Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

b-it Bonn-Aachen International Center for Information Technology

Study Project on Lecture
"Probabilistic Reasoning"

Occupancy Grid Map and Particle Filter Localization in ROS2

Hamsa Datta Perur (hperur2s)
Kishan Ravindra Sawant (ksawan2s)

Supervised by

Deebul Nair, M.Sc.

January 2023

1 Problem Overview

In this project occupancy grid mapping and particle filter will be implemented in simulation environment [1] in ROS2. There exist many assumptions in the current open-source implementations of particle filters including the ROS navigation tools. The main goal of this project is to understand the influence of these assumptions and parameterise them. This can be used as a generalised software for robot navigation.

The private repository of this project can be found here.

2 Occupancy Grid Mapping

2.1 Overview

Occupancy grid mapping is a technique of representing and storing the environment, which can be further used for localisation and navigation. For a given input size of the region to be mapped in an environment and for a predefined resolution, the map is represented as an array with every individual grid representing the probability of occupancy.

In the simulated environment, the odometry data representing the location of the robot is considered to be accurate without any uncertainty. The Bayesian rule is used to update the state of occupancy of individual cells based on the cells being hit or passed through by the laser scanner data. Finally, based on a defined threshold, the occupancy status of a cell is determined and it forms a map, which is a representation of an environment.

2.2 Common Problems

Some of the problems faced in the occupancy grid mapping are:

1. In occupancy grid mapping occupancy status of an individual cell is considered to be independent of that of neighbouring cells, which is not an accurate assumption while dealing with uncertain perception data. Considering the

neighbourhood of a cell in the measurement model and parameterising the mapping task by the size of the neighbourhood.

2. Some works consider odometry data as certain knowledge. By using Rao-Blackwellized Particle Filter, posterior over the map as well as position can be predicted [2].

3. Allowing different types of noises to model sensor and odometry data.

4. Based on confidence of localisation, updating the map. This might help to handle dynamic obstacles.

2.3 Code

The following implementation of occupancy grid mapping is derived from grid-mapping-in-ROS repository.

2.3.1 Structure

```
probabilistic_reasoning_hbrs
 ├── src
 │    └── occupancy_grid_mapping
 │         └── occupancy_grid_mapping
 │              ├── gmapping_node.py
 │              └── submodules
 │                   ├── bresenham.py
 │                   ├── grid_map.py
 │                   ├── utils.py
 │                   └── message_handler.py
 └── README.md
```

2.3.2 Description of Code

1. The probability values for cells being occupied, free or unseen (assigned with prior probability) are set. The size, resolution and name of the grid map are set.

```
P_prior = 0.5  # Prior occupancy probability
P_occ = 0.9    # Probability that cell is occupied
P_free = 0.3   # Probability that cell is free
RESOLUTION = 0.03   # Grid resolution in [m]
MAP_NAME = 'world'  # map name without extension

map_x_lim = [-10, 10]
map_y_lim = [-10, 10]
```

2. 'GMappingClass()' is used to continuously access polar positions of individual scan data and the pose of the robot with respect to its odom frame of reference.

```
distances, angles, _ = node.distances, node.angles
x_odom, y_odom, theta_odom = node.x_odom, node.y_odom,
                             node.theta_odom
```

