[16-833] Homework 1 : Written Report

Bharath Somayajula, Christopher Klammer

February 27, 2023

Contents

1 Introduction

For this homework, we implemented particle filter for localization of robot when the map is known. We used Python language for implementation. In the sections below we describe the details of motion model, sensor model, ray casting, resampling, parameter tuning and suggestions for future work.

2 Motion Model

2.1 Description

The motion model acts as the prediction step in the particle filter. We follow the odometry motion model mentioned in [1] as shown below:

Algorithm sample_motion_model_odometry(u_t, x_{t-1})**:**

$\delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$

$\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$

$\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

$\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta^2_{\text{rot1}} + \alpha_2 \delta^2_{\text{trans}})$

$\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3 \delta^2_{\text{trans}} + \alpha_4\ \delta^2_{\text{rot1}} + \alpha_4\ \delta^2_{\text{rot2}})$

$\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta^2_{\text{rot2}} + \alpha_2 \delta^2_{\text{trans}})$

$x' = x + \hat{\delta}_{\text{trans}}\ \cos(\theta + \hat{\delta}_{\text{rot1}})$

$y' = y + \hat{\delta}_{\text{trans}}\ \sin(\theta + \hat{\delta}_{\text{rot1}})$

$\theta' = \theta + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$

return $x_t = (x', y', \theta')^T$

Figure 1: Psuedocode for Odometry Motion Model [1]

This algorithm assumes that two odometry measurements are given along with the previous state. The algorithm estimates the change in rotation and translation for the robot between time steps. This model also samples noise from a guassian distribution in order to add some perturbation. This perturbation allows uncertainity to be modelled and, in practice, allows particles to explore different paths. This approach, however, does require for the noise parameters to be tuned effectively in order to receive accurate localization.

2.2 Implementation

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

        # See if there is consequential motion
        # if np.sum(np.abs(u_t1 - u_t0)) <= 1e-10:
        #     return x_t0

        # Output vector for the pose at time t
        x_t1 = np.zeros_like(x_t0)

        # Find the rotation from time (t-1) to the centroid at time t from odometry measurements
        # Angle between the translation vector and the first angle measurement
        delta_rot_1 = np.arctan2(u_t1[1] - u_t0[1], u_t1[0] - u_t0[0]) - u_t0[2]
        delta_trans = sqrt((u_t1[0] - u_t0[0]) ** 2 + (u_t1[1] - u_t0[1]) ** 2)
        # This is just the angle between the translation vector and the second angle measurement
        delta_rot_2 = u_t1[2] - u_t0[2] - delta_rot_1

        # Remove the independent noise
        delta_rot_1_var = self._alpha1 * delta_rot_1**2 + self._alpha2 * delta_trans**2
        delta_trans_var = self._alpha3 * delta_trans**2 + self._alpha4 * (delta_rot_1**2 + delta_rot_2**2)
        delta_rot_2_var = self._alpha1 * delta_rot_2**2 + self._alpha2 * delta_trans ** 2

        delta_rot_1 -= np.random.normal(0.0, np.sqrt(delta_rot_1_var), size = x_t0.shape[0])
        delta_trans -= np.random.normal(0.0, np.sqrt(delta_trans_var), size = x_t0.shape[0])
        delta_rot_2 -= np.random.normal(0.0, np.sqrt(delta_rot_2_var), size = x_t0.shape[0])

