TEL AVIV UNIVERSITY אוניברסיטת תל-אביב

THE IBY AND ALADAR FLEISCHMAN FACULTY OF ENGINEERING

הפקולטה להנדסה
על שם איבי ואלדר פליישמן

Project 2

Mapping and perception for an autonomous robot/ 0510-7591

By Avi Epstein

Tel Aviv University

December 2021

Abstract

In this project we will be implementing and analyzing the following Algorithmes:

- Kalman Filter
- Extended kalman filter
- EKF - SLAM

In the first section we will implement the classic kalman filter, we will extract the KITTI OXTS GPS trajectory from recorded data 2011_09_26_drive_0022, from this data we will get the

- lat: latitude of the oxts-unit (deg)
- long: longitude of the oxts-unit (deg)
- Extract timestamps from KITTI data and convert them to seconds elapsed from the first one

We will then transform these LLA coordinates to ENU coordinate system, add Gaussian noise to x and y of the ENU coordinates and then implement the kalman filter on the constant velocity model given the noise trajectories in order to approximate the ground truth trajectory we will see how to calibrate and initialize the appropriate matracis and initial conditions in order to minimize the RMSE error as best as possible to get a maxE less than 7. We will also see the result of the covariance matrix of state vector and dead reckoning the kalman gain after 5 sec and how it impacts the estimated trajectory, and analyze the estimated x-y values separately and corresponding sigma value along the trajectory.
As a bonus i will implement the same problem for the constant acceleration model.

In the second section we will implement the Extended kalman filter and as the same as in the first section we will use the noised trajectories as inputs of the same data as in section 1 we will see how we can deal with a nonlinear motion model and still apply kalman filter on it (EKF) again we will initialize and compute the appropriate matrices and plot the same results and analysis as in section 1 and see if we can reduce the RMSE and maxE to get a better approximation and how it deals differently with dead reckoning.

In the third section we will implement the EKF - SLAM algorithm, we will run the odometry motion model where the inputs “u” are gaussian noised, and we have assumed measurements from our state with some Gaussian noise. Here will implement the predicted state of our motion and then implement the correction of our state computing the effect of each observed landmark on the kalman gain, corrected mean and uncertainty matrix, for each time step to fully compute the estimated localization and mapping. Our inputs will be observations at the relevant time steps and motion commands. We will then analyze the results to reach minimum RMSE and maxE values. Moreover will analyze the estimation error of X,Y,Theta and of 2 landmarks.

Contents

תוכן עניינים

List of Figures

List of Movies

Attached movies:

- animation of GT KF estimate and dead reckoning
- animation of GT EKF estimate and dead reckoning
- animation of Trajectory of EKF-SLAM

List of Tables

Solutions:

Kalman Filter :

a) Recorded data: 2011_09_26_drive_0022 KITTI GPS sequence (OXT) was downloaded.
b) In this section we have extracted vehicle GPS trajectory from KITTI OXTS senser packets which are treated as ground truth in this experiment:
   - lat: latitude of the oxts-unit (deg)
   - long: longitude of the oxts-unit (deg)
   - Extract timestamps from KITTI data and convert them to seconds elapsed from the first one
c) Here we transformed the GPS trajectory from [lat, long, alt] to local [x, y, z] ENU coordinates in order to enable the Kalman filter to handle them and plotted the GT LLA and ENU coordinates.

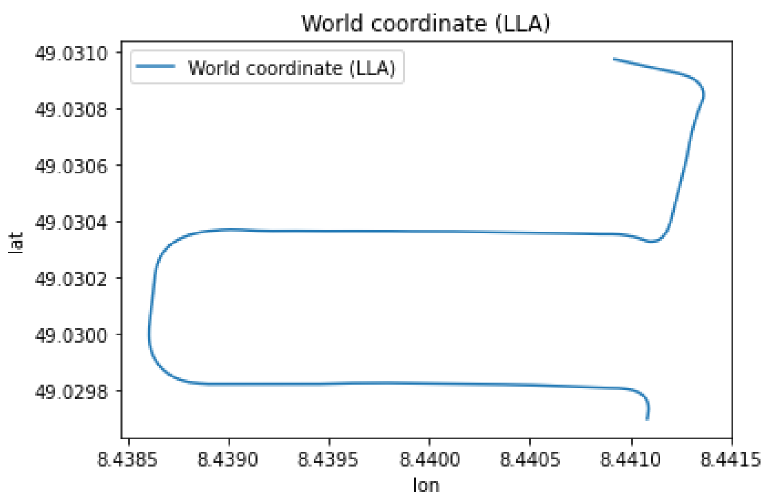


Figure 1: World coordinate (LLA)

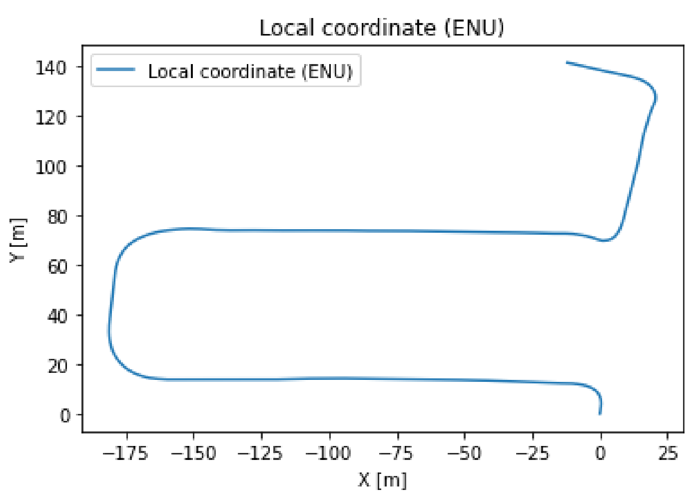


Figure 2: Local coordinate (ENU)

We can see that the vehicle drove North 20m -> West for 175m -> NNE 60m and -> East 180m -> NNE 60m -> WWN 30m .

d) Here we added Gaussian noise to the ground-truth GPS data which will be used as noisy observations fed to the Kalman filter. Noise added with standard deviation of observation noise of x and y in meter ($\sigma x = 3$, $\sigma y = 3$). In the next figure we can see the original GT and observations noise:

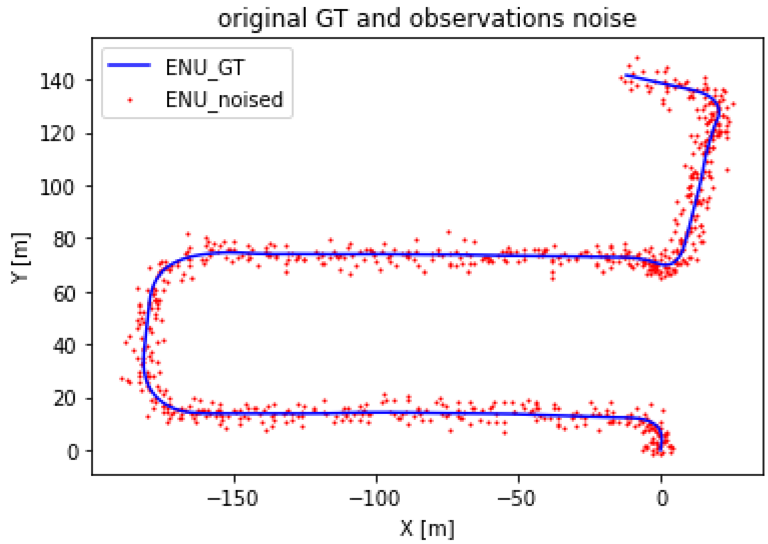


Figure 3: graph of original GT and observations noise

e) Here we will apply a linear Kalman filter to the GPS sequence in order to estimate vehicle 2D pose based on constant velocity model
Will suppose initial 2D position [x, y] estimation starts with the first GPS observation (the noised one), GPS observation noise of X and Y is known (σx =3, σy =3).

Our goal will be to minimize the RMSE which is defined as:

$$RMSE \triangleq \sqrt{\frac{1}{N}\sum_{i=100}^{N}[e_x^2(i) + e_y^2(i)]}$$

$$e_x(i) \triangleq x_{GT}(i) - x_{Estimate}(i)$$
$$e_y(i) \triangleq y_{GT}(i) - y_{Estimate}(i)$$

$$maxE \triangleq max\{|e_x(i)| + |e_y(i)|\} \quad ,$$

$$100 \le i \le N$$

N is last sample.

1) Initial conditions: according to your first observation the values of standard deviations initialized:

$$\bar{\mu} = \begin{bmatrix} x_0 \\ v_{x0} \\ y_0 \\ v_{y0} \end{bmatrix} = \begin{bmatrix} x_{est0} \\ 0 \\ y_{est0} \\ 1 \end{bmatrix}$$

This is because we can see from the initial observation that at the beginning of the drive is north so we set v_{x0} to zero and assumed a value for v_{y0} to 1.

$$\Sigma_0 = \begin{bmatrix} \sigma_x^2 & 0 & 0 & 0 \\ 0 & 100 & 0 & 0 \\ 0 & 0 & \sigma_y^2 & 0 \\ 0 & 0 & 0 & 100 \end{bmatrix}$$

This is because we want our first covariance uncertainty's of our state to be relatively hi at the beginning as we have not yet got corrections of our state so we are less certain of our initial condition after we run the algorithm this uncertainty covariance's will converge to contain 66% of the error if it contains more than 66 percent we can decrees the appropriate uncertainty. hence we chose the uncertainty of X and Y according to the variance of the measurements and for the velocities we choose a hi enough number to be able to handle the unknown velocities.

2) Matrixes: A ,B ,C:

$$A = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad B = None\ (const\ velocity)\ , \qquad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Corresponding to const velocity model $\bar{\mu}_t = A_t\mu_{t-1} + B_t u_t$ while we only observe x and y form here matrix C.

3) Measurement covariance (Q):

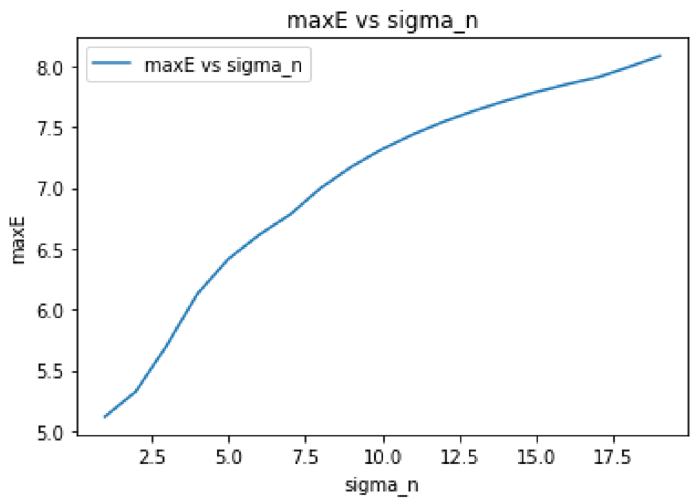
$$Q = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}$$

Corresponding to the measurement noise, we only measure x and y hence matrix of size 2x2.

4) transition noise covariance R:
containing the process noise in the const velocity model this is the source of the change in speed making it dynamic hence after analyzing different values

$$R = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta t \end{bmatrix} * \sigma_n^2,\ \ \sigma_n^2 = 1$$

You can see in the next graphs the values RMSE an maxE compared to values of σ_n:



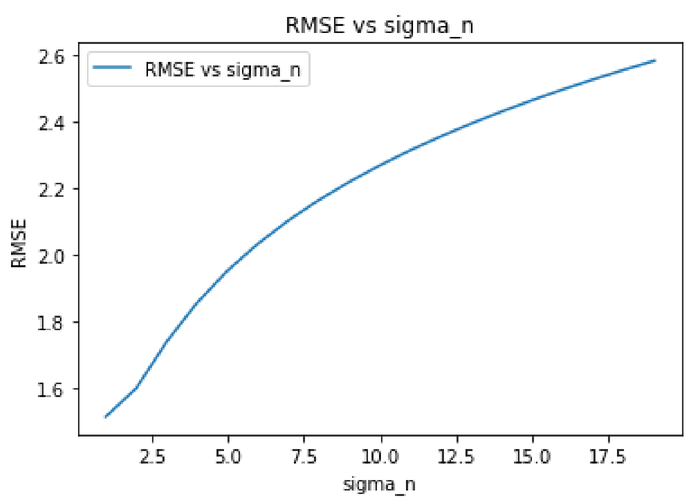


Table 1: RMSE, maxE vs sigma_n

Hence sigma_n was set to 1

<u>Kalman filter main routine:</u>

the state mean and uncertainty covariance matrix was initialized as above sections, from here the function the performe_KalmanFilter was called this function organized the inputs of the initial state and created the list of states and covariance's and then iterated over all time steps and ran the one step one_step_of_KalmanFilter and saved the state and covariance's.
the kalman step itself contains
the prediction of the location and uncertainty covariance by calculating:

$$(1)\ \bar{\mu}_t = A_t \mu_{t-1} + B_t u_t$$
$$(2)\ \bar{\Sigma}_t = A_t \Sigma_{t-1} {A_t}^T + R_t$$

And the correction step:

$$(3)\ K_t = \bar{\Sigma}_t {C_t}^T (C_T \bar{\Sigma}_t {C_t}^T + Q_t)^{-1}$$
$$(4)\ \mu_t = \bar{\mu}_t + K_t (z_t - C_t \bar{\mu}_t)$$

$$(5)\ \Sigma_t = (I - K_t C_t)\overline{\Sigma_t}$$

step 1: we predict our mean location based on our model (contained in A and B)
step 2: we predict how our uncertainty carries on to the next time step by our motion (in A) and process noise R.
step 3: we compute our Kalman gain which controls the emphasis on the deviation between what we predicted and what was measured. Where C_t maps our predicted state to the observed state
Step 4: computes the corrected mean based on the kalman gain computed and deviation of expected location and observation.
Step 5: computes the corrected covariance matrix also based on the kalman gain.
because our model is linear all transformations can be done by matrix multiplication and assumption of Gaussian distributions hold throw all transforms.

The code:

Q1 -> KalmanFilter.performe_KalmanFilter -> iterates over for each time step: one_step_of_KalmanFilter (computes dead reckoning if required)

f) Result analysis:

1) Ground-truth and estimated results:

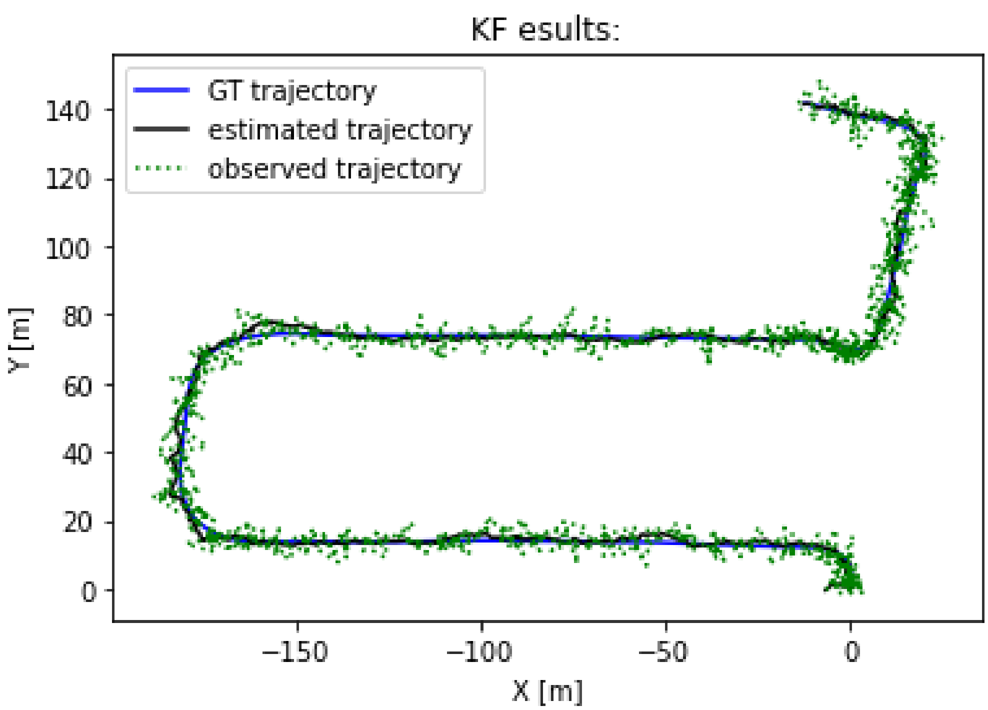


Figure 4: Ground-truth and estimated results

We can see that even with relatively hi noise and constant velocity model which is not the most exact motion modelfor our case we still get pretty good performance for the Kalman filter. However, we do see deviations in the turns which is probably do to deviation from constant speed.

