**Summary of** **ITB de Labo’s**

**Extended Kalman Filter (EKF) Research Activities in 2022**

* 1. **Outline**

1. **Introduction**
2. Background Problems
3. Fundamentals of Kalman Filter
4. **Mathematical Derivation and Programming Integration**
5. Algorithm Procedures
6. Kinematics Model
7. Prediction Step
8. Update Step (measurement & fusion)
9. Headings Calibration
10. **Research Output**
11. Simulation & Tuning
    1. **Reports**
12. **Introduction**
13. **Background Problems**

The development of the MSD700 project needs information on the robot’s position at any time to enable a fully autonomous system. For outdoor use, GPS is the primary solution to obtain a global position, but its current performance still left much to be desired. The problems relating to the GPS’s position acquisition performance can be listed as follow:

* The precision and accuracy of GPS’s position and its displacement are too dependent on the environment, such as the number of satellites detected, surrounding tall buildings, cloudy sky, etc. (highest error up to 6 m)
* The GPS’s compass and IMU data are unavailable because of module communication issues, which means there is no headings information.
* The measurement frequency is still too low for autonomous control implementation (close to 1Hz or 1 measurement/sec.)

Those problems can be alleviated through sensor/data fusion. One of the most common solutions is by implementing Kalman Filter. ITB de Labo creates the Kalman filter algorithm and the simulation environment on MATLAB and converts it to Python script so that it can be used on Raspberry Pi minicomputer. The algorithm is deployed on MSD700’s prototype with the architecture as shown in **Figure 1**.

Timeline

Description automatically generated**\\**

**Figure 1.** MSD700 prototype’s architecture

1. **Fundamentals of Kalman Filter**

Kalman filter is an algorithm that fuses (combines) a series of measurements and predicted states of a system to produce the best (updated/corrected) state estimates. The estimation (data fusion) is based on the system’s mathematical model, previous states, control/driving inputs, noisy values, and their probability distributions over time. Kalman filter exclusively assumes that the probability distribution is Gaussian, which means the probability is normally distributed (the mean/expected value is in the middle).

For GPS’s problem in obtaining the best position (state) estimates, **Figure 2** illustrated how the Kalman filter works. From the previous position () and its probability distribution (), we can predict **(1)** current position (). Since the prediction is calculated with a mathematical model which may not be described perfectly by the system. It results in the predicted position’s probability distribution () becoming wider. In other words, it has a high probability of deviating from the actual position. The model derivation is further explained on **page 7.**

The further prediction will only continue to lower the confidence in the predicted position. This is where the measurements **(2)** come to play. The measured position () may be different from the predicted position () and it has its probability distribution (). The predicted value and measured value is then fused **(3)** to estimate the actual position () and its new probability distribution (). This estimated value will have a better probability distribution (smaller distribution area, higher confidence in expected/mean value). The mathematical operation is further explained on **page 10.**

For easier visualization, the fusion (update/correction) step can be thought of as taking the overlapping area in a Venn Diagram as shown in **Figure 3**. In the sense of probability distribution, this sliced area between two circles (prediction and measurement) has a smaller area (less deviation, narrower probability distribution) and you are sure that it has the same member as both circles (higher confidence of being correct).

Shape

Description automatically generated

**Figure 2.** Kalman Filter illustration ([Machine Learning TV](https://www.youtube.com/watch?v=LioOvUZ1MiM))

Diagram, venn diagram

Description automatically generated

**Figure 3.** Venn Diagram visualization

In our case, those procedures are run for each time step which creates an iterative (looping) process as seen in **Figure 4.** This process is done in a microcontroller and/or minicomputer. The GPS problems described previously are solved as follows:

* GPS’s precision and accuracy are improved by estimating the position (and displacement if necessary by doing simple math outside the EKF process (after – before position) through combining (fuse) both measurement and prediction value.
* Headings can now be calculated through the prediction step based on the estimated position, though a suitable calibration is needed for acceptable prediction/estimation (further explanation on **page 13**)
* Whenever a new GPS measurement is not yet available, the update step (measurement and fusion) can be bypassed and the position is obtained solely from the prediction step. This setting provides more position data at a much higher frequency compared to being dependent on GPS only.

The next part will explain the mathematical and programming aspect of the Extended Kalman Filter (EKF), one of the most versatile kinds of Kalman filters. The notation which is used in further explanations may be different from the ones introduced previously, but the whole concept is the same. In addition, the integration of EKF on MSD700’s prototype will also be described for better context. Readers are expected to have basics on mathematics, and programming to understand this document, but the basics of statistical importance will be briefly explained in each section.

