Department of Electronic and
Telecommunication Engineering

University of Moratuwa
Sri Lanka

EN4593 - Autonomous Systems

Mini Project 02: State Estimation Filters

Particle Filter for Robot Localization

May 17, 2024

A.M.A.D. Adikari

190021A

Contents

List of Figures

1 Introduction

State estimation filters are essential tools in numerous domains such as robotics, signal processing, control theory, etc. They are used to estimate the unobservable state of a system based on existing measurements and knowledge of system dynamics. In the modern world, these filters are used for many applications with their strengths and weaknesses. Traditional state estimation filters like the Kalman filter assume linear Gaussian models, making them highly efficient in many scenarios. However, when faced with non-linearities, non-Gaussian noise, or complex system dynamics, these filters may falter.

The particle filter is a state estimation filter that can be used for such non-linear, non-Gaussian systems and is a powerful alternative to traditional state estimation filters. Compared to the Kalman filters, these are tractable for a lot of large or high-dimensional problems with non-linear dynamics. Furthermore, particle filters are non-parametric filters that allow us to employ the full, complex model, and find approximate solutions.

Particle filters do not rely on a fixed functional form of posterior. They approximate posteriors by a finite number of values, each roughly corresponding to a region in state space. One of the main advantages of these particle filters is their ability to represent complex multimodal beliefs (multiple hypotheses) in state estimations. The particle filter operates in a recursive manner, updating the particle weights based on measurements and resampling them to focus computing resources on regions with high probability density. This adaptability allows the particle filter to accurately track the state of the system even in the presence of significant uncertainty and complexity. The quality of the state approximation depends on the number of particles used to represent the posterior.

These particle filters are used in many industrial applications such as object tracking, Simultaneous Localization and Mapping (SLAM), altitude estimation, signal processing, etc. In this mini project, Robot Localization is implemented using a particle filter. For this scenario, people generally use methods like MCL, AMCL with ROS to localize robots while navigating. Here, robot's pose (position and orientation) would be estimated by using the particle filter.

2 Methodology

In this project, robot's 2D pose is estimated by using a particle filter. The robot's position is given by x and y coordinates and the orientation is given by yaw (θ). Therefore, the estimated pose represents three estimations of the robot's pose.

$$X = \begin{bmatrix} x & y & \theta \end{bmatrix}^T$$

The particle filter uses Monte Carlo methods which are experimental techniques that rely on repeated random sampling to obtain numerical results. And this uses particles to represent the estimation of the state. In this case, the robot poses are represented as particles. A single particle contains an estimation of the robot pose. Therefore, this filter makes multiple hypotheses with the number of particles for the estimation. It starts with random particles and then converge to the actual location where the robot is using motion and sensor data over time.

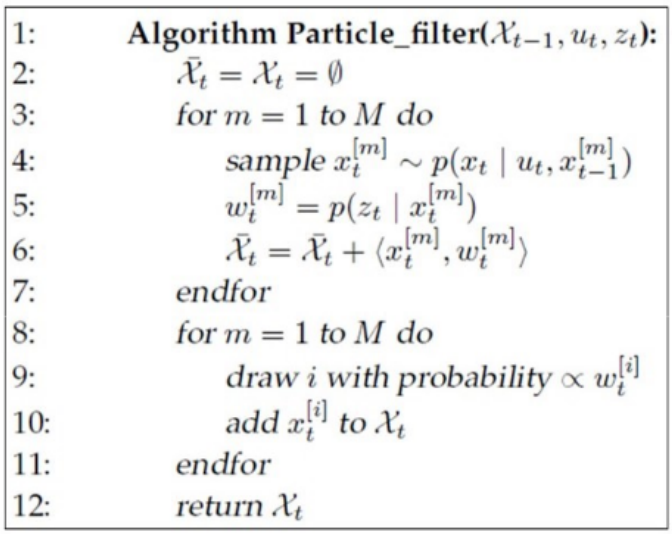


Figure 1: Particle Filter Algorithm

Fig.1 shows the algorithm of the generic particle filter. The algorithm uses previous states, motion data and sensor data as the inputs. The particle filter can be mainly categorized into four steps as follows.

1. Particle initialization
2. Prediction
3. Correction
4. Resample

2.1 Particle initialization

First, M number of random particles are generated from uniform or Gaussian distribution. All the particles are initialized with a weight of 1/M. The particles are represented as follows.

$$\mathcal{X}_t = [x_t^{[1]}\ x_t^{[2]}\ x_t^{[3]} \dots\ x_t^{[M]}]$$

$$w^{[m]} = \tfrac{1}{M};\ \text{m=1,2,3,...,M}$$

2.2 Prediction

The prediction step is done using the motion updates. The robot is driven with the velocity commands. The latest motion control (velocity) command and the previous state are then used to predict the robot's current pose. All the particles are updated with this prediction. Then the belief of the particle set is obtained.

$$\bar{\mathcal{X}}_t = [\bar{x}_t^{[1]}\ \bar{x}_t^{[2]}\ \bar{x}_t^{[3]} \dots\ \bar{x}_t^{[M]}]$$

2.3 Correction

In this step, the importance weights are calculated based on the sensor data of the robot. Then the weights of all particles will be updated and normalized. Therefore, after this correction step, we will have a set of weighted particles which represents a probability distribution of the estimated pose based on particle weights.

$$\mathcal{X}_t = \bar{\mathcal{X}}_t + < \bar{x}_t^{[m]}, w^{[m]} >$$

2.4 Resample

2.4.1 Degeneracy Problem

In a sequential importance sampling framework, the variance of importance weights can only increase over time. After certain number of recursive steps, all but one particle will have normalized weights. Most of particles are at less probable areas while too little particles are at highly likely areas. This is called as degeneracy problem.

2.4.2 Resampling

The resampling is a technique that overcomes the degeneracy problem. In this method, particles with low importance weights are eliminated and the rest are augmented. This increases the accuracy of the pose estimation. The resampling is done using the cumulative density function of the existing normalized particle weights. Using that, new particle set is generated. There are different resampling methods to generate this particle set such as multinomial, stratified, systematic, residual resampling.

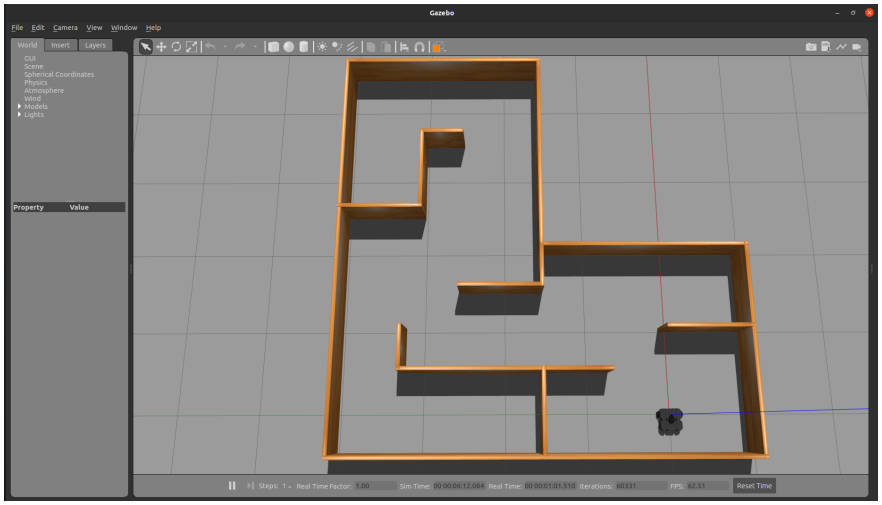
In this robot localization project, all of these particle filtering steps are implemented with different types of distributions and resampling methods.

