Homework 5: Inertial Measurement Units and Sensor Fusion

EE267 Virtual Reality 2025

Due: 05/08/2025, 11:59pm

Instruction

Students should use the Arduino environment and JavaScript for this assignment, building on top of the provided starter code found on the course website. We recommend using the Chrome browser for debugging purposes (using the console and built-in debugger). Make sure hardware acceleration is turned on in the advanced settings in Chrome. Other browsers might work too, but will not be supported by the teaching staff in labs, piazza, and office hours.

The theoretical part of this homework is to be done individually, while the programming part can be worked on in groups of up to two. If you work in a group, make sure to acknowledge your team member when submitting on Gradescope. You can change your teams from assignment to assignment.

Homeworks are to be submitted on Gradescope (sign-up code: **D3PYB3**). You will be asked to submit both a PDF containing all of your answers, plots, and insights in a **single PDF** as well as a zip of your code (more on this later). The code can be submitted as a group on Gradescope, but each student must submit their own PDF. Submit the PDF to the Gradescope submission titled *Homework 5: Inertial Measurement Units and Sensor Fusion* and the zip of the code to *Homework 5: Code Submission*. For grading purposes, we include placeholders for the coding questions in the PDF submission; select any page of the submission for these.

When zipping your code, make sure to zip up the directory for the current homework. For example, if zipping up your code for Homework 1, find `render.html`, go up one directory level, and then zip up the entire `homework1` directory. In addition, for this homework, **delete the `node_modules` folder before zipping** the directory, which will make the upload go more smoothly on Gradescope. Your submission should only have the files that were provided to you. Do not modify the names or locations of the files in the directory in any way besides the deletion of the `node_modules` folder.

For this week's topic, course notes are available on the course website. These notes go into more detail for the mathematical concepts discussed in class. Please review the lecture slides and read the course notes before you start with this homework.

Please complete this week's lab and watch the video before you start to work on the programming part of this homework.

1 Theoretical Part

1.1 Rotation Quaternions (*5pts*)

Given rotation angle θ and normalized rotation axis $\mathbf{v} = \begin{pmatrix} v_x, & v_y, & v_z \end{pmatrix}^T$, i.e., $\|\mathbf{v}\| = 1$, show that the corresponding rotation quaternion always has unit length.

1.2 3-D Gyro Integration (*18pts*)

Using a gyro that samples at a rate of $f = 1$ Hz, we have recorded the following four angular velocity measurements $\boldsymbol{\omega}^{(i)} = \begin{pmatrix} \omega_x^{(i)}, & \omega_y^{(i)}, & \omega_z^{(i)} \end{pmatrix}^T$ in *sensor coordinates* at successive time steps $i = 1, 2, 3, 4$:

$$\boldsymbol{\omega}^{(1)} = \begin{pmatrix} \frac{\pi}{2} \\ 0 \\ 0 \end{pmatrix}, \quad \boldsymbol{\omega}^{(2)} = \begin{pmatrix} 0 \\ 0 \\ -\frac{\pi}{2} \end{pmatrix}, \quad \boldsymbol{\omega}^{(3)} = \begin{pmatrix} 0 \\ -\frac{\pi}{2} \\ 0 \end{pmatrix}, \quad \boldsymbol{\omega}^{(4)} = \begin{pmatrix} 0 \\ 0 \\ \frac{\pi}{2} \end{pmatrix}, \quad \text{(unit: } rad/s\text{)}.$$

(i) For each time step, report the rotation axis and amount of rotation since the previous time step (in degrees) in the following format: rotation around x-axis by $-10°$. Assume that the angular velocity was constant since the previous measurement. (*3pts*)

(ii) What should the orientation of the gyro be after this sequence of rotations with respect to its starting orientation? Feel free to apply these rotations to your hand or another object to determine the final orientation. (*5pts*)

(iii) In class, we discussed how we can use the Taylor expansion to derive a simple integration scheme that keeps track of a single orientation angle in flatland. For this integration scheme, we need an initial condition (i.e., initial orientation $\boldsymbol{\theta}^{(0)}$) and the length of the time step $\Delta t = 1/f$.