1. **Mathematical Derivation and Programming Integration**

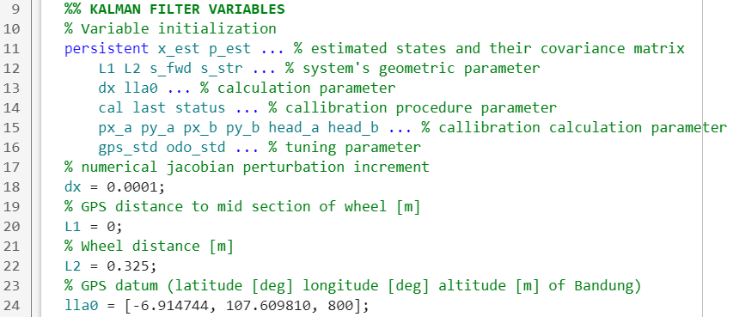
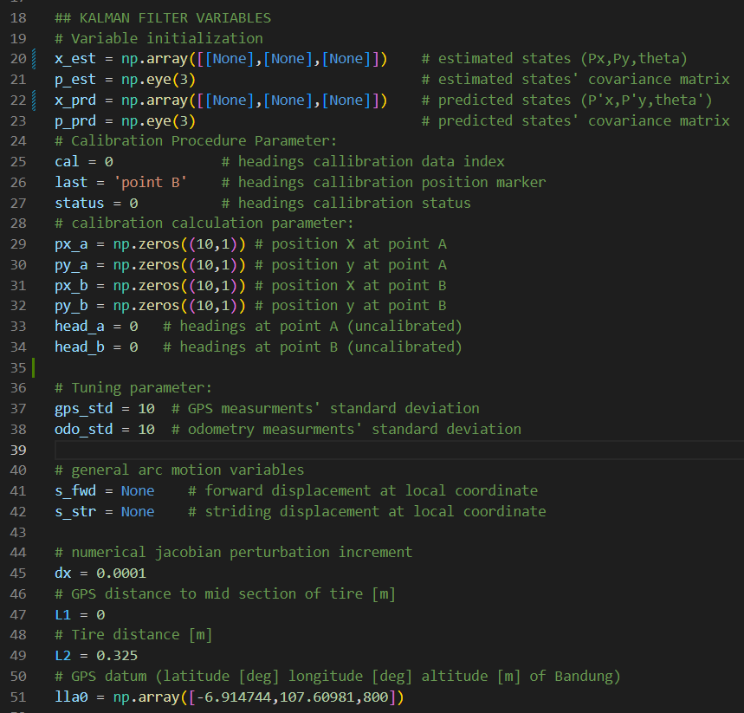
Diagram

Description automatically generated **Figure 4.** EKF flowchart

1. **Algorithm Procedures**

**Main Procedure**

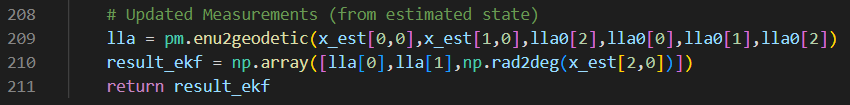
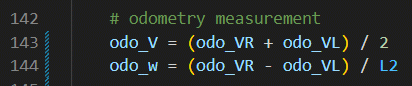
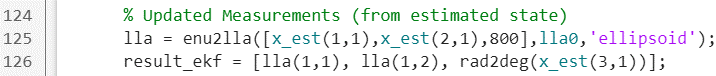
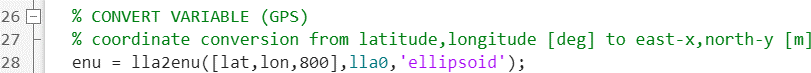
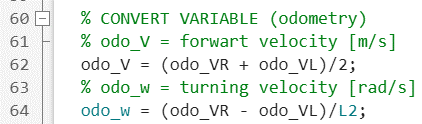
Based on the flowchart in **Figure 4**, the EKF algorithm’s main procedure is denoted by solid arrows. This procedure is the same as the one explained in the introduction in which odometry data is used for the prediction step while GPS data and the predicted state are combined through the update step to obtain the state estimate. All the required variables can be seen in **Figure 5** as they were initialized on the program (code). The algorithm’s input is the left and right wheel’s velocities (in m/s) from odometry and GPS’s latitude and longitude (in degrees), while the output is the estimated/updated latitude, longitude, and headings (all in degrees). The purpose of each variable will be explained later on in this document. This main procedure is run by inserting mode = 1 into the EKF function, as shown in **Figure 7**.



