Error-State Extended Kalman Filter for AV Localization

David Tarazi

March 29, 2022

1 Background

Vehicle localization is a crucial component in developing a fully autonomous vehicle system. Localization includes finding the position and orientation of a robot in the world and enables the vehicle to reference its surroundings in the world with accurate positions. This concept is important because localization can be combined with maps and information about the world to trigger warnings and early avoidance for autonomous vehicles. Consider an autonomous vehicle driving down the highway in the right lane and a potentially dangerous merge is coming up. With accurate localization and information about the world, the vehicle and avoid this potentially dangerous situation before it happens. In the same way that human drivers are more comfortable when driving in a known environment based on past experiences, localization and mapping enables autonomous vehicles to achieve a similar increase in safety.

1.1 Technical Context

In order to perform localization, different sensors provide various information about the surrounding environment or data that can be used in order to determine where the vehicle is. Consider GNSS, or satellite positioning, where a sensor on the vehicle receives positioning information from satellite systems. While GNSS can be somewhat accurate, it cannot provide the level of accuracy on its own for a vehicle to know exactly which lane on a highway it's in. Furthermore, GNSS fails in tunnels and can be less accurate in cities where there are more reflections in the signal before reaching the sensor. One other sensor that can be used in localization is an IMU, which tracks the linear and angular acceleration of the vehicle. While these measurements and recorded very often and can be useful to determine position over short periods of time, error stacks over time and causes massive drift over long periods of time. LIDAR is a sensor that scans the environment and calculates a 3D position of each laser that is sent into the environment at different angles. Using algorithms to match points between different LIDAR scans, LIDAR can be an effective sensor even in the dark to measure how a vehicle has moved between time steps, though it's not perfect. While there are many other sensors including various types of cameras that can provide information for localization, no single sensor can accurately localize a vehicle in the environment. As a result, we need a way to combine data from different sensors and determine the best possible estimate for the vehicle's current position and orientation.

The Kalman Filter (KF) is known to be one of the most effective ways to combine data from different sensors and can be used to localize vehicles. The Linear Kalman Filter in the context of state estimation predicts the state (position, velocity, orientation, etc.) based on a motion model and sensors such as an IMU or wheel encoders. However, since IMU and wheel encoder readings can stack error quickly, the KF then updates and corrects the predictions using a measurement model when data from other sensors such as GNSS signals arrive to negate those stacking prediction errors. The KF treats both the predictions from the motion model and the measurement model as Gaussian distributions where the predicted state is the predicted mean with some varaince from noise in the model and sensors that is also modeled in the filter. If the motion model has a very high variance, then the filter will rely more heavily on the measurement values, but if there is a high variance on the measurements, then the filter will rely more heavily on the motion model. While the Linear Kalman Filter is great, it does not account for non-linear systems that appear in the real world. The Extended Kalman Filter (EKF) accounts for non-linear models by using the first order

Taylor Series linearization of the models centered around the previous prediction/correction. While this accounts for non-linearizations, the EKF performs poorly in very non-linear systems. The state of a vehicle can be far from linear, but the error between the true state and the predicted state from the motion model often act closer to linear over time. We can tweak the EKF to use the non-linear motion model to predict the nominal state (without error) and use the EKF to predict the error-state between the true state that we care about and the nominal state from our motion model. Then we can update the estimated true state by adding our filtered error-state to our nominal state prediction. This approach is called the Error-State Extended Kalman Filter (ES-EKF) and is the approach that I took to approximate the state of an AV in a simulated environment. Now we will dive into the specifics of the localization problem and how I have modeled it.

2 Problem and Model

In the simulation, we receive LIDAR data, GNSS data, and IMU data. There are a few givens that must be accounted for. In a real-world raw LIDAR packet, we would receive many points corresponding to different objects in the environment that we could then process and follow using algorithms like Iterative Closest Point (ICP) to track how far we have moved from a previous scan. However, this processing in the simulation has already been completed and we are receiving a LIDAR based position of the vehicle as an observed state of the vehicle. In this simulation, we can simply use the data from LIDAR as a measurement of the vehicle's position as opposed to the heavy processing that would normally be needed to distill this information in a real-world scenario. Furthermore, we could include variables in the state such as the IMU biases that could be present in the system. Over time, IMU's can drift in their average output and this could cause inaccuracies in our data, but for simplicity, we will be excluding those values from our model. Finally, we are beginning our simulation with a known initial state that we can base our next predictions off of. In the real world, you may not know the initial state of the vehicle and there are additional steps needed to find that initial state. The IMU data is received at a much more frequent interval than the LIDAR and GNSS data and will be used in the non-linear motion model to predict the nominal state. As we receive LIDAR and GNSS data, we will update the error-state and adjust our nominal state to account for small errors from the motion model. This control loop can be seen below.

