ERP420

Research Project

Practical 2 Report

Name and Surname	Student Number	Signature
Karishma George	19003618	

By signing this assignment I confirm that I have read and am aware of the University of Pretoria's policy on academic dishonesty and plagiarism and I declare that the work submitted in this assignment is my own as delimited by the mentioned policies. I explicitly declare that no parts of this assignment have been copied from current or previous students' work or any other sources (including the internet), whether copyrighted or not. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work. I understand that I will be subjected to disciplinary actions should it be found that the work I submit here does not comply with the said policies.

October 7, 2022

1. Introduction & Baseline

1.1. Introduction

Having sensors implemented in all three axes of motion, magnetic, angular rate, and gravity (MARG) sensors have been shown to be extremely useful in their ability to obtain inertial and magnetic measurements about their body frame. This implies that the sensor is able to detect and quantify any rotary disturbances in the form of changes in angular velocity, gravitational force, and magnetic field distortions. Sensor fusion can be used to combine these readings along all three axes to approximate the orientation of the sensor-bearing body.

This paper examines the application of an unscented Kalman filter (UKF) for sensor fusion and, consequently, orientation estimation from motion. First, a concise introduction to the baseline experiment, which implements sensor fusion using the gradient descent algorithm, will be developed and carried out. Following this, the conceptualization and implementation of the UKF model will be presented. The model will be evaluated using an experimental setup that compares the performance of the baseline filter and the UKF over a series of iterative runs to evaluate the model's overall accuracy in orientation estimation.

1.2. Baseline model

A computationally inexpensive method developed by Madgwick, as described in [1] is used as the baseline model to apply sensor fusion and consequently estimate orientation. The filter makes use of a quaternion representation of orientation in the model state. The change in the orientation from experiencing angular velocity is described by equation 1.1

$$ {}^{S}_{E}\dot{\boldsymbol{q}} = \frac{1}{2}{}^{S}_{E}\hat{\boldsymbol{q}} \otimes {}^{S}\boldsymbol{\omega} \text{ and } {}^{S}_{E}\boldsymbol{q}_{\omega,t} = {}^{S}_{E}\hat{\boldsymbol{q}}_{est,t-1} + {}^{S}_{E}\dot{\boldsymbol{q}}_{\omega,t}\Delta t \tag{1.1} $$

where ${}^{S}\boldsymbol{\omega}$ represent a 4-input vector equation of the radian angular velocity. The angular velocity is transformed into a quaternion in the earth's frame of reference which is integrated over the sample time and added to the current quaternion estimate. While integration of the gyroscopic rate provides an angular displacement estimate of the system, it is measured relative to the body frame. The magnetometer readings are used to compute a unique orientation estimate relative to the magnetic field of earth. To produce a unique solution the gravitational vector is also used as a frame of reference. A unique solution is found by identifying the orientation that minimised measured deviations from the earth's magnetic field and gravitational force. Equation 1.2 below,

$$ \begin{gathered} {}^{S}_{E}\boldsymbol{q}_{\nabla,t} = {}^{S}_{E}\hat{\boldsymbol{q}}_{est,t-1} - \mu_t \frac{\nabla \boldsymbol{f}}{\|\nabla \boldsymbol{f}\|} \\ \nabla \boldsymbol{f} = \begin{cases} \boldsymbol{J}_{g}^{T}\left({}^{S}_{E}\hat{\boldsymbol{q}}_{\mathrm{est}\,t-1-1}\right) \boldsymbol{f}_{g}\left({}^{S}_{E}\hat{\boldsymbol{q}}_{est,t-1}, {}^{S}\hat{\boldsymbol{a}}_{t}\right) \\ \boldsymbol{J}_{g,b}^{T}\left({}^{S}_{E}\hat{\boldsymbol{q}}_{est,t-1}, {}^{E}\hat{\boldsymbol{b}}\right) \boldsymbol{f}_{g,b}\left({}^{S}_{E}\hat{\boldsymbol{q}}_{est,t-1}, {}^{S}\hat{\boldsymbol{a}}, {}^{E}\hat{\boldsymbol{b}}, {}^{S}\hat{\boldsymbol{m}}\right) \end{cases} \end{gathered} \tag{1.2} $$

defines the magnetic field vector and gravitational vector as quaternions in the sensor frame which is used to derive the Jacobian. Equation **??** also shows the gradient descent estimation, which uses the Jacobian to generate a new quaternion estimate. Sensor fusion is applied as given in

$$ {}^{S}_{E}\boldsymbol{q}_{est,t} = \gamma_t {}^{S}_{E}\boldsymbol{q}_{\nabla,t} + (1-\gamma_t){}^{S}_{E}\boldsymbol{q}_{\omega,t}, \quad 0 \leq \gamma_t \leq 1 \tag{1.3} $$

where the γ weight is used to proportionally combine the quaternion estimate derived from the gyroscope and the gradient descent algorithm to obtain the next orientation estimate.

The implementation of Madgwick's gradient descent filter consists of a globally accessible 4 $\times$ 1 state vector which is used to represent the quaternion components. The sample period is the inverse of the sample frequency, which is constant for every iteration. The implementation of [1] uses radian gyroscopic rates, and unit vector representations of the accelerometer and magnetometer readings. The objective function from 1.2 is derived by stacking the derived magnetometer and accelerometer equations in [1] into a 6 $\times$ 1 matrix. The Jacobian equations are also similarly obtained. The normalised objective function is computed by calculating the product of the Jacobian transpose and the stacked objective function matrix, which estimates the orientation when the sensors align with the earth's gravitational field and magnetic vector. Similarly, the gyroscope rotation quaternion estimate is obtained using is computed using 1.3. The two calculated measurements calculate the rate of quaternion change, and the actual quaternion estimate is obtained by time integrating the sum of these estimates. The baseline implementation also caters for magnetic distortions by using the quaternion estimates to update the magnetometer observation model in each iteration, and hence cater for erroneous incline detection. The estimate-update actions are performed by a single iteration of the model, which is implemented as a single function call.

2. Research Model

2.1. Theoretical modelling

The unscented Kalman filter (UKF) is a sensor fusion algorithm that is used to combine data measured using a multi-sensor system to provide a non-linear approximation of a system state. When performing orientation estimation, the UKF combines the measurements obtained from the MARG sensor to update the quaternion-based system orientation estimates.

The UKF is an expansion of the Kalman filter (KF), which is adapted to generate non-linear orientation estimates. As result the process of generating orientation estimates are similar to that of the KF. Consequently, the UKF filter has three main matrix components that are used to describe the estimation system. The state vector $\boldsymbol{x}$ represents the current orientation state of the vector. The process model is a system of equations that are used to represent the advancement in the state vector estimates relative to the previous estimates and the combination of process noise $\boldsymbol{w}$. Equation 2.1 described in [2] represents a general form of the process model, where A represent the update function associated with the estimate.

$$x_{k+1} = A\left(x_k, \mathrm{w}_k\right) = \begin{pmatrix} q_k q_{\mathrm{w}} q_{\Delta} \\ \vec{\omega}_k + \overrightarrow{\mathrm{w}}_{\omega} \end{pmatrix} \tag{2.1}$$

A second system of equations named the measurement model described in [2] is used to show the mathematical relation between the state vector and the accelerometer and magnetometer measurements using the equation

$$z_k = H\left(x_k, \mathrm{v}_k\right). \tag{2.2}$$

Equation 2.2 also has an additional measurement noise term that influences the state of the system. The orientation system takes the direction of the gravitational vector (down) and the magnetic field vector of the earth (North) as reference directions. Consequently the measurements are converted into this frame of reference by generating vectors quaternions as given in 2.3 and a vector rotation using the current quaternion estimate is performed as given in 2.4 [2].

$$g \equiv (0, [g_x, g_y, g_z]) \equiv (0, \vec{g}) \tag{2.3}$$

$$g' = q_k] \times g \times q_k^{-1} \tag{2.4}$$

Since a non-linear measurement model is present, an alternative method of approximating the next estimate, namely the unscented transform is used in the UKF. As shown in Figure 2.1, the unscented transform is able to use the mean and covariance of one point and apply a non-linear function to obtain these statistics in the non-linear frame, thus updating the measurements. The mean and covariance give very little detail about the actual approximation. Therefore using a single point only i.e the mean to approximate the next estimate may yield an inaccurate projection. Therefore additional points that are projected around the mean, named sigma points are generated and the unscented transform applied. Cholesky decomposition is applied on the symmetric covariance matrix P_{k-1}, to generate the S-matrix. The S-matrix's column vector is extracted and multiplied by a factor of $\sqrt{2n}$ to generate these sigma points. The sigma points are generated according to 2.5[2]

$$\begin{aligned} S &= \sqrt{P_{k-1}} \quad \text{and} \\ \mathscr{W}_{i,i+n} &= \text{ columns } \left(\pm\sqrt{2n \cdot (P_{k-1})}\right). \end{aligned} \tag{2.5}$$

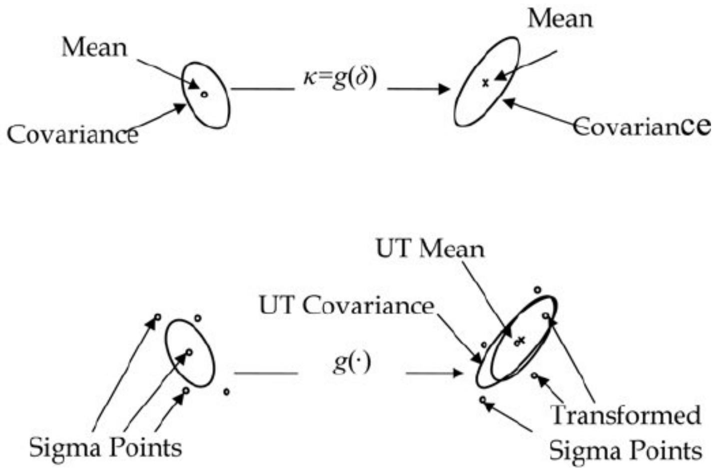

Figure 2.1: Visualisation of the unscented transform[3].

