TTIC 31170: Robot Learning and Estimation (Spring 2025)

Problem Set #3

Due Date: May 8, 2025

1 SLAM as a Least Squares Problem [15 pts]

Consider a smoothing formulation of SLAM, where the objective is to maintain the belief over the entire history of robot poses and map, $p(\boldsymbol{x}_{0:t}, M|\boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t})$, where M is a vector of landmarks $\boldsymbol{m}_j \in M$ (e.g., represented as geometric primitives) and $\boldsymbol{z}_{1:t}$ is a set of K observations. Note that this can also be interpreted as a pose graph with landmarks, as discussed in class. Figure 1 presents the Bayesian network for this SLAM formulation.

Assume that the process and measurement models have the following form

$$\boldsymbol{x}_i = f(\boldsymbol{x}_{i-1}, \boldsymbol{u}_i) + \boldsymbol{v}_i \tag{1a}$$
$$\boldsymbol{z}_k = h(\boldsymbol{x}_{i_k}, \boldsymbol{m}_{j_k}) + \boldsymbol{w}_k \tag{1b}$$

where $f(\cdot)$ and $h(\cdot)$ are nonlinear functions, $\boldsymbol{x}_{i_k}$ and $\boldsymbol{m}_{j_k}$ are vectors that denote the robot pose and map variables involved in the observation at time t (we assume known data association), and $\boldsymbol{v}_i \sim \mathcal{N}(0, R)$ and $\boldsymbol{w}_k \sim \mathcal{N}(0, Q)$ are zero-mean, independent Gaussian random variables that represent noise.

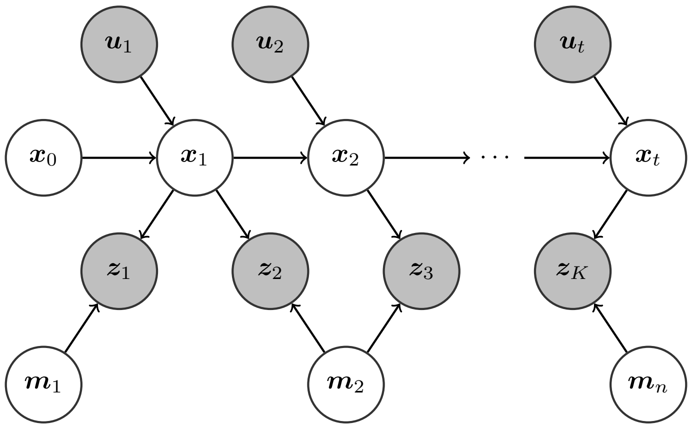


Figure 1: Bayesian network representation of a smoothing formulation of SLAM.

For this problem, we are interested in the maximum a posteriori (MAP) estimates for the robot pose history and map

$$\boldsymbol{x}^*_{0:t}, M^* = \underset{\boldsymbol{x}_{0:t}, M}{\arg\max}\; p(\boldsymbol{x}_{0:t}, M|\boldsymbol{u}_{0:t}, \boldsymbol{z}_{1:t}) \tag{2a}$$
$$= \underset{\boldsymbol{x}_{0:t}, M}{\arg\max}\; p(\boldsymbol{x}_{0:t}, M, \boldsymbol{u}_{0:t}, \boldsymbol{z}_{1:t}) \tag{2b}$$

(a) **[4pts]** Show that solving for the MAP pose history and map (2) is equivalent to a nonlinear least squares problem. **Hint**: Factorize the distribution according to the independencies expressed in the Bayesian network (e.g., Markovity) and use the fact that taking the $\log$ doesn't affect the maximization.

$$p(x_{0:t}, M, u_{0:t}, z_{1:t}) = p(x_0)\prod_{s=1}^{t} p(x_s|x_{s-1}, u_s)\prod_{s=1}^{t} p(z_s|x_s, M)$$

$$\log p(x_{0:t}, M, u_{0:t}, z_{1:t}) = \log p(x_0) + \sum_{s=1}^{t}\log p(x_s|x_{s-1}, u_s) + \sum_{s=1}^{t}\log p(z_s|x_s, M)$$

We have the motion model likelihood as

$$p(x_i|x_{i-1}, u) = \mathcal{N}(x_i; f(x_{i-1}, u_i), R)$$

$$\log p(x_i|x_{i-1}, u_i) \sim -\frac{1}{2}||x_i - f(x_{i-1}, u_i)||^2_{R^{-1}}$$

and the observation model likelihood as

$$\log z(z_k|x_{i_k}, m_{j_k}) \sim -\frac{1}{2}||z_k - h(x_{i_k}, m_{j_k})||^2_{Q^{-1}}$$

So the full MAP optimization becomes

$$||x_0 - \bar{x}_0||^2_{\Sigma^{-1}} + \sum_i ||x_i - f(x_{i-1}, u_i)||^2_{R^{-1}} + \sum_k ||z_k - h(x_{i_k}, m_{j_k})||^2_{Q^{-1}}$$

Each term is clearly a nonlinear residual squared, weighted by the inverse of the noise covariance. Thus, this is equivalent to a nonlinear least squares problem because f and h are nonlinear.

(b) **[10pts]** Assume that the process and measurement models can be linearized about an initial estimate for the robot pose $\boldsymbol{x}_i^0$ and map $\boldsymbol{m}_j^0$. Show that the optimization can be formulated as

$$\boldsymbol{\theta}^* = \arg\min_{\delta\boldsymbol{\theta}} \|A\delta\boldsymbol{\theta} - \boldsymbol{b}\|^2 \tag{3}$$

where $\delta\boldsymbol{\theta}$ is a vector of elements of the form $\boldsymbol{x}_i - \boldsymbol{x}_i^0$ and $\boldsymbol{m}_{j_k} - \boldsymbol{m}_{j_k}^0$ (i.e., as in a Taylor series expansion), and $\boldsymbol{b}$ is a vector. **Hint**: You can convert a weighted least squares objective (i.e., weighted by the inverse noise covariance matrices) to a standard least squares by a change of variables ($\|\boldsymbol{a}\|^2_\Sigma = \boldsymbol{a}^\top\Sigma^{-1}\boldsymbol{a} = (\Sigma^{-\top/2}\boldsymbol{a})^\top(\Sigma^{-\top/2}\boldsymbol{a}) = \|\Sigma^{-\top/2}\boldsymbol{a}\|^2$).

