Course Notes: 3-DOF Orientation Tracking with IMUs
CSE 490V: Virtual Reality Systems

Gordon Wetzstein
gordon.wetzstein@stanford.edu

This document serves as a supplement to the material discussed in Lectures 9 and 10. This is not meant to be a comprehensive review of orientation tracking for virtual reality applications, but rather an intuitive introduction to inertial measurement units (IMUs) and the mathematical concepts of orientation tracking with IMUs. While we will use the "VRduino" throughout this document as an experimental platform for practical and low-cost orientation tracking, the underlying concepts apply equally to the Arduino-based orientation tracking in CSE 490V.

1 Overview of Inertial Measurement Units

Modern inertial sensors are microelectromechanical systems (MEMS) that can be mass-manufactured at low cost. Most phones, VR headsets, controllers, and other input devices include several inertial sensors.

The three types of inertial sensors we discuss are gyros(copes), accelerometers, and magnetometers. Usually, several of these sensors are integrated into the same package – the inertial measurement unit (IMU). Common types of IMUs include 3-axis gyros, 3-axis accelerometers, and 3-axis magnetometers. All of these devices combine 3 sensors of the same kind oriented orthogonal to each other. Such a 3-DOF (degree of freedom) device allows us to record reliable measurements in 3D space. Oftentimes, gyros and accelerometers can be found in the same package: a 6-DOF IMU. Similarly, a 9-DOF IMU combines a 3-axis gyro, a 3-axis accelerometer, and a 3-axis magnetometer.

Please do not confuse the degrees of freedom of the IMU with those of the actual tracking output. For example, a 9-DOF IMU is still only capable of performing 3-DOF tracking, i.e. tracking of all three rotation angles. For 6-DOF pose tracking, where we get the three rotation angles as well as three-dimensional position of an object, we typically need even more sensors.

A gyro measures angular velocity in radians per second. A common model for the measurements taken by a single gyro is

$$\widetilde{\omega} = \omega + b + \eta_{gyro}, \qquad \eta_{gyro} \sim \mathcal{N}\left(0, \sigma^2_{gyro}\right), \tag{1}$$

where $\widetilde{\omega}$ is the measurement, ω is the ground truth angular velocity, b is a bias, and η_{gyro} is i.i.d. Gaussian noise with variance σ^2_{gyro}. For short-term operation, the bias can be modeled as a constant and estimated via calibration, but it may slowly vary over time. Gyro bias is affected by temperature and other factors.

An accelerometer measures linear acceleration in m/s^2. Linear acceleration is a combination of gravity $a^{(g)}$ and other external forces $a^{(l)}$, such as object motion, that act on the sensor:

$$\widetilde{a} = a^{(g)} + a^{(l)} + \eta_{acc}, \qquad \eta_{acc} \sim \mathcal{N}\left(0, \sigma^2_{acc}\right) \tag{2}$$

Accelerometers are also affected by measurement noise η_{acc}.

When combining 3 sensors in the same package, they may end up not being mounted perfectly orthogonal to each other, in which case the gyro measurements could be modeled as

$$\begin{pmatrix} \widetilde{\omega}_x \\ \widetilde{\omega}_y \\ \widetilde{\omega}_z \end{pmatrix} = \underbrace{\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}}_{\mathbf{M}_{gyro}} \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} + \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} + \eta_{gyro} \tag{3}$$

