49274: Advanced Robotics **Spring 2018**

Assignment 1 - Part 2: Particle Filter Localisation

Total Marks: 8 marks

University of Technology Sydney (CAS) Due date: **week 4 (16 Aug. 2018)**

Introduction

Almost all of the probabilistic Localisation algorithms are based on the Bayes filter. In Tutorial 1-1 we solved the localisation problem in a 1-dimensional (discrete) environment using the *discrete* Bayes filter. In reality we need to localise our robot in a 2D or 3D continuous space. Unfortunately in these cases we are *not* able to implement the exact Bayes filter (e.g., due to the non-linearity of our measurement functions). Therefore we have to *approximate* the Bayes filter.

Table 1: Properties of localisation algorithms

	State Space	**Belief**	**Final Solution**
`Histogram Filter`	discrete	multi-modal	approximate
`Extended Kalman Filter`	continuous	uni-modal	approximate
`Particle Filter`	discrete*	multi-modal	approximate

In Table 1 we have listed a number of popular localisation algorithms. For example in Histogram filter we discretize the space; i.e., we approximate the continuous space by a set of discrete "cells" (similar to what you did in Tutorial 1-1). In Extended Kalman filter (EKF) we linearise the non-linear models; i.e., we approximate the non-linear models with linear models.

In **Particle filter localisation** (also known as Monte Carlo Localization, or MCL) we use *random samples* (also called *particles*) to represent our *belief* about the robot pose (i.e., position and orientation). Each particle is a **potential** robot pose (a guess); e.g., in 2D environments, each particle has three fields: x, y (position) and θ (orientation).

In this assignment you implement 2D **MCL**, the most successful localization algorithm!

Particle Filter

Each particle has three fields (x, y and θ) and a weight (a positive number between 0 and 1). Weight of each particle describes how *consistent* that particle is with the data obtained

from robot sensors and our map. You should normalise the weights (i.e., sum of the weights must be equal to 1) at the end of each time step.

As it was mentioned earlier, our particles describe our belief. Therefore at each step we **update** our previous *belief* using the new control input and observation. Similar to the discrete Bayes filter we do this in two steps:

1. **Prediction:** predict the new belief only using the new control input.
2. **Update:** use the predicted belief and the new observation to obtain the final belief.

In the prediction step we update the position of each particle based on the given control input. In the update step we update the weight of each particle based on the new observation (and our map).

For this assignment a template code is given to implement a particle filter. You will need to replace the missing sections of the code.

Important:

- It is good to first understand what the code is actually doing before implementing anything. Also try to work step by step and test before moving forward.

Compiling and Running in ROS

Compilation

Download zip package, extract the file into

`/home/vmuser/catkin_ws/src/`

then execute the following command sequence:

- cd ~/catkin_ws
- catkin_make
- source devel/setup.bash

How to run?

The package can be readily run using the launch file.

- source devel/setup.bash
- chmod +x src/pf_localization/src/scripts/map_frame.py
- roslaunch pf_localization pf_localization.launch

Note: You can also run the package step-by-step by using the following command sequence. The commands need to be run in different terminal windows.

- `roscore`
- `rosrun map_server map_server config/map.yaml`
- `rosrun stage_ros stageros config/map.world`
- `rosrun rviz rviz` and load **pf_conf.rviz** file
- `rosrun pf_localization pf_localization`
- `rosrun teleop_twist_keyboard teleop_twist_keyboard.py`

Data Structure in Brief

Here we introduce the definition of the important variables in the **pf_localization** package. All the following definition is also available in `include/pf_localization.h` header file. Even though you are not supposed to change it, you can use it as a manual if you forget about the definition of the function or the variables.

Variables

- **Particles**
  - `n_particles_`: number of particles, in our set-up, the number is set to be 1000
  - `particles_`: a vector(array) of `Particle` and each `Particle` contains 4 attributes: x, y, o(orientation) and *weight*
  - `dist_noise_` and `ori_noise_` denote the standard deviation value σ_d and σ_θ when you add noise to particles' motion model. These two values are set as 0.01 and $\pi/60$ and will be used in function `motion`
- **Robot Pose**

  - `prev_pos_` and `current_pos_` are used to compute the odometry of the robot `robot_odom_`. You don't have to do with these 3 variables
  - `esti_pos_` is the estimated robot pose given the `particles_` using weighted average. The value is computed in function `calc_estimate`
  - `step_count_` is to count the step numbers, you need it to set the frequency of re-sampling using the given function `resampling`;

- **Laser Scan**
  - `scan_data_` is the vector contains 5 beams of laser scan data

Functions

PFLocalization::init_particles()

Firstly you need to initialise the particle locations and weights. Use the structure `Particle` and add these to the particle array `particles_`. You should uniformly distribute your particles position in accordance to the map. Initial weights should be equal, and they should add up to **1**.

Hint:

- Use the `PFLocalization::uniform_sampling(...)` function for generating uniform random numbers between two values.
- x and y should be in metres, and θ in radians.

PFLocalization::motion(...)