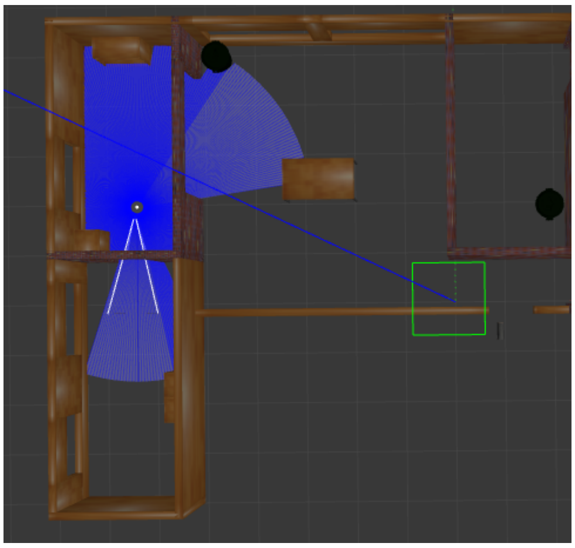
3. 'descritize' method from the 'GridMap' class (imported from submodules/-grid_map.py) is used to assign the index to the odometry data with respect to the initialised size of the grid. It should be noted that the size of the map should be large enough to accommodate all possible odometry values.

```
x1, y1 = gridMap.discretize(x_odom, y_odom)
```

```
def discretize(self, x_cont, y_cont):
    """
    Discretize continious x and y
    """
    x = int((x_cont - self.X_lim[0]) / self.resolution)
    y = int((y_cont - self.Y_lim[0]) / self.resolution)
    return (x,y)
```

It can be observed that if the robot is at [map_x_lim[0], map_y_lim[0]], then it is assigned with the indices of [0,0].

4. $X2$ and $Y2$ variables are used to store all points measured from the laser scanner, which is represented by green colour in the grid map while visualising the map. It also consists of the points which have reached the maximum range of the laser scanner.


(a) Environment for mapping

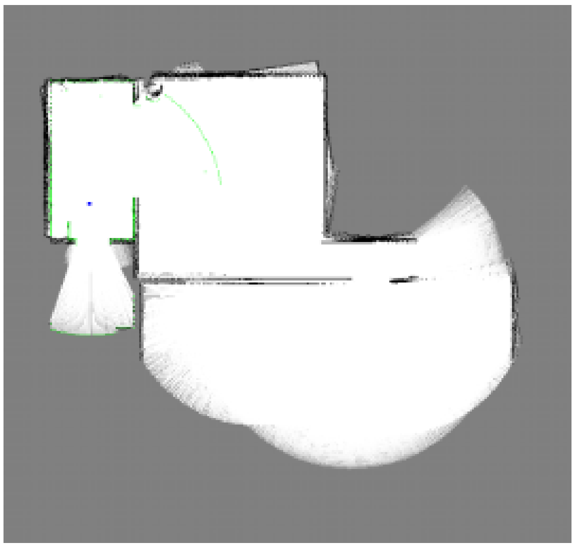

(b) Grid map visualisation

Figure 1: Visualisation of the process of mapping

In Figure 1(b), the green color scan data represents the cells being hit at current instance, the white cells are the detected free cells, the black cells are the occupied cells and the grey colored cells are yet to be discovered and are assigned with prior probability.

‘bresenham’ method form ‘submodules/bresenham.py’ is used to determine all the cells which are free and this information is updated in the grid map. All the cells which are within the maximum range of the laser scanner are considered occupied and are updated in the grid map.

```python
# for BGR image of the grid map
X2 = []
Y2 = []

for (dist_x, dist_y, dist) in zip(distances_x, distances_y,
    distances):
    # x2 and y2 for Bresenham's algorithm
    x2, y2 = gridMap.discretize(dist_x, dist_y)

    # draw a discrete line of free pixels, [robot position ->
     laser hit spot)
    for (x_bres, y_bres) in bresenham(gridMap, x1, y1, x2, y2
    ):
        gridMap.update(x=x_bres, y=y_bres, p=P_free)

    # mark laser hit spot as ocuppied (if exists)
    if dist < node.range_max:
        gridMap.update(x=x2, y=y2, p=P_occ)

    # for BGR image of the grid map
    X2.append(x2)
    Y2.append(y2)
```

5. update method from ‘grid_map.py’ is used to update the occupancy status of a cell based on inverse sensor model $p(x|z_t)$, where x is the state of the cell being occupied and z_t is the measurement from laser scan data at time t.