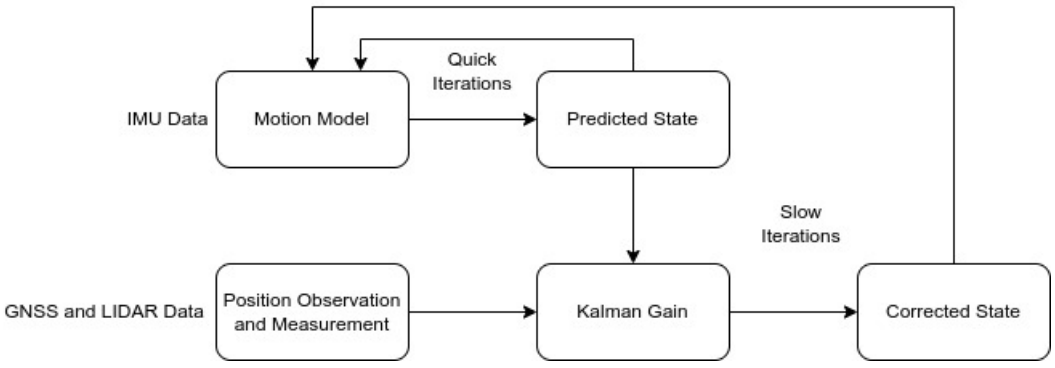

Figure 1: High level loop of the Kalman Filter where the predicted state either feeds back into the motion model, or into the Kalman Gain if a position observation is made since the last iteration.

Every time an IMU measurement is made, our filter will loop and if there is a GNSS or LIDAR update since the last loop, we will go through the Kalman Gain and measurement model to correct our estimated true state with our error-state. Otherwise, we will continue to loop between the motion model and our predicted nominal state. A mathematical definition of this model can be seen below.

$$\mathbf{x} = \hat{\mathbf{x}} + \delta\mathbf{x} \tag{1}$$

In the equation above, $\mathbf{x}$ represents our estimate of the true state of the vehicle, $\hat{\mathbf{x}}$ represents our predicted nominal state from the motion model, and $\delta\mathbf{x}$ represents the error-state that we will be finding through the ES-EKF. In our case, the motion model is non-linear and can be used to find the nominal state, but we will eventually need to linearize it in order to treat the error-state as a multi-variable Gaussian distribution and use the Kalman Filter. The list below gives a high level description of the steps we need to take in order to reach a state estimate.

1. An IMU measurement is received and we update our predicted nominal state ($\check{\mathbf{x}}$) based on our previous state using the non-linear motion model that will be described later.

2. In step 1 we are simply using the non-linear motion model to compute a nominal prediction of the state. Now, we need to compute the linearized motion model at this time step in order to solve through for the error state and treat the error-state as a Gaussian distribution for the ES-EKF.

3. Every time we predict the nominal state, our error stacks and that uncertainty in our prediction is important to the Kalman Filter. This uncertainty helps us understand how certain we are about our predictions and which sensor values we can trust most and must rely more heavily on at a given time step when we start correcting our state later down the line. During this step, we will update our uncertainty, or our error in our estimation of the error-state.

4. At this point, if we don't have a GNSS or LIDAR reading since the last time step, we can loop and repeat steps 1-3 to update our prediction and uncertainty again. However, if we have received a GNSS or LIDAR reading, we will continue to the following steps and we have the opportunity to correct for errors in the prediction of the true state before looping.

5. Now we can compute the Kalman Gain, which is a factor based on our uncertainty values to inform how much we trust our predicted state and how much we trust our new position observation. In the case where our uncertainty in the predicted state is high, the Kalman Gain will weight the position observation highly and the predicted state low, and vise versa.

6. We then use the Kalman Gain to compute an error state between the "true" state and the nominal predicted state, which we will then use in the next step to correct our predicted state.

7. We will use our error state to correct our predicted state by combining the predicted state with our error state.

8. Finally, we update the state covariance which will be fed back into the motion model on the next iteration of the loop. From here we wait for the next IMU reading and repeat all of the steps.

While some of these steps might not make sense just yet, we will dive into the details and mathematics behind them to clarify.

2.1 Prediction Step and Motion Model

The state (x_k) that we are trying to accurately estimate contains the position, velocity, and orientation of the vehicle and will be represented by the 10 element vector shown below where k is a discrete time step at some point in the vehicle operation.

$$x_k = \begin{bmatrix} p_k \\ v_k \\ q_k \end{bmatrix} \tag{2}$$

$$p_k = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}, \; v_k = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}, \; q_k = \begin{bmatrix} q_w \\ \mathbf{q_v} \end{bmatrix} = \begin{bmatrix} q_w \\ q_i \\ q_j \\ q_k \end{bmatrix} \tag{3}$$

The first 3 elements represent the position in 3D space, the next 3 elements represent the velocity in 3D space, and the last 4 elements represent the orientation as a quaternion with respect to the navigation

frame, a fixed frame in the world that all state variables are relative to. This quaternion is effectively the transformation and rotation from the sensor frame (where all measurements are recorded) to the navigation frame. The quaternion can also be defined by q_w, the real part of the quaternion sometimes referred to as the scalar, and $\mathbf{q_v}$, a 3 element vector containing the weights for the three imaginary axes defining that quaternion. Using the IMU as the main input into the motion model, we have 3 measurements of linear acceleration and 3 measurements of rotational acceleration that we can use to predict the state variables we have defined using kinematic equations. The IMU inputs can be defined as

$$a_k = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}, \; w_k = \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} \tag{4}$$