The sigma point are assigned weights to further improve the accuracy of estimation. These weights are given by 2.6 which are adapted from [4]. The constants α, β and κ are UKF constants equal to $0.001, 2$ and 0 respectively, and n represents the number of sigma points.

$$w_0^c = \frac{\lambda}{n+\lambda} + \left(1 - \alpha^2 + \beta\right) \text{ and } w_i^c = \frac{\lambda}{2(n+\lambda)} \tag{2.6}$$

$$\text{where } \lambda = \alpha^2(n+\kappa) - n \tag{2.7}$$

The sigma point estimates are passed through the process and state models respectively, and the mean z_k^- and covariance P_{zz} of the outputs of each model are computed, with zero process or measurement noise updates. The innovation v_k, and the innovation covariance P_{vv}, which is the difference between the estimated orientation and the measured orientation is computed using the equations given in 2.8. R matrix is the covariance of the measurement model.

$$\nu_k = z_k - z_k^- \text{ and } P_{\nu\nu} = P_{zz} + R[2] \tag{2.8}$$

The state estimates are updated by the equation 2.9 where K_k represents the Kalman gain for the sample.

$$\hat{x}_k = \hat{x}_k^- + K_k \nu_k[2]. \tag{2.9}$$

The Kalman gain is required to be updated for the next estimate using equation 2.10

$$K_k = P_{xz} P_{\nu\nu}^{-1}[2]. \tag{2.10}$$

However,equation 2.10 requires the co-variance matrix P_{vv} as given in 2.8 and the cross-correlation P_{xz} given by 2.11 as :

$$P_{xz} \hat{=} \frac{1}{2n} \sum_{i=1}^{2n} \left[\mathscr{Y}_i - \hat{x}_k^-\right] \left[\mathscr{Z}_i - z_k^-\right]^T [2]. \tag{2.11}$$

It is important to note that the co-variances will be multiplied by the calculated sigma point weights at each summation step. The state covariance for the state vector is also subsequently updated using equation 2.12 obtained from [2]

$$P_k = P_k^- - K_k P_{\nu\nu} K_k^T \tag{2.12}$$

2.2. Model implementation

A 4 $\times$ 1 vector is used to represent the quaternion orientation of the vector given by 2.13 where q_0 represents the scalar component of the quaternion and q_1 to q_3 represent the vector quaternions.

$$\left[\mathbf{q} = q_0 \quad q_1 \quad q_2 \quad q_3 .\right] \tag{2.13}$$

As described in [5], the discretised process model which is used to update the quaternion estimate can be represented by 2.14, where $\mathbf{I}_4$ represents an identity matrix. The angular velocity update term that is used is derived from the equation 2.15, which is the dfferential model representing the rate of angular rotation in the body frame:

$$\boldsymbol{q}_k = \left(\boldsymbol{I}_4 + \frac{1}{2}\delta t [\boldsymbol{\Omega}\times]_k \right) \boldsymbol{q}_{k-1} + \boldsymbol{w}_k \tag{2.14}$$

$$\begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{pmatrix} \tag{2.15}$$

The prediction state of the UKF utilises equation 2.15, and 2.5 to make a forward estimate using the gyroscope's angular displacement that is obtained by time integrating the angular velocity can be modelled according to the following algorithm given in 1

Algorithm 1 Process model angular displacement for forward estimate

---

Require: angular velocity, time sample (dt)
```
g ← x, y and z axis angular acceleration
I ← generate identity matrix
F ← I + gyroscope velocity matrix * dt * 0.5
return F
```

The sigma points, state and covariance weights are generated using the unscented transform algorithm given in 2

Algorithm 2 Unscented transformation

Require: X, P, λ,,α, κ
```
λ ←α² * ( n + κ) - n
L ← cholesky factors of P
F ← I + gyroscope velocity matrix * dt * 0.5
Weights ← compute the necessary weights
for number of columns in L do
    S ← sqrt(Pk)
    W ← add positive sigma points
    W ← add negative sigma points
    return sigma points, weights
end for
```

In order to begin the prediction process, the angular displacement of the gyroscope needs to be described in the quaternion reference frame according to the model 2.16. The covariance

Q of this model is also computed to aid in the prediction process. Algorithm 3 shows the generation of this initial quaternion model which will be used in the prediction process.

$$\dot{\boldsymbol{q}} = \frac{1}{2}\boldsymbol{q} \otimes \omega[5] \tag{2.16}$$

Algorithm 3 Process model angular displacement for forward estimate

---

Require:

```
L ← 0.5 * q̂_ω
Q ← cov{gyro} * cov{L} * cov{gyro}^T
F ← I + gyroscope velocity matrix * dt * 0.5
sigma points, state weights ← generate sigma points of the state estimates
projection ← F * sigma points to forward project estimates
projected mean, projected covariance n ← use state weights to obtain the sigma points
    mean and covariance.
projected covariance ← projected covariance + Q
return projected mean, projected covariance
```

The forward estimates are iterated through stages of the update state,which is used to update the sigma estimates using the accelerometer and gravity reference, and the magnetometer reference. Algorithm 4 shows the processes followed for each update, which results in state and covariance model that contained updated estimates. Algorithms 1 to 4 are used in sequence for every MARG measurement sample obtained.

Algorithm 4 Update state

Require: X, P, sigma points, weights, accelerometer/magnetometer measurement, noise covariance R

```
Y-obs ← obtain true observation estimate
for all the sigma points do
    Y-i ← perform measurement model update using weighted estimates
    Y-mean ← addition of weighted mean value
end for
for all the sigma points do
    Y-diff ← difference between updated sigma points and the Y mean
    X-diff ← difference between sigma points and the X value
    Pyy ← sums weighted Y-diff * Y-diff.T
    Pxy ← sums weighted X-diff * Y-diff.T
end for
Pyy ← Pyy + R
K ← Pxy * Pyy.I
Innovation ← Y-obs - y-mean
X ← X + Innovation
P ← P - K * Pyy * P.T
```

---

3. Implementation Verification

3.1. Experimental setup

A controlled experiment used to quantify the orientation estimation capability of the baseline and research models is conducted. In order to ensure a controlled experiment, a sampling frequency of 10Hz is used for both filter implementations so that the time interval over which the estimates are updated remains constant. A sequence of rotations is performed at 5s intervals over a period of 50s. The following sequence of rotations is applied : remain stationary, pitch up 30°, pitch down to 0°, roll right 60°, roll back to 0°, yaw right 120°, yaw back to 0°, pitch down 45° and roll left 20°, return to 0° back to the initial state, and remain stationary. Both the gradient descent and UKF are initially configured to compensate for bias errors, measurement deviation, and gyroscopic drift. The aforementioned bias compensation constants are computed by simulating stationary MARG sensors for a period of 60s and finding the mean and standard deviation results, which are used to configure the filters. The filter is also set to begin in the 0° position across all axes by setting the initial quaternion state to (1,0,0,0).

The ground truth represents the expected ideal orientation estimate and is calculated by linearly interpolating the final orientation angle of the sensor at the end of each 5s interval. The filter is visually verified by plotting the detected orientation of the baseline and UKF against the ground truth for a single execution of the filter, along with confidence bands indicating the precision of the estimate. The experiment is validated by visually comparing the similarity of the filters to the trend provided by the ground truth. Quantitative validation is performed by simulating 100 iterations of estimation using the baseline and UKF with the aforementioned sequence and plotting the root mean square errors (RMSE) at each sample point. The RMS is utilized to generate a statistically valid approximation of the baseline and UKF's average error rate. This allows one to quantitatively compare the estimation accuracy of the filters' performance.

