49274 Advanced Robotics (Spring 2023)

Assignment 1: Particle filter localisation

Report

- **Introduction**

  Particle filter is a method that is used to localise a robot in a given environment. It uses random samples to collectively represent a probabilistic estimate of the robot pose. In this assignment, the particle filter approach was focused step by step for 6 tasks. Every task was implemented and completed, and the entire simulation was fully operational. This report delves into the details of these tasks, highlighting the strategies, methodologies, and outcomes achieved during the pursuit of enhancing robot localisation through the particle filter method.

- **Development methodology**

  The tasks were to implement initialising the particles, normalising the weights, motion updates, observation updates, pose estimation and magnetometer data fusion. These tasks were performed in the order and tested after each completion. Tasks 2-5 worked at the first time. In the Task 1, particle initialising was implemented at the first time, but it was quite challenging to remove particles from the occupied space. This happened due to the mapping between the distances and the pixel values in the map image and trial and error method was used to solve the issue. Magnetometer data fusion task was also quite challenging since it needed a method to fuse data comparing the existing orientation. According to the reliability of the data, an average method was used, and trial and error method was used to tune that. Overall, it was a good learning-based task sequence for developing particle filtering techniques.

- **Video**

  The video includes both the localisations with the compass and without the compass. The video link and the timelines are as follows.

  - Video link - https://drive.google.com/file/d/1e6-f5n3hSqnqEl04KLNIeBNL0iAIl0UO/view?usp=drivesdk
  - Timeline - Localisation without the compass: 00.00 – 00.30
    Localisation with the compass: 00.31 – 00.56

- **Tasks**

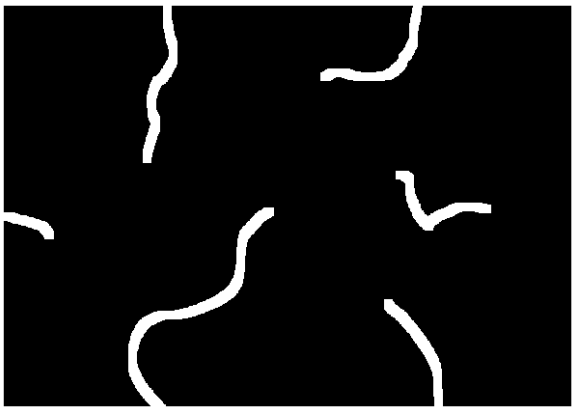
  1. Initialise the particles

     First, particles were randomly generated with x, y, theta random values and each weight of 1.0. There were 500 particles. Some particles were generated on or near the walls. Therefore, they were removed with using an image processing method. The map image was converted to a black and white image (binary image) by thresholding. Then, the binary image was dilated with a kernel since the robot should not be very close to the walls. After that, x and y values of each particle were mapped to the pixel values and removed the particles that were in wall regions in the dilated image.

```
    # initialise random particles
    for particle_ in range(self.num_particles_):
        x = random_uniform(self.map_x_min_, self.map_x_max_)
        y = random_uniform(self.map_y_min_, self.map_y_max_)
        theta = random_uniform(0, 2*math.pi)
        weight = 1.0
        p = Particle(x, y, theta, weight)    # create a struct for the particle
        self.particles_.append(p)


    # Particles may be initialised in occupied space but the map has thin walls so it should be OK
    # TODO inflate the occupancy grid and check that particles are not in occupied space

    # dilation for inflate the occupancy grid
    kernel = np.ones((7,7), np.uint8)   # kernel for dilation
    # thresholding before the dilation since occupancy grid has more the two colors
    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

    dilated_map_image = cv.dilate(binary_image, kernel, iterations=1)   # dilate

    # removing particles in the occupied space
    valid_particles = []
    for p in self.particles_:
        # get the x and y pixel values for x and y distance vakues
        x_pixel = int((self.map_.info.width-1)*(p.x-self.map_x_max_)/(self.map_x_max_-self.map_x_min_))
        y_pixel = int((self.map_.info.height-1)*(self.map_y_max_-p.y)/(self.map_y_max_-self.map_y_min_))

        # check whether the particle in the occupied space
        if dilated_map_image[y_pixel][x_pixel] != 255:
            valid_particles.append(p)

    self.particles_ = valid_particles
```

Code snippet of the particle initialisation

Binary image of map2



Dilated image of map2

2. Normalise the weights

   To normalise the weights, all the weights of the particles were added together and then, weight of each particle was divided by the sum of the weights.

```
    # normalize the weights
    # get the sum of weights
    sum_of_weights = 0
    for p in self.particles_:
        sum_of_weights = sum_of_weights + p.weight

    # normalizing
    for p in self.particles_:
        p.weight = p.weight / sum_of_weights
```

