Landmark-Based EKF-SLAM Derivation and Demo

Kevin Robb

Contents

1 Introduction to the Simulator

I've created a rudimentary simulator entirely from scratch in Robot Operating System (ROS Noetic) for this project. In order to work with any filter or localization method that I want to test/demo, I designed it to contain several "nodes" which each handle an important part of the simulation; these nodes communicate with each other by publishing or subscribing to certain topics, and all nodes run simultaneously. A node graph of the EKF-SLAM demo follows.

EKF-SLAM node graph showing topics and nodes.

Some base scripts/nodes run in every demo to setup the structure and generate data for the filter being examined. All filters have the option to subscribe to the following:

- Control Commands:
  - Forward velocity.
  - Angular velocity (radians).
- Measurements; for all landmarks within range/FOV of vehicle's camera:
  - Landmark's unique ID.
  - Range to landmark from vehicle position.
  - Bearing to landmark relative to vehicle orientation (radians).
- True Landmark Map, for localization-only filters.
- True Vehicle Pose, for mapping-only filters.
- Occupancy Grid Map.

For instance, the EKF-SLAM node will subscribe to the control commands and the landmark measurements, and will publish its current estimate of the "state", which includes the estimated vehicle pose and estimated landmark positions.

In this 2D simulation, 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, which is in the range $[-\pi, \pi]$, with $\theta = 0$ being to the right. The world frame can be defined however we like; for simplicity, we usually assume the robot's starting pose is the origin, but in my architecture I define a global "map" coordinate system w.r.t. which the robot's initial pose is specified. This helps the plots to keep everything visible and centralized.

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 could be dedicated to using real-world data of some kind to generate abstract landmark measurements of this form. Thus, our measurement $\boldsymbol{z}$ at some timestep will be zero, one, or more $\boldsymbol{z}_{t,id} = \begin{bmatrix} id & r & \beta \end{bmatrix}^T$, where id specifies the landmark that has been detected. Any number of landmarks can be detected on a single timestep, and the measurements for all are concatenated together to be used by the filter.

We have a second map in the form of a color image that will be thresholded to form a binary occupancy grid. This is used for path-planning, but does not obstruct vision or prevent vehicle motion. A more robust

simulation would potentially handle collisions and perform raycasting to determine line-of-sight, but these details were not necessary for the goals of this project (so far).

The robot can be controlled once per timestep with a certain odometry, consisting of a forward velocity and angular velocity, resulting in that distance being achieved in the next timestep (For simplicity in the math, we assume a timestep is one "time unit" in length, so $d = v \cdot t = v$). We call this command $\boldsymbol{u}$ by convention. Thus, the command for some timestep t will be $\boldsymbol{u}_t = \begin{bmatrix} \mathrm{d}\, d & \mathrm{d}\, \theta \end{bmatrix}^T$. I've allowed for these commands to be generated in a few ways:

- The `sim_node` treats the landmark map as a "travelling salesman problem" (TSP), and generates a trajectory that forces the vehicle to come near all landmarks. With this method, all commands for the full trajectory are generated up-front, and fed to the simulation one at a time.
- The user clicks on the screen, and the vehicle directly pursues this point.
- The user clicks on the screen, and the `goal_pursuit_node` uses A* to find the optimal path there through free space, and navigates with Pure Pursuit.
- The `goal_pursuit_node` autonomously chooses goal points, and navigates to them.

After a command is chosen for a timestep, the `sim_node` enacts a noisy version of it on the "real" vehicle, determines what the robot measures given the true vehicle pose and true landmark positions, and publishes a noisy version of these measurements for the filters to use.

The robot has some constraints on the maximum possible velocities 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. I've made as many things modifiable as I could, in the `base_pkg/config/params.txt` of my codebase; options include changing noise profiles, plotting options, initial vehicle pose, and more.

2 Kalman Filtering Overview

A Kalman Filter (KF), or more generally a Bayes Filter, is simply an iterative process of estimating some state in the most effective way possible. Generally, the inputs include commands sent to the vehicle ($\boldsymbol{u}$), and a measurement obtained after the commanded motion is performed ($\boldsymbol{z}$). The output will be the current estimate of some state, which takes the form of some probability distribution. For most of these filters, we assume the state can be represented by a Gaussian distribution, so we track the state mean vector $\boldsymbol{x}$ and covariance matrix $\boldsymbol{P}$. The elements of $\boldsymbol{x}$ are all the variables we want to estimate, and $\boldsymbol{P}$ gives an idea of the uncertainty of the current state estimates; a more spread-out distribution shows less confidence than a highly-peaked one.

