Deadline: *Wednesday 29. October 23:59* on Blackboard.

Groups size: This assignment can be solved in groups of two or three. You are allowed to complete the assignment on your own if you want, but we advice you to solve it in groups as we believe that group discussions are important. We have set up an own forum on Blackboard that can be used to find groups. Via Blackboard you should also be able to see if there are groups with open spots that you could contact. It is okay to discuss the assignment with other groups, but each group should write an individual report.

Group signup: You find the group signup under "Groups" in the menu on the left in Blackboard. You and your group should sign up in the same group.

Hand in: Hand in the code as **the zip created by running *create_handin.py*** under "Group Assignment 1 code". Do not use any libraries that are not part of the TTK4250 environment. Hand in the report as a single pdf under "Group Assignment 1 report". Only one student per group should hand in both the report and the code.

Code content:

The score on the code will be based on the number of tests passed, just as for the individual assignments. If you notice the tests having false negatives or positives, or behaving weirdly, please let us know.

Report content:

For the report you are limited to 1000 words and a total of 6 pages (1 frontpage, 6 pages of content, 1 page of bibliography). Figure captions do not count toward the limit, as long as they are concise and purely descriptive. The group number and group members should be clearly stated.

In the report we simply require you to clearly answer Task 5 and 6. As the report is short you can skip the introduction and conclusion. You can assume the person reading the report has good knowledge of the subject. You can simply refer to equations and algorithms in the book when what you want to say is already stated there. Ex: "Given what's stated in (12.34) and equation (56.78) in the book the oscillations in Figure 9 are expected" (When refering to equations, figures etc. you should make it clear if you are refering to equations, figures etc. in the book or report).

The goal of the report is to show that you understand the code and the theory behind it by presenting and discussing the plots and results. Remember to argue your points and explain your reasoning. Points are not given if what you state is right but you don't explain why. Ex: If you write "These parameters gave bad results", you should also include something like "as can be seen by the high RMSE values in Figure 9".

It is not expected that you present all the results, plots, and tuning variables from the code, but rather that you present and discuss the ones you find most relevant for each part of the assignment. The report should, however, include all four main types of plots (state, NIS, NEES, error).

If you have any questions please make a post on the forum and we will try to answer them as soon as possible or contact the student assistants during the excise class.

Some information about the code

- The files you will modify are `eskf.py`, `models.py`, `sensors.py`, `config.py`, `tuning_real.py`, and `tuning_sim.py`.
- `states.py` contains the different state types used. It is recommended to read through them.
- The `NamedArray` class is a wrapper around `np.ndarray` that let instances behave both as a `ndarray` and a `dataclass`. The `vel: AtIndex[3:6] | WithXYZ` syntax means that `nom.vel.y` is equivalent to `nom[3:6][1]`, which is mainly used to facilitate plotting.
- `plotting.py` contains all the plotting code necessary to get a full score on the assignment, but you are of course also free to create other plots.
- `config.py` contain some overall configuration parameters. `RUN` decides which data to use (`'simulated'` or `'real'`). `DEBUG` can be set to `False` to increase speed by disabling sanity checks. `PLOT_MIN_DT` defines the minimum difference in time between two elements in the plots. This value might be necessary to increase if you are running on the full dataset with all IMU measurements.

- `tuning_real.py` and `tuning_sim.py` contain all tunable parameters. The `[gnss|imu]_min_dt_[real|sim]` parameters are used to downsample the data. Ex: `imu_min_td_real=0.05` means that there will be at least 0.05 seconds between each imu measurement when running on real data, which is less precise but a lot faster.

Some more background on IMU measurements

An IMU is never perfect. Even if it is well calibrated there is always some residual error. The main errors are scale and orthogonality errors in addition to the measurement noise. In addition, we are never able to mount the IMU perfectly aligned with the body coordinates. This latter problem could be neglected if we simply define the body coordinate axes to be aligned with the IMU axes, but this will then later give a problem if we want to use the estimates to control heading, for example.

The mounting error can be specified by a rotation matrix R_M. If we denote the body coordinate frame by superscript b and IMU coordinate frame by superscript imu we get that the acceleration in the different coordinate systems are related through $a^{\text{imu}} = R_M a^b$ and the rotation rate likewise $\omega^{\text{imu}} = R_M \omega^b$. The orthogonality error can be described by a 3×3 matrix, with rows corresponding to the direction of the axes it really measures. Calling this matrix O_a for the accelerometers and O_g for the gyros we have that the measured direction of the accelerations are given by $\tilde{a}^{\text{imu}} = O_a R_M a^b$ and $\tilde{\omega}^{\text{imu}} = O_g R_M \omega^b$ for the rotation rate. Notice that the rows of $O_a R_M$ and $O_g R_M$ specify the direction of measurement in body coordinates. The last error we accommodate is the scale which can be represented by a diagonal matrix with positive entries. Letting D_a and D_g denote this matrix for acceleration and rotation rate, respectively, we get the relations

