PARTICLE FILTER LOCALIZATION REPORT

Introduction:

In this assignment, we aimed to develop a robot localization system using a particle filter approach. We successfully initialized the particle filter with randomly distributed particles representing potential robot poses. Then particles were moved according to control inputs, with added noise to simulate real-world uncertainties. We integrated sensor data, including LiDAR, terrain and compass measurements, to update particle weights based on their alignment with observed data.

Reflection:

I faced issues in task 6 first. We needed to incorporate the weights with weighted average instead of highest weight of the particle. Also, we have to try to include the weights in the formula when calculating the weighted average. Then I faced issue with task 8 to incorporate the compass measurements. First we tired to replace the compass reading in place of previous particle values and got bad results as the predicted location was off most of the time. I have discussed the method adopted later in the task 8.

Video Link:

https://drive.google.com/file/d/1J7G0GaV5eLiS0RwVE2x__qfFu4BpCUnN/view?usp=drive_link

Task 1:

For each 500 particles created under *self.num_particles_,* we generate random position x, y and orientation (θ) through the *for* loop within the map. All the particles are given equal weight of 1. Now x, y, theta and weight are passed to the particle by calling the particle class and appending it to the *self.particles_ v*ariable.

```
228         ## TASK 1         ##
229         ####################
230         for i in range(self.num_particles_):
231             x = random_uniform(self.map_x_min_, self.map_x_max_)
232             y = random_uniform(self.map_y_min_, self.map_y_max_)
233             theta = random_uniform(0, 2 * math.pi)
234             weight = 1.0
235             particle = Particle(x, y, theta, weight)
236             self.particles_.append(particle)
```

Task 2:

loop iterates over each particle p in the *self.particles_* list and the weight of each particle to the total weight variable. The second loop normalizes the weight across all the particles making the weight same for all the particles. We divide the total weight with the weight of each particle.

```
288         ## Task 2         ##
289         ####################
290         total_weight = 0
291         for p in self.particles_:
292             total_weight += p.weight
293         for p in self.particles_:
294             p.weight /= total_weight
```

Task 3:

Whenever the human operator clicks “publish point” we retrieve the x and y coordinates from the *clicked_point_msg which stores them*. geometry_msgs/PointStamped message contains 2 fields, a header and a pointer. Pointer contains the coordinates of the clicked point.

In the loop we go through each particle to find the coordinates x and y which are sampled from gaussian distribution with standard deviation *self.clicked_point_std_dev_* centered around the clicked point. Orientation θ is sampled from a uniform distribution between 0 and 2π. Each particle is made sure they have equal weights distributed across them and added to *self.particles_*.

```
        ## Task 3         ##
        ####################

        for p in range(self.num_particles_):
            x = np.random.normal(clicked_point_msg.point.x, self.clicked_point_std_dev_)
            y = np.random.normal(clicked_point_msg.point.y, self.clicked_point_std_dev_)
            theta = np.random.uniform(0, 2 * np.pi)
            particle = Particle(x, y, theta, weight=1.0 / self.num_particles_)

            self.particles_.append(particle)

```

Task 4:

The distance moved and rotation turned are given in the variable distance and rotation. We generate variables *distance_noise* and *rotation_noise* using a Gaussian (normal) distribution with a standard deviation specified by self.motion_distance_noise_stddev_ and *self.motion_rotation_noise_stddev_*, respectively. The *random_normal* function provided already generates numbers with a mean of 0 so I didn't specify the 0 again when calling it.

$$x_{t+1} = x_t + (d + \omega_d) \times cos(\theta_t)$$
$$y_{t+1} = y_t + (d + \omega_d) \times sin(\theta_t)$$
$$\theta_{t+1} = \theta_t + \Delta\theta + \omega_\theta$$

In the loop here we go through each particle and update its position and orientation based on the equations provided. *wrap_angle* ensures the angle stays within the range of 0 to 2 π.

```
        ## Task 4         ##
        ####################
        for p in self.particles_:
            # Add noise to distance and rotation
            distance_noise = random_normal(self.motion_distance_noise_stddev_)
            rotation_noise = random_normal(self.motion_rotation_noise_stddev_)

            # Update particle pose
            p.x += (distance + distance_noise) * math.cos(p.theta)
            p.y += (distance + distance_noise) * math.sin(p.theta)
            p.theta = wrap_angle(p.theta + rotation + rotation_noise)