2) minimum values of maxE and RMSE achieved:

   RMSE reached: 1.5143

   maxE reached : 5.1180

   we could possibly even have gotten better results trying sigma_n smaller than 1

3) in the animation we can see the Trajectory of GT and KF results and the estimate the trajectory based on the prediction without observation of after ~5 seconds (dead reckoning, Kalman gain=0):

   we can see that at the beginning of the animation the covariance is large corresponding to the large initialization and decreases as it starts converge to its variance of location.

   In addition, we see that when we set kalman gain to 0 the trajectory continues in a straight line this is because it depends now only on the motion model which is constant velocity and not on the measuerments hence we get a straight line in the direction that we were when we set K to zero.

v)

Here we will plot and analyze the estimated x-y values separately and corresponded sigma value along the trajectory.

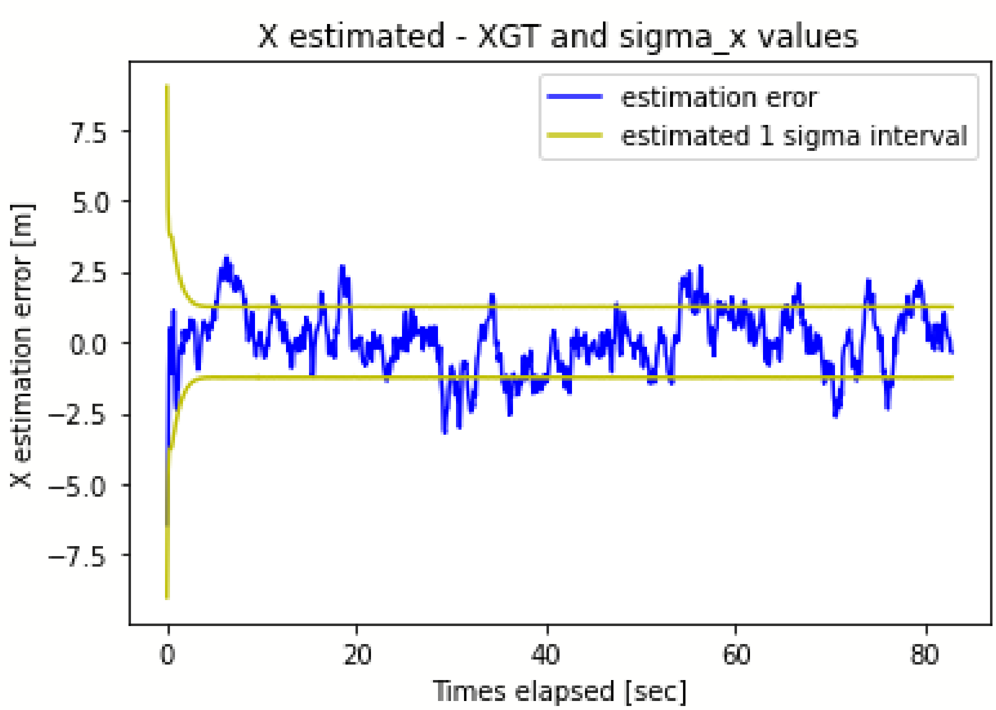


Figure 5: xestimated-xGT and σx values

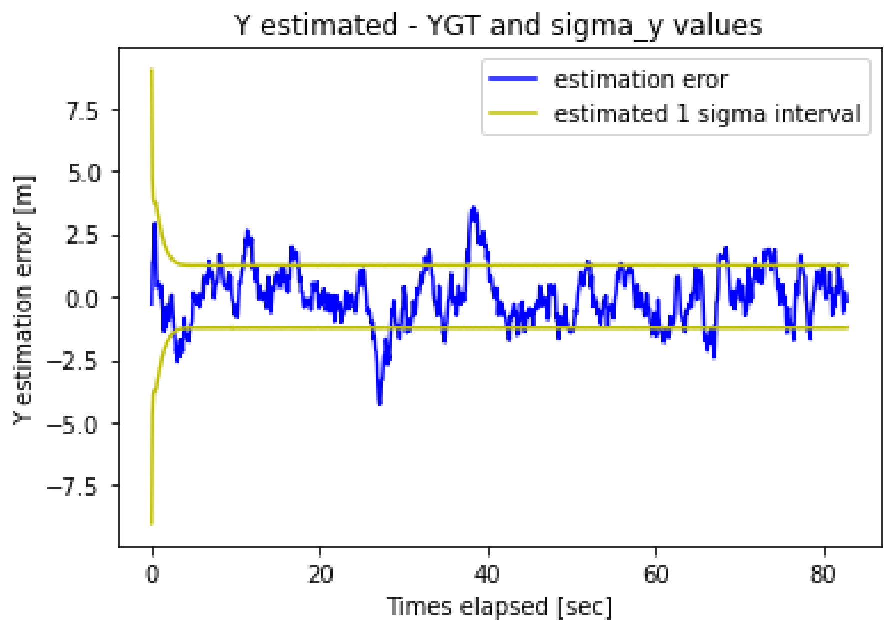


Figure 6: yestimated-yGT and σy values

Here we can see that in both X and Y estimation errors get good performance, around 66% of the error is contained between the corresponding sigma meaning our selection of initialization process noise and measurement noise where correct. Possibly trying even smaller sigma_n could have given on even better result because it mostly controls the converged sigma which could be a bit smaller. Moreover, we see that because where using a linear model the sigmas stay relatively linear (converge to straight lines) and don't change as much as will see in the EKF.

l) Implement constant-acceleration model and compare the results with constant-velocity model:

After implementation we got RMSE const_acc 1.8107176179159632 maxE const acc 5.83407 which is relatively similar to the const velocity model possibly with better calibration wed get a better result because it more simulates the real trajectory were speed is dynamic and acceleration is relatively constant.

Etended Kalman Filter :

a) will use same KITTI GPS/IMU sequence from last section
b) we will extract vehicle GPS trajectory, yaw angle, yaw rate, and forward velocity from KITTI senser packets (OXT).
   - lat: latitude of the oxts-unit (deg)
   - lon: longitude of the oxts-unit (deg)
   - yaw: heading (rad)
   - vf: forward velocity, i.e. parallel to earth-surface (m/s)
   - wz: angular rate around z (rad/s)
   - Extract timestamps from KITTI data and convert them to seconds elapsed from the first one
   These are tereted as GT in this experimant

c) Transform GPS trajectory from [lat, long, alt] to local [x, y, z] cord in order to enable the Kalman filter can handle it.
   Plot ground-truth GPS trajectory:

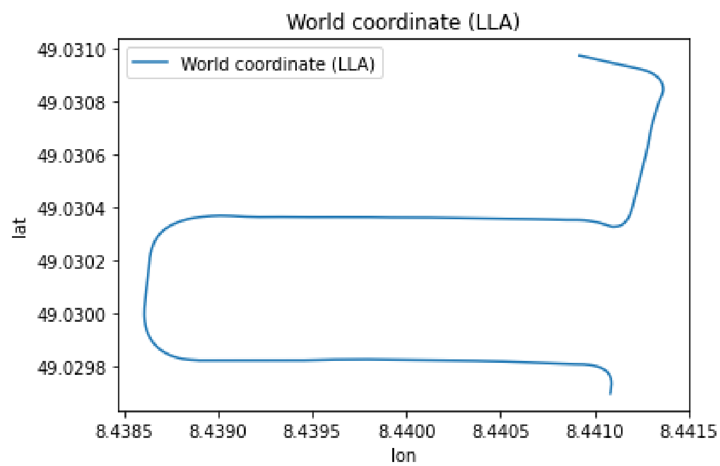


Figure 7: world coordinate (LLA)

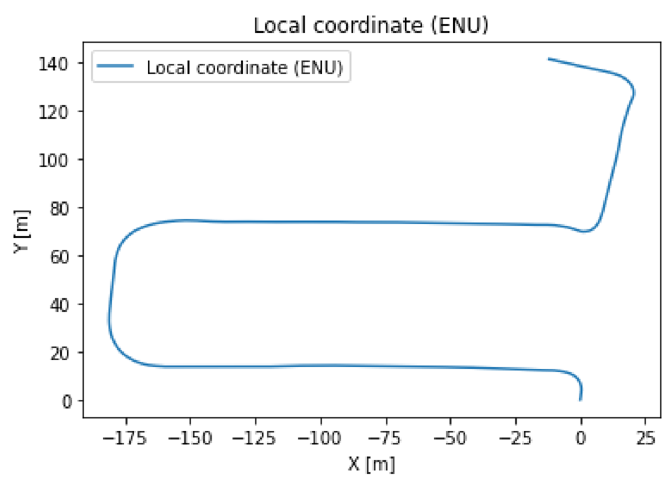


Figure 8: Local coordinate (ENU)

ground-truth yaw angles, yaw rates, and forward velocities:

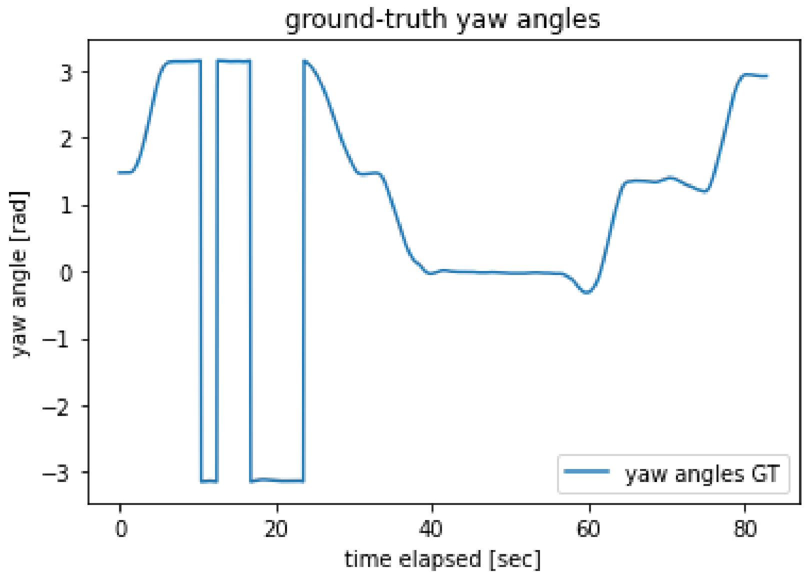


Figure 9: ground-truth yaw angles

We can see the we start with ange pi/2 corresponding to heading north I ENU coordinates then turn left to head west we get angle of pi notice the fluctuation pi and – pi when heading west this is because the wraparound of the angle(-pi, pi) we the go back to angle zero when going east and go up again to pi/2 heading north and then pi again heading west.

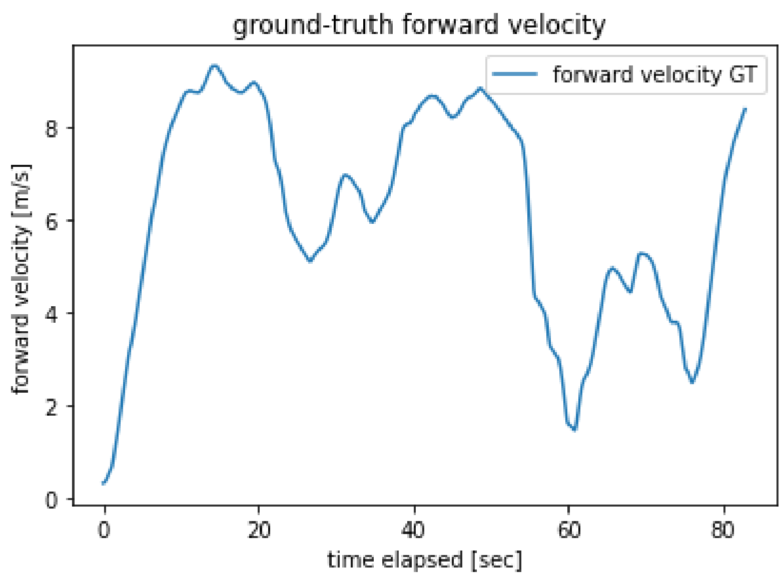


Figure 10: ground-truth forward velocities

Here we see that when driving straight the vehicle increased its speed and slowed down before the turns.

ground-truth yaw rates:

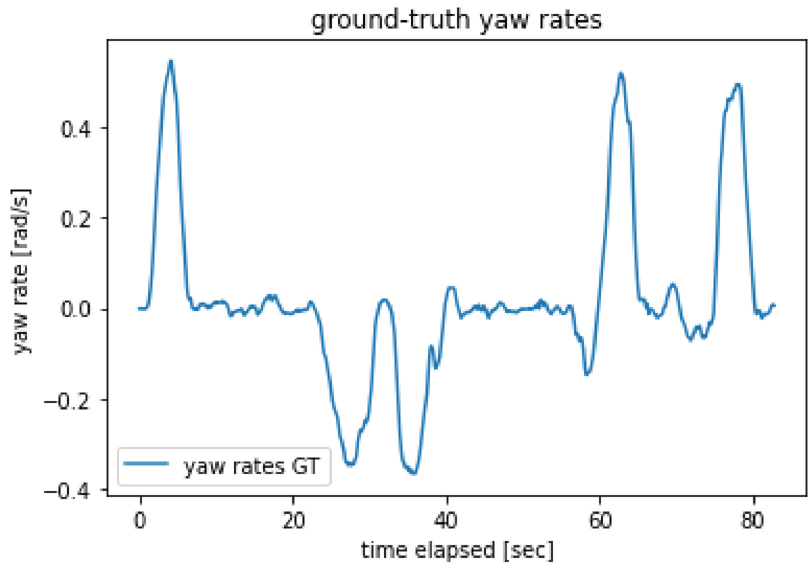


Figure 11: ground-truth yaw rates

Here we can see the change in angle in the turns first turning left (first bump in graph) driving straight then turning right 90 deg then turning right again 90 deg then driving straight and again turning left 90 deg and again left 90 deg this corresponds to the GT trajectory seen in figure ENU figure.

d) Here we added Gaussian noise to the ground-truth GPS data which will be used as noisy observations fed to the Extended Kalman filter. Noise added with standard deviation of observation noise of x and y in meter ($\sigma x = 3$, $\sigma y = 3$).

e) Here will apply an Extended Kalman filter to the GPS sequence in order to estimate the vehicle's 2D pose velocity-based model (non-linear model):
-will suppose initial 2D position [x, y] estimation begins with the first GPS observation
- GPS observation noise of X and Y is known (σx =3, σy =3)
-will Implement an EKF based on velocity-based model and compare results to the constants-velocity model (set the same initial conditions).