   From Bayes’ rule, we know that,

$$p(x|z_{1:t}) = \frac{p(x|z_t)p(z_t)}{p(x)} \frac{p(x|z_{1:t-1})}{p(z_t|z_{1:t-1})} \tag{1}$$

   Similarly, the probability of a cell being free given the history of laserscan data will be,

$$p(\bar{x}|z_{1:t}) = \frac{p(\bar{x}|z_t)p(z_t)}{p(\bar{x})} \frac{p(\bar{x}|z_{1:t-1})}{p(z_t|z_{1:t-1})} \tag{2}$$

The log of ratio of $p(x|z_{1:t})$ and $p(\bar{x}|z_{1:t})$ can be written as a recurcive formulation as,

$$l_t(x) = log\frac{p(x|z_t)}{p(\bar{x}|z_t)} + log\frac{p(\bar{x})}{p(x)} + l_{t-1}(x) \tag{3}$$

```python
def update(self, x, y, p):
    """
    Update x and y coordinates in discretized grid map
    """
    # update probability matrix using inverse sensor model
    self.l[x][y] += log_odds(p)
```

'log_odds(p)' method from 'grid_map.py' is used to update the occupancy probability of every cell. Here $(1 - p)$ represents the probability of cell being free.

```python
def log_odds(p):
    """
    Log odds ratio of p(x)
    """
    return np.log(p / (1 - p))
```

6. In the final step, 'calc_MLE()' method is used to discretize the probability value assigned to each cell as occupied, free or unknown. Further, 'calc_surface_normal()' method is used to calculate and store the normals at each occupied cells. This will help to overcome 'particle degeneracy' problem that will be discussed in the following section on 'Particle Filter Localization'.

```python
def calc_MLE(self):
    """
    Calculate Maximum Likelihood estimate of the map
    """
    for x in range(self.l.shape[0]):
        for y in range(self.l.shape[1]):
            # if cell is free
            if self.l[x][y] < log_odds(TRESHOLD_P_FREE):
                self.l[x][y] = log_odds(0.01)
            # if cell is occupied
            elif self.l[x][y] > log_odds(TRESHOLD_P_OCC):
                self.l[x][y] = log_odds(0.99)
            # if cell state uncertain
            else:
                self.l[x][y] = log_odds(0.5)
```

3 Particle Filter Localization

3.1 Overview

- A particle filter is a recursive algorithm that uses a set of discrete particles to represent the probability density function (PDF) of the state of a system. Each particle represents a possible state of the system and it is associated with a weight that represents the probability of that state being the exact representation of the system.

- At each time step, the filter uses data from sensors, such as cameras or LIDAR, and other sources, such as odometry, to update the weights of each particle. This is done by comparing the sensor data to the predicted measurements for each particle, using a likelihood function. Particles with higher likelihoods are given higher weights, while particles with lower likelihoods are given lower weights.

- After the weights have been updated, the filter resamples the particles to create a new set of particles, with each particle being chosen with probability proportional to its weight. This ensures that the particles with higher weights are more likely to be selected and that the filter is not dominated by a few highly likely particles.

- The process of updating the weights and resampling the particles is repeated at each time step, creating an estimate of the true state of the system.

3.2 Common Problems

The below are some of the problems of the particle filter based on the survey by J. Elfring et al.[3].

- Particle depletion: If the number of particles is not high enough, the filter can become dominated by a few highly likely particles, leading to particle depletion. This can cause the filter to become stuck in a local minimum, and can lead to poor estimates of the state of the system.