        # Estimate the pose of the robot at time t
        x_t1[:, 0] = x_t0[:, 0] + delta_trans * np.cos(x_t0[:, 2] + delta_rot_1)
        x_t1[:, 1] = x_t0[:, 1] + delta_trans * np.sin(x_t0[:, 2] + delta_rot_1)
        x_t1[:, 2] = x_t0[:, 2] + delta_rot_1 + delta_rot_2

        return x_t1
```

Figure 2: Implementation of the Odometry Motion Model

2.3 Alterations

One improvement made on this algorithm was adding a check on whether there was consequential motion before predicting the state.

$$x_{t1} = \begin{cases} motion\_model\_odometry(u_{t-1}, u_t, x_{t-1}) & \|u_{t1} - u_{t0}\| > \tau \\ x_{t0} & else \end{cases}$$

That is, if there is consequential motion, we will run the prediciton step with the odometry motion model. Otherwise, we will not update the state estimate for the particle. We found this change to reduce the particle jitter and enforce a much more cohesive and smooth trajectory.

3 Sensor Model

3.1 Ray Casting

3.1.1 Description

RayCasting is needed for estimation of true ranges at various angles.

To make implementation easier, we created a *RayCasting* module that performs ray casting. The *get_true_ranges_vec* function is invoked to compute ranges at various angles for all particles. *sensor_location* function adjusts the state of robots by 25 cm to take into account the distance between robot and the sensor.

3.1.2 Implementation

Ray Casting is the most computationally expensive part of the code. We implemented the following changes to naive implementation of ray casting to improve the performance.

1. To avoid redundant computations, we perform ray casting only once in constructor function using the *relative_ray_casting* function. This function casts rays relative to the robot in it's canonical orientation. At all future time steps, the points along these rays are simply rotated and translated to match the position and orientation of robot! This led to massive gains in performance. The implementation of *relative_ray_casting* function is shown below.

    -

```
def relative_ray_casting(self):
    """Perform ray casting relative to robot's position

    Returns:
        np.ndarray: array containing x,y values of each point along each ray
    """
    #angles at which rays are cast
    # Take the angles going counter-clockwise
    angles = np.arange(-90, 90, self._subsampling)

    #distances at which z is computed along each ray
    diag_length = math.ceil(math.sqrt(self.h**2 + self.w**2))
    dists = np.arange(0, diag_length, 1)

    #number of angles and points along each ray
    num_angles = angles.size
    num_points = dists.size

    #create array to store x and y for each point in ray
    rays = np.zeros((num_angles, num_points, 2))

    #perform ray casting relative to robot's location
    for i, a in enumerate(angles):
        for j, d in enumerate(dists):
            x = d*np.cos((np.pi/180)*a)
            y = d*np.sin((np.pi/180)*a)
            rays[i, j] = [x, y]

    #duplicate rays for every particle
    rays = np.tile(rays, (self.num_particles, 1, 1, 1))

    #scale distances
    rays = self.resolution*rays

    return rays
```

Figure 3: Pre-computation of points along rays

2. Once the rays have been oriented based on location and orientation of a particle, the logic used to measure the range along each ray at which it encounters an obstacle or goes out of the map has been vectorized. Compared to the naive implementation of this logic using nested for-loops, the fully vectorized implementation is approximately **9.5x faster**. The vectorized logic is shown below

3. Vectorize the code to enable computation of ranges for all particles simultaneously instead of performing ray casting in a loop seperately for each particle

-

```python
def get_true_ranges_vec(self, x_t):
        """Find true depths at all angles for a given robot state

        Args:
            x_t (list): states of robot represeented by particles
        """

        #adjust robot's state into laser's state
        x_t = self.sensor_location(x_t)

        #adjust rays based on robot's state
        rays = self.transform_rays(x_t)

        #find obstacle along each ray
        num_angles = rays.shape[1]

        #array to store ranges
        z_true = np.zeros((self.num_particles, num_angles))

        #convert cm to px
        x_int = np.round(rays[:,:,:,0]/self.resolution).astype(np.int32)
        y_int = np.round(rays[:,:,:,1]/self.resolution).astype(np.int32)

        #filter coordinates outside map
        m_lx = x_int < 0
        m_ly = y_int < 0
        m_hx = x_int >= self.w
        m_hy = y_int >= self.h