In the next figure we can see EKF results with no noise in the commands

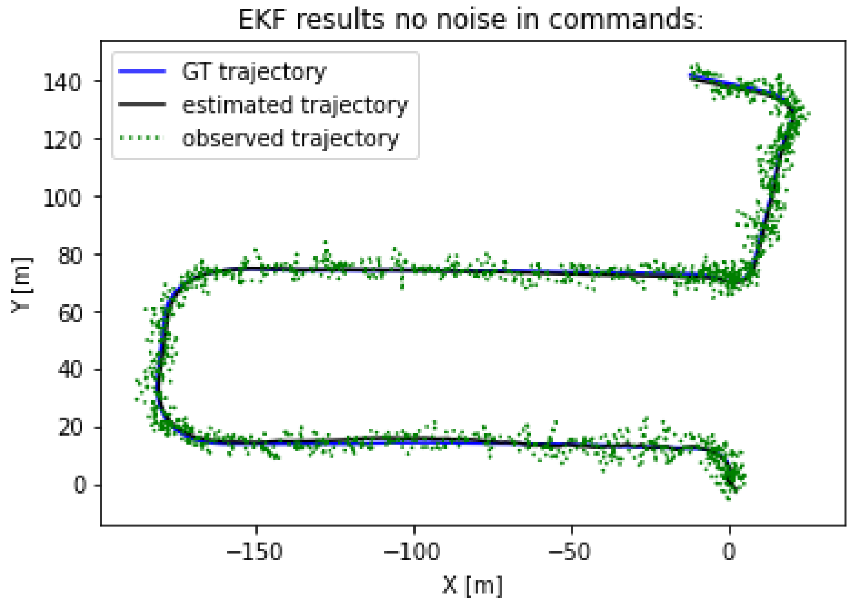


Figure 12: EKF results no noise in commands

In this case we are able to achieve much better results compared to the constant velocity model especially if we look at the turns. Moreover in this case the RMSE = 1.1782 and maxE = 2.3047 (compared to RMSE = 1.5143 maxE = 5.1180 in constant velocity model) this is because the model is much more dynamic and contains nonlinear components that more precisely model the real trajectory (these non linarites in the model are linearized so they will be able to use them in matrix form of Kalman filter and to uphold the Gaussian assumption).

f) Adding gaussian noise to the IMU data:
Will add noise to yaw rates standard deviation of yaw rate in rad/s (σw = 0.2) and plot graphs of GT+ noise yaw rate:

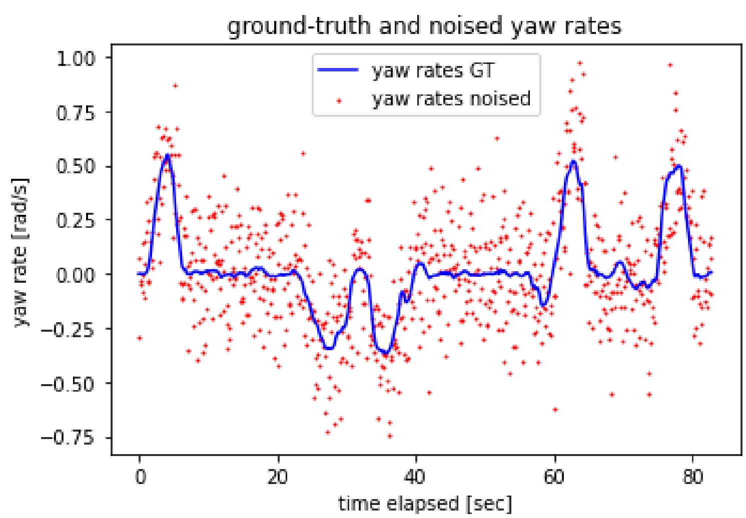


Figure 13: graphs of GT+ noise yaw rate

Will Add noise to forward velocities adding standard deviation of forward velocity in m/s ($\sigma f v$ =2) plot graphs of GT+ noise velocities:

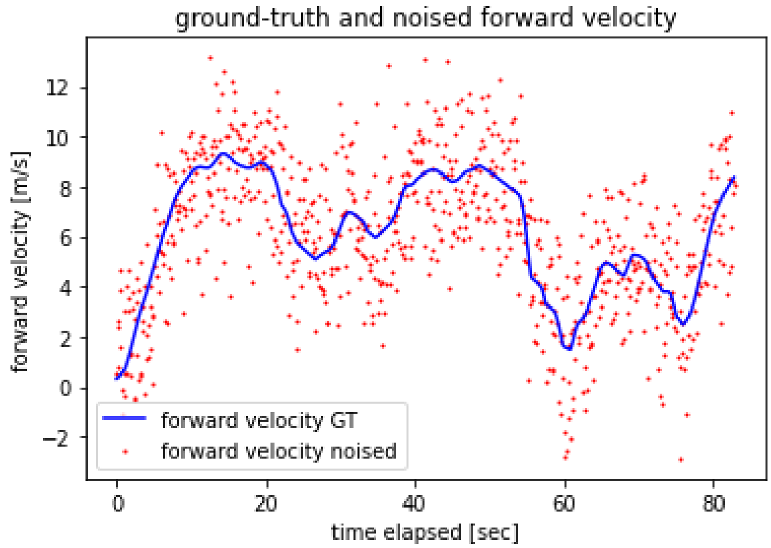


Figure 14: graphs of GT+ noise forward velocities

Our goal was to minimze RMSE while maxE <5:

Find which calibration has the best performance according to the above criteria.

1) Initial conditions: according to first observation

$$\mu_0 = \begin{bmatrix} x_0 \\ y_0 \\ \theta_0 \end{bmatrix}$$

We initialized the mean of our state to be our first observation and the assumed angel of heading north pi/2

$$\Sigma_0 = \begin{bmatrix} \sigma_{x_0}^2 & 0 & 0 \\ 0 & \sigma_{v_0}^2 & 0 \\ 0 & 0 & \sigma_{\theta_0}^2 \end{bmatrix}, \sigma_x = 3, \sigma_v = 2, \sigma_\theta \approx 1.2$$

These values are chosen as such because they are linearized with their Jacobean V_t to fit the uncertainty of x y an theta at the at the beginning of the path.

2) Jacobians G, V and C:

$$G_t = \begin{bmatrix} \frac{dg_1}{dx_1} & \frac{dg_1}{dy_1} & \frac{dg_1}{d\theta_1} \\ \frac{dg_2}{dx_1} & \frac{dg_2}{dy_1} & \frac{dg_2}{d\theta_1} \\ \frac{dg_3}{dx_1} & \frac{dg_3}{dy_1} & \frac{dg_3}{d\theta_1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\frac{v_t}{w_t}\cos(\theta_{t-1}) + \frac{v_t}{w_t}\cos(\theta_{t-1} + w_t\Delta t) \\ 0 & 1 & -\frac{v_t}{w_t}\sin(\theta_{t-1}) + \frac{v_t}{w_t}\sin(\theta_{t-1} + w_t\Delta t) \\ 0 & 0 & 1 \end{bmatrix}$$

$$V_t = \begin{bmatrix} \frac{dg_1}{dv_1} & \frac{dg_1}{dw_1} \\ \frac{dg_2}{dv_1} & \frac{dg_2}{dw_1} \\ \frac{dg_3}{dv_1} & \frac{dg_3}{dw_1} \end{bmatrix} =$$

$$= \begin{bmatrix} -\frac{1}{w_t}\sin(\theta_{t-1}) + \frac{1}{w_t}\sin(\theta_{t-1} + w_t\Delta t) & \frac{v_t}{w_t^2}\sin(\theta_{t-1}) - \frac{v_t}{w_t^2}\sin(\theta_{t-1} + w_t\Delta t) + \frac{v_t}{w_t}\cos(\theta_{t-1} + w_t\Delta t)\Delta t \\ \frac{1}{w_t}\cos(\theta_{t-1}) - \frac{1}{w_t}\cos(\theta_{t-1} + w_t\Delta t) & -\frac{v_t}{w_t^2}\cos(\theta_{t-1}) + \frac{v_t}{w_t^2}\cos(\theta_{t-1} + w_t\Delta t) + \frac{v_t}{w_t}\sin(\theta_{t-1} + w_t\Delta t)\Delta t \\ 0 & \Delta t \end{bmatrix}$$

$$H_t = \begin{bmatrix} \frac{dh_1}{dx_1} & \frac{dh_1}{dy_1} & \frac{dh_1}{d\theta_1} \\ \frac{dh_2}{dx_1} & \frac{dh_2}{dy_1} & \frac{dh_2}{d\theta_1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

These Jacobeans linearize the nonlinear model around our state vector so they will be able to be fed into the kalman filter and still retain the Gaussian assumption.

3) Covariance (Q and R):

$$\tilde{R}_t = \begin{bmatrix} \sigma_v^2 & 0 \\ 0 & \sigma_w^2 \end{bmatrix}$$

$$Q_t = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}$$

$$R_t = V_t \widetilde{R_t} V_t + R_n$$

$$R_n = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \Delta t \end{bmatrix} \sigma_n, \qquad \sigma_n = 1$$

Moreover, in this implementation we added the RMSE and maxE the values of GT – est of the angle theta
Meaning:
maxE = max(|e_x[i]| +|e_y[i]| +|e_theta[i]|)
and the same for RMSE

EKF main routine:

the state mean and uncertainty covariance matrix was initialized as above sections, from here the class ExtendedKalmanFilter inherited class Kalman filter and override the calcRMSE function to contain the theta error, in addition all the same functions where called as in the regular kalmen filter only we added function overloading with the appropriate inputs hence for the EKF the EKF was set to True and the function ran appropriately,

the function the performe_KalmanFilter was called, with EKF =True this function organized the inputs of the initial state and created the list of states and covariance's and then iterated over all time steps and ran the one step one_step_of_KalmanFilter (with EKF = True) and saved the state and covariance's.

the kalman step itself contains

the prediction of the location and uncertainty covariance by calculating:

$$
\begin{aligned}
&\textbf{Extended\_Kalman\_filter}(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)\textbf{:}\\
&\qquad \bar{\mu}_t = g(u_t, \mu_{t-1})\\
&\qquad \bar{\Sigma}_t = G_t\, \Sigma_{t-1}\, G_t^T + R_t\\
&\qquad K_t = \bar{\Sigma}_t\, H_t^T (H_t\, \bar{\Sigma}_t\, H_t^T + Q_t)^{-1}\\
&\qquad \mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))\\
&\qquad \Sigma_t = (I - K_t\, H_t)\, \bar{\Sigma}_t\\
&\qquad return\ \mu_t, \Sigma_t
\end{aligned}
$$

step 1: we predict our mean location based on our nonlinear model (contained in non linear motion model: g)

$$
\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} -\frac{v_t}{\omega_t} sin\theta_{t-1} + \frac{v_t}{\omega_t} sin\,(\theta_{t-1} + \omega_t \Delta t) \\ \frac{v_t}{\omega_t} cos\theta_{t-1} - \frac{v_t}{\omega_t} cos\,(\theta_{t-1} + \omega_t \Delta t) \\ \omega_t \Delta t \end{bmatrix}
$$

step 2: we predict how our uncertainty carries on to the next time step by our point linearized motion (of nonlinear model) calculated in our Jacobean of "g" (meaning G_t) and process noise R which contains the process noise transformed by V and general process noise added to contain the angle in the sigma 1.

step 3: we compute our Kalman gain which controls the emphasis on the deviation between what we predicted and what was measured. Where H_t maps our predicted state to the observed state

Step 4: computes the corrected mean based on the kalman gain computed and deviation of expected location and observation.

Step 5: computes the corrected covariance matrix also based on the kalman gain.

because our model is non linear all transformations most be point state linearized by the jacobian so the assumption of Gaussian distributions will hold throw all transforms.

In addition in each step we normalized the angle to be between –pi and pi

The code:
Q2 -> EXtendedKalmanFilter.performe_KalmanFilter -> iterates over for each time step: one_step_of_KalmanFilter with EKF = True (computes dead reckoning if required)

g) Result analysis:
   1) Ground-truth and estimated results

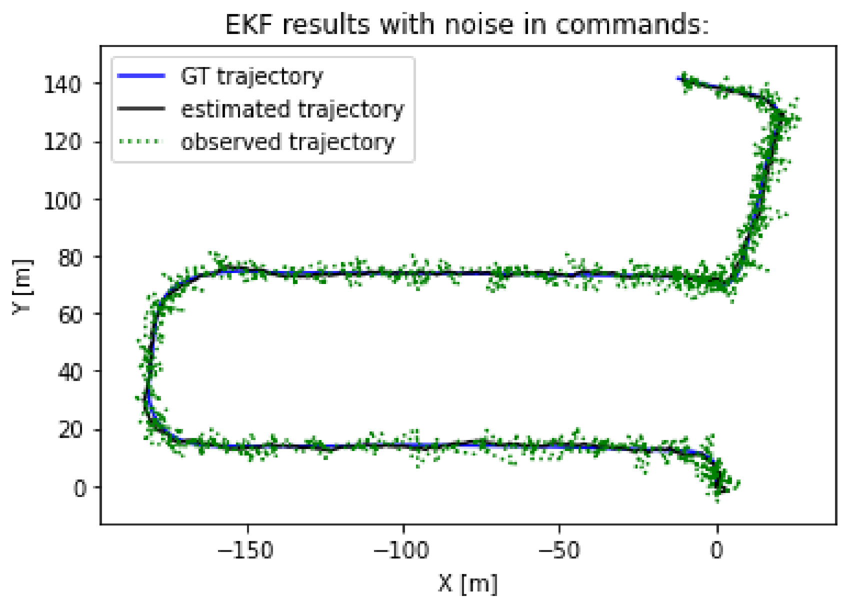


Figure 15: EKF results with noise in commands

   2) Show Kalman filter performance: We can see that even with noise in location and commands we get a relatively good estimate of the GT perhaps lowering sigma_n would have given an even better result and reduce the fluctuation

   3) the minimum values of maxE and RMSE achieved after running different values of sigma_n and initialization cov matrix

      RMSE = 1.1437 , maxE = 3.302 (with out theta error)
      RMSE = 2.2592 maxE 5.4723 (with theta error)

      Hence even with noise in the commands and measerments we still get better results than the regular Kalman filter.

4) In the animation we can see the Trajectory of GT and EKF results and the estimate the trajectory based on the prediction without observation of after ~5 seconds (dead reckoning, Kalman gain=0):
we can see that at the beginning of the animation the covariance is large corresponding to the large initialization and decreases as it starts converge to its variance of location. It is possible that setting the uncertainty of the angle to be higher there would be less jumping in the angles.
In addition, we see that when we set kalman gain to 0 the trajectory becomes skew this is because we are depending only on our motion model (which is nonlinear) and not on the measurements hence we get a trajectory that is not aligned with the observations but still similar to the original trajectory.
5) Plot and analyze the estimated x-y-θ values separately and corresponded sigma value along the trajectory:

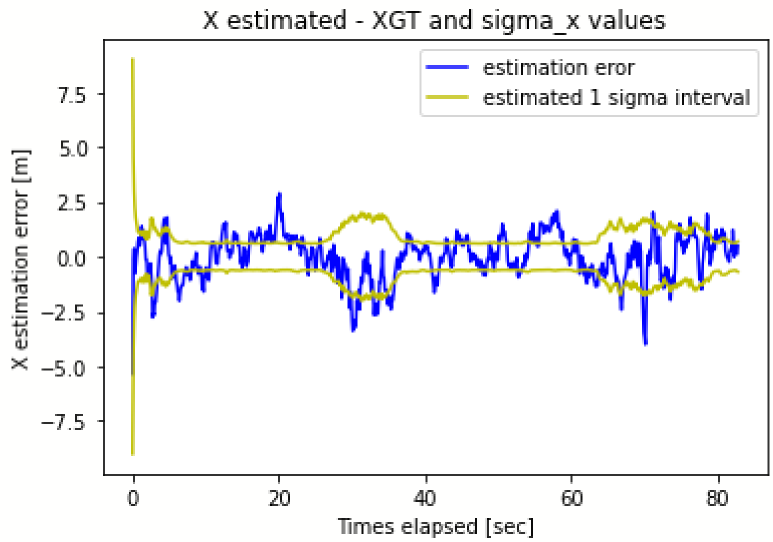


Figure 16: xestimated-xGT and σx values

We can see that 66% of the errors are contained in the estimated 1 sigma interval and that the covariance is dynamic growing larger when ther is a hi difference between observations and predictions.