**Figure 5.** Variable Initialization (black = Python, white = MATLAB)

**Variable/Unit Conversion**

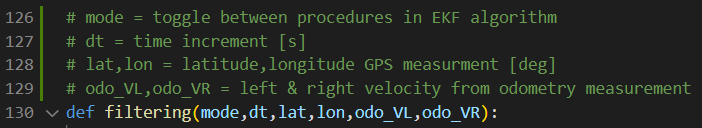
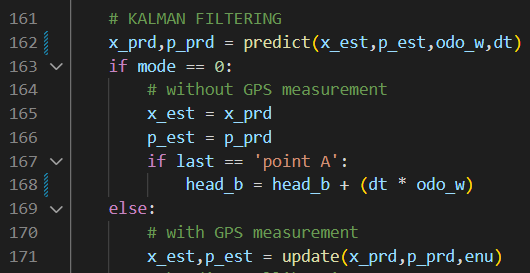
To simplify the mathematical operation, positions are calculated as x and y coordinate in meter (m) and headings in radians (rad) with the control input of the system being forward velocity (m/s) and turning velocity (m/s). A global position datum is used for positions x = 0 and y = 0, and the position is converted to x-east and y-north about the datum through a readily available function both in MATLAB and Python. The calculated heading is also converted from radian to degree with a readily available function as well. The left and right wheels’ velocities are converted to the desired form based on the kinematics model of the system which will be explained in detail on **page 7**. This procedure is denoted by the “convert variable” block in **Figure 4** and written in code as shown in **Figure 6**.

**Figure 6.** Variable/unit conversion (black = Python, white = MATLAB)

**Auxiliary Procedure: Bypassing Update Step**

Whenever the GPS’s measurements are unavailable, the update step is bypassed and the predicted states become the output. This procedure is denoted by the orange block and dotted line in **Figure 4**. This is necessary because GPS has a lower data acquisition rate than odometry and is susceptible to signal loss which renders the update state ineffective at times. This procedure can be run by inserting mode = 0 into the EKF function, as shown in **Figure 7**.

**  
  
Text

Description automatically generated  
Text

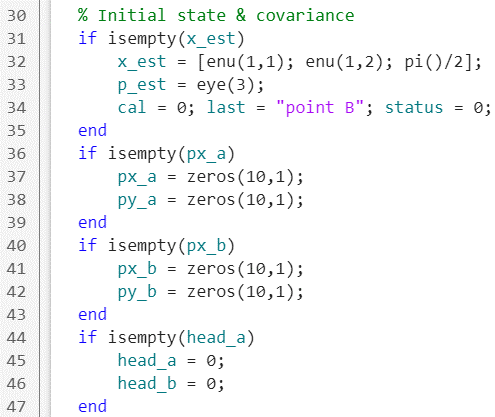
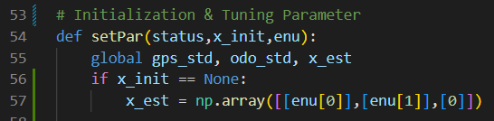
Description automatically generated  
Figure 7.** Mode input, main procedure, and bypass procedure (black = Python, white = MATLAB)

**Auxiliary Procedure: Heading Calibration**

In addition, headings calibration is required to ensure correct estimations of headings. This is necessary because we do not have the means to measure the heading directly (do not have other values for comparison/fusion). Hence, the initial heading needs to be as close to the truth as possible. The calibration procedure is conducted only for the initial deployment of the architecture. This additional step is shown by the purple block and dotted line in **Figure 4** and explained in detail on **page 13**.

**Initial State & Covariance**

Besides the heading, EKF also needs to have an initial position and its probability distribution (the value about the previous introduction) at the start of its deployment. This initial value should not be arbitrary to prevent a divergent estimation result. GPS measurement is used for this, which means the algorithm can only be deployed after the first GPS measurement. The initial probability distribution (in the form of a covariance matrix) can be arbitrary, as shown in **Figure 8**. The statistical meaning of probability distribution and covariance for the EKF algorithm will be further explained briefly on **page 10**.