Let's try to apply this method to 3D orientation tracking by integrating each of the three angles as

$$\begin{pmatrix} \theta_x \\ \theta_y \\ \theta_z \end{pmatrix}^{(t+\Delta t)} \approx \begin{pmatrix} \theta_x \\ \theta_y \\ \theta_z \end{pmatrix}^{(t)} + \Delta t \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix}^{(t+\Delta t)},$$

where $\theta_x^{(t)}$, $\theta_y^{(t)}$, and $\theta_z^{(t)}$ are the estimated pitch, yaw, and roll angles at time t, and $\omega_x^{(t)}$, $\omega_y^{(t)}$, and $\omega_z^{(t)}$ are the angular velocity measurements at time t. Assuming that the initial condition is $\boldsymbol{\theta}^{(0)} = (0,\ 0,\ 0)^T$, calculate and report the three integrated angles at all four time points $\boldsymbol{\theta}^{(t)}$ using the formula above.

Now, the question is if these integrated angles are the Euler angles. To verify this, construct 3×3 rotation matrices $R^{(t)}$ from each of the integrated angles $\boldsymbol{\theta}^{(t)}$ by assuming that $(\theta_x, \theta_y, \theta_z)$ are pitch, yaw, and roll angles. For constructing the rotation matrix, you can assume that the rotation order is the yaw-pitch-roll order discussed in class, i.e. $R = R_z(-\theta_z)R_x(-\theta_x)R_y(-\theta_y)$.

Does $R^{(4)}$ match your expectation from (ii)? (*5pts*)

(iv) If $R^{(4)}$ in (iii) does not match your expectation, briefly explain what went wrong with the proposed 3D integration scheme. (*5pts*)

Programming Part PDF Deliverables

The following questions in the programming part ask you to provide written responses:

- 2.1.1 Bias Estimation (*3pts*)
- 2.1.2 Noise Variance Estimation (*3pts*)
- 2.2.4 Flatland Orientation Tracking Comparison (*1pts*)
- 2.4.4 Quaternion-Based Orientation Tracking Algorithm Comparison (*5pts*)
- 2.5.3 Head & Neck Model Discussion (*2pts*)

Make sure to append your responses to the end of the PDF submission.

2 Programming Part

All of the following tasks, except for Section 2.5, should be compiled for and tested with your VRduino using Arduino programming. Detailed documentation for each function can be found in the header (`.h`) files. Before implementing a function, read the documentation in addition to the question writeup.

A brief overview of the code:

1. `vrduino.ino`: This file initializes all the variables and calls the tracking functions during each loop iteration. It also handles all serial input and output.
2. `OrientationTracker.cpp`: This class manages the orientation tracking. It contains functions to implement the IMU sampling, and the variables needed to perform orientation tracking.
3. `Quaternion.h, OrientationMath.h`: These files implement most of the math related to quaternion and orientation tracking.

Before starting the programming part, please check that the IMU is functioning correctly. After uploading the starter program to the Teensy, open the serial monitor and input `3` and `4` to check the sampled gyroscope and accelerometer values. You should expect to see a stream of varying numbers populating the serial monitor. If you see the same number (most likely 0.0) being printed over and over, check that the Teensy is properly mounted onto the VRduino. If you cannot resolve this, contact the course staff as you may have a faulty VRduino.

2.1 Noise and Bias Estimation (*10pts*)

How great it would be if the measurements we get from the IMU were perfect! Unfortunately, this is not the case in practice and, like most real-world sensors, all IMU sensors are subject to noise and potentially also bias. In this task, we will calibrate both measurement noise and bias for all three axes of the gyro and accelerometer.

Note that the bias terms may change in different environments (possibly even if the IMU is just turned on/off) due to changes in temperature, mechanical stress on the system, or other factors. Here, you will only calibrate these values once and ignore possibly changing values.

The answers you obtain from solving this question are critical for your orientation tracking algorithms and also simple, so this question will help you get started programming the VRduino.

2.1.1 Bias Estimation (*3pts*)

Let's start by estimating the bias terms of the gyroscope and accelerometer. To calculate bias, we simply average a large number of samples from each of the sensors **while the VRduino is perfectly still** (e.g., put it on a table and don't touch it while taking the measurements). Specifically, in `measureImuBiasVariance()` of `OrientationTracker.cpp`, compute the mean of 1000 consecutive x, y, z gyroscope and accelerometer measurements. Store the bias values in the `gyrBias` and `accBias` arrays defined in the `OrientationTracker` class. Remember that in C++, you have to be careful with types (`float` or `int`) when performing any arithmetic operations.

To print these variables, set `streamMode = INFO` in `vrduino.ino`. Then, open the Serial Monitor (`Tools > Serial Monitor`) in the Arduino IDE and in the bottom right corner of the Serial Monitor, set the baud rate to 115200 and line ending to `No line ending`. Make sure to update the Serial Port to the Teensy's port (`Tools > Serial Port`). You can recalculate the bias at any time by sending the 'b' character to the Teensy.