where a_k represents the linear accelerations from the accelerometer component of the IMU and w_k represents the rotational accelerations from the gyroscope component of the IMU. Given this information along with the time between readings, we can formulate the equations of our motion model to predict the state of the vehicle. For our position prediction, we can use the previous state of position, previous state of velocity, and the previous IMU readings to generate the equation:

$$p_k = p_{k-1} + \Delta t * v_{k-1} + \frac{\Delta t^2}{2}(C_{ns}a_{k-1} + g) \tag{5}$$

g is used to offset the Earth's gravitational acceleration since the IMU readings will show that gravitational acceleration even though we aren't accounting for it in the navigational frame that our vehicle is positioned to:

$$g = \begin{bmatrix} 0 \\ 0 \\ -9.81 \end{bmatrix} \tag{6}$$

Furthermore, C_{ns} represents the transformation from the sensor frame to the navigation frame since we are tracking the state in the navigation frame and the IMU readings are in the sensor frame. While we are representing the transformation as a quaternion, we use the last updated quaternion q_{k-1} converted to a rotation matrix in order to perform the operations defined in the equations. We represent this rotation matrix as a quaternion to save data as the quaternion only has 4 elements as opposed to the 9 elements in a rotation matrix and because it doesn't suffer from singularities in the same way that rotation matrices do. Δt is the time between IMU readings that we use to inform our position and velocity and is not a constant value. For the velocity prediction, we once again use the previous velocity and IMU data in the equation below.

$$v_k = v_{k-1} + \Delta t(C_{ns}a_{k-1} + g) \tag{7}$$

Lastly, we update our orientation, which becomes slightly more complicated as we are dealing with quaternions. Quaternion multiplication involves effectively manipulating a four-dimensional space and then inverting that manipulation to result in rotations about the three-dimensional axes that we care about. I won't dive into the mathematics and full intuition about how this works exactly, but if you understand how complex numbers and complex multiplication works in order to "slide" a point along a 1D line, it's effectively the same concept with a couple more imaginary dimensions added. More information can be found in this https://eater.net/quaternions and other online references. In effect, that quaternion multiplication creates a shift in the orientation in 3D space, so we can take the quaternion orientation from the previous prediction, then conduct quaternion multiplication with a quaternion representation of our XYZ angular change to create an orientation shift predicting where we have moved. We find this XYZ angular change by multiplying our IMU rotational accelerations by the time passed since the past reading. This operation is shown below where $\bigotimes$ represents a quaternion multiplication. Since quaternion multiplication is not commutative, we must make sure that the previous value is on the left for this prediction.

$$q_k = q_{k-1} \bigotimes q(w_{k-1}\Delta t) \tag{8}$$

However, we want to represent this operation in matrix mathematics form. To conduct this operation, we can represent values from the quaternions in the following form.

$$q_n = q(w_{k-1}\Delta t) = \begin{bmatrix} q_{n_w} \\ \mathbf{q_{n_v}} \end{bmatrix} \tag{9}$$

$$q_k = \begin{bmatrix} q_{n_w} & 0 & 0 & 0 \\ 0 & q_{n_w} & 0 & 0 \\ 0 & 0 & q_{n_w} & 0 \\ 0 & 0 & 0 & q_{n_w} \end{bmatrix} + \begin{bmatrix} 0 & -\mathbf{q}_{\mathbf{n_v}}^{\mathbf{T}} \\ \mathbf{q_{n_v}} & -[\mathbf{q_{n_v}}]_\times \end{bmatrix} \mathbf{q_{k-1}} \tag{10}$$

Bold symbols in the equation above represent elements that are vectors. Furthermore, $-[\mathbf{q_{n_v}}]_\times$ represents the 3x3 skew symmetric matrix representation of the vector $\mathbf{q_{n_v}}$. The above update using the most recent IMU data and previous state estimation encapsulates step 1 from the list of steps at the beginning of this section. Now that we have completed step 1 of updating, we can move into step 2: linearizing the motion model. Taking a step back, we will now take a look at how the ES-EKF really works and lay out some terms that will be needed later to make everything work. Remembering from before, we have defined the following equation to represent our approach to finding the state with a clear definition for how to find $\hat{\mathbf{x}}$, but what about the error-state itself?

$$\mathbf{x} = \hat{\mathbf{x}} + \delta\mathbf{x} \tag{11}$$

or rearranged,

$$\delta\mathbf{x} = \mathbf{x} - \hat{\mathbf{x}} \tag{12}$$

At a higher level with only a consideration of the prediction step, can define the estimated state from the motion model as a function of the previous state($\hat{x}_{k-1}$), some control input that is in our case the IMU values (u_{k-1}), and some process noise (n_{k-1}):

$$\hat{x}_k = f_k(\hat{x}_{k-1}, u_{k-1}, n_{k-1}) \tag{13}$$

Since our motion model as we've seen is non-linear, we must now linearize that model using a first-order Taylor Series expansion as shown below:

$$x_k \approx f_k(\hat{x}_{k-1}, u_{k-1}, 0) + F_{k-1}(x_{k-1} - \hat{x}_{k-1}) + L_{k-1}(n_{k-1}) \tag{14}$$