Now you need to write the motion model and update the position of each particle (x, y and θ). We assume the robot is using a bicycle model and can only move forward and turn in its own coordinate frame. In the equation below, d is the distance traveled, $\Delta\theta$ is the change in orientation, w_d is the distance noise and w_θ is orientation noise. We assume the noise variables are distributed according to a Gaussian (normal) distribution with **zero** mean, and variance σ_d^2 and σ_θ^2, respectively.

$$\begin{aligned} x_{t+1} &= x_t + (d + w_d)\cos(\theta_t) \\ y_{t+1} &= y_t + (d + w_d)\sin(\theta_t) \\ \theta_{t+1} &= \theta_t + (\Delta\theta + w_\theta) \end{aligned} \tag{1}$$

Hint:

- Generate different noise samples for each particle, and then compute the new position for each particle using the given control input (i.e., d and $\Delta\theta$), generated noise samples and the previous position of that particle.
- Use `PFLocalization::gaussian_sampling(...)` function to sample from a Gaussian distribution.
- When adding motion to current particles, the orientation needs to be in the range of $[0, 2\pi]$. Please use `fmod` to wrap the angle within the correct range.

PFLocalization::sense(...)

To calculate the measurement probability, we want to know how likely is a measurement to have occurred given that we know some evidence about the measurement. In the particle case, for each particle we wish to calculates its likelihood compared to the actual sensor reading. The steps below show how this can be done:

- Since our sensor is a laser scanner, there will be multiple laser beams with distance (range) measurements at different bearings. Limiting our laser down to more tolerable beam numbers should be done first.
- For each particle we ray cast virtual laser beams to the map to simulate the actual beams. It is important to consider both the particles location and the orientation.
- The virtual beam range are then compared against the real beam range to give us the beams likelihood, i.e., how consistent our prediction (based on the location of that particle) is with the read measurement.
- Finally, by multiplying together all the likelihoods of the beams we can obtain the overall likelihood for one particle.

It is important to understand that the laser sensor itself has noise associated with it. We assume the observation noise is also Gaussian. Then the measurement likelihood for each particle and the i^{th} laser beam can be computed according to:

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

where z_i is the real measurement, $\hat{z}_i$ is the *predicted* measurement based on the location of that particle and our map, and σ_z^2 is the variance of our measurement noise.

The method for ray tracing beams and calculating range error $(\hat{z}_i - z_i)$ is given to you. Now you need to complete the implementation of `PFLocalization::sense(...)` function to return the measurement probability.

Hint:

- The measurement probability is the *product* of all the individual likelihoods.

- σ_z is the standard deviation, you may play around with this value but a recommended value would be around 0.5.
- Optional: The Gaussian function can also be simplified.

PFLocalization::process()

In `process` function, you need to add another code block to update the belief using Bayes rule. This can be done by updating each particle's weight according to:

$$\text{for each particle:} \quad \text{new weight} = \frac{\text{measurement probability} \times \text{old weight}}{\text{normaliser constant}} \tag{3}$$

Hint:

- Remember the normaliser value makes the sum of all new weights equal to 1

PFLocalization::calc_estimate()

In addition to our belief, in many applications we also need to estimate the robot position. There are many ways to calculate an estimate from our belief. We will leave this part up to you. Write your code in `PFLocalization::calc_estimate()`

Hint:

- You can try weighted average of particles, the particle with largest weight, or the weighted average of particles with large weights.

Remark: The robot orientation θ, unlike its x and y coordinates, is a circular quantity. Think of the average of 0 and 2π, it is $(0 + 2\pi)/2 = \pi$! which is clearly wrong. In order to calculate the correct average, we need to map the angles on a unit circle, i.e. transforming from polar coordinates to Cartesian coordinates. Given the angles $\theta_1, \theta_2, ..., \theta_n$, the average is given by

$$\bar{\theta} = \text{atan2}(\frac{\sum_{i=1}^{n} \sin(\theta_i)}{n}, \frac{\sum_{i=1}^{n} \cos(\theta_i)}{n}) \tag{4}$$

PFLocalization::resampling()

Congratulations, you are almost finished. Now you can comment in the resampling function `PFLocalization::resampling()`.

Once we have obtained the new probability distribution of particles, we want to be able to keep the particles which have high probability apposed the lower ones. However we shouldn't remove all the low weighted particles completely since they may still have a chance of having

higher probability in the future. So a good resampling function should base itself on the probability of particle surviving being proportional to the particles weight. The re-sampling algorithm will need to randomly draw `n_particles_` new particles from the particle set, with replacement, in proportion to their weight.

There are many techniques for resampling available. Fortunately, one has been provided for you. For more information on this technique see `http://www.mrpt.org/Resampling_Schemes`.

Re-sampling too often can often lead to undesired convergence of particles before enough measurements have been made. A simple but naive way to resample is to run resampling every n update steps.

Testing

Now that your program is up and running, it will be good to show if each step is working as it should. For this you can use the line `cin.get()`. This will pause your code at any given point and only start once the ENTER key is pressed. This will be important during marking so make sure you know how to turn on and off parts of your code.

To get full marks, you need to show that the algorithm can run robustly on ROS. To do so try fiddling with noise parameters, particle sizes and beam numbers. Remember you will be limited to how quickly the computer can process your algorithm, so try not to go overboard.