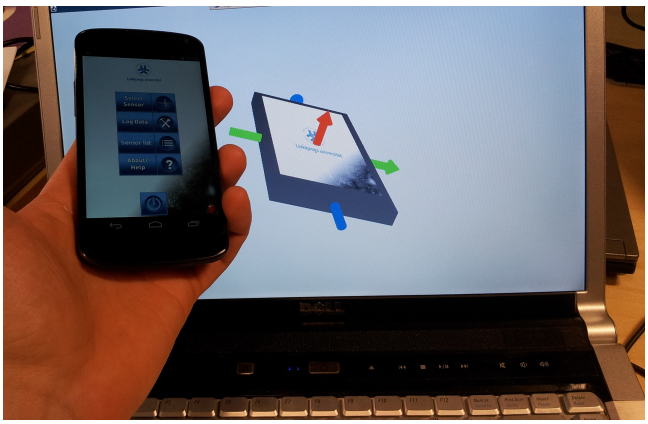
Project 1: Orientation estimation using smartphone sensors



SSY345 – Sensor fusion and nonlinear filtering

Lars Hammarstrand
Dept. of Signals and Systems
Chalmers University of Technology
March 12, 2020

This project was originally developed by Gustaf Hendeby and Fredrik Gustafsson at Linköping University for the course, TSRT14 Sensor Fusion. The instructions found here have been edited to fit the course SSY345 at Chalmers University of Technology.

1 Introduction

Navigation is one of the first and one of the most important applications of sensor fusion. At the same time, it is quite challenging both from a complexity and a numerical point of view. The core in most navigation systems is an orientation filter that integrates inertial information from gyroscopes and accelerometers with magnetometer measurements and other supporting sensors that relate to the orientation of the platform with respect to the world.

The goal of this project is to develop and implement an Extended Kalman filter (EKF) that performs orientation estimation, in real time, based on measurements provided by (streamed from) a standard smartphone. The project has several important **learning objectives**: after completing the project students should be able to

- understand the properties of the measurements obtained from key sensors on a smartphone (gyroscopes, accelerometers and magnetometers),
- develop measurement models by examining properties of real data (at least for examples that resemble the current one),
- design outlier detectors for different types of measurements and describe the benefits with such detectors,
- explain how different sensor measurements complement each other, in terms of the type of orientation information that they provide.

Instructions: The project should be performed in groups of 2 people (or less). Your solutions and results need to be properly documented, but it is sufficient to submit one report per group. Further information about the submission deadline is given on the course homepage. Note that every group needs to have access to an Android smartphone, so please take this into account when forming the groups. Also, though this is true for most modern phones, please download the application and verify that the phone has a gyroscope, accelerometer and magnetometer (using the *Select Sensor* window).

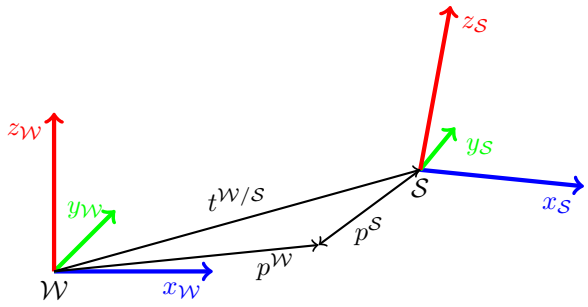


Figure 1: Illustration of the two inertial frames used here; the world fixed frame, $\mathcal{W}$, and the sensor (phone) fixed frame, $\mathcal{S}$.

2 Summary of Relevant Theory

This section provides relevant background theory on the state representation that we use. Above all, we motivate why we use quaternions and present important results needed in order to develop our EKF. At the end of this section, we discuss which of the measurements that we may view as inputs to the filter. More details can be found, e.g., in [1].

2.1 Representations of rotations

In this project we consider two inertial frames:

- The *world frame*, $\mathcal{W}$, which is fixed to earth, is aligned with its x-axis pointing east, its y-axis pointing north, and its z-axis pointing up. This way an orthogonal right-hand system is formed, sometimes denoted an ENU-system (east-north-up-system).
- The *sensor frame*, $\mathcal{S}$, is centered in the smartphone and moves with it. Ideally it should be centered in the accelerometer to avoid measuring accelerations due to pure rotations, but as the exact location of it is unknown, the frame is considered positioned in the center of the phone. Laying with the screen face up the x-axis points right, the y-axis points forward, and the z-axis points up. This is the same coordinate system used in both the Android and the iOS APIs for writing apps.

Points in the world and sensor systems are related via a linear transformation, illustrated in Figure 1, which can be described mathematically as

$$p^{\mathcal{W}} = R^{\mathcal{W}/\mathcal{S}} p^{\mathcal{S}} + t^{\mathcal{W}/\mathcal{S}}, \tag{1}$$

where $p^{\mathcal{S}}$ is a point expressed in the sensor frame and $p^{\mathcal{W}}$ is the same point expressed in the world frame. The relative rotation between the two frames is given by $R^{\mathcal{W}/\mathcal{S}}$, which aligns the world frame with the sensor frame. The translation between the centers of the coordinate frames is given by $t^{\mathcal{W}/\mathcal{S}}$; however,

as only the rotation is of interest in this project, $t^{\mathcal{W}/\mathcal{S}}$ will not be considered further.