3 Implementation

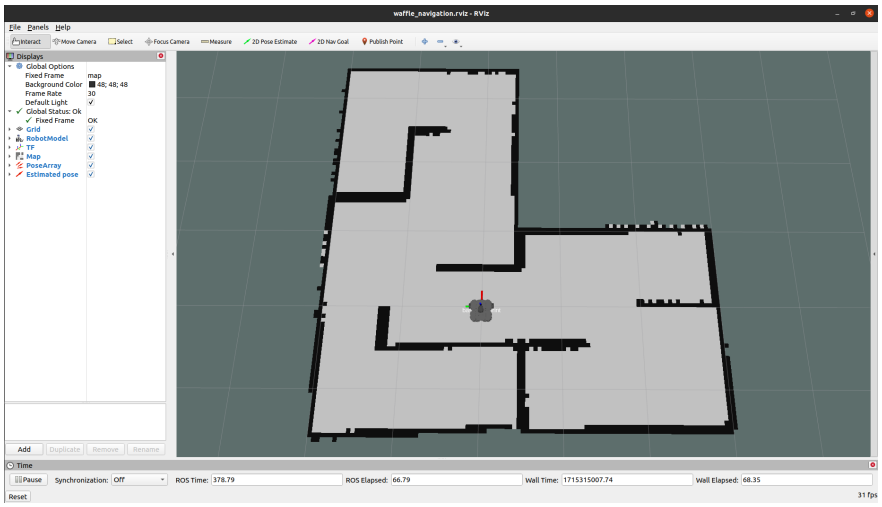
3.1 Introduction

For this project, the particle filter was implemented with ROS noetic framework in Ubuntu 20.04 and the robot localization was simulated using Gazebo and RViz softwares. The turtlebot3 burger robot was used for this project and the built-in Unified Robotics Description Format (URDF) file of the turtlebot3 burger robot was used. To implement the particle filter, only LiDAR and Odometry data were obtained from the robot.

3.2 Simulation Environments



(a) Gazebo environment



(b) RViz environment

Figure 2: Simulation environments at the start of the localization

A pre-built arena was used in Gazebo as shown in the Fig. 2a. And Fig. 2b shows the RViz environment at the corresponding time. First, the arena was mapped using Gmapping SLAM and the teleop-controller. Then, that map was taken for the robot navigation, and the localization was done while the robot was navigating in the given map using the teleop-controller.

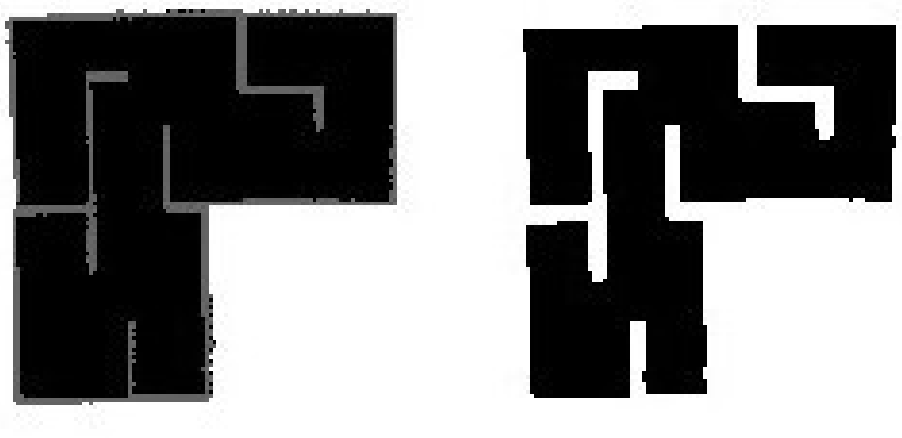
3.3 Particle Filter Implementation

The particle filter was written in Python in Visual Studio Code and built as the *particle_filter_localization* ROS package. Therefore, this works as a single ROS node. For convenience, in the following sections, this localization process will be denoted using Particle Filter Localization (PFL) node. For image processing tasks, OpenCV was used.

3.3.1 Preprocessing

Implementation

Before the localization process, first, the PFL node obtains the static map by calling the *static_map* service from the *map_server* ROS package. This map is received as an occupancy grid and then, it is converted into a 2D image. For calculation purposes, the map image is then converted into a dilated image and also a distance transformation is generated. The dilated image separates the free space, obstacle areas and unknown space from the map image while in the distance transformation, for each pixel in the free space, the distance to the nearest obstacle is calculated. Fig. 3a shows the image of the map. Black area represents the free space and gray areas represent the obstacles. Fig. 3b shows the dilated image of the map.



(a) Map image (b) Dilated map image

Figure 3: Processed map images

Code Explanation

```python
# get the static map
def getMap(self):
    # call the map service
    rospy.loginfo("Waiting for static map service.")
    rospy.wait_for_service("static_map")
    try:
        get_map = rospy.ServiceProxy('/static_map', GetMap)
        response = get_map()
        self.map = response.map     # get the map
        rospy.loginfo("Map received.")
    except rospy.ServiceException as e:
        print("Service call failed:", e)

    # map width and height
    self.map_width = self.map.info.width
    self.map_height = self.map.info.height

    # map origin
    self.map_origin = self.map.info.origin

    # get an image of the map
    self.map_image = np.reshape(self.map.data, (self.map_height, self.map_width)).astype(np.uint8)

    # dilated map image
    kernel = np.ones((3, 3), np.uint8)   # kernel for dilation
    threshold_value = 90
    _, binary_image = cv.threshold(self.map_image.astype(np.uint8), threshold_value, 255, cv.THRESH_BINARY)     # thresholding
    binary_image = np.flip(binary_image, axis=0)    # flip the image since it has a flipped x axis in the RViz map
    self.dilated_map_image = cv.dilate(binary_image, kernel, iterations=1)   # dilate

    # define the map boundaries
    self.map_x_min = self.map_origin.position.x + 10
    self.map_x_max = self.map_x_min + 5
    self.map_y_min = self.map_origin.position.y + 10
    self.map_y_max = self.map_y_min + 5

    # Preprocess the distance transform for fast ray casting
    self.map_image_distance_transform = self.distance_transform(self.map_image)
```

Figure 4: Receiving the map and the preprocessing

Fig. 4 shows the code snippet for the map receiving and the preprocessing section. First, PFL node obtains the map by calling the *static_map* service and saves the map. It extracts the map details like origin, map width and map height and converted into the 2D gray scale image. After applying a binary thresholding, the map image is flipped since RViz uses a flipped map. Then the dilated image is generated. After that, squared boundaries of the map are defined using the origin details and the map size. Finally, the distance transformation is applied to the map image for use in the sensor observation section to reduce the computational complexity.

Fig. 5 shows the distance transformation for the map image. It uses an inverse binary image of the map and calculates the pixel-wise Euclidean (L2) distance from each non-zero pixel to the nearest obstacle.

```python
# get the distance tranformed image
def distance_transform(self, image):
    image_reformat = np.float32(image)

    # Threshold the image to convert a binary image
    ret, thresh = cv.threshold(image_reformat, 50, 1, cv.THRESH_BINARY_INV)
    # Determine the distance transform.
    dist = cv.distanceTransform(np.uint8(thresh), cv.DIST_L2, 0)

    return dist
```

Figure 5: Distance transformation of the map

3.3.2 Particle Initialization

Implementation

After preprocessing, particles are randomly generated inside the map by sampling from a uniform or Gaussian distribution. The distribution type can be selected via a ROS parameter. By default, uniform distribution is set for this purpose. The number of particles can also be defined via a ROS parameter and is set to 500 by default. There are three random generations for a single particle to represent the state (pose) in terms of x, y and θ. The initialization weight is given as 1.0 for each particle. Then, the PFL node checks if the generated particle is in the free space of the map. If not, it discards that particle. This runs in a loop until the PFL node generates the specified number of valid particles. After that, all particle weights are normalized.