$$a_m = D_a O_a R_M a_t^b + a_{bt} + a_n \tag{1}$$
$$a^b = R_M^T O_a^{-1} D_a^{-1} (a_m - a_{bt} - a_n) = S_a (a_m - a_{bt} - a_n) \tag{2}$$
$$\omega_m = D_g O_g R_M \omega_t^b + \omega_{bt} + \omega_n \tag{3}$$
$$\omega^b = R_M^T O_\omega^{-1} D_\omega^{-1} (\omega_m - \omega_{bt} - \omega_n) = S_g (\omega_m - \omega_{bt} - \omega_n), \tag{4}$$

where we have also included the biases and measurement noises. Since we do not care about the intermediate values, we let the matrices $S_a = R_M^T O_a^{-1} D_a^{-1}$ and $S_g = R_M^T O_\omega^{-1} D_\omega^{-1}$ specify all the systematic linear IMU measurement errors. In the following situation, you can assume these matrices to be known, and they will be given to you in the data sets. In code, this is compensated for by preprocessing a_m, a_b, ω_m and ω_b by multiplying with these matrices, and by multiplying the entries of the error state system matrices from the right. This will be taken care of for you in the code that is handed out.

Note that an IMU is causal and that it measures the acceleration over the last time step in some manner, and not the acceleration/rotation into the future. This means that it makes sense to use the measurements at a time step k to predict the state from $k-1$ to k and not from k to $k+1$.

Some background on GNSS measurements

Global navigation satellite systems include GPS, GLONASS, Galileo and BeiDou. These systems consist of several satellites in a specific constellation that sends out a signal. A receiver can pick up these signals and can calculate what is known as pseudo-ranges for each received signal. From these pseudo-ranges and knowing the satellite position one can estimate the position and time of the receiver. The uncertainty of this estimate is highly dependent on how many satellites that are received and their geometric configuration. As such it is also calculated an uncertainty estimate for each measurement is based on the geometry and pseudo-range. In this assignment, however, the uncertainty is set to a constant value for all measurements.

Task 1: *Quaternion functions*

Finish the `RotationQuaternion` class in In `quaternion.py`.

(a) Finish the `quaternion.RotationQuaterion.multiply` method.

(b) Finish the `quaternion.RotationQuaterion.conjugate` method.

Task 2: *ESKF implementation, IMU part*

Finish the `ModelIMU` class in In `models.py`.

(a) Finish the `models.ModelIMU.correct_z_imu` method to correct for the IMU measurements.

(b) Finish the `models.ModelIMU.predict_nom` method to do a prediction step.

Assume that $a \approx R(q)(a_m - a_b) + g$ and $\omega \approx \omega_m - \omega_b$ is constant over the sampling time period.

Hint: The assumptions give that $v(k+1) = v(k) + T_s a$, $p(k+1) = p(k) + T_s v(k) + \frac{T_s^2}{2} a$ and local rotation vector increment $\kappa = T_s \omega$ in body frame. The local rotation vector increment k in body coordinates gives the predicted quaternion $q(k+1) = q(k) \otimes e^{\frac{\kappa}{2}} = q(k) \otimes \begin{bmatrix} \cos\left(\frac{||\kappa||_2}{2}\right) & \sin\left(\frac{||\kappa||_2}{2}\right) \frac{\kappa^T}{||\kappa||_2} \end{bmatrix}^T$. You can also use `RotationQuaterion.from_avec` to create the quaternion. The biases follow equation (10.50).

(c) Finish the `models.ModelIMU.A_c` method to get the continuous transition matrix.

(d) Finish the `models.ModelIMU.get_error_G_c` method to get the **G** matrix in (10.68).

(e) Finish the `models.ModelIMU.get_discrete_error_diff` method to discretize the error state matrices.

Hint: Van Loan gives that

$$\exp\left(\begin{bmatrix} -A & GQG^T \\ 0 & A^T \end{bmatrix} T_s\right) = \begin{bmatrix} \exp(-AT_s) & \exp(-AT_s)Q_d \\ 0 & \exp(A^T T_s) \end{bmatrix} = \begin{bmatrix} \exp(-AT_s) & \exp(-AT_s)Q_d \\ 0 & \exp\left(AT_s\right)^T \end{bmatrix},$$

and therefore gives both the matrices you need in the latter columns.

(f) Finish the `models.ModelIMU.predict_err` method to predict the error state.

Task 3: *ESKF implementation, GNSS part*

Finish the `SensorGNSS` class in In `sensors.py`.

(a) Finish the `sensors.SensorGNSS.H` method.

Hint: As the GNSS is mounted outside the origin of the body frame, the measured position depends on the orientation of the drone.

(b) Finish the `sensors.SensorGNSS.pred_from_est` used to get the measurement prediction.

Task 4: *ESKF implementation, Filter part*

Finish the `ESKF` class in In `eskf.py`.