The focus of this project is to estimate the rotation $R^{\mathcal{W}/\mathcal{S}}$ based on the sensor measurements available from a smartphone. The rotation describes how to rotate points in the sensor frame, $\mathcal{S}$, to describe them in the world frame, $\mathcal{W}$. It can also be interpretated as the rotation that is applied to the world frame to align it with the sensor system.

A rotation can be described in many ways. One way is to use a rotation matrix. This is a description most of us are familiar with and know well how to manipulate. However, it uses 9 values to represent 3 degrees of freedom. Hence, it is not a minimal representation, which makes estimating the rotation matrix difficult. Another alternative is to use Euler angles, i.e., a representation in terms of three consecutive rotations around predefined axes. Even though represented by three values, Euler angles are not unique — several combinations of rotations result in the same final rotation — the representation has discontinuities in the parameters, and suffers from the gimbal lock effect. These are all factors that make the Euler angles unsuitable for estimation purposes. A third alternative is to use a unit length quaternion representation. The quaternions suffer less from the problems of the rotation matrix and the Euler angles and is therefore a popular choice for orientation estimation.

2.1.1 The quaternion representation

A quaternion, q, can always be written in terms of an axis-angle representation

$$q = \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{pmatrix} = \begin{pmatrix} \cos(\frac{1}{2}\alpha) \\ \sin(\frac{1}{2}\alpha) \begin{pmatrix} \hat{v}_x \\ \hat{v}_y \\ \hat{v}_z \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \cos(\frac{1}{2}\alpha) \\ \sin(\frac{1}{2}\alpha)\hat{v} \end{pmatrix},$$

where α is an angle and $\hat{v}$ is a vector of unit length, i.e., $\|\hat{v}\| = 1$, which also implies that $\|q\| = 1$. If q describes a rotation of a coordinate system, then $\hat{v}$ is the axis around which the system is rotated and α describes the rotation angle. Note that the quaternion representation is not unique since q and $-q$ represent the same rotation. (To show this you can for instance evaluate q for $\alpha = \beta$ and $\alpha = \beta + 2\pi$, where β is some angle.) However, this is much less of a problem than in the case with Euler angles. In most cases, $q_0 \geq 0$ is assumed and enforced to get an unambiguous representation.

You main task in this project is to estimate the time-varying quaternion vector using an EKF. The following mathematical relations are useful when working with orientation estimation using the quaternion representation (especially when you wish to develop an EKF):

- The formula to convert a unit length quaternion to a rotation matrix,

$$R^{\mathcal{W}/\mathcal{S}} = Q(q) = \begin{pmatrix} 2q_0^2 - 1 + 2q_1^2 & 2q_1q_2 - 2q_0q_3 & 2q_1q_3 + 2q_0q_2 \\ 2q_1q_2 + 2q_0q_3 & 2q_0^2 - 1 + 2q_2^2 & 2q_2q_3 - 2q_0q_1 \\ 2q_1q_3 - 2q_0q_2 & 2q_2q_3 + 2q_0q_1 & 2q_0^2 - 1 + 2q_3^2 \end{pmatrix}. \quad \text{(2a)}$$

This is implemented in `Qq(q)` in Appendix A.2.3. Note how all occurrences of q_i are in pairs, hence, changing the sign on all quaternion components does not change the resulting rotation matrix $Q(q) = Q(-q)$, as noted previously. As a simple exercise, please verify that $\alpha = 0$ yields the rotation matrix that you would expect.

The reverse operation is given by

$$q = Q^{-1}(R) = \begin{pmatrix} q_0 \\ (R_{23} - R_{32})/(4q_0) \\ (R_{31} - R_{13})/(4q_0) \\ (R_{12} - R_{21})/(4q_0) \end{pmatrix}, \tag{2b}$$

where $q_0 = \frac{1}{2}\sqrt{R_{11} + R_{22} + R_{33} + 1}$. A certain degree of caution must be applied to ensure numerical stability performing this conversion.

- The angle, Δ, between two unit length quaternions q^0 and $\hat{q}$ can be calculated as

$$\Delta = 2 \arccos\left(\sum_{i=0}^{3} q_i^0 \hat{q}_i\right). \tag{2c}$$

This way Δ describes the difference between $\hat{q}$ and q^0 as the angle $\hat{q}$ must be turned to obtain q^0.

- The time derivative of a quaternion, in terms of angular velocities, ω, (where the quaternion represents $R^{\mathcal{W}/\mathcal{S}}$ and ω is given in the sensor frame) can be shown to be

$$\dot{q} = \frac{1}{2} S(\omega) q = \frac{1}{2} \begin{pmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{pmatrix} q \tag{2d}$$

$$= \frac{1}{2} \bar{S}(q) \omega = \frac{1}{2} \begin{pmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \end{pmatrix} \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix}. \tag{2e}$$

The functions $S(\omega)$ and $\bar{S}(q)$ are implemented in `Somega(omega)` and `Sq(q)`, in Appendices A.2.5 and A.2.4, respectively. It is sometimes also useful to note that $S(\omega_1 + \omega_2) = S(\omega_1) + S(\omega_2)$ and $\bar{S}(q_1 + q_2) = S(q_1) + S(q_2)$.