        # Filter out pixels on or beyond the boundary
        m_filter = np.logical_or.reduce((m_lx, m_ly, m_hx, m_hy))

        # Clip x and y coordinates to their max and min values
        x_int[m_filter] = 0
        y_int[m_filter] = 0

        #find coordinates that hit the obstacle
        obstacles = np.bitwise_or(self.map[y_int, x_int] == -1, \
                                  self.map[y_int, x_int] >= self._min_probability)

        #take intersection of filter and obstacle masks
        # This is all the occupied areas of the map
        occupied = np.bitwise_or(m_filter, obstacles).astype("int")

        # Travel along each ray for a given particle and angle
        # Once we hit a point that is intraversible, our cumsum will be 1
        m_overall_cumsum = np.cumsum(occupied, axis=2)

        # Look at the particle, angle, and distance indexes to look for the boundaries
        particles, angs, dists = np.where(m_overall_cumsum == 1)

        #populate z_true based on x and y
        z_true[particles, angs] = dists

        #scale range
        z_true *= self.resolution

        return z_true, x_int, y_int
```

Figure 4: Vectorized logic to compute ranges

3.2 Results

The figure below shows rays cast from a sample robot state. The time taken to measure ground truth ranges at 36 angles for 1000 particles using our optimized implementation is just around **1.05 seconds** on Apple Macbook M1 Pro system.

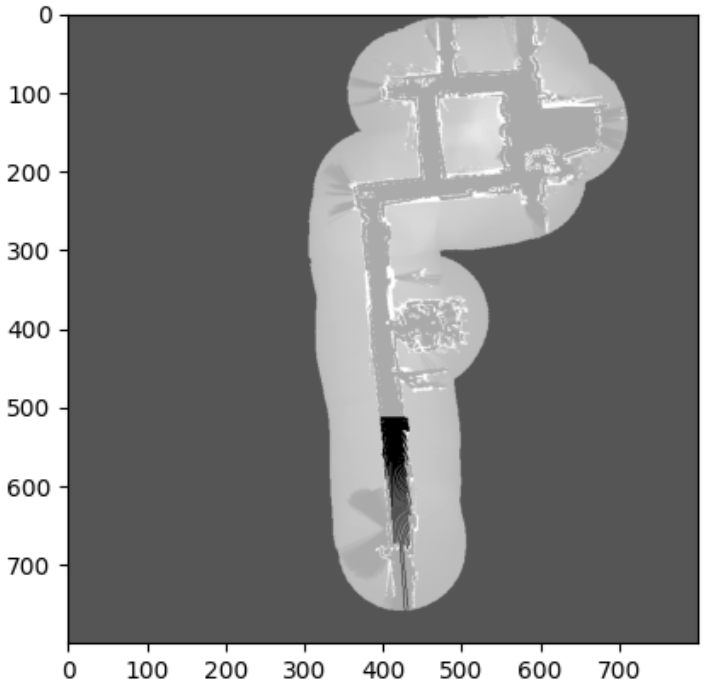

Figure 5: Rays cast at a sample point and orientation

3.3 Sensor Model

3.3.1 Description

The Sensor Model is responsible for estimating how well each particle explains the observed range sensor data. For our implementation, we followed the range sensor model described in [1] where the probability distribution of sensor measurement is modeled as a weighted average of four components that take into account randomness, presence of obstacles and errors in measurements leading to the sensor incorrectly measuring maximum range. These four components are shown in the figure below.

