Landmark-Based EKF-SLAM Derivation and Demo

Kevin Robb

1 Introduction and Assumptions

I'm using Robot Operating System (ROS Noetic) for this demo. Simply, ROS is a code architecture that allows separate nodes to run simultaneously and communicate with each other via "publishers" and "subscribers". The EKF will be implemented as a ROS node that subscribes to the odometry commands and sensor measurements, and publishes its estimate of the state.

In this demo, the robot/vehicle is completely defined in the space by its x, y, and θ components, i.e., its 2D position in the world and its global heading. We define the world frame relative to the robot's starting position and heading, so $\boldsymbol{x}_0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$.

The robot can be controlled once per timestep with a certain odometry, consisting of a forward distance and a relative heading from the current pose. We call this $\boldsymbol{u}$ by convention. Thus, the command for some timestep t will be $\boldsymbol{u}_t = \begin{bmatrix} \mathrm{d}\,d & \mathrm{d}\,\theta \end{bmatrix}^T$. In a real application, a separate ROS node will generate these commands; this could be done with some intelligent planner, taken live from a driver, or by some other method. We would then have yet another node to handle the transformation from these odometry commands to actual motor values to drive the robot.

The environment has no border, and consists of features called landmarks. A map is then simply a dictionary mapping each landmark's ID to its position. In a real-world application, these would be unique, recognizable features, such as AprilTags or other visual fiducials, brightly colored balls that can be easily thresholded in HSV and clustered, or known objects that can be fit to a depth pointcloud with RANSAC. Regardless, we assume a layer of abstraction from what a "landmark" actually is; a different ROS node will be dedicated to recognizing landmarks and publishing their ID and position to the EKF. For the EKF measurement step, we will get zero, one, or more landmark detections, each including the landmark's unique ID and its range and bearing relative to the robot. The ID will be used to ensure each landmark is tracked separately. Thus, our measurement $\boldsymbol{z}$ at some timestep will be $\boldsymbol{z}_{t,i} = \begin{bmatrix} r & \beta \end{bmatrix}^T$, where i specifies the landmark that has been detected.

The robot has some constraints on its maximum possible odometry command it can execute in a single timestamp, as well as some vision constraints on the range and field-of-view (FOV) in which landmarks can be detected.

2 EKF Overview

In general, a Kalman Filter is a method of estimating some state $\boldsymbol{x}$ as time progresses. The Extended Kalman Filter (EKF) expands the KF logic to allow for nonlinear systems, by linearizing them on each timestep with Jacobian matrices. The basic components of the KF are as follows.

- The state mean, $\boldsymbol{x}$. This is a $n \times 1$ column vector containing all variables we want to estimate.
- The state covariance, $\boldsymbol{P}$. This is a $n \times n$ square matrix. The diagonal entries represents our uncertainty in each variable in the state. The off-diagonal elements describe how every variable is correlated with every other variable.

- The commands, $\boldsymbol{u}$. This is a column vector containing some input to the system that is used to predict the next state.
- The measurements, $\boldsymbol{z}$. This is a vector describing some observation from the environment that is used to update our prediction.

The KF is an iterative process that loops back and forth between the "predict" and "update" stages to converge on the state distribution.

3 EKF for Localization Only

In the localization case, we assume the true map is known. Our goal is to use the commands and measurements to create a "live" estimate of the vehicle's current pose. As such, our state will contain exactly that. The vehicle is initially at the origin.

$$\boldsymbol{x}_0 = \begin{bmatrix} x_v \\ y_v \\ \theta_v \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

We initialize the covariance as a diagonal matrix (since initially there is no correlation between state variables) whose entries reflect our perceived variance of each state variable. This need not be extremely precise, as it will be changed on each EKF iteration.

$$\boldsymbol{P}_0 = \begin{bmatrix} 0.01^2 & 0 & 0 \\ 0 & 0.01^2 & 0 \\ 0 & 0 & 0.005^2 \end{bmatrix}$$