- Differentiating $Q(q)$ with respect to q is a bit more tricky as the result is a three-dimensional tensor. Differentiating with respect to one component

at the time yields:

$$\frac{dQ(q)}{dq_0} = 2\begin{pmatrix} 2q_0 & -q_3 & q_2 \\ q_3 & 2q_0 & -q_1 \\ -q_2 & q_1 & 2q_0 \end{pmatrix}, \tag{2f}$$

$$\frac{dQ(q)}{dq_1} = 2\begin{pmatrix} 2q_1 & q_2 & q_3 \\ q_2 & 0 & -q_0 \\ q_3 & q_0 & 0 \end{pmatrix}, \tag{2g}$$

$$\frac{dQ(q)}{dq_2} = 2\begin{pmatrix} 0 & q_1 & q_0 \\ q_1 & 2q_2 & q_3 \\ -q_0 & q_3 & 0 \end{pmatrix}, \tag{2h}$$

$$\frac{dQ(q)}{dq_3} = 2\begin{pmatrix} 0 & -q_0 & q_1 \\ q_0 & 0 & q_2 \\ q_1 & q_2 & 2q_3 \end{pmatrix}. \tag{2i}$$

The quaternion derivative is implemented in `dQqdq(q)`, in Appendix A.2.6.

2.2 Sensor Fusion

We assume that our state space model is

$$x_{k+1} = f(x_k, u_k, v_k), \tag{3a}$$

$$y_k = h(x_k, u_k, e_k), \tag{3b}$$

where x_k is the state at time k, u_k is the known (or measured) input to the system, v_k is the process noise, y_k is the measurement vector, and e_k is the measurement noise. The goal is to estimate the orientation, represented by the quaternion q_k, and we will let that be a part of our state vector, x_k. The sensors provide 3D measurements of:

- Angular rates ω_k, denoted y_k^ω when used as measurements, at sampling rate f_s^ω.
- Accelerations a_k, denoted y_k^a when used as measurements, at sampling rate f_s^a.
- Magnetic field m_k, denoted y_k^m when used as measurement, at sampling rate f_s^m.

When designing a filter to estimate the phone's orientation there are several things to take into consideration: Should some of the measurements be described as input signals? Do any of the measurements contain biases that we need to estimate? Are the biases time-varying? How can we handle asynchronous measurements? In what follows next we focus on the question regarding the input signals and we return to the other questions later.

2.2.1 Inputs or outputs (measurements)?

In certain situations it is desirable to view some measurements as input signals (in this context you may think of these as noise free measurements or known state variables). An important benefit is that it may enable us to remove the corresponding variables from the state vector, which can simplify the solution significantly[1].

There are many ways to divide the measured quantities into inputs u_k and outputs y_k in (3). For clarity, everything that is measured is referred to as measurements, and can be used as inputs or outputs depending on the chosen filter structure. A few main structures are as follows:

1. A simple model is to let

$$x_k = q_k, \qquad u_k = \omega_k, \qquad y_k = \begin{pmatrix} y_k^{a,T} & y_k^{m,T} \end{pmatrix}^T, \tag{4a}$$

   and not to attempt any bias estimation. This alternative is appealing, e.g., if none of the measurements are biased and the gyroscope measurements are accurate.

2. A considerably more complex model is to let

$$x_k = \begin{pmatrix} q_k^T & \omega_k^T & b_k^{\omega,T} & b_k^{a,T} \end{pmatrix}^T, \quad y_k = \begin{pmatrix} y_k^{\omega,T} & y_k^{a,T} & y_k^{m,T} \end{pmatrix}^T, \tag{4b}$$

   and estimate both the accelerometer bias, b^a, and the gyroscope bias, b^ω (but not the magnetometer bias).

   - Angular rates ω_k, denoted y_k^ω when used as measurements, at sampling rate f_s^ω.
   - Accelerations a_k, denoted y_k^a when used as measurements, at sampling rate f_s^a.
   - Magnetic field m_k, denoted y_k^m when used as measurement, at sampling rate f_s^m.

3. An intermediate model is

$$x_k = \begin{pmatrix} q_k^T & b_k^{\omega,T} & b_k^{a,T} \end{pmatrix}^T, \quad u_k = \omega_k, \quad y_k = \begin{pmatrix} y_k^{a,T} & y_k^{m,T} \end{pmatrix}^T, \tag{4c}$$

   where the gyroscope measurement is considered as input, but the biases are estimated.

4. Since the accelerometer measures both the specific force f_k^a, that accelerates the platform, and the gravity g^0 (see the section about the accelerometer below), another alternative is to include position and velocity in the state, and to let acceleration either be a state or an output. That is, to also estimate $t^{\mathcal{W}/\mathcal{S}}$. This is a much harder problem which needs some sort of absolute measurement to be observable, or the estimate will quickly start to drift.