Following the hint, we can rewrite each term as

$$||\Sigma^{-T/2}(x_0 - \bar{x}_0)||^2 + \sum_i ||R^{-T/2}(x_i - x_i^0)||^2 + \sum_k ||Q^{-T/2}(z_k - z_k^0)||^2$$

We linearize as suggested with

$$\delta x_i = x_i - x_i^0, \delta m_j = m_{j_k} - m_{j_k}^0$$

$$f(x_{i-1}, u_i) \approx f(x_{i-1}^0, u_i) + F_i \delta x_{i-1}$$

$$h(x_{i_k}, m_{j_k}) \approx h(x_{i_k}^0, m_{j_k}^0) + H_{k,x}\delta x_{i_k} + H_{k,m}\delta m_{j_k}$$

$$r_i = x_i - f(x_{i-1}, u_i) \approx (x_i^0 + \delta x_i) - (f(x_{i-1}^0, u_i) + F_i \delta x_{i-1}) = (x_i^0 - f(x_{i-1}^0, u_i)) + \delta x_i - F_i \delta x_{i-1}$$

$$r_k \approx (z_k - h(x_{i_k}^0, m_{j_k}^0)) - H_{k,x}\delta x_{i_k} - H_{k,m}\delta m_{j_k}$$

Our total objective is then

$$\sum_i ||r_i||^2_{R^{-1}} + \sum_k ||r_k||^2_{Q^{-1}}$$

We change variables as in the hint to remove the weighting,

$$||A\delta\theta - b||^2$$

where A is a matrix of the stacked and weighted Jacobians, $F_i, H_{k,x}, H_{k,m}$ and b is a vector from the weighted predicted error terms like $x_i^0 - f(x_{i-1}^0, u_i)$ and $z_k - h(x_{i_k}^0, m_{j_k}^0)$.

(c) **[1pt]** A problem with smoothing formulations of SLAM is that the state vector grows linearly with time. Is there a particular property of A that we can exploit to solve this least squares problem more efficiently?

We see that the rows of A correspond to individual residuals and the columns correspond to particular δx_i or δm_j. Since each residual only involves two or so terms, the vast majority of A will be sparse. This means that we could do a decomposition or reordering of the variables to solve the proble more efficiently.

2 Particle Filter Localization [45 pts]

In Problem Set 2, you implemented an Extended Kalman Filter to estimate the pose of your robot vacuum cleaner. A limitation of this solution is that it required placing a sensor on the ceiling of each room. Now, you're asked to develop a method that uses particle filter localization to estimate the robot's pose relative to an occupancy grid map of the environment using a LIDAR sensor.

What is included: As part of this problem set, you are provided with the following Python modules, where those with an asterisk are files that you are expected to edit.

- `Gridmap.py`: A class that provides functions for working with occupancy grids
- `Laser.py`*: A class that defines routines for working with LIDAR data
- `Visualization.py`: A set of visualization routines
- `PF.py`*: The main particle filter class
- `RunPF.py`: Python code that instantiates and runs the particle filter

The code directory includes a `README.txt` file that describes the functions associated with each of these classes. We also provide a set of pickle files (loaded by `RunPF.py`), each of which includes the control data and LIDAR (range and bearing) observations recorded as the robot navigated in one of three different environments shown in Figure 2. Some of the files also include the robot's initial pose as well as the ground-truth pose.

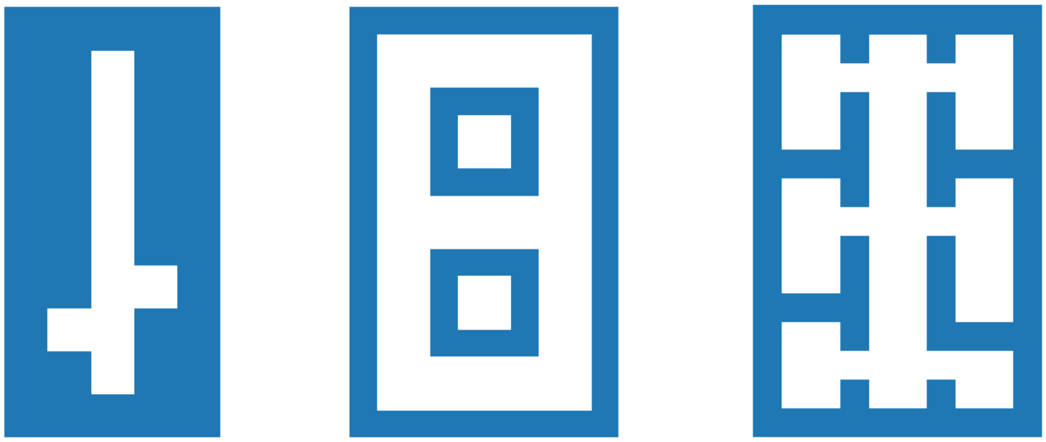

(a) Simple world (b) Loops world (c) Rooms world

Figure 2: Three different environments represented as occupancy grids, where white and blue denote free and occupied space, respectively.

(a) In the `prediction` function defined in `PF.py`, implement the algorithm provided in Table 5.3 of the Probabilistic Robotics book that samples from the following motion model:

$$x_t = x_{t-1} + \frac{\bar{u}_{t,1}}{\bar{u}_{t,2}}\big(\sin(\theta_{t-1} + \bar{u}_{t,2}\Delta t) - \sin(\theta_{t-1})\big) \tag{4a}$$

$$y_t = y_{t-1} + \frac{\bar{u}_{t,1}}{\bar{u}_{t,2}}\big(\cos(\theta_{t-1}) - \cos(\theta_{t-1} + \bar{u}_{t,2}\Delta t)\big) \tag{4b}$$

$$\theta_t = \theta_{t-1} + \bar{u}_{t,2}\Delta t + \gamma_t \Delta t \tag{4c}$$

where $\boldsymbol{u}_t = \begin{bmatrix} u_{t,1} & u_{t,2} \end{bmatrix}^\top$ is the control (velocity) input

$$\bar{u}_{t,1} = u_{t,1} + v_{t,1} \tag{4d}$$

$$\bar{v}_{t,2} = u_{t,2} + v_{t,2} \tag{4e}$$

and

$$v_{t,1} \sim \mathcal{N}(0, \alpha_1 u_{t,1}^2 + \alpha_2 u_{t,2}^2) \tag{4f}$$

$$v_{t,2} \sim \mathcal{N}(0, \alpha_3 u_{t,1}^2 + \alpha_4 u_{t,2}^2) \tag{4g}$$

$$\gamma_t \sim \mathcal{N}(0, \alpha_5 u_{t,1}^2 + \alpha_6 u_{t,2}^2) \tag{4h}$$

are independent, zero-mean Gaussian noise terms and the α_i terms are model parameters (defined in the `PF` class).