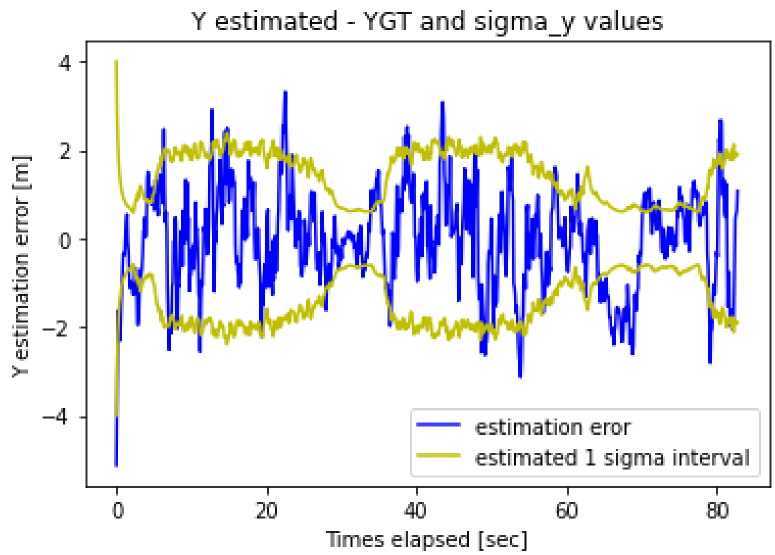


Figure 17: yestimated-yGT and σy values

Again we see that the 66% error is contained within the 1 sigma interval and how the sigma is dynamic.

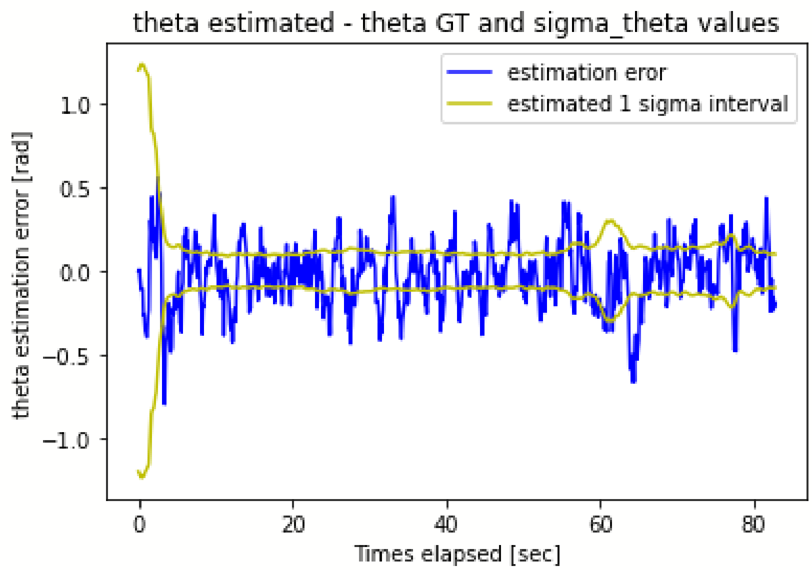


Figure 18: thetaestimated-thetaGT and $\sigma \mathrm{theta}$ values

Again we see that around 66% error is contained within the 1 sigma interval and how the sigma is dynamic however adding more variance to the theta (via sigma_n) might be able to contain a bit more of the error. In addition, we the error here needed to be normalized to account for wraparound of the angle.

EKF - SLAM :

a)

Here we load attached inputs and code Python files.

- Landmarks location
- Odometry and sensor data
- filled in missing parts inside the attached code.

b) Here we will run Odometry data according to odometry model and plot the GT trajectory.

$$\begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} = \begin{pmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{pmatrix} + \begin{pmatrix} \delta_{trans}\cos(\theta_{t-1} + \delta_{rot1}) \\ \delta_{trans}\sin(\theta_{t-1} + \delta_{rot1}) \\ \delta_{rot1} + \delta_{rot2} \end{pmatrix}$$

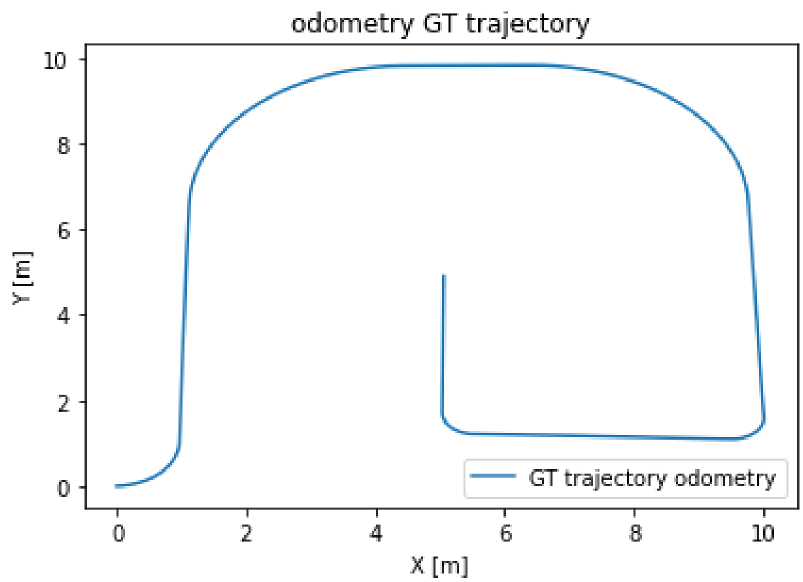


Figure 19: odomatry motion model GT

We start from 0,0 heeding east and start to turn north then wrap around to get to center

c) Here we added Gaussian noise in the motion model assume ($\sigma rot1$=0.01, $\sigma trans$=0.1, $\sigma rot2$=0.01).

Now we will Apply Extended Kalman SLAM filter:
The goal: minimize RMSE while maxE < 1.5:

$$RMSE \triangleq \sqrt{\frac{1}{N}\sum_{i=20}^{N}[e_x^2(i)+e_y^2(i)]}$$

$$e_x(i) \triangleq x_{GT}(i) - x_{Estimate}(i)$$
$$e_y(i) \triangleq y_{GT}(i) - y_{Estimate}(i)$$

$$maxE \triangleq max\{|e_x(i)| + |e_y(i)|\} \quad ,$$

$$20 \le i \le N$$

N is last sample.

d) Initialize initial conditions μ_0, Σ_0:

$$\mu_0 = \begin{bmatrix} x_0 \\ y_0 \\ \theta_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, 2\mathrm{N}+3 \text{ dimensions}$$

Containing the mean of the pose and landmark locations [x,y : for each landmark hence 2N] and
pose = [0.096,0.0101,0.1009]

$$\Sigma_0 = \begin{bmatrix} \sigma_x^2 & 0 & 0 & 0 & \cdots & 0 \\ 0 & \sigma_y^2 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \sigma_\theta^2 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 100 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 100 \end{bmatrix}, \inf \textit{ is set to } 100$$

sigma_x_y_theta = [2,2, 0.6]
all landmark are set to uncertainty of infinity because they have not yet been observed

e) Here we Implement the prediction step of the EKF SLAM algorithm in the function "predict" Use the odometry motion model. We compute the predicted mean:

$$\bar{\mu}_t = \bar{\mu}_{t-1} + F_X^T \begin{pmatrix} \delta_{trans}\cos(\theta_{t-1}+\delta_{rot1}) \\ \delta_{trans}\sin(\theta_{t-1}+\delta_{rot1}) \\ \delta_{rot1}+\delta_{rot2} \end{pmatrix}$$

Where:

$$F_x = \begin{pmatrix} 1 & 0 & 0 & 0\cdots0 \\ 0 & 1 & 0 & 0\cdots0 \\ 0 & 0 & 1 & 0\cdots0 \end{pmatrix}$$

Meaning the new predicted mean is the old predicted mean plus the motion step timed by F_x so it won't affect the land marks

f) We compute its Jacobian $Gt\ x$ to construct the full Jacobian matrix Gt :

$$G_t^x = I + \begin{pmatrix} 0 & 0 & -\delta_{trans}\ \sin(\theta_{t-1} + \delta_{rot1}) \\ 0 & 0 & \delta_{trans}\ \cos(\theta_{t-1} + \delta_{rot1}) \\ 0 & 0 & 0 \end{pmatrix}$$

$$G_t = \begin{pmatrix} G_t^x & 0 \\ 0 & I \end{pmatrix}$$

G_t is the Jacobean of the nonlinear motion model g with respect to u_t-1, that impacts only the pose uncertainty's

g) We compute its Jacobian V to construct the full Jacobian matrix R_t^x and R_t

$$V_t = \begin{bmatrix} \frac{dg_1}{dx_1} & \frac{dg_1}{dy_1} & \frac{dg_1}{d\theta_1} \\ \frac{dg_2}{dx_1} & \frac{dg_2}{dy_1} & \frac{dg_2}{d\theta_1} \\ \frac{dg_3}{dx_1} & \frac{dg_3}{dy_1} & \frac{dg_3}{d\theta_1} \end{bmatrix} = \begin{bmatrix} -\delta_{trans}\sin(\theta + \delta_{rot1}) & \cos(\theta + \delta_{rot1}) & 0 \\ \delta_{trans}\cos(\theta + \delta_{rot1}) & \sin(\theta + \delta_{rot1}) & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\tilde{R} = \begin{bmatrix} \sigma_{rot1}^2 & 0 & 0 \\ 0 & \sigma_{trans}^2 & 0 \\ 0 & 0 & \sigma_{rot2}^2 \end{bmatrix}$$

$$R_t^x = V_t \widetilde{R_t} V_t + R_n$$

$$R_n = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \Delta t \end{bmatrix} \sigma_n, \qquad \sigma_n = 1.3$$

$$R_t = F_x^T R_t^x F_x$$

Where V_t is the Jacobean local linearization of g with respect to u_t also impact the pose only.

The whole prediction step impacts the mean pose and pose uncertainty and dous not effect on the landmarks.
R_t is the process noise from commands
R_n is general process noise
To get full uncertainty covariance we compute:

$$\bar{\Sigma}_t = G_t \ \Sigma_{t-1} \ G_t^T + R_t$$

h) Here we Implement the correction step in the function “update”:
The argument $\boldsymbol{z}$ of this function is a struct array containing $\boldsymbol{m}$ landmark observations made at time step t. Each observation $\boldsymbol{z}(\boldsymbol{i})$ has an id $\boldsymbol{z}(\boldsymbol{i}).\boldsymbol{id}$, a range $\boldsymbol{z}(\boldsymbol{i}).\ \boldsymbol{range}$, and a bearing $\boldsymbol{z}(\boldsymbol{i}).\ \boldsymbol{bearing}$. We will Iterate over all measurements ($\boldsymbol{i}$ = $\mathbf{1}$,.., $\boldsymbol{m}$) and compute the Jacobian $\boldsymbol{Ht\_i}$
we compute a block Jacobian matrix $\boldsymbol{Ht}$ by stacking the $\boldsymbol{Ht\_i}$ matrices corresponding to the individual measurements.
The land mark measurement is:

$$z_t^i = (r_t^i, \phi_t^i)^T$$

The update to the predicted measurement if it is already initialized is:

$$\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{j,x} - \bar{\mu}_{t,x} \\ \bar{\mu}_{j,y} - \bar{\mu}_{t,y} \end{pmatrix}$$

If landmark hasn't been seen before it is initialized to:

$$\begin{pmatrix} \bar{\mu}_{j,x} \\ \bar{\mu}_{j,y} \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{t,x} \\ \bar{\mu}_{t,y} \end{pmatrix} + \begin{pmatrix} r_t^i \ \cos(\phi_t^i + \bar{\mu}_{t,\theta}) \\ r_t^i \ \sin(\phi_t^i + \bar{\mu}_{t,\theta}) \end{pmatrix}$$

The expected observation is then:

$$\hat{z}_t^i = \begin{pmatrix} \sqrt{q} \\ \operatorname{atan2}(\delta_y, \delta_x) - \bar{\mu}_{t,\theta} \end{pmatrix}$$

When q is defined as:

$$q = \ \delta^T \delta$$

In order to treat each landmark separately we use vector F_x,j to decouple them which is defined:

$$F_{x,j} = \begin{pmatrix} 1 & 0 & 0 & 0 \cdots 0 & 0 & 0 & 0 \cdots 0 \\ 0 & 1 & 0 & 0 \cdots 0 & 0 & 0 & 0 \cdots 0 \\ 0 & 0 & 1 & 0 \cdots 0 & 0 & 0 & 0 \cdots 0 \\ 0 & 0 & 0 & 0 \cdots 0 & 1 & 0 & 0 \cdots 0 \\ 0 & 0 & 0 & \underbrace{0 \cdots 0}_{2j-2} & 0 & 1 & \underbrace{0 \cdots 0}_{2N-2j} \end{pmatrix}$$

Ht_i is then:

$$H_t^i = \frac{1}{q} \begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & +\sqrt{q}\delta_x & \sqrt{q}\delta_y \\ \delta_y & -\delta_x & -q & -\delta_y & +\delta_x \end{pmatrix} F_{x,j}$$

Ht_i is the stacked to get the block Jacobin Ht
In general, we are computing the difference between the predicted location and the assumed location of the landmark to achieve the predicted measurement and the Jacobean of the observation with respect to the mean prediction.
The kalman gain is then:

$$K_t^i = \bar{\Sigma}_t \; H_t^{iT} (H_t^i \; \bar{\Sigma}_t \; H_t^{iT} + Q_t)^{-1}$$

While Q_t is the noise in the sensor model is a diagonal matrix with alternating values on its diagonal of $\sigma_r = 0.1$, $\sigma_\theta = 0.01$. (we changed σ_θ from 0.001 adding uncertainty because with it the solution didn't converge

The corrected mean and uncertainty matrix is no calculated by:

$$\bar{\mu}_t = \bar{\mu}_t + K_t^i (z_t^i - \hat{z}_t^i)$$
$$\bar{\Sigma}_t = (I - K_t^i \; H_t^i) \; \bar{\Sigma}_t$$

This is done for each land mark that has been observed in that time step.

i) Analyze results:
1) Here will show the trajectory of EKF-SLAM results.
in an animation, plot covariance matrix of state vector as ellipse.
We can see that the covariance matrix of the pose starts out large and goes down as it recognizes its position from the landmarks the landmarks covariance also goes down as it matches the expected observation however it doesn't go under the uncertainty of the pose

the minimum values of maxE and RMSE achieved is:
RMSE 0.6618343643733547 maxE 1.6603582784285091

As will be seen in the analyzation there is a problem with the uncertainty of the pose angle however we were still able to get a decent result.
2) Analyze estimation error of X, Y and Theta:

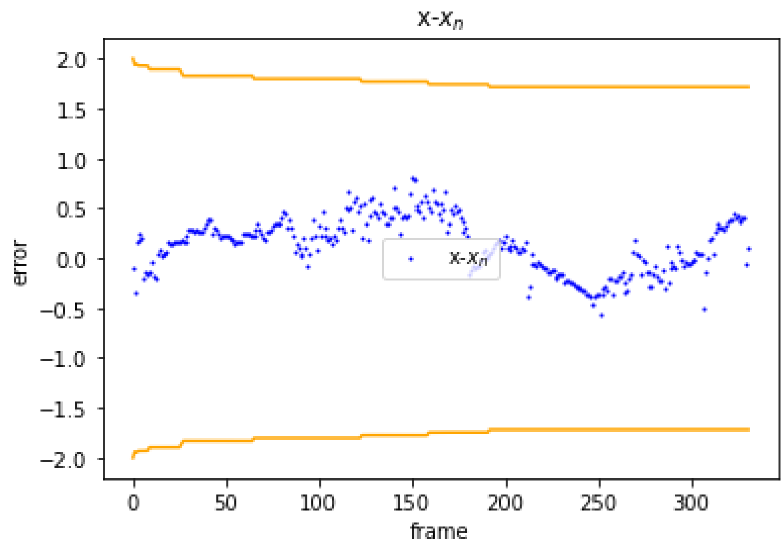


Figure 20: estimation error of x

We can see that more than 66% is in the 1 sigma hence the uncertainty could have been smaller however we can see that the error is pretty small.