In order to limit your alternatives, you are expected to use the gyroscope measurements as inputs in this project, i.e., $u_k = \omega_k$. One reason why this is a good choice is that gyroscope measurements provided by smartphones are normally rather accurate.

[1] It is an interesting philosophical question to discuss what the difference between an input and output really is from a filtering perspective.

- **Task 1:** Discuss pros and cons regarding this choice of input. Can you imagine a situation where this would not be a good choice? When would it be better to include angular velocities in the state vector? You do not need to provide a long discussion, but your statements/examples should be clearly explained.

3 The Sensor Fusion Android app

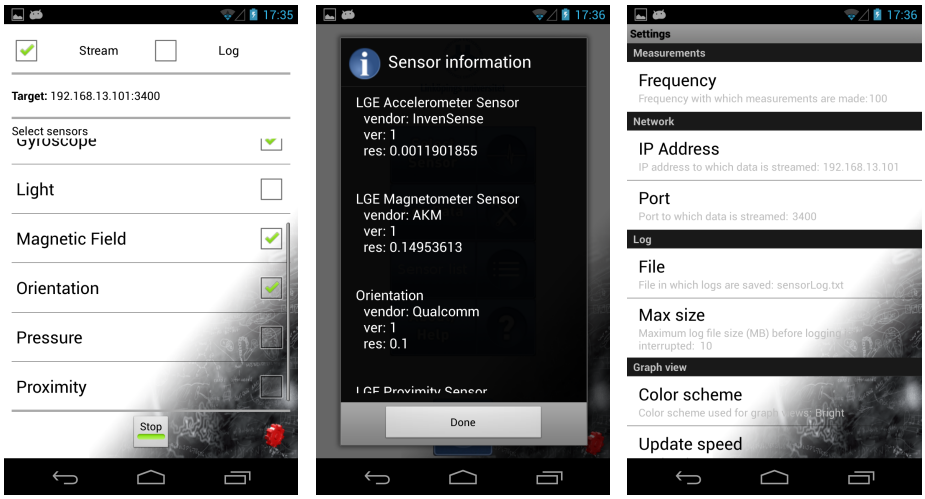
During the project you will work with data that you collect online with an Android smartphone (technically any Android device with suitable sensors that can be connected to Internet) using an app that streams the measurements from the phone to a PC, where it can be accessed in for example Matlab. The application is available for free from Google Play (`http://goo.gl/0qNyU`) under the name *Sensor Fusion*, see also [2].

First time launching the Sensor Fusion app you end up in its home screen (Figure 2(a)), from which all the app's basic functionality can be reached.

Select Sensor Clicking "Select Sensor" yields a new view similar to the one in Figure 2(b). Exactly what sensors that show up depends on what sensors are available in your phone, which varies between different brands and models. When selecting a sensor, sensor readings from it are visualized. Figure 2(c) shows the result from clicking the "Gyroscope" button. The gyroscope measurements are visualized as three curves in a graph, together with the norm of the measurements.

(a) Home screen. (b) Sensor selection screen. (c) Graph of gyroscope data.

Figure 2: Screen shots from the Sensor Fusion app.



(a) Stream view. (b) Sensor information. (c) Settings screen.

Figure 3: Screen shots from the Sensor Fusion app.

Log Data Clicking "Log Data" brings up a view (Figure 3(a)) from which it is possible to stream data over internet or save it to a file on the device to be read out and processed at a later time. What sensor data is collected is determined by checking the respective check box in the sensor list. The check boxes "Stream" and "Log" determine if the collected measurements should be streamed over internet and/or saved to a log file, respectively. The details about where to stream the data and the log file are displayed in the field below and can be changed by clicking the red gear at the bottom right of the screen. Data collection is started and stopped using the button at the bottom of the page. Note that before starting to stream data a receiver must be available, or the connection will be automatically shut down. More information regarding how to log the data in Matlab is given in Section 4.

Sensor list Clicking "Sensor list" brings up a dialog (Figure 3(b)) enumerating all sensory measurements that are available on request by an Android app. Note that some of the listed sensors are physical sensors, e.g., accelerometers and magnetometers; whereas others are virtual sensors (the measurements are obtained from fusing other sensors), e.g., orientation and gravity. (The orientation sensor will be used to benchmark the orientation filter you construct in this project.)

About/Help This displays some information about the app.

Clicking the red gear in any view where it is present brings up the pref-

erences view (Figure 3(c)). Changes made in this view are stored in the device. The most important fields are the Measurement>Frequency which determines the measurement rate (default: 100 Hz); the Network settings which are needed to determine where data is streamed; and the Log settings which decide where the log file is created.

4 Data collection and filter design

It is finally time for us to collect sensor data and develop our filter. We will do this in steps, starting with some data analysis and then move on to the prediction and update steps of the filter. Along the way, you will have to make several design choices, derive suitable models and study the consequences using your smartphone data.