3.2. Baseline model

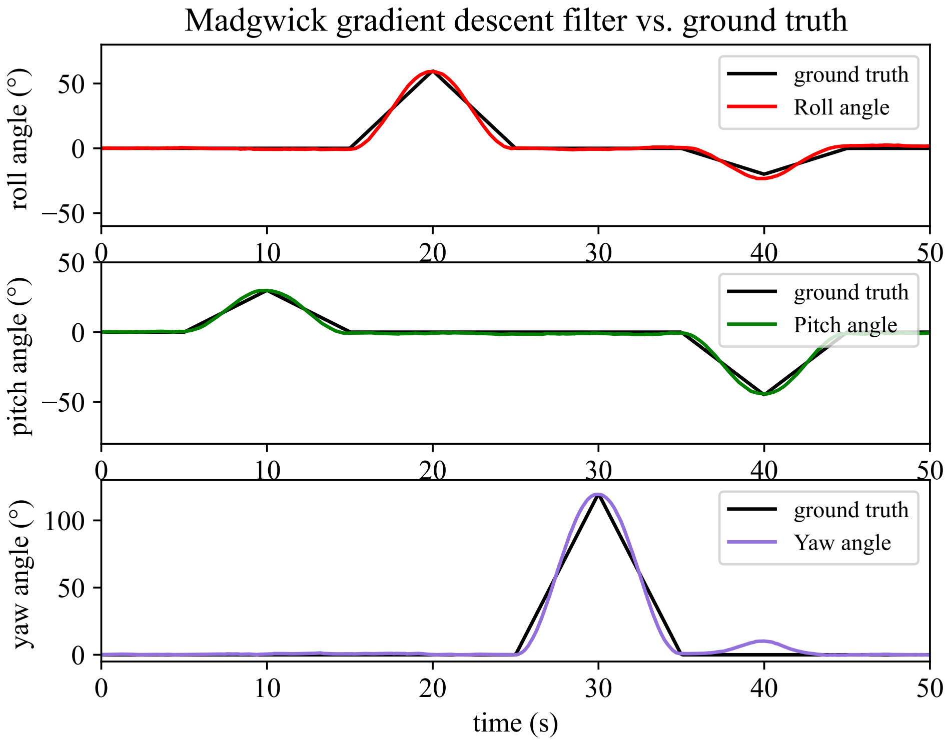

Figure 3.1: Madgwick gradient descent filter estimation

Figure 3.1 depicts a comparison between the orientation estimate for a single simulation run and the ground truth estimate .In the estimation of the roll, pitch, and yaw during steady-state periods, such as [0,15]s and [25,35]s for roll, [15, 35]s for pitch, and [0,25]s for yaw, it is observed that the filter estimate can remain close to the steady-state angle of 0 °. After applying a rotation and reversing it, the baseline filter is able successfully to reduce the effect of large gyroscopic drift errors along all three axes. The baseline implementation still shows slight drifting when performing rotations, as the ground truth estimate is slightly visible on the roll and pitch axes when a rotation is performed across and the yaw axis. This could be due to rounding when performing magnetometer compensation, resulting in a slight overestimation of the steady state by the gradient descent implementation after a rotation.

In the time frames in which a rotation is performed, the estimate provided by the baseline is a nonlinear approximation of angular changes. This is evident in the approximation's undershoot as the angular estimation increases up to the halfway point of the dynamic motion, at which point it begins to overestimate its approximation. The estimation error increases as the angle of rotation during the 5 second interval increases. The overshoot of the pitch angle appears to be smaller than the roll axis rotations and relatively insignificant in comparison to the yaw axis rotations. On the other hand, a smaller angle of rotation results in an undershoot of the rotation peaks, as the ground truth peaks are more distinct than the smooth change in rotation angles computed by the baseline implementation. With the baseline filter, a simultaneous rotation across two axes also results in a large error estimate on the third axis. This estimation occurs due to the interdependence of the axes during simultaneous rotations. Due to the simultaneous rotations, the readings from the magnetometer and accelerometer show a change in the y-axis measurements. This makes the estimate do a yaw-estimate compensation even though the axis isn't moving.

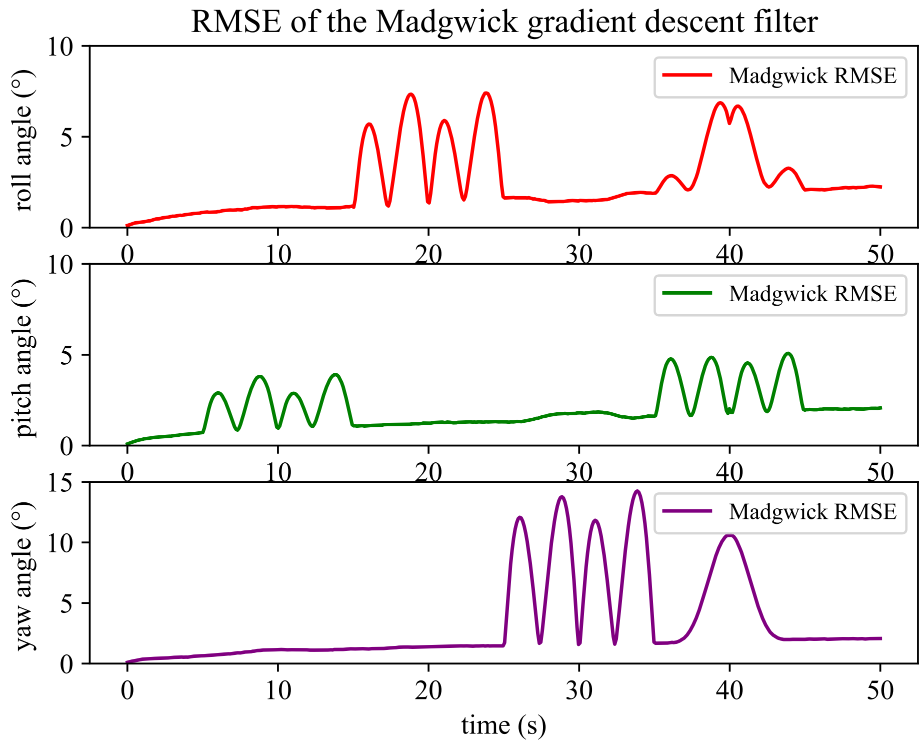

Figure 3.2: Madgwick gradient descent filter RMS errors over time

Figure 3.2 quantifies a better approximation of the overall fit of the baseline model's results relative to the ground truth reference when averaging over 100 iterations of the simulation execution. Across all the time samples, the roll angle error does not exceed 7°, pitch angle error does not exceed 5°, and yaw angle error does not exceed 15°. When considering steady-state approximations, the effect of drifting is more evident compared to Figure 1. The drift effect is calculated to occur at 0.40°/s for roll, 0.03°/s for pitch, and 0.02°/s for yaw. This is caused by rounding errors when performing bias compensation in the filter, resulting in the accumulation of estimation errors. The effect is not readily apparent but can grow unbounded over larger execution periods. The most significant RMSE errors are caused by non-linear estimation differences, which are represented as peaks that occur every 2.5s and reveal undershoot and overshoot errors when the system's orientation changes. Estimation errors form four peaks per change in rotation over the axes, where peak pairs represent the overshoot and undershoot errors for each action. With a yaw of 120° representing the rotation with the highest RMSE error of 15° and a pitch of 30° having the lowest error peaks of 5°, it can be observed that the RMSE error increases proportionally to the estimated angle change. As a result of the low degree of the approximation function, rotations below 20° lack the standard peaking, and the large errors are caused by overestimation. At 40s, the errors caused by motion in two axes as described approximate a 10° error in the RMSE iterations, indicating that the effect is alarmingly problematic when estimating orientation of multidimensional motion.

3.3. Research model

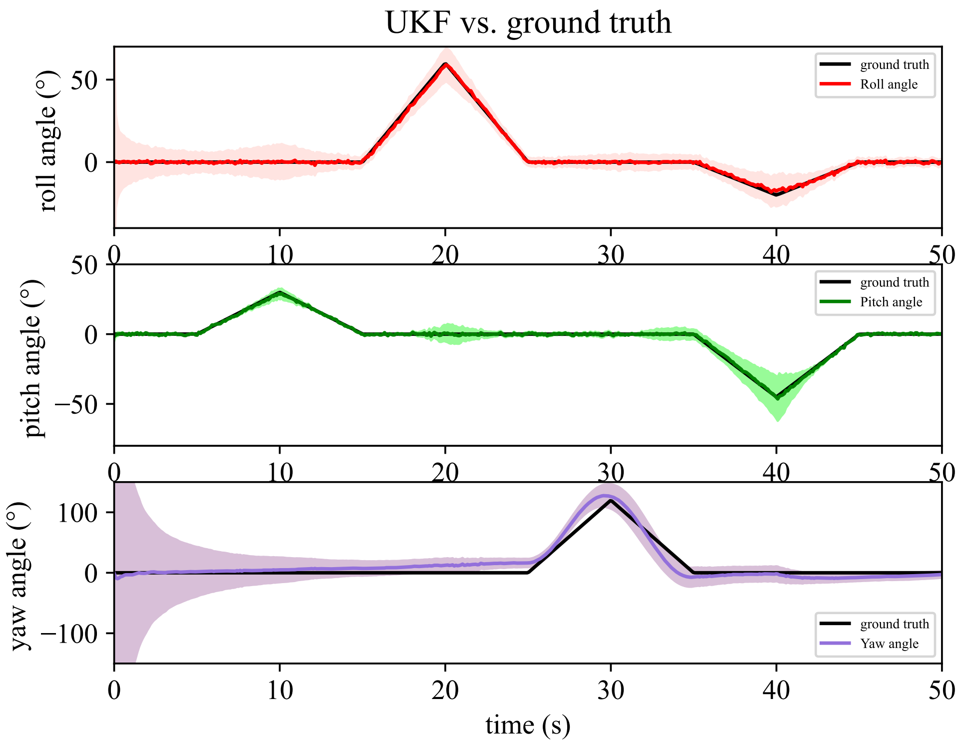

Figure 3.3: UKF estimation

When comparing the UKF's estimates to the ground truth in Figure 3.3 at the peak points where orientation changes (20s and 40s for roll, 10s and 40s for pitch, and 30s for yaw), the UKF is able to accurately estimate these sharp changes in the orientation without any overshoot. Compared to the baseline filter, where a non-linear approximation method is used to estimate the dynamic orientation, the UKF estimates the orientation by projecting the certainty of the next state estimate of the orientation across each axis. This means linear estimation outputs are also probable from applying the filter. For the roll, pitch, and yaw, the estimate lines show staggering approximations close to the true value of the estimate in comparison to the smooth results obtained by the baseline model. The measurement noise resulting from the sensor readings may result in projections that fluctuate about the actual orientation. The 95% confidence intervals, indicate stability of the estimate. For steady state motion where the orientation is 0°, the confidence interval is limited to within 0.01°.

