# Coding Problem Set 3 - Particle Filter Localization

Joshua Mangelson

November 12, 2021

**Purpose:** The purpose of this lab is to give you hands on experience implementing and the ability to play around with particle filter localization.

## 1 Setup and Initial Instructions

### 1.1 Pulling the Code

You will be able to access the code for each of the labs in this course via Git. For this lab specifically, you will be able to access it via the following link: `https://bitbucket.org/byu_mobile_robotic_systems/lab3-particle-filter-localization`

### 1.2 Setting Up Your Environment

The programming language for this lab is Python. We assume you will be using Python3. The code has the following dependencies:

1. matplotlib - (this is a fundamental python graphing library)
2. numpy - (this is a fundamental python matrix library)
3. tqdm - (used to visualize progress)
4. pytest - (unit test runner)
5. scipy - (Common scientific computing library - used in the tests)
6. numba - (compiles python code as C-code for faster execution)

On ubuntu, you should be able to install what you need via the following commands, run from the cloned repository:

```
sudo apt-get install python3
sudo apt-get install python3-pip
pip3 install -r requirements.txt
```

Alternatively, if you are using conda, activate the environment you setup for use with the previous labs, and then run the following from the base directory of this lab's repository:

```
pip install -r requirements.txt
```

and throughout the rest of these instructions, if you are using conda, replace 'python3' in any commands with 'python'.

### 1.3 Code Overview

The code needed for this lab is found inside the **lab3** folder. This file contains the **occupancy_grid_map.py** file from lab2 with a few modifications and a new file named **particle_filter_localization.py** with the following class and function:

- **ParticleFilterLocalizer** - You will implement this class to apply the particle filter localization algorithm we discussed in class to localize a robot in the map of the environment from lab2.

- **main** - this function is the function that is run if you type the following command in a terminal:

```
python3 lab3/particle_filter_localization.py
```

# 2 Lab Overview

For this lab, you will implement the particle filter localization algorithm discussed in class.

Try running the following command from the repository root directory:

```
python3 lab3/particle_filter_localization.py
```

This command calls the **main** function which loops through all of the data stored in **location-dataset.npz**

**Your goal in this lab is to process this stream of data localize a robot in the environment we mapped in lab2.**

If the **-p** option is set when the python file is ran, then the function will physically plot to the screen every 5 steps. Otherwise, if the **-p** option is not used, no plotting will be done.

For more info on running the file, including a list of the available options, run:

```
python3 lab3/particle_filter_localization.py --help
```

## 2.1 Data Stream

To generate the data used in this lab, we simulated a robot with 20 laser range finders spread out at various angles, just as in lab 2. Each of these laser range finders is directed in a slightly different direction on the front of the robot. We then navigated the robot through a simulated building, entering each room one-by-one.

At each time step we recorded (noisy) measurements of the robots change in position from time step $t$ to $t+1$ ($\mathbf{u}_t$) and the range returned by each of the 20 laser range finders ($\mathbf{z}_t$) collected after the robot moved according to $\mathbf{u}_t$.

The data stored in **location-dataset.npz** includes the following:

- **angles** - a list of 20 angles (in degrees) representing the angles of the corresponding laser-range finders
- **U_t** - a list of lists of the form

  $$[\mathbf{u}_0 = [\delta x_0, \delta y_0, \delta\theta_0], \mathbf{u}_1 = [\delta x_1, \delta y_1, \delta\theta_1], \cdots, \mathbf{u}_{n-1} = [\delta x_{n-1}, \delta y_{n-1}, \delta\theta_{n-1}]]$$

  where each internal list specifies the change in pose (or motion) that occurred between timesteps $t$ and $t+1$.
- **Z_tp1** - a list of lists of the form

  $$[\mathbf{z}_0 = [z_{0,0}, z_{0,1}, \cdots, z_{0,m}, \cdots, z_{0,19}], \mathbf{z}_1 = [z_{1,0}, z_{1,1}, \cdots, z_{1,m}, \cdots, z_{1,19}], \cdots, \mathbf{z}_T = [z_{T,0}, z_{T,1}, \cdots, z_{T,m}, \cdots, z_{T,19}]]$$

  where each internal list specifies the set of 20 range measurements $\{z_{t,m}\}_{m=0}^{19}$ returned from the laser-range finders at time $t$.
- **X_t** - a list of lists of the form

  $$[\mathbf{x}_0 = [x_0, y_0, \theta_0], \mathbf{x}_1 = [x_1, y_1, \theta_1], \cdots, \mathbf{x}_n = [x_n, y_n, \theta_n]]$$

  where each internal list specifies the true $xy$-position and $\theta$-orientation of the robot. This list defines the true pose of the robot and should be used only for evaluation and plotting.