In the equation above, F_{k-1} is a Jacobian matrix that represents the partial derivatives of each variable in the motion model with respect to each variable in the state. I won't dive into detail on how to evaluate the Jacobian, but this matrix represents the first-order rate of change of the system, or slope, of the motion model. As you may have noticed, F_{k-1} is multiplying against the difference of the true and predicted state, also known as the error-state. L_{k-1} likewise is the Jacobian of the noise in the system and is multiplying by the noise itself. We can rearrange this function and substitute error state variables into the equation as shown below to simplify the equation.

$$x_k - f_k(\hat{x}_{k-1}, u_{k-1}, 0) \approx F_{k-1}(x_{k-1} - \hat{x}_{k-1}) + L_{k-1}(n_{k-1}) \tag{15}$$

$$\delta x_k \approx F_{k-1}(\delta x_{k-1}) + L_{k-1}(n_{k-1}) \tag{16}$$

Now that we have reworked our motion model to find the error-state, we can redefine the error-state and find some of the new matrices that are needed to evaluate the error-state. We define the error-state slightly differently to our true state by representing errors in orientation by X, Y, and Z orientation as opposed to the quaternion representation used in the nominal and true state.

$$\delta\mathbf{x_k} = \begin{bmatrix} \delta\mathbf{p_k} \\ \delta\mathbf{v_k} \\ \delta\boldsymbol{\phi}_\mathbf{k} \end{bmatrix}, \ \delta\boldsymbol{\phi}_\mathbf{k} = \begin{bmatrix} \delta o_{x_k} \\ \delta o_{y_k} \\ \delta o_{z_k} \end{bmatrix} \tag{17}$$

$\mathbf{F_{k-1}}$ is linearized about the error-state as intended. The main difference is that this linearized motion model is made to fit with the error-state representation above, so orientation is represented as it is in the error-state, $\delta\mathbf{x_k}$ as defined above.

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{Z} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{18}$$

$$\mathbf{F_{k-1}} = \begin{bmatrix} \mathbf{I} & \mathbf{I} * \Delta t & \mathbf{Z} \\ \mathbf{Z} & \mathbf{I} & -[\mathbf{C_{ns}} \mathbf{f_{k-1}}]_\times * \Delta t \\ \mathbf{Z} & \mathbf{Z} & \mathbf{I} \end{bmatrix} \tag{19}$$

L_{k-1} and n_{k-1} are terms that capture error due to sensor noise in the motion model, which in our case is the IMU. IMU's typically list their variance in data sheets, but we could also test and estimate the variance in our IMU readings if needed. I will represent the IMU noise as a variance matrix so that it can be seamlessly added to the noise from our motion model as shown below.

$$\mathbf{Q} = \begin{bmatrix} \sigma^2_{acc_x} & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma^2_{acc_y} & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma^2_{acc_z} & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma^2_{gyro_x} & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma^2_{gyro_y} & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma^2_{gyro_z} \end{bmatrix} * \Delta t^2 \tag{20}$$

In order to align and combine the IMU variances with the positions in the error-state vector and linearized motion model, we can multiply $\mathbf{Q}$ by the matrix below.

$$\mathbf{L} = \begin{bmatrix} \mathbf{Z} & \mathbf{Z} \\ \mathbf{I} & \mathbf{Z} \\ \mathbf{Z} & \mathbf{I} \end{bmatrix} \tag{21}$$

At this point, we have completed step 2 and have a linearized motion model centered about the error-state. We have an error-state that represents the error for every position, velocity, and orientation in 3D space that we will update and use to estimate our true state. However, we will not update the error-state in each loop of the estimation algorithm. We only update the error-state when we observe the position through GNSS or LIDAR measurements which we will get to later. Despite not always updating the error-state itself, we are accumulating error that must be accounted for when we do end up updating the error-state. As a result, we need to update our uncertainty, or covariance matrix, $\mathbf{P}$ that will be used when updating the error-state. Remember that we are defining the error-state as a multi-variable Gaussian distribution where the error-state ($\delta\mathbf{x}$) is our mean and the matrix, $\mathbf{P}$, is our covariance. $\mathbf{P}$ represents the covariances between each variable in the error-state that we are estimating and can be updated using the linearized motion model and variance in the IMU sensor readings. As we predict each new state, our uncertainty due to the motion model grows because we aren't characterizing factors like tire drift, small changes in acceleration over the time interval, and other small factors that influence the motion of the vehicle. Furthermore, we need to characterize the actual variance in the measurements from the IMU because they will never be perfect no matter what sensor you use. Since these different factors are independent, we can treat them as additive noise. The covariance matrix at any given time step can be updated after the motion model through the following operation.

$$\check{P}_k = F_{k-1} P_{k-1} F_{k-1}^T + LQL^T \tag{22}$$

While the motion model will change depending on the IMU values at a given time step, the IMU variances will be modeled as static. $\check{P}_k$ represents the uncorrected covariance matrix at time step k. Later on, we will have to correct the covariance matrix based on variance in our GNSS and LIDAR and measurement models. P_{k-1} in the equation above could either be a corrected covariance matrix, or just a predicted covariance matrix depending on whether or not we got a LIDAR or GNSS reading in the last step of the algorithm. Now that we have updated our covariance matrix, we have completed step 2 from the list of steps in the state estimation algorithm. If we did not receive a LIDAR or GNSS measurement since the last time step, we could loop here and continue to update our state estimation based on this predicted state and our motion model. However, if we did receive a LIDAR or GNSS signal, we need to process that measurement and correct our state. This is where the magic of the Kalman Filter comes into play.

