TTK4250 Sensor Fusion

Graded Assignment 2

Code and results hand in (5%): *Wednesday 28. October 16:00* on Blackboard as a single zip file.
Report hand in (10%): *Wednesday 30. October 16:00* on Blackboard as a single PDF file.

Pairs: This assignment is to be solved in pairs. These pairs will be the same for all the graded assignments.

Code hand in: Functioning python code that produces your results for task 3. This will account for one third of the possible points for this assignment (5% of total grade).

Report hand in: A 4 page (in total!) report is to be handed in by 16.October 16.00 as a PDF on Blackboard. Points will be withdrawn for reports longer than 4 pages. As the report is quite short, you should *consider carefully* what to have in it.

Report content: The algorithms are given in the book, so you can simply refer to the book when what you want to say is already stated there. For task 3, we want to see

- plot of the different states over time
- plot of the different true error states over time,
- plot of the NIS along with your chosen confidence bound over time,
- plot of the total, position, velocity, attitude and the two biases' NEES along with your chosen confidence bounds over time for these,
- the averaged NIS along with your confidence bound,
- the numbers of the averaged NEESes along with your confidence bounds,
- the number of times the NEESes fall within your confidence regions
- and the different RMSEs (mean taken over time)

for a selected parameter set. For task 4 this list reduces to

- plot of the NIS along with your chosen confidence bound over time,
- the averaged NIS along with the confidence bound,

due to the lack of ground truth.

We then expect you to discuss what these number and parameters means for the tracker in practice, and what is gained or lost by changing the parameters and why. Points can also be given for other plots, numbers and considerations, but what to show is *up to you* to decide. Answers and analysis should connect theory and results to the real world, and show your understanding for the problem and solution. Try to connect the results on the simulated data to the results of the real data where it is possible.

To help tuning you can for instance split the NEES/NIS into the different axes and/or look at the biases, size and whiteness of the error or innovation sequence.

Note:

Again, this is not supposed to be a "try one million parameter sets" assignment. You are supposed to show your understanding of the algorithms in theory and practice. Yes, you do have to test a fair share of parameters, but doing just that will not give you the best grade. This is supposed to be an engineering/scientific assignment, where in some sense you as a consultant have the task of showing a company what they can get from these algorithms on their datasets. That is, making sure that the algorithm is correct (you can get help with that), making hypotheses to test, selecting which parameters to try in a reasonable fashion, and documenting your findings in terms of consistency, the stated metrics and the estimated trajectory, is what will give you a good result. Any theoretical or practical insight from the lectures or book elaborated upon or applied to the data sets also counts positive.

Some more background on IMU measurements

An IMU is never perfect. Even if it is well calibrated there is always some residual errors. 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 $O_g R_M$ specifies 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 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.

Some background on GNSS measurements

Global navigation satellite systems include GPS, GLONASS, Galileo and BeiDou. These consists 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 is received and their geometric configuration. As such it is also calculated an uncertainty estimate for each measurement based on the geometry and pseudo-ranges.

In the given real dataset this is represented by a single number and you can use this in the measurement covariance matrix if you want.

Some background on template code

A skeleton and template code is handed out to give you a starting point to write your code. You are free to rewrite the template, or start a new if you wish. But this will give you a good idea of where to start, some checks to help avoid some possible mistakes in your functions, some documentation, and a starting point for the plots you need for the report. The files handed out are:

- `utils.py`: The skew-symmetric matrix function, and any other utility function you may need.
- `quaternion.py`: All quaternion functions you need as skeletons.
- `mytypes.py`: Some predefined types for annotation of variables and parameters.
- `eskf.py`: The ESKF-class skeleton, and related methods.
- `run\_INS\_simulated.py`: Template for running and analyzing the ESKF methods on simulated data.
- `run\_INS\_real.py`: Template for running and analyzing the ESKF methods on real data.
- `cat\_slice.py`: The CatSlice-class to (hopefully) simplify concatenating index slices.

The CatSlice-class

Although the CatSlice-class is another layer of abstraction, it can make the code simpler and more readable. NumPy can be difficult in accessing sub-matrices using lists or arrays, and there seem to be no obvious nice way to concatenate slices. Internally `CatSlice(start=0, stop=3)` is simply stored as `np.array([0, 1, 2])`, but the class defines some syntactic sugar.