Analyzing the confidence intervals also provides insight into the system's stability under dynamic motion. In the first 5s of estimation, the roll estimate has a standard deviation of 25° and the yaw estimate has a deviation greater than 100°. The roll estimation is accurate when compared to the ground truth, and the large standard deviation is a result of uncertainty caused by the absence of previous estimates. In contrast, the initial estimate of the yaw angle differs by 8°, resulting in a larger standard deviation and less stability. As additional points are sampled and the system becomes more stable, the deviation becomes more precise. Despite the deviation expectation, the system is able to estimate the orientation with near-perfect accuracy, especially for orientation changes as abrupt as 10s, 20s, and 40s, respectively.

While the yaw angle estimates follow the trend predicted by the ground truth estimate, orientation estimates still contain measured errors. It is observed that the yaw angle increases prior to 25 seconds, which is when the rotation is executed. The estimate is ahead of the actual orientation, resulting in an overshoot despite the fact that the peak angle of the rotation is estimated precisely. Returning to 0° also causes a nonlinear change that rapidly undershoots the estimation. The standard deviation indicates that the results are stable, which leads to the conclusion that the magnetometer values used to update the yaw may be improperly calibrated, resulting in a drift in the yaw detection, which leads to preemptive estimations of the orientation.

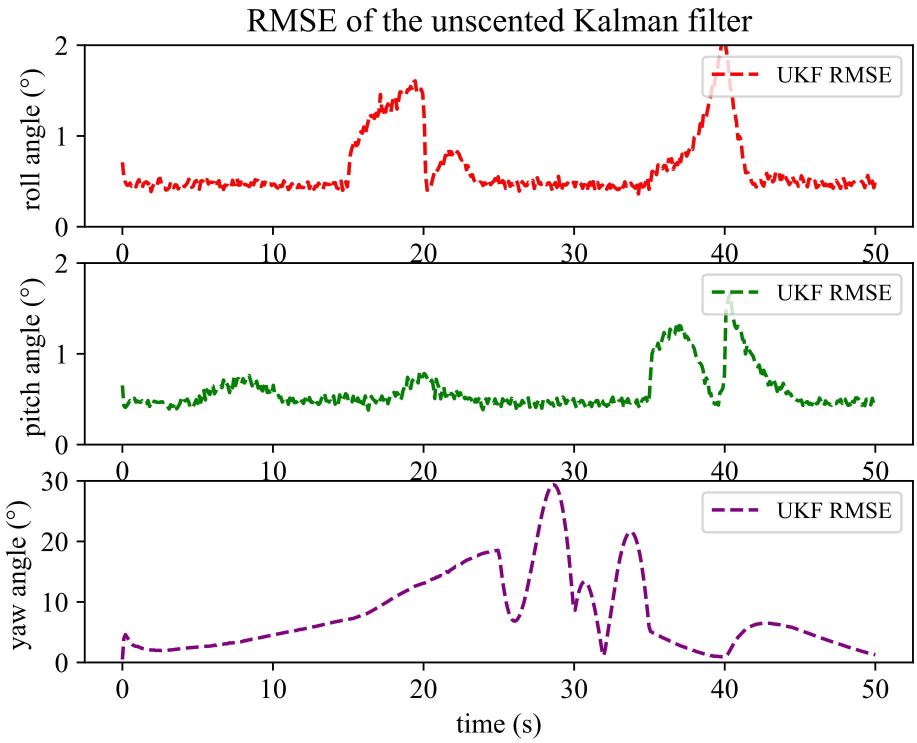

Figure 3.4: UKF RMS error over time

Figure 3.4 illustrates the RMSE following 100 rotation sequence iterations. The UKF is more resistant to estimation errors than the baseline, as demonstrated by a comparison of the results to Figure 2. Although steady-state drifts were prominent in the roll and pitch estimates of the baseline, the UKF error estimates have remained constant. The roll and pitch estimates have an error of only 0.4 degrees, indicating that the system estimates are extremely precise. A maximum of two degrees of error is also observed in these axes during rotations. Nevertheless, this is due to measurement errors caused by dynamic motion and the estimated orientation remains within 1 to 2 degrees of actual orientation estimation. Due to the estimated overshoot during a yaw, the yaw error can reach 30 degrees. In contrast to Figure 3.2, rotations on multiple axes have no impact on the estimated yaw as seen at 40s. This shows that utilizing the UKF renders the estimation of orientation across axes independent.

4. Addendum

4.1. [GA5.1] Application of appropriate engineering methods

The STEVAL-MKI217V1 evaluation kit, which includes an inertial measurement unit (IMU) and a magnetometer, is utilised as the sensor input due to its capacity to measure rotational velocity, acceleration due to motion, and magnetic field variations around its body. This sensor system detects changes for 9-degrees of freedom, which is superior to systems that only use an IMU unit. IMU systems that have temperature measurement are also available for consideration, however its advantage is to refine orientation estimates rather than to identify a unique orientation solution, and is thus not required to be implemented. The communication protocol selected for use with the evaluation kit is I^2C. I^2C is much slower than its SPI counterpart, but it only necessitates a single wire for communication. Because only read operations are performed on the evaluation kit, I^2C, a half-duplex communication protocol, is more appropriate. Real-world sensor measurements may be tedious to perform iterative inputs over to measure UKF performance, so as an alternative a simulator which emulates the MARG measurements and has the same measurement distribution as the real-world sensor is used as input into the experimental trials when quantifying performance.

To maintain the compactness of the hardware, an ESP8266 micro-controller unit (MCU) was selected as the embedded platform for interfacing with the MEMS sensors. Memory and clock speed of the MCU are significantly lower than those such as the STM32 model of processing platforms. As the sensor readings are processed off-board, memory constraints are not of concern. The compact design allows freedom of rotation, as the ESP8266 can be placed together with the sensor on a single piece of Veroboard instead of implementing them as separate subsystems. The MCU communicates with the PC in which the UKF filter is applied. A wired RS232 UART connection is employed to transmit the collected sensor data. The wired connection unfortunately restricts freedom of motion so it is more difficult to manoeuvre, despite requiring less setup than WiFi based communication protocols that could also be used.

Python is used as the software development framework for communicating with the MCU, developing the algorithm, performing statistical analysis, and data visualisation. Due to the abundance of libraries that perform data processing operations for measurement and display, the simplicity of integrating measurement and filter processing capabilities makes it an attractive option. However, compared to more optimised programming languages, Python's operations are memory- and time-intensive, resulting in lengthy data processing times, especially when a large number of software iterations must be executed. The UKF filter implementation makes extensive use of the Numpy library due to the mathematical operations (particularly matrix operations). To visualise data distributions, the seaborn library is employed, as it offers additional data analysis and visualisation capabilities compared to matplotlib statistic functions. Matplotlib is highly customisable, as a result a lot of the elements of data visualisation is left up to the developer. Statistical instruments used include mean and standard deviation, RMSE computations, and the optimal test. Inferential statistical tests provide statistical tests to analyse performance and accuracy, which is more useful than descriptive statistics such as the mean and median. Scopus and IEEE Explorer are used to navigate the statistical sensor fusion implementations that can be researched. The sensor fusion algorithms presented are typically application-specific, and deviate from the required sensor fusion algorithm that is required to be implemented.

4.2. [GA5.2] Using appropriate engineering skills and tools

The above-mentioned tools are used as follows in developing and analysing a statistical sensor fusion algorithm for orientation estimation: The MEMS sensor is used to obtain information about the sensor's acceleration, magnetic field intensity, and angular velocity in all three Euler axes. The evaluation kit transmits these measurements at a specified sample frequency. These measurements are transmitted through a wired I^2C connection into an ESP8266 MCU. The sensor system and MCU are placed on a rotation platform that is capable of recording the orientation changes, which will be used as the truth reference orientation. By connecting a PC to the serial communication port on the MCU, the sampled measurements are sent from the MCU to the Python environment using RS232 UART.

The collected sensor data from the MARG unit was used to determine the data's mean and standard deviation. The statistics are calculated using the Numpy mean and standard deviation functions, as well as the Seaborn distplot functions for visualising the distribution of these plots. Using the provided statistics, a simulator that emulates the MARG sensors is developed. The simulator will return the anticipated measurements of the selected rotation sequence in order to assess the performance of the sensor fusion filters. Using a Numpy random number generator that generates normalised noise values, these simulated sensors simulate the measurement noise of real-world devices.

To implement the sensor fusion algorithm, orientation sequences are processed using the quaternion orientation representation model. In the Python framework, a set of methods capable of performing conversions between orientation frames and common arithmetic operations in the quaternion domain have been developed. Using function calls, these implemented methods are simply interfaced into the design to perform these operations in the orientation estimation filter calculations.

Both the research and baseline models are represented using a matrix-based system. Matrix operations from the Numpy library are subsequently utilised in these models when operations such as finding the transpose, matrix multiplication, and matrix inverses must be computed. Additionally, rows and columns are extracted from these matrices via array splicing to facilitate read and write operations. The Cholesky factors required for the UKF calculation are adapted from the Numpy package for linear algebra. The orientation estimation filters are implemented as class functions, where a single invocation of the function takes a stream of sensor readings obtained from a sequence of rotations as input and returns the orientation estimate for each sample provided. These estimates are compared with the ground value estimate. To ensure a statistically valid result, the root-mean-square error (RMSE) is used as a measure of the estimation filter's accuracy. To determine which orientation changes are more susceptible to error, it is computed for all time intervals over the entire iteration period. Another statistical approximation employs confidence intervals to determine the filter stability for orientation estimation. This ensures that the baseline filter and UKF function uniformly for all rotation operations. The Optimal test, a statistical approximation method for dynamic models, will be used to compare the performance of the UKF filter to that of the Madgwick filter in order to validate the hypothesis that argues that UKF will perform better in orientation estimation.

