49274 Advanced Robotics (Spring 2019)

Assignment 1 Part 2: Particle filter

- Due date: 12/08/2019 (week 4) at 6 pm
- Total marks: 5 (5% of the final mark for the subject)

Introduction

Almost all probabilistic localisation algorithms are based on Bayes filter. In the first part of the assignment we solved a 1-dimensional (discrete) localisation problem using the *discrete* Bayes filter. In reality we want to localise the robot in a 2D or 3D *continuous* space. In these cases we are not able to implement the exact Bayes filter (due to the nonlinearity of the measurement models), therefore we must approximate the Bayes filter.

	State space	**Belief**	**Solution**
Histogram filter	Discrete	Multimodal	Approximate
Extended Kalman filter	Continuous	Unimodal	Approximate
Particle filter	Discrete*	Multimodal	Approximate

The above table lists a number of popular localisation algorithms. For example in the histogram filter we discretise the space, i.e. we approximate the continuous space by a set of discrete cells (similar to the first part of the assignment). In the extended Kalman filter (EKF) we linearise the nonlinear measurement models.

In particle filter (also known as Monte Carlo localisation) we use random samples (particles) to represent our belief about the robot pose. Each particle is a possible robot pose, in 2D this is an X and Y coordinate and an orientation. Each particle also has a weight: the probability that the particle is the pose of the robot.

Particle filter

Just like the Bayes filter, we initialise, then perform a motion updates, observation updates, and normalise. The particle filter has an additional step where we resample the particles, so that we have a higher number of particles around the poses where we have a high probability of the robot being located.

There a number of methods to estimate the pose of the robot, the simplest is to take the pose of the particle with the highest probability. A better method would be to take a weighted average of the poses of all particles, or a weighted average of the poses of high probability particles.

Initialisation

To initialise the particle filter, we create a set number of particles, each with a random pose and the same weight.

Motion update

In the motion update, we move every particle according to the control input. We also add noise to the movement to reflect the fact that the control input has noise.

Observation update

In the observation update we compare the actual sensor data with the expected sensor data of each particle to calculate a measurement probability, and apply the measurement probability to the particle.

Normalisation

Normalisation is similar to the normalisation performed in part 1 of the assignment: we modify the weights of the particles so that the sum of all weights is equal to 1.

Resampling

To resample we create new particles by randomly selecting existing particles, with particles with higher weights having a higher probability of being selected, and then add some noise.

Your task

Your task is to complete the code in 5 methods, each worth 1 mark:

- *initialiseParticles* (from line 252)
- *normaliseWeights* (from line 274)
- *odomCallback* (from line 464)
- *scanCallback* (from line 538)
- *estimatePose* (from line 287)

Obviously the line numbers will change once you start to add your code. The locations where you should insert your code are marked with a comment “YOUR CODE HERE”

Overview of the node

The node starts with the main function on line 571 of the template code. The main function sets up the node, creates an instance of the *ParticleFilter* class, and “spins” to process callbacks.

wrapAngle function

The *wrapAngle* function is defined on line 22. You can use this to wrap an angle between 0 and 2*Pi.

Particle structure

The *Particle* structure is defined on line 42, and is used by the *ParticleFilter* class The data structure has the fields *x*, *y*, *theta* and *weight*.

ParticleFilter class

The *ParticleFilter* class has a number of variables which are declared starting at line 57. The most important is the *particles_* variable, which is a C++ vector of *Particle* structures.

There are a number of parameters you will use in your code:

- *motion_distance_noise_stddev_*: standard deviation of distance noise for motion update
- *motion_rotation_noise_stddev_*: standard deviation of rotation noise for motion update
- *sensing_noise_stddev_*: standard deviation of sensing noise
- *map_x_min_*, *map_x_max_*, *map_y_min_*, and *map_y_max_*: limits of the map

There are also a number of other parameters you can change to change the behaviour of the particle filter:

- *num_particles_*: number of particles to create
- *num_motion_updates_*: number of motion updates before a sensor update
- *num_scan_rays_*: (approximate) number of scan rays to evaluate
- *num_sensing_updates_*: number of sensing updates before resampling

The class methods are declared starting from line 111. You can use the *randomUniform* and *randomNormal* methods to get random numbers.

There are 4 callback methods declared starting from line 124. callbacks are functions/methods that are called by another part of the system. *publishParticles* and *publishEstimatedPose* are both called by a timer. You do not need to modify these methods.

odomCallback is called when wheel odometry is received. You will complete the code to perform the motion update.

scanCallback is called when a laser scan is received. You will complete the code to perform the observation update.

There are two other methods that you will not need to modify. *hitScan* is used in *scanCallback* to determine the expected sensor readings for a particle. *resampleParticles* is called by *scanCallback* after a certain number of observation updates have been performed.

ParticleFilter constructor

When an instance of the *ParticleFilter* class is created the constructor is called. The constructor is defined on line 131.

initialiseParticles method

In *initialiseParticles* you will fill the values of the particles in the *particles_* variable.