4.1 Initial data examination

Given the Android application you can already visualize data on your phone, but in order to run an EKF in Matlab we would like to **transfer the data to a computer:**

- Download all the supporting Matlab files and the `sensorfusion.jar` file the course homepage. Extract the files in the zip-archive to a folder on your computer.
- Run the `startup` function to initialize the environment correctly. Note, this must be done each time Matlab is restarted.
- Connect the smartphone to a local Wi-Fi, see Appendix B.
- Determine the IP of your lab computer. One way to do this is to run `showIP` in Matlab, which is one of the project files.
- Enter the lab computer's IP into the app's configuration view (see Figure 3(c).) At the same time, make sure the port is set to 3400 and that the measurement frequency is 100 Hz.
- Always start your Matlab program (e.g., `filterTemplate` discussed below) before initiating the streaming from the smartphone, it acts as a server to which the client on the phone tries to connect to. Otherwise, some bufferts will be full and the app can crash.

Get to know your data: Spend a few minutes to play around with the app and the sensors in the "Select Sensor" view (on your phone) to get an initial feeling for the data you are going to work with.

- **Task 2:** We previously discussed how to describe our system in terms of inputs, biases, etc. A basic insight regarding these question is that the answers depend on the problem at hand and the sensor properties.

Use `filterTemplate` to collect a few seconds of data and compute mean and variance for the accelerometer, gyroscope, and magnetometer. Also, analyse the measurements to see if the noise is Gaussian and if it has any trends. What do the results tell you? Your results should be used to tune the filter.

To summarise your conclusions your report should contain: 1) Histograms of measurements for some sensors and axes. 2) A plot of the signals over time. Describe if there are any trends and biases; if there are, discuss how you can deal with them. 3) Compute mean and covariances for the acceleration vector, angular velocity vector, and magnetic field when the phone is placed flat on table.

Note: Most smartphones automatically calibrate their magnetometer measurements. To facilitate this process, turn on the sensor and spin the phone gently in many different directions. Also note that magnetometers are sensitive to its environment and may for instance provide significantly different values when placed close to metallic objects. To observe its behavior in this respect you can study how the magnitude of the field (measured in μT) varies when you place it, e.g., on different surfaces.

Hint: Unavailable measurements are marked with a NaN (not a number) value. Make sure to take this into consideration when computing means etc. The function `isnan(x)` can be used to find NaN values; the average acceleration can be computed as:

```
mean(meas.acc(:, ~any(isnan(meas.acc), 1)), 2)
```

4.2 Design the EKF time update step

In order to derive a discrete time prediction model we make use of the continuous time model in (2d). We further assume that the process noise enters additively on $\omega(t)$, but in order to simplify the expressions we assume that $\omega(t) = \omega_{k-1}$ and $v(t) = v_{k-1}$ are piece-wise constant between the sampling times t_{k-1} and t_k. The complete continuous time model is

$$\dot{q}(t) = \frac{1}{2} S(\omega_{k-1} + v_{k-1}) q(t), \qquad \text{for } t \in [t_{k-1}, t_k), \tag{5}$$

where $v_k \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_v)$.

- **Task 3:** Solve the differential equation in (5) and use the relation $\exp \mathbf{A} \approx \mathbf{I} + \mathbf{A}$ to derive a discretized model of the form

$$q_k = F(\omega_{k-1}) q_{k-1} + G(\hat{q}_{k-1}) v_{k-1}, \tag{6}$$

  where $q_k = q(t_k)$ and $q_{k-1} = q(t_{k-1})$. Your task is thus to derive the expressions for F and G.

  Hint: At some point you should obtain a term $G(q_{k-1})v_{k-1}$. As indicated in (6), we suggest that you can approximate this term as $G(\hat{q}_{k-1})v_{k-1}$. You can do that, but please include a short motivation why this is what the EKF procedure would suggest you to do.

- **Task 4:** Based on the model that you derived in the previous task, write a Matlab function `[x, P] = tu_qw(x, P, omega, T, Rw)` that implements the time update function, where `omega` is the measured angular rate, `T` the time since the last measurement, and `Rw` the process noise covariance matrix. For your convenience, the functions $\bar{S}(q)$ and $S(\omega)$ are available in App. A.2.4 and A.2.5, respectively.

  Furthermore, write a similar function to handle the time update in the case that there is no angular rate measurement available.

- **Task 5:** Make a copy of `filterTemplate.m` and add your own function `[x, P] = tu_qw(x, P, omega, T, Rw)` to do the time update. Use the previously estimated covariance as input to the filter.

  You should now have an EKF (without measurement updates) that provides very reactive estimates of the rotations, but that lacks absolute orientation. How does it perform? What happens if you start the filter with the phone on the side instead of laying face up on the desk? Why? Shake the phone and explain what happens.

Note: In order for the quaternion representation of the orientation to work, it is essential that the quaternion is of unit length. The extended Kalman filter does not automatically maintain this quaternion property. Among the files that you have downloaded, there is a function `[x, P] = mu_normalizeQ(x, P)` that normalizes the quaternion, but you need to include that function at suitable places in your filter in order to ensure unit length.

4.3 An EKF update using accelerometer measurements

The measurements from the accelerometer can be modelled using the relation

$$y_k^a = Q^T(q_k)(g^0 + f_k^a) + e_k^a, \tag{7}$$

where g^0 is the nominal gravity vector, f_k^a is a force which is non-zero if the sensor accelerates in the world frame and e_k^a is the measurement noise.