Code Explanation

```
# Initialize the particles
def initializeParticles(self):

    # empty array
    particles = []

    # get particles
    for p in range(self.num_particles):
        while True:
            # random particles using defined sampling
            # method (uniform or normal)
            particle = self.getParticles()
            if self.isValidParticle(particle):
                particles.append(particle)
                break

    self.particles_array = particles

    # normalize particle weights
    self.normalizeWeights()
```

Figure 6: Particle initialization

Fig. 6 shows the code snippet for the particle initialization. For generating the random particles, the function shown in Fig. 7 is used. It samples particles from the defined distribution with a weight of 1.0. The x and y values are randomly drawn from [*map_x_min*, *map_x_max*), [*map_y_min*, *map_y_max*) intervals respectively which were defined as map boundaries in the preprocessing section. The θ values are drawn from the [-π, π) interval. Then, the generated particle is checked if that is in the free space of the map. This is done using the function shown in Fig. 8. To check that, x and y coordinates of the particle is converted into a pixel of the map image, and that pixel value of the dilated map image is checked. If it is black, the particle has been generated in the free space. Otherwise, the particle will be discarded. Particle generation runs in a loop until the number of valid particles reaches the specified number of particles.

```python
# get random particles using defined sampling method
def getParticles(self):

    # uniform sampling
    if self.sampling_method == "uniform":
        x = self.uniformSampling(self.map_x_min, self.map_x_max)
        y = self.uniformSampling(self.map_y_min, self.map_y_max)
        theta = self.uniformSampling(-math.pi, math.pi)

        particle = Particle(x, y, theta, 1.0)

    # normal (gaussian) sampling
    elif self.sampling_method == "normal":
        x = self.normalSampling(self.uniformSampling(self.map_x_min, self.map_x_max), 0.05)
        y = self.normalSampling(self.uniformSampling(self.map_y_min, self.map_y_max), 0.05)
        theta = self.normalSampling(self.uniformSampling(-math.pi, math.pi), math.pi/60)
        particle = Particle(x, y, theta, 1.0)

    return particle
```

Figure 7: Random Particle generation

```python
# check whether the particle is inside the map, on an obstacle or outside the map
def isValidParticle(self, particle):
    # get the x and y pixel values for x and y distance vakues
    p_x = self.map_origin.position.x - particle.x
    p_y = self.map_origin.position.y - particle.y
    x_pixel = int(abs(p_x) / self.map.info.resolution)
    y_pixel = self.map_height - int(abs(p_y) / self.map.info.resolution)

    # check the particle with the dilated map
    if self.dilated_map_image[y_pixel][x_pixel] != 255:
        return True
    return False
```

Figure 8: Check for validity of the particle

After generating the particles, all weights are then normalized. Fig. 9 shows function for the normalization. First, it gets the sum of all weights, and then divides each weight by the sum. Then the weights are plotted for visualization purpose.

```python
# normalize the particle weights
def normalizeWeights(self):
    # sum of all weights
    sum_of_weights = sum([p.weight for p in self.particles_array])

    if sum_of_weights == 0:
        print("weights are zero.", len(self.particles_array))
        return

    # normalize
    for p in self.particles_array:
        p.weight = p.weight/sum_of_weights

    # plot the weights
    x = [p.x for p in self.particles_array]
    y = [p.weight for p in self.particles_array]
    self.plotter.set_data(x, y)
    self.plotter.update_plot()
```

Figure 9: Normalization of weights

3.3.3 Motion Update (Prediction)

Implementation

The robot is controlled by using the *teleop_twist_keyboard* ROS package. Instead of using velocity commands, PFL node uses odometry data to predict the robot pose.

$$
\begin{aligned}
\Delta x_t &= x_t - x_{t-1} \\
\Delta y_t &= y_t - y_{t-1} \\
\Delta \theta_t &= \theta_t - \theta_{t-1} \\
\Delta d_t &= \sqrt{(\Delta x_t)^2 + (\Delta y_t)^2}
\end{aligned} \tag{1}
$$

The above equations are used to calculate the position and orientation change of the robot. These are then used to update the particles with some noise as follows.

$$
\begin{aligned}
x_{t+1} &= x_t + (\Delta d_t + n_t) \times \cos(\theta_t) \\
y_{t+1} &= y_t + (\Delta d_t + n_t) \times \sin(\theta_t) \\
\theta_{t+1} &= \theta_t + \Delta \theta_t + r_t
\end{aligned} \tag{2}
$$

where n_t and r_t are random noise of position and orientation respectively at time t. For each particle, noise terms are randomly generated to add the uncertainty. After updating, each particle is checked to see if it has been moved out of the free space of the map. If so, the particle is discarded. Since some particles have been discarded, all of the particle weights are normalized after checking. Then the particles weights are normalized.

Code Explanation

```python
def odomCallback(self, odom_msg):
    # Skip the first call since we are looking for movements
    if not self.prev_odom_msg:
        self.prev_odom_msg = odom_msg
        return

    # # Distance moved since the previous odometry message
    global_delta_x = odom_msg.pose.pose.position.x - self.prev_odom_msg.pose.pose.position.x
    global_delta_y = odom_msg.pose.pose.position.y - self.prev_odom_msg.pose.pose.position.y

    distance = math.sqrt(math.pow(global_delta_x, 2) + math.pow(global_delta_y, 2))

    # Previous robot orientation
    prev_theta = 2 * math.acos(self.prev_odom_msg.pose.pose.orientation.w)

    if self.prev_odom_msg.pose.pose.orientation.z < 0:
        prev_theta = -prev_theta

    # Figure out if the direction is backward
    if (prev_theta < 0 and global_delta_y > 0) or (prev_theta > 0 and global_delta_y < 0):
        distance = -distance

    # Current orientation
    theta = 2 * math.acos(odom_msg.pose.pose.orientation.w)

    if odom_msg.pose.pose.orientation.z < 0:
        theta = -theta

    # Rotation since the previous odometry message
    rotation = theta - prev_theta

    # Return if the robot hasn't moved
    if abs(distance) < 0.01 and abs(rotation) < 0.05:
        # print("removing: ", distance, rotation)
        return
```

Figure 10: Motion update: pose change calculation

Fig. 10 shows the odometry callback function that is used to receive the robot's odometry data. The robot's position and orientation changes are calculated by using (1). If there is

no considerable motion in robot, the particles are not updated. If not, the particles' poses are updated with noises by using (2) as in Fig. 11. After updating, particles outside the free space are discarded and the weights of remaining particles are normalized.

```
# add the distance and rotation for every particle
for p in self.particles_array:
    distance_noise = self.normalSampling(0, self.motion_distance_noise_stddev)
    rotation_noise = self.normalSampling(0, self.motion_rotation_noise_stddev)
    p.x = p.x + (distance + distance_noise) * math.cos(p.theta)
    p.y = p.y + (distance + distance_noise) * math.sin(p.theta)
    theta_ = p.theta + delta_theta + rotation_noise
    p.theta = self.wrap_angle(theta_)

# After updating, keep only the particles that are inside the map
old_particles = copy.deepcopy(self.particles_array)
particles_array = []

for p in old_particles:
    if self.isValidParticle(p):
        particles_array.append(p)

self.particles_array = particles_array

# Normalise particle weights
self.normalizeWeights()
```

Figure 11: Motion update: update particles

After that, the motion update is added to the previously estimated pose as well. This helps to visualize the pose changes in RViz. This does not affect to the overall pose estimation process. Fig. 12 shows the code snippet for this update.

```
# If the estimated pose is valid move it too
# if self.estimated_pose_valid_:
estimated_pose_theta = 2. * math.acos(self.estimated_pose.orientation.w)

if self.estimated_pose.orientation.z < 0.:
    estimated_pose_theta = -estimated_pose_theta

self.estimated_pose.position.x += math.cos(estimated_pose_theta) * distance
self.estimated_pose.position.y += math.sin(estimated_pose_theta) * distance

estimated_pose_theta = self.wrap_angle(estimated_pose_theta + rotation)

self.estimated_pose.orientation.w = math.cos(estimated_pose_theta / 2.)
self.estimated_pose.orientation.z = math.sin(estimated_pose_theta / 2.)

# Increment the motion update counter
self.motion_update_count = self.motion_update_count + 1
```

Figure 12: Motion update: update the previously estimated pose

3.3.4 Sensor Update (Correction)

Implementation

For this step, laser data from the turtlebot3 burger robot LiDAR sensor is used. It gives distances to obstacles along with angle data. This sensor contains 360 laser beams in a completely 360 degree coverage. It is computationally expensive if all beams are considered

for calculation. Therefore, limited number of beams are only considered and this number can be defined via a ROS parameter. These beams are selected with equal angle difference.