Report all 6 calibrated bias values in your PDF writeup! Briefly comment on what values one would expect to see for the gyro and accelerometer if these sensors were perfect.

Note: To avoid having to calibrate this every time, you can set the `measureImuBias` variable to `false`. Set the variable `gyrBiasSet` to the values you measured above. The bias value can change due to temperature, so you may need to recalibrate every so often for best performance.

2.1.2 Noise Variance Estimation (*3pts*)

Now that the bias term is estimated, also calibrate the noise variance for the IMU. As before, we simply want to observe a large number of samples from the sensors **without moving it** and get an estimate of the variance in the data, which corresponds to the noise inherent to these sensors. Modify the `measureImuBiasVariance()` function to compute the noise variance for the gyro and accelerometer. You should only need to take 1000 measurements to compute the bias and variance. Store these values in `gyrVariance` and `accVariance`. Again, the values can be printed to the serial port and observed with the serial monitor of the Arduino IDE.

Report the noise variance values in your PDF writeup!

2.1.3 IMU Sampling and Bias Subtraction (*4pts*)

Implement `updateImuVariables()` in `OrientationTracker.cpp`, which is called at every loop iteration. In the starter code, we store the raw accelerometer and gyroscope values in the variables `acc` and `gyr`, which are going to be used for orientation tracking in the following questions.

First, to compensate for the bias computed in the previous question, you need to modify the variable `gyr` such that it equals the gyroscope value **minus the estimated bias**. Also update the timing variables, `deltaT` and `previousTimeImu`. Call `micros()` to get the current time; pay special attention to the units and data types that these variables and functions use. You should only call `micros()` once in your function.

2.2 Orientation Tracking in Flatland (*10pts*)

For starters, we will track the orientation of the VRduino in 1D, i.e., in *flatland*. For this purpose, we will track the rotation angle of the VRduino around its z axis, so that we are estimating the angle between the global y axis and the local y axis of the VRduino, i.e., the *roll* angle of the device. In this particular case, we can represent the orientation of the VRduino by a single angle θ. Please review section 3 of the course notes on "3-DOF Orientation Tracking with IMUs" for further details.

Throughout this task, you need to hold the VRduino as shown in the video and rotate only around the z-axis. You do not need to use JavaScript for this task. The values calculated by the following subtasks can be visualized using the serial plotter of the Arduino IDE as described at the end of this section.

2.2.1 Gyro-only Orientation Tracking (*3pts*)

Your first tracking task is simple: implement the simple forward Euler integration scheme we discussed in class when only a single gyro measurement is available at each time step. Implement `computeFlatlandRollGyr()` in `OrientationMath.cpp`. You should only need to use the z element in the gyro measurement.

Then in `updateOrientation()` of `OrientationTracker.cpp`, call this function and update the `flatlandRollGyr` variable. The inputs to the function should only come from the class variables you updated in the previous step.

2.2.2 Accelerometer-only Orientation Tracking (*3pts*)

Now let's use only the accelerometer values for tracking orientation. Implement `computeFlatlandRollAcc()` in `OrientationMath.cpp`. You only need to use the x and y components of the measurement. For the accelerometer, we do not need to integrate anything. We will compute the orientation angle θ (in degrees) directly from the two accelerometer measurements as discussed in class and in the course notes. Call this function in

`updateOrientation()` and update the `flatlandRollAcc` variable.

2.2.3 Flatland Orientation Tracking with a Complementary Filter (*3pts*)

Now that you have implemented both the gyro-only and the accelerometer-only version of flatland orientation tracking, let's go ahead and fuse them using a complementary filter. Applying this filter should remove the noise of the accelerometer and prevent the drift of the gyro.

Note that the complementary filter combines the integrated gyro angle with the angle estimated by the accelerometer. The proper way to integrate the angle in this case is to add the current gyro measurements, weighted by the time step, to the complementary-filtered angle of the last time step and not to the gyro-only angle from the last time step. Therefore, $\theta^{(t-1)}$ in Equation 9 of the course notes should be the angle estimated by the complementary filter at the previous time step.

Implement `computeFlatlandRollComp()` in `OrientationMath.cpp`. Call it in `updateOrientation()` in `OrientationTracker.cpp` and update the `flatlandRollComp` variable.

