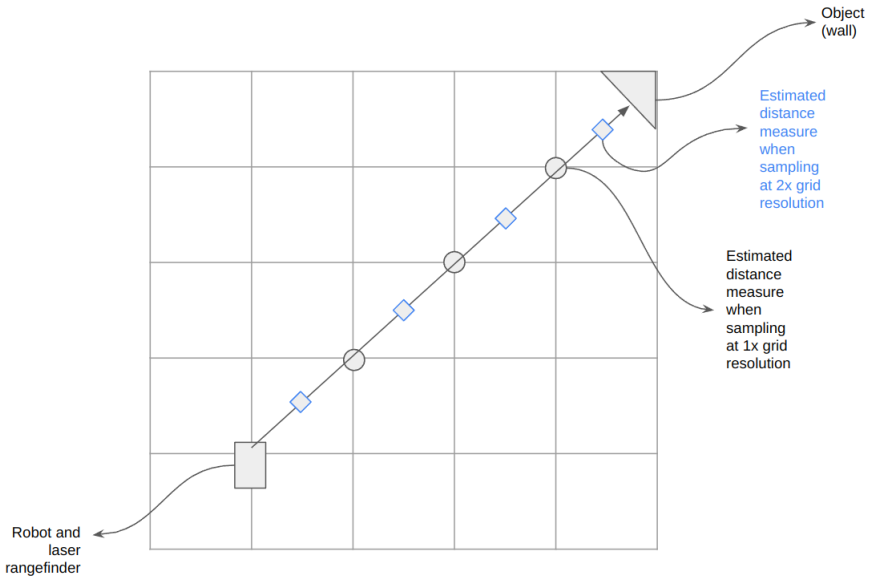


Figure 7: Comparison of sampling rate

4.3 Initialization of Particles

The occupancy map gives us probabilities of cells being occupied, or not having been mapped. We found that there are two ways in initializing the particles in the map:

4.3.1 Random Initialization

This was provided in the starter code and randomly initialized particles in the occupancy map by uniform sampling within the map. The code is shown below:

```
def init_particles_random(num_particles, occupancy_map):

    # initialize [x, y, theta] positions in world_frame for all particles
    y0_vals = np.random.uniform(0, 7000, (num_particles, 1))
    x0_vals = np.random.uniform(3000, 7000, (num_particles, 1))
    theta0_vals = np.random.uniform(-3.14, 3.14, (num_particles, 1))

    # initialize weights for all particles
    w0_vals = np.ones((num_particles, 1), dtype=np.float64)
    w0_vals = w0_vals / num_particles

    X_bar_init = np.hstack((x0_vals, y0_vals, theta0_vals, w0_vals))

    return X_bar_init
```

However, since it's unlikely that the robot (represented by a particle) can possibly be in unmapped regions of the map (where occupancy_value = -1), we tried to constrain this initialization.

4.3.2 Constrained Initialization

We determined that initializing particles in regions where $occupancy\_value = -1$ would not be beneficial. Additionally, in regions where $occupancy\_value > 0.15$, it was assumed that those regions would be occupied by objects and therefore cannot contain the robot. Hence, the following initialization method was used:

```
def init_particles(num_particles, occupancy_map):
    empty = np.argwhere(np.logical_and(occupancy_map < 0.15, occupancy_map != -1))
    indices = np.random.choice(list(range(len(empty))), num_particles)

    # resolution.
    xs = empty[indices, 1].reshape(-1, 1) * 10
    ys = empty[indices, 0].reshape(-1, 1) * 10

    angles = np.ones((num_particles, 1)) * np.radians(np.random.uniform(170, 190) - 5)
    which_axis = np.random.randint(0, 4, num_particles)

    angles[which_axis == 0] = np.radians(np.random.uniform(-5, 5) - 5)
    angles[which_axis == 1] = np.radians(np.random.uniform(85, 95) - 5)
    angles[which_axis == 2] = np.radians(np.random.uniform(175, 185) - 5)
    angles[which_axis == 3] = np.radians(np.random.uniform(265, 275) - 5)

    theta0_vals = np.random.uniform(-3.14, 3.14, (num_particles, 1))
    w0_vals = np.ones((num_particles, 1), dtype=np.float64)
    w0_vals = w0_vals / num_particles

    # change this to theta vals
    X_bar_init = np.hstack((xs, ys, angles, w0_vals))
    return X_bar_init
```

4.4 Selective Sensor Model Integration

When the odometry readings imply that the robot is not moving, the sensor readings must have been capturing the same scene, which can be redundant for the importance weight calculation and the resampling step. Therefore, we tried adding a boolean flag to ensure that the resampling step and the sensor model integration step only happens when the robot is moving (obtained from the odometry).

```
# After motion model update.
print("Moving by ", np.linalg.norm(u_t1[:2] - u_t0[:2]))
if np.linalg.norm(u_t1[:2] - u_t0[:2]) > threshold:
    print("Now moving.")
    moving = True
else:
    print("Now stopping")
    moving = False

# Sensor model update.
if meas_type == "L" and moving:
    z_t = ranges
    if moving:
        W_t = sensor_model.beam_range_finder_model_vectorized(z_t, X_bar_new[:, :3])
        X_bar_new[:, 3] = W_t

# Resampling.
if moving:
    X_bar = resampler.low_variance_sampler(X_bar)
```

In the test cases, however, we do not see noticeable improvements when this is applied. This might be because the robot only stays idle for very short times, reducing the overall effect of the change. In practice, however, we argue that this change is necessary for a robust Particle Filter convergence.

