Localisation assignment 2

Towards Accurate Localisation with a Particle Filter

The code for this assignment can be found at:

https://gitlab.tue.nl/mobile-robot-control/mrc_localization_students

In the previous assignment you have developed an algorithm commonly described as a dead-reckoning approach. The main strategy was to process the odometry information you received to update the best estimate of the current robot pose. In the assignment you showed that, under certain conditions, this is indeed possible. However, the method breaks down once the assumption on perfect odometry is dropped. Wheel slip and noisy data lead to an ever increasing difference between the actual and estimated pose.

Within the lecture and this assignment we continue on our path towards robot localisation. The main insight from the lecture was that only given the imperfect information provided by the odometry sensor we do not have enough information to infer our current location and orientation. By including information from multiple sources, i.e. sensor fusion, we can get a better estimate than by using the information of a single sensor.

Within this assignment we will explore the inner workings of the particle filter. You will develop the core functionality of the particle filter framework we have developed for you. After completing the assignment you will not only understand the core concepts, but also be able to locate your robot within a known occupancy grid map of the environment in which you deploy your robot.

Before you start the assignment, please note the following:

> ⚠ **Note**
>
> Make sure you are running the *mrc-sim* when doing the assignments

> ⚠ **Note**
>
> Throughout this assignment we will use tests to make sure that your intermediate results are implemented correctly. A correct result indicates that your implementation is likely correct, but does not guarantee it. There is a possibility that you've introduced unforeseen bugs into your implementation.

Getting Familiar with the Framework

Assignment 0: Explore the code-base

We, the tutors and organizers of this course, understand that the code-base in front of you might seem daunting, or even scary. Don't be! We will use this zero-th assignment to make you comfortable with working with the code.

We do assume, however, that you have completed the C++ tutorials. Furthermore, we assume that you have a basic understanding of the underlying concepts of the particle filter, i.e. if you have followed and understood the lecture you're good to go.

To run the tests we've written for you, such that you can efficiently verify the correct implementation of your methods follow these instructions

- In **VScode**
  1. Bring up your command palette with *ctrl-shift-p*
  2. Select *Cmake: Build Target*
  3. In the menu that pops up select all.
  4. Bring up your command palette with *ctrl-shift-p*
  5. Select *Cmake: Run tests*
  6. The output tab of the terminal window will show the results of your tests
- Using Cmake in terminal
  - 1. Build your project
  - 2. Run the binary (located in ./bin) of the test you want to run (i.e. *./assignment1* for the first assignment)

We've furthermore provided simplified versions of the main file, such that you can test the relevant parts of the software piece by piece. These files can be recognized by **main_ex1.cpp** through ***main_ex3.cpp**. Their respective executables are named **main1** through **main3**. To visualize the working of your code use the demo executables.

The Assignment

Preparation

- [ ] Download the code-base and make sure it is opened correctly in vscode
- [ ] Without changing anything, compile the code, make sure there are no errors.
- [ ] Make sure you can run the tests above, do not worry when all of them return a fail or a segfault.

Exploration

- [ ] Explain in a few concise sentences per item how the code is structured.
  - What is the difference between the *ParticleFilter* and *ParticleFilterBase* classes, and how are they related to each other?
  - How are the *ParticleFilter* and *Particle* class related to eachother.
  - Both the *ParticleFilter* and *Particle* classes implement a propagation method. What is the difference between the methods?

> *Tip*: The comments in the header files are often a great way to help your understanding of what each method implements

Assignment 1: Initialize the Particle Filter

Having obtained a bit of insight into the core working of the code-base, let's start with the implementation of the core functionality of the particle filter. As you know, the particle filter estimates the pose of the robot through a set of weighted particles, each particle represents an hypothesis of the current robot pose. The set of all particles approximates the probability distribution over all possible robot poses.