(a) Finish the `eskf.ESKF.predict_from_imu` method.

(b) Finish the `eskf.ESKF.update_err_from_gnss` method.

(c) Finish the `eskf.ESKF.inject` method.

(d) Finally, finish the `eskf.ESKF.update_from_gnss` method.

Task 5: *Run ESKF on simulated data*

Set the `RUN='sim'` in the `config.py` file to use simulated data.

A fixed wing UAV has been simulated with 100 Hz IMU measurements, and 1 Hz GNSS measurements for 900 seconds.

(a) Keep the initial parameters. Looking at the orientation error plot you should see that the estimated error does not converge to a constant value, even though the covariance of the IMU and the GNSS is constant. Try to explain the reason behind this.

(b) You should see that both the estimated and true orientation error is greatest in the yaw direction. Why is this? Could the estimated and or true error diverge indefinitely? In that case, under what conditions will this happen?

(c) Try to round off the `accm_correction` and `gyrom_correction` to the nearest integer using `np.round`, neglecting small variations in mounting errors, scale errors and orthogonality errors. What does it do to your estimates? Why do you believe it does so?

(d) Find three different sets of changes to the tuning parameters that you find interesting and present them. You are free to choose whatever parameters you want to change, including the drone parameters and `[gnss|imu]_min_dt_sim`. One way to choose parameters is to try to exaggerate or minimize some effects observed with the default parameters. Use this as an opportunity to show your knowledge of the system.

Task 6: *Run ESKF on real data*

Set the `RUN='real'` in the `config.py` file to use real data.

You are given a data set from a real fixed-wing UAV flight where a STIM300 IMU gives 250 Hz measurements, and two Ublox-8 GNSS receivers gives 1 Hz measurements. The UAV can be seen in Figure 1, where the two black bits sticking up are the two GNSS antennas that are on board. You are only given the position data from the front one. A plot of the GNSS trajectory can be seen in Figure 2.

(a) In real life, GNSS measurements can arrive between two IMU measurements. Can you suggest a way to handle this when using an ESKF?

(b) Similarly to what you did for the simulated data; round off the `accm_correction` and `gyrom_correction` to the nearest integer using `np.round`.

How does this change your estimates? If you did not know, could you tell that you are now using the "wrong" IMU measurements? What does this tell you about setting up an ESKF to be used in a real-world application?

(c) Again, find three different sets of changes to the tuning parameters that you find interesting and present them. At least two of the sets should showcase different challenges of using an ESKF in a real-world application without

Figure 1: The UAV used in the experiment

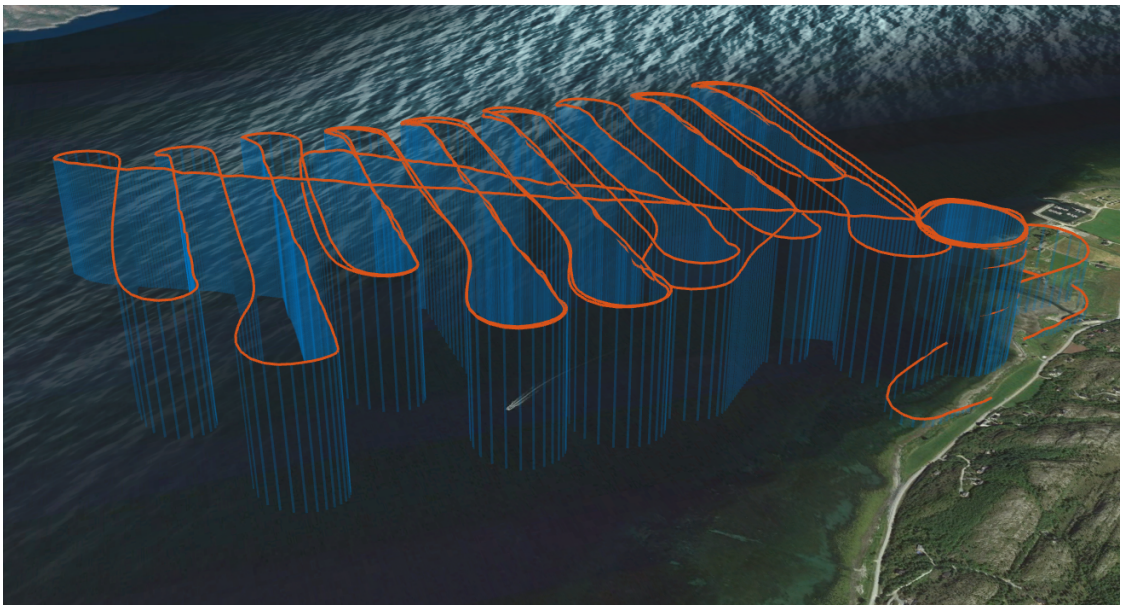

Figure 2: The path taken by the UAV

access to ground truth. Use this as an opportunity to show your knowledge of the system.