The `__add__`-method is overwritten, so that `CatSlice(start=0, stop=3) + CatSlice(start=6, stop=10)` becomes `np.array([0, 1, 2, 6, 7, 8, 9])` internally. This allows us to access and change different parts of a state vector at the same time, which can save some lines of code, and increase readability.

The `__mul__`- and `__pow__`-methods are also overwritten, to simplify multidimensional indexing. As an example, if we have a covariance matrix `P: np.ndarray`, and need the sub-matrix with rows 0 to 3 and columns 6 to 10, we get the correct indices through `P[CatSlice(start=0, stop=3) * CatSlice(start=6, stop=10)]` (uses `np.ix_` internally). For simplicity, if we wish to slice the same index twice, we can for instance write `P[CatSlice(start=0, stop=3)**2]`. Note that as it is implemented now it does not work with the normal multidimensional indexing in NumPy, so we have to write `P[0][CatSlice(start=0, stop=3) * CatSlice(start=6, stop=10)]` or `P[(0,) + CatSlice(start=0, stop=3) * CatSlice(start=6, stop=10)]` if we wish to index an array of shape `(S, M, N)`.

Finally, several useful slices are predefined in `eskf.py` and imported into `run\_INS\_simulated.py` and `run\_INS\_real.py`. These are defined according to the equations (10.52) in the book. You should see the definitions yourself, but the indexing only differs from nominal state to the error state because the quaternion has one more state than rotation vector. The indices with the `ERR_`-prefix are defined according to the error state, or using Euler angles, while the other are either common or for the nominal states.

Task 1: *Quaternion functions*

(a) Implement the function

```
def cross_product_matrix(vector: ArrayLike) -> np.ndarray:
```

in `utils.py`. It should implement equation (10.5). That is, it takes a vector $n \in \mathbb{R}^3$ and creates the skew symmetric matrix that corresponds to the cross product as matrix multiplication.

(b) Implement the quaternion product in the function

```
def quaternion_product(
        ql: np.ndarray,
        qr: np.ndarray
) -> np.ndarray:
```

Let the function operate on both 3 and 4 dimensional vectors representing a pure and general quaternion, respectively. Either equation (10.21) or (10.34) gives a formula you can use.

(c) Implement the function

```
def quaternion_to_rotation_matrix(
        quaternion: np.ndarray,
        debug: bool=True
) -> np.ndarray:
```

that returns a rotation matrix corresponding to a given quaternion.

(d) Implement the function

```
def quaternion_to_euler(quaterion: np.ndarray) -> np.ndarray:
```

that returns the Euler angles corresponding to a given quaternion.

Task 2: *ESKF implementation*

This task is worth up to 3% for code hand in.

(a) Implement the discrete time prediction of equation (10.58) in the function

```
def predict_nominal(
        self,
        x_nominal: np.ndarray,
        acceleration: np.ndarray,
        omega: np.ndarray,
        Ts: float,
) -> np.ndarray:
```

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, and assume that `acceleration` and `omega` are debiased. Remember to keep the predicted quaternion normalized.

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 $q(k+1) = q(k) \otimes e^{\frac{\kappa}{2}} = q(k) \otimes \left[\cos\left(\frac{||\kappa||_2}{2}\right) \quad \sin\left(\frac{||\kappa||_2}{2}\right) \frac{\kappa^T}{||\kappa||_2} \right]^T$.

(b) Implement the error state system matrix $A(x)$ in equation (10.68) in the function

```
def Aerr(
        self,
        x_nominal: np.ndarray,
        acceleration: np.ndarray,
```

```
        omega: np.ndarray,
) -> np.ndarray:
```

Assume that acc and omega are debiased. The matrices S_a and S_g are handled for you already at the end of the function.

(c) Implement the noise input matrix $G(x)$ in equation (10.68) in the function

```
def Gerr(
        self,
        x_nominal: np.ndarray,
) -> np.ndarray:
```

Hint: The functions `np.vstack` and `la.block_diag` might simplify things.

(d) Discretize the error state matrices using Van Loan's method in the function

```
def discrete_error_matrices(
        self,
        x_nominal: np.ndarray,
        acceleration: np.ndarray,
        omega: np.ndarray,
        Ts: float,
) -> Tuple[np.ndarray, np.ndarray]:
```

Assume that acceleration and omega are debiased.

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.

Hint: If you profile your code you might find that `la.expm` makes your code slow, and that you want to use a simpler Taylor or Padé approximation.