Within assignment 1 we will implement the methods which construct this set of particles.
As you might have discovered, the *ParticleFilter* classes contain vectors storing all their particles. These vectors are initialized when either one of their constructors are called:

```
ParticleFilterBase::ParticleFilterBase(const World &world, const int &N)

ParticleFilterBase::ParticleFilterBase(const World &world,

const double mean[3],

const double sigma[3],

const int &N)
```

more specifically the particles itself are initialized by calling

```
Particle::Particle(const World &world,
                                  const double &weight,
                  std::default_random_engine *generatorPtr)

Particle::Particle(const World &world,
                                  const double mean[3],
                                  const double sigma[3],
                                  const double &weight,
```

```
                                    std::default_random_engine
*generatorPtr)
```

The Assignment

Implementation

- [ ] What is the difference between the two constructors?
- [ ] Complete both constructors
- [ ] Run the tests to validate that your methods function correctly.

> *Tip*: Do not forget to implement the ParticleFilterBase Constructor

Explanation

- [ ] Explain in a few concise sentences per item
  - What are the advantages/disadvantages of using the first constructor, what are the advantages/disadvantages of the second one?
  - In which cases would we use either of them?

Assignment 2: Calculate the filter prediction

Having initialized the filter, we are interested in extracting the pose prediction from the filter. As stated in the lectures, the filter approximates the probability distribution of the robot pose by a cloud of particles. The filter prediction is then the expected value of this distribution.
Within our code-base, the expected value (or the average pose), is calculated in the following method.

```
Pose ParticleFilterBase::get_average_state();
```

The Assignment

Implementation

- [ ] Complete the get_average_state method
- [ ] Run your code, and examine the output of the method.
- [ ] Run the test to verify your implementation

Explanation

- [ ] Explain in a few concise sentences per item:

- Interpret the resulting filter average. What does it resemble? Is the estimated robot pose correct? Why?
- Imagine a case in which the filter average is inadequate for determining the robot position.

> *Tip*: The averaging of one of the three state-variables may be a non-trivial exercise

The Prediction Step

Assignment 3: Propagation of Particles

The particles in our filter represent hypothesis of our current robot pose. So far we've initialized these particles given some prior knowledge of our robot pose, either uniformly across our map or spread around our initial estimate. However, as you may know, robots are not meant to be stationary objects in our world, most robots tend to move around. In the prediction step we incorporate the sensor information that corresponds to this movement in our filter estimates.

Our odometry information consists of three values, two translational x and y components and a rotational component θ. These values represent the distance driven or angle rotated since the robot was started. And are thus defined with respect to the odometry reference frame. These measurements are corrupted by noise, wheel slip, and other phenomena which were not modeled however. The actual distance and angle driven is thus the sum of the received sensor information and an **unknown** noise component.

To update the poses of our set of particles, we run the following method:

```
ParticleFilterBase::propagateSamples(Pose dPose,

const double offset_angle);

void Particle::propagateSample(const Pose &dPose,
                                                    const double
proc_noise[2],
                                                    const double
&offset_angle);
```

in which dPose, is the distance and angle traveled since the last propagation step, proc_noise is the magnitude of the noise we inject during the propagation, and offset_angle is the current rotation between the odometry frame and the robot frame.

The Assignment

Implementation

- [ ] Complete the propagateSample method
- [ ] Run your code, and examine the output of the method.
- [ ] Run the test to verify your implementation

> *Tip*: In order to perform an accurate propagation, first transform $dPose$ into robot frame, afterwards transform $dPose\_robotFrame$ into the map frame.

Explanation

- [ ] Explain in a few concise sentences per item:
  - Why do we need to inject noise into the propagation when the received odometry infromation already has an $unkown$ noise component.
  - What happens when we stop here, and do not incorporate a correction step?

The Correction Step

Assignment 4: Computation of the likelihood of a Particle

In the previous assignments we have implemented the initialization, estimation and propagation of the particle filter. An observant programmer would however have noticed that we, so far, have not improved over the methods implemented in localisation assignment 1. One could even argue that we have implemented an inferior approach, due to the higher computational complexity and the inclusion of an even larger amount of uncertainty due to the injection noise in the propagation step.