4.3. Research model code appendix

Listing 4.1: UKF implementation

```
import numpy as np

# sample frequency of the system
Fs = 10



def quaternion_conjugate(q):
    """find the conjugate of the quaternion by converting the vector components
    to their negatives.
    @ params : q -> quaternion vector
    returns quaternion vector """
    res = [q[0], -1 * q[1], -1 * q[2], -1 * q[3]]
    return res


def quaternion_product(q1, q2):
    """multiplies two quaternions together to generate a rotated matrix of the quaternion
    @ params : q1, q2 -> quaternion vectors to be multiplied
    returns quaternion vector """
    prod = [0.0, 0.0, 0.0, 0.0]
    prod[0] = q1[0] * q2[0] - q1[1] * q2[1] - q1[2] * q2[2] - q1[3] * q2[3]
    prod[1] = q1[0] * q2[1] + q1[1] * q2[0] - q1[2] * q2[3] + q1[3] * q2[2]
    prod[2] = q1[0] * q2[2] + q1[1] * q2[3] + q1[2] * q2[0] - q1[3] * q2[1]
    prod[3] = q1[0] * q2[3] - q1[1] * q2[2] + q1[2] * q2[1] + q1[3] * q2[0]
    return np.array(prod)


def rotate_vector_by_quaternion(q, v):
    """rotates a given vector with a specific quaternion
      @ params : q -> quaternion to perform the rotation
      @ params2 : v -> vector to be rotated
      returns rotated vector """
    v_vec = np.insert(v, 0, 0)
    res = quaternion_product(quaternion_product(q, v_vec),
```

```
        quaternion_conjugate(q))[1:]
    return res


def cholesky(A):
    """ generates the lower triangular Cholesky factors of a matrix
     @ params : co-variance vector that will be used to generate Cholesky factors
      returns cholesky factors
      """
    # Create zero matrix for L
    factors_ = [[0.0] * len(A) for i in range(len(A))]

    # Perform the Cholesky decomposition
    for i in range(len(A)):
        for k in range(i + 1):
            tmp_sum = np.sum([factors_[i][j] * factors_[k][j] for j in range(k)])
            if i == k:
                factors_[i][k] = np.sqrt(A[i][i] - tmp_sum)
            else:
                factors_[i][k] = (1.0 / factors_[k][k] * (A[i][k] - tmp_sum))
    return np.array(factors_)


def unscented_transformation(X, P, alpha=0.001, beta=2, kappa=0):
    """returns the sigma points, state and covariance weights of a given state matrix
    and its co-variance vector. The sigma points are generated using the alpha, beta and
    kappa values that are provided
    @ params : X-> state vector
    @ params2 : P -> covariance vector
    returns sigma points, state weights and covariance weights """

    # X is the state vector at time k-1, n x 1
    # P is the uncertainty matrix at time k-1, n x n
    n = X.shape[0]

    _lamda = alpha ** 2 * (n + kappa) - n
    state_weights_1 = []
    cov_weights_1 = []
```

```python
    state_weights_2 = []
    cov_weights_2 = []

    # determines the cholesky factor of the covariance array
    L = cholesky(P)
    # create an array for positive and negative sigma points
    X_sig_pos = np.zeros((L.shape[0], L.shape[1]))
    X_sig_neg = np.zeros((L.shape[0], L.shape[1]))

    # state weights
    W_1 = _lamda / (n + _lamda)
    # covariance weights
    W_2 = (_lamda / (n + _lamda)) + (1.0 - alpha ** 2.0 + beta)
    # appends the actual mean estimate
    state_weights_1.append(W_1)
    cov_weights_1.append(W_2)

    # positive weight and negative sigma and weights are added consecutively
    for i in range(0, L.shape[1]):
        X_sig_pos[:, i] = (X + np.sqrt(n + _lamda) * np.array([L[:, i]]).T)[:, 0]
        X_sig_neg[:, i] = (X - np.sqrt(n + _lamda) * np.array([L[:, i]]).T)[:, 0]

        W = 1 / (2 * (n + _lamda))

        # appends weights twice for positive and negative weights that are added
        state_weights_1.append(W)
        state_weights_2.append(W)
        cov_weights_1.append(W)
        cov_weights_2.append(W)

    all_sigma_points = X
    all_sigma_points = np.hstack([all_sigma_points, X_sig_pos, X_sig_neg])
    state_weights = np.array(state_weights_1 + state_weights_2)
    cov_weights = np.array(cov_weights_1 + cov_weights_2)
    return all_sigma_points, state_weights, cov_weights


def calculate_gyro(g, dt):
```

```
    """implementation of the gyroscope differential equation to obtain the next gyroscope estimate
      @ params : g -> 3- axis gyroscope measurements
      @ params2 : dt -> sample time of rotations
       returns time integrated vector of angular displacement values """
    gx = g[0]
    gy = g[1]
    gz = g[2]
    Identity = np.eye(4, dtype=float)
    F = Identity + (np.array([[0.0, -gx, -gy, -gz],
                              [gx, 0.0, gz, -gy],
                              [gy, -gz, 0.0, gx],
                              [gz, gy, -gx, 0.0]]) * 0.5 * dt)
    return F


def prediction(F, sigma_points, state_weights, cov_weights, Q):
    """rotates a given vector with a specific quaternion
      @ params : F -> forward projected vector
      @ params2 : sigma_points, state_weights, cov_weights -> unscented transform sigma points
       @params3 : Q -> measurement model co-variance vector
       returns the forward prediction from the gyroscope measurements and their unscented
        transform weights that are obtained"""
    n = sigma_points.shape[1]
    m = sigma_points.shape[0]
    state_cov = np.zeros((m, m), dtype=float)
    forward_project = np.zeros((m, n), dtype=float)

    # forward prediction using the sigma points
    for i in range(n):
        # multiply the projection with the relevant sigma points to obtain the estimate
        mat = np.matmul(F, np.array([sigma_points[:, i]]).T)
        forward_project[:, i] = mat[:, 0]

    # find the mean state values of the points by using the weighting provided for the points
    state_mean = np.matmul(forward_project, np.array([state_weights]).T)

    for i in range(n):
```

```python
        W = cov_weights[i]
        X_diff = np.array([forward_project[:, i]]).T - state_mean
        P_k = np.matmul(X_diff, X_diff.T)
        state_cov += (W * P_k)

    state_cov = state_cov + Q
    return state_mean, state_cov


def update_roll_pitch(X, P, a, sigma_points, state_weights, cov_weights, R):
    # normalise the acc values
    """update state for the roll and pitch using the accelerometer measurement model
        which is used to update the estimate using the sigma points obtained from the
        accelerometer estimates
          @ params : X, P -> state model and covariance vector
          @ params2 : sigma_points, state_weights, cov_weights -> unscented transform sigma points
          @ params3 : R accelerometer measurement model covariance vector
          returns updated X, P vectors """
    norm = np.sqrt (a[0] ** 2 + a[1] ** 2 + a[2] ** 2)
    ax = a[0] / norm
    ay = a[1] / norm
    az = a[2] / norm

    n = sigma_points.shape[1]
    y_obs = np.array([[np.arctan(ay / np.sqrt(ax ** 2 + az ** 2))], [np.arctan(ax / az)]])
    y_i = np.zeros((2, n), dtype=float)
    y_mean = np.zeros((2, 1), dtype=float)

    for i in range(n):
        w = sigma_points[0, i]
        x = sigma_points[1, i]
        y = sigma_points[2, i]
        z = sigma_points[3, i]

        y_i[0, i] = np.arctan((2 * (w * x + y * z)) / (1.0 - 2.0 * (x ** 2 + y ** 2)))
        y_i[1, i] = np.arcsin((2 * (w * y - z * x)) / (w ** 2 + x ** 2 + y ** 2 + z ** 2))
```

```
        y_mean = y_mean + (state_weights[i] * np.array([y_i[:, i]]).T)

    Pyy = np.zeros((2, 2), dtype=float)
    Pxy = np.zeros((X.shape[0], 2), dtype=float)
    for i in range(n):
        y_diff = np.array([y_i[:, i]]).T - y_mean
        x_diff = np.array([sigma_points[:, i]]).T - X
        Pyy += cov_weights[i] * np.matmul(y_diff, y_diff.T)
        Pxy += cov_weights[i] * np.matmul(x_diff, y_diff.T)

    Pyy += R
    Pyy_inv = np.linalg.inv(Pyy)
    K = np.matmul(Pxy, Pyy_inv)
    y_diff_obs = y_obs - y_mean
    X += np.matmul(K, y_diff_obs)
    P -= np.matmul(K, np.matmul(Pyy, K.T))

    return X, P


def update_yaw(X, P, m, sigma_points, state_weights, cov_weights, R):
    """update state for the yaw using the magnetometer measurement model
      which is used to update the estimate using the sigma points obtained from the
      magnetometer estimates
        @ params : X, P -> state model and covariance vector
        @ params2 : sigma_points, state_weights, cov_weights -> unscented transform sigma points
        @ params3 : R magnemtometer measurement model covariance vector
        returns updated X, P vectors """
    norm = np.sqrt(m[0] ** 2 + m[1] ** 2 + m[2] ** 2)
    m /= norm

    Q = [X[0, 0], X[1, 0], X[2, 0], X[3, 0]]
    mag_rot = rotate_vector_by_quaternion(Q, m)
    bx = np.sqrt(mag_rot[0] ** 2 + mag_rot[1] ** 2)
    bz = mag_rot[2]

    n = sigma_points.shape[1]
    y_obs = np.array([m]).T
```

```
    y_i = np.zeros((3, n), dtype=float)
    y_mean = np.zeros((3, 1), dtype=float)

    for i in range(n):
        qw = sigma_points[0, i]
        qx = sigma_points[1, i]
        qy = sigma_points[2, i]
        qz = sigma_points[3, i]

        h_x = np.array([[bx * (qx ** 2 + qw ** 2 - (qz **
            2 + qy ** 2)) + 2 * bz * (qx * qz + qw * qy)],
                        [2 * bx * (qx * qy + qw * qz) + 2
                            * bz * (qy * qz - qw * qx)],
                        [2 * bx * (qx * qz - qw * qy) + bz
                             * (qz ** 2 + qw ** 2 - (qy **
                            2 + qx ** 2))]])

        y_i[0, i] = h_x[0, 0]
        y_i[1, i] = h_x[1, 0]
        y_i[2, i] = h_x[2, 0]
        y_mean += state_weights[i] * np.array([y_i[:, i]])
            .T

    Pyy = np.zeros((y_i.shape[0], y_i.shape[0]), dtype=
        float)
    Pxy = np.zeros((X.shape[0], y_i.shape[0]), dtype=float
        )

    for i in range(n):
        y_diff = np.array([y_i[:, i]]).T - y_mean
        x_diff = np.array([sigma_points[:, i]]).T - X
        Pyy += cov_weights[i] * np.matmul(y_diff, y_diff.T
            )
        Pxy += cov_weights[i] * np.matmul(x_diff, y_diff.T
            )

    Pyy += R
    Pyy_inv = np.linalg.inv(Pyy)
    K = np.matmul(Pxy, Pyy_inv)
    y_diff_obs = y_obs - y_mean
    X += np.matmul(K, y_diff_obs)
    P -= np.matmul(K, np.matmul(Pyy, K.T))
    return X, P


def quaternion_to_euler_vector(q):
    """
```

```
    calls the quaternion euler function to return a vector-representation of the trasformation
    from quaternion vectors to euler vectors
    @params : quaternion vector q
    return rpy euler vector
    """
    r, p, y = quaternion_to_euler_angle(q[0], q[1], q[2], q[3])
    return [r, p, y]


def quaternion_to_euler_angle(w, x, y, z):
    """
    converts the quaternion parameters w (scalar), x, y, z (vector) to their
    euler angle representations
    @params : w (scalar), x, y, z (vector) quaternion parameters
    return rpy euler vector
    """
    num_1 = 2.0 * (w * x + y * z)
    denom_1 = 1.0 - 2.0 * (x * x + y * y)
    X = -1 * np.rad2deg(np.atan2(num_1, denom_1))

    term_asin = +2.0 * (w * y - z * x)
    term_asin = +1.0 if term_asin > +1.0 else term_asin
    term_asin = -1.0 if term_asin < -1.0 else term_asin
    Y = -1 * np.rad2deg(np.asin(term_asin))

    num_2 = +2.0 * (w * z + x * y)
    denom_2 = +1.0 - 2.0 * (y * y + z * z)
    Z = np.rad2deg(np.atan2(num_2, denom_2))

    return X, Y, Z


def normalize_quaternion(Q):
    """
     normalises the quaternion into its unit representation
     @params : w (scalar), x, y, z (vector) quaternion parameters
     return rpy euler vector
     """

    norm = np.sqrt(Q[0] ** 2 + Q[1] ** 2 + Q[2] ** 2 + Q[3] ** 2)
```

```
    Q[0] = Q[0] / norm
    Q[1] = Q[1] / norm
    Q[2] = Q[2] / norm
    Q[3] = Q[3] / norm
    return Q


def simulate_ukf(times, acc, gyro, mag):
    """
     obtains measurements from the accelerometer,
        magnetometer and gyroscope to
     produce an estimate of the orientation of the MARG
        unit from the UKF.
     @params : time (sample spaces, accelerometer [],
        magnetometer [] and gyroscope []
     return rpy euler vector
     """
    total_samples = times.shape[0]

    # standard deviations
    gyro_noise_std_x = np.deg2rad(1.0140)
    gyro_noise_std_y = np.deg2rad(0.9721)
    gyro_noise_std_z = np.deg2rad(0.9844)
    ax_std = 0.0101
    ay_std = 0.0099
    az_std = 0.0106
    mag_std = 0.9

    # variance
    ax_var = ax_std ** 2
    ay_var = ay_std ** 2
    az_var = az_std ** 2

    # Gyroscope Characteristics
    x_var = gyro_noise_std_x ** 2
    y_var = gyro_noise_std_y ** 2
    z_var = gyro_noise_std_z ** 2

    mag_var = mag_std ** 2

    X = np.array([[1], [0.0], [0.0], [0.0]])

    P = np.array([[1.0, 0.0, 0.0, 0.0],
                  [0.0, 1.0, 0., 0.0],
                  [0.0, 0.0, 1.0, 0.0],
                  [0., 0.0, 0.0, 1.0]])