2.2 Correction Step

When we receive a measurement from LIDAR or GNSS, we use that new information along with our certainties about that position and the predicted state to estimate the error-state. We will then use that error-state to correct our predicted state and update our true state of the vehicle. In order to do that, we first compute the Kalman Gain as mentioned in step 5 of the process. The Kalman Gain is a weight matrix that represents our certainty, or how much we trust the predicted state against as well as our measurement from GNSS or LIDAR. We will later use the Kalman Gain to find the error-state. The Kalman Gain (K) can be found with the following equation.

$$K_k = P_k H_k^T (H_k P_k H_k^T + R)^{-1} \tag{23}$$

As we remember from the last subsection, P_k is our covariance matrix representing uncertainty from the motion model prediction of all the error-state variables. R is a matrix representing the sensor variance similar to how the IMU variance was represented in the initial evaluation of the uncertainty matrix, but for either GNSS or LIDAR depending on which measurement we receive.

$$R = \begin{bmatrix} \sigma^2_{sensor_x} & 0 & 0 \\ 0 & \sigma^2_{sensor_y} & 0 \\ 0 & 0 & \sigma^2_{sensor_z} \end{bmatrix} \tag{24}$$

H_k is a matrix that allows us to weight sensor uncertainty with P and correct for differences between the measurement representation against the state representation. In our case, this matrix will mostly be a zero matrix since our readings are already represented as euclidean position coordinates and we don't have any measurements for velocity or orientation. As a result, we use this matrix to eliminate some of the cross-correlations between values that have no relationship to position. However, in some cases with sensors like radars, this matrix would convert polar representations of variances into euclidean space, or any other way of aligning measurement variance to state variance representation to keep the uncertainty consistent. The 3x9 matrix is shown below.

$$H_k = \begin{bmatrix} \mathbf{I} & \mathbf{Z} & \mathbf{Z} \end{bmatrix} \tag{25}$$

The math here may not make much sense yet, but bear with me as it will in just a moment. The Kalman Gain is a weighting that tells us how much we should trust the measurement versus how much we should trust the state prediction based on these uncertainties that we have kept track of along the way. We then use that Kalman Gain to compute the error-state as shown below.

$$\delta x_k = K_k(y_k - p_k) \tag{26}$$

y_k is a matrix representing the observed LIDAR or GNSS position values and p_k is the position prediction. Let's take a deeper look at the Kalman Gain matrix. Once we compute it, we are left with a 9x3 matrix containing correlations and cross-correlations between position and each other state variable that have been distilled into weights used to compute the error-state. If we change the representation of the Kalman Gain equation, some parts of it make a little more sense.

$$K_k = P_k H_k^T (H_k P_k H_k^T + R)^{-1} = \frac{P_k H_k^T}{H_k P_k H_k^T + R} \tag{27}$$

Now we have represented the Kalman Gain as a ratio between the state prediction variance, P_k, and the measurement sensor variance, R. If we consider R becoming all zeros, then we are left with $K_k = H_k^{-1}$ and when computing the error state, we would fully trust the measurement and our error-state would simply be the difference between the observed and predicted state. However, if P_k goes to 0, then K_k also goes to 0 and our error-state is also 0 since we fully trust our predicted state. In between these edge-cases are nuances that determine how much we trust our measurement versus how much we trust our predicted state in computing the error between them. At this point, we have our error-state and have completed steps 5 and 6 of the process. We can now move into step 7, actually correcting the error and completing the state estimation. Our output δx can be split back up into position, velocity, and orientation again for us to combine with the nominal state prediction from the motion model and compute the true state estimate.

$$p_k = p_k + \delta p_k \tag{28}$$

$$v_k = v_k + \delta v_k \tag{29}$$

and finally, we can convert the error in orientation back to a quaternion and use quaternion multiplication to correct our orientation.

$$q_k = q(\delta\phi) \bigotimes q_k \tag{30}$$

The last step after correcting our state estimate is to update the covariance matrix, P once again since the state has been corrected and some of that IMU drift is now accounted for.

$$P_k = (\mathbf{I} - K_k H_k) P_k \tag{31}$$

Now we have completed the theory behind the state estimation pipeline and we will jump into my implementation using simulated data for this problem and the results.

3 Implementation

While it is often ideal to program using a faster language for real-time robotic algorithms, the simulated data and algorithm is not running in real-time and I used Python as the language of choice. In addition, it is often difficult to validate the ground truth state of the vehicle to compare the accuracy of the state estimation algorithm in the real world. However, with the simulation, we are also presented with a ground truth position that we can use to validate the accuracy of the algorithm. To begin with my implementation, we first load in the data and process it to have matching time-stamps with position data all in the same coordinate frame as the LIDAR data is recorded at a different location on the vehicle than the IMU and GPS system. The following is a figure presenting the ground truth position of the vehicle through most of its path, but not all of it. We don't have all of the ground truth position to verify that we can still predict a reasonable trajectory beyond having that data available. The figure below represents the true position of the vehicle as it drives a path.