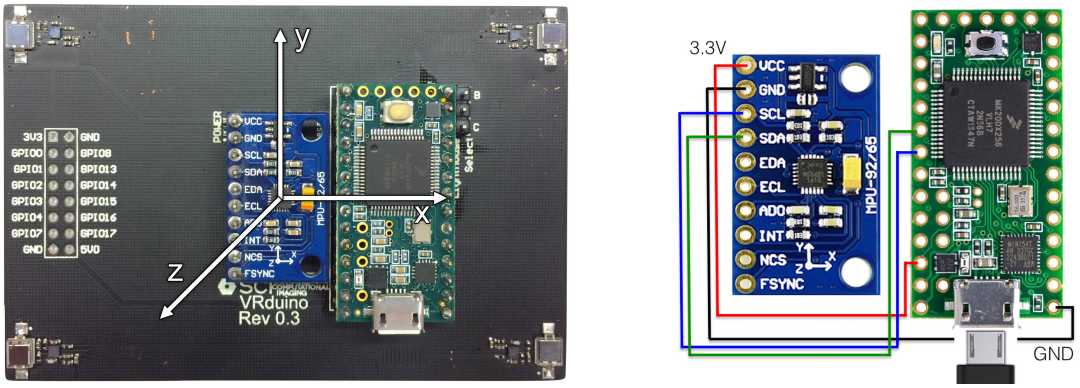

Figure 1: *The VRduino is an open source platform for tracking with "do-it-yourself (DIY)" VR systems. Left: photograph of a VRduino showing the Teensy 3.2 microcontroller, the InvenSense MPU-9250 IMU, and other components of the VRduino, including exposed Teensy pins on the left and photodiodes in the corners. The parts that are important for orientation tracking are the IMU and the Teensy. For the purpose of this course, we define the local coordinate system of the VRduino to be a right-handed frame, centered at the IMU, with the z-axis coming out of the VRduino. Right: for the Teensy to read data from the IMU, we connect power (3.3 V) and ground as well as the data (SDA) and clock (SCL) channels of the I2C protocol.*

and the accelerometers as

$$\begin{pmatrix} \widetilde{a}_x \\ \widetilde{a}_y \\ \widetilde{a}_z \end{pmatrix} = \underbrace{\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}}_{\mathbf{M}_{acc}} \begin{pmatrix} a_x^{(g)} + a_x^{(l)} \\ a_y^{(g)} + a_y^{(l)} \\ a_z^{(g)} + a_z^{(l)} \end{pmatrix} + \eta_{acc} \tag{4}$$

The mixing matrices $\mathbf{M}_{gyro}$ and $\mathbf{M}_{acc}$ model the weighted contributions of the orthogonal axes that we would like to measure, even when the physical sensors are not mounted perfectly orthogonal. For the purpose of this class, we assume that the sensors are orthogonal and we will thus model all inertial sensors as directly measuring the respective quantity along either the x, y, or z axis, i.e. $\mathbf{M}_{gyro} = \mathbf{M}_{acc} = \mathbf{I}$.

Finally, magnetometers measure the magnetic field strength along along one dimension in units of Gauss or μT. Magnetometers are quite sensitive to distortions of the magnetic field around them, which could be caused by computers or other electronic components close by. Therefore, per-device, per-environment calibration of these sensors may be required. For the remainder of this document and in most of this course, we will not use magnetometers even though our VRduino platform includes a 3-axis magnetometer. Feel free to explore this sensor in more detail in your course project.

2 The VRduino

The VRduino is an open source platform that allows us to experiment with orientation tracking and pose tracking. The VRduino was designed by Keenan Molner for the Spring 2017 offering of Stanford EE 267.

Two components of the VRduino are important for orientation tracking. The first is a 9-DOF IMU (InvenSense MPU-9250), which includes a 3-axis gyro, a 3-axis accelerometer, and a 3-axis magnetometer[1]. The second is an Arduino-style microcontroller that reads the IMU data, implements sensor fusion, and the streams out the estimated orientation via serial over USB. We use a Teensy 3.2 microcontroller because it is fully compatible with the Arduino IDE and therefore low-cost, easy to program, and supported by a large community online. Any other Arduino would

[1]Note that the gyro and accelerometer are mounted on one die and the magnetometer (Asahi Kasei Microdevices AK8963) on a different die. In the datasheet, InvenSense calls the magnetometer a "3rd party device", even though they are all contained in the same package.

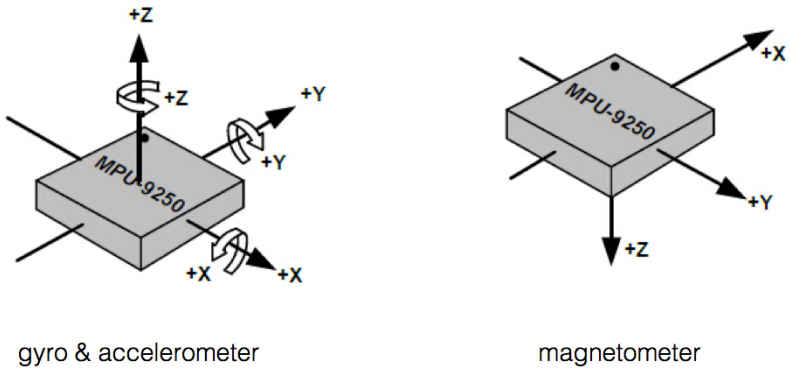

Figure 2: *Left: the gyro and accelerometer of the MPU-9250 use the same right-handed coordinate system as the VRduino. Right: the magnetometer uses a different coordinate system, which can be converted to the one we are working with by flipping the x and y coordinates and inverting the sign of the z coordinate. This is already implemented in the provided starter code.*

be equally useful for this task[2]. The Arduino and IMU communicate via the I2C protocol, which only requires 2 lines in addition to power and ground. The Arduino is powered with USB and an external computer can read data from the Arduino via serial over USB. The wiring diagram is shown in Figure 1.