```

Task 5:

This code adjusts the weight of each particle based on how consistent its location's terrain class is with the observed terrain class. We update the particles weight by the formula given,

$$p.\,weight = P(z = terrain\ class\ of\ observation \mid x = terrain\ class\ of\ particle) \times p.\,weight$$

terrain_msg.data variable is the observed terrain class and self.visual_terrain_map_.get_ground_truth(p.x, p.y) method retrieves the terrain class at the particle's location. We calculate the likelihood of the observed class given the terrain class at the particle's location using the confusion matrix given. We update the weights of the particle by multiplying its current weight by likelihood. Particles will increase their probability if the terrain class observation and particles terrain location is same and decrease if they are not same.

```
636             ## Task 5          ##
637             ####################
638             # likelihood = ??
639             # p.weight = ??
640             particle_class = self.visual_terrain_map_.get_ground_truth(p.x, p.y)
641
642             likelihood = self.visual_terrain_map_.confusion_matrix[observed_class, particle_class]
643
644             p.weight *= likelihood
```

Task 6:

In this task we combine the particles into a single best estimate for the robot's location using a weighted average. For the x and y coordinates we multiply the particles coordinates with the particle weights. For angle, account for the wrap-around nature of angles by converting the angles to Cartesian coordinates, averaging them, and converting back to an angle and stored in SinSum and CosSum.

```
354         ## Task 6          ##
355         ####################
356         sin_sum = 0
357         cos_sum = 0
358
359         for p in (self.particles_):
360             estimated_pose_x += p.x * p.weight
361             estimated_pose_y += p.y * p.weight
362             sin_sum += math.sin(p.theta) * p.weight
363             cos_sum += math.cos(p.theta) * p.weight
364         estimated_pose_theta += math.atan2(sin_sum, cos_sum)
365
366         # Set the estimated pose message
367         self.estimated_pose_.position.x = estimated_pose_x
368         self.estimated_pose_.position.y = estimated_pose_y
```

Task 7:

In this task we introduce laser scan readings. Similar to task 5 we need to update the particles weight by multiplying the particle weight to the likelihood but here instead of confusion matrix we use the below model,

$$\prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma_z^2}} e^{-\left\{\frac{(\hat{z}-z_i)^2}{2\sigma_z^2}\right\}}$$

We split the exponential term and normalizing term into 2 different variables for easier calculation.

```
            ## Task 7          ##
            ####################
            # likelihood = ??
            exponential_term = math.exp(- (particle_range - scan_range) ** 2 / (2 * self.sensing_noise_stddev_ ** 2))

            normalization_factor = 1 / math.sqrt(2 * math.pi * self.sensing_noise_stddev_ ** 2)

            likelihood *= exponential_term * normalization_factor

        # Update the particle weight with the likelihood
        p.weight *= likelihood
```

Task 8 & Challenges:

We tried to incorporate the compass measurements into particle filter. First I tried to replace the current orientation theta of the particles with the compass reading and ended up with wrong localization most of the time. Every time I need to correct the position by publish point through Rviz. Yet the localization isn't great.

Then I tried to incorporate weight adjustment based on the difference between the current orientation theta and the compass reading. This difference (diff) indicates how far the particle's orientation is from the compass reading. A positive diff means the particle's orientation is greater than the compass measurement and reduces the particle's orientation, and a negative diff means it is less and increase the orientation. This gave better results compared to the last method. The identified location is closer to the actual robots' position most of the time.

```
        ## Task 8          ##
        ####################
        for p in self.particles_:
            diff = p.theta - self.compass_

            if diff >= 0:
                p.theta -= diff * (self.magnetometer_noise_stddev_ / (self.magnetometer_noise_stddev_ + diff))
            else:
                p.theta -= diff * (self.magnetometer_noise_stddev_ / (self.magnetometer_noise_stddev_ - diff))
```

Conclusion:

Overall, we achieved good results on localizing the actual robots' position with the help of three sensors. In task 8 what I tried to achieve is a stripped-down version of Kalman filter I believe. If I had more time then I would like to delve into the Kalman filter in detail and include more elements to the scaling factor to make full use of the compass measurements.