```

```
    quat_ukf = []
    quat_times = []

    rpy = []

    P_vals = []
    for i in range(total_samples):
        # obtain quaternion estimates
        qw_ = X[0, 0]
        qx_ = X[1, 0]
        qy_ = X[2, 0]
        qz_ = X[3, 0]

        gyros = np.deg2rad(gyro[i])

        # compensate for bias in the MARG measurements
        gyros[0] += 0.0104
        gyros[1] += 0.0110
        gyros[2] += 0.0146

        a = acc[i]
        ax = -1 * a[0] - 0.0072
        ay = -1 * a[1] - 0.0045
        az = -1 * a[2] - 0.00029

        m = [mag[i][0] + 0.1260, mag[i][1] + 0.0431, mag[i][2] + 0.2499]

        dt = 1 / Fs

        # time step
        quat_times.append(times[i])

        # gyroscope covariance vector
        gyro_cov = np.array([[x_var, 0.0, 0.0], [0.0, y_var, 0.0], [0.0, 0.0, z_var]])

        # Create UKF
        L = 0.5 * np.array([[-qx_, -qy_, -qz_],
                            [qw_, -qz_, qy_],
                            [qz_, qw_, -qx_],
                            [-qy_, qx_, qz_]])

        # rotate covariance vector using measurement model
        Q = np.matmul(L, np.matmul(gyro_cov, L.T))

        F = calculate_gyro(gyros, dt)
```

```
        # UKF Prediction
        sigma_points, state_weights, cov_weights = unscented_transformation(X, P)
        X, P = prediction(F, sigma_points, state_weights, cov_weights, Q)

        # normalise the state vector
        X = np.array([normalize_quaternion(X.T[0])]).T

        # Calculate R for Accelerometer Update
        A = np.array([[az / (ax ** 2 + az ** 2), 0.0, -ax / (ax ** 2 + az ** 2)],
                      [-ay * ax / (ax ** 2 + ay ** 2 + az ** 2) * np.sqrt(ax ** 2 + az ** 2),
                       np.sqrt(ax ** 2 + az ** 2) / (ax ** 2 + ay ** 2 + az ** 2),
                       -ay * az / ((ax ** 2 + ay ** 2 + az ** 2) * np.sqrt(ax ** 2 + az ** 2))]])

        R_acc_var = np.array([[ax_var, 0.0, 0.0], [0.0, ay_var, 0.0], [0.0, 0.0, az_var]])

        R_acc = np.matmul(A, np.matmul(R_acc_var, A.T))

        # Calculate R for Magnetometer Update
        R_mag = [[mag_var, 0., 0.],
                 [0., mag_var, 0.],
                 [0., 0., mag_var]]

        # UKF Update
        sigma_points, state_weights, cov_weights = unscented_transformation(X, P)
        X, P = update_roll_pitch(X, P, a, sigma_points, state_weights, cov_weights, R_acc)
        sigma_points, state_weights, cov_weights = unscented_transformation(X, P)
        X, P = update_yaw(X, P, m, sigma_points, state_weights, cov_weights, R_mag)
        X = np.array([normalize_quaternion(X.T[0])]).T

        res = [X[0, 0], X[1, 0], X[2, 0], X[3, 0]]
        quat_ukf.append(res)

        res_rpy = quaternion_to_euler_vector(res)
```

```
            rpy.append(res_rpy)
            a, b, c = quaternion_to_euler_angle(np.sqrt(P[0,
                0]), np.sqrt(P[1, 1]), np.sqrt(P[2, 2]), np.
                sqrt(P[3, 3]))
            P_vals.append([a, b, c])

        return quat_times, rpy, P_vals
```

Listing 4.2: Baseline simulator interface

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import interp1d
import Magnetometer_sim as magnetometer
import Accelerometer_sim as accelerometer
import Gyroscope_sim
import baseline
import Simulate_UKF

plt.rcParams.update({'font.size': 12})
plt.rcParams.update({'font.family': "Times␣New␣Roman"})

R1 = (0, 0, 0)
R2 = (0, 30.0, 0)
R3 = (0, -30.0, 0)
R4 = (60.0, 0, 0)
R5 = (-60.0, 0, 0)
R6 = (0.0, 0, 120.0)
R7 = (0.0, 0, -120.0)
R8 = (-20.0, -45.0, 0)
R9 = (20.0, 45.0, 0)
R10 = (0, 0, 0)

Fs = 10
RMSE = True

Rotations = np.array([R1, R2, R3, R4, R5, R6, R7, R8, R9,
    R10])

# determine the truth time span
x_truth_int = [(i + 1) * 5.0 for i in range(0, len(
    Rotations))]
x_truth_int.insert(0, 0.0)

summation = 0.0
x_rotation_int = [0.0]
for i in range(0, len(Rotations)):
    summation += Rotations[i][0]
```