- **Task 6:** Describe the EKF update using y_k^a in an equation and write a Matlab function `[x, P] = mu_g(x, P, yacc, Ra, g0)` that implements the accelerometer measurement update under the assumption that $f_k^a = 0$. Here `yacc` is shorthand for y_k^a and `Ra` is the measurement noise covariance matrix. You can estimate g^0 from data where you keep the phone flat on a table.

  The functions to compute $Q(q)$ and $dQ(q)/dq$ are available in App. A.2.3 and A.2.6, respectively.

- **Task 7:** Add the *accelerometer update* to the EKF. Now, the orientation should be correct up to a rotation in the horizontal plane. Test the sensitivity to specific forces (body accelerations) with different experiments. As an example, slide the device quickly back and forth on the horizontal surface of a table.

A weakness of the acceleration update is that it is based on the assumption that f_k^a is zero, i.e., that the phone is not accelerating in the world frame. If we use the same update function when f_k^a is large, it is not likely to perform well.

- **Task 8:** Add a simple *outlier rejection* algorithm for a_k, based on the assumption that $\|a_k\| \approx g = 9.81$ when f_k^a is zero. The idea is to simply skip the updates when we think the measurement is an outlier. Use the orientation view's `setAccDist` function to indicate, in the display, when the accelerometer measurements are considered disturbed/outliers (simply write `ownView.setAccDist(accOut)` where accOut is either zero or one).

  Repeat the experiments in the previous task.

4.4 An EKF update using magnetometer measurements

The magnetometer measures the earth magnetic field in sensor coordinates and can be modelled as

$$y_k^m = Q^T(q_k)(m^0 + f_k^m) + e_k^m, \tag{8}$$

where m^0 is the earth magnetic field in world coordinates, f_k^m represents other sources of magnetic fields and e_k^m is the measurement error. By definition m_0 should have no field component in the west-east direction, which is the underlying reason why compasses are able to point towards north. Hence, the nominal magnetic field can be derived from a magnetic field measurement, $m = \begin{bmatrix} m_x & m_y & m_z \end{bmatrix}^T$, as the phone lies horizontally without knowledge of the direction of north, using the relation

$$m^0 = \begin{pmatrix} 0 & \sqrt{m_x^2 + m_y^2} & m_z \end{pmatrix}^T. \tag{9}$$

- **Task 9:** Write down the EKF update using y_k^m in an equation and implement[2] a Matlab function `[x, P] = mu_m(x, P, mag, m0,Rm)` that implements the magnetometer measurement update under the assumption that $f_k^m = 0$. Here `mag` is shorthand for y_k^m, `m0` is m_0 and `Rm` is the measurement noise covariance matrix, where `m0` and `Rm` should be estimated from training data (preferably containing very little disturbances).

- **Task 10:** Add the magnetometer update to your EKF and study its properties. You should now have an orientation filter that behaves much like the one implemented in the phone. What happens if you introduce a magnetic disturbance? (The fields next to your computer should be enough.)

[2] It should be noted that the direct usage of (8) is far from the only possible strategy to handle magnetometer measurements. A popular one is, for instance, to use the current estimate of q to project y_k^m onto the xy-plane in the world frame and only use that part of the measurement in the update.

- **Task 11:** Similar to the accelerometer update, we have based our update on the assumption that the disturbances are zero, i.e., that $f_k^m = 0$. Try to come up with a way to perform *outlier rejection* for magnetic disturbances, e.g., by estimating the strength of the magnetic field and reject measurements when the field strength differs too much from what you expect. Note that the expected magnitude, say L_k, of m_k may drift slowly over time even in the absense of disturbances (i.e., the magnitude of m^0 may drift); a simple filter that estimates the magnitude could, for instance, be a AR(1)-filter of the type $\hat{L}_k = (1-\alpha)\hat{L}_{k-1} + \alpha\|m_k\|$, where α is a small positive number.

  Implement the outlier rejection, and evaluate if it helps. Use the orientation view's `setMagDist` function to indicate when the magnetic field measurements are considered disturbed/outliers. What assumptions do you rely on? When are they reasonable? What happens now when you introduce a magnetic disturbance? Why?

  Use the `q2euler` function (Appendix A.2.2) and plot the Euler angles of both your orientation filter and the built in filter in the phone.

- **Task 12:** Finally, evaluate your filter for different combinations of sensors, for instance, only accelerometer and magnetometer (no gyroscope).

A Provided Code

The following functionality is provided in the zip-archive `http://goo.gl/t1j03N`. Make sure to run the included `startup` function to initialize the package before doing anything else. Without this step it is not possible to establish a connection with the phone. When using `filterTemplate` always start the Matlab program before initializing the streaming from the phone as the Matlab program acts as a server for the data from the phone.