**Figure 8.** Initial State and Covariance Matrix (black = Python, white = MATLAB)

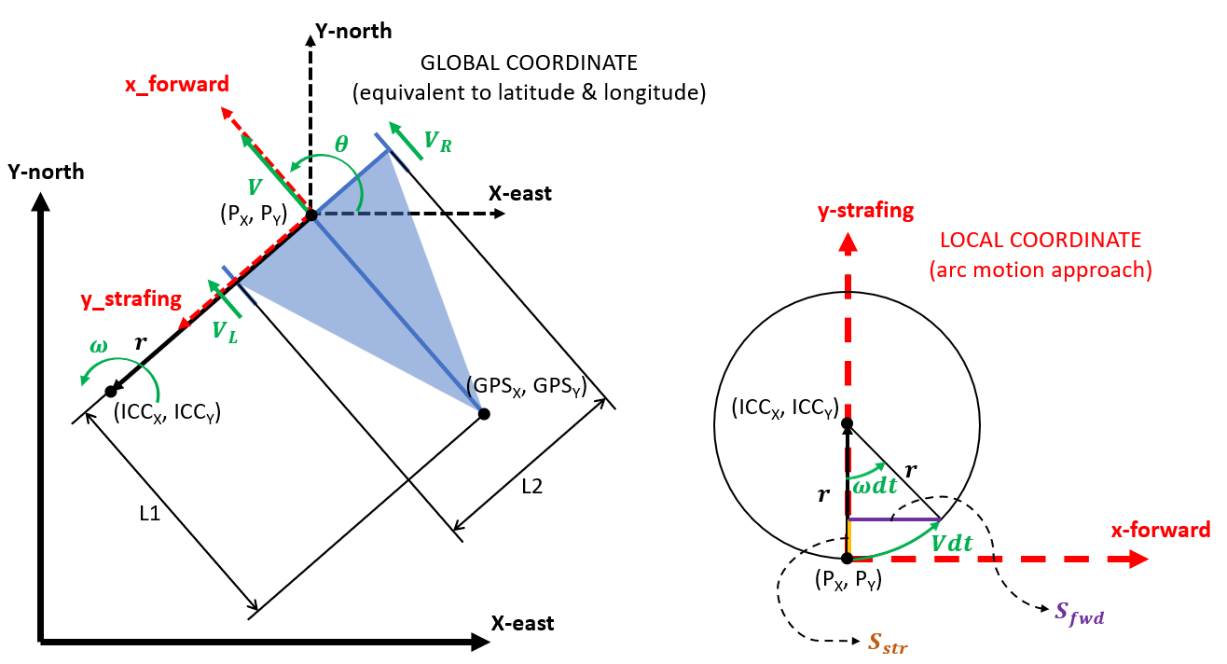
1. **Kinematics Model**

**Skid-Steering Mechanism**

Take a look at **Figure 9** for the complete kinematics model and variable. The MSD700 utilize a drive mechanism known as a differential drive (skid-steering) which consists of 2 drive wheels mounted on a common axis, and each wheel can independently be driven either forward or backwards. While we can vary the velocity of each wheel, for the robot to perform rolling/turning motion, the robot must rotate about a point that lies along their common left and right wheel axis. The point that the robot rotates about is known as the Instantaneous Center of Curvature (ICC). The robot’s state that we want to predict is its position (Px,Py) and heading () in X-east and Y-north coordinate with positive theta in the counter-clockwise direction X-east axis.

The known variables are and which are the left and the right wheel velocities while L2 is the distance between both wheels. The general location of the robot (Px,Py) is the mid-section distance of both wheels. The GPS may be positioned somewhere along the mid-section with a distance of L1. The general velocity of the robot (), the rate of rotation (), and the radius of curvature about the ICC () can be calculated from and using **Equation (1.x)**

**(1.a)** **(1.b)** **(1.c)**

**Figure 9.** Kinematics model of the differential drive (skid-steering) robot  
(Left = global coordinate, Right = local coordinate)

**State Prediction in Local Coordinate**

The general idea to predict the position of the robot after a single discrete timestep () is to calculate changes of position in local coordinate (x-forward, y-strafing), then transform (rotate) the local coordinate based on the headings in global coordinate (x-east,y-north). On the other hand, change in the heading can be calculated directly in global coordinates For the general rotating (arc) motion of the differential drive robot in local coordinates, the position will change in the x-forward direction denoted by and y-strafing (sideways) direction denoted by as shown in **Figure 9**. These changes within the arc/circular motion approach in local coordinates can be calculated using **Equation (2.x).**

**(2.a)**

**(2.b)**

A special case occurs when the rate of rotation () equals zero (moving in a straight line). This case leads to division by zero in the radius of curvature about the ICC () formula. For this case, a linear kinematics approach can be used where the change of position in the x-forward direction is calculated by multiplying the general velocity () and discrete timestep () as shown in **Equation (3.x).** In this assumption of moving in a straight line, there is no change of position in the y-strafing (sideways) direction.