PFL node uses a likelihood method to correct the pose estimation. It compares the LiDAR distance data with the expected distances and calculates the likelihood for each particle by using the following equations.

$$Likelihood = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma_z^2}} \exp\left(-\frac{(\bar{z}_i - z_i)^2}{2\sigma_z^2}\right) \tag{3}$$

where,

n - number of laser beams

z_i - range value (obstacle distance) of i^{th} laser beam

$\bar{z}_i$ - expected range value of i_{th} laser beam

σ_z - standard deviation of sensor noise

These likelihoods are then used to update the weights of particles. After that, the robot pose is estimated. It is calculated by using the following equations.

$$\begin{aligned} x_{est} &= \frac{1}{m}\sum_{i=1}^{m} x_i \\ y_{est} &= \frac{1}{m}\sum_{i=1}^{m} y_i \\ \\ \theta_x &= \frac{1}{m}\sum_{i=1}^{m} \cos(\theta_i) \\ \theta_y &= \frac{1}{m}\sum_{i=1}^{m} \sin(\theta_i) \\ \theta_{est} &= \tan^{-1}\left(\frac{\theta_y}{\theta_x}\right) \end{aligned} \tag{4}$$

where m is the number of particles.

Code Explanation

LiDAR data is received by using the callback function shown in Fig. 13. First, it calculates a step count to select only a defined number of laser beams. For each particle, it starts with a likelihood of 1.0 and then, calculates likelihood for each selected beam using (3) and updates the particle's likelihood. After that, the particle's weight is updated with the likelihood by multiplying it. After this process, all the particle weights are then normalized and the robot pose is estimated.

```python
# callback function for laser data from the LiDAR sensor
def scanCallback(self, msg):
    # Corrections after defined number of motion updates
    if self.motion_update_count < self.num_motion_updates:
        return

    # select only few lidar rays to check the likelihood in order to reduce the computational complexity
    # calculate a step count to skip the lidar rays
    step = int(math.floor(float(len(msg.ranges))/self.num_scan_rays))

    self.lock.acquire()

    # Correction step: get the likelihood and multiply with particle weights
    for p in self.particles_array:
        likelihood = 1      # starting likelihood

        for i in range(0, len(msg.ranges), step):

            scan_range = msg.ranges[i]      # laser ray
            laser_angle = msg.angle_increment * i + msg.angle_min       # laser angle
            global_angle = self.wrap_angle(p.theta + laser_angle)

            # get the obstacles distances around the robot
            particle_range = self.hit_scan(p.x, p.y, global_angle, 10.0)

            # calculate the likelihood per ray
            ray_likelihood = 1/math.sqrt(2*math.pi*math.pow(self.sensing_noise_stddev, 2))*math.exp(-math.pow((particle_range-scan_range),2)/(2*math.pow(self.sensing_noise_stddev,2)))

            # update the likelihood
            likelihood *= ray_likelihood

        # update the particle weight with the calculated likelihood
        p.weight *= likelihood

    # normalize the weights
    self.normalizeWeights()

    # estimate the robot pose
    self.estimate_pose()
```

Figure 13: Sensor Update: main code

```python
# Find the nearest obstacle from position start_x, start_y (in meters) in direction theta
def hit_scan(self, start_x, start_y, theta, max_range):

    # Start point in map occupancy grid coordinates
    start_point = [int(round((start_x - self.map.info.origin.position.x) / self.map.info.resolution)),
                   int(round((start_y - self.map.info.origin.position.y) / self.map.info.resolution))]

    # End point in real coordinates
    end_x = start_x + math.cos(theta) * max_range
    end_y = start_y + math.sin(theta) * max_range

    # End point in occupancy grid coordinates
    end_point = [int(round((end_x - self.map.info.origin.position.x) / self.map.info.resolution)),
                 int(round((end_y - self.map.info.origin.position.y) / self.map.info.resolution))]

    # Find the first "hit" along scan
    hit = self.find_obstacles(self.map_image_distance_transform, start_point, end_point)

    # Convert hit back to world coordinates
    hit_x = hit[0] * self.map.info.resolution + self.map.info.origin.position.x
    hit_y = hit[1] * self.map.info.resolution + self.map.info.origin.position.y

    return math.sqrt(math.pow(start_x - hit_x, 2) + math.pow(start_y - hit_y, 2))
```

Figure 14: Sensor Update: convert pixels to world coordinates

The above Fig. 14 shows the function to convert the map pixels into world coordinates when calculating the obstacle distances. First, it convert world coordinates into map pixels to calculate the obstacle distances using the distance transformed image. For this purpose, it uses another function shown in Fig. 15. This function finds the closest obstacle distance for the given beam using the distance transformed image. Using the particle's position and orientation, it checks obstacles along the beam and return the map pixel with the least non-zero obstacle distance. Then, this pixel is converted into world coordinates and the obstacle distance from the particle's position is then calculated. This distance is then used to calculate the likelihood of the beam.

```python
# check whether there are obstacles on a line between two points
def find_obstacles(self, img, p1, p2):
    # coordinates of points
    x1 = float(p1[0])
    y1 = float(p1[1])
    x2 = float(p2[0])
    y2 = float(p2[1])

    # x and y differences
    dx = x2 - x1
    dy = y2 - y1

    # calculate the distance between the two points and normalize
    l = (dx**2 + dy**2)**0.5
    dx = dx / l
    dy = dy / l

    step = 1.0 # pixels
    min_step = 3 # pixels -- too large risks jumping over obstacles
    max_steps = int(l)

    # check obstacles
    dist = 0
    while dist < max_steps:

        # Get the next pixel
        x = int(round(x1 + dx*dist))
        y = int(round(y1 + dy*dist))

        # Check if it's outside of the image
        if x < 0 or x >= img.shape[1] or y < 0 or y >= img.shape[0]:
            return [x, y]

        current_distance = img[y, x]    # distance to the nearest obstacle from the next point

        # Check for osbtacles
        if current_distance <= 0:
            return [x, y]

        # Otherwise, adjust the step size according to the distance transform
        step = current_distance - 1

        # keep the minimum step size
        if step < min_step:
            step = min_step

        # Move along the ray
        dist += step

    # No obstacles found between the two points
    return p2
```

Figure 15: Sensor Update: find obstacle ranges

3.3.5 Resampling

Implementation

In this project, multinomial resampling is used as the resampling method. This resampling occurs when one of the following two conditions occurs.