The power of the particle filter approach starts to become apparent once we include multiple types of sensor information. As we have seen, odometry information is a valuable source of localisation information, but as we will see in this assignment the inclusion of visual information, in the form of LRF scan, makes the prediction more reliable over the longer term.

In order to incorporate these LRF measurements (R, Θ) we will assign a weight to each particle given a prediction of the measurement for that particle. Each measurement (r_i, θ_i) is treated independently. In this assignment it is your task to generate the prediction, given the build in methods of the world model, and to compute the likelihood of each measurement given this prediction, and the parameters in the provided config file. To implement the last step consult the following description here, and find the following empty methods in your code:

```
LikelihoodVector ParticleFilterBase::computeLikelihoods(

const measurementList &measurement,

World &world)

Likelihood Particle::computeLikelihood(const measurementList &data,

World &world,

const MeasModelParams &lm)
```

```
double Particle::measurementmodel(const measurement &prediction,
                                                                    const
measurement &data,
                                                                    const
MeasModelParams &lm) const
```

The Assignment

Implementation

- [ ] Complete the measurementmodel method
- [ ] Complete the computelikelihood method
- [ ] Run your code, and examine the output of the method.
- [ ] Run the test to verify your implementation

Explanation

- [ ] Explain in a few concise sentences per item:
  - What does each of the component of the measurement model represent, and why is each necessary.
  - With each particle having $N >> 1$ rays, and each likelihood being $\in [0, 1]$, where could you see an issue given our current implementation of the likelihood computation.

Resampling

Assignment 5: Resampling our Particles

So far we've implemented the main parts of the particle filter. We are able to generate particles, take their average, propagate the samples and compute their likelihoods.

However, as you might have guessed from the section title, a last step is to resample the particles periodically, to quote *Probabilistic Robotics*:

> *"The resampling step has the important function to force particles back to the posterior* $bel(x_t)$*. In fact, an alternative (and usually inferior) version of the particle filter would never resample, but instead would maintain for each particle an importance that is initialized by 1 and updated multiplicatively (...) Such a particle filter algorithm would still approximate the posterior, but many of its particles would end up in regions of low posterior probability. As a result, it would require many more particles; how many depends on the shape of the posterior."*

In other words, if we do not resample, a lot of particles will end up in regions of the environment which are very unlikely to be the accurate robot pose. When we resample, we redraw our samples randomly but make sure that regions with high likelihood are represented heavily in the new particle set, regions with low likelihood are represented less.
Or as *Probabilistic Robotics* puts it:

> *" The resampling step is a probabilistic implementation of the Darwinian idea of survival of the fittest: It refocuses the particle set to regions in state space with high posterior probability. By doing so, it focuses the computational resources of the filter algorithm to regions in the state space where they matter the most"*

A wide variety of resampling algorithms exist, however many of them rely on largely the same insights. In the assignment you will be implementing the *stratified* and *multinomial* resampling schemes as they are outlined in the pseudo code below (based on here and here).

```
STRATIFIED RESAMPLING
GIVEN: Particles x and size N
-------------------------------
n = 0
m = 1

Q_0:N = cummulative_sum(particle_weights)

while n <= N:
        u_0 ~ U(0,1/N]
        u = u0 + n/N

        while Q_m < u
                m = m + 1

        n = n + 1
        y_n = x_m
```

```
------------------------------
RETURN Particles y
```

```
MULTINOMIAL RESAMPLING
GIVEN: Particles x and size N
------------------------------
n = 0

Q_0:N = cummulative_sum(particle_weights)

while n <= N:
        m = 1
        u ~ U(0,1]

        while Q_m < u
                m = m + 1

        n = n + 1
        y_n = x_m
------------------------------
RETURN Particles y
```

The methods you need to implement are

```
void Resampler::_multinomial(ParticleList &Particles, const int N)
```

and

```
void Resampler::_stratified(ParticleList &Particles, const int N)
```

The Assignment

Implementation

- [ ] Complete the _multinomial method
- [ ] Complete the _stratified method.

Testing the result

The Assignment

Test the developed particle filter framework in simulation. How accurate is the implemented algorithm? What are strengths and weaknesses? Write down your observations.