**(3.a)**

**(3.b)**

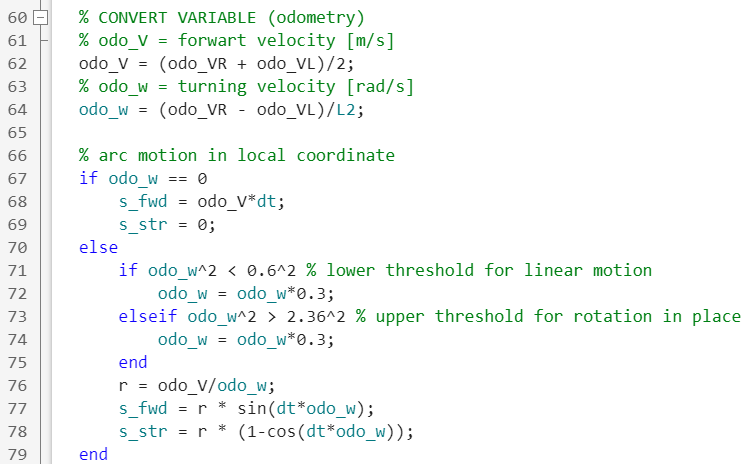
**State Prediction in Global Coordinate**

The changes of position in global coordinates can be calculated through matrix multiplication of changes in local coordinates and general rotational transform matrix. On the other hand, the change of heading in global coordinate can be calculated by multiplying the rate of rotation () and discrete timestep (). Finally, the robot’s states after a discrete timestep are predicted by adding previous states with the changes of each state. The full formula is represented in matrix form as shown in **Equation (4.x).**

**(4.a)**

Because there is no direct heading measurement, the heading prediction is prone to instability (sensitive to noise error). To tackle this problem, a pseudo-band-pass filter is conducted on the calculation of the rate of rotation . to maintain the headings on straight line movement (lower threshold) and reduce overshoot on rotating movement (higher threshold). The overall programming is shown in **Figure 10** and **Figure 11**.

Text

Description automatically generated  
**Figure 10.** Kinematic model calculation in local coordinate source code  
(black = Python, white = MATLAB)

1. **Prediction Step**

**State Prediction**

Based on the prediction block in **Figure 4**, we have two calculation that needs to be calculated, namely the predicted state and predicted covariance for the next time step. The state prediction is the system’s kinematic model **Equation (4.x)** as explained before with previous timestep estimated state and current timestep predicted state are positions and heading of the robot before and after timestep increment. The control input is the robot’s general velocity () and rate of rotation (), while noise/disturbance input is assumed to be a mean value of zero 0. The kinematic model function then becomes **Equation (5.x)**

