Assignment localization

0. Assignment introduction

In the lectures, you have learned about the localization problem and why it is relevant for mobile robots. This assignment is an opportunity to implement a localization algorithm to get an even better understanding. Furthermore, the code that is developed during this assignment can be used for the final challenge of the course.

You will implement a particle filter as introduced in the lectures. A base for the code is provided so that you only have to add the core functionality of the particle filter, and not be distracted by tedious practicalities. Furthermore, tests are provided so that you can easily verify your implementation.

Before starting the assignments, make sure to:

- install the MRC exercise tools and complete the C++ tutorials (see the exercises of week 1)
- have a basic understanding of the particle filters from watching the lectures
- Pull the exercise code framework from your group's git repo

0.1 Explore the code framework

Open the code framework and see how it is structured. The comments in the header files (`ParticleFilter.h`, `ParticleFilterBase.h`, etc.) are often a great way to help your understanding of what each method implements.

Assignment 0.1

Explain in a few concise sentences per item:

- How is 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 `ParticleFilterBase` and `Particle` classes implement a propagation method. What is the difference between the methods?

0.2 Compile and run the code

To check if everything works as intended, you must first compile the code (for a full explanation, see the introduction excercises from week 1). Open a terminal in the assignment folder `\3_localization\` and enter the following:

```
mkdir build
cd build
cmake ..
make
```

Afterward, run the tests you just compiled:

1. In the terminal, navigate to `\3_localization\bin\`
2. Run the the binary of the test you want to run (e.g. `./assignment1_1` for the first assignment)

Assignment 0.2

- Without changing anything in the code base, compile the code, make sure there are no errors.
- Make sure you can run the tests. Do not worry when all of them return "test fails".

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.

1. Assignments for the first week

1.1 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 1.1 we will implement the methods which construct this set of particles. A particle can be constructed with either of the following methods:

```
Particle::Particle(&world, &weight, *generatorPtr);

Particle::Particle(&world, mean[3], sigma[3],  &weight, *generatorPtr);
```

In the `Particle.cpp` file you received, both of these methods initializes the particles with pose (x, y, theta) = (0, 0, 0). This does not fulfill the desired behavior as described in `Particle.h`. You should adapt both constructor methods to sample from either a Gaussian or uniform distribution.