The KF infinitely loops back and forth between the "predict" and "update" stages to converge on the state distribution. Noise is always injected into the system during the prediction stage, and noise plays a part in the update stage as well. This is very robust, since it better represents the real system, but also means that we can never be certain about our state estimate. The main downside of the standard KF is that its formulation expects Gaussian noise profiles, a Gaussian state distribution, and a linear model for the predictions and measurements.

As evident from my simulator setup, the vehicle is described by not only a position, but an orientation (yaw) as well. This makes our system nonlinear, since the change in position on each timestep relies on the sine and cosine of the current yaw. Fortunately, we can still represent the state as a Gaussian distribution,

since each state variable's distribution is symmetric and unimodal about some mean. One method to estimate the state in such a system is the Extended Kalman Filter.

The Extended Kalman Filter (EKF) keeps the same general flow as the standard KF, but requires the calculation of several Jacobian matrices on each iteration; these are used to linearize the system about the current estimated mean, and carry out the state propagation *as if* it were a linear system. This works great in many situations, but will fail for highly-nonlinear systems whose behavior may change drastically with a slight change in input.

In place of the EKF, we can use an Unscented Kalman Filter (UKF). This filter avoids any linearization, and is able to accurately represent the nonlinear nature of a system. It can also do this with the same runtime complexity as the EKF. It does this by calculating a small set of so-called "sigma points". This is a set of weighted points in the same form as the state mean vector which characterize the state distribution. We can then feed all these points through the true, potentially nonlinear system model, and compute a new covariance matrix from these propagated sigma points. This bypasses the problem of passing the covariance matrix through a nonlinear system without simply linearizing the system, as the EKF does.

If we have a system in which even the state space cannot be represented by a Gaussian distribution, none of these filters can help us, as they all estimate the state as a mean and covariance. This realm includes the classic "kidnapped robot" problem, in which a robot has a known map and is equipped with some sensor, but does not know its initial position on that map. As the robot moves, it can narrow down the possibilities for its current pose until there is a concentrated, unimodal area of belief. We can represent this distribution as a collection of particles, forming the Particle Filter (PF). Any distribution can be characterized by a sufficient number of such particles. The PF process involves randomly distributing particles on the map, propagating them with the commands sent to the robot, and comparing the real measurement to what each particle would see if it were correct. Particles are weighted according to their likelihood, and through repeated resampling and propagation, eventually we hope that they will cluster around the true vehicle pose.