1. **N_eff < min_particles**
   - effective sample size should be less than a defined minimum number of particles

2. **sensing_update_count > num_sensing_updates**
   - number of sensor updates should be greater than a defined number of sensor update count

num_motion_update, *num_sensing_updates* and *min_particles* parameters are defined at the initialization of the PFL node. These parameters can also be set via ROS parameters. The effective sample size is calculated as follows.

$$N_{eff} = \frac{1}{\sum_{i=1}^{m} w_i^2} \tag{5}$$

Code Explanation

```python
# Resampling
def resample_particles(self):

    # Copy old particles
    old_particles = copy.deepcopy(self.particles_array)
    particles = []

    # Iterator for old particles
    old_particles_i = 0

    # Find a new set of particles by randomly stepping through the old set, biased by their probabilities
    while len(particles) < self.num_particles:

        value = self.uniformSampling(0.0, 1.0)
        sum = 0.0

        # Loop until a particle is found
        particle_found = False
        while not particle_found:

            # If the random value is between the sum and the sum + the weight of the particle
            if value > sum and value < (sum + old_particles[old_particles_i].weight):

                # Add the particle to the array
                particles.append(copy.deepcopy(old_particles[old_particles_i]))

                # Add some noise to the particle
                particles[-1].x = particles[-1].x + self.normalSampling(0, 0.02)
                particles[-1].y = particles[-1].y + self.normalSampling(0, 0.02)
                particles[-1].theta = self.wrap_angle(particles[-1].theta + self.normalSampling(0, math.pi / 120))

                # Break out of the loop
                particle_found = True

            # Add particle weight to sum and increment the iterator
            sum = sum + old_particles[old_particles_i].weight
            old_particles_i = old_particles_i + 1

            # If the iterator passes number of particles, loop back to the beginning
            if old_particles_i >= len(old_particles):
                old_particles_i = 0

    # resampled particle array
    self.particles_array = particles

    # Normalise the new particles
    self.normalizeWeights()

    # Don't use the estimated pose just after resampling
    self.estimated_pose_valid = False

    # Induce a sensing update
    self.motion_update_count = self.num_motion_updates
```

Figure 16: Resampling

Fig. 16 shows the code snippet used for the resampling. First, it keeps the old particles and then draws new random particles from the cumulative distribution of old particles set with some noise. After generating, all the particle weights are normalized.

3.4 ROS Implementation

3.4.1 Publishers and Subscribers

```
# ROS publishers and subscribers
self.particlesPub = rospy.Publisher("/particlecloud", PoseArray, queue_size=10)
rospy.Subscriber("/scan", LaserScan, self.scanCallback, queue_size=1)
rospy.Subscriber("/odom", Odometry, self.odomCallback, queue_size=1)

self.particles_sequence_number = 0      # publishing sequence number of particles array
self.particles_pub_timer = rospy.Timer(rospy.Duration(0.1), self.publishParticles)      # publish particles at 0.1 second intervals

self.estimated_pose_pub = rospy.Publisher('estimated_pose', PoseStamped, queue_size=1)
self.estimated_pose_sequence_number = 0     # estimated pose sequence number for publishing
self.estimated_pose_pub_timer = rospy.Timer(rospy.Duration(0.1), self.publishEstimatedPose)     # publish the estimated pose at 0.1 second intervals

self.transform_broadcaster = tf2_ros.TransformBroadcaster()     # sending transformations
self.transform_sequence_number = 0          # transformation sequence number for publishing
```

Figure 17: ROS publisher and subscribers

Fig. 17 shows the code snippet for initialization of ROS publishers and subscribers. There are two publishers for particles and estimated pose publishing and two subscribers for receiving odometry and LiDAR data. Publishers are initialized to publish data at 0.5 second time intervals. There is also a transform broadcaster to send transformations from "map" frame to "odom" frame using the estimated pose.

```
#  publish the transformation from map to base_footprint
transform = TransformStamped()

transform.header.frame_id = "map"
transform.header.stamp = rospy.Time.now()
transform.header.seq = self.transform_sequence_number
self.transform_sequence_number = self.transform_sequence_number + 1

transform.child_frame_id = "base_footprint"

transform.transform.translation.x = self.estimated_pose.position.x
transform.transform.translation.y = self.estimated_pose.position.y

transform.transform.rotation.w = self.estimated_pose.orientation.w
transform.transform.rotation.z = self.estimated_pose.orientation.z


# get odom to base_footprint tf
listener = TransformListener()
while not rospy.is_shutdown():
    try:
        # print("getting odom")
        listener.waitForTransform("/odom", "/base_footprint", rospy.Time(), rospy.Duration(10.0))
        (odom_to_base_trans,odom_to_base_rot) = listener.lookupTransform('/odom', '/base_footprint',  rospy.Time())

        break
    except Exception as e:
        continue

# calculate map to odom tf using map to base_footprint estimated pose and obtained odom to base_footprint tf
map_to_base_trans = [self.estimated_pose.position.x, self.estimated_pose.position.y, self.estimated_pose.position.z]
map_to_base_rot = [self.estimated_pose.orientation.x, self.estimated_pose.orientation.y, self.estimated_pose.orientation.z, self.estimated_pose.orientation.w]

map_to_base_matrix = quaternion_matrix(map_to_base_rot)
map_to_base_matrix[:3, 3] = map_to_base_trans

odom_to_base_matrix = quaternion_matrix(odom_to_base_rot)
odom_to_base_matrix[:3, 3] = odom_to_base_trans

odom_to_base_inverse = inverse_matrix(odom_to_base_matrix)

map_to_odom_matrix = concatenate_matrices(map_to_base_matrix, odom_to_base_inverse)

map_to_odom_rot = quaternion_from_matrix(map_to_odom_matrix)
map_to_odom_trans = map_to_odom_matrix[:3, 3]

# create the tf
map_to_odom_transform = TransformStamped()
map_to_odom_transform.header.stamp = rospy.Time.now()
map_to_odom_transform.header.frame_id = "map"
map_to_odom_transform.child_frame_id = "odom"
map_to_odom_transform.header.seq = self.transform_sequence_number
self.transform_sequence_number = self.transform_sequence_number + 1

map_to_odom_transform.transform.translation.x = map_to_odom_trans[0]
map_to_odom_transform.transform.translation.y = map_to_odom_trans[1]
map_to_odom_transform.transform.translation.z = map_to_odom_trans[2]
map_to_odom_transform.transform.rotation.x = map_to_odom_rot[0]
map_to_odom_transform.transform.rotation.y = map_to_odom_rot[1]
map_to_odom_transform.transform.rotation.z = map_to_odom_rot[2]
map_to_odom_transform.transform.rotation.w = map_to_odom_rot[3]

# publish
self.transform_broadcaster.sendTransform(map_to_odom_transform)
```

Figure 18: Create transformations from "map" to "odom"

Fig. 18 shows the code snippet for transformation creation from "map" to "odom" frame. The turtlebot3 burger robot contains a "base_footprint" link. The estimated pose gives an estimation of "map" to "base_footprint" frame transformation. After a pose estimation, transformation of "odom" to "base_footprint" frame is obtained by using a transform listener. Two transformation matrices can then be created using this transformations data. Then, we can get the transformation of "map" to "odom" frame as follows.

$$T_{odom}^{map} = T_{base\_footprint}^{map} \times \left(T_{base\_footprint}^{odom}\right)^{-1} \tag{6}$$

From this calculated transformation, PFL node extracts the robot position and orientation data and send the "map" to "odom" transformation.

3.4.2 ROS parameters

```
# ROS parameters
self.sampling_method = rospy.get_param("~sampling_method", "uniform")   # Sampling method for particle initialization
self.num_particles = rospy.get_param("~num_particles", 500) # Number of particles
self.num_motion_updates = rospy.get_param("~num_motion_updates", 5) # Number of motion updates before a sensor update
self.num_scan_rays = rospy.get_param("~num_scan_rays", 20) # Number of scan rays to evaluate
self.num_sensing_updates = rospy.get_param("~num_sensing_updates", 5) # Number of sensing updates before resampling
self.motion_distance_noise_stddev = rospy.get_param("~motion_distance_noise_stddev", 0.01) # Standard deviation of distance noise for motion update
self.motion_rotation_noise_stddev = rospy.get_param("~motion_rotation_noise_stddev", math.pi / 120) # Standard deviation of rotation noise for motion update
self.sensing_noise_stddev = rospy.get_param("~sensing_noise_stddev", 0.5) # Standard deviation of sensing noise
```

Figure 19: ROS parameters

Fig. 19 shows ROS parameters of the PFL node. These parameters can be set via a launch file or when the script is run. The figure shows the parameter list and the default values. Here, we can set the sampling method as "uniform" or "normal" to initialize the particles. Other parameters are values and any relevant value can be set.