Next, we want to initialize many particles. As you might have discovered, the `ParticleFilterBase` class contains a vector `_particles` storing all its 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);
```

These constructor methods should be adapted such that the particle vector gets populated.

Assignment 1.1

Implementation

- Take a look at `Particle.h` to read the description of how the constructor methods *should* work
- Complete both `Particle()` constructors
- Complete both `ParticleFilterBase()` constructors
- Run the tests `assignment1_1` to validate that your methods function correctly.
- Run `main` (while also running `mrc-sim` and `sim-rviz`) to check whether you see particles colored in red. You should see something similar like in the image below.

Wiki documentation

- 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?

- Add a screenshot of rviz to show that the code is working

1.2 Calculate the pose estimate

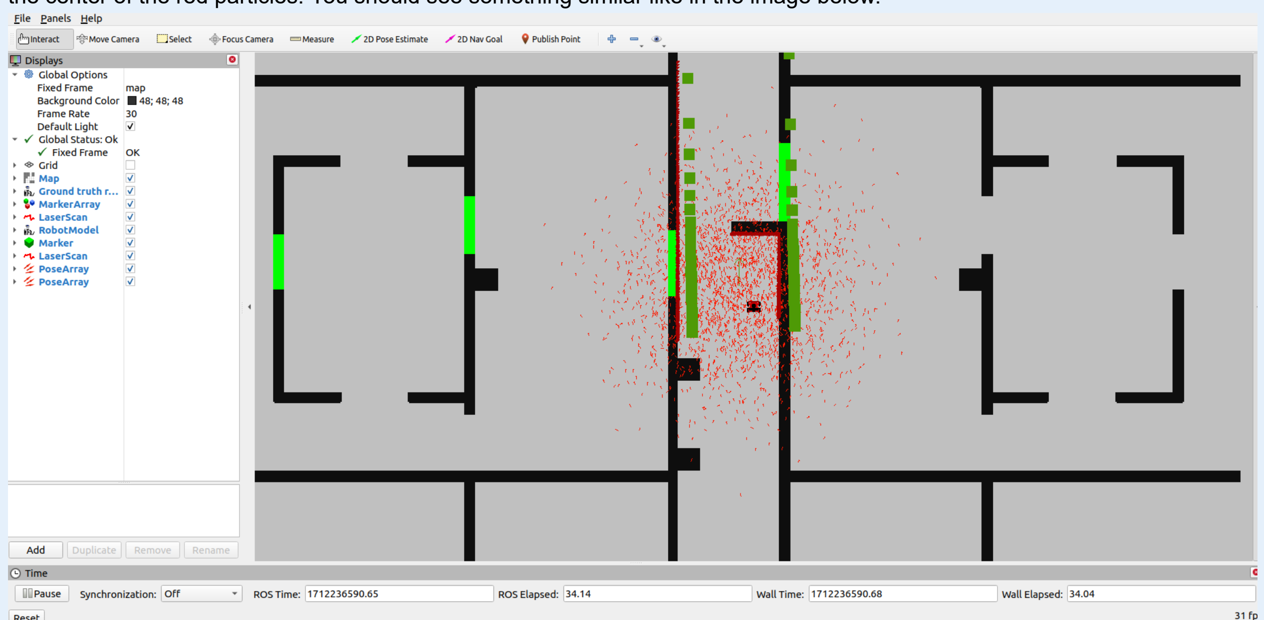
Having initialized the filter, we are interested in extracting the pose estimate from the filter. As stated in the lectures, the filter approximates the probability distribution of the robot pose by a cloud of particles. The filter estimate is then the expected value of this distribution.
Within our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

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

Assignment 1.2

Implementation

- Complete the `get_average_state()` method
- Run the tests `assignment1_2` to validate that your methods function correctly.
- Run `main` (while also running `mrc-sim` and `sim-rviz`) to check whether you see the weighted pose (a thin green arrow) at the center of the red particles. You should see something similar like in the image below.



Wiki documentation

- 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.
- Add a screenshot of rviz to show that the code is working

> *Tip*: Consider how you can find the circular mean. For example, what is the average of $-\pi$ and π when considering rotations?

1.3 Propagate the particles with odometry

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 previous step, 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:

```
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 standard deviation of the zero-mean noise we inject during the propagation, and offset_angle is the current rotation between the odometry frame and the robot frame.

Assignment 1.3

Implementation

- Complete the `propagateSample()` method
- Run the tests `assignment1_3` to validate that your methods function correctly.
- Run `main` (while also running `mrc-sim` and `sim-rviz`). Also run `mrc-teleop` so that you can move the robot. The particles and the weighted average should move as the robot drives around. It is expected that the particles will slowly diverge from the actual pose.

> *Tip*: Note that the odometry data is collected in the robot frame, whereas the pose of the particles is stored in map frame. In order to perform an accurate propagation, first transform `dPose` into the map frame.

Wiki documentation

- 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?
- Add a screenrecodring of rviz while driving to show that the code is working

> *Tip:* You can start a recording your ubuntu screen with `Ctrl + Alt + Shift + R`. Pressing the same keys again will stop the recording. The video is saved in the videos folder.

2. Assignment for the second week

2.1 Correct the particles with LiDAR

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, are not doing any better than simply integrating the odometry. 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 localization 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 we will assign a weight to each particle.

To assign a weight to a particle that represents a pose, you can first predict what measurements you expect at that pose. This prediction can be compared with the actual measurement based on a sensor model to compute the likelihood of the measurement. You can then use likelihood as the weight of the particle.