3 EKF for Localization

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 both C++ and Python3 using the `remainder` function with divisor $\tau = 2\pi$:

```
beta_est = remainder(atan2(y_l - y_v,x_l - x_v) - theta_v, 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

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.

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

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", $\boldsymbol{Y}_z$.

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

$$\boldsymbol{Y}_z = \begin{bmatrix} \boldsymbol{I}_{2M\times 2M} & \mathbf{0}_{2M\times 2} \\ \mathbf{0}_{2\times 2M} & \boldsymbol{G}_z \end{bmatrix}$$

We can now update our covariance.

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

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, $\boldsymbol{Y}_z$.

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

$$\boldsymbol{G}_x = \begin{bmatrix} 1 & 0 & -z_r \sin(\theta_v + z_\beta) \\ 0 & 1 & z_r \cos(\theta_v + z_\beta) \end{bmatrix}$$

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

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.

5.c Demo

In all plots, the empty circles are the true landmark positions, and the red circles are the EKF estimates. Orange ellipses, when shown, are the EKF covariances for each landmark estimate. The blue path is the true trajectory the vehicle follows for the total duration; if a blue path is not shown, the blue arrow shows the current true position. The green arrow shows the current EKF estimate for the vehicle pose, and the grey ellipse shows the current EKF covariance for the vehicle's position.

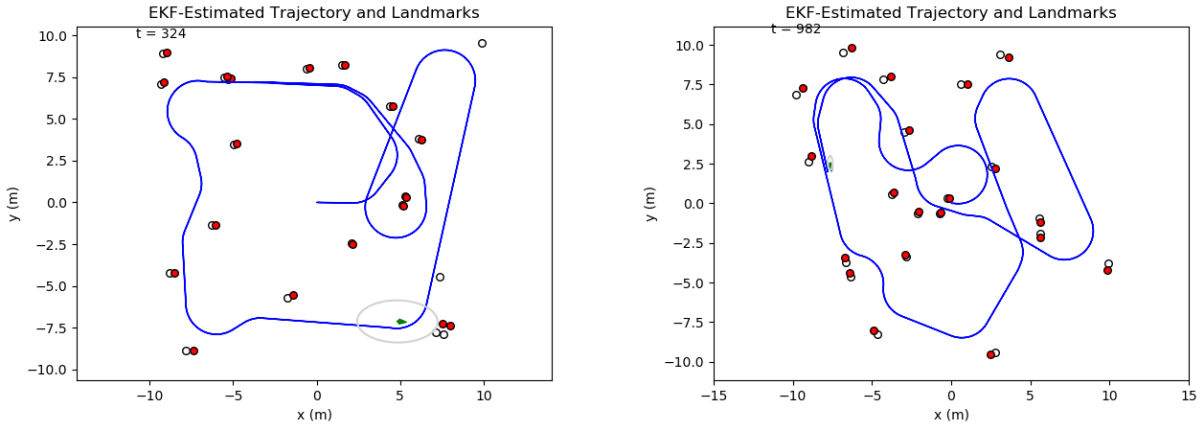

Figure 1: Performance with random maps.

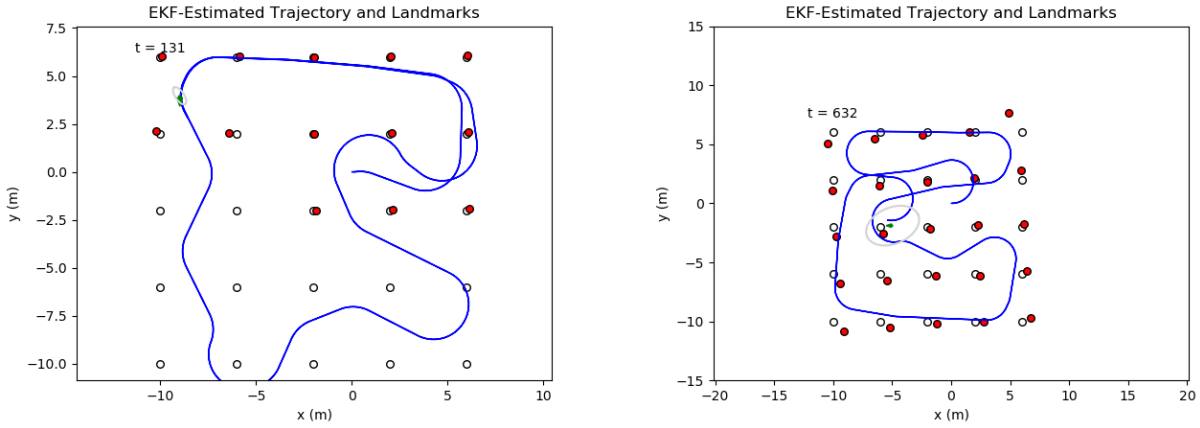

Figure 2: Performance with grid map.

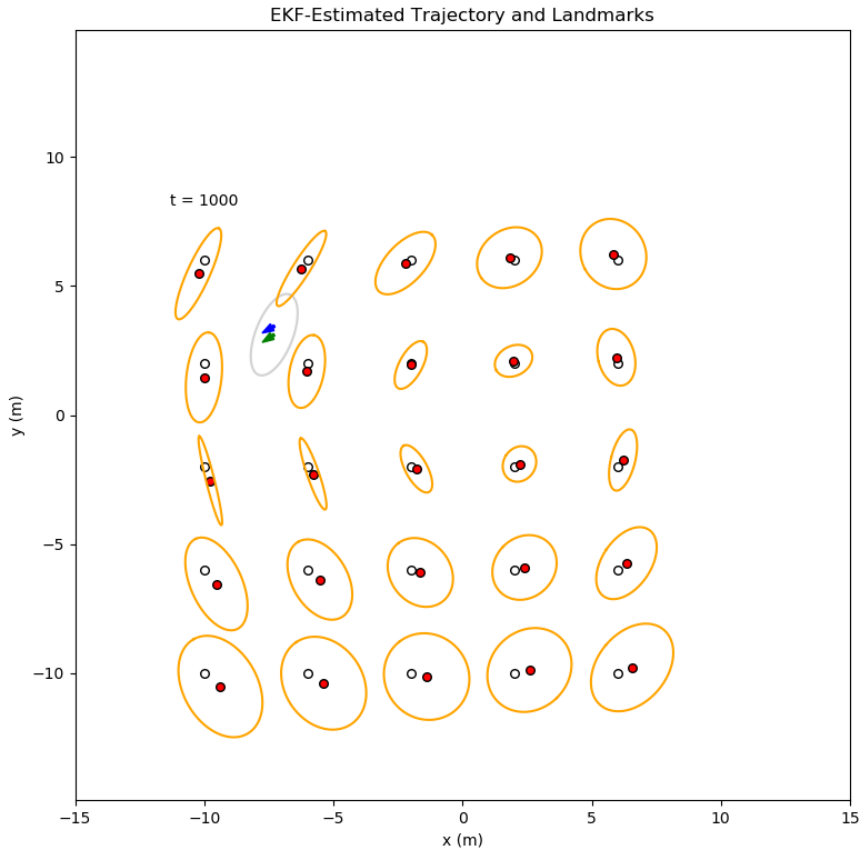

Figure 3: Performance with all covariances shown.

6 Unscented Kalman Filter (UKF)

6.a UKF Motivation

The Extended Kalman Filter introduced Jacobian matrices to linearize the system about the current mean at each iteration, so that even nonlinear systems can be estimated, as we've done in the previous sections. However, the state is still represented as a Gaussian random variable, and the linearized propagation technique only captures the first-order approximation of the nonlinear dynamics. This is the main drawback of the EKF, and can cause performance to fall off and even lead to divergence.

A different approach is the UKF, which entails choosing multiple "sigma points" to represent the characteristics of the state distribution, applying the nonlinear system to these points, and estimating a new distribution from the resulting set of points. If we think of the initial distribution as an infinite collection of points that has a particular mean and covariance, this process is analogous to down-sampling; we extract a set of $2n + 1$ points that have the same mean and covariance as the larger set. This way, when we propagate this small, finite set of points through the nonlinearity, we can convert it back into a Gaussian distribution very effectively, preserving much more of the distribution's characteristics than the

EKF's Jacobian methods. This is effectively accurate up to a third-order approximation of the system, where the EKF had only first-order accuracy with the same order of complexity.