Code snippet for normalising the weights

3. Motion update

When the robot is moving, the amount of the distance or rotation it made was considered to the particles. Equations for updating were given and implemented in the code. Original standard deviation values were used for the calculations. The theta angle was wrapped into 0-2π range using wrap_angle function.

```
# update particle values with motion updates
for p in self.particles_:
    distance_noise = random_normal(self.motion_distance_noise_stddev_)
    rotation_noise = random_normal(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 + rotation + rotation_noise
    p.theta = wrap_angle(theta_)
```

Code snippet for motion updates

4. Observation update

Particles were updated when scan data was received. Likelihood for each particle was compared with each ray of the sensor data using a statistical method. Correctly estimated particles had a higher likelihood, and the product combines these likelihoods for the set of rays into a single likelihood for each particle. The equation for calculating likelihood was given and implemented in the code. After calculating likelihood for each particle, those were multiplied with their weights. Original sensing noise standard deviation value was used.

```
# calculate likelihood values
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)))
likelihood *= ray_likelihood
```

Code snippet for observation update

5. Pose estimate

In this task, the robot current position in the arena was estimated using the x, y, theta values of particles. To estimate the x and y values of the robot, the average values of x and y values of all particles were calculated by taking the sum of all and dividing by the number of current samples. Average value of theta cannot be used as the estimated theta for robot's pose due to the wrap around, so a weighted average method was used. Angles were needed to be convert into cartesian coordinates. Equation of the method was given and implemented in the code.

```
# weighted average pose
x = 0
y = 0
theta_sin = 0
theta_cos = 0
for p in self.particles_:
    x += p.x
    y += p.y
    theta_sin += math.sin(p.theta)
    theta_cos += math.cos(p.theta)

n = len(self.particles_)
estimated_pose_x = x/n
estimated_pose_y = y/n
estimated_pose_theta = math.atan2(theta_sin/n, theta_cos/n)
```

Code snippet for pose estimate

6. Magnetometer fusion

   To fuse the magnetometer data with current orientation, a complementary filter method was used. Weights were given to both magnetometer data and current orientation. The priority was given to the magnetometer since it gives accurate orientation value.

```
# using complemetary method for fuse magnetometer data with existing orientation
if self.motion_update_count_ >= self.num_motion_updates_:
    alpha = 0.8   # weight
    for p in self.particles_:
        # if the difference between two orientation values is greater than pi, then the calculation should be done for the reflex angle
        if abs(p.theta-self.compass_) < math.pi:
            p.theta = alpha*self.compass_ + (1-alpha)*p.theta
        else:
            p.theta = wrap_angle(alpha*self.compass_ + (1-alpha)*p.theta + math.pi)
```

Code snippet for magnetometer fusion

- **Magnetometer data fusion**

The magnetometer gave global orientation values with some noise. The estimated orientation without the magnetometer was based on odometry and scan data. Those are compatible for distances but not much for orientations without knowing the initial orientation. Since the magnetometer gave global orientation values, they were more accurate than the estimated orientations. But since there were noise, they could not be directly used as the robot's orientation. Therefore, complementary filter method was used to fuse those data. It was a method that giving weights for data types based on the priority or accuracy. As per the code snippet for the previous magnetometer fusion task section, an alpha value was selected, and fusion was done based on that alpha value. Since, the magnetometer gave the accurate values than the existing orientation, more weight was given for magnetometer data. This reduces the estimation errors due to the drift and noises.

When estimating the robot's pose without magnetometer, robot needed much data and time to localise in the arena. Original launch file parameters were kept unchanged and sometimes, the robot localised well but sometimes failed to localise due to the incorrect orientation. When the magnetometer was used with data fusion, the robot could localise well with estimating the correct orientation also with less data and time. It provided more reliable localisation than the one without the magnetometer.

- **Testing with parameters**

  The localisation without the magnetometer needed much data and time. Therefore, num_motion_updates and num_sensing_updates were kept at 5. The reliability of localisation was reduced when those parameter values were reduced. But when using the magnetometer, the robot could localise well when those parameter values were reduced to 2 since the robot needed less data and time. The robot could localise well in all 3 maps. Other parameters were unchanged during the testing since the robot localised successfully.

- **Improving with the time**

  There were a number of parameters in the source code and the map data. The robot was tested under only a subset of parameter values. If there was more time, the particle filter method could be tested under the remaining parameter conditions. There might be a fine-tuning of other parameters such as num_particles, num_scan_arrays and especially alpha value that used for data fusion. Also, there might be better data fusion methods than the used one. But overall, localisation was accurate and reliable.