We also need to define some noise terms that represent some multidimensional Gaussian distributions that affect the EKF. The process noise is denoted by

$$\boldsymbol{v} \sim \mathcal{N}\left(\begin{bmatrix} v_d \\ v_\theta \end{bmatrix}, \boldsymbol{V} \right),$$

and the measurement noise by

$$\boldsymbol{w} \sim \mathcal{N}\left(\begin{bmatrix} w_r \\ w_\beta \end{bmatrix}, \boldsymbol{W} \right).$$

The former represents error in our prediction, such as the robot not actually moving the exact amount commanded, while the latter represents errors in measurement, such as sensor noise. We assume the nominal case of no systematic bias, so the mean terms are 0. The variance matrices $\boldsymbol{V}$ and $\boldsymbol{W}$ will be used when updating the state, and are set manually by the designer. In this demo, I use the following values.

$$\boldsymbol{V} = \begin{bmatrix} 0.02^2 & 0 \\ 0 & \left(\frac{\pi}{360}\right)^2 \end{bmatrix}$$
$$\boldsymbol{W} = \begin{bmatrix} 0.1^2 & 0 \\ 0 & \left(\frac{\pi}{180}\right)^2 \end{bmatrix}$$

With all requisites now defined, we can enter the EKF loop.

3.a Predict Stage

In the predict stage, we receive the command for the current timestep,

$$\boldsymbol{u}_t = \begin{bmatrix} u_d \\ u_\theta \end{bmatrix}$$

and can use this to predict the state mean at the next timestep,

$$\boldsymbol{x}_{pred} = f(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{v}) = \begin{bmatrix} x_v + (u_d + v_d)\cos(\theta_v) \\ y_v + (u_d + v_d)\sin(\theta_v) \\ \theta_v + u_\theta + v_\theta \end{bmatrix}.$$

To propagate the state covariance, we need to compute some Jacobian matrices to handle the nonlinearity caused by including θ in the state.

$$\boldsymbol{F}_x = \frac{\partial f}{\partial \boldsymbol{x}} = \begin{bmatrix} 1 & 0 & -u_d\sin(\theta_v) \\ 0 & 1 & u_d\cos(\theta_v) \\ 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{F}_v = \frac{\partial f}{\partial \boldsymbol{v}} = \begin{bmatrix} \cos(\theta_v) & 0 \\ \sin(\theta_v) & 0 \\ 0 & 1 \end{bmatrix}$$

The propagated state covariance is then calculated as

$$\boldsymbol{P}_{pred} = \boldsymbol{F}_x \cdot \boldsymbol{P}_t \cdot \boldsymbol{F}_x^T + \boldsymbol{F}_v \cdot \boldsymbol{V} \cdot \boldsymbol{F}_v^T$$

3.b Update Stage

In this section, x_v, y_v, and θ_v refer to the elements of the predicted state mean, $\boldsymbol{x}_{pred}$.

Recall that our measurements will come in the form of an ID, range, and bearing for each landmark detected this timestep. If nothing is detected, the update stage is skipped, and we simply use our predictions to propagate the state for this timestep:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_{pred}$$
$$\boldsymbol{P}_{t+1} = \boldsymbol{P}_{pred}$$

If there is at least one landmark detection, we perform the update step for each in series. For one such measurement, then, we will have

$$\boldsymbol{z}_{t+1} = \begin{bmatrix} z_{id} \\ z_r \\ z_\beta \end{bmatrix}.$$

Since we're in the localization-only case, we assume that we know the true map. Let

$$\boldsymbol{p}_l = \begin{bmatrix} x_l \\ y_l \end{bmatrix}$$

be the known true position of the landmark denoted by z_{id} in the world frame.