This may sound similar to a Particle Filter or other form of Monte-Carlo Localization; the difference is that for MCL, we have no assumptions about the state distribution, and use many hundreds of particles for them to eventually converge on the true state without fear of particle depletion. The UKF can use a very small number of points, since we still assume the state itself can be represented as a Gaussian. We only apply propagation to these few vectors, rather than some massive swarm of particles, meaning the runtime is far quicker.

6.b Prediction Stage

The core innovation of the UKF is to represent the state distribution (normally done with a mean vector $\boldsymbol{x}_t$ and a covariance matrix $\boldsymbol{P}_t$) as a set of representative points. We will then be able to easily apply our true nonlinear model to this set of points, after which we must fit a new "posterior" state distribution to them.

So, given the "prior" n-dimensional state distribution is $\sim \mathcal{N}(\boldsymbol{x}_t, \boldsymbol{P}_t)$, we will compute a representative set of points and their weights. These will be the matrix $\mathcal{X} \in \mathbb{R}^{n\times(2n+1)}$ and vector $\mathcal{W}$. Each column is a $n \times 1$ vector in the same format as the state mean, and column $\mathcal{X}_i$ is the sigma vector whose corresponding weight is $\mathcal{W}_i$.

These columns are computed as the following:

$$\mathcal{X}_i = \begin{cases} \boldsymbol{x}_t & \text{for } i = 0 \\ \boldsymbol{x}_t + \left(\sqrt{\frac{n}{1-\mathcal{W}_0}\boldsymbol{P}_t}\right)_i & \forall i \in [1, n] \\ \boldsymbol{x}_t - \left(\sqrt{\frac{n}{1-\mathcal{W}_0}\boldsymbol{P}_t}\right)_{i-n} & \forall i \in [n+1, 2n] \end{cases}$$

$$\mathcal{W}_i = \frac{1-\mathcal{W}_0}{2n} \qquad \forall i \in [1, 2n]$$

where we choose $\mathcal{W}_0 \in [-1, 1]$ to set the spread of the sigma points' distribution. A lower value will keep points closer to the origin. For this project, I've chosen $\mathcal{W}_0 = 0.2$. Note that all weights must sum to 1.

Computing the $\left(\sqrt{\frac{n}{1-\mathcal{W}_0}\boldsymbol{P}_t}\right)$ matrix requires special attention; a matrix only has a square root if it is symmetric and positive-definite, which our covariance matrix is close to, but likely insufficient. In this step, then, we use the Frobenius norm formulation to find the closest matrix to $\boldsymbol{P}_t$ which satisfies these requirements. We do this with the following procedure:

1. Find the best *symmetric* matrix.
$$\boldsymbol{Y} = (\boldsymbol{P}_t + \boldsymbol{P}_t^T)/2$$
2. Multiply by the inner term for sigma points calculation.
$$\boldsymbol{Y} = \frac{n}{1-\mathcal{W}_0} \cdot \boldsymbol{Y}$$
3. Find the eigen decomposition of $\boldsymbol{Y}$ to obtain $\boldsymbol{D}$, a $n \times 1$ vector of eigenvalues, and $\boldsymbol{Q}_v$, an $n \times n$ matrix of the corresponding eigenvectors.
4. Cap all eigenvalues to be $\geq \varepsilon$ for some small $\varepsilon > 0$.
5. Convert the vector of eigenvalues into a diagonal matrix, $\boldsymbol{D}_+$.

6. Compute $\boldsymbol{Z}$, the nearest symmetric, positive semi-definite matrix to $\boldsymbol{P}_t$.

$$\boldsymbol{Z} = Q_v \cdot \boldsymbol{D}_+ \cdot \boldsymbol{Q}_v^T$$

In C++, I've implemented this in the following function.