A.1 Code Skeleton

```
function [xhat, meas] = filterTemplate(calAcc, calGyr, calMag)
% FILTERTEMPLATE  Filter template
%
% This is a template function for how to collect and filter data
% sent from a smartphone live.  Calibration data for the
% accelerometer, gyroscope and magnetometer assumed available as
% structs with fields m (mean) and R (variance).
%
% The function returns xhat as an array of structs comprising t
% (timestamp), x (state), and P (state covariance) for each
% timestamp, and meas an array of structs comprising t (timestamp),
% acc (accelerometer measurements), gyr (gyroscope measurements),
% mag (magnetometer measurements), and orint (orientation quaternions
% from the phone).  Measurements not availabe are marked with NaNs.
%
% As you implement your own orientation estimate, it will be
```

```
% visualized in a simple illustration.  If the orientation estimate
% is checked in the Sensor Fusion app, it will be displayed in a
% separate view.
%
% Note that it is not necessary to provide inputs (calAcc, calGyr, calMag).

  %% Setup necessary infrastructure
  import('com.liu.sensordata.*');  % Used to receive data.

  %% Filter settings
  t0 = [];  % Initial time (initialize on first data received)
  nx = 4;   % Assuming that you use q as state variable.
  % Add your filter settings here.

  % Current filter state.
  x = [1; 0; 0 ;0];
  P = eye(nx, nx);

  % Saved filter states.
  xhat = struct('t', zeros(1, 0),...
                'x', zeros(nx, 0),...
                'P', zeros(nx, nx, 0));

  meas = struct('t', zeros(1, 0),...
                'acc', zeros(3, 0),...
                'gyr', zeros(3, 0),...
                'mag', zeros(3, 0),...
                'orient', zeros(4, 0));
  try
    %% Create data link
    server = StreamSensorDataReader(3400);
    % Makes sure to resources are returned.
    sentinel = onCleanup(@() server.stop());

    server.start();  % Start data reception.

    % Used for visualization.
    figure(1);
    subplot(1, 2, 1);
    ownView = OrientationView('Own filter', gca);  % Used for visualization.
    googleView = [];
    counter = 0;  % Used to throttle the displayed frame rate.

    %% Filter loop
    while server.status()  % Repeat while data is available
      % Get the next measurement set, assume all measurements
      % within the next 5 ms are concurrent (suitable for sampling
      % in 100Hz).
      data = server.getNext(5);

      if isnan(data(1))  % No new data received
        continue;        % Skips the rest of the look
      end
      t = data(1)/1000;  % Extract current time

      if isempty(t0)  % Initialize t0
        t0 = t;
```

```
        end

        acc = data(1, 2:4)';
        if ~any(isnan(acc))  % Acc measurements are available.
          % Do something
        end
        gyr = data(1, 5:7)';
        if ~any(isnan(gyr))  % Gyro measurements are available.
          % Do something
        end

        mag = data(1, 8:10)';
        if ~any(isnan(mag))  % Mag measurements are available.
          % Do something
        end

        orientation = data(1, 18:21)';  % Google's orientation estimate.

        % Visualize result
        if rem(counter, 10) == 0
          setOrientation(ownView, x(1:4));
          title(ownView, 'OWN', 'FontSize', 16);
          if ~any(isnan(orientation))
            if isempty(googleView)
              subplot(1, 2, 2);
              % Used for visualization.
              googleView = OrientationView('Google filter', gca);
            end
            setOrientation(googleView, orientation);
            title(googleView, 'GOOGLE', 'FontSize', 16);
          end
        end
        counter = counter + 1;

        % Save estimates
        xhat.x(:, end+1) = x;
        xhat.P(:, :, end+1) = P;
        xhat.t(end+1) = t - t0;

        meas.t(end+1) = t - t0;
        meas.acc(:, end+1) = acc;
        meas.gyr(:, end+1) = gyr;
        meas.mag(:, end+1) = mag;
        meas.orient(:, end+1) = orientation;
      end
    catch e
      fprintf(['Unsuccessful connecting to client!\n' ...
        'Make sure to start streaming from the phone *after*'...
              'running this function!\n']);
      fprintf(['Error: ' e.message '\n']);
    end
end
```

A.2 Utility Functions

A.2.1 Normalize quaternion

```
function [x, P] = mu_normalizeQ(x, P)
% MU_NORMALIZEQ  Normalize the quaternion

  x(1:4)  = x(1:4) / norm(x(1:4));
  if x(1) < 0,
    x(1:4) = -x(1:4);
  end
end
```

A.2.2 Convert quaternions to Euler angles

```
function euler = q2euler(q)
% Q2EULER  Convert quaternions to Euler angles

  euler = zeros(3, size(q, 2));

  xzpwy = q(2, :).*q(4, :) + q(1, :).*q(3, :);

  IN = xzpwy+sqrt(eps)>0.5;  % Handle the north pole
  euler(1, IN) = 2*atan2(q(2, IN), q(1, IN));
  IS = xzpwy-sqrt(eps)<-0.5;  % Handle the south pole
  euler(1, IS) = -2*atan2(q(2, IS), q(1, IS));

  I = ~(IN | IS);  % Handle the default case
  euler(1, I) = atan2(-2*(q(1, I).*q(3, I) - q(1, I).*q(4, I)),...
                      1-2*(q(3, I).^2 + q(4, I).^2));
  euler(3, I) = atan2(2*(q(3, I).*q(4, I) - q(1, I).*q(2, I)),...
                      1-2*(q(2, I).^2 + q(3, I).^2));

  euler(2, :) = asin(2*xzpwy);

  euler(1, :) = rem(euler(1, :), 2*pi);
end
```