We can compute the distance from our predicted vehicle position and the known landmark position,

$$d = \sqrt{(x_l - x_v)^2 + (y_l - y_v)^2},$$

and use this to estimate the measurement we would have gotten if our state estimate is completely correct,

$$\boldsymbol{z}_{est} = h(\boldsymbol{x}_{pred}, \boldsymbol{w}) = \begin{bmatrix} r_{est} \\ \beta_{est} \end{bmatrix} = \begin{bmatrix} d \\ \arctan\left(\frac{y_l - y_v}{x_l - x_v}\right) - \theta_v \end{bmatrix} + \begin{bmatrix} w_r \\ w_\beta \end{bmatrix},$$

where β_{est} should remain in the range $(-\pi, \pi)$. This can be achieved in Python 3 using the `remainder` function:

```
import math
beta_est = math.remainder(math.atan2(y_l - y_v,x_l - x_v) - theta_v, math.tau)
```

Using $\boldsymbol{z}_{est}$, we can compute the "innovation", which gives a sense of how the actual measurement differs from our expectation given our predicted state.

$$\nu = \begin{bmatrix} z_r \\ z_\beta \end{bmatrix} - \begin{bmatrix} r_{est} \\ \beta_{est} \end{bmatrix}$$

To proceed, we must compute some more Jacobian matrices to perform linearization.

$$\boldsymbol{H}_x = \frac{\partial h}{\partial \boldsymbol{x}} = \begin{bmatrix} -\frac{x_l - x_v}{d} & -\frac{y_l - y_v}{d} & 0 \\ \frac{y_l - y_v}{d^2} & -\frac{x_l - x_v}{d^2} & -1 \end{bmatrix}$$

$$\boldsymbol{H}_w = \frac{\partial h}{\partial \boldsymbol{w}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

We can now compute the Kalman Gain, which captures the ideal relative weights of our prediction vs the new measurement.

$$\boldsymbol{S} = \boldsymbol{H}_x \cdot \boldsymbol{P}_{pred} \cdot \boldsymbol{H}_x^T + \boldsymbol{H}_w \cdot \boldsymbol{W} \cdot \boldsymbol{H}_w^T$$

$$\boldsymbol{K} = \boldsymbol{P}_{pred} \cdot \boldsymbol{H}_x^T \cdot \boldsymbol{S}^{-1}$$

We can then update our prediction of the state mean and covariance:

$$\boldsymbol{x}_{pred} = \boldsymbol{x}_{pred} + \boldsymbol{K} \cdot \nu$$

$$\boldsymbol{P}_{pred} = \boldsymbol{P}_{pred} - \boldsymbol{K} \cdot \boldsymbol{H}_x \cdot \boldsymbol{P}_{pred}$$

We should also ensure that the vehicle's heading estimate θ_v stays within $(-\pi, \pi)$ using the same method described above for β_{est}.

After running this update process for each landmark detected this timestep, we update the state estimate.

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_{pred}$$

$$\boldsymbol{P}_{t+1} = \boldsymbol{P}_{pred}$$

At this point, we move to the next timestep and the loop goes back to the predict stage.

4 EKF for Landmark Mapping Only

In the mapping case, we assume the vehicle's true position is known for all timesteps. This is achieved in our architecture by assuming the odometry commands are exactly correct and correspond to the robot's true movement during that timestep. Our goal is to use this known position along with any measurements to produce a list of all landmarks and their positions, i.e., a map. Note that a landmark which is never detected by the robot will not appear in the map; this is fine, as in a real application we will either design our trajectory to intentionally observe all parts of the environment, or only need the map for local navigation and thus do not care about parts of the map that aren't near the robot.

The state in the mapping case is a vector of landmark positions,

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ y_1 \\ . \\ . \\ x_M \\ y_M \end{bmatrix}$$

We also keep a list of landmark IDs, $L = [id_1, ..., id_M]$. The position of an ID in L corresponds to its position in the state. When we begin, there are no landmarks in the state.

$$\boldsymbol{x}_0 = [\quad]$$
$$\boldsymbol{P}_0 = [\quad]$$

With each detection, we will either insert that landmark into the state, or update its entry if it already appears.