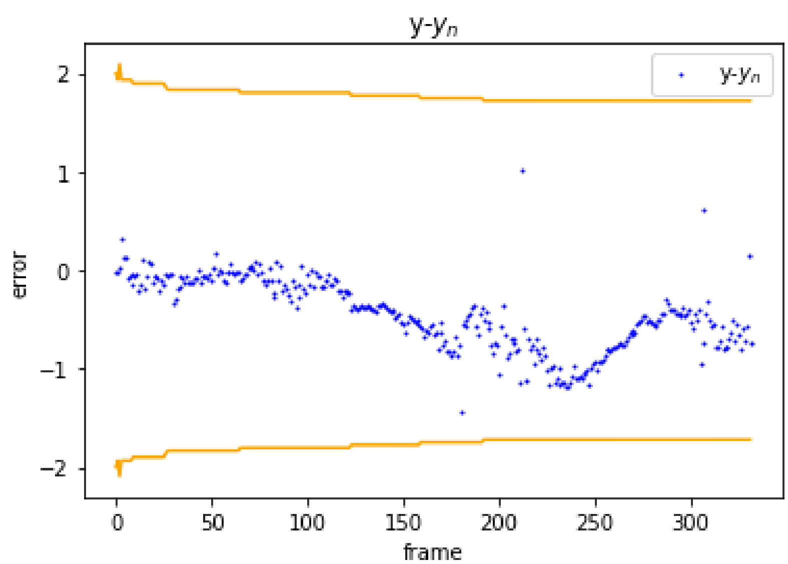


Figure 21: estimation error of y

Again we can see that more than 66% is in the 1 sigma hence the uncertainty could have been smaller however we can see that the error is still pretty small and close to zero.

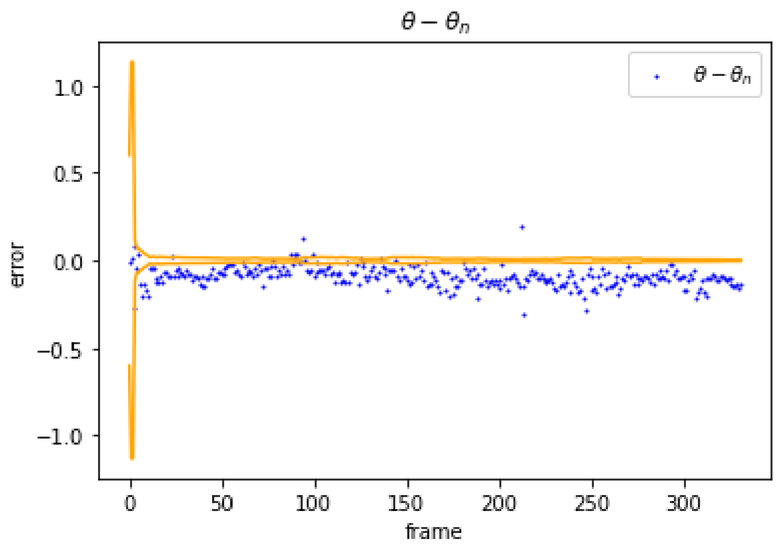


Figure 22: estimation error of theta

Here we can see that our uncertainty was to small when dealing with the angle perhaps adding additional uncertainty to the angle would have gotten better results, however we do see that the error itself is small.

3) Here we picked 2 landmarks and analyzed them:

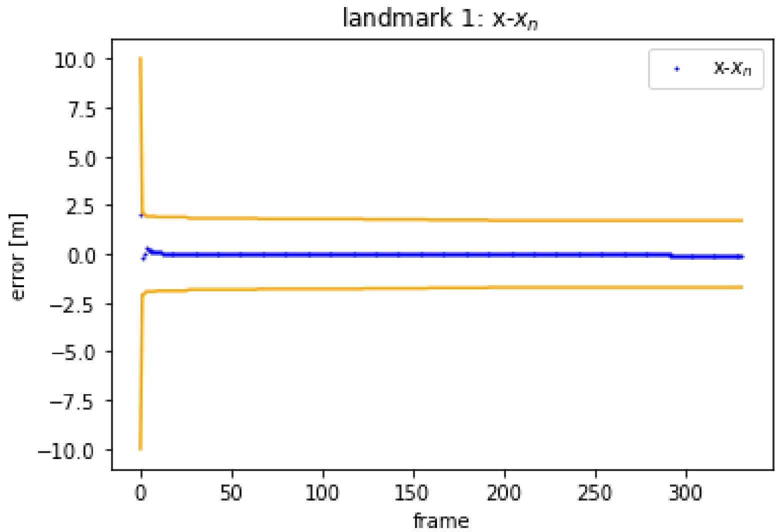


Figure 23: estimation error of landmark 1 x

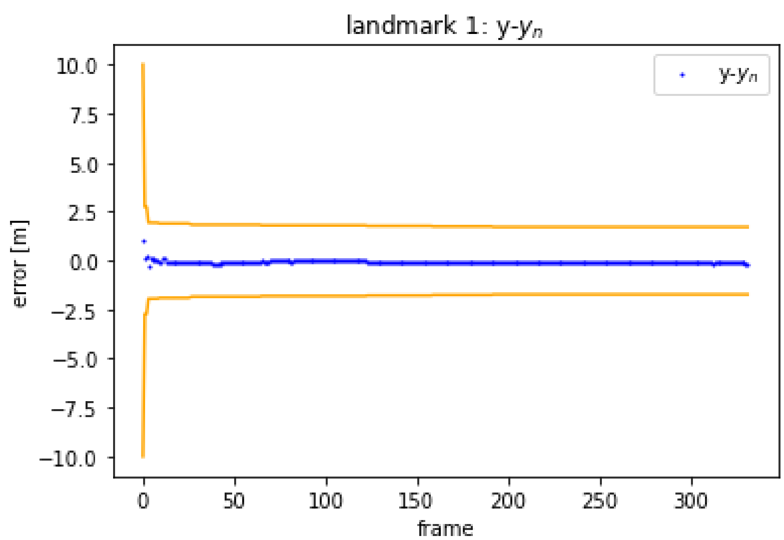


Figure 24: estimation error of landmark 1 y

Here we see that both x and y errors contain more than 66% meaning there covariance's could have been smaller, more over you can see that the uncertainty converges to the uncertainty of the pose.

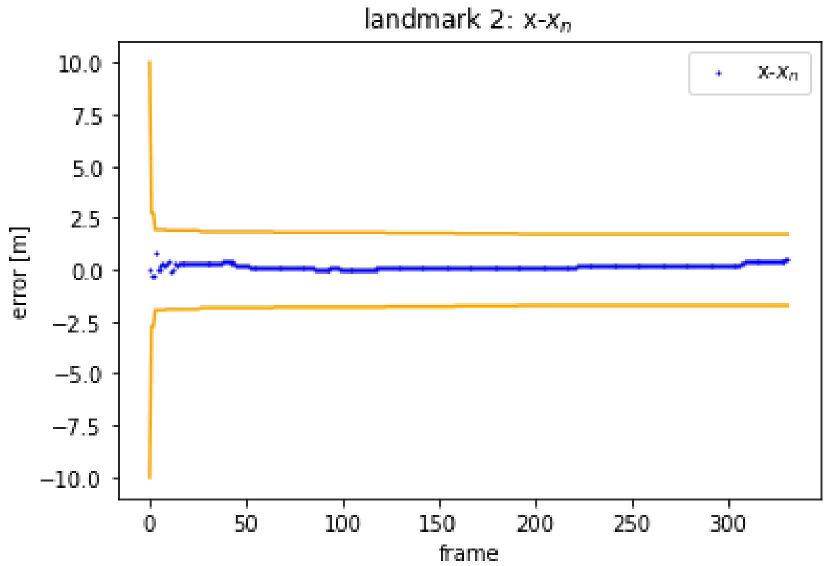


Figure 25: estimation error of landmark 2 x

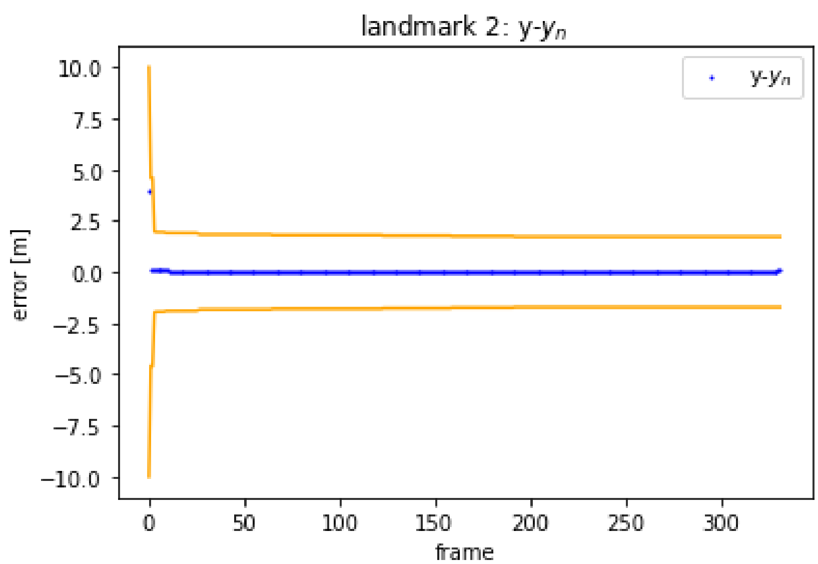


Figure 26: estimation error of landmark 2 y

Also here we can see that the uncertainty could have been smaller and the convergence to pose uncertainty

Summary

In this project we implemented the extended kalman filter and saw how with noisy inputs on the inputs we are able to get a pretty good estimation of the GT trajectory based on a linear constant model ,(in the bonus constant acceleration) we then saw how we can improve this result by using a nonlinear model and using the Jacobeans to point linearize the nonlinear motion model to get an even better estimate and still upholding the gussian assumption, we then saw how to implement EKF SLAM were we both estimate our pose and our landmarks based on noisy measurements of the land marks and nonlinear motion model with noised commands and also here we were able to get a good result of the GT trajectory and land mark estimation.

We saw how different parameters of the process noise R measurement noise Q and intial conditions affect the RMSE error and maxE and how to minimize them.

The result of the EKF SLAM could still do with a little more calibration.

In general, we were able to get a pretty good estimation and have witnessed the power of KF, EKF and EKF SLAM.