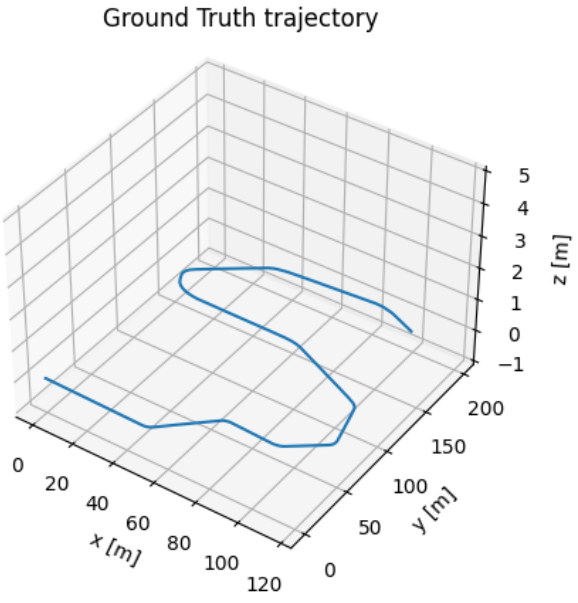

Figure 2: Ground truth position of the vehicle over the majority of its total path for the simulation.

Plotting the ground truth gives an idea of the path and while it's not a long duration, there is a fairly large amount of ground covered throughout the simulation that IMU values along would quickly fail to predict. Following the data setup and visualization of the ground truth path, I then set up some of the static variables that will be used later in the algorithm such as defining the variance for sensor noise, gravity vector, L matrix, and H matrix as shown below.

```
var_imu_f = 0.10    # accel variance (m/s2)
var_imu_w = 0.25    # accel variance (rad/s2)
var_imu_vect = np.array([var_imu_f,
                         var_imu_f,
                         var_imu_f,
                         var_imu_w,
                         var_imu_w,
                         var_imu_w]).reshape(6,1)
var_gnss  = 0.1     # meters
var_lidar = 3.00    # meters

g = np.array([0, 0, -9.81])  # gravity
l_jac = np.zeros([9, 6])
l_jac[3:, :] = np.eye(6)  # motion model noise jacobian
h_jac = np.zeros([3, 9])
h_jac[:, :3] = np.eye(3)  # measurement model jacobian
```

The source of the simulation data defined initial variances that I tweaked to improve accuracy through trial and error later on in the process. However, these variances would change depending on the environment that we are operating in. For instance, if the simulation was done in an area with many super reflective objects, I may want to increase the LIDAR variance component as I likely wouldn't trust that data as much. Furthermore, I could in the future model the variance as a variable that doesn't remain static and could adapt to different environments. The next step is to set up some variables to track the state estimate over time so that we can reference the previous estimates and later plot them. I use the shape of the IMU data because I will loop and update the state each time there is an IMU reading, so the number of estimates will equal the number of IMU readings over the simulation. Lastly, I populate the initial conditions so that we can reference the initial state in the first loop of the algorithm as the previous state and create counters for tracking when a GNSS or LIDAR reading comes in. In the code, I am tracking the state variables of position, velocity, and orientation separately for easier access to the previous states and plotting later on; this will make the program more readable.

```
p_est = np.zeros([imu_f.data.shape[0], 3])  # position estimates
v_est = np.zeros([imu_f.data.shape[0], 3])  # velocity estimates
q_est = np.zeros([imu_f.data.shape[0], 4])  # orientation estimates as quaternions
p_cov = np.zeros([imu_f.data.shape[0], 9, 9])  # covariance matrices at each timestep

# Set initial values.
p_est[0] = gt.p[0]
v_est[0] = gt.v[0]
q_est[0] = Quaternion(euler=gt.r[0]).to_numpy()
p_cov[0] = np.zeros(9)  # covariance of estimate
gnss_i  = 0
lidar_i = 0
```

There is a **Quaternion** class that I created in order to more easily conduct quaternion multiplications and convert between euler angles, rotation matrices, and quaterions. I will not dive into the details of the code there, but the full source code is available at the end of this article and reflects the mathematical equations expressed in Section **2**. Finally, we can begin the loop and start estimating the state through the simulation! The following block represents steps 1-4 where I use the non-linear motion model to predict the next state, linearize the motion model, and propagate for uncertainty.

```
for k in range(1, imu_f.data.shape[0]):  # start at 1 b/c we have initial prediction at 0
    delta_t = imu_f.t[k] - imu_f.t[k - 1]

    # 1. Update state with IMU inputs
    f_k1 = imu_f.data[k-1]
    Cns = Quaternion(*q_est[k-1]).to_mat()