4.a Predict Stage

For the mapping-only case, there is no process noise. We know the robot's true position at all times, and we know that all landmarks are static. Thus, the prediction stage contains only a single equation which keeps track of the vehicle position given the known odometry for that timestep.

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{v}) = \begin{bmatrix} x_v + u_d \cos(\theta_v) \\ y_v + u_d \sin(\theta_v) \\ \theta_v + u_\theta \end{bmatrix}$$

The state "prediction" simply initializes $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t$, since we predict landmarks will remain where they are.

4.b Update Stage

If no landmarks are detected, we skip straight back to the prediction stage for the next timestep. If one or more landmarks are detected, we first need to check if each landmark has been seen before, or needs to be

added to the state. A detection for one landmark takes the same form as previously discussed,

$$\boldsymbol{z}_{t+1} = \begin{bmatrix} z_{id} \\ z_r \\ z_\beta \end{bmatrix}.$$

4.b.i Update - Existing Landmark

If z_{id} appears in L, we have seen this landmark before and added it to the state. Let the index of z_{id} in L be denoted i. We know then that our current estimate of this landmark's position is at the $2i$ and $2i+1$ entries in the state.

$$\boldsymbol{p}_l = \begin{bmatrix} \boldsymbol{x}_t[2i] \\ \boldsymbol{x}_t[2i+1] \end{bmatrix} = \begin{bmatrix} x_l \\ y_l \end{bmatrix}$$

Now we can perform the same steps as in the localization case to update our belief about this landmark. First we compute the distance from the known robot position to where we currently believe that landmark to be,

$$d = \sqrt{(x_l - x_v)^2 + (y_l - y_v)^2},$$

and use this to estimate the measurement we would have gotten if our state estimate is correct,

$$\boldsymbol{z}_{est} = h(\boldsymbol{x}_t, \boldsymbol{w}) = \begin{bmatrix} r_{est} \\ \beta_{est} \end{bmatrix} = \begin{bmatrix} d \\ \arctan\left(\frac{y_l - y_v}{x_l - x_v}\right) - \theta_v \end{bmatrix} + \begin{bmatrix} w_r \\ w_\beta \end{bmatrix},$$

where β_{est} should remain in the range $(-\pi, \pi)$ as discussed in the previous section. This allows us to compute the innovation,

$$\nu = \begin{bmatrix} z_r \\ z_\beta \end{bmatrix} - \begin{bmatrix} r_{est} \\ \beta_{est} \end{bmatrix}.$$

Now we compute some Jacobian matrices:

$$\boldsymbol{H}_p = \frac{\partial h}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{x_l - x_v}{d} & \frac{y_l - y_v}{d} \\ -\frac{y_l - y_v}{d^2} & \frac{x_l - x_v}{d^2} \end{bmatrix}$$

$$\boldsymbol{H}_w = \frac{\partial h}{\partial \boldsymbol{w}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Since this will be multiplied with the state covariance, we need to make sure its dimensions match while still only affecting the current landmark in question. We do this by padding it with zeros. The length of the state is $2M$, where M is the number of landmarks stored. Recall that i is the index of this landmark in L.

$$\boldsymbol{H}_x = \begin{bmatrix} \boldsymbol{0}_{2\times(2i-2)} & \boldsymbol{H}_p & \boldsymbol{0}_{2\times(2M-2i)} \end{bmatrix}$$

We can then use this to compute the Kalman Gain and update the state.