4.5 Particle Quantity Decay

It was found that a large number of particles were necessary in the beginning to explore the search space and not get caught in local minima. However, once the particles have converged into a set of clusters, there is no longer a need to maintain a large number of particles. Thus, the number of particles could be reduced to save computation.

For simplicity, we applied an exponentially decaying function of the number of particles. We started off by initializing large number of particles (e.g $N_0 \in [5000, 1000]$). For every time instance where we we have a new measurement, we sort the particles based on its importance weights and take the top N_i element, where N_i is an exponentially decaying value across across model integration steps i.

Given N_i to be the number of particles at a particular time step, and λ to be the decay factor. We formulate the number of particles at every time step to be

$$N_t = N_0 \times \lambda^i \tag{7}$$

$$\lambda \in [0, 1] \tag{8}$$

```
if meas_type == 'L' and moving:
    top_k = int(args.decrease_factor * len(X_bar))
    num_particles = top_k

    if num_particles < args.num_particles_min:
        num_particles = args.num_particles_min

    X_bar = np.array(sorted(X_bar, key=lambda x: x[3])[::-1])
    X_bar = X_bar[:num_particles, :]
```

5 Results

Repeatability was tested in the following manner:

- Testing with random seed
- Testing with multiple log files

Only initial convergence time varied across tests and was captured in the below table:

Trial Number	Convergence Time (s)
Trial 1	6
Trial 2	6.2

Although the convergence times were consistent across trials, the particles failed to converge at the right location with random seed consistently. Only an approximately 20% successful convergence rate was noted as seen in Fig. **??**. This is due to the map features being very similar at certain locations, and the robot hypothesis forms a multimodal distribution across different clusters.

5.0.1 Localisation Results Video Log

- [Robot Data log 1]
- [Robot Data log 2]

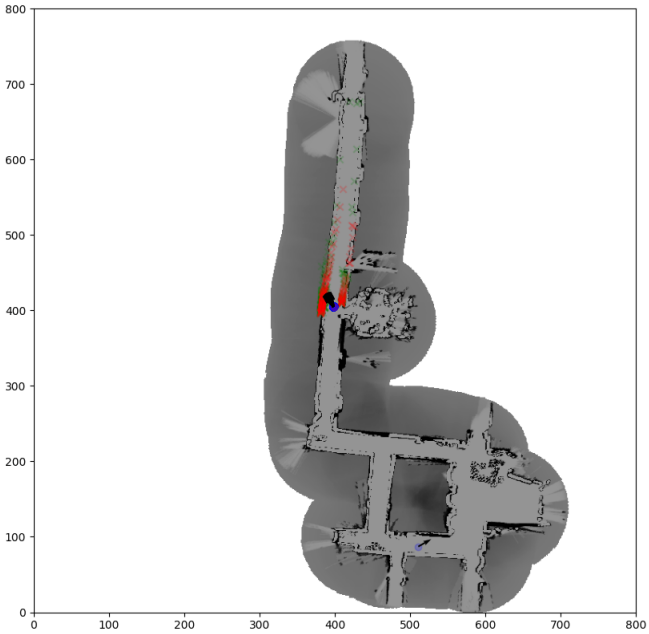


Figure 8: Correct Convergence

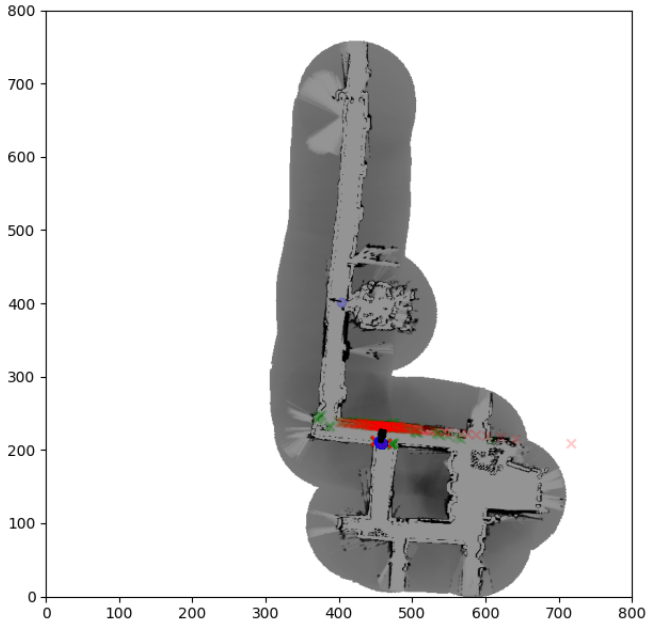


Figure 9: Incorrect Convergence

References

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