- *x* should be a random value with uniform distribution between *map_x_min_* and *map_x_max_*
- *y* should be a random value with uniform distribution between *map_y_min_* and *map_y_max_*
- *theta* should be a random value with uniform distribution between 0 and 2*Pi
- *weight* should initially be equal for all particles

normaliseWeights method

In *normaliseWeights* you will normalise the weight of the particles in the *particles_* vector, so that the sum of all weights is equal to 1.0

odomCallback method

In the *odomCallback* method you will implement the motion update. The distance and rotation of the robot has been calculated for you, and is given in the *distance* and *rotation* variables.

The equations for updating the position of the particles are:

$$x_{t+1} = x_t + (d + \omega_d) \times cos(\theta_t)$$

$$y_{t+1} = y_t + (d + \omega_d) \times sin(\theta_t)$$

$$\theta_{t+1} = \theta_t + \Delta\theta + \omega_\theta$$

where:

- d is the distance
- $\Delta\theta$ is the rotation
- ω_d is distance noise
- ω_θ is rotation noise

Distance and rotation noise can be generated with the *randomNormal* method and the *motion_distance_noise_stddev_* and *motion_rotation_noise_stddev_* variables.

Remember to wrap the angle with the *wrapAngle* method.

scanCallback method

To perform an observation update we compare the range values from the laser scanner with the expected range values of the particle. To reduce computation, instead of comparing every range value in the scan we will only compare a limited number of rays.

To update the weight of the particle we multiply it will the likelihood of each particle. The likelihood of a particle is the product of the likelihood of each ray, which is calculated with the equation:

$$\frac{1}{\sqrt{2\pi\sigma_z^2}} \times exp(-\frac{(\hat{z}_i - z_i)^2}{2\sigma_z^2})$$

where:

- z_i is the range value for the scan ray (variable *scan_range*)
- $\hat{z}_i$ is the expected range value of the particle (variable *particle_range*)
- σ_z is the standard deviation of sensing noise (variable *sensing_noise_stddev_*)

A likelihood variable is already given to you. You need to update this variable with the likelihood of each ray so that the particle weight will be updated.

estimatePose method

In the *estimatePose* method you will set the values of the variables *estimated_pose_x*, *estimated_pose_y*, and *estimated_pose_theta*, using the particles in the *particles_* vector.

The simplest method is to select the pose of the particle with the highest weight, however if you do this you will only receive 0.5 marks (out of a possible 1 mark for completing this method). A weighted average is a better way of estimating the pose however we are not going to tell you how to do it. We will leave it to you to figure out.

Note: if you do implement a weighted average you can't simply take an average of an angle. You need to convert the angle to Cartesian coordinates to determine the average, such as:

$$\bar{\theta} = atan2(\frac{\sum_{i=1}^{n} sin(\theta_i)}{n}, \frac{\sum_{i=1}^{n} cos(\theta_i)}{n})$$

Compiling and running the node

Creating a workspace

First you need to create a workspace. You only need to do this once.

- Open a terminal
- If you are using the UTS computer labs, first run:

  ```
  singularity shell /images/singularity_containers/ros-melodic-ar.sif
  ```

- Run:

  ```
  mkdir -p catkin_ws/src
  cd catkint_ws/src
  catkin_init_workspace
  ```

- Copy the “particle_filter_localisation” folder into “catkin_ws/src”

Compiling the node

- Open a terminal
- If you are using the UTS computer labs, first run:

  ```
  singularity shell /images/singularity_containers/ros-melodic-ar.sif
  ```

- First navigate to the “catkin_ws” folder:

  ```
  cd ~/catkin_ws
  ```

- Compile all packages/nodes in the workspace by running:

  ```
  catkin_make
  ```

Running the node

- Open a terminal
- If you are using the UTS computer labs, first run:

  ```
  singularity shell /images/singularity_containers/ros-melodic-ar.sif
  ```

- To set up your environment (you only need to do this when you open a new terminal) run:

  ```
  source ~/catkin_ws/devel/setup.bash
  ```

- Run the particle filter localisation node and other required nodes with (the command is one line):

```
    roslaunch particle_filter_localisation
particle_filter_localisation.launch
```

- Control the robot with the keyboard by opening another terminal and running:

```
    rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

- Or if you are using the UTS computer labs run (**the command is one line**):

```
    singularity exec /images/singularity_containers/ros-melodic-ar.sif
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

- Use *Ctrl+C* to exit a running program in the terminal
- The launch file starts a "roscore", which is terminated when you kill quit the launch file. You will need to restart "teleop_twist_keyboard" whenever you restart "particle_fitler_localisaiton.launch"
- Avoid this by starting a "roscore" before "particle_fitler_localisaiton.launch":

```
    roscore
```

- On the FEIT computers (**the command is one line**):

```
    singularity exec /images/singularity_containers/ros-melodic-ar.sif
roscore
```

Changing the parameters

There are a number of parameters you can change to experiment with improving the particle filter.

Starting position

- You can change the starting position of the robot by changing the pose value on line 91 of "config/map.world"
- The values are x, y, and z position (z should always be 0) and orientation in degrees.

Launch file parameters

- A number of parameters can be modified by uncommenting lines in the launch file ("launch/particle_filter_localisation.launch")

Modifying the code

- If you're feeling ambitious, you can try modifying the rest of the code to improve localisation.