Note that you can use the NumPy `random.normal()` function to sample from the noise distributions. Calling this function to draw a single sample inside a for loop can be computationally expensive. It is more efficient to call this function once to draw N samples than to call it N drawing a single sample each. In settings where you anticipate having to draw multiple samples (e.g., inside a `for` or `while` loop), you are better off drawing N samples before entering the loop.

(b) In the `scanProbability` function defined in `Laser.py`, implement the algorithm specified in Table 6.1 of the Probabilistic Robotics book that evaluates the likelihood of a given LIDAR scan according to the robot's pose in a given map. The `Laser.py` file defines a default set of parameters, while others are provided in the `README.txt` file included with the code. **Note**: The notation in the file differs slightly from the text. In particular, we use `pX` to denote the mixture weight for component `X` of the model, while the text uses `zX`.

(c) Implement a standard particle filter that performs the following for each time step:

(i) samples from the motion model defined above;

(ii) reweights the particles according to the observation likelihood defined above; and

(iii) resamples N particles according to their weights.

This will require writing code for several functions defined in `PF.py` as specified in the `README.txt` file included with the code. Note that you are not required to evaluate on all sets of parameter settings and are free to tweak the parameters as you see fit.

(d) Run your particle filter implementation on all of the pickle files, with the exception of `simple_world_path2` and `rooms_world_path2`. For each of these six scenarios, the robot's initial pose is known (specified by the variable `X0`). Consequently, you can generate the initial set of particles from a Gaussian distribution centered at the initial pose using the `sampleParticlesGaussian` function defined in `PF.py`.

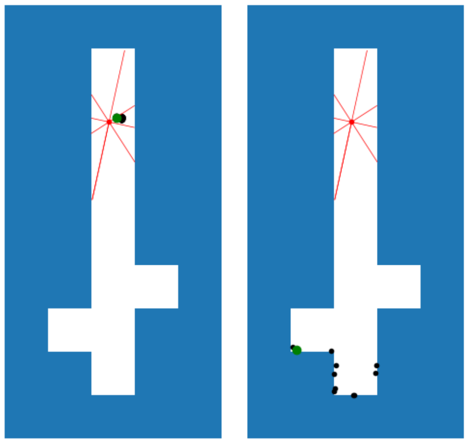
For each of the six scenarios, experiment with the number of particles. Using too few particles will make the filter prone to diverging. Describe any interesting phenomena that you see.

What to hand in: In addition to a working version of the code (see below), submit images of the final particle distribution for different numbers of particles, one for which the filter fails and at least one for which it performs well (you can define "well" visually, or quantitatively based on the error between the mean estimate and ground-truth pose). Name or label the figures to make it clear how many particles were used for each experiment (by default, the code should save figures to a file with a name indicative of the number of particles).

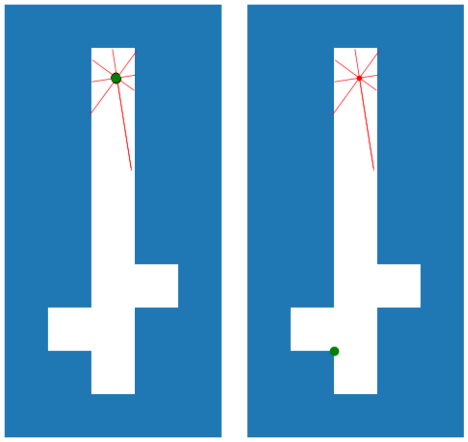
Note: You may want to use one of the `simple_world` scenarios when developing your implementation since the environment is simpler than the other two.

It seems that lower numbers of particles generally correlate to more resampling (in an inverse square-root relationship), which makes sense. I also recorded N-eff after each step in a couple runs and observed its variance decrease dramatically for larger N.

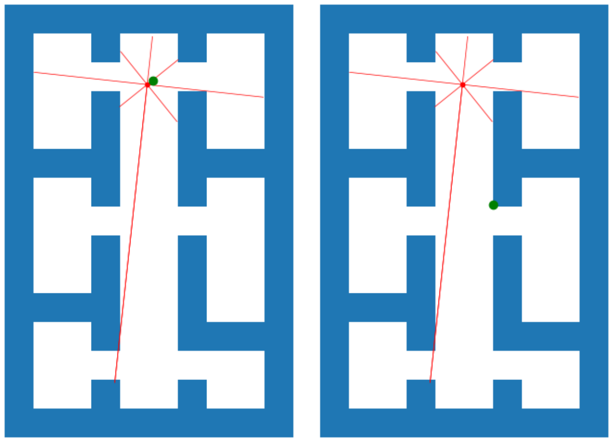
- Simple World Path 1: 1000 particles on the left and 10 particles on the right.



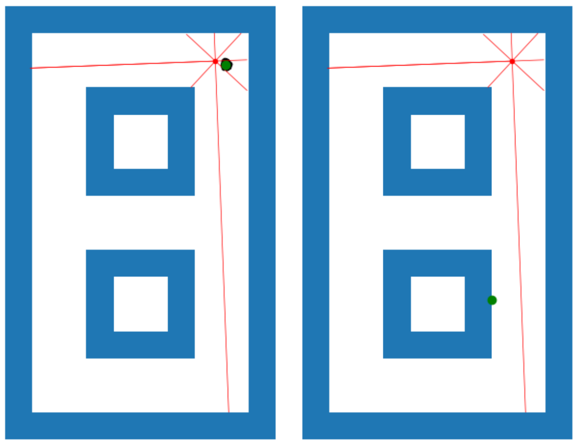
- Simple World Path 3: 2000 Particles on the left and 1000 particles on the right.