```
// find the nearest symmetric positive (semi)definite matrix to P_t using Frobenius Norm.
Eigen::MatrixXd UKF::nearestSPD() {
    // first compute nearest symmetric matrix.
    this->Y = 0.5 * (this->P_t + this->P_t.transpose());
    // multiply by inner coefficient for UKF.
    this->Y *= (2*this->M+3)/(1-this->W_0);
    // compute eigen decomposition of Y.
    Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd> solver(this->Y);
    this->D = solver.eigenvalues();
    this->Qv = solver.eigenvectors();
    // cap eigenvalues to be strictly positive.
    this->Dplus = this->D.cwiseMax(0.00000001);
    // compute nearest positive semidefinite matrix.
    return this->Qv * this->Dplus.asDiagonal() * this->Qv.transpose();
}
```

Since we now have a symmetric, positive semi-definite covariance matrix, we can take its square root, which will also be a symmetric $n \times n$ matrix. Thus the terms such as $\left(\sqrt{\frac{n}{1-\mathcal{W}_0}\boldsymbol{Z}}\right)_i$ can refer to either the ith row or ith column of the resulting square root matrix, whose elements are identical.

Recall that our system dynamics are contained in the nonlinear function f (i.e., our motion model in the prediction step). We use the same SLAM model as with the EKF, and the same control command $\boldsymbol{u} = \begin{bmatrix} u_d & u_\theta \end{bmatrix}^T$ as input.

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

Rather than computing the Jacobians $\boldsymbol{F}_x$ and $\boldsymbol{F}_v$, we directly propagate our sigma vectors through f to obtain $\boldsymbol{\mathcal{X}}_{pred} \in \mathbb{R}^{n \times (2n+1)}$, where each column is $\boldsymbol{\mathcal{X}}_{pred,i} = f(\boldsymbol{\mathcal{X}}_i)$. We can then compute the mean and covariance of this new distribution:

$$\boldsymbol{x}_{pred} = \sum_{i=0}^{2n} \mathcal{W}_i \cdot \boldsymbol{\mathcal{X}}_{pred,i}$$

$$\boldsymbol{P}_{pred} = \sum_{i=0}^{2n} \mathcal{W}_i \cdot (\boldsymbol{\mathcal{X}}_{pred,i} - \boldsymbol{x}_{pred}) \cdot (\boldsymbol{\mathcal{X}}_{pred,i} - \boldsymbol{x}_{pred})^T + \boldsymbol{V}$$

A new problem to note is this weighted averaging when applied to the yaw component. Recall that $\theta \in$

$[-\pi, \pi]$; Thus, if one sigma vector is close to π, and a second is close to $-\pi$, their average should point to the left, but a direct average will yield $\theta = (\pi + -\pi)/2 = 0$, pointing to the right. To avoid this problem, all averages involving angles are performed on their representations as vectors in the complex plane, and the final result is converted back to radians. i.e., the state mean calculation is implemented as follows.