```
    x_rotation_int.append(summation)

summation = 0.0
y_rotation_int = [0.0]
for i in range(0, len(Rotations)):
    summation += Rotations[i][1]
    y_rotation_int.append(summation)

summation = 0.0
z_rotation_int = [0.0]
for i in range(0, len(Rotations)):
    summation += Rotations[i][2]
    z_rotation_int.append(summation)

# time plots for every gyro measurement
time_values = np.linspace(0, 5 * len(Rotations), Fs * 5 *
    len(Rotations))

# interpolation of truth function
x_interp = interp1d(x_truth_int, x_rotation_int)
y_interp = interp1d(x_truth_int, y_rotation_int)
z_interp = interp1d(x_truth_int, z_rotation_int)

x_rotation_truth = [x_interp(time_values[i]) for i in
    range(0, len(time_values))]
y_rotation_truth = [y_interp(time_values[i]) for i in
    range(0, len(time_values))]
z_rotation_truth = [z_interp(time_values[i]) for i in
    range(0, len(time_values))]

gyroscope = Gyroscope_sim.Gyroscope()


def simulate_baseline():
    roll_RMSE_iter = []
    pitch_RMSE_iter = []
    yaw_RMSE_iter = []

    if RMSE:
        for j in range(0, 100):
            gyroscope.set_rotations(Rotations)
            accelerometer.set_rotations(Rotations)
            magnetometer.set_rotations(Rotations)

            gyroscope.set_samplingfreq(Fs)
            accelerometer.set_samplingfreq(Fs)
            magnetometer.set_samplingfreq(Fs)
```

```python

    gyro_time, gyroscope_values = gyroscope.generate_gyroscope()
    magnetometer_time, magnetometer_values = magnetometer.generate_magnetometer()
    accelerometer_time, accelerometer_values = accelerometer.generate_accelerometer()

    roll_angle = []
    pitch_angle = []
    yaw_angle = []

    roll_RMSE = []
    pitch_RMSE = []
    yaw_RMSE = []

    baseline.setAngles(1.00, 0.00, 0.00, 0.00, 1.00, 0.00, 0.00, 0.00, 0.00)
    for i in range(len(time_values)):
        baseline.filterUpdate(np.deg2rad(gyroscope_values[0][i]), np.deg2rad(gyroscope_values[1][i]),
                              np.deg2rad(gyroscope_values[2][i]),
                              accelerometer_values[0][i], accelerometer_values[1][i],
                              accelerometer_values[2][i],
                              magnetometer_values[0][i], magnetometer_values[1][i], magnetometer_values[2][i])
        x, y, z = baseline.getAngles()
        roll_angle.append(x)
        pitch_angle.append(y)
        yaw_angle.append(z)

        roll_RMSE.append(np.power(x - x_rotation_truth[i], 2))
        pitch_RMSE.append(np.power(y - y_rotation_truth[i], 2))
        yaw_RMSE.append(np.power(z -
```

```
                z_rotation_truth[i], 2))

            roll_RMSE_iter.append(roll_RMSE)
            pitch_RMSE_iter.append(pitch_RMSE)
            yaw_RMSE_iter.append(yaw_RMSE)

    roll_error = np.array([np.sqrt(np.sum(np.array(roll_RMSE_iter)[:, i]) / 100) for i in range(len(time_values))])
    pitch_error = np.array(
        [np.sqrt(np.sum(np.array(pitch_RMSE_iter)[:, i]) / 100) for i in range(len(time_values))])
    yaw_error = np.array([np.sqrt(np.sum(np.array(yaw_RMSE_iter)[:, i]) / 100) for i in range(len(time_values))])

    print("std roll M", np.std(roll_error), "std roll M", np.mean(roll_error))
    print("std pitch M", np.std(pitch_error), "std pitch M", np.mean(pitch_error))
    print("std yaw M", np.std(yaw_error), "std yaw M", np.mean(yaw_error))

    time_UKF, roll_error_UKF, pitch_error_UKF, yaw_error_UKF = Simulate_UKF.simulate_RMSE()

    print("std roll U", np.std(roll_error_UKF), "std roll U", np.mean(roll_error_UKF))
    print("std pitch U", np.std(pitch_error_UKF), "std pitch U", np.mean(pitch_error_UKF))
    print("std yaw U", np.std(yaw_error_UKF), "std yaw U", np.mean(yaw_error_UKF))
    # plotting of truth function

    plt.figure(4)
    plt.subplot(3, 1, 1)
    plt.title("unscented Kalman filter RMSE over time ")
    # plt.plot(time_values, x_rotation_truth, color="black", label="ground truth")
    # plt.plot(time_values, roll_error, color="red", label="Madgwick RMSE")
    plt.plot(time_UKF, roll_error_UKF, color="red", label="UKF RMSE", linestyle="dashed")
    plt.ylabel("roll angle ( ) ")
    plt.ylim(0, 2)
```

```
    plt.legend(loc='upper right', prop={'size': 10})

    plt.subplot(3, 1, 2)
    # plt.plot(time_values, y_rotation_truth, color="black", label="ground truth")
    # plt.plot(time_values, pitch_error, color="green", label="Madgwick RMSE")
    plt.plot(time_UKF, pitch_error_UKF, color="green", label="UKF RMSE", linestyle="dashed")
    plt.ylim(0, 2)
    plt.ylabel("pitch angle ( ) ")
    plt.legend(loc='upper right', prop={'size': 10})
    #
    plt.subplot(3, 1, 3)
    # plt.plot(time_values, z_rotation_truth, color="black", label="ground truth")
    # plt.plot(time_values, yaw_error, color="purple", label="Madgwick RMSE")
    plt.plot(time_UKF, yaw_error_UKF, color="purple", label="UKF RMSE", linestyle="dashed")
    plt.xlabel("time (s)")
    plt.ylabel("yaw angle ( ) ")
    plt.ylim(0, 30)
    plt.legend(loc='upper right', prop={'size': 10})
    plt.savefig("RMSE_UKF.pdf", dpi=200)

    plt.figure(0)
    plt.subplot(3, 1, 1)
    plt.title(" Madgwick gradient descent RMSE over time")
    # plt.plot(time_values, x_rotation_truth, color="black", label="ground truth")
    plt.plot(time_values, roll_error, color="red", label="Madgwick RMSE")
    # plt.plot(time_UKF, roll_error_UKF, color="red", label="UKF RMSE", linestyle="dashed")
    plt.ylabel("roll angle ( ) ")
    plt.ylim(0, 10)
    plt.legend(loc='upper right', prop={'size': 10})

    plt.subplot(3, 1, 2)
    # plt.plot(time_values, y_rotation_truth, color="black", label="ground truth")
    plt.plot(time_values, pitch_error, color="green", label="Madgwick RMSE")
    # plt.plot(time_UKF, pitch_error_UKF, color="green", label="UKF RMSE", linestyle="dashed")
```

```
        plt.ylabel("pitch␣angle␣(   )␣")
        plt.ylim(0, 10)
        plt.legend(loc='upper␣right', prop={'size': 10})
        #
        plt.subplot(3, 1, 3)
        # plt.plot(time_values, z_rotation_truth, color="black", label="ground truth")
        plt.plot(time_values, yaw_error, color="purple", label="Madgwick␣RMSE")
        # plt.plot(time_UKF, yaw_error_UKF, color="purple", label="UKF RMSE", linestyle="dashed")
        plt.xlabel("time␣(s)")
        plt.ylabel("yaw␣angle␣(   )␣")
        plt.ylim(0, 15)
        plt.legend(loc='upper␣right', prop={'size': 10})
        plt.savefig("RMSE_Baseline.pdf", dpi=200)

    roll_angle = []
    pitch_angle = []
    yaw_angle = []

    gyroscope.set_rotations(Rotations)
    accelerometer.set_rotations(Rotations)
    magnetometer.set_rotations(Rotations)

    gyroscope.set_samplingfreq(Fs)
    accelerometer.set_samplingfreq(Fs)
    magnetometer.set_samplingfreq(Fs)

    gyro_time, gyroscope_values = gyroscope.generate_gyroscope()
    magnetometer_time, magnetometer_values = magnetometer.generate_magnetometer()
    accelerometer_time, accelerometer_values = accelerometer.generate_accelerometer()

    baseline.setAngles(1.00, 0.00, 0.00, 0.00, 1.00, 0.00, 0.00, 0.00, 0.00)
    for i in range(len(time_values)):
        baseline.filterUpdate(np.deg2rad(gyroscope_values[0][i]), np.deg2rad(gyroscope_values[1][i]),
                              np.deg2rad(gyroscope_values[2][i]),
                              accelerometer_values[0][i], accelerometer_values[1][i],
                              accelerometer_values[2][i],
```