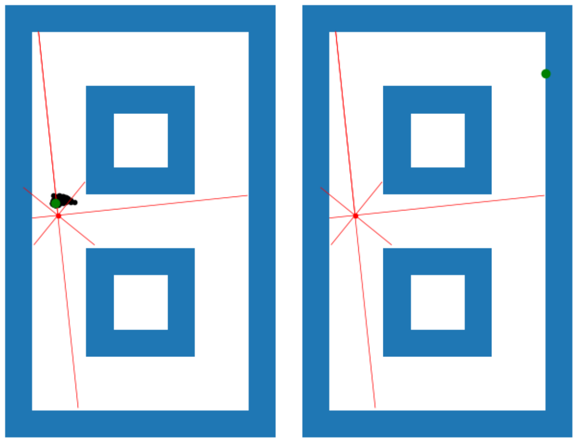
- Rooms World Path 1: 1000 particles and then 100 particles.



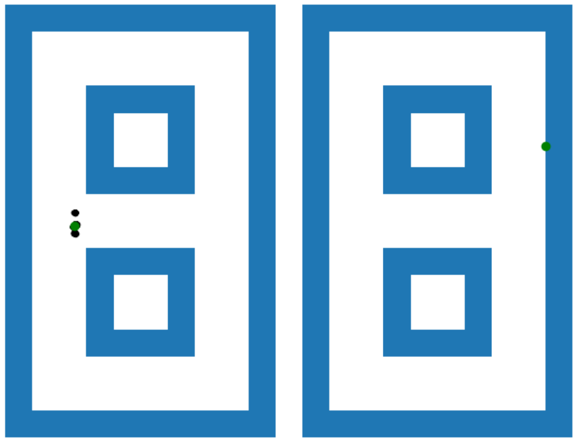
- Loops World Path 0: 1000 particles and then 50 particles.



- Loops World Path 1: 1000 particles and then 100 particles.



- Loops World Path 2: 1000 particles and then 100 particles.



(e) Two of the files, `simple_world_path2` and `rooms_world_path2`, do not include the robot's initial pose or the ground-truth pose. The lack of a prior over the robot's pose is referred to as the "kidnapped robot" problem. In this case, you can generate the initial set of particles by sampling from a uniform distribution (see `sampleParticlesUniform` in `PF.py`).

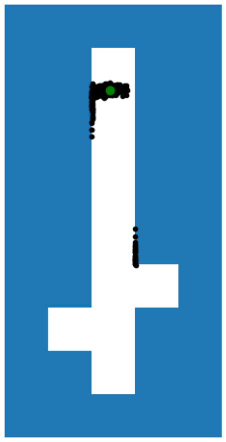
For each of the two scenarios, experiment with different numbers of particles and explain the behavior that you observe.

What to hand in: Provide the number of particles that you think are suitable for each of the scenarios (these numbers need not be the same for the two) and submit images of the final particle distribution for your chosen number of particles.

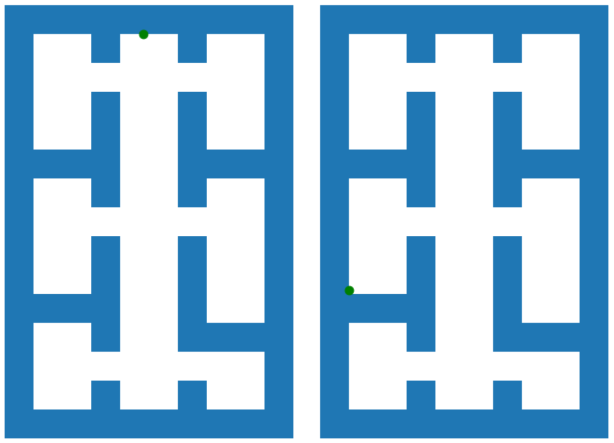
Note: based on the data provided, it seems that loops world path 2 is the second pickle file without an initial pose or ground-truth pose as opposed to rooms world path 2, so I used that instead.

I experimented a fair bit with particle counts for both and mostly judged ”success” pretty subjectively just based on the behavior of the filter and what generally made sense. After a fair amount of experimentation, I ended up back at the first number I tried for both, which was just the default of 1000.

- Simple World Path 2: 1000 Particles.



- Rooms World Path 2: 1000 particles and then 100 particles.



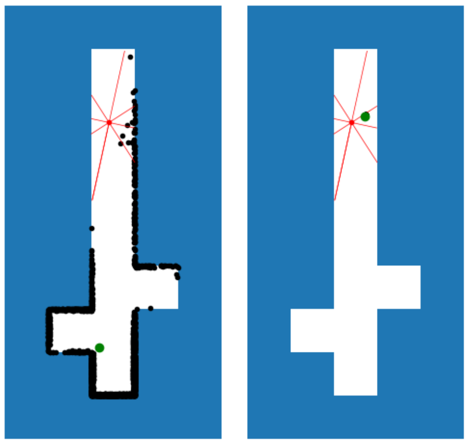
(f) Explore the effects of resampling by considering a variation of your filter that never resamples and a second that resamples when the number of effective particles (defined in the lecture notes) drops below a threshold (e.g., $2N/3$). For each of the eight scenarios, try the values for N that you identified above and compare the effects of the three different resampling strategies (including resampling at each timestep as above). If the filter performs poorly, what N did you have to use in order for the filter to converge (by either a subjective or objective measure)?

What to hand in: In addition to the discussion, provide a visualization of the final particle distribution without resampling and when resampling according to the number of effective particles for the values of N identified in parts (d) and (e). If you had to change N to get the filter to perform well, also submit an image for the corresponding particle distribution.