Code

Writtin in google colab

```
import os
from data_preparation import *
from kalman_filter import *
import graphs
#import numpy as np


class ProjectQuestions:
    def __init__(self, dataset):
        self.dataset = dataset
        # Extract vehicle GPS trajectory from KITTI OXTS senser packets.
        self.LLA_GPS_trajectory = build_LLA_GPS_trajectory(self.dataset)

        # Transform GPS trajectory from [lat, long, alt] to local [x, y, z
] coordinates in order to enable the Kalman filter to handle them.
        self.ENU_locations_array, self.times_array , self.yaw_vf_wz_array
 = build_GPS_trajectory(self.dataset)
        self.number_of_frames = self.ENU_locations_array.shape[0]

    def Q1(self ,basedir):
        """
        That function runs the code of question 1 of the project.
        Loads from kitti dataset, set noise to GT-
gps values, and use Kalman Filter over the noised values.
        """

        # Plot ground-truth GPS trajectory
        graphs.plot_single_graph(self.LLA_GPS_trajectory,'World coordinate
 (LLA)','lon','lat','World coordinate (LLA)')
        graphs.plot_single_graph(self.ENU_locations_array,'Local coordinat
e (ENU)','X [m]','Y [m]','Local coordinate (ENU)')

        # Add gaussian noise to the ground-truth GPS data:
        sigma_x_y = [3,3]

        self.ENU_locations_array_noised = np.concatenate((add_gaussian_noi
se(self.ENU_locations_array[:,0],sigma_x_y[0]).reshape(self.number_of_fram
es,1),add_gaussian_noise(self.ENU_locations_array[:,1],sigma_x_y[1]).resha
pe(self.number_of_frames,1)),axis = 1)


        # plot on the same graph original GT and observations noise.
```

```
        graphs.plot_graph_and_scatter(self.ENU_locations_array,self.ENU_lo
cations_array_noised,'original GT and observations noise','X [m]','Y [m]',
'ENU_GT', 'ENU_noised',)

        #print(self.ENU_locations_array_noised.shape)

        # sigma_x_y, sigma_Qn =

        predicted_mean_0 = np.array([self.ENU_locations_array_noised[0][0]
,0,self.ENU_locations_array_noised[0][1],1])
        predicted_uncertinty_0 = np.array([ [math.pow(sigma_x_y[0],2),0,0,
0], [0,100,0,0], [0,0,math.pow(sigma_x_y[1],2),0],[0,0,0,100] ])

        #observation_z_t_array = np.concatenate((self.ENU_locations_array_
noised[:,0].reshape(self.number_of_frames,1),np.zeros((self.ENU_locations_
array_noised.shape[0],1)),self.ENU_locations_array_noised[:,1].reshape(sel
f.number_of_frames,1),np.zeros((self.ENU_locations_array_noised.shape[0],1
))),axis = 1)
        observation_z_t_array = self.ENU_locations_array_noised
        #print(observation_z_t_array)
        #print(observation_z_t_array.shape)
        KF = KalmanFilter()

        # KalmanFilter

        maxE_list = []
        RMSE_list = []

        for i in range(1,20,1):
          sigma_n = i
          X_Y_est , uncertinty_cov_list = KalmanFilter.performe_KalmanFilt
er(KF,predicted_mean_0,predicted_uncertinty_0,observation_z_t_array,self.t
imes_array, sigma_x_y, sigma_n)
          X_Y_est = X_Y_est[:,[0,2]]
          #print("uncertinty_cov_list.shape" ,uncertinty_cov_list.shape)
          #print("uncertinty_cov_list: ", uncertinty_cov_list)
          RMSE, maxE = KalmanFilter.calc_RMSE_maxE(self.ENU_locations_arra
y,X_Y_est)
          maxE_list.append(maxE)
          RMSE_list.append(RMSE)

        maxE_array = np.array(maxE_list, ndmin = 2).T
        RMSE_array = np.array(RMSE_list, ndmin = 2).T
        sigma_n_array = np.array([i for i in range(1,20,1)], ndmin = 2).T
```

```
        maxE_array_sigma_n = np.concatenate((sigma_n_array,maxE_array),axis=1)
        RMSE_array_sigma_n = np.concatenate((sigma_n_array, RMSE_array),axis=1)
        graphs.plot_single_graph(maxE_array_sigma_n, 'maxE vs sigma_n', 'sigma_n','maxE', 'maxE vs sigma_n')
        graphs.plot_single_graph(RMSE_array_sigma_n,'RMSE vs sigma_n','sigma_n','RMSE', 'RMSE vs sigma_n')
        print("maxE_array shape", maxE_array.shape)

        sigma_n = np.argmin(maxE_list)+1

        X_Y_est , uncertinty_cov_list = KalmanFilter.performe_KalmanFilter(KF,predicted_mean_0,predicted_uncertinty_0,observation_z_t_array,self.times_array, sigma_x_y, sigma_n)
        X_Y_est = X_Y_est[:,[0,2]]
        print("RMSE " ,RMSE_list[np.argmin(maxE_list)] ,"maxE" , min(maxE_list), 'sigma_n' , sigma_n)



        # build_ENU_from_GPS_trajectory
        graphs.plot_three_graphs( self.ENU_locations_array, X_Y_est,self.ENU_locations_array_noised, 'KF esults:' , 'X [m]', 'Y [m]','GT trajectory', 'estimated trajectory', 'observed trajectory')

        # make dead reckoning after 5 seconds (inserting dead reckoning = true):
        KF_dead_reckoning = KalmanFilter()
        X_Y_est_dead_reckoning , uncertinty_cov_list_dead_reckoning = KalmanFilter.performe_KalmanFilter(KF_dead_reckoning,predicted_mean_0,predicted_uncertinty_0,observation_z_t_array,self.times_array, sigma_x_y, sigma_n, dead_reckoning = True )
        X_Y_est_dead_reckoning = X_Y_est_dead_reckoning[:,[0,2]]

        # make animation from gt est and est dead reckoning:
        X_Y_GT_locations = self.ENU_locations_array[:,:2]
        print("X_Y_GT_locations[0] " , X_Y_GT_locations[0], "X_Y_est[0] ", X_Y_est[0],"X_Y_est_dead_reckoning", X_Y_est_dead_reckoning[0], "uncertinty_cov_list[0] " ,uncertinty_cov_list[0] )
        X_XY_XY_Y_uncertinty_cov_list = uncertinty_cov_list[:,[0,2,8,10]]
        ani = graphs.build_animation(X_Y_GT_locations, X_Y_est, X_Y_est_dead_reckoning, X_XY_XY_Y_uncertinty_cov_list, 'trajectories', 'X [m]', 'Y [m]', 'GT', 'KF_estimat', 'dead reckoning')
```

```
        graphs.save_animation(ani, basedir, 'animation of GT KF estimate a
nd dead reckoning')

        # Plot and analyze the estimated x-
y values separately and corresponded
        # sigma value along the trajectory. (e.g. show in same graph xesti
mated-xGT and
        # σx values and explain your results):
        X_Y_estimated_minus_X_Y_GT = X_Y_est - self.ENU_locations_array[:,
:2]
        times_array = self.times_array.reshape((self.times_array.shape[0],
1))

        X_estimate_minus_X_GT = X_Y_estimated_minus_X_Y_GT[:,0].reshape((X
_Y_estimated_minus_X_Y_GT[:,0].shape[0],1))
        X_estimate_minus_X_GT_and_times = np.concatenate((times_array,X_es
timate_minus_X_GT),axis =1)
        sigma_x = X_XY_XY_Y_uncertinty_cov_list[:,0]
        sigma_x = np.reshape(sigma_x,(sigma_x.shape[0],1))
        sigma_minus_x = (-sigma_x)
        sigma_x_with_times = np.concatenate((times_array, sigma_x),axis =1
)
        sigma_minus_x_with_times = np.concatenate((times_array, sigma_minu
s_x),axis =1)
        graphs.plot_two_graphs_one_double(X_estimate_minus_X_GT_and_times,
sigma_x_with_times,sigma_minus_x_with_times,'X estimated - XGT and sigma_x
 values', 'Times elapsed [sec]', 'X estimation error [m]' ,'estimation ero
r', 'estimated 1 sigma interval')


        #Y_estimate_minus_Y_GT_and_times = hstack(self.times_array, X_Y_es
timated_minus_X_Y_GT[1])
        Y_estimate_minus_Y_GT = X_Y_estimated_minus_X_Y_GT[:,1].reshape((X
_Y_estimated_minus_X_Y_GT[:,1].shape[0],1))
        Y_estimate_minus_Y_GT_and_times = np.concatenate((times_array,Y_es
timate_minus_Y_GT),axis =1)
        sigma_y = X_XY_XY_Y_uncertinty_cov_list[:,3]
        sigma_y = np.reshape(sigma_y,(sigma_y.shape[0],1))
        sigma_minus_y = (-sigma_y)
        sigma_y_with_times = np.concatenate((times_array, sigma_y),axis =1
)
        sigma_minus_y_with_times = np.concatenate((times_array, sigma_minu
s_y),axis =1)
        graphs.plot_two_graphs_one_double(Y_estimate_minus_Y_GT_and_times,
sigma_y_with_times,sigma_minus_y_with_times,'Y estimated - YGT and sigma_y
```

```
 values', 'Times elapsed [sec]', 'Y estimation error [m]' ,'estimation ero
r', 'estimated 1 sigma interval')

        # (bonus! 5%). Implement constant-
acceleration model and compare the
        # results with constant-velocity model
        predicted_mean_0 = np.array([self.ENU_locations_array_noised[0][0]
,0,0,self.ENU_locations_array_noised[0][1],1,0])
        predicted_uncertinty_0 = np.array([ [3*math.pow(sigma_x_y[0],2),0,
0,0,0,0], [0,100,0,0,0,0],[0,0,100,0,0,0] ,[0,0,0,3*math.pow(sigma_x_y[1],
2),0,0],[0,0,0,0,100,0],[0,0,0,0,0,100] ])
        X_Y_est , uncertinty_cov_list = KalmanFilter.performe_KalmanFilter
(KF,predicted_mean_0,predicted_uncertinty_0,observation_z_t_array,self.tim
es_array, sigma_x_y, sigma_n,const_acc =True)
        X_Y_est = X_Y_est[:,[0,3]]
        RMSE, maxE = KalmanFilter.calc_RMSE_maxE(self.ENU_locations_array,
X_Y_est)
        print("RMSE const acc" , RMSE, "maxE const acc" , maxE)




    def Q2(self,basedir):

      # 2.a+2.b performed in intlization


        #2.c Plot ground-truth GPS trajectory Plot ground-
truth yaw angles, yaw rates, and forward velocities
        graphs.plot_single_graph(self.LLA_GPS_trajectory,'World coordinate
 (LLA)','lon','lat','World coordinate (LLA)')
        graphs.plot_single_graph(self.ENU_locations_array,'Local coordinat
e (ENU)','X [m]','Y [m]','Local coordinate (ENU)')

        times_array = self.times_array.reshape((self.times_array.shape[0],
1))
        yaw_array = np.array(self.yaw_vf_wz_array[:,0],ndmin = 2).T
        vf_array = np.array(self.yaw_vf_wz_array[:,1],ndmin = 2).T
        wz_array = np.array(self.yaw_vf_wz_array[:,2],ndmin = 2).T

        yaw_array_and_times = np.concatenate((times_array,yaw_array),axis
= 1)
        vf_array_and_times = np.concatenate((times_array,vf_array),axis =
1)
```

```
        wz_array_and_times = np.concatenate((times_array,wz_array), axis =
 1)

        graphs.plot_single_graph(yaw_array_and_times,'ground-
truth yaw angles', 'time elapsed [sec]','yaw angle [rad]','yaw angles GT'
)
        graphs.plot_single_graph(vf_array_and_times,'ground-
truth forward velocity ', 'time elapsed [sec]','forward velocity [m/s]','f
orward velocity GT' )
        graphs.plot_single_graph(wz_array_and_times,'ground-
truth yaw rates', 'time elapsed [sec]','yaw rate [rad/s]','yaw rates GT' )

        # 2.d. Add gaussian noise to the ground-
truth GPS/IMU data.  Those are used as noisy observations given to Kalman
filter later. standard deviation of observation noise of x and y in meter
        sigma_x_y = [3,3]
        self.ENU_locations_array_noised = np.concatenate((add_gaussian_noi
se(self.ENU_locations_array[:,0],sigma_x_y[0]).reshape(self.number_of_fram
es,1),add_gaussian_noise(self.ENU_locations_array[:,1],sigma_x_y[1]).resha
pe(self.number_of_frames,1)),axis = 1)
        #print(self.ENU_locations_array_noised.shape)
        #X_Y_theta_noised = np.concatenate((self.ENU_locations_array_noise
d,yaw_array), axis =1)

        # intializtion:
        predicted_mean_0 = np.array([self.ENU_locations_array_noised[0][0]
,self.ENU_locations_array_noised[0][1],yaw_array[0][0]])
        predicted_uncertinty_0 = np.array([ [math.pow(sigma_x_y[0],2),0,0]
, [0,math.pow(2,2),0], [0,0,0.0174533] ])

        observation_z_t_array = self.ENU_locations_array_noised
        sigma_n = 0
        ekf = ExtendedKalmanFilter()
        X_Y_theta_est , uncertinty_cov_list = ExtendedKalmanFilter.perform
e_KalmanFilter(ekf,predicted_mean_0,predicted_uncertinty_0,observation_z_t
_array,self.times_array, sigma_x_y, sigma_n, EFK = True, vf_array = vf_arr
ay, wz_array = wz_array)
        X_Y_est = X_Y_theta_est
        #print("self.ENU_locations_array.shape" , self.ENU_locations_array
.shape)
        #print("X_Y_theta_est.shape" , X_Y_theta_est.shape)
        X_Y_theta_GT = np.concatenate((self.ENU_locations_array[:,:2],yaw_
array),axis= 1)
        #print("X_Y_wz_GT shape", X_Y_wz_GT.shape)
        # build_ENU_from_GPS_trajectory
```

```
        graphs.plot_three_graphs( self.ENU_locations_array, X_Y_theta_est,
self.ENU_locations_array_noised, 'EKF results no noise in commands:' , 'X
[m]', 'Y [m]','GT trajectory', 'estimated trajectory', 'observed trajector
y')
        RMSE, maxE = KalmanFilter.calc_RMSE_maxE(X_Y_theta_GT,X_Y_theta_es
t)
        print ("EKF: no noise in command","RMSE: ", RMSE, "maxE: ",maxE )
        # need to update rmse ans maxE calc to meet all gruond truths.


        # f. Add gaussian noise to the IMU data: (5%)
        # Add noise to yaw ratesstandard deviation of yaw rate in rad/s (σ
w =0.2) plot graphs of GT+ noise yaw rate
        wz_array_noised = add_gaussian_noise(self.yaw_vf_wz_array[:,2],0.2
).reshape(self.yaw_vf_wz_array[:,2].shape[0],1)
        #print(wz_array_noised.shape)
        wz_array_noised_and_times = np.concatenate((times_array,wz_array_n
oised),axis = 1)
        graphs.plot_graph_and_scatter(wz_array_and_times, wz_array_noised_
and_times,'ground-
truth and noised yaw rates', 'time elapsed [sec]','yaw rate [rad/s]','yaw
rates GT','yaw rates noised' )

        # Add noise to forward velocitiesadd standard deviation of forward
 velocity in m/s (σfv =2) plot graphs of GT+ noise velocities
        vf_array_noised = add_gaussian_noise(self.yaw_vf_wz_array[:,1],2).
reshape(self.yaw_vf_wz_array[:,1].shape[0],1)
        vf_array_noised_and_times = np.concatenate((times_array,vf_array_n
oised),axis = 1)
        graphs.plot_graph_and_scatter(vf_array_and_times, vf_array_noised_
and_times,'ground-
truth and noised forward velocity', 'time elapsed [sec]','forward velocity
 [m/s]','forward velocity GT','forward velocity noised' )


        X_Y_theta_est , uncertinty_cov_list = ExtendedKalmanFilter.perform
e_KalmanFilter(ekf,predicted_mean_0,predicted_uncertinty_0,observation_z_t
_array,self.times_array, sigma_x_y, sigma_n , EFK = True, vf_array = vf_ar
ray_noised, wz_array = wz_array_noised, sigma_vf = 2, sigma_wz = 0.2, yaw_
rate_and_vf_noised = True)
        X_Y_est = X_Y_theta_est[:,:2]
        graphs.plot_three_graphs( X_Y_theta_GT, X_Y_theta_est,self.ENU_loc
ations_array_noised, 'EKF results with noise in commands:' , 'X [m]', 'Y [
m]','GT trajectory', 'estimated trajectory', 'observed trajectory')
        RMSE, maxE = KalmanFilter.calc_RMSE_maxE(X_Y_theta_GT,X_Y_est)
```

```
        print ("EKF: with noise in command ","RMSE: ", RMSE, "maxE: ",maxE
 )
        sigma_n = 0.12
        maxE_list = []
        RMSE_list = []
        k_array = []
        for i in range(1,20,1):

          k = i
          #predicted_uncertinty_0 = np.array([ [k*math.pow(sigma_x_y[0],2)
,0,0], [0,k*math.pow(sigma_x_y[1],2),0], [0,0,0.3*k] ])
          predicted_uncertinty_0 = np.array([ [k*math.pow(sigma_x_y[0],2),
0,0], [0,k*math.pow(2,2),0], [0,0,0.0174*k] ])

          X_Y_theta_est , uncertinty_cov_list = ExtendedKalmanFilter.perfo
rme_KalmanFilter(ekf,predicted_mean_0,predicted_uncertinty_0,observation_z