```
                                magnetometer_values[0][i], magnetometer_values[1][i], magnetometer_values[2][i])
        x, y, z = baseline.getAngles()
        roll_angle.append(x)
        pitch_angle.append(y)
        yaw_angle.append(z)

    plt.figure(1)
    plt.subplot(3, 1, 1)
    plt.title("Madgwick gradient descent filter  orientation estimation over time")
    plt.plot(time_values, x_rotation_truth, color="black", label="ground truth")
    plt.plot(time_values, roll_angle, label="Roll angle", color="red")
    plt.legend(loc='upper right', prop={'size': 10})
    plt.ylim(-50, 65)
    plt.ylabel("roll angle (  ) ")
    plt.xlim(0, 50)

    plt.subplot(3, 1, 2)
    plt.plot(time_values, y_rotation_truth, color="black", label="ground truth")
    plt.plot(time_values, pitch_angle, label="Pitch angle", color="green")
    plt.legend(loc='upper right', prop={'size': 10})
    plt.ylim(-50, 50)
    plt.xlim(0, 50)
    plt.ylabel("pitch angle (  ) ")

    plt.subplot(3, 1, 3)
    plt.plot(time_values, z_rotation_truth, color="black", label="ground truth")
    plt.plot(time_values, yaw_angle, label="Yaw angle", color="mediumpurple")
    plt.legend(loc='upper right', prop={'size': 10})
    plt.ylim(-5, 130)
    plt.xlim(0, 50)
    plt.xlabel("time (s)")
    plt.ylabel("yaw angle (  ) ")
    plt.savefig("Baseline.pdf", dpi=200)

    Simulate_UKF.simulate_certainty()

    plt.show()
```

```python


simulate_baseline()
```

Listing 4.3: Baseline implementation

```python
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import interp1d
import Magnetometer_sim as magnetometer
import Accelerometer_sim as accelerometer
import Gyroscope_sim
import ukf

plt.rcParams.update({'font.size': 12})
plt.rcParams.update({'font.family': "Times New Roman"})

R1 = (0, 0, 0)
R2 = (0, 30.0, 0)
R3 = (0, -30.0, 0)
R4 = (60.0, 0, 0)
R5 = (-60.0, 0, 0)
R6 = (0.0, 0, 120.0)
R7 = (0.0, 0, -120.0)
R8 = (-20.0, -45.0, 0)
R9 = (20.0, 45.0, 0)
R10 = (0, 0, 0)


RMSE = False
Fs = 10

Rotations = np.array([R1, R2, R3, R4, R5, R6, R7, R8, R9,
    R10])

# determine the truth time span
x_truth_int = [(i + 1) * 5.0 for i in range(0, len(
    Rotations))]
x_truth_int.insert(0, 0.0)

summation = 0.0
x_rotation_int = [0.0]
for i in range(0, len(Rotations)):
    summation += Rotations[i][0]
    x_rotation_int.append(summation)

summation = 0.0
y_rotation_int = [0.0]
```

```python
for i in range(0, len(Rotations)):
    summation += Rotations[i][1]
    y_rotation_int.append(summation)

summation = 0.0
z_rotation_int = [0.0]
for i in range(0, len(Rotations)):
    summation += Rotations[i][2]
    z_rotation_int.append(summation)

# time plots for every gyro measurement
time_values = np.linspace(0, 5 * len(Rotations), Fs * 5 * len(Rotations))

# interpolation of truth function
x_interp = interp1d(x_truth_int, x_rotation_int)
y_interp = interp1d(x_truth_int, y_rotation_int)
z_interp = interp1d(x_truth_int, z_rotation_int)

x_rotation_truth = [x_interp(time_values[i]) for i in range(0, len(time_values))]
y_rotation_truth = [y_interp(time_values[i]) for i in range(0, len(time_values))]
z_rotation_truth = [z_interp(time_values[i]) for i in range(0, len(time_values))]

gyroscope = Gyroscope_sim.Gyroscope()


def simulate_certainty():
    gyroscope.set_rotations(Rotations)
    accelerometer.set_rotations(Rotations)
    magnetometer.set_rotations(Rotations)

    gyroscope.set_samplingfreq(Fs)
    accelerometer.set_samplingfreq(Fs)
    magnetometer.set_samplingfreq(Fs)

    gyro_time, gyroscope_values = gyroscope.generate_gyroscope()
    magnetometer_time, magnetometer_values = magnetometer.generate_magnetometer()
    accelerometer_time, accelerometer_values = accelerometer.generate_accelerometer()

    accelerometer_values_T = np.array(accelerometer_values).T
```

```
    gyroscope_values_T = np.array(gyroscope_values).T
    magnetometer_values_T = np.array(magnetometer_values).T

    quat_times, rpy_ukf, P_vals = ukf.simulate_ukf(time_values, accelerometer_values_T, gyroscope_values_T,
                                                                     magnetometer_values_
                                                                     )

    r = np.array(rpy_ukf)[:, 0]
    p = np.array(rpy_ukf)[:, 1]
    y = np.array(rpy_ukf)[:, 2]
    r_var = (np.array(P_vals)[:, 0] * 100)
    p_var = (np.array(P_vals)[:, 1] * 300)
    y_var = (np.array(P_vals)[:, 2] * 100)

    plt.figure(2)
    plt.subplot(3,1,1)
    plt.title("UKF orientation estimation over time")
    plt.plot(time_values, x_rotation_truth, color="black", label="ground truth")
    plt.plot(quat_times, r, label="Roll angle", color="red")
    plt.fill_between(quat_times, r - r_var, r + r_var,
                     color="mistyrose")
    plt.legend(loc='upper right', prop={'size': 6})
    plt.ylim(-65, 65)
    plt.ylabel("roll angle ( ) ")
    plt.xlim(0, 50)

    plt.subplot(3, 1, 2)
    plt.plot(time_values, y_rotation_truth, color="black", label="ground truth")
    plt.plot(quat_times, p, label="Pitch angle", color="green")
    plt.fill_between(quat_times, p - p_var, p + p_var,
                     color="palegreen")
    plt.legend(loc='upper right', prop={'size': 6})
    plt.ylim(-60, 60)
    plt.xlim(0, 50)
    plt.ylabel("pitch angle ( ) ")

    plt.subplot(3, 1, 3)
    plt.plot(time_values, z_rotation_truth, color="black", label="ground truth")
    plt.plot(quat_times, y, label="Yaw angle", color="
```

```
           mediumpurple")
    plt.fill_between(quat_times, y - y_var, y + y_var,
                     color="thistle")
    plt.legend(loc='lower right', prop={'size': 6})
    plt.ylim(-45, 150)
    plt.xlim(0, 50)
    plt.xlabel("time (s)")
    plt.ylabel("yaw angle (  ) ")
    plt.savefig("UKF.pdf", dpi=200)

    # plt.show()


def simulate_RMSE():
    roll_RMSE_iter = []
    pitch_RMSE_iter = []
    yaw_RMSE_iter = []
    for j in range(0, 100):
        gyroscope.set_rotations(Rotations)
        accelerometer.set_rotations(Rotations)
        magnetometer.set_rotations(Rotations)

        gyroscope.set_samplingfreq(Fs)
        accelerometer.set_samplingfreq(Fs)
        magnetometer.set_samplingfreq(Fs)

        gyro_time, gyroscope_values = gyroscope.
           generate_gyroscope()
        magnetometer_time, magnetometer_values =
           magnetometer.generate_magnetometer()
        accelerometer_time, accelerometer_values =
           accelerometer.generate_accelerometer()

        accelerometer_values_T = np.array(
           accelerometer_values).T
        gyroscope_values_T = np.array(gyroscope_values).T
        magnetometer_values_T = np.array(
           magnetometer_values).T

        quat_times, rpy_ukf, P_vals = ukf.simulate_ukf(
           time_values, accelerometer_values_T,
           gyroscope_values_T,
                                                           magnetometer_va
                                                             )

        r = np.array(rpy_ukf)[:, 0]
```

```
        p = np.array(rpy_ukf)[:, 1]
        y = np.array(rpy_ukf)[:, 2]

        # plt.figure(0)
        # plt.subplot(3, 1, 1)
        # plt.plot(quat_times, r)
        # plt.subplot(3, 1, 2)
        # plt.plot(quat_times, p)
        # plt.subplot(3, 1, 3)
        # plt.plot(quat_times, y)

        roll_RMSE_iter.append(np.power(r - np.array(
            x_rotation_truth), 2))
        pitch_RMSE_iter.append(np.power(p - np.array(
            y_rotation_truth), 2))
        yaw_RMSE_iter.append(np.power(y - np.array(
            z_rotation_truth), 2))

    roll_error = np.array([np.sqrt(np.sum(np.array(
        roll_RMSE_iter)[:, i])/100) for i in range(len(
        time_values))])
    pitch_error = np.array([np.sqrt(np.sum(np.array(
        pitch_RMSE_iter)[:, i])/100) for i in range(len(
        time_values))])
    yaw_error = np.array([np.sqrt(np.sum(np.array(
        yaw_RMSE_iter)[:, i])/100) for i in range(len(
        time_values))])

    return time_values, roll_error, pitch_error, yaw_error
```

Bibliography

[1] S. Madgwick *et al.*, “An efficient orientation filter for inertial and inertial/magnetic sensor arrays,” *Report X-IO and Univ. of Bristol (UK)*, vol. 25, pp. 113–118, 2010.

[2] E. Kraft, “A quaternion-based unscented Kalman filter for orientation tracking,” in *Sixth Int. Conf. of Inf. Fusion, 2003.*, vol. 1, 2003, pp. 47–54.

[3] J. Han, Q. Song, and Y. Q. He, “Adaptive Unscented Kalman Filter and Its Applications in Nonlinear Control,” 04 2009.

[4] S. H. Pourtakdoust and H. Ghanbarpour Asl, “An adaptive unscented Kalman filter for quaternion-based orientation estimation in low-cost AHRS,” *Aircr. Eng. and Aerosp. Tech.*, vol. 79, no. 5, pp. 485–493, 2007.

[5] Z. Zhang, Z. Zhou, S. Du, C. Xiang, and C. Kuang, “Unscented Kalman Filter Based Attitude Estimation with MARG Sensors,” in *Chin. Satell. Navigation Conf. (CSNC) 2019 Proceedings*, vol. 2, 05 2019, pp. 490–502.