**(5.a)**

**Graphical user interface, text

Description automatically generated**Text

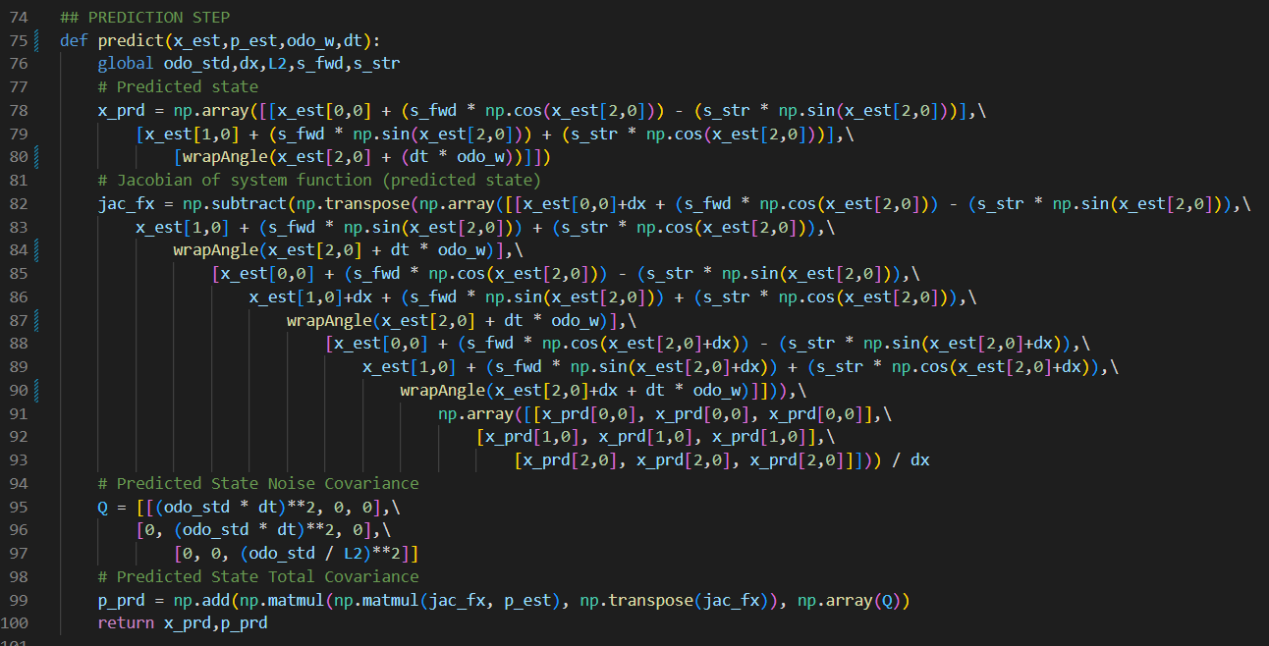
Description automatically generated  
**Figure 11.** Kinematic model calculation in global coordinate source code  
(black = Python, white = MATLAB)

**Covariance Prediction & Jacobian Matrix**

To predict covariance (probability distribution), the formula is used with being the systems’ previous covariance and being the noise covariance The covariance matrix is simply a mathematical way to represent the standard deviation of a multivariable system. In this case, since our system has three states/variables (Px,Py,theta), we have a three-by-three (3x3) covariance matrix. For a normally distributed noise with a mean value of zero, we can write the covariance matrix as a diagonal matrix with each non-zero element corresponding to the covariance value of each state’s noise. For our case where each state (variable) is basically from odometry, we can use **Equation (6.x)** as the noise covariance matrix. The timestep increment and L2 variables are used to match each state’s units (from m/s to m and rad), while covariance equals standard deviation to the power of 2.

**(6.a)**

The nabla operation on the system’s kinematic model is a jacobian matrix which basically means a partial derivative of an n-row matrix function with respect to n-row vector at value or . To simplify the calculation, a numerical derivative method in matrix form is used which implement the fundamental definition of derivation which is . For , the result is 1 or identity matrix since there is no noise variables on the system’s kinematic model. For , we now have **Equation (7.x)** with perturbation value being as small as possible. The overall programming is shown in **Figure 12**.

****Text

Description automatically generated  
**Figure 11.** State & Covariance prediction source code  
(black = Python, white = MATLAB)

**(7.a)**

**(7.b)**

1. **Update Step (measurement & fusion)**

**Measurement Innovation**

Based on the update block in **Figure 4**, there are three sub-procedure in the update step. The first one is measurement innovation (error calculation) which is calculating the difference between measured value and predicted value . The function is called the system’s measurement model. It is meant to transform (convert) the state variables into an equivalent form of the measurement variable with an assumed zero mean value of noise. Since we can only measure the robot’s position from GPS, we cannot include the headings in our equation. Moreover, since the GPS is positioned distance of the general position, an additional calculation is added to alter the GPS measurement as shown in **Equation (8.x)**. Note that the measured value has been converted from degree (latitude,longitude) to meter (x-east,y-north).

**(8.a)**

**Measurement Innovation**

The next sub-procedure is to calculate the Kalman gain which is the weight factor to determine whether the best state estimate is closer to the predicted value or the measured value. The formula with uses the same principle for the nabla operation explained on **page 10** where for , the result is 1 or two-by-two identity matrix since there are no noise variables on the system’s measurement model. Since the measured value is obtained from GPS, the measurement covariance matrix will use GPS standard deviation. Also since the measurement model is simply picking the first two predicted state with no alteration and there are three state variables, we now have a two-by-three jacobian matrix and the equation for kalman gain can be arranged as shown in **Equation (9.x)**.