$$\boldsymbol{S} = \boldsymbol{H}_x \cdot \boldsymbol{P}_t \cdot \boldsymbol{H}_x^T + \boldsymbol{H}_w \cdot \boldsymbol{W} \cdot \boldsymbol{H}_w^T$$
$$\boldsymbol{K} = \boldsymbol{P}_t \cdot \boldsymbol{H}_x^T \cdot \boldsymbol{S}^{-1}$$
$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_{t+1} + \boldsymbol{K} \cdot \nu$$
$$\boldsymbol{P}_{t+1} = \boldsymbol{P}_{t+1} - \boldsymbol{K} \cdot \boldsymbol{H}_x \cdot \boldsymbol{P}_{t+1}$$

4.b.ii Update - New Landmark

To add a new landmark, we first append z_{id} to the end of L.

$$L = \begin{bmatrix} L & z_{id} \end{bmatrix}$$

We then use our detection to compute the global position of the landmark, and append it to the end of the state mean.

$$g(\boldsymbol{x}, \boldsymbol{z}) = \begin{bmatrix} x_v + z_r \cos(\theta_v + z_\beta) \\ y_v + z_r \sin(\theta_v + z_\beta) \end{bmatrix}$$
$$\boldsymbol{x}_{t+1} = \begin{bmatrix} \boldsymbol{x}_t \\ g(\boldsymbol{x}, \boldsymbol{z}_{t+1}) \end{bmatrix}$$

To update the covariance, we need to compute an "insertion Jacobian", $\boldsymbol{Y}_z$.

$$\boldsymbol{G}_z = \begin{bmatrix} \cos(\theta_v + z_\beta) & -z_r \sin(\theta_v + z_\beta) \\ \sin(\theta_v + z_\beta) & z_r \cos(\theta_v + z_\beta) \end{bmatrix}$$
$$\boldsymbol{Y}_z = \begin{bmatrix} \boldsymbol{I}_{2M \times 2M} & \boldsymbol{0}_{2M \times 2} \\ \boldsymbol{0}_{2 \times 2M} & \boldsymbol{G}_z \end{bmatrix}$$

We can now update our covariance.

$$\boldsymbol{P}_{t+1} = \boldsymbol{Y}_z \cdot \begin{bmatrix} \boldsymbol{P}_{t+1} & \boldsymbol{0}_{2M \times 2} \\ \boldsymbol{0}_{2 \times 2M} & \boldsymbol{W} \end{bmatrix} \cdot \boldsymbol{Y}_z^T$$

5 EKF for Localization and Mapping (SLAM)

Now that we have derived the EKF for both localization and mapping separately, we can combine them without too much effort. Note that this implementation of EKF-SLAM will estimate the current vehicle pose as well as the full map, but will **not** retroactively estimate all previous vehicle poses. That wider problem is referred to as "Full SLAM" or "Pose-Graph SLAM". We are doing an easier problem called "Online SLAM", since we run it online to track the best possible estimate of the *current* state only.

We use the same process noise $\boldsymbol{v}$ and measurement noise $\boldsymbol{w}$ as before. Our initial state mean and covariance will be a combination of those for both localization and mapping, which is trivial as they were

initially empty in the latter.

$$\boldsymbol{x}_0 = \begin{bmatrix} x_v \\ y_v \\ \theta_v \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\boldsymbol{P}_0 = \begin{bmatrix} 0.01^2 & 0 & 0 \\ 0 & 0.01^2 & 0 \\ 0 & 0 & 0.005^2 \end{bmatrix}$$

The state will grow to contain both the vehicle pose and all landmark estimates, and will have length $3+2M$.

$$\boldsymbol{x}_t = \begin{bmatrix} x_v \\ y_v \\ \theta_v \\ x_1 \\ y_1 \\ . \\ . \\ x_M \\ y_M \end{bmatrix}$$

We will track landmark IDs in the same way as in EKF mapping, using a list $L = [id_1, ..., id_M]$.