To plot the data, set `streamMode = FLAT` in `vrduino.ino`. You can also change the stream mode by sending '1' to the Teensy with the Serial Monitor. Then, open `Tools > Serial Plotter` to plot all the angles at each time step. This overlays the time-varying plots of the angle estimated from only the gyro in red, the angle estimated from only the accelerometer in green, and the angle estimated by the complementary filter in yellow. The color might shift depending on Arduino versions, but the order of the legend should be the gyro, the accelerometer and the complementary filter.

2.2.4 Flatland Orientation Tracking Comparison (*1pts*)

Take a screenshot of the above methods in action. Include it in your PDF submission along with a brief discussion of the differences you see between the three methods. Also vary the blending parameter `alphaImuFilter` in `vrduino.ino` and briefly comment on its effect in your PDF writeup.

2.3 Quaternion Algebra (*10pts*)

While Euler angles are intuitive, they cannot be used for orientation tracking in 3D. We need quaternions for that. To use quaternions on the VRduino, we will need to implement some of the basic quaternion operations discussed in class like multiplication, inversion, rotation, etc. We already implemented some of these functions for you in `Quaternion.h` and we also included a few simple unit tests in `TestOrientation.cpp` to help you verify your implementation. To run the tests, set `test = true` in `vrduino.ino`. You will need these functions to work properly for the next parts of this homework, so make sure to test them thoroughly. You can add your own tests to `TestOrientation.cpp`.

(i) Implement `setFromAngleAxis()`. This function should create a quaternion from a rotation axis and an angle given in degrees. (*2pts*)

(ii) Implement `length()`. This function should return the length of the quaternion. (*1pts*)

(iii) Implement `normalize()`. This function should return the normalized quaternion. (*1pts*)

(iv) Implement `inverse()`. This function should return the inverted quaternion. (*2pts*)

(v) Implement `multiply()`. This function should return the product of two given quaternions. (*2pts*)

(vi) Implement `rotate()`. This function should rotate the quaternion. (*2pts*)

2.4 3D Orientation Tracking with Quaternions (*35pts*)

Now that our quaternion library is set up, we can implement quaternion-based orientation tracking in 3D. Visualizing 3D orientations in the serial monitor is challenging so we built a 3D orientation visualizer for you which renders a digital twin of your VRduino. Your implementations in the next sections will cause the digital VRduinos to rotate as the physical one does. To run it, open `visualizer/visualize.html` from the homework directory, and select "3D Orientation Mode.'"

To stream data from the IMU to the browser you will need to setup Node and run `node server.js`. For instructions on how to install Node, please refer to the lab writeup. You should see values streaming out depending on the stream mode. Send '1' to change it to "FLAT" mode. You should see the boards "rolling" in the visualizer!

As you debug your code, you may want to reset the tracking. Sending 'r' resets all orientation estimates to 0.

2.4.1 Gyro-only Orientation Tracking (*5pts*)

Implement `updateQuaternionGyr()` in `OrientationMath.cpp`. This implements quaternion-based orientation tracking using only the gyro integration model, as discussed in class and in the course notes. Be careful with the units – make sure you always know what units you are working with (e.g., degrees or radians) for each task. Also watch out for division by zero. For this purpose, check the magnitude of the 3 gyro measurements and if any are below some threshold, e.g. `1e-8`, ignore that measurement. In addition, always normalize your quaternions to ensure that they represent valid rotations.

Call this function in `updateOrientation()` in `OrientationTracker.cpp` to update the `quaternionGyr` variable. Again, the input arguments should only come from the class variables.

Use the visualizer to see if your implementation behaves as expected. Set `streamMode = THREED` in `vrduino.ino`, or press '2' in the shell running `server.js`.

2.4.2 Tracking Tilt with the Accelerometer (*5pts*)

Now, let's use only the accelerometer to track the tilt angles, i.e., pitch and roll. Implement `computeAccPitch()` and `computeAccRoll()` in `OrientationMath.cpp`. We cannot track yaw this way, but that may be okay for certain applications.

We will track this in Euler angles and compare it with the quaternion-based tracking.

Call the functions in `updateOrientation()` and update the `eulerAcc` variable.