**(9.a)**

**(9.b)**

**(9.c)**

**State Estimation (Update)**

The final sub-procedure is to calulate the best estimate. The state estimate and the covariance matrix estimate are calculated using **Equation (10.x)**. This resultant value will be used as the previous/last state and covariance value for the next timestep increment calculation, hence the loop on the flowchart. The overall programming is shown in **Figure 12**

**(10.a)**

**(10.b)**

**Text

Description automatically generated**

Text

Description automatically generated  
**Figure 12.** State & Covariance Estimation (Update) sourcecode  
(black = Python, white = MATLAB)

1. **Headings Calibration**

**Linear Gradient (Slope)**

As previously mentioned, there is no direct measurement to obtain the initial value of the robot’s heading. One way to determine the initial heading is by controlling the robot to move forward in a straight line and calculate the gradient of this straight line . With noisy GPS measurement, the general concept is to have the robot stay in position A (AX,AY) at a certain time, move forward to position B (BX,BY) and then again stay there. The mean value at position A & position B will then be used to calculate the gradient using **Equation (11.x)**.

**(11.a)**

**(11.b)**

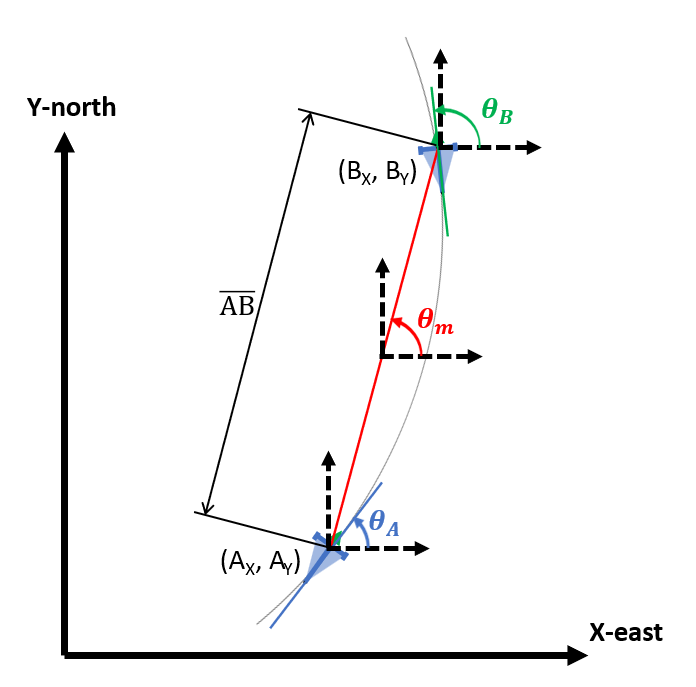
**Arc Motion Approach**

Though the linear gradient concept does work, the control system of the robot is not perfect and a circular motion as shown in **Figure 13** may occur. This means the robot’s heading at position A (), at position B (), and the heading from linear gradient calibration () are all different. By assuming that the heading at position A and position B coincide with the perfect arc motion, we have . By rearranging it to solve , we can find with known change of heading between position A to position B as shown by **Equation (12.x)**.

**(12.a)**

**(12.b)**

**(12.c)**

  
**Figure 13.** Arc motion during calibration

**Programming Implementation**

As shown in the heading calibration source code in **Figure 14**, a ‘mode’ variable is used to toggle between procedures in EKF algorithm, including during calibration. When the calibration has not been done, the variable ‘status’ equals false (status = 0). During the stay period at point A, mode = 2 is used to save the position and heading at point A. Variable ‘cal’ is used to increment the position value Into an array, which then the mean position can be calculated. During the forward movement onto point B (), either mode = 1 or mode = 0 is used to calculate the change of heading between point A and point B depending on whether GPS measurement is available or not. During the stay period at point B, mode = 3 is used to save the position and heading at point B. Finally, mode = 4 is used to calculate **Equation (11.x)** and **Equation (12.x)** as well as changing the variable to status = 1.

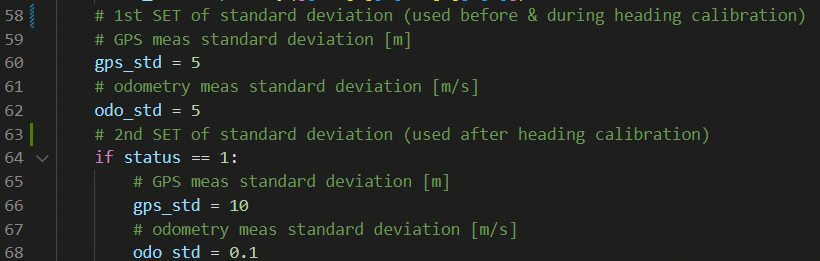
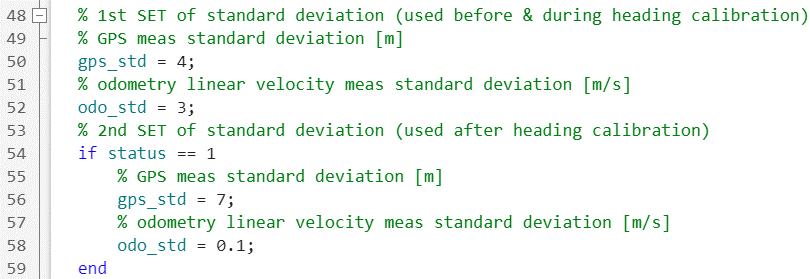
**Note on Standard Deviation**