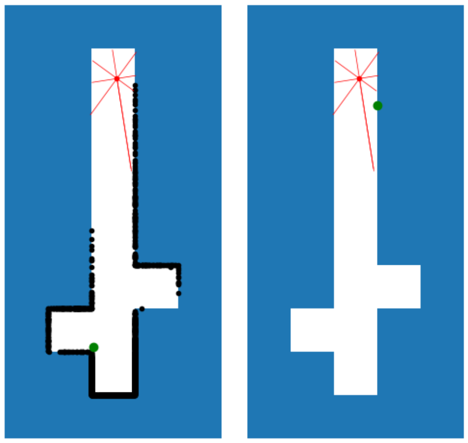
The variation that never resampled always ended up with a large number of particles pushed along the edges of the space (i.e. along the walls), but it seemed that even in those cases, some of the particles were still performing well in approximating the actual position and so in some cases, no tweaking of N was necessary. In other cases though, I wasn't able to get the filter to perform that well even across a range of tested values (up to 5000).

On the other hand, it seems that setting a threshold of 2N/3 is still reasonably effective out of the box, but pretty computationally costly as it leads to a lot of resampling, especially with a lot of particles.

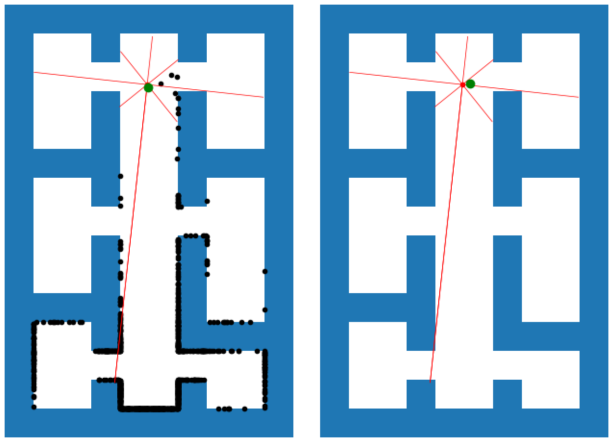
- Simple World Path 1 Never resampling on the left with 2000 particles and a threshold of 2N/3 on the right with 1000 particles.



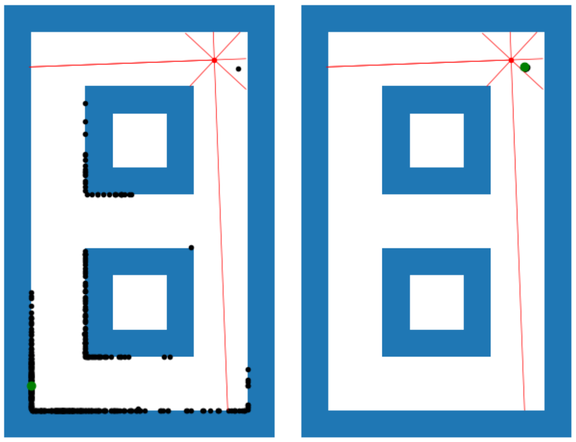
- Simple World Path 3 Never resampling on the left with 2000 particles and a threshold of 2N/3 on the right with 2000 particles.



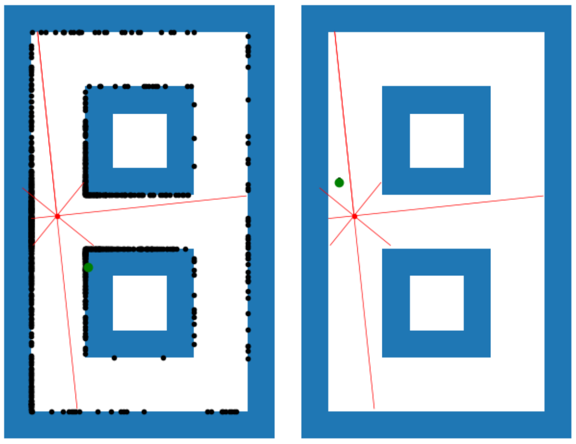
- Rooms World Path 1 Never resampling on the left with 1000 particles and a threshold of 2N/3 on the right with 1000 particles.



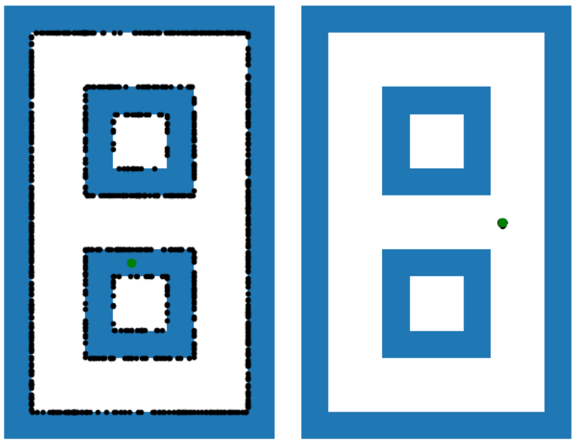
- Loops World Path 0 Never resampling on the left with 500 particles and a threshold of 2N/3 on the right with 500 particles.



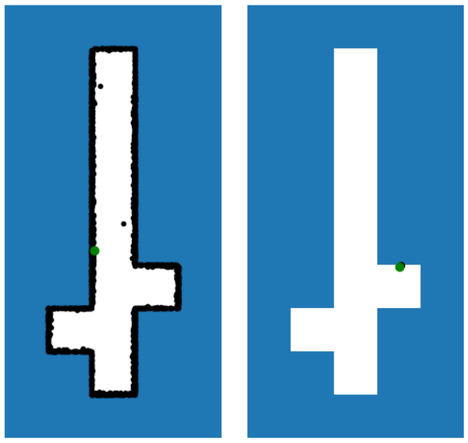
- Loops World Path 1 Never resampling on the left with 1000 particles on the left and a threshold of 2N/3 on the right with 1000 particles.