(e) Implement the function that predicts the error state covariance

```
def predict_covariance(
        self,
        x_nominal: np.ndarray,
        P: np.ndarray,
        acceleration: np.ndarray,
        omega: np.ndarray,
        Ts: float,
) -> np.ndarray:
```

Assume that acceleration and omega are debiased.

Hint: This now amounts to a discrete time KF covariance prediction using the previous functions to get the discrete system matrices A_d and Q_d.

(f) Combine the above into a single function that predicts both the nominal state and the error state covariance in the function

```
def predict(
        self,
        x_nominal: np.ndarray,
        P: np.ndarray,
        z_acc: np.ndarray,
```

```
        z_gyro: np.ndarray,
        Ts: float,
) -> Tuple[
        np.array, np.array
]:
```

Hint: zAcc and zGyro are the real measurements and should be corrected and debiased before further use. The correction using S_a and S_g is already handled for you, and debiasing is done through $\omega = \omega_m - \omega_b$, where ω_b is the estimated bias.

(g) Implement the error state injection in the function

```
def inject(
        self,
        x_nominal: np.ndarray,
        delta_x: np.ndarray,
        P: np.ndarray,
) -> Tuple[
        np.ndarray, np.ndarray
]:
```

This should implement the equations (10.85) and (10.86). Again, remember to keep the quaternion normalized.

(h) Implement the function that give the innovation and innovation covariance for a GNSS position measurement

```
def innovation_GNSS_position(
        self,
        x_nominal: np.ndarray,
        P: np.ndarray,
        z_GNSS_position: np.ndarray,
        R_GNSS: np.ndarray,
        lever_arm: np.ndarray = np.zeros(3),
) -> Tuple[
        np.ndarray, np.ndarray
]:
```

The compensation of a potential lever arm is implemented for you.

(i) Implement the GPS update function in

```
def update_GNSS_position(
        self,
        x_nominal: np.ndarray,
        P: np.ndarray,
        z_GNSS_position: np.ndarray,
        R_GNSS: np.ndarray,
        lever_arm: np.ndarray = np.zeros(3),
) -> Tuple[
        np.ndarray, np.ndarray
]:
```

Note that the output should be the values after injection. The compensation of a potential lever arm is implemented for you.

(j) Implement the function that calculates the NIS of a GNSS position measurement

```
def NIS_GNSS_position(
        self,
        x_nominal: np.ndarray,
        P: np.ndarray,
        z_GNSS_position: np.ndarray,
        R_GNSS: np.ndarray,
        lever_arm: np.ndarray = np.zeros(3),
) -> float:
```

(k) Implement the function that calculates the true error state

```
def delta_x(
        cls,
        x_nominal: np.ndarray,
        x_true: np.ndarray,
) -> np.ndarray:
```

Hint: $q_t = q \otimes \delta q \Rightarrow \delta q = q^* \otimes q_t$ and $\mathrm{Im}(\delta q) \approx \frac{1}{2}\delta\theta$.

(l) Implement the function that give out the NEES of the total state and the NEES of the "individual" substates. You might also want to expand this to include total NEES for position, velocity and attitude as well.

```
def NEESes(
        cls,
        x_nominal: np.ndarray,
        P: np.ndarray,
        x_true: np.ndarray,
) -> np.ndarray:
```

Task 3: *Run ESKF on simulated data x*

This task is worth up to 2% for code hand in and 4% for report hand in.

A fixed wing UAV has been simulated with 100Hz IMU measurements, and 1Hz GNSS measurements for 900 seconds. You are given a data set consisting of

- zAcc (3×90000): the accelerometer measurements
- zGyro (3×90000): the gyro measurements
- timeIMU (1×90000): the timing of the IMU measurements starting at 0.
- S_a (3×3): the accelerometer correction matrix
- S_g (3×3): the gyro correction matrix
- zGNSS (3×900): the GNSS position measurements
- timeGNSS (1×900): the timing of the GNSS measurements with the same clock as the IMU measurements

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 $k+1$.

A Python script has been handed out to you to get you started on this task. Note that this script is only there for your convenience and you can make your own from scratch if you like.

(a) Tune the parameters of the filter. These includes the IMU measurement noises, the IMU bias time constants (or maybe rather the reciprocal of the time constant) and its driving disturbance and the

Figure 1: The UAV used in the experiment

GNSS position noise. You can do this by looking at both NIS for the GNSS measurements and NEES, as well as the errors.