4 Results

4.1 Transformation (TF) tree

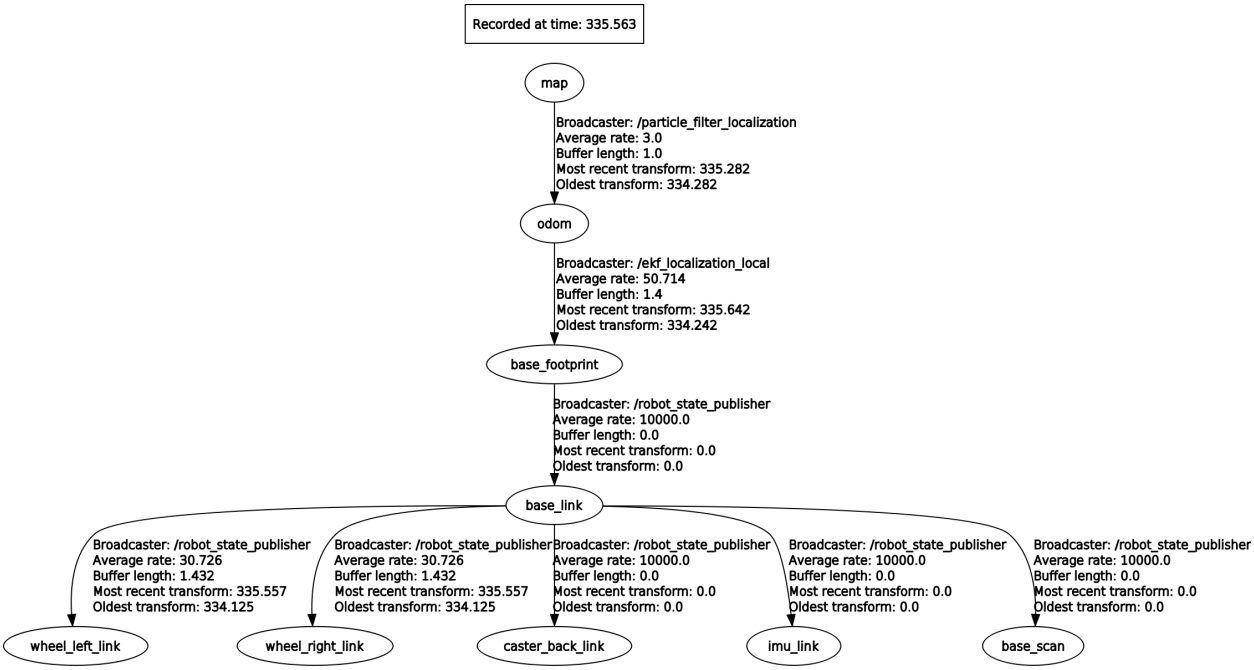


Figure 20: Transformation (TF) tree

Fig. 20 shows the complete transformation (tf) tree for this particle filter localization. The ROS package is used to publish link transformations including the main wheels, caster wheel, IMU and lidar of the turtlebot3 burger robot. In this project, *robot_localization* ROS package is used to fuse odometry and IMU data of burger robot. This is done by using an extended kalman filter (EKF) node in the *robot_localization* package. And that node also publishes the "odom" to "base_footprint" transformation. As mentioned previously, PFL node publishes the "map" to "odom" transformation using the estimated robot pose.

4.2 ROS nodes

Fig. 21 shows the ROS nodes graph with the initialized ROS nodes. These ROS nodes are used for following purposes.

- **particle_filter_localization** - localize the robot in the given map using the particle filter.
- **gazebo** - simulate the robot navigation
- **rviz** - visualize the robot navigation and localization
- **robot_state_publisher** - publish the robot link transformations

- **joint_state_publisher** - publish the joint states (For the turtlebot3 robot, it publishes states of two main wheel joints)
- **map_server** - publishing the static map
- **ekf_localization_local** - fuse odometry and IMU data into a single combined odom and publish "odom" to "base_footprint" transformation
- **teleop_twist_keyboard** - publish velocity commands to the robot

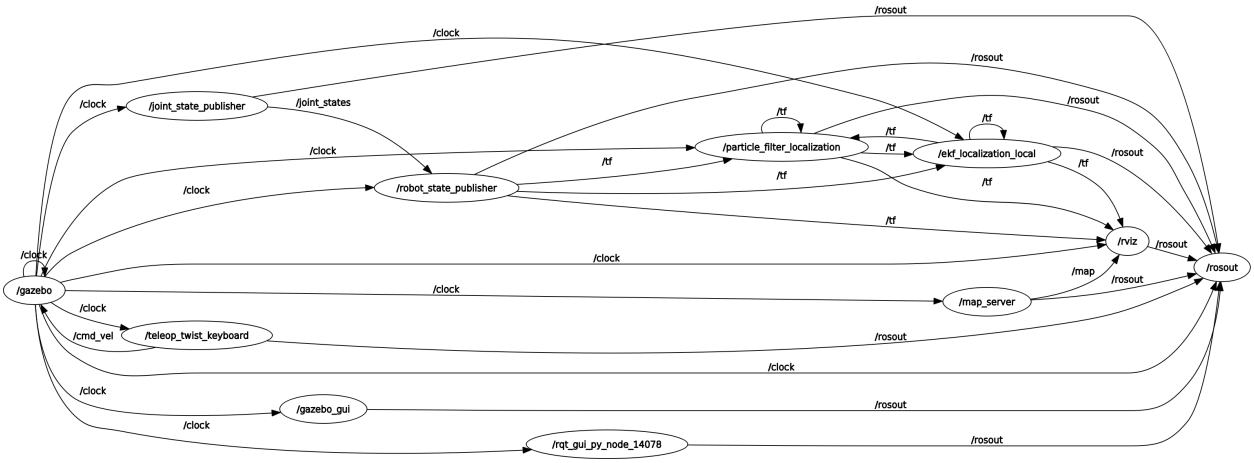


Figure 21: ROS nodes graph

4.3 Demonstration Video

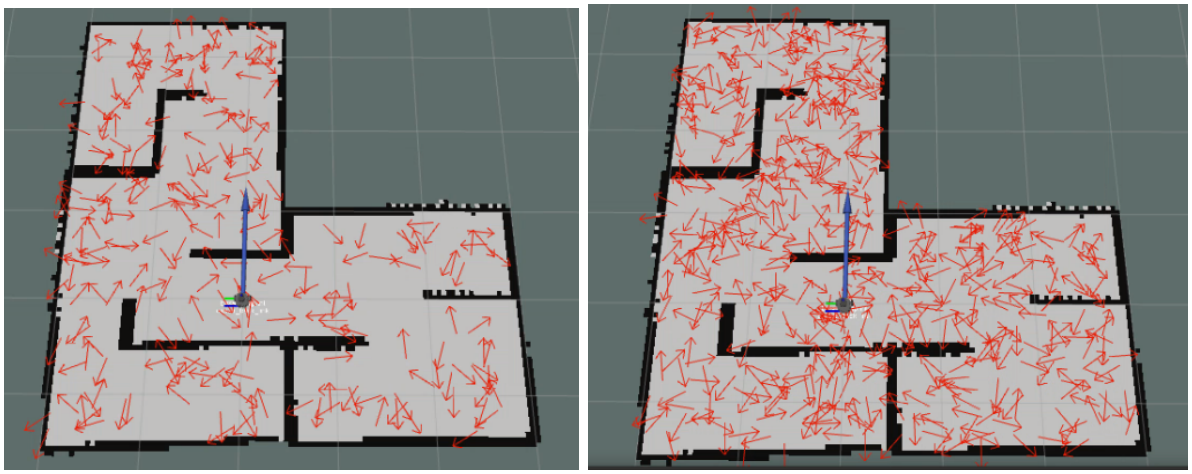
Two demonstration videos have been uploaded to the DMS. One video is for localization with 250 particles and the other one is with 500 particles.