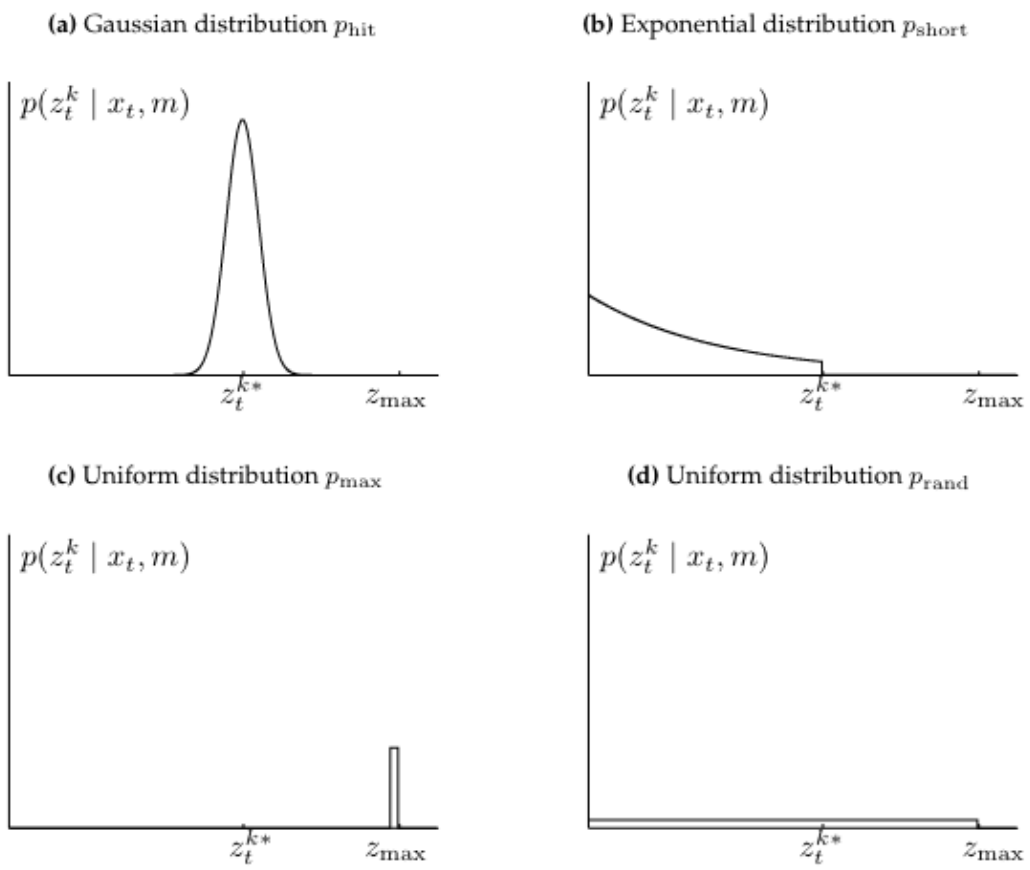

Figure 6: The four components of Sensor model

The Sensor Model has been implemented in *SensorModel* class in *sensor_model.py*. The constructor accepts the occupancy map and number of particles as inputs.

The function *beam_range_finder_model* is called from *main.py* everytime a laser reading is encountered. This function performs three important tasks:

1. Get ground truth range readings using ray casting module
2. Use ground truth and sensor readings to compute probabilities of each measurement for each particle
3. Aggregate probabilities *for* each particle using sum of logarithm of probabilities for numerical stability and then normalize probabilities *across* particles using softmax operation to ensure that probabilities sum to 1.

3.3.2 Implementation

Algorithm beam_range_finder_model(z_t, x_t, m)**:**

$q = 1$

for $k = 1$ *to* K *do*

compute z_t^{k*} *for the measurement* z_t^k *using ray casting*

$$p = z_{\text{hit}} \cdot p_{\text{hit}}(z_t^k \mid x_t, m) + z_{\text{short}} \cdot p_{\text{short}}(z_t^k \mid x_t, m)$$
$$+ z_{\text{max}} \cdot p_{\text{max}}(z_t^k \mid x_t, m) + z_{\text{rand}} \cdot p_{\text{rand}}(z_t^k \mid x_t, m)$$

$q = q \cdot p$

return q

Figure 7: Description of Sensor model

The *SensorModel* module takes up almost all (99.1% according to our estimates) of execution time. This makes optimization absolutely essential. We made the following changes to naive implementation to speed up the code:

1. We vectorized the function *beam_range_finder_model* and all the functions it invokes to compute probabilities of all particles at once instead of invoking the function iteratively for each particle

2. The functions to compute the four components of probability- *p_hit, p_short, p_max, p_rand* have all been vectorized to compute probabilities for all particles simultaneously. An example is shown in figure below