```
this->x_pred.setZero(n);
this->complex_angle.setZero(2);
for (int i=0; i<2*n+1; ++i) {
    this->x_pred += this->Wts(i) * this->X_pred.col(i);
    // convert angles to vectors in complex plane to average them correctly (assume unit circle).
    this->complex_angle(0) += this->Wts(i) * cos(this->X_pred.col(i)(2)); // real component.
    this->complex_angle(1) += this->Wts(i) * sin(this->X_pred.col(i)(2)); // imaginary component.
}
// convert averaged heading back from complex to angle. also cap w/in (-pi, pi).
this->x_pred(2) = remainder(atan2(this->complex_angle(1), this->complex_angle(0)), 2*pi);
```

This slight inelegance can be avoided by directly representing these complex vector components in the state, rather than tracking the yaw itself.

6.c Update Stage

We also use the same measurements $\boldsymbol{z}_{id} = \begin{bmatrix} z_r & z_\beta \end{bmatrix}^T$ as in EKF-SLAM. For each landmark detection, we check if it's being tracked in our state; if not, this is the first detection, and it must be inserted into our state. We have a separate function for each mode. On one timestep, we could potentially have several of each type of detection; for simplicity, we perform all landmark updates first, and then perform all landmark insertions.

6.c.i Landmark Update

This step is used when the landmark detected is already being estimated in our state. If we're running the UKF in localization-only mode (meaning we have access to the true landmark map), this step is also used.

The nonlinear observation model h estimates the measurement we'd expect if our current state prediction were correct.

$$h(\boldsymbol{x}_t, \boldsymbol{w}) = \begin{bmatrix} r_{est} \\ \beta_{est} \end{bmatrix} = \begin{bmatrix} \sqrt{(x_l - x_v)^2 + (y_l - y_v)^2} + w_r \\ \arctan\left(\frac{y_l - y_v}{x_l - x_v}\right) - \theta_v + w_\beta \end{bmatrix}$$

We will run each propagated sigma vector through this sensing model to obtain $\boldsymbol{\mathcal{X}}_{zest} \in \mathbb{R}^{2\times(2n+1)}$, where each column is $\boldsymbol{\mathcal{X}}_{zest,i} = h(\boldsymbol{\mathcal{X}}_{pred,i})$.

We can then compute our overall measurement estimate

$$\boldsymbol{z}_{est} = \sum_{i=0}^{2n} \mathcal{W}_i \cdot \boldsymbol{\mathcal{X}}_{zest,i}.$$

Similarly, we must average the β_{est} using their complex vector representations, and cap the result in $[-\pi, \pi]$. We then compute covariance matrices from these sigma points: the innovation covariance, $\boldsymbol{S}$, and the cross-

covariance between $\boldsymbol{x}_{pred}$ and $\boldsymbol{z}_{est}$, denoted $\boldsymbol{C}$.