Hint: The GNSS has been simulated with a constant noise covariance. The planar uncertainties can safely be taken to be the same, while the altitude uncertainty tends to be higher than the planar for GNSS. We have no indication that anything other than diagonal measurement noise covariance should be used. The bias time constants are probably better specified in hours rather than in seconds (ie. they should be relatively large, perhaps even infinite giving a random walk).

(b) Take a look at the attitude estimation error. If you have done it right, it should be quite small. Roll and pitch can be estimated since gravity acts on the accelerometer, so that is fine. But what is it that makes the heading observable for us? Would you trust this system to give the correct heading if it was standing still as well?

Hint: Think of what happens when a GNSS measurement arrives if you have integrated up the IMU measurements in the wrong direction.

(c) Try to set the IMU misalignment matrices (`eskf.S_a` and `eskf.S_g` for an ESKF object `eskf`) to the identity matrix, effectively neglecting any mounting errors, scale errors and orthogonality errors. What does it do to your estimates, and why do you believe it does so? Could you disregard this matrix in real life?

Task 4: *Run ESKF on real data*

This task is worth up to 6% for report hand in.

You are given a data set from a real fixed wing UAV flight where a STIM300 IMU gives 250Hz 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. The dataset you are given consists of

- zAcc ($3 \times K$): The accelerometer measurements

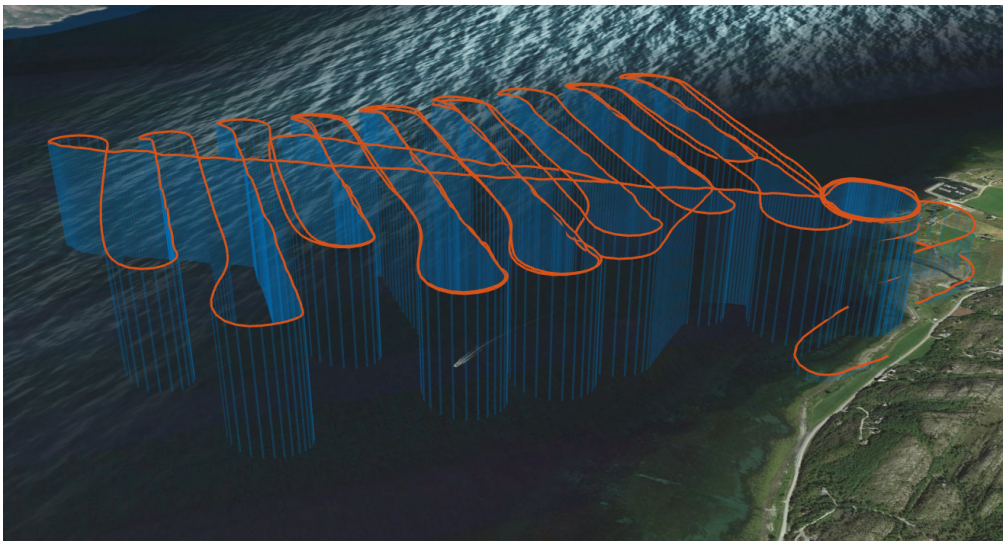

Figure 2: The path taken by the UAV

- zGyro ($3 \times K$): The gyro measurements
- timeIMU ($1 \times K$): The timing of the IMU measurements in seconds into the GPS week.
- S_a (3×3): Acceleration correction matrix.
- S_g (3×3): Gyro correction matrix.
- zGNSS ($3 \times KGNSS$): GNSS position measurements
- timeGNSS ($1 \times KGNSS$): The timing of the GNSS measurements in the same "time variable" as the IMU measurements.
- GNSSaccuracy ($1 \times KGNSS$): The position accuracy claimed by the receiver in meters.
- leverarm (3×1): The lever arm in meters. Specified as the vector from the IMU center to the GNSS antenna center.

(a) Tune the parameters of the filter. It is the same parameters to tune as for the last task. You do not have the ground truth anymore, but you can still use NIS.

The data sheet for the STIM300 (available online) can give you some hints on where to start tuning. Also, a good starting point would be the same values as for the last task. If discrete noises were specified they will need to be changed to the new sampling time.

You can use the GNSSaccuracy in some manner (eg. scale a matrix) to specify the position covariance or use a constant one. This is up to you.

Hint: Dividing the NIS into planar and altitude can possibly be helpful.

(b) Try to set the IMU misalignment to the given simplified version (simply rounded) only compensating for the known mounting orientation, effectively neglecting any mounting errors, scale errors and orthogonality errors. 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 a ESKF to be used in a real world application?