```

```
    f_ext = Cns @ imu_f.data[k-1]
    p_check = p_est[k - 1] + delta_t * v_est[k - 1] + 0.5 * (delta_t ** 2) * (f_ext + g)
    v_check = v_est[k - 1] + delta_t * (f_ext + g)

    theta = imu_w.data[k-1] * delta_t
    q_check = Quaternion(euler=angle_normalize(theta)).quat_mult_right(q_est[k - 1])

    # 2. Linearize the motion model and compute Jacobians
    F = np.eye(9)
    F[:3, 3:6] = np.eye(3) * delta_t
    F[3:6, 6:9] = -skew_symmetric(Cns @ f_k1) * delta_t
    Q = (np.eye(6) * var_imu_vect) * delta_t**2

    # 3. Propagate uncertainty
    p_cov_check = p_cov[k-1]
    p_cov_check = F @ p_cov_check @ F.T + l_jac @ Q @ l_jac.T

    # 4. Check availability of GNSS and LIDAR measurements
    while gnss_i < gnss.t.shape[0] and gnss.t[gnss_i] <= imu_f.t[k]:
        if gnss.t[gnss_i] == imu_f.t[k]:
            y_k = gnss.data[gnss_i]
            p_check, v_check, q_check, p_cov_check = measurement_update(var_gnss,
                                    p_cov_check, y_k, p_check, v_check, q_check)
            break
        gnss_i += 1

    while lidar_i < lidar.t.shape[0] and lidar.t[lidar_i] <= imu_f.t[k]:
        if lidar.t[lidar_i] == imu_f.t[k]:
            y_k = lidar.data[lidar_i]
            p_check, v_check, q_check, p_cov_check = measurement_update(var_lidar,
                                    p_cov_check, y_k, p_check, v_check, q_check)
            break
        lidar_i += 1

    # Update states (save)
    # Set final state predictions for timestep
    p_est[k, :] = p_check # 1x3
    v_est[k, :] = v_check # 1x3
    q_est[k, :] = q_check # 1x4
    p_cov[k, :, :] = p_cov_check  # covariance of estimate
```

p_check, *v_check*, and *q_check* represent the predicted states while *imu_f* is the linear IMU acceleration values. Steps 5-8 occur in the **measurement_update** function which conducts a correction given the variance of a specific sensor. The @ symbol represents a matrix multiplication operation in Python as opposed to a scalar multiplication on the vector or matrix. Now we will dive into the **measurement_update** function.

```
def measurement_update(sensor_var, p_cov_check, y_k, p_check, v_check, q_check):

    # 5 Compute Kalman Gain
    sensor_var = np.array([[sensor_var], [sensor_var], [sensor_var]])
    K = p_cov_check @ h_jac.T @ (inv(h_jac @ p_cov_check @ h_jac.T + np.eye(3)*sensor_var))

    # 6 Compute error state
    delta_x = K @ (y_k - p_check)
    delta_p = delta_x[:3]
    delta_v = delta_x[3:6]
    delta_q = delta_x[6:]

    # 7 Correct predicted state
    p_hat = p_check + delta_p
    v_hat = v_check + delta_v
    q_hat = Quaternion(euler=angle_normalize(delta_q)).quat_mult_left(q_check)

    # 8 Compute corrected covariance
    p_cov_hat = (np.eye(p_cov_check.shape[0]) - K @ h_jac) @ p_cov_check

    return p_hat, v_hat, q_hat, p_cov_hat
```

This covers the entire ES-EKF! As mentioned before, we have saved all the state updates over time and we can no look at the results plotted against the ground truth data and some of the errors visualized along with standard deviations from our P covariance matrix.

4 Results

Once the simulation ran, I plotted the state estimation of position on top of the ground truth data to compare. These two angles show that the estimation actually did a good job of estimating the state!

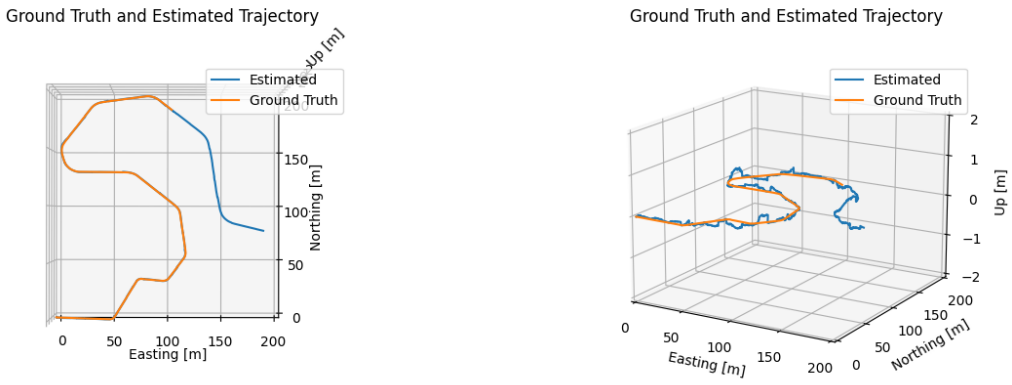

Figure 3: Estimated trajectory against the ground truth data.