## 2.2 Implementation

We have given you the general framework that iterates through the data and plots particle positions to the screen. You will need to complete the implementation of the following functions:

- **propagate_motion**
- **expected_measurement**

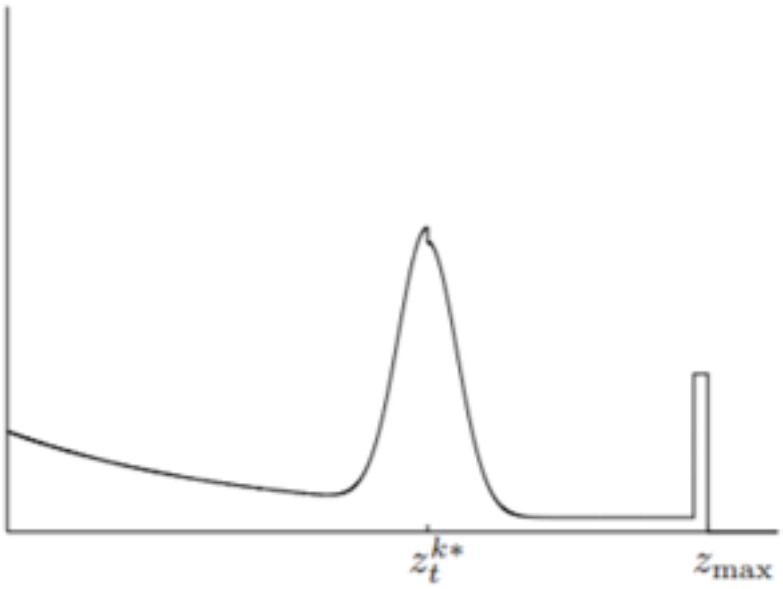


**Figure 6.4** "Pseudo-density" of a typical mixture distribution $p(z_t^k \mid x_t, m)$.

Figure 1: The range finder model described in class and in the Probabilistic Robotics book.

- **update_weight**
- **normalize_weights**
- **resample_particles**
- **iterate**
- **main**

## 2.3 Process (Motion) Model

To implement this, you should use the following process model:

$$\mathbf{x}_{t+1} = \mathbf{x}_t \oplus \mathbf{u}_t + \epsilon$$

with

$$\epsilon \sim \mathcal{N}(\mathbf{0}, \Sigma_{\mathbf{u}_t})$$

and

$$\Sigma_{\mathbf{u}_t} = \begin{bmatrix} 0.04 & 0 & 0 \\ 0 & 0.04 & 0 \\ 0 & 0 & 0.01 \end{bmatrix}$$

## 2.4 Laser Range Finder Measurement Model

You will also need to implement the measurement model we described in class (as shown in Figure 1) for laser range sensors. This model evaluates the probability of your measurements given the current estimate of the map:

$$p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{m})$$

To evaluate this you will need to determine a set of weights ($\alpha_{hit}$, $\alpha_{unexp}$, $\alpha_{max}$, $\alpha_{rand}$) for the weighted sum. In this dataset, there are no unexpected obstacles and the max distance of the sensor is 20 meters. **IMPORTANT**: For $p_{hit}$, use a normal distribution with $\sigma = 1$. If you don't, tests WILL fail. You're welcome to experiment with other values, but this worked well in testing. This will all be done in **update_weight**.

You will also need to determine the expected range $\mathbf{z}_t^{k*}$. This can be estimated using the particle's pose and the prior map. However, since the prior map is an occupancy grid, we will need to convert from a probabilistic representation of occupancy to a binary true/false value. To do so, round the probabilistic value to either 1 or 0 to determine if a cell is occupied or not. (Note the occupancy map is in log odds, so closer to 0 probability corresponds to $< 0$ log odds and closer to 1 probability is $> 0$). This will all be done in **expected_measurement**. Further, search out to a max of 25 meters, for the case where no obstacle existed within range. This will be far enough out for the gaussian $p_{hit}$ to not interfere.

**TODO:** Include several images showing your algorithm's output at different time steps.

**TODO:** Include the specific parameter values you selected in your writeup.

## 2.5 Evaluation

For computational purposes, we've added in the line

$$self.particles[0] = np.array([1.5, 1.5, 0])$$

to "jumpstart" the algorithm by adding a particle at the exact starting position of the robot.

**TODO:** Remove the jumpstart particle, and test how many particles it takes to properly localize. This will slow down your program, so feel free to kill it after the first couple of time steps. Include the number of particles in your writeup.

Though not ideal, sometimes the mean position/rotation of the particles is used as a way to summarize the distribution.

**TODO:** Generate two plots showing the error between the mean position/rotation of the particles and the true values stored in the **X_t** vector, versus time. This error should shrink to zero as your filter converges to the true solution.

**TODO:** Include these plots in your writeup.

## 2.6 Testing

Included in the lab are a number of tests to ensure that everything has been implemented properly. To run them, when in the base repo directory simply run

```
pytest
```

Note these tests are necessary, but not sufficient conditions, meaning if they fail, you've done something wrong, but if they pass, it's not a 100% guarantee that all your code is functioning properly.

**TODO:** Include the output of '`pytest -v -rN --tb=no --no-header`' in your writeup.

## 2.7 Wrap Up

**TODO:** Create a short 30 second video of you showing your algorithm operating, your final solution, and explaining your understanding of the method. Please also briefly explain some of the pros/cons of particle filters.

**TODO:** Upload as **separate** attachments to Learning Suite:

- Your video (as a .mp4 file)
- Your writeup (as a .pdf file)

- A zipfile of your code (this should only contain a single file inside: **particle_filter_localization.py**)

.

## 2.8 A note on Numba

Numba is a really cool python package for accelerating python code. From their website "Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code."

By default, we've setup numba to speed up the **expected_measurement** and **update_weight** functions. When running on our implementation, it was the difference between the entire algorithm running in 1 second vs 60. You can only use certain functions - python native functions, most numpy functions, and other functions with "@njit" - inside of functions you're trying to translate. This can be difficult to do so if numba is causing too many hassles, feel free to remove the line "@njit", right above the function definition for the **expected_measurement** and **update_weight** functions. But be aware your code will be much slower.