In this assignment, you must implement the measurement model. Furthermore, you must use this to find the likelihood of the measurement.

```
Likelihood Particle::computeLikelihood(const measurementList &data,
World &world, const MeasModelParams &lm)

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

The Assignment 2.1

Implementation

- Complete the `measurementmodel()` method
- Complete the `computeLikelihood()` method
- Run the tests `assignment2_1` to validate that your methods function correctly.

- (Running `main` should not show any major difference, since the likelihood is not directly visualized. You might see a slight improvement in the weighted average pose, however.)

Wiki documentation

- 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.

2.2 Re-sample the 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)
```

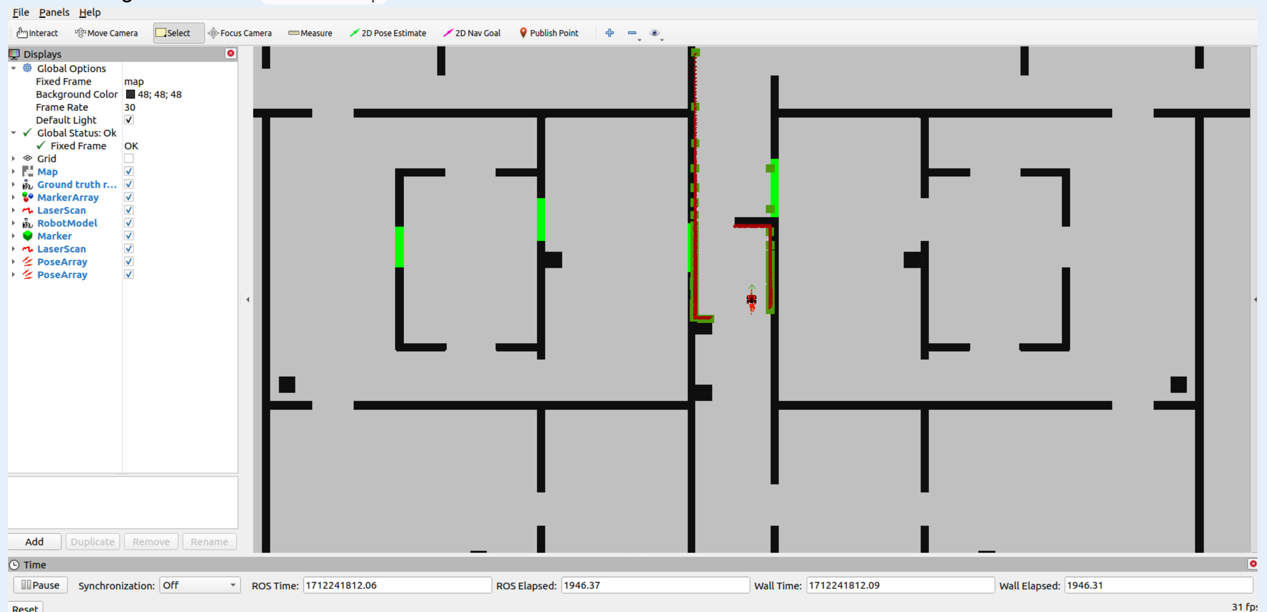
and

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

Assignment 2.2

Implementation

- Complete the `_multinomial()` method
- Complete the `_stratified()` method.
- (you can select which sampling algorithm to use in `params.json`)
- Run `main` (while also running `mrc-sim` and `sim-rviz`) to check whether the particles converge to the actual pose, even when driving around with `mrc-teleop`.



Wiki documentation

- Explain in a few concise sentences per item:
  - What are the benefits and disadvantages of both the multinomial resampling and the stratified resampling?
  - 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?

2.3 Test on the physical setup

By now, you have created a localization algorithm and validated it in simulation. Let's find out it if we can use it on a real robot.

The Assignment

Implementation

- Run your localization code on the real robot setup. See if you can load a map that reflects the real scenario.
- Tune the parameters of the algorithm (see `params.json`). It is recomended you test this in simulation first.
- (optional:) compare your localization algorithm with the ground truth given by the opti-track system.

Wiki documentation

- Explain in a few concise sentences per item:

    - How you tuned the parameters.
    - How accurate your localization is, and whether this will be sufficient for the final challenge.
    - How your algorithm responds to unmodeled obstacles.
    - Whether your algorithm is sufficiently fast and accurate for the final challenge.
- Include a video that shows your algorithm working on the real robot
- When finished with the assignments, push the code to your group's git repo