From the image on the left, it seems as though there is basically no error. However, the scale of the path is a couple hundred meters, so a small error of even a meter will be hard to observe even though it appears nearly perfect. However, when we look at the verticle axis, the axis is scaled to only a couple meters on either side of 0, and you can see that the path isn't perfect. There are some squiggles representing z position variation over the path. While the path is nice to observe, we need to look at error plots to figure out how accurate the estimation truly is. In the figure below, the position and orientation are plotted as their true states against their estimates in blue, along with a variance on either side of the estimate from our covariance matrix in red. This represents how certain we were of our estimate over time.

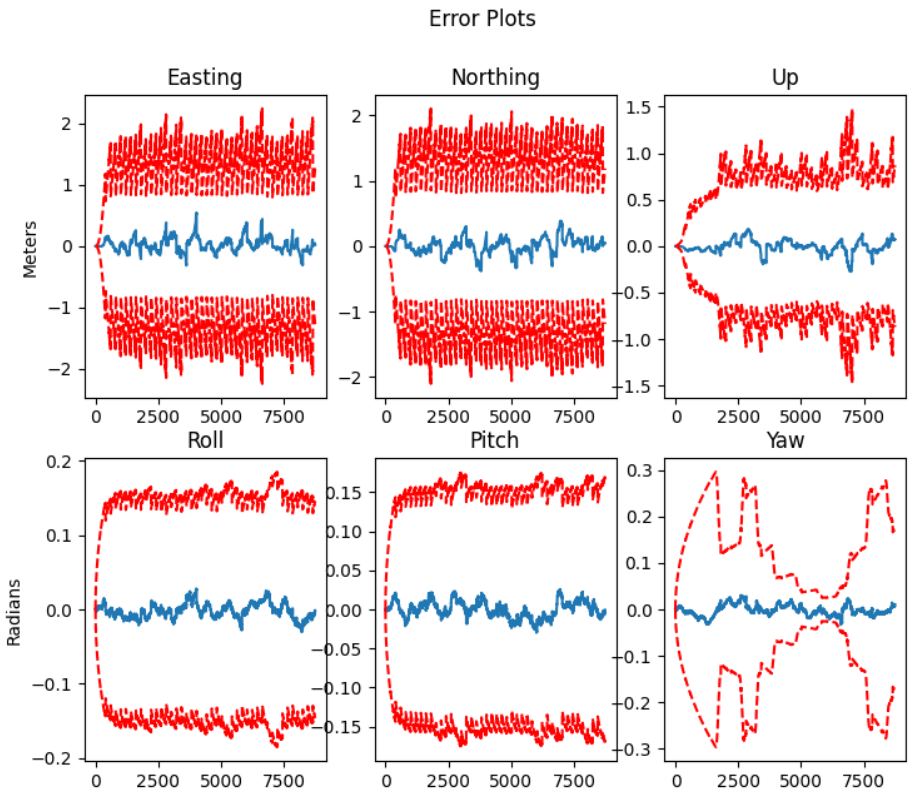

Figure 4: Error in the estimate against the ground truth over the simulation along with the variance representing uncertainty in the estimate.

Looking at these plots, none of the position estimates deviate more than a meter from the ground truth! However, they are relatively noisy just as a result of going for some time without a correction step or having GNSS/LIDAR data becoming less reliable in different environments that the simulated vehicle is driving through. However, once there is a good position observation, the error seems to return closer to 0 and get back on the right track. The uncertainty is also interesting to look at. In the beginning, we are very certain that the estimation is correct, but that certainty quickly deviates away from the estimate, but within a reasonable range. One key point is in the yaw orientation where the certainty comes back closer to the estimate meaning that the confidence is high for that period of time, likely due to some consistently good measurements and minimal changes and drift in the yaw direction which would make sense given the vehicle drives on a flat plane and doesn't move seemingly at all in the z direction. The confidence in the z position is also high due to the same reasoning. If there aren't many changes in acceleration in the z-direction, it would make sense that we're more confident that our estimation is correct.

5 Discussion

State estimation is a problem that can be solved fairly accurately given the right environment and sensors, but it also has many edge cases and each problem requires a different level of accuracy. In the case of our self-driving vehicle, it is probable that being within a meter of our true position will be effective enough to drive on the main road, but there are also cases where tunnels would effectively eliminate GNSS signals and that may cause much larger errors. Depending on what the state estimation would be used for, errors may need to be on the order of even 0.1m. For instance, if you have a robot parking into a charging station based on a map of the building, you may need to have an accuracy that is within tight tolerances to get

into the charging port. However, if you are using the localization data to understand which part of a global map to load, a few meters of error is likely enough to know where you are in the world. Regardless, state estimation is an important component in many robots and choosing the right sensors and understanding how to best estimate that state can create a huge difference in overall performance. The ES-EKF is not the state-of-the-art way to perform this estimation anymore as there are other potentially more accurate ways to estimate a state such as the unscented Kalman Filter or the Particle Filter, but it is more simple and requires less computational power for smaller systems where the accuracy requirements aren't as essential.

6 Sources

https://www.coursera.org/learn/state-estimation-localization-self-driving-cars/home/welcome
https://notanymike.github.io/Error-State-Extended-Kalman-Filter/
https://eater.net/quaternions

7 Code

You can find the source code that I have written along with some frameworks from Coursera's State Estimation course on Github (https://github.com/davidt315/coursera_state_estimation/tree/main/simulated_esekf).