- ```
  def p_short(self, z_t, z_gt):
          #SHORT PROBABILITY
          #initialize probabilties
          p2 = np.zeros_like(z_t)

          #mask for non zero probabilities
          mask = z_t <= z_gt

          #compute normalization factors
          eta = 1/(1 - np.exp(-self._lambda_short*z_gt[mask]))

          #compute probabilties
          p2[mask] =  eta * self._lambda_short*np.exp(-self._lambda_short*z_t[mask])

          return p2
  ```

Figure 8: Vectorized probability computation
  ```

4 Resampling

The inherent nature of this algorithm is survivial of the fittest. Particles with a high probability realtive to the other particles, have a high chance of being retained. Meanwhile, low quality particles have a low probability of being maintained. However, if purely sampling by particle importance weights in a multinomial fashion, there is always an intrinsic variance or "luck" factor involved with random sampling. Therefore, we develop low variance resampling algorithm mentioned in [1] as follows:

Algorithm Low_variance_sampler($\mathcal{X}_t, \mathcal{W}_t$):

$\bar{\mathcal{X}}_t = \emptyset$
$r = \text{rand}(0; M^{-1})$
$c = w_t^{[1]}$
$i = 1$
for $m = 1$ *to* M *do*
  $U = r + (m-1) \cdot M^{-1}$
  while $U > c$
    $i = i + 1$
    $c = c + w_t^{[i]}$
  endwhile
  add $x_t^{[i]}$ *to* $\bar{\mathcal{X}}_t$
endfor
return $\bar{\mathcal{X}}_t$

Figure 9: Psuedo-Code for Low Variance Resampling [1]

Notice in this implementation that only one random number is drawn and there is a constant step size before resampling each particle. This low variance variant ensures that highly probable particles have more of an guarantee of being sampled and can even be sampled multiple times if they are probable enough. This creates a necessity for parameter tuning but also allows for more consistency when resampling.

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

    - Select samples independently of one another
    - We want to make sure that when we have many of the same particles, we don't destroy the variance a
s an estimator
    - We want to make sure we want to not lose diversity and just create samples of the same particle ov
er and over again

    param[in] X_bar : [num_particles x 4] sized array containing [x, y, theta, wt] values for all partic
les
    param[out] X_bar_resampled : [num_particles x 4] sized array containing [x, y, theta, wt] values for
resampled set of particles
    """
    # Resampled particles
    X_bar_resampled =  np.zeros_like(X_bar)

    # Init variables
    M      = X_bar.shape[0]
    r      = np.random.uniform(0, 1.0/M)
    w      = X_bar[:, 3]
    w     /= np.sum(w)
    c      = w[0]
    i      = 0

    # Here we will go through the particles and start the interplay of c and U
    # U will increment a fixed amount each iteration

# c will essentially cumulatively keep track of the sum of the particles weights so far and will look to
exceed U

# Therefore, Very small weighted particles are unlikely but could possibly be sampled, higher weights ha
ve a proportionally better chance
    for m in range(M):
        # Upper bound, when c surpasses this threshold, sample the particle
        U = r + m * (1.0/M)

        while U > c:
            i += 1
            c += w[i]