Each of the inertial sensors is internally digitized by a 16 bit analog-to-digital converter (ADC). So the Arduino has access to 9 raw sensor values, each reporting signed integer values between -2^{15} and $2^{15} - 1$. To be able to trade precision for range of raw sensor values, each sensor can be configured with the following range of values:

- gyro: $\pm 250, \pm 500, \pm 1000$, or $\pm 2000\,^\circ/s$
- accelerometer: $\pm 2, \pm 4, \pm 8$, or $\pm 16\,g$
- magnetometer: $\pm 4800\,\mu T$

A larger range results in lower precision and vice versa. When a value exceeds the configured range, it will be clipped. What is important to note is that the gyros actually report values in degrees per second and the accelerometer uses the unit g, with $1g = 9.81 m/s^2$. Keeping tracking of which physical units you are working with at any time is crucial.

A raw sensor value is converted to the corresponding value in metric units as

$$metric\_value = \frac{raw\_sensor\_value}{2^{15} - 1} \cdot max\_range \tag{5}$$

In the MPU-9250 datasheet, the maximum sampling rate of the gyroscope is listed as 8000 Hz, that of the accelerometer as 4000 Hz, and that of the magnetometer as 8 Hz.

Example code to initialize the communication between Arduino and IMU and to read raw values will be provided in the lab. Note that the coordinate systems used by gyro & accelerometer and magnetometer are slightly different. Whereas gyro and accelerometer use a right-handed coordinate system similar to the VRduino, the magnetometer uses a slight different coordinate system. The latter can easily be converted to our right-handed coordinate system as outlined in Figure 2.

[2]The 2016 offering at Stanford used an Arduino Metro Mini, which was also great and a little bit cheaper than the Teensy 3.2. However, at 48 MHz the Teensy is very fast while being affordable, which turned out to be crucial for pose tracking.

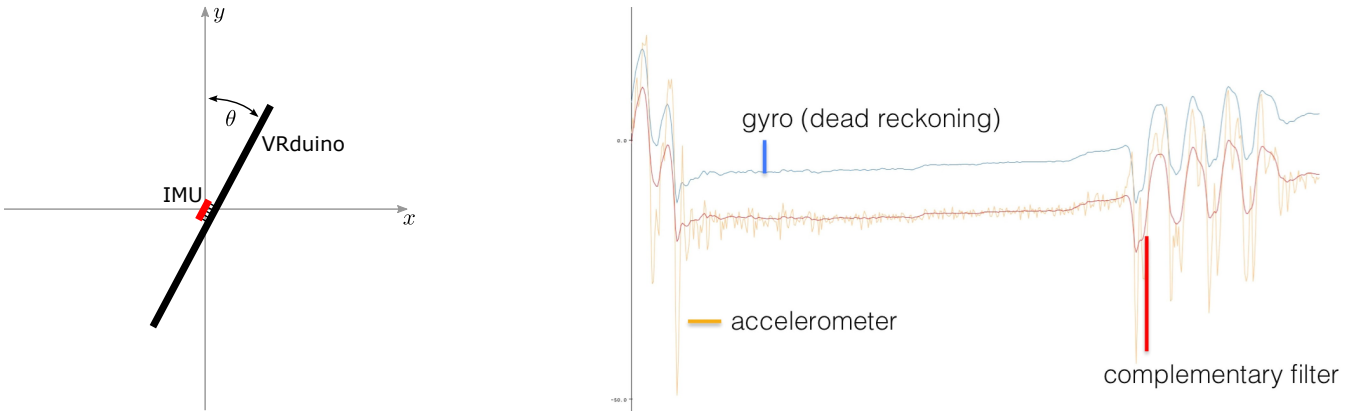

Figure 3: *Left: illustration of the "flatland" problem. The IMU on the VRduino rotates around the origin of a 2D coordinate system. The goal of orientation tracking in this case is to estimate θ. Right: plots showing roll angle estimated with inertial sensors on the VRduino. The blue line shows integrated gyro measurements, which are smooth but drift away from the other estimates over time. The accelerometer (orange) results in globally correct but locally noisy angle estimates. The complementary filter applies a high-pass filter on the gyro and a low-pass filter on the accelerometer before fusing these sensor data. Thus, the best characteristics from both types of sensors are utilized for the fused measurements.*

3 Orientation Tracking in Flatland

Let's do some tracking. We will start in "flatland", i.e. there is one rotation angle θ that needs to be tracked. Imagine a 2D coordinate system with x and y axes and a rotating 1D VRduino centered in the origin. As illustrated in Figure 3 (left), the unknown rotation angle θ is measured with respect to the y axis. To keep it consistent with later definitions of 3D Euler angles, we call this angle *roll*.