_t_array,self.times_array, sigma_x_y, sigma_n , EFK = True, vf_array = vf_
array_noised, wz_array = wz_array_noised, sigma_vf = 2, sigma_wz = 0.2, ya
w_rate_and_vf_noised = True)
          #X_Y_theta_est = X_Y_theta_est[:,[0,3]]
          #print("uncertinty_cov_list.shape" ,uncertinty_cov_list.shape)
          #print("uncertinty_cov_list: ", uncertinty_cov_list)
          RMSE, maxE = ExtendedKalmanFilter.calc_RMSE_maxE(X_Y_theta_GT,X_
Y_theta_est)

          maxE_list.append(maxE)
          RMSE_list.append(RMSE)
          k_array.append(k)
        maxE_array = np.array(maxE_list, ndmin = 2).T
        RMSE_array = np.array(RMSE_list, ndmin = 2).T
        k_array = np.array(k_array, ndmin = 2).T
        maxE_array_k = np.concatenate((k_array,maxE_array),axis=1)
        RMSE_array_k = np.concatenate((k_array, RMSE_array),axis=1)
        graphs.plot_single_graph(maxE_array_k, 'maxE vs coaficiant k', 'k'
,'maxE', 'maxE vs k')
        graphs.plot_single_graph(RMSE_array_k,'RMSE vs coaficaiant k','k',
'RMSE', 'RMSE vs k')
        #get the best estimate:
        k = np.argmin(maxE_list)+1
        #print("k", k "but realy equals 1")
        k=1

        #predicted_uncertinty_0 = np.array([ [k*math.pow(sigma_x_y[0],2),0
,0], [0,k*math.pow(sigma_x_y[1],2),0], [0,0,1.2] ])
```

```
        predicted_uncertinty_0 = np.array([ [k*math.pow(sigma_x_y[0],2),0,
0], [0,k*math.pow(2,2),0], [0,0,1.2] ])

        X_Y_theta_est , uncertinty_cov_list = ExtendedKalmanFilter.perform
e_KalmanFilter(ekf,predicted_mean_0,predicted_uncertinty_0,observation_z_t
_array,self.times_array, sigma_x_y, sigma_n, EFK = True, vf_array = vf_arr
ay_noised, wz_array = wz_array_noised, sigma_vf = 2, sigma_wz = 0.2, yaw_r
ate_and_vf_noised = True)
        RMSE, maxE = ExtendedKalmanFilter.calc_RMSE_maxE(X_Y_theta_GT,X_Y_
theta_est)
        print("RMSE", RMSE, "maxE", maxE)
        #print("theta est list:",X_Y_theta_est[:,2] )
        graphs.plot_single_graph(X_Y_theta_est[:,2],'est yaw angles', 'tim
e elapsed [sec]','yaw angle [rad]','yaw angles GT' )

        #print("RMSE " ,RMSE_list[np.argmin(maxE_list)] ,"maxE" , min(maxE
_list), 'k' , k)
        X_Y_est = X_Y_theta_est[:,:2]

        # build_ENU_from_GPS_trajectory
        graphs.plot_three_graphs( self.ENU_locations_array, X_Y_theta_est,
self.ENU_locations_array_noised, 'EKF results with noise in commands:' , '
X [m]', 'Y [m]','GT trajectory', 'estimated trajectory', 'observed traject
ory')

        # make dead reckoning after 5 seconds (inserting dead reckoning =
true):
        EKF_dead_reckoning = ExtendedKalmanFilter()
        X_Y_theta_est_dead_reckoning , uncertinty_cov_list_dead_reckoning
= ExtendedKalmanFilter.performe_KalmanFilter(ekf,predicted_mean_0,predicte
d_uncertinty_0,observation_z_t_array,self.times_array, sigma_x_y, sigma_n
, EFK = True, vf_array = vf_array_noised, wz_array = wz_array_noised, sigm
a_vf = 2, sigma_wz = 0.2, yaw_rate_and_vf_noised = True, dead_reckoning =
True)
        X_Y_est_dead_reckoning = X_Y_theta_est_dead_reckoning[:,:2]

        # make animation from gt est and est dead reckoning:
        X_Y_GT_locations = self.ENU_locations_array[:,:2]

        #print("uncertinty_cov_list shape", uncertinty_cov_list.shape, "un
certinty_cov_list",uncertinty_cov_list)
        X_XY_XY_Y_uncertinty_cov_list = uncertinty_cov_list[:,[0,1,3,4]]
        print("X_XY_XY_Y_uncertinty_cov_list.shape:" ,X_XY_XY_Y_uncertinty
_cov_list.shape,"X_XY_XY_Y_uncertinty_cov_list",X_XY_XY_Y_uncertinty_cov_l
ist)
```

```
        ani = graphs.build_animation(X_Y_GT_locations, X_Y_est, X_Y_est_de
ad_reckoning, X_XY_XY_Y_uncertinty_cov_list, 'trajectories', 'X [m]', 'Y [
m]', 'GT', 'EKF_estimat', 'dead reckoning')
        graphs.save_animation(ani, basedir, 'animation of GT EKF estimate 
and dead reckoning')


        # Plot and analyze the estimated x-
y values separately and corresponded
        # sigma value along the trajectory. (e.g. show in same graph xesti
mated-xGT and
        # σx values and explain your results):

        X_Y_theta_estimated_minus_X_Y_theta_GT = X_Y_theta_est - X_Y_theta
_GT
        times_array = self.times_array.reshape((self.times_array.shape[0],
1))

        X_estimate_minus_X_GT = X_Y_theta_estimated_minus_X_Y_theta_GT[:,0
].reshape((X_Y_theta_estimated_minus_X_Y_theta_GT[:,0].shape[0],1))
        X_estimate_minus_X_GT_and_times = np.concatenate((times_array,X_es
timate_minus_X_GT),axis =1)
        sigma_x = X_XY_XY_Y_uncertinty_cov_list[:,0]
        sigma_x = np.reshape(sigma_x,(sigma_x.shape[0],1))
        sigma_minus_x = (-sigma_x)
        sigma_x_with_times = np.concatenate((times_array, sigma_x),axis =1
)
        sigma_minus_x_with_times = np.concatenate((times_array, sigma_minu
s_x),axis =1)
        graphs.plot_two_graphs_one_double(X_estimate_minus_X_GT_and_times,
sigma_x_with_times,sigma_minus_x_with_times,'X estimated - XGT and sigma_x
 values', 'Times elapsed [sec]', 'X estimation error [m]' ,'estimation ero
r', 'estimated 1 sigma interval')

        # calc sigma y grapgh
        Y_estimate_minus_Y_GT = X_Y_theta_estimated_minus_X_Y_theta_GT[:,1
].reshape((X_Y_theta_estimated_minus_X_Y_theta_GT[:,1].shape[0],1))
        Y_estimate_minus_Y_GT_and_times = np.concatenate((times_array,Y_es
timate_minus_Y_GT),axis =1)
        sigma_y = X_XY_XY_Y_uncertinty_cov_list[:,3]
        sigma_y = np.reshape(sigma_y,(sigma_y.shape[0],1))
        sigma_minus_y = (-sigma_y)
        sigma_y_with_times = np.concatenate((times_array, sigma_y),axis =1
)
```

```
        sigma_minus_y_with_times = np.concatenate((times_array, sigma_minu
s_y),axis =1)
        graphs.plot_two_graphs_one_double(Y_estimate_minus_Y_GT_and_times,
sigma_y_with_times,sigma_minus_y_with_times,'Y estimated - YGT and sigma_y
 values', 'Times elapsed [sec]', 'Y estimation error [m]' ,'estimation ero
r', 'estimated 1 sigma interval')

        # calc sigma theta grapgh theta = yaw

        theta_estimate_minus_theta_GT = X_Y_theta_estimated_minus_X_Y_thet
a_GT[:,2].reshape((X_Y_theta_estimated_minus_X_Y_theta_GT[:,2].shape[0],1)
)
        index_theta_estimate_minus_theta_GT = theta_estimate_minus_theta_G
T > np.pi
        theta_estimate_minus_theta_GT[index_theta_estimate_minus_theta_GT]
 -= 2*np.pi
        index_theta_estimate_minus_theta_GT = theta_estimate_minus_theta_G
T < -np.pi
        theta_estimate_minus_theta_GT[index_theta_estimate_minus_theta_GT]
 += 2*np.pi
        theta_estimate_minus_theta_GT_and_times = np.concatenate((times_ar
ray,theta_estimate_minus_theta_GT),axis =1)
        sigma_theta = uncertinty_cov_list[:,[8]]
        sigma_theta = np.reshape(sigma_theta,(sigma_theta.shape[0],1))
        sigma_minus_theta = (-sigma_theta)
        sigma_theta_with_times = np.concatenate((times_array, sigma_theta)
,axis =1)
        sigma_minus_theta_with_times = np.concatenate((times_array, sigma_
minus_theta),axis =1)
        graphs.plot_two_graphs_one_double(theta_estimate_minus_theta_GT_an
d_times,sigma_theta_with_times,sigma_minus_theta_with_times,'theta estimat
ed - theta GT and sigma_theta values', 'Times elapsed [sec]', 'theta estim
ation error [rad]' ,'estimation eror', 'estimated 1 sigma interval')


        '''
        # sigma_samples =

        # sigma_vf, sigma_omega =

        # build_LLA_GPS_trajectory

        # add_gaussian_noise to u and measurments (locations_gt[:,i], sigm
a_samples[i])
```

```
        # ekf = ExtendedKalmanFilter(sigma_samples, sigma_vf, sigma_omega)
        # locations_ekf, sigma_x_xy_yx_y_t = ekf.run(locations_noised, tim
es, yaw_vf_wz_noised, do_only_predict=False)

        # RMSE, maxE = ekf.calc_RMSE_maxE(locations_gt, locations_ekf)

        # build_animation
        # save_animation(ani, os.path.dirname(__file__), "ekf_predict")
        '''

    def Q3(self,basedir):

        landmarks = self.dataset.load_landmarks()
        sensor_data_gt = self.dataset.load_sensor_data()
        #print("landmarks.shape" ,landmarks.shape)
        print("landmarks:" ,landmarks)
        #print("sensor_data_gt.shape", sensor_data_gt.shape)
        print("sensor_data_gt:", sensor_data_gt)

        sigma_x_y_theta =[2,2, 0.6]# [0.1,0.1, 0.02] #TODO
        variance_r1_t_r2 = [0.01,0.1,0.01]#TODO

        variance_r_phi = [0.1,0.01] #TODO

        sensor_data_noised = add_gaussian_noise_dict(sensor_data_gt, list(
np.sqrt(np.array(variance_r1_t_r2))))

        import matplotlib.pyplot as plt
        fig = plt.figure()
        ax = fig.add_subplot(111)

        ekf_slam = ExtendedKalmanFilterSLAM(sigma_x_y_theta, variance_r1_t
_r2, variance_r_phi)

        frames, mu_arr, mu_arr_gt, sigma_x_y_t_px1_py1_px2_py2 = ekf_slam.
run(sensor_data_gt, sensor_data_noised, landmarks, ax)

        graphs.plot_single_graph(mu_arr_gt, 'odometry GT trajectory', 'X [
m]', 'Y [m]', 'GT trajectory odometry' )
        maxE = 0
        e_x = mu_arr_gt[20:,0] - mu_arr[20:,0]
        e_y = mu_arr_gt[20:,1] - mu_arr[20:,1]
        maxE = max(abs(e_x)+abs(e_y))
        RMSE = np.sqrt(sum(np.power(e_x,2)+np.power(e_y,2))/(mu_arr_gt.sha
pe[0]-20))
```

```
        print("RMSE", RMSE, "maxE", maxE)
        graphs.plot_single_graph(mu_arr_gt[:,0] - mu_arr[:,0], "x-
$x_n$", "frame", "error", "x-$x_n$",
                                 is_scatter=True, sigma=np.sqrt(sigma_x_y_
t_px1_py1_px2_py2[:,0]))
        graphs.plot_single_graph(mu_arr_gt[:,1] - mu_arr[:,1], "y-
$y_n$", "frame", "error", "y-$y_n$",
                                 is_scatter=True, sigma=np.sqrt(sigma_x_y_
t_px1_py1_px2_py2[:,1]))
        graphs.plot_single_graph(normalize_angles_array(mu_arr_gt[:,2] - m
u_arr[:,2]), "$\\theta-\\theta_n$",
                                 "frame", "error", "$\\theta-\\theta_n$",
                                 is_scatter=True, sigma=np.sqrt(sigma_x_y_
t_px1_py1_px2_py2[:,2]))

        graphs.plot_single_graph((np.tile(landmarks[1][0], mu_arr.shape[0]
) - mu_arr[:,3]),
                                 "landmark 1: x-
$x_n$", "frame", "error [m]", "x-$x_n$",
                                 is_scatter=True, sigma=np.sqrt(sigma_x_y_
t_px1_py1_px2_py2[:,3]))
        graphs.plot_single_graph((np.tile(landmarks[1][1], mu_arr.shape[0]
) - mu_arr[:,4]),
                                 "landmark 1: y-
$y_n$", "frame", "error [m]", "y-$y_n$",
                                 is_scatter=True, sigma=np.sqrt(sigma_x_y_
t_px1_py1_px2_py2[:,4]))

        graphs.plot_single_graph((np.tile(landmarks[2][0], mu_arr.shape[0]
) - mu_arr[:,5]),
                                 "landmark 2: x-
$x_n$", "frame", "error [m]", "x-$x_n$",
                                 is_scatter=True, sigma=np.sqrt(sigma_x_y_
t_px1_py1_px2_py2[:,5]))
        graphs.plot_single_graph((np.tile(landmarks[2][1], mu_arr.shape[0]
) - mu_arr[:,6]),
                                 "landmark 2: y-
$y_n$", "frame", "error [m]", "y-$y_n$",
                                 is_scatter=True, sigma=np.sqrt(sigma_x_y_
t_px1_py1_px2_py2[:,6]))

        ax.set_xlim([-2, 12])
        ax.set_ylim([-2, 12])

        from matplotlib import animation
```

```
        ani = animation.ArtistAnimation(fig, frames, repeat=False)
        graphs.show_graphs()
        #ani.save('im.mp4', metadata={'artist':'me'})
        graphs.save_animation(ani, basedir, 'animation of Trajectory of EK
F-SLAM')
    '''
    def run(self):
        self.Q1()
    '''
```

```
import numpy as np
import math
import matplotlib.pyplot as plt
#from utils.plot_state import plot_state
from plot_state import plot_state

from data_preparation import normalize_angle, normalize_angles_array


class KalmanFilter:

  #TODO
  def one_step_of_KalmanFilter(self,previus_mean_t_minus_1,
                                uncertinty_of_previus_belef_t_minus_1,
                                observation_z_t,
                                delta_t,
                                sigma_x_y,
                                sigma_n,
                                const_acc = False,
                                EKF = False,
                                yaw_rate_and_vf_noised = False,
                                vf_t = None,
                                wz_t = None,
                                sigma_vf = 1,
                                sigma_wz = 1,
                                set_kalman_gain_to_zero = False,
                                control_command_t = None ):

      A_t = np.array([[1,delta_t,0,0],[0,1,0,0],[0,0,1,delta_t],[0,0,0,1]]
)
      C_t = np.array([[1,0,0,0],[0,0,1,0]])
```

```
      R_t = np.array([[0,0,0,0],[0,delta_t,0,0],[0,0,0,0],[0,0,0,delta_t]]
)*math.pow(sigma_n,2)
      Q_t = np.array([[sigma_x_y[0]*sigma_x_y[0],0],[0,sigma_x_y[1]*sigma_
x_y[1]]])
      B_t = None
      if(const_acc == True):
        A_t = np.array([[1,delta_t,np.power(delta_t,2)/2,0,0,0],[0,1,delta
_t,0,0,0],[0,0,1,0,0,0],[0,0,0,1,delta_t,np.power(delta_t,2)/2],[0,0,0,0,1
,delta_t],[0,0,0,0,0,1]])
        C_t = np.array([[1,0,0,0,0,0],[0,0,0,1,0,0]])
        R_t = np.array([[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,delta_t,0,0,0],[0
,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,delta_t]])*math.pow(sigma_n,2)
        Q_t = np.array([[sigma_x_y[0]*sigma_x_y[0],0],[0,sigma_x_y[1]*sigm
a_x_y[1]]])
        B_t = None

      # predictin step:

      if (EKF == True):
        #previus_mean_t_minus_1[2][0] = np.clip(previus_mean_t_minus_1[2][
0], a_min = - math.pi, a_max = math.pi)
        if(previus_mean_t_minus_1[2][0] > (math.pi)):
          previus_mean_t_minus_1[2][0] = previus_mean_t_minus_1[2][0] - 2*
math.pi
        if(previus_mean_t_minus_1[2][0] < - (math.pi)):
          previus_mean_t_minus_1[2][0] = previus_mean_t_minus_1[2][0] + 2*
math.pi

        #  g(control_command_t,previus_mean_t_minus_1)
        # A_t is G_T
        yaw_t_minus_1 = previus_mean_t_minus_1[2][0]
        v_cos_theta_devided_by_w = float(vf_t*np.cos(yaw_t_minus_1)/wz_t)
        v_cos_theta_plus_w_delta_t_devided_by_w = float(vf_t*np.cos(yaw_t_
minus_1 + wz_t*delta_t)/wz_t)
        v_sin_theta_devided_by_w = float(vf_t*np.sin(yaw_t_minus_1)/wz_t)
        v_sin_theta_plus_w_delta_t_devided_by_w = float(vf_t*np.sin(yaw_t_
minus_1 + wz_t*delta_t)/wz_t)

        C_t = np.array([[1,0,0],[0,1,0]])
        V_t = np.array([[ -
v_sin_theta_devided_by_w/vf_t + v_sin_theta_plus_w_delta_t_devided_by_w/vf
_t, (v_sin_theta_devided_by_w - v_sin_theta_plus_w_delta_t_devided_by_w)/w
z_t + v_cos_theta_plus_w_delta_t_devided_by_w*delta_t ], [v_cos_theta_devi
ded_by_w/vf_t - v_cos_theta_plus_w_delta_t_devided_by_w/vf_t, (-
v_cos_theta_devided_by_w + v_cos_theta_plus_w_delta_t_devided_by_w)/wz_t +