5.a Predict Stage

Similarly to EKF localization, we receive the command for the current timestep,

$$\boldsymbol{u}_t = \begin{bmatrix} u_d \\ u_\theta \end{bmatrix}$$

and use it to predict the state mean at the next timestep. However, we need to account for the fact that the state now potentially includes more than just the 3×1 vehicle pose. Recall that our prediction for all landmark estimates is that they will remain constant.

$$\boldsymbol{x}_{pred} = f(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{v}) = \begin{bmatrix} x_v + (u_d + v_d)\cos(\theta_v) \\ y_v + (u_d + v_d)\sin(\theta_v) \\ \theta_v + u_\theta + v_\theta \\ x_1 \\ y_1 \\ . \\ . \\ x_M \\ y_M \end{bmatrix}.$$

To propagate the state covariance, we need to compute some Jacobians.

$$\boldsymbol{F}_{xv} = \begin{bmatrix} 1 & 0 & -u_d \sin(\theta_v) \\ 0 & 1 & u_d \cos(\theta_v) \\ 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{F}_x = \frac{\partial f}{\partial \boldsymbol{x}} = \begin{bmatrix} \boldsymbol{F}_{xv} & \boldsymbol{0}_{3\times 2M} \\ \boldsymbol{0}_{2M\times 3} & \boldsymbol{I}_{2M\times 2M} \end{bmatrix}$$

$$\boldsymbol{F}_{vv} = \begin{bmatrix} \cos(\theta_v) & 0 \\ \sin(\theta_v) & 0 \\ 0 & 1 \end{bmatrix}$$

$$\boldsymbol{F}_v = \frac{\partial f}{\partial \boldsymbol{v}} = \begin{bmatrix} \boldsymbol{F}_{vv} \\ \boldsymbol{0}_{2M\times 2} \end{bmatrix}$$

The sub-entries $\boldsymbol{F}_{xv}$ and $\boldsymbol{F}_{vv}$ exactly match the Jacobians in EKF localization; we have simply padded them to the correct dimensionality for EKF SLAM. The propagated state covariance is then calculated as

$$\boldsymbol{P}_{pred} = \boldsymbol{F}_x \cdot \boldsymbol{P}_t \cdot \boldsymbol{F}_x^T + \boldsymbol{F}_v \cdot \boldsymbol{V} \cdot \boldsymbol{F}_v^T$$

5.b Update Stage

In this section, x_v, y_v, and θ_v refer to the vehicle pose in the predicted state mean, $\boldsymbol{x}_{pred}$.

If no landmarks are detected, the update stage is skipped, and we simply use our predictions to propagate the state for this timestep:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_{pred}$$
$$\boldsymbol{P}_{t+1} = \boldsymbol{P}_{pred}$$

If one or more landmarks are detected, we first need to check if each landmark has been seen before, or needs to be added to the state. A detection for one landmark takes the same form as previously discussed,

$$\boldsymbol{z}_{t+1} = \begin{bmatrix} z_{id} \\ z_r \\ z_\beta \end{bmatrix}.$$

5.b.i Update - Existing Landmark

If z_{id} appears in L, we have seen this landmark before and added it to the state. Let the index of z_{id} in L be denoted i. We know then that our current estimate of this landmark's position is at the $3+2i$ and $3+2i+1$ entries in the state.

$$\boldsymbol{p}_l = \begin{bmatrix} \boldsymbol{x}_t[3+2i] \\ \boldsymbol{x}_t[3+2i+1] \end{bmatrix} = \begin{bmatrix} x_l \\ y_l \end{bmatrix}$$