Before calibration the heading is not reliable which means the predictied step is unreliable as well. To prevent the EKF give a divergent result, there will be two sets of odometry and GPS standard deviation**.** As shown in **Figure 15**,the first set is used before and during calibration, while the second set is used after heading calibration. The first set does not need much tuning as it only minimizes the GPS’s standard deviation while increasing the odometry’s (increasing the GPS reliability in comparison to odometry). On the other hand, the second set determines the overall performance of EKF and needs further tuning. Further explanation on tuning as well as simulation will be shown through video tutorial on **page 16**.

**Text

Description automatically generated  
Text

Description automatically generated  
Figure 14.** Heading Calibration sourcecode  
(black = Python, white = MATLAB)

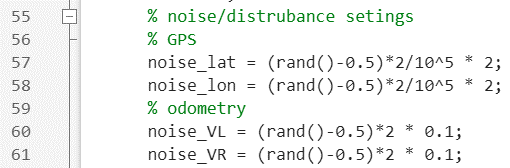
**  
  
Figure 15.** Two sets of standard deviation sourcecode  
(black = Python, white = MATLAB)

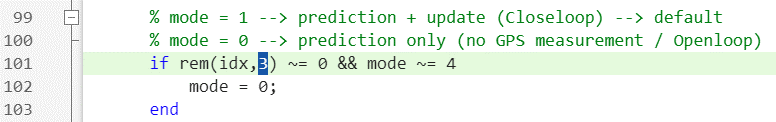
1. **Research Output**
2. **Simulation & Tuning**

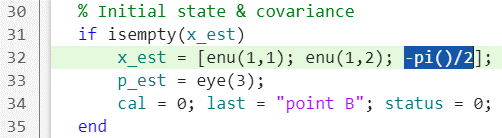
I have uploaded a tutoril video in [**this link**](https://youtu.be/Y5tN2CsXfiA) which explains how to use the simulation on MATLAB and how tune the EKF. In general this is what you have to do:

1. Make sure you have all the files in the same directory, then open the ekf.m and try\_ekf.m files using MATLAB.
2. In the try\_ekf.m file:
   1. Set the noise/disturbance in for the GPS measurement and odometry measurement as shown in **Figure 16**.
   2. Configure the GPS availability (toggle settings between mode 0 & 1) as shown in **Figure 17**.
3. In the ekf.m file:
   1. Set the initial headings as shown in **Figure 18**, better to try with significantly wrong value (our actual headings in the simulation is **π/2**)
   2. Configure the standard deviation as shown in **Figure 15**, especially the 2nd set which is used after heading callibration
4. Run the program (try\_ekf.m), then analyze the results (trajectory plot, error graph, and error comparison in command window)
5. Standard deviation tuning concept
   1. Before calibration, simply set GPS’ standard deviation lower than GPS’ standard deviation (meaning we trust GPS more than odometry)
   2. If the EKF estimates after calibration diverge from GPS measurement, either lower GPS’ standard deviation or increase odometry’s standard deviation.
   3. If the EKF estimates after calibration seems perturbed by GPS measurement way too much, either increase GPS’ standard deviation or lower odometry’s standard deviation.
   4. As a rule of thumb, do not set the standard deviation way too large (more than 100) to prevent divergent results.

This simulation was made especially for people to understand the concept and workings behind EKF algorithm. There will be some differences in actual implementation such as error characteristics, calculation ferquency (timestep), sensor types, and system’s kinematic models, but the theory and general idea is the same.

**  
Figure 16.** Noise/disturbance configurations (MATLAB, try\_ekf.m)

**  
Figure 17.** GPS availability configurations (MATLAB, try\_ekf.m)

**  
Figure 18.** Initial headings configurations (MATLAB, ekf.m)