```

```
 v_sin_theta_plus_w_delta_t_devided_by_w*delta_t], [0,delta_t]],dtype = fl
oat)

        #print("V_t shape:", V_t.shape , V_t)
        A_t  = np.array([[1, 0, -
v_cos_theta_devided_by_w + v_cos_theta_plus_w_delta_t_devided_by_w ], [0,
1, -
v_sin_theta_devided_by_w + v_sin_theta_plus_w_delta_t_devided_by_w], [0,0,
1]], dtype = float)
        if (yaw_rate_and_vf_noised):
          R_t_top =  np.array([[sigma_vf*sigma_vf,0],[0, sigma_wz*sigma_wz
]])

        else:
          R_t_top = np.array([[0,0],[0,0]])
        R_t = np.dot(V_t,np.dot(R_t_top,V_t.T))  +  np.array([[0,0,0],[0,0
,0],[0,0,delta_t]])*sigma_n
        predicted_mean_t = previus_mean_t_minus_1 + np.array([-
vf_t*np.sin(yaw_t_minus_1)/wz_t + vf_t*np.sin(yaw_t_minus_1 + wz_t*delta_t
)/wz_t, vf_t*np.cos(yaw_t_minus_1)/wz_t - vf_t*np.cos(yaw_t_minus_1 + wz_t
*delta_t)/wz_t, normalize_angle(wz_t*delta_t)])

      else:

        predicted_mean_t = np.dot(A_t,previus_mean_t_minus_1) # + np.matmu
l(B_t,control_command_t)


      predicted_uncertinty_t = np.matmul(A_t, np.matmul(uncertinty_of_prev
ius_belef_t_minus_1,A_t.T)) + R_t

      # correction step:
      if set_kalman_gain_to_zero == True:
        kalman_gain_t = np.zeros((4,2))
        if EKF== True:
          kalman_gain_t = np.zeros((3,2))
      else:
        kalman_gain_input = np.matmul(C_t, np.matmul(predicted_uncertinty_
t, C_t.T)) + Q_t
        #kalman_gain_input = np.array(kalman_gain_input)
        #print (" kalman_gain_input shape " , kalman_gain_input.shape, kal
man_gain_input)
        kalman_gain_t = np.matmul(predicted_uncertinty_t,np.matmul(C_t.T,n
p.linalg.inv(kalman_gain_input)))
```

```
      corrected_mean_t = predicted_mean_t +np.matmul(kalman_gain_t,(observ
ation_z_t-np.matmul(C_t, predicted_mean_t)))
      if EKF == True:
        corrected_mean_t[2] = normalize_angle(corrected_mean_t[2])

      I = np.identity(predicted_mean_t.shape[0])
      corrected_uncertinty_t = np.matmul(I -
np.matmul(kalman_gain_t,C_t),predicted_uncertinty_t)
      corrected_uncertinty_t[2][2] = np.clip(corrected_uncertinty_t[2][2],
 a_min = 0,a_max = 2*math.pi)
      return corrected_mean_t , corrected_uncertinty_t


  def performe_KalmanFilter(self,predicted_mean_0, predicted_uncertinty_0,
 observation_z_t_array,times_array ,sigma_x_y, sigma_n,const_acc = False,
EFK = False,
                                  yaw_rate_and_vf_noised = False,
                                  vf_array = None,
                                  wz_array = None,
                                  sigma_vf = 1,
                                  sigma_wz = 1,
                                  dead_reckoning = False):
    X_Y_est = []
    previus_mean_t_minus_1 = predicted_mean_0.T.reshape(predicted_mean_0.T
.shape[0],1)

    X_Y_est.append(np.squeeze(predicted_mean_0))
    uncertinty_cov_list = []
    uncertinty_of_previus_belef_t_minus_1 = predicted_uncertinty_0
    uncertinty_cov_list.append(np.squeeze(uncertinty_of_previus_belef_t_mi
nus_1).flatten())
    set_kalman_gain_to_zero = False
    vf_t = None
    wz_t = None
    for i in range(observation_z_t_array.shape[0]-1):
      #for i in range(4):
      delta_t = times_array[i+1] - times_array[i]
      #print("times_array[i]", times_array[i])
      if (dead_reckoning and (times_array[i] > 5)) :
        set_kalman_gain_to_zero = True
      if(EFK == True):
        vf_t = vf_array[i]
        wz_t = wz_array[i]

      observation_z_t = observation_z_t_array[i+1].T.reshape((2,1))
```

```
        corrected_mean_i , corrected_uncertinty_i = self.one_step_of_KalmanF
ilter(previus_mean_t_minus_1,uncertinty_of_previus_belef_t_minus_1, observ
ation_z_t, delta_t, sigma_x_y, sigma_n,const_acc, EFK, yaw_rate_and_vf_noi
sed, vf_t, wz_t, sigma_vf, sigma_wz, set_kalman_gain_to_zero)
        #print("corrected_mean_i shape " , corrected_mean_i)
        #print("corrected_uncertinty_i shape", corrected_uncertinty_i)
        X_Y_est.append(np.squeeze(corrected_mean_i))
        previus_mean_t_minus_1 = corrected_mean_i
        uncertinty_of_previus_belef_t_minus_1 = corrected_uncertinty_i
        uncertinty_cov_list.append(np.squeeze(corrected_uncertinty_i).flatte
n())
        #print("X_Y_est", X_Y_est)
        #print("uncertinty_cov_list",uncertinty_cov_list)

    return np.array(X_Y_est) ,np.array(uncertinty_cov_list)



  #@staticmethod
  def calc_RMSE_maxE(X_Y_GT, X_Y_est):
        """
        That function calculates RMSE and maxE

        Args:
            X_Y_GT (np.ndarray): ground truth values of x and y
            X_Y_est (np.ndarray): estimated values of x and y

        Returns:
            (float, float): RMSE, maxE
        """
        maxE = 0
        e_x = X_Y_GT[100:,0] - X_Y_est[100:,0]
        e_y = X_Y_GT[100:,1] - X_Y_est[100:,1]
        maxE = max(abs(e_x)+abs(e_y))
        RMSE = np.sqrt(sum(np.power(e_x,2)+np.power(e_y,2))/(X_Y_GT.shape[0]
-100))





        return RMSE, maxE


class ExtendedKalmanFilter(KalmanFilter):
```

```
    #TODO

    @staticmethod
    def calc_RMSE_maxE(X_Y_GT, X_Y_est):
      """
      That function calculates RMSE and maxE

      Args:
          X_Y_GT (np.ndarray): ground truth values of x and y
          X_Y_est (np.ndarray): estimated values of x and y

      Returns:
          (float, float): RMSE, maxE
      """
      maxE = 0
      e_x = X_Y_GT[100:,0] - X_Y_est[100:,0]
      e_y = X_Y_GT[100:,1] - X_Y_est[100:,1]
      e_yaw = X_Y_GT[100:,2] - X_Y_est[100:,2]
      abs_e_yaw = abs(e_yaw)
      index_e_yaw = abs_e_yaw > np.pi
      #print(index_e_yaw)
      abs_e_yaw[index_e_yaw] -= 2*np.pi
      #print(e_yaw)
      maxE = max(abs(e_x) + abs(e_y) + abs(abs_e_yaw))
      RMSE = np.sqrt(sum(np.power(e_x,2)+np.power(e_y,2) + np.power(e_yaw,
2))/(X_Y_GT.shape[0]-100))
      return RMSE, maxE



class ExtendedKalmanFilterSLAM:
    def __init__(self, sigma_x_y_theta, variance_r1_t_r2, variance_r_phi):
        self.sigma_x_y_theta = sigma_x_y_theta  #TODO
        self.variance_r_phi = variance_r_phi   #TODO
        #self.R_x = np.array([[np.power(variance_r1_t_r2[0],2), 0, 0], [0,
np.power(variance_r1_t_r2[1],2),0], [0, 0, np.power(variance_r1_t_r2[2],2)
]])  #TODO cheek again!!!!!
        self.R_x = np.array([[np.power(variance_r1_t_r2[0],2), 0, 0], [0,n
p.power(variance_r1_t_r2[1],2),0], [0, 0, np.power(variance_r1_t_r2[2],2)]
])  #TODO cheek again!!!!!

    def predict(self, mu_prev, sigma_prev, u, N):
        # Perform the prediction step of the EKF
        # u[0]=translation, u[1]=rotation1, u[2]=rotation2
```

```
        delta_trans, delta_rot1, delta_rot2 = u['t'], u['r1'], u['r2']
#TODO
        #print("mu_prev", mu_prev.shape)
        #print(mu_prev[2])
        theta_prev = normalize_angle(mu_prev[2])   #TODO

        F = np.hstack((np.identity(3),np.zeros((3,2*N))))     #TODO
        G_x = np.identity(3) + np.array([[0, 0, -
delta_trans*np.sin(theta_prev + delta_rot1)], [0, 0, delta_trans*np.cos(th
eta_prev + delta_rot1)], [0,0,0]])  #TODO jacobian of motion
        G = np.vstack((np.hstack((G_x,np.zeros((3,2*N)))) , np.hstack((np.
zeros((2*N,3)),np.identity(2*N)))))    #TODO decide size of I and replace
it altenitive id to add matrix rows and vetores to G_x step 4 in slideshow
        V = np.array([[-
delta_trans*np.sin(theta_prev + delta_rot1), np.cos(theta_prev + delta_rot
1), 0], [delta_trans*np.cos(theta_prev + delta_rot1), np.sin(theta_prev +
delta_rot1),0], [1,0,1]])  #TODO
        R_hat_x = np.dot(V,np.dot(self.R_x,V.T)) + np.array([[0,0,0],[0,0,
0],[0,0,1.3]])
        mu_est = mu_prev + np.dot(F.T, np.array([ delta_trans * np.cos(the
ta_prev + delta_rot1), delta_trans * np.sin(theta_prev + delta_rot1) , nor
malize_angle(delta_rot1 + delta_rot2)]).T)     #TODO step3 in slideshow
        sigma_est = np.dot(G, np.dot(sigma_prev, G.T)) + np.vstack((np.hst
ack((R_hat_x, np.zeros((3,2*N)))), np.zeros((2*N,2*N+3))))     #TODO  + n
p.dot(F.T, np.dot(R_hat_x, F))

        return mu_est, sigma_est

    def update(self, mu_pred, sigma_pred, z, observed_landmarks, N):
        # Perform filter update (correction) for each odometry-
observation pair read from the data file.
        mu = mu_pred.copy()      #mu is [m_j_x,m_j_y] of all landmarks
        sigma = sigma_pred.copy()
        theta = mu[2]

        m = len(z["id"])
        Z = np.zeros(2 * m)
        z_hat = np.zeros(2 * m)
        H = None

        for idx in range(m):
            j = z["id"][idx] - 1
            r = z["range"][idx]
            phi = z["bearing"][idx]
```

```
            mu_j_x_idx = 3 + j*2
            mu_j_y_idx = 4 + j*2
            Z_j_x_idx = idx*2
            Z_j_y_idx = 1 + idx*2

            if observed_landmarks[j] == False:
                mu[mu_j_x_idx: mu_j_y_idx + 1] = mu[0:2] + np.array([r * n
p.cos(phi + theta), r * np.sin(phi + theta)])
                observed_landmarks[j] = True

            Z[Z_j_x_idx : Z_j_y_idx + 1] = np.array([r, phi])

            delta = mu[mu_j_x_idx : mu_j_y_idx + 1] - mu[0 : 2]
            q = delta.dot(delta)
            z_hat[Z_j_x_idx : Z_j_y_idx + 1] = np.array([np.sqrt(q),  norm
alize_angle(np.arctan2(delta[1],delta[0]) - theta)]) #.T       #TODO expec
ted observation of landmark j


            I = np.diag(5*[1])
            F_j = np.hstack((I[:,:3], np.zeros((5, 2*j)), I[:,3:], np.zero
s((5, 2*N-2*(j+1)))))

            Hi = np.dot([[-np.sqrt(q)*delta[0], -
np.sqrt(q)*delta[1], 0, np.sqrt(q)*delta[0], np.sqrt(q)*delta[1]],[delta[1
], -delta[0], -q, -delta[1], delta[0]]], F_j)/q    #TODO

            if H is None:
                H = Hi.copy()
            else:
                H = np.vstack((H, Hi))

        Q = np.zeros((H.shape[0],H.shape[0]))
        np.fill_diagonal(Q, [np.power(self.variance_r_phi[0],2),np.power(s
elf.variance_r_phi[1],2)] ) #TODO


        #print("sigma_pred ",sigma_pred.shape)
        #print("H:" , H.shape)
        #print("hi", Hi.shape)
        S = np.linalg.inv(np.dot(H,np.dot(sigma_pred,H.T)) + Q) #TODO
        K = np.dot(sigma_pred,np.dot(H.T,S)) #
        #print("k", K.shape)
```

```
        #print("Z" , Z.shape)
        #print("z_hat", z_hat.shape)
        diff =  Z - z_hat   #TODO

        diff[1::2] = normalize_angles_array(diff[1::2])

        mu = mu + K.dot(diff)
        sigma = np.dot((np.identity(2*N+3) - np.dot(K,H)),sigma_pred)  #TO
DO uncertinty matrix of full cov

        mu[2] = normalize_angle(mu[2])

        # Remember to normalize the bearings after subtracting!
        # (hint: use the normalize_all_bearings function available in tool
s)

        # Finish the correction step by computing the new mu and sigma.
        # Normalize theta in the robot pose.


        return mu, sigma, observed_landmarks

    def run(self, sensor_data_gt, sensor_data_noised, landmarks, ax):
        # Get the number of landmarks in the map
        N = len(landmarks)

        # Initialize belief:
        # mu: 2N+3x1 vector representing the mean of the normal distributi
on
        # The first 3 components of mu correspond to the pose of the robot
,
        # and the landmark poses (xi, yi) are stacked in ascending id orde
r.
        # sigma: (2N+3)x(2N+3) covariance matrix of the normal distributio
n


        init_inf_val = 100 #TODO

        #mu_arr = [np.hstack((self.sigma_x_y_theta,np.zeros(2*N))).T] #TOD
O

        mu_arr = [np.hstack(([0.096,0.0101,0.01009],np.zeros(2*N))).T] #TO
DO
        #mu_arr = [np.hstack(([0,0,0],np.zeros(2*N))).T]
```

```
        sigma_prev = np.vstack((np.hstack(([[np.power(self.sigma_x_y_theta
[0],2), 0, 0], [0, np.power(self.sigma_x_y_theta[1],2), 0], [0, 0, np.powe
r(self.sigma_x_y_theta[2],2)]],np.zeros((3,2*N)))), np.hstack((np.zeros((2
*N,3)),init_inf_val*np.identity(2*N)))))   #TODO

        # sigma for analysis graph sigma_x_y_t + select 2 landmarks
        landmark1_ind= 3    #TODO
        landmark2_ind= 4 #TODO

        Index=[0,1,2,landmark1_ind,landmark1_ind+1,landmark2_ind,landmark2
_ind+1]
        sigma_x_y_t_px1_py1_px2_py2 = sigma_prev[Index,Index].copy()

        observed_landmarks = np.zeros(N, dtype=bool)

        sensor_data_count = int(len(sensor_data_noised) / 2)
        frames = []

        mu_arr_gt = np.array([[0, 0, 0]])

        for idx in range(sensor_data_count):
            mu_prev = mu_arr[-1]

            u = sensor_data_noised[(idx, "odometry")]
            # predict
            mu_pred, sigma_pred = self.predict(mu_prev, sigma_prev, u, N)
            # update (correct)
            mu, sigma, observed_landmarks = self.update(mu_pred, sigma_pre
d, sensor_data_noised[(idx, "sensor")], observed_landmarks, N)

            mu_arr = np.vstack((mu_arr, mu))
            sigma_prev = sigma.copy()
            sigma_x_y_t_px1_py1_px2_py2 = np.vstack((sigma_x_y_t_px1_py1_p
x2_py2, sigma_prev[Index,Index].copy()))

            delta_r1_gt = sensor_data_gt[(idx, "odometry")]["r1"]
            delta_r2_gt = sensor_data_gt[(idx, "odometry")]["r2"]
            delta_trans_gt = sensor_data_gt[(idx, "odometry")]["t"]

            calc_x = lambda theta_p: delta_trans_gt * np.cos(theta_p + del
ta_r1_gt)
            calc_y = lambda theta_p: delta_trans_gt * np.sin(theta_p + del
ta_r1_gt)

            theta = delta_r1_gt + delta_r2_gt
```

```
            theta_prev = mu_arr_gt[-1,2]
            mu_arr_gt = np.vstack((mu_arr_gt, mu_arr_gt[-
1] + np.array([calc_x(theta_prev), calc_y(theta_prev), theta])))

            frame = plot_state(ax, mu_arr_gt, mu_arr, sigma, landmarks, ob
served_landmarks, sensor_data_noised[(idx, "sensor")])

            frames.append(frame)

        return frames, mu_arr, mu_arr_gt, sigma_x_y_t_px1_py1_px2_py2


if __name__ == "__main__":
    basedir = '/content/drive/MyDrive/mapping_and_perception/project_2/kit
ty_data'#example
    date = '2011_09_26' #example (fill your correct data)#2011_09_26_drive
_0022
    drive = '0022' #The recording number I used in the sample in class  (f
ill your correct data)
    dat_dir = os.path.join(basedir,"Ex3_data")

    dataset = DataLoader(basedir, date, drive, dat_dir)

    project = ProjectQuestions(dataset)
    project.Q1(basedir)
    project.Q2(basedir)
    project.Q3(basedir)

    #project.run()
```

Appendix

The code:

Q1 -> KalmanFilter.performe_KalmanFilter -> iterates over for each time step: one_step_of_KalmanFilter (computes dead reckoning if required)

The code:

Q2 -> EXtendedKalmanFilter.performe_KalmanFilter -> iterates over for each time step: one_step_of_KalmanFilter with EKF = True (computes dead reckoning if required)

Q3 ->EKFslamrun sets up and iterates over all time steps perfeoms -> predict +update