- Curse of dimensionality: As the number of dimensions of the state space increases, the number of particles required to accurately represent the PDF increases exponentially. This can make it computationally infeasible to use particle filters in high-dimensional state spaces.
- Sensitivity to initialization: Particle filters are sensitive to the initial set of particles used to represent the PDF. If the initial particles are not chosen carefully, the filter may converge to a poor estimate of the state of the system.
- Computational complexity: Particle filters can be computationally expensive, particularly when the number of particles is high. This can make them infeasible for real-time applications.
- Dealing with multimodal distributions: Particle filter could struggle to estimate the state of the system when the true state has multiple modes, which means multiple regions of high probability density.
- Handling non-linearity and non-gaussianity: Particle filter is designed to handle non-linear and non-gaussian systems, but it still can be challenging to design a good proposal distribution in these cases.

3.3 Code

The following implementation of particle filter is derived from particle_filter_project repository, which is converted to ROS2.

3.3.1 Structure

```
probabilistic_reasoning_hbrs
└── src
    └── particle_filter
        ├── launch
        │   └── server.launch.py
        └── map
            ├── house_map.pgm
            └── house_map.yaml
└── particle_filter
    └── pf.py
```

3.4 Description of Code

1. ‘map’ variable is assigned to OccupancyGrid() message-type, which will be used to store the grid map by listening to the map_server. 2000 random particles are used to represent possible robot poses. The average of these poses is considered to be the estimated pose of the robot. Whenever there is a linear motion of 0.2 m in ‘x’ or ‘y’ direction or when there is a rotation of 30 degrees, the particles are updated along with the estimated pose of the robot.

   The particle weights will be further updated by considering the closeness of the laser scan data in the directions corresponding to angles in ‘directions_to_check’. These laser scan data are obtained by subscribing to the ‘scan’ topic. The particles and the estimate of the robot pose are published and they can be visualised in rviz visualiser.

```

        # inialize our map (message type)
        self.map = OccupancyGrid()

        # the number of particles used in the particle filter
        self.num_particles = 2000

        # initialize the particle cloud array
        self.particle_cloud = []

        # initialize the estimated robot pose
        self.robot_estimate = Pose()

        # set threshold values for linear and angular
   movement before we perform an update
        self.lin_mvmt_threshold = 0.2
        self.ang_mvmt_threshold = (np.pi / 6)

        self.odom_pose_last_motion_update = None
```

```python

        # indicate the angles that we plan on checking for particle weight update
        self.directions_to_check = [(math.pi/2*index) for index in range(4)]

        # Setup publishers and subscribers

        # publish the current particle cloud
        self.particles_pub = self.create_publisher(
            PoseArray, 'particle_cloud', 10)

        # publish the estimated robot pose
        self.robot_estimate_pub = self.create_publisher(
            PoseStamped, 'estimated_robot_pose', 10)

        # subscribe to the map server
        self.create_subscription(
            OccupancyGrid, self.map_topic, self.get_map, QoSProfile(depth=10, durability=QoSDurabilityPolicy.TRANSIENT_LOCAL))

        # subscribe to the lidar scan from the robot
        self.create_subscription(
            LaserScan, self.scan_topic, self.robot_scan_received, 10)
```

2. Once the laser scan is received, 'robot_scan_received()' method will be called, where it is checked if all initialisations are completed in the constructor. Once it is satisfied, the laser scan data is transformed with respect to the base frame.

```python

    def robot_scan_received(self, data):
        print("robot_scan_received start")
        # wait until initialization is complete
        if not (self.initialized):
            print("not initialized")
            return

        # we need to be able to transfrom the laser frame to the base frame

        if not (tf2_ros.Buffer.can_transform(self.base_frame, data.header.frame_id, data.header.stamp)):
            return

        # wait for a little bit for the transform to become avaliable (in case the scan arrives
        # a little bit before the odom to base_footprint transform was updated)

        tf2_ros.Buffer.can_transform(
            self.base_frame, self.odom_frame, data.header.stamp, Duration(seconds=0.5))

        if not (tf2_ros.Buffer.can_transform(self.base_frame, data.header.frame_id, data.header.stamp)):
            return
```

3. In the same function, the 'odom_pose' is set to the current pose of the robot, measured in the 'odom' frame of reference. It is stored as a previous instance of pose of data to decide when to update the particles.