        # c caught up to U, sample the particle
        X_bar_resampled[m] = X_bar[i]

    return X_bar_resampled
```

Figure 10: Implementation of resampling

4.1 Alterations

Similar to the motion model, we do not perform resampling when the robot is not moving. We found this helps with algorithm speed and removes some noise during the "warmup" period

$$x_t = \begin{cases} Low\_variance\_sampler(x_t, w_t) & \|u_{t1} - u_{t0}\| > \tau \\ x_t & else \end{cases}$$

5 Performance

The total time taken for execution of particle filter on *robotdata1.log* with 1000 particles on Apple Macbook M1 Pro is **19.6 min**. The execution time without the optimizations in ray casting is more than 3 hours(180 min).

Log File	Time(min)
robotdata1.log	19.6
robotdata2.log	63.5
robotdata3.log	34.7
robotdata4.log	17.1
robotdata5.log	27.5

6 Parameter Tuning

6.1 Discussion

Parameter tuning was undoubtedly the most rigorous and important part of the performance for this assignment. For the motion model, tuning $\alpha 1$ and α_2 had the most effect. A value that was too big caused jittering in the angle of the robot. A value that was too small caused the robot to not rotate enough. α_3 and $alpha_4$ had a similar effect but for translation.

For the sensor model, the two most important parameters were z_{rand} and z_{hit}. We used many different values for z_{rand} but found the best performance when this was a large magnitude bigger than the other parameters. When z_{rand} was too small, the particle clouds would collapse and certainly not converge. When z_{rand} was too large, particles would stay alive for far too long and the measurements would have little impact. z_{hit} had a similar effect, modelling our belief in the sensor providing the correct distance. When z_{hit} was too small, there was little chance of convergence. When z_{hit} was too large, there was too much belief in the measurements being made and would collapse potentially correct particle clouds.

6.2 Final Parameters

6.2.1 Motion Model

α_2	α_2	α_2	α_2
1e-4	1e-4	7.5e-4	7.5e-4

6.2.2 Sensor Model

z_{hit}	z_{short}	z_{max}	z_{rand}	σ_{hit}	λ_{short}
10	5	5	1000	75	0.5

7 Results

The results can be found at the links below. Our results match our observations based on the provided reference GIF *robotmovie1.gif* for Log 1. However we suspect that there are errors in result on Log 2.

The errors could be due to our extremely optimized implementation of Ray Casting whose one drawback is introduction of quantization noise because we pre-compute all our rays and round particles along rays to integer coordinates on maps.

The Google Drive links to our videos are provided below. Please note that the video for Log 2 is at a 4x speed to reduce the run time from over 4 minutes to just over 1 minute.

Robot Log 1	Robot Log 2
Log 1	Log 2

8 Future Work

1. As discussed in previous sections, our optimized ray casting performs 9.5x better than naive implementation. However there is scope for further improvement in performance with the use of GPUs that excel at matrix multiplications and additions

2. Another interesting way to speed up ray casting algorithm that we haven't explored for the homework is to approximate all obstacles as a set of straight lines. Under this approximation, we just need to compute points of intersection of a set of lines and any given ray and choose one of the points of intersection as the obstacle. This requires a lot fewer computational resources than ray traversal.

3. Though we do well after tuning parameters, we still have occasional cases where we fail. For logs besides log 1, we are not as robust as we would like. We start to see the particles start to converge but then jump to the wrong location. I think we could get better results if we optimized for each log but that is not ideal in the slightest.

4. In genreal, if we are in an area with more open space and less measureable landmarks, we may require different parameters. For this, having an adaptive number of particles and dynamic/learned parameter values may help. For the adaptive number of particles, as we increase and decrease confidence in the measurements around us, it would be ideal to add more particles to explore and reduce particles when we become more confident in our estimate. Furthermore, learned or adaptive parameters that could have a similar effect but may be able to infer about a shift in the sensor distribution and account for more hardware and noise related issues.

9 Extra Credit

9.1 Kidnapped robot problem

The kidnapped robot problem presents a challenge to robot localization problem since the particles in particle filter tend to concentrate around robot's location. This means that when a robot is kidnapped and taken to a new location, is is highly likely that there are no particles around the robot's new location.