Now we follow the same procedure as before to update our belief about this landmark. First we compute the distance from the predicted robot position to the estimated landmark position,

$$d = \sqrt{(x_l - x_v)^2 + (y_l - y_v)^2},$$

and use this to estimate the measurement we would have gotten if our state estimate is correct,

$$\boldsymbol{z}_{est} = h(\boldsymbol{x}_t, \boldsymbol{w}) = \begin{bmatrix} r_{est} \\ \beta_{est} \end{bmatrix} = \begin{bmatrix} d \\ \arctan\left(\frac{y_l - y_v}{x_l - x_v}\right) - \theta_v \end{bmatrix} + \begin{bmatrix} w_r \\ w_\beta \end{bmatrix},$$

where β_{est} should remain in the range $(-\pi, \pi)$ as discussed previously. This allows us to compute the innovation,

$$\nu = \begin{bmatrix} z_r \\ z_\beta \end{bmatrix} - \begin{bmatrix} r_{est} \\ \beta_{est} \end{bmatrix}.$$

Now we compute some Jacobian matrices. Again the submatrices will look familiar, and we arrange their placement in larger matrices to fit all dimensionality correctly.

$$\boldsymbol{H}_{xv} = \begin{bmatrix} -\frac{x_l - x_v}{d} & -\frac{y_l - y_v}{d} & 0 \\ \frac{y_l - y_v}{d^2} & -\frac{x_l - x_v}{d^2} & -1 \end{bmatrix}$$

$$\boldsymbol{H}_{xp} = \begin{bmatrix} \frac{x_l - x_v}{d} & \frac{y_l - y_v}{d} \\ -\frac{y_l - y_v}{d^2} & \frac{x_l - x_v}{d^2} \end{bmatrix}$$

$$\boldsymbol{H}_x = \frac{\partial h}{\partial \boldsymbol{x}} = \begin{bmatrix} \boldsymbol{H}_{xv} & \boldsymbol{0}_{2\times(2i-2)} & \boldsymbol{H}_{xp} & \boldsymbol{0}_{2\times(2M-2i)} \end{bmatrix}$$

$$\boldsymbol{H}_w = \frac{\partial h}{\partial \boldsymbol{w}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

We can then use these to compute the Kalman Gain and update the state.

$$\boldsymbol{S} = \boldsymbol{H}_x \cdot \boldsymbol{P}_t \cdot \boldsymbol{H}_x^T + \boldsymbol{H}_w \cdot \boldsymbol{W} \cdot \boldsymbol{H}_w^T$$

$$\boldsymbol{K} = \boldsymbol{P}_t \cdot \boldsymbol{H}_x^T \cdot \boldsymbol{S}^{-1}$$

$$\boldsymbol{x}_{pred} = \boldsymbol{x}_{pred} + \boldsymbol{K} \cdot \nu$$

$$\boldsymbol{P}_{pred} = \boldsymbol{P}_{pred} - \boldsymbol{K} \cdot \boldsymbol{H}_x \cdot \boldsymbol{P}_{pred}$$

5.b.ii Update - New Landmark

To add a new landmark, we first append z_{id} to the end of L.

$$L = \begin{bmatrix} L & z_{id} \end{bmatrix}$$

We then use our detection to compute the position of the landmark in the world frame and append it to the end of the state mean.

$$g(\boldsymbol{x}, \boldsymbol{z}) = \begin{bmatrix} x_v + z_r \cos(\theta_v + z_\beta) \\ y_v + z_r \sin(\theta_v + z_\beta) \end{bmatrix}$$

$$\boldsymbol{x}_{pred} = \begin{bmatrix} \boldsymbol{x}_{pred} \\ g(\boldsymbol{x}, \boldsymbol{z}_{t+1}) \end{bmatrix}$$

Now we compute the insertion Jacobian, $\boldsymbol{Y}_z$.

$$\boldsymbol{G}_z = \begin{bmatrix} \cos(\theta_v + z_\beta) & -z_r \sin(\theta_v + z_\beta) \\ \sin(\theta_v + z_\beta) & z_r \cos(\theta_v + z_\beta) \end{bmatrix}$$

$$\boldsymbol{Y}_z = \begin{bmatrix} \boldsymbol{I}_{(3+2M)\times(3+2M)} & \boldsymbol{0}_{(3+2M)\times 2} \\ \boldsymbol{0}_{2\times(3+2M)} & \boldsymbol{G}_z \end{bmatrix}$$