DMS link - https://dms.uom.lk/s/iTHmqYqmyYgdy3i

4.4 Particle Filter Localization

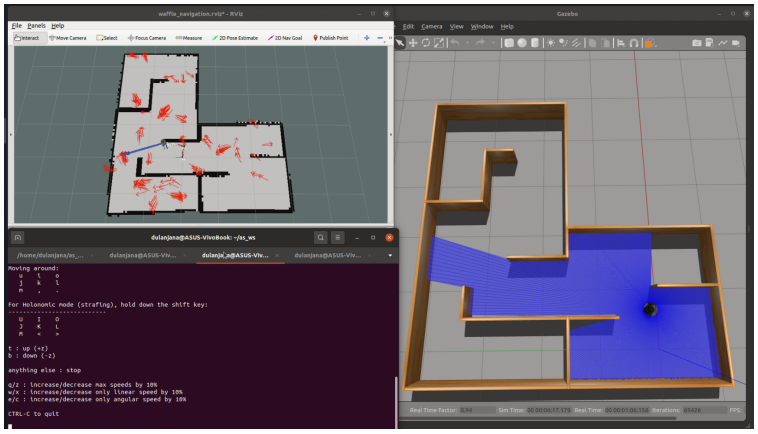
Fig. 22 shows the particle initializations with 250 and 500 particles in two separate runs. We can see that the particle coverage in Fig. 22a is not good compared to Fig. 22b.

Fig. 23 shows resampled particles at a middle stage of the localization. For both these runs, minimum number of particles was set to 300 to compare with the effective sample size. We can clearly see that there are many particle clusters on the map. The number of clusters are higher for larger number of particles. Then, we can see the localized moments of two runs in Fig. 24. Here, the PFL with 500 particles has taken longer to localize due to the larger number of clusters. Fig. 25 shows the localizations of two PFL runs at the destinations.