This will cause the particle filter to continue to believe that it is in it's pre-kidapping location and not adjust to it's new position. We can think of two ways to deal with this problem

1. Randomly distribute a small fraction of the particles all over the map inspite of the outcome of particle filter. This will increase the chance of presence of a particle near

robot's new location and thereby increase the likelihood that the robot will adjust to it's new surroundings

2. When a robot is taken to a new location, the sensor readings taken immediately after being placed in the new location will not align with the particles which are still concentrated around the previous state of robot before it was kidnapped. Therefore, one way to detect the kidnapping of a robot is by computing a metric that measures the compatibility of current sensor readings with previous particle states. One such measure could simply be the likelihood $p(z_t|x_t)$ estimated by the sensor model. When the robot is transported to a new location, it is reasonable to assume that $p(z_t|x_t)$ will drop suddenly. Therefore, kidnapping of a robot can be identified when there is a sudden drop in likelihood estimates measured by sensor model. Once the kidnapping has been identified, we can re-initialize particles to random locations to help the robot identify it's new location.

9.2 Adaptive Number of Particles

While implementing this homework, we found increasingly that a high number of particles are extremely useful in the "warmup period" of obtaining sensor readings, it is not as useful once the algorithm is close to convergence. Many of the particles are redundant and will not be useful unless more divergent paths appear (the particles are closer to a uniform distribution).

From these observations, we present the idea of analyzing the entropy of the particles' probabilities in order to inform our algorithm if we need more or less particles. The entropy acts as a level of uncertainity about the distriubtion, with 0 being a single particle with all the probability and 1 being a uniformly distributed particle set. We define entropy w.r.t. weights $w_t^{(i)}$ as:

$$H(w_t) = -\sum_{i=1}^{M} log(w_t^{(i)}) w_t^{(i)}$$

However, entropy by itself is not suitable. It may be more convenient to utilize the entropy at two consecutive timesteps and calculate the the information gain from frame t to frame t+1: If our entropy is low, we want to reduce the number of particles
In the first frame, we have particles that are all evenly weighted. As our probability distribution shifts, we want to adjust the number of particles by whether the difference in entropy (or information gain) is positive or negative. If positive, that means we have become more uncertain so we need to increase the number of particles. If negative, that means we are become less certain so we need to decrease the number of particles. We define the formula to calculate number of particles as:

$$M = Me^{H(w_{t_1}) - H(w_{t_0})}$$

If there is no information gained, the number of particles will stay the same. If the entropy increases, more particles will be added, if the entropy decreases, particles will be removed.

A few implementation details include the frequency of changing the number of particles. Our initial experimentation had a max frequency of 2. That is, at most, the number of particles would change every other time step. This was done to ensure that we are not

artifically inflating the entropy by adding more particles. Therefore, we only calculate information gain when two timesteps have the same number of particles. Lastly, we enforce a minimum number of particles of 50 to ensure our particles do not drop too low.

```python
class AdaptiveParticleCalculator:
    def __init__(self, num_initial, min_particles = 10):
        self.num_initial   = num_initial
        self.min_particles = min_particles
        self.prev_entropy  = None
        self.alpha         = 10000

    def calculate_naive(self, X_bar):
        num_particles = X_bar.shape[0]
        probs         = X_bar[:, -1]
        probs /= np.sum(probs)
        entropy       = -probs * np.log(probs)

        # Only update number of particles if we have same # of particles as last time and we are not in ts 1
        if self.prev_entropy is not None and self.prev_entropy.shape[0] == num_particles:
            num_particles *= np.exp(self.alpha*(np.sum(entropy) - np.sum(self.prev_entropy)))
            num_particles =  max(round(num_particles), self.min_particles)

        # Set previous entropy
        self.prev_entropy = entropy

        sort_idxs = np.argsort(probs)[:num_particles]
        X_bar = X_bar[sort_idxs]

        return X_bar, sort_idxs
```

Figure 11: Code for adaptive particles

References

[1] Sebastian Thrun, Wolfram Burgard, and Dieter Fox (2005) *Probabilistic Robotics*, MIT Press