We can then update the covariance.

$$\boldsymbol{P}_{pred} = \boldsymbol{Y}_z \cdot \begin{bmatrix} \boldsymbol{P}_{pred} & \boldsymbol{0}_{(3+2M)\times 2} \\ \boldsymbol{0}_{2\times(3+2M)} & \boldsymbol{W} \end{bmatrix} \cdot \boldsymbol{Y}_z^T$$

5.b.iii Update - End

After running this update process for each landmark detected this timestep, we update the state estimate.

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_{pred}$$
$$\boldsymbol{P}_{t+1} = \boldsymbol{P}_{pred}$$

At this point, we move to the next timestep and the loop goes back to the predict stage.

6 Demo

6.a Map Creation

As discussed, a map is completely described by a dictionary mapping landmark IDs to positions. We can thus generate a random map by specifying some paramaters.

- `num_landmarks`.
- `bound`. All landmarks will lie within $\pm$`bound` in both x and y.
- `min_sep`. No landmarks will be within `min_sep` of another landmark.

We can then create a map by randomly generating coordinates within bounds and saving or rejecting them based on `min_sep` until `num_landmarks` has been reached. For simplicity, the IDs are the integers 0 to `num_landmarks` $- 1$.

6.b Trajectory Generation

The trajectory could be created in any number of ways. The options I considered are the following.

- Have a user control the vehicle live as the EKF runs.
- Use the same pre-created trajectory for every run.
- Randomize the odom command for all timesteps.
- Generate a trajectory that will come close to all landmarks at least once.

I've implemented all of these and find the last one to be the most interesting, so that's what I'll carry on with for the demo. I did this by treating the map as a travelling salesman problem (TSP); this is basically a general approach in graph theory for creating a path that visits all nodes in some graph. The simplest solution to the TSP is the nearest-neighbors heuristic. This strategy entails choosing the nearest node to the current node that has not yet been visited, and choosing to go there next. This continues until all nodes have been added to a list. This is usually not the most efficient TSP solution, but since our goal is to just move around the map, it will suffice.

My trajectory generator creates such a list for the map that has been provided, and then generates odometry commands within constraints to get within some threshold of each landmark. When a landmark is "visited", it is moved to the end of the list; this way, the vehicle will continue moving until time runs out, rather than stopping once it has visited all landmarks.

This strategy helps to minimize the uncertainty in the EKF estimates, since the uncertainty grows every timestep and can only shrink when a landmark is detected. The trivial approach to truly minimize uncertainty would be to find a landmark and then sit stationary in front of it for the duration, but this isn't very interesting.

We track the true position for the entire trajectory (since it's needed for further planning and for generating measurements) and also save a noisy version of all odometry commands that has the process noise $\boldsymbol{v}$ applied to it. This ensures the EKF isn't being fed perfect data, and brings this simulation closer to reality. The noisy commands are published one at a time to the EKF at some specified frequency `DT`.

6.c Measurement Generation

Generating measurements given the map and true vehicle position is fairly straightforward. We compare all landmark positions to the robot's to get a distance and bearing to each. We then reject all landmarks whose range is further than the robot's maximum vision range or whose bearing is outside the robot's constrained FOV. For all suitable landmarks that satisfy both vision constraints, their ID, range, and bearing are saved and obfuscated with the measurement noise $\boldsymbol{w}$. These are then published for the EKF one timestep at a time at a specified frequency `DT`.

6.d Plot

I've created a dedicated `plotting_node` which shows a live view of the ground truth map and trajectory overlaid with the EKF's current estimate for the map and the vehicle pose. This is the best way to get an idea of the effectiveness of the EKF for Landmark SLAM, and see how this effectiveness may change depending on the random layout of the map and the trajectory.

put in some plots, links to running the thing, link to video(s)