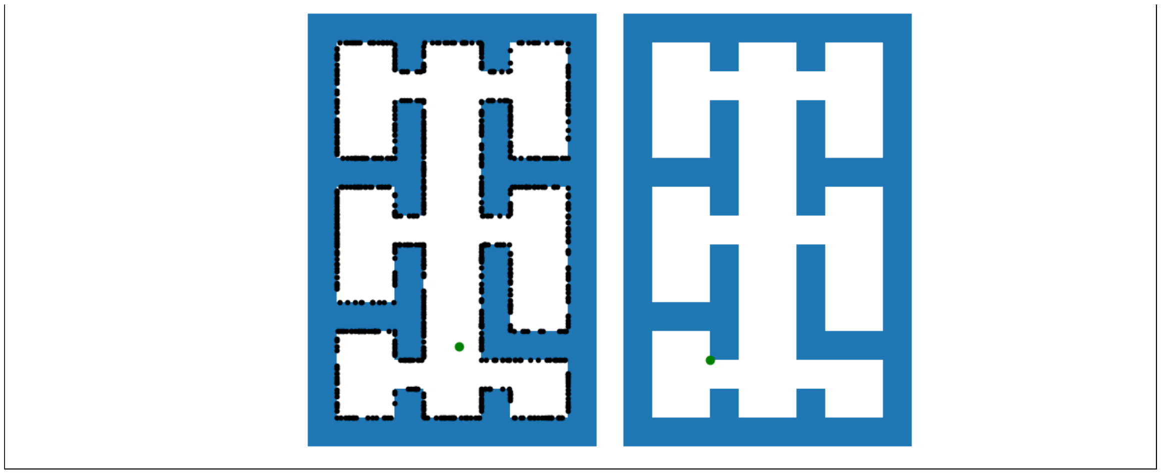
- Loops World Path 2 Never resampling on the left with 1000 particles and a threshold of 2N/3 on the right with 1000 particles.



- Simple World Path 2 Never resampling on the left with 2000 particles and a threshold of 2N/3 on the right with 1000 particles.



- Rooms World Path 2 Never resampling on the left with 1000 particles and a threshold of 2N/3 on the right with 1000 particles.



What to hand in: A zip or tar file containing your `PF.py` implementation along with the rest of the Python code released as part of the problem. Your code should run via a call of the form

```
$ python RunPF.py [--numParticles N] [--sparsity M] [-a] pkl_file
```

where `N` is the number of particles (the default is 1000), `-a` is a flag to animate the filter as it is running, `M` is the factor by which you can downsample the LIDAR returns (i.e., take every `M`-th value, which will speed up the filter; the default is 5), and `pkl_file` is the name of the pickle file.

Things to try: If you are interested, you can experiment with the filter's behavior when using the different measurement models specified in the `README.txt` file in the code directory. You can also use Python to generate movies that show the evolution of the particle filter.

3 EKF SLAM [40 pts]

Particle filter-based localization, as with any localization method, requires having access to a map of the environment. SLAM provides an alternative formulation to the problem whereby we will jointly estimate a map of the environment along with the robot's pose as it navigates.

Consider a nonholonomic robot with the following dynamics

$$\begin{aligned} x_t &= x_{t-1} + (d_t + v_{1,t}) \cos(\theta_{t-1}) \\ y_t &= y_{t-1} + (d_t + v_{1,t}) \sin(\theta_{t-1}) \\ \theta_t &= \theta_{t-1} + \Delta\theta_t + v_{2,t} \end{aligned} \tag{5}$$

where the control data $\boldsymbol{u}_t = [d_t \ \Delta\theta_t]^\top$ consists of the body-relative forward distance that the robot moved d_t and its change in orientation $\Delta\theta_t$, and $\boldsymbol{v}_t = [v_{1,t} \ v_{2,t}]^\top \sim \mathcal{N}(\mathbf{0}, R)$ is zero-mean Gaussian noise that captures uncertainty in the forward velocity and angular rate.[1]

Assume that the environment consists of a collection of point features (e.g., table legs). One can thus use a feature-based representation of the map (i.e., (x, y) position of each landmark). Assume also that the robot observes the *relative* (x, y) coordinates of each feature within the field-of-view of it's sensor (e.g., a LIDAR with maximum range of $4\,\text{m}$ and an angular field-of-view of 180

[1]Note that this model differs from the one that we considered in Problem Set 2.

degrees) according to the following measurement model

$$\boldsymbol{z}_t = \begin{bmatrix} \cos\theta_t & \sin\theta_t \\ -\sin\theta_t & \cos\theta_t \end{bmatrix} \begin{bmatrix} x_m - x_t \\ y_m - y_t \end{bmatrix} + \boldsymbol{w}_t \tag{6}$$

where (x_m, y_m) denotes the coordinates of the observed landmark and $\boldsymbol{w}_t \sim \mathcal{N}(\mathbf{0}, Q_t)$ denotes zero-mean Gaussian measurement noise.

What is included: As part of this problem set, you are provided with the following Python modules, where those with an asterisk are files that you are required to edit.

- `Renderer.py`: A set of visualization routines
- `EKFSLAM.py`*: The main EKF SLAM class
- `RunEKFSLAM.py`: Python code that instantiates and runs the EKF SLAM filter

The code directory includes a `README.txt` file that describes the functions defined in each of these modules. We also provide a set of pickle files (loaded by `RunEKFSLAM.py`), each of which includes the control data and robot-relative landmark observations recorded as the robot navigated according to two different combinations of motion and measurement noise parameters. The files also include the robot's initial pose as well as the ground-truth robot and map pose.

(a) **[5pts]** Adding landmarks to the map involves a closed-form expression for the landmark pose in terms of the robot state and measurement. Derive this expression based on the measurement model (Eqn. 6) as well as the corresponding Jacobian, which is necessary for the EKF.

We are given $z_t = R(\theta)^T(X_m - X_t) + w_t$, so a closed form expression while ignoring the zero-mean noise gives us

$$x_m = x_t + \cos\theta_t z_x - \sin\theta_t z_y$$

$$y_m = y_t + \sin\theta_t z_x + \cos\theta_t z_y$$

We also need to linearize around the mean, so

$$J_x = \begin{bmatrix} 1 & 0 & -\sin\theta_t z_X - \cos\theta_t z_y \\ 0 & 1 & \cos\theta_t - \sin\theta_t z_y \end{bmatrix}$$

and the Jacobian wrt the measurement,

$$J_z = \begin{bmatrix} \cos\theta_t & -\sin\theta_t \\ \sin\theta_t & \cos\theta_t \end{bmatrix} = R(\theta_t)$$

Therefore, the landmark covariance when it is first detected is

$$J_x P_{xx} J_x^T + J_z Q_t J_z^T$$

And the cross-covariance between the new landmark and the robot is of the form

$$P_{mx} = J_x P_{xx}$$

We would append these to the state covariance.