2.4.3 Quaternion-based Orientation Tracking with the Complementary Filter (*20pts*)

Implement `updateQuaternionComp()` in `OrientationMath.cpp`. This implements a complementary filter with quaternions. Make sure you review the lecture material and Section 5 of the course notes. Here's is a list of steps you should follow in each time step:

(i) Query all 3 gyro sensors and calculate the rotation update or instantaneous rotation q_Δ from these measurements. Prevent division by zero! For this purpose, check the magnitude of the 3 gyro measurements and if it is below some threshold, e.g. `1e-8`, then do not try to normalize gyro measurements by that magnitude. (*2pts*)

(ii) Update the complementary filter quaternion estimated in the previous time step by q_Δ using quaternion multiplication. (*2pts*)

(iii) Query the accelerometer and create a vector quaternion from these measurements. (*2pts*)

(iv) This vector quaternion is given in sensor coordinates, so rotate it from the sensor frame into the inertial frame using the current estimate of the rotation quaternion of the complementary filter (output of (ii)). Normalize

the resulting rotated vector quaternion. (*3pts*)

(v) Compute the angle ϕ between the rotated, normalized accelerometer vector quaternion (output of (iv)) and the y axis (0, 1, 0) using the dot product. (*4pts*)

(vi) Compute the rotation axis **n** between the rotated, normalized accelerometer vector quaternion (output of (iv)) and the y axis (0, 1, 0) using the cross product. Normalize this axis. (*3pts*)

(vii) Implement the quaternion-based complementary filter by multiplying the tilt correction quaternion (using ϕ, **n**, and the blending weight α) and the current estimate of the rotation quaternion of the complementary filter (output of (ii)). Normalize the resulting vector quaternion. (*4pts*)

Call this function in `updateOrientation()` in `OrientationTracker.cpp` to update the `quaternionComp` variable.

2.4.4 Quaternion-Based Orientation Tracking Algorithm Comparison (*5pts*)

Run the visualizer to compare the 3 algorithms above by setting `streamMode = THREED` in `vrduino.ino`, or sending '2' to the Teensy. Compare the following cases in your writeup:

1. Quaternions with gyro only vs quaternions with complementary filtering (α = 0.9).
2. Euler angles with accelerometer only vs quaternions with complementary filtering (α = 0).

You might find using the pre-recorded IMU data useful here when comparing the different modes. Setting `simulateImu = true` in `vrduino.ino` will continuously stream pre-recorded IMU data instead of live data.

2.5 Head orientation and the Head & Neck Model (*12pts*)

Now that we can accurately estimate the IMU's orientation, we can track the user's head orientation in VR and use that as a natural way to control the virtual camera such that we can look around in a 3D scene. As shown in the video, mount the VRduino to the front of the HMD using the double-sided tape provided for you in the lab space. *Do not use duct tape or any other tape that leaves permanent marks on the View-Master!*[1] By the end of this task you will have implemented a rudimentary version of the Google Cardboard or Samsung GearVR all by yourself!

With the IMU orientation accessible in JavaScript, implement the view matrix updates in `transform.js` which will be run upon opening `render.html`. Similar to the way we used the visualizer discussed above, you need to run the WebSocket server to access IMU orientation data from the serial port. For this purpose, you need to stream 3D quaternion data from complementary filtering only, so either set the `streamMode` variable to `QC` in `vrduino.ino` or enter `5` into the terminal running the WebSocket server with Node.js. Additionally, you can stop printing/displaying the data in the terminal by entering `d` into the terminal, which improves the latency between the orientation tracking and the rendering. The most recent IMU orientation data is stored in `imuQuaternion` of the `state` object within `stateController.js`.

2.5.1 Head Orientation (*5pts*)

In `computeViewTransformFromQuatertion()` in `transform.js`, make updates to the view matrix based on the IMU's orientation to make the viewing content update properly with head rotation. Compute the rotation matrix for the view matrix directly from the streamed quaternion. You might find the THREE function `Matrix4().makeRotationFromQuaternion()` useful. We have already implemented the IPD translation and world-to-view-space translation for you. You just need to compute the rotation matrix and insert it at the correct location. If the teapots don't appear in front of you at first, try reseting the 3D orientation by entering the 'r' character

[1] If we find permanent tape marks on the ViewMaster when you return it, you will have to replace it with a new unit.

into the terminal running Node.

2.5.2 Head & Neck Model (*5pts*)

In `computeViewTransformFromQuatertionWithHeadNeckmodel()`, incorporate the kinematic constraints on head motion with the head and neck model discussed in class. Use the neck and head length parameters found in `displayParameters.js`. You can toggle the head and neck model on and off by pressing `h` on your keyboard with the window focus to the browser (not in the terminal).

2.5.3 Head & Neck Model Discussion (*2pts*)

Compare the visual experience of the orientation tracking with and without the head and neck model. Does the motion parallax created by this model improve the experience? Briefly comment on the effectiveness of the head and neck model in your writeup.

Questions?

First, Google it! It is a good habit to use the Internet to answer your question. For 99% of all your questions, the answer is easier found online than asking us. If you cannot figure it out this way, post on piazza or come to office hours.