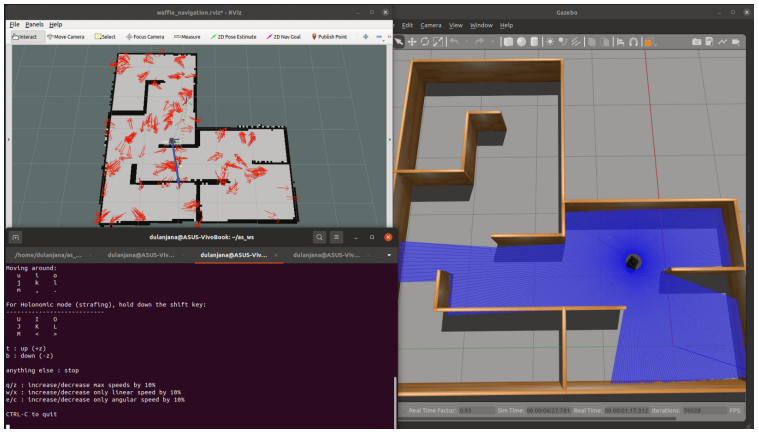


(a) PFL with 250 particles (b) PFL with 500 particles

Figure 22: Particle Initialization

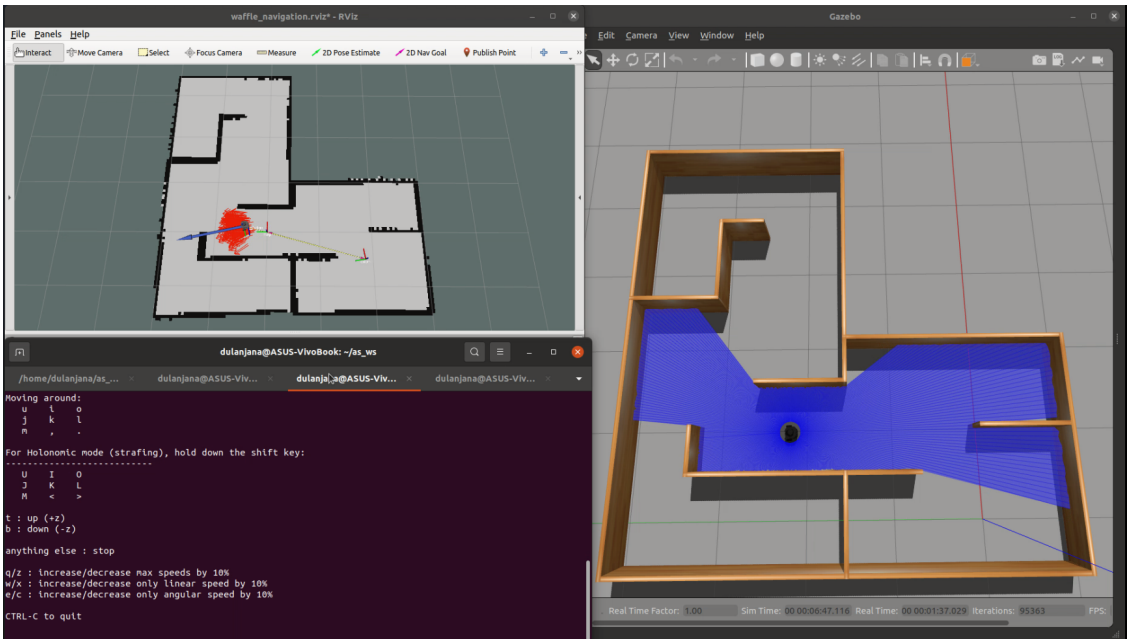


(a) PFL with 250 particles

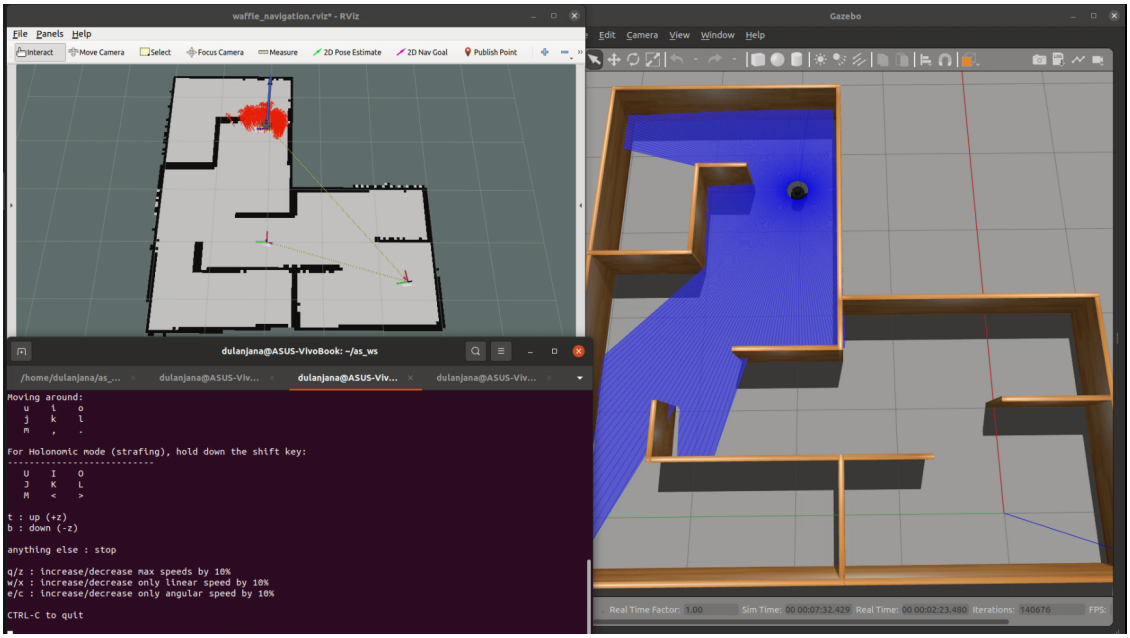


(b) PFL with 500 particles

Figure 23: An intermediate resample

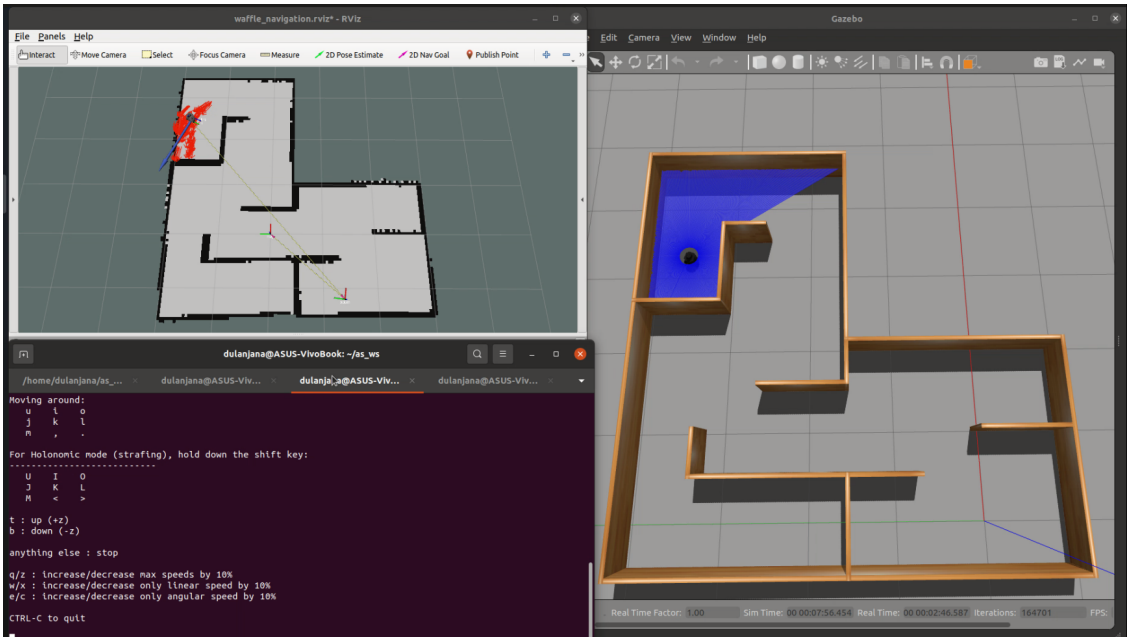


(a) PFL with 250 particles

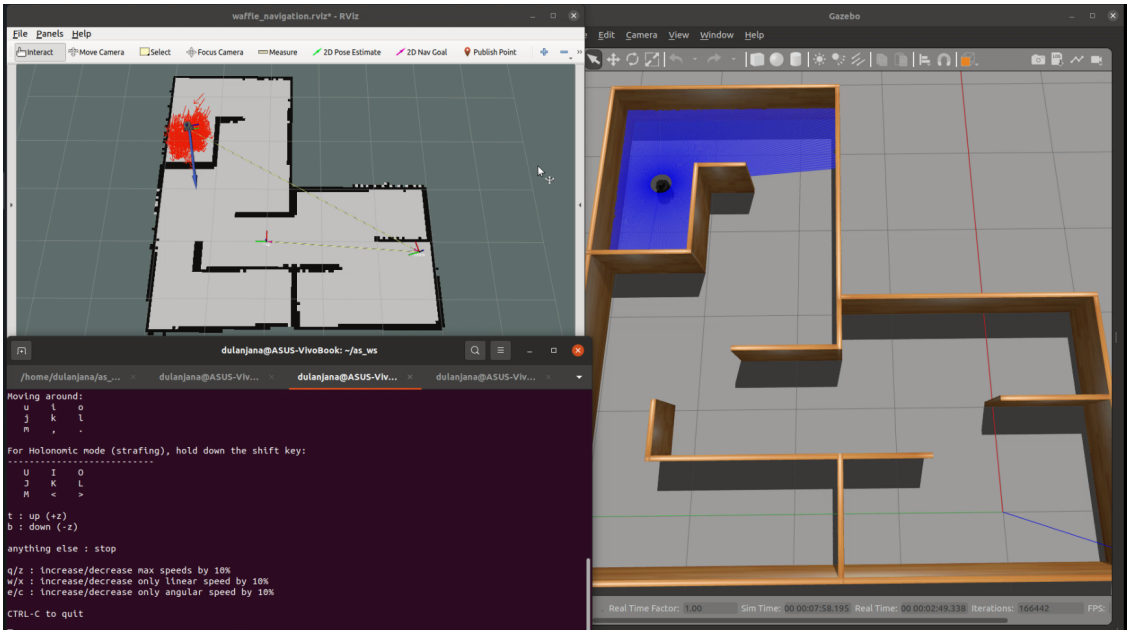


(b) PFL with 500 particles

Figure 24: Localized moment



(a) PFL with 250 particles



(b) PFL with 500 particles

Figure 25: At the destination

4.5 Analysis

In this two runs, we can clearly see that the PFL with larger number of particles is initialized well with a complete coverage on the free space of the map by analysing Fig. 22. If the number of particles are low, random particles may not be covered the whole free space. This may lead to a localization failure. We should use an enough number of particles to localize the robot well.

By analysing Fig. 24, we can see that the PFL with larger number of particles takes more time to localize on the map. In Fig. 23, there are more number of particle clusters for PFL with 500 particles. Therefore, it takes more time to localize to the correct cluster. Then, we can see if the number of particles is larger, it takes longer to localize.

By analysing two videos, we can see the accuracy of the PFL with 500 particles is better than the other, but the variance of the localized cluster may be higher. Overall, the robot localization is correctly done using the particle filter.

5 Discussion

In this mini project, the robot localization was implemented with a particle filter in a given environment. The experimental results showed that the particle filter could effectively handle the localization also with the noise and uncertainties.

For this localization, I experimented many steps by changing the parameters. I used both normal and uniform distributions for particle initialization. But for this environment, it did not much change the output results. Then, I experimented with the number of beams to use in sensor update. First, it was 6 beams, but the results was not good. Then, I experimented by increasing it and finally set it to 20.

In motion update, first I used velocity commands to update the motions of estimated pose and the particles. But it was too much computationaly expensive. Then, I used the odometry for this purpose and it worked well. I experimented each multinomial, stratified and systematic method for resampling and multinomial method gave good results. Therefore, I used multinomial resampling method in the above runs.

The particle filter gives best results in robot localization. But if the parameters are not properly set, it leads to a localization failure. In this project also, I faced some limitations. As in Fig. 22, the PFL node with lower number of particles does not cover the whole free space when initialized. Sometimes, there were no particles at the robot's actual ground truth position. In such scenarios, localization can not be done properly. Also, in Fig. 24, it takes longer to localize properly when the number of particles are larger. Larger number of particles help to cover the whole free space but takes more time to localize. Therefore, we need to find a proper number of particles to a fast and accurate localization.

In the industry, particle filters are used for robot localization with algorithmic modifications. This study can be further developed by using adaptive algorithms, more sensor fusions and computational efficiency.

6 Conclusion

In conclusion, we found that the particle filter is an effective tool for robot localization, offering robust performance despite sensor noise and environmental uncertainties. While there are limitations regarding computational demands and parameter sensitivity, the benefits far outweigh these challenges. This mini project is a good opportunity to learn the theoretical background of the particle filter while applying it to a real-world application. Also, this was a good start to get a foothold in these fields.

7 References

1. Resampling Methods for Particle Filtering - Resampling Methods for Particle Filtering
2. AMCL ROS package - https://github.com/ros-planning/navigation
3. MCL_PI ROS package - https://github.com/or-tal-robotics/mcl_pi_gazebo