(b) **[35pts]** This next question asks you to implement an EKF-based SLAM algorithm. Provided along with the problem set is an array of control inputs $\boldsymbol{u}^T = \begin{bmatrix}\boldsymbol{u}_1 & \boldsymbol{u}_2 \ldots & \boldsymbol{u}_T\end{bmatrix}^\top$, where each column denotes the control input at each point in time. You are also provided with an array of measurements $\boldsymbol{z}^T = \begin{bmatrix}\boldsymbol{z}_1 & \boldsymbol{z}_2 \ldots & \boldsymbol{z}_T\end{bmatrix}^\top$ of the relative position of landmarks, where each column $\boldsymbol{z}_i = \begin{bmatrix}t & \texttt{id} & z_x & z_y\end{bmatrix}^\top$ denotes a robot-relative measurement (z_x, z_y) of landmark `id` acquired at time step t.

We provide you with two pickle files, `ekf-slam-small-noise.pickle` and `ekf-slam-large-noise.pickle`, that denote different magnitudes of motion and measurement noise. You can run your EKF SLAM implementation by calling

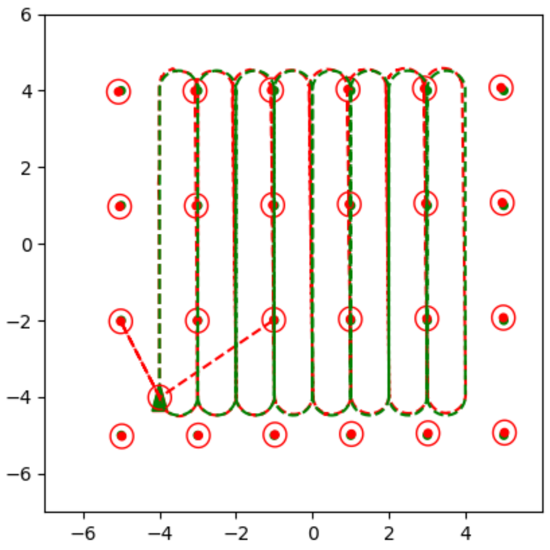
```
$ python RunEKFSLAM.py file.pickle
```

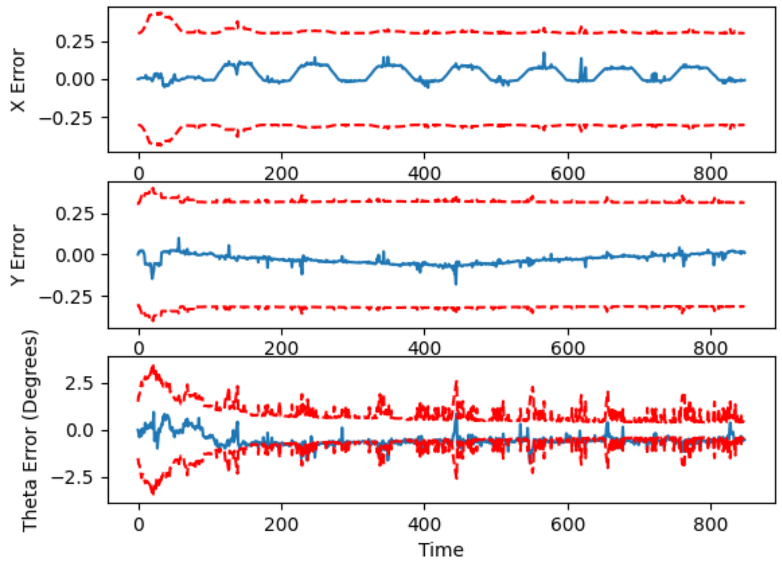
Run your code on each of these files and compare the pose estimates when using SLAM to those that you would get via dead-reckoning (you can emulate dead-reckoning by only performing the prediction step of the filter). Note that you will want to update the code to maintain a the history of pose estimates at each point in time for the sake of plotting the entire trajectory.

What to hand in: A zip or tar file containing your `EKFSLAM.py` implementation along with the rest of the Python code released as part of the problem. Your code should run via a call of the form shown above. Also, for each data file, submit (i) a plot of the final robot position and map estimates along with their uncertainty ellipses and the final ground-truth position and map and (ii) a plot that compares the dead-reckoned trajectory with the trajectory formed by your robot position estimates.

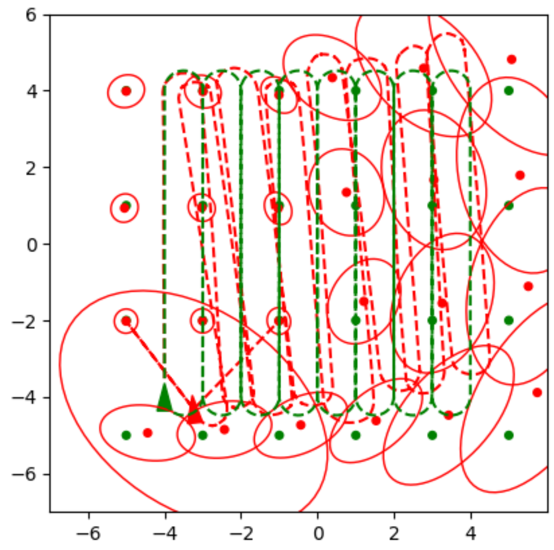
I should note that I had some numerical instability issues with the covariance over many iterations (it would end up with negative or unsymmetric values), so I manually enforced symmetry with ”self.Sigma = 0.5 * (self.Sigma + self.Sigma.T)” after the update step and that seemed to resolve it.