   When the robot satisfies the conditions based on net motion, the methods of 'update_particles_with_motion_model', 'update_particle_weights_with_measurement_model', 'normalize_particles', 'resample_particles', 'update_estimated_robot_pose', 'publish_particle_cloud' and 'publish_estimated_robot_pose' are called in sequence. Individual functions will be explained in the following points.

```
        # determine where the robot thinks it is based on its odometry
        p = PoseStamped(
            header=Header(stamp=data.header.stamp,
                          frame_id=self.base_frame),
            pose=Pose())

        self.odom_pose = self.tf_buffer.transform(p, self.odom_frame)
```

```python

        # we need to be able to compare the current odom pose to the prior odom pose
        # if there isn't a prior odom pose, set the odom_pose variable to the current pose
        if not self.odom_pose_last_motion_update:
            self.odom_pose_last_motion_update = self.odom_pose
            return

        if self.particle_cloud:

            # check to see if we've moved far enough to perform an update

            curr_x = self.odom_pose.pose.position.x
            old_x = self.odom_pose_last_motion_update.pose.position.x
            curr_y = self.odom_pose.pose.position.y
            old_y = self.odom_pose_last_motion_update.pose.position.y
            curr_yaw = get_yaw_from_pose(self.odom_pose.pose)
            old_yaw = get_yaw_from_pose(self.odom_pose_last_motion_update.pose)

            if (np.abs(curr_x - old_x) > self.lin_mvmt_threshold or
                np.abs(curr_y - old_y) > self.lin_mvmt_threshold or
                    np.abs(curr_yaw - old_yaw) > self.ang_mvmt_threshold):

                # This is where the main logic of the particle filter is carried out

                self.update_particles_with_motion_model()
                self.update_particle_weights_with_measurement_model(data)
                self.normalize_particles()
                self.resample_particles()
                self.update_estimated_robot_pose()
                self.publish_particle_cloud()
                self.publish_estimated_robot_pose()
                self.odom_pose_last_motion_update = self.
    odom_pose
```

4. Movement model: Update each particle's posture in accordance with the robot's movements.

   'update_particles_with_motion_model' is used to measure the movement of the robot with respect to the previous odometry pose. Based on these measurements, the particle poses are updated along with a small amount of noise.

```
def update_particles_with_motion_model(self):
    print("update_particles_with_motion_model start")
    # based on the how the robot has moved (calculated from
    its odometry), we'll  move
    # all of the particles correspondingly
    xpos_delta = self.odom_pose.pose.position.x - \
        self.odom_pose_last_motion_update.pose.position.x
    ypos_delta = self.odom_pose.pose.position.y - \
        self.odom_pose_last_motion_update.pose.position.y
    magnitude = math.sqrt(xpos_delta**2 + ypos_delta ** 2)
    angle_delta = get_yaw_from_pose(
        self.odom_pose.pose) - get_yaw_from_pose(self.
    odom_pose_last_motion_update.pose)
    for particle in self.particle_cloud:
        # generate some noise for particle movement
        movement_noise = np.random.normal(0, 0.07, 2)
        # noise for particle direction, approx 4 degree
    standard deviation
        angle_noise = np.random.normal(0, 0.068, 1)
        particle_dir = get_yaw_from_pose(particle.pose)
        # We estimate that the difference between the motion
    vector of the particle and the orientation
        # is angle_delta/2, and use this to transform the
    given motion vector to the coordinates of the particle
        particle.pose.position.x += magnitude * \
            math.cos(particle_dir - angle_delta/2) + movement_noise[0]
        particle.pose.position.y += magnitude * \
            math.sin(particle_dir - angle_delta/2) + movement_noise[1]
        # update angle of particle based on turn
        new_euler_angle = particle_dir + angle_delta + angle_noise[0]
        new_qtrn_arr = quaternion_from_euler(0, 0, new_euler_angle)
        particle.pose.orientation = Quaternion(
            new_qtrn_arr[0], new_qtrn_arr[1], new_qtrn_arr[2], new_qtrn_arr[3])
```

5. Measurement model: Based on how near the robot sensor readings match what they should be, assign a weight to each particle.