A.2.3 $Q(q)$

```
function Q=Qq(q)
% The matrix Q(q)
   q0=q(1);   q1=q(2);   q2=q(3);   q3=q(4);
   Q = [2*(q0^2+q1^2) - 1   2*(q1*q2-q0*q3)     2*(q1*q3+q0*q2);
        2*(q1*q2+q0*q3)     2*(q0^2+q2^2) - 1   2*(q2*q3-q0*q1);
        2*(q1*q3-q0*q2)     2*(q2*q3+q0*q1)     2*(q0^2+q3^2) - 1];
end
```

A.2.4 $\bar{S}(q)$

```
function S=Sq(q)
% The matrix S(q)
    q0=q(1);    q1=q(2);    q2=q(3);    q3=q(4);
    S=[-q1 -q2 -q3;
        q0 -q3  q2;
        q3  q0 -q1;
       -q2  q1  q0];
end
```

A.2.5 $S(\omega)$

```
function S=Somega(w)
% The matrix S(omega)
    wx=w(1);    wy=w(2);    wz=w(3);
    S=[ 0  -wx  -wy  -wz;
       wx    0   wz  -wy;
       wy  -wz    0   wx;
       wz   wy  -wx    0];
end
```

A.2.6 $dQ(q)/dq$

```
function [Q0, Q1, Q2, Q3] = dQqdq(q)
% The derivative of Q(q) wrt qi, i={0,1,2,3}

    q0=q(1);    q1=q(2);    q2=q(3);    q3=q(4);
    Q0 = 2* [2*q0   -q3    q2;
               q3  2*q0   -q1;
              -q2    q1  2*q0];
    Q1 = 2* [2*q1    q2    q3;
               q2     0   -q0;
               q3    q0     0];
    Q2 = 2* [   0    q1    q0;
               q1  2*q2    q3;
              -q0    q3     0];
    Q3 = 2* [   0   -q0    q1;
               q0     0    q2;
               q1    q2  2*q3];
end
```

A.2.7 OrientationView

The class orientation view provides a way to easily visualize orientations. The class is used in the filter template.

self = OrientationView(figname) Create a OrientationView figure with a given figure name.

setOrientation(self, q, P) Set the orientation to display, if P is given it is used to visualize the uncertainty in the orientation.

title(self, str) Set the title of the view to str.

setStandStill(self, flag) Set the stand still indicator on or off.

setAccDist(self, flag) Set the acceleration disturbance indicator on or off.

setMagDist(self, flag) Set the magnetometer disturbance indicator on or off.

A.2.8 showIP()

showIP provides easy access to the IP the computer is using.

A.3 sensordata.jar

The Java package sensordata.jar provides the functionality needed to communicate with the Sensor Fusion app and to integrate the measurements in real time into for instance Matlab. The documentation of the public interface can be found here: http://goo.gl/I7KpfL.

B Configure the Phone for eduroam

We would to stream measurement data from the Android device to a computer. If you are at home, it normally works if you connect both to the same router. If you are at Chalmers we normally recommend that you connect via eduroam.

To connect via a wireless network to eduroam the following instructions are applicable for most Android phones (see also Figure 4):

1. Drag down the status bar at the top.
2. Click the configuration to bring out the settings menu.
3. Select the wifi settings.
4. Find eduroam in the list, and configure it with the following values:
   - EAP method: PEAP
   - Phase 2 authentification: MSCHAPV2
   - Username: CID@chalmers.se
   - Password: CID-password

If you have difficulties connecting your phone, please contact Lennart Svensson for further assistance.

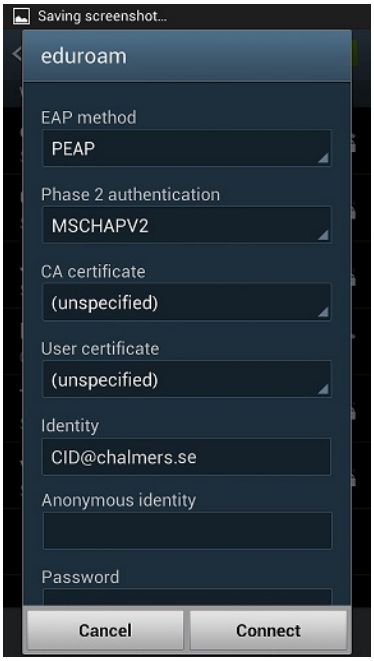


Figure 4: Illustration of the basic steps needed to configure the device for the eduroam network.

References

[1] Fredrik Gustafsson, *Statistical Sensor Fusion*, Studentlitteratur, 2010.

[2] G. Hendeby, F. Gustafsson, and N. Wahlström, Teaching Sensor Fusion and Kalman Filtering using a Smartphone. *The 19th World Congress of the International Federation of Automatic Control (IFAC)*, Cape Town, South Africa, August, 2014.