$$\boldsymbol{S} = \sum_{i=0}^{2n} \mathcal{W}_i \cdot (h(\mathcal{X}_i) - \boldsymbol{z}_{est}) \cdot (h(\mathcal{X}_i) - \boldsymbol{z}_{est})^T + \boldsymbol{W}$$
$$\boldsymbol{C} = \sum_{i=0}^{2n} \mathcal{W}_i \cdot (f(\mathcal{X}_i) - \boldsymbol{x}_{pred}) \cdot (h(\mathcal{X}_i) - \boldsymbol{z}_{est})^T$$

We can then compute the Kalman Gain,

$$\boldsymbol{K} = \boldsymbol{C} \cdot \boldsymbol{S}^{-1}.$$

And now we can finish the update step by computing the posterior distribution.

$$\boldsymbol{x}_{pred} = \boldsymbol{x}_{pred} + \boldsymbol{K} \cdot (\boldsymbol{z} - \boldsymbol{z}_{est})$$
$$\boldsymbol{P}_{pred} = \boldsymbol{P}_{pred} - \boldsymbol{K} \cdot \boldsymbol{S} \cdot \boldsymbol{K}^T$$

We update the predicted state for each landmark, and after all updates/insertions have completed, we finalize the state estimate.

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

6.c.ii Landmark Insertion

Inserting a new landmark into the state is fairly simple; since this new measurement is the only information we have about the new landmark, we simply use it as our estimate. The process is much more straightforward than in EKF-SLAM, since no linearization needs to take place.

Compute the position of the landmark in the world frame (according to this new measurement) and append it to the end of the state mean.

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

Expand the covariance matrix with the sensing noise covariance, $\boldsymbol{W}$.

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

Expand the process noise covariance matrix, since it's additive in the UKF and needs to be the same dimensions as $\boldsymbol{P}$. We don't want to inject noise into all landmark estimates on every prediction step, so the noise covariance for all added rows/columns will be zeros.

$$\boldsymbol{V} = \begin{bmatrix} \boldsymbol{V} & \boldsymbol{0}_{(3+2M)\times 2} \\ \boldsymbol{0}_{2\times(3+2M)} & \boldsymbol{0}_{2\times 2} \end{bmatrix}$$

Lastly, increment the number of landmarks, $M = M + 1$.

After all updates/insertions have completed for this timestep, we finalize the state estimate.

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

6.d Demo

The UKF is very effective in my simulator in both localization mode and SLAM mode. It has a slight stutter when facing directly to the left, due to the previously-discussed averaging problem for the yaw, but this comes down to mistakes in the code and/or my formulation of the state, not in the general UKF equations.

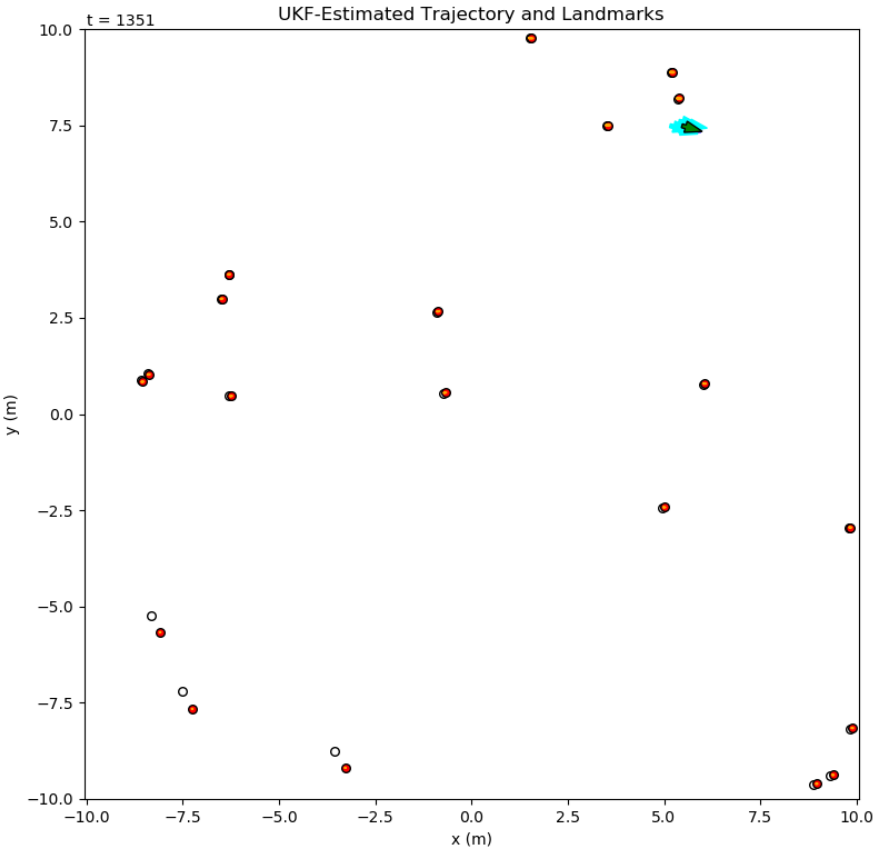

Figure 4: UKF performing online SLAM.