   ‘update_particle_weights_with_measurement_model’ updates the weights assigned with individual particles which are inversely proportional to the sum of distances of deviation of laser scan data from the occupied cells of the map in the ‘directions_to_check’ directions.

```
def update_particle_weights_with_measurement_model(self, data):
    print("update_particle_weights_with_measurement_model start")
    for particle in self.particle_cloud:
        difference_sum = 0
        # for every angle to check, calculate difference between lidar values and estimated value
        for angle in self.directions_to_check:
            difference_sum += abs(min(data.ranges[int(angle*180/math.pi)], data.range_max) - min(
                self.estimate_particle_lidar(particle, angle), data.range_max))
        particle.w = 1/max(difference_sum, 0.01)
```

   ‘normalize_particles’ is used to normalize particle weights such that the sum of the weights is equal to 1. This will represent the prior probability of the robot’s pose is equal to the particle pose.

```python
def normalize_particles(self):
    print("normalize_particles start")
    total_weight = 0
    # get sum of weights
    for particle in self.particle_cloud:
        total_weight += particle.w
    # divide by sum
    for particle in self.particle_cloud:
        particle.w = particle.w/total_weight
```

6. Re-sampling: Re-sample all the particles based on the probabilities we obtain after normalizing the particles by weight.

   'resample_particles' is used to sample particles from the 'particle_cloud' based on the weights assigned to them and the number of required particles.

```python

def resample_particles(self):
    print("resample_particles start")
    self.particle_cloud = draw_random_sample(
        self.particle_cloud, [p.w for p in self.
particle_cloud], self.num_particles)
```

7. Updating estimated robot pose: According to the average pose of all the particles, update the estimated robot pose.

   'update_estimated_robot_pose' is used to update the estimated robot pose as the average pose from all particles. Further, these particles and robot poses are published which can be visualised in rviz.

```python

def update_estimated_robot_pose(self):
    print("update_estimated_robot_pose start")
    # initialize variables for summing average particle
positions/orientations
    x = y = z = ox = oy = oz = ow = 0
    n = len(self.particle_cloud)
    # loop through all particles to sum up position and
orientation values
    for particle in self.particle_cloud:
        p = particle.pose

        x += p.position.x
        y += p.position.y
        z += p.position.z

        ox += p.orientation.x
        oy += p.orientation.y
        oz += p.orientation.z
        ow += p.orientation.w

    # calculate average positions/orientations and update estimated position
    self.robot_estimate.position.x = x/n
    self.robot_estimate.position.y = y/n
    self.robot_estimate.position.z = z/n

    self.robot_estimate.orientation.x = ox/n
    self.robot_estimate.orientation.y = oy/n
    self.robot_estimate.orientation.z = oz/n
    self.robot_estimate.orientation.w = ow/n

    return
```

4 Future Work and Conclusion

- The particle filter is not functional in ROS2 yet as the transformation from one frame to another is not yet supported in python. An alternative to this will be implemented.
- The normals stored in the occupancy grid map will be used to estimate robot pose in particle filter implementation.
- Recreating particle degeneracy problem and testing the inclusion of the knowledge of normals to surface for localisation.

References

[1] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[2] G. Grisetti, C. Stachniss, and W. Burgard, "Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling," in *Proceedings of the 2005 IEEE international conference on robotics and automation*, pp. 2432–2437, IEEE, 2005.

[3] J. Elfring, E. Torta, and R. van de Molengraft, "Particle filters: A hands-on tutorial," *Sensors*, vol. 21, no. 2, p. 438, 2021.