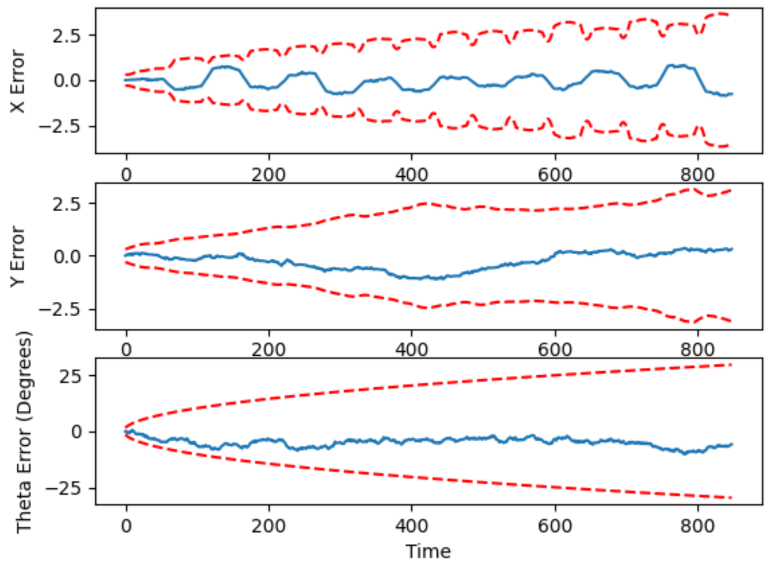
Small Noise, with updates:



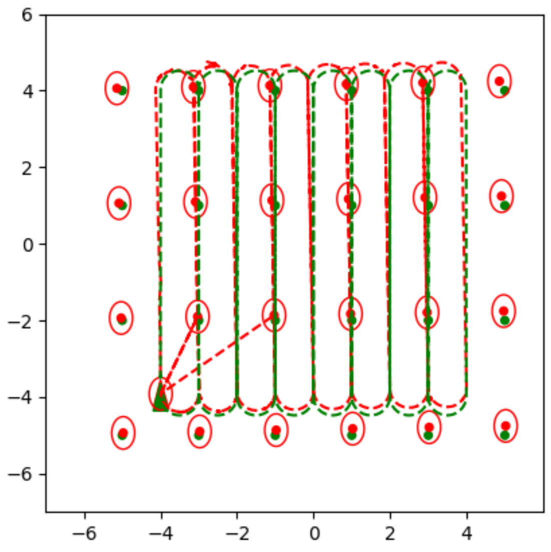


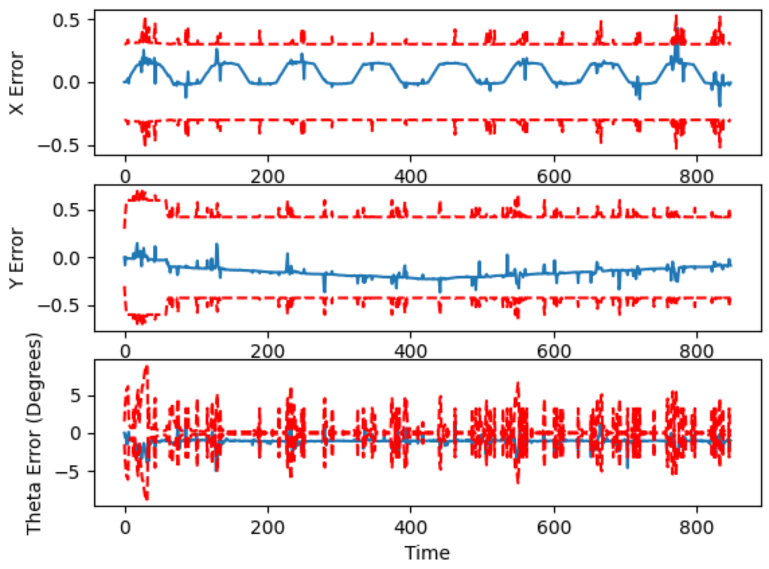
Small Noise, dead-reckoning:



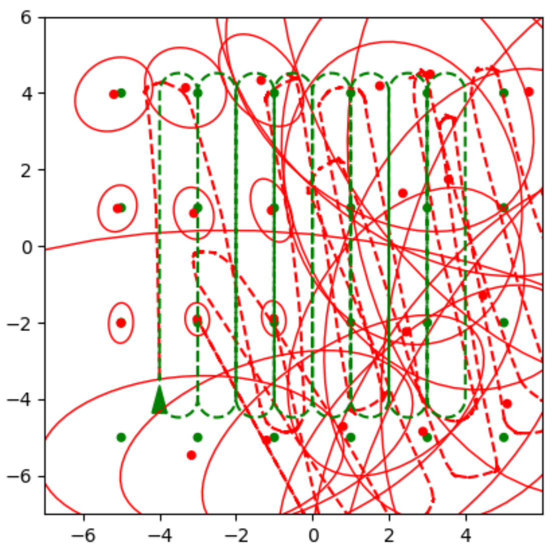


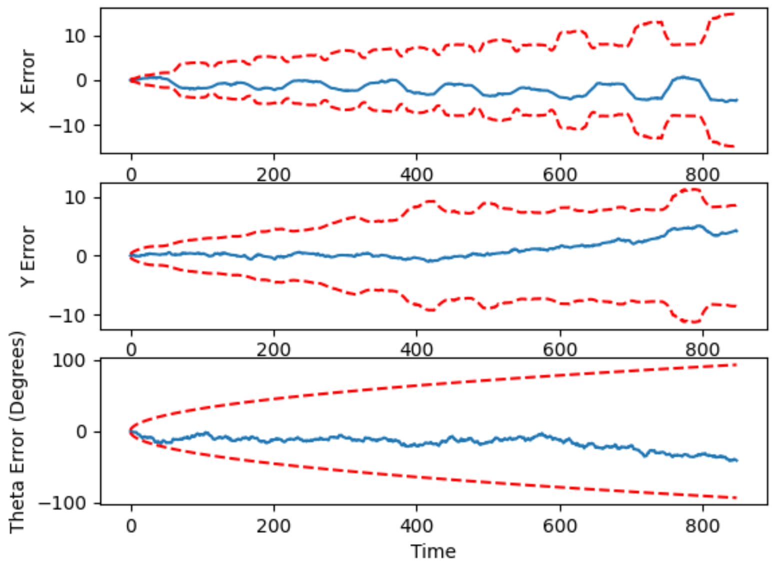
Large Noise, with updates:





Large Noise, dead-reckoning:





4 Time and Collaboration Accounting

(a) **[1pt]** Did you work with anyone on this problem set? If so, who?

N/A

(b) **[1pt]** How long did you spend on this problem set?

Between 10-15 hours. I also spent a lot of time (probably 5-10 hours) waiting for my particle filter implementation to run on certain scenarios.