For the flatland problem, we have a single gyroscope and a two orthogonal accelerometers at our disposal.

3.1 Integrating Gyro Measurements

The gyro measures a combination of the true angular velocity and noise. We assume that any potential bias is calibrated and already removed from these measurements. To relate angular velocity to the unknown angle, we use a first-order Taylor series around time t

$$\theta\left(t+\Delta t\right) \approx \theta\left(t\right) + \frac{\partial}{\partial t}\theta\left(t\right)\Delta t + \epsilon \tag{6}$$

Here, we know our estimate of θ at time t and we directly measure the angular velocity $\frac{\partial}{\partial t}\theta \equiv \omega$. The time step Δt is also known, e.g. by measuring the time difference between two gyro measurements with the microcontroller. Even without any measurement noise, the Taylor series is an approximation with an error that is proportional to the squared time step $\epsilon \sim \mathcal{O}\left(\Delta t^2\right)$.

We integrate gyro measurements as

$$\theta_{gyro}^{(t)} = \theta_{gyro}^{(t-1)} + \widetilde{\omega}\Delta t \tag{7}$$

to estimate $\theta_{gyro}^{(t)}$ from our previous estimate $\theta_{g}^{(t-1)}$, the measured velocity $\widetilde{\omega}$, and Δt. The estimated angle will always be relative to the initial orientation $\theta_{gyro}^{(0)}$, which we can set to a user-defined value such as 0 or the angle estimated by the accelerometers right after initialization.

One of the biggest challenges with gyro integration is not necessarily just the measurement noise or bias, but the approximation error ϵ which accumulates over time. Together, noise and approximation error create *drift* in the estimated orientation. We could use faster sensors and processors to make Δt shorter and improve noise characteristics of the sensor, but this only delays the inevitable: over time, our estimated angle θ_{gyro} drifts away from the

ground truth with a rate that is proportional to the squared time step or worse. Thus, gyro measurements are great for estimating short-term changes in orientation, but they become unreliable over time due to drift.

Figure 3 (right) plots the estimated angle of real gyro measurements in blue. Estimations that exclusively rely on gyro measurements are also know as *dead reckoning*. You can see that the blue line is not very noisy but that there is a global offset to the other plots, which increases over time (the horizontal axis). This is drift inherent to all dead reckoning approaches.

3.2 Estimating the Angle from the Accelerometer

Our two accelerometers measure linear acceleration in two dimensions $\widetilde{\mathbf{a}} = (\widetilde{a}_x, \widetilde{a}_y)$. We make the assumption that, on average, the measured acceleration vector points up with a magnitude of $9.81\, m/s^2$. Of course this assumption is violated when external forces act on the device or when measurements are corrupted by noise, but we simply ignore these cases and estimate the unknown angle as

$$\theta_{acc} = \text{atan}^{-1}\left(\frac{\widetilde{a}_x}{\widetilde{a}_y}\right) = \text{atan2}\left(\widetilde{a}_x, \widetilde{a}_y\right) \tag{8}$$

Most programming languages provide the *atan2* function, which prevents division by zero and adequately handles the different combinations of signs in $\widetilde{a}_x$ and $\widetilde{a}_y$. If at all possible, use *atan2*!

As expected, this works well when no forces other than gravity act on the device and in the absence of sensor noise. Unfortunately, this is not really the case in practice, so the estimated angle will also be quite noisy. However, what is important to understand is that the gravity vector provides a stable reference direction that remains constant over time. So, even if individual measurements and angle estimates θ_{acc} are noisy, there is *no drift*. Thus, the accelerometer is reliable in the longer run, but unreliable for individual measurements.

The orange plot in Figure 3 (right) shows the angle estimated exclusively from the accelerometer measurements. This is real data measured with the VRduino and it is quite noisy. However, the estimated angle remains close to the "ground truth", here approximated by the complementary filter.

3.3 Drift Correction via Complementary Filtering

The shortcomings of gyros (i.e., drift) and accelerometers (i.e., noise) are complementary and can thus be mitigated by sensor fusion. The basic idea for sensor fusion in this application is to apply a low-pass filter to the accelerometer measurements that removes noise and a high pass filter to the gyro measurements that removes drift before combining these measurements.

One of the simplest, yet most effective approaches to inertial sensor fusion is the complementary filter, which is simply a weighted combination of both estimates:

$$\theta^{(t)} = \alpha\left(\theta^{(t-1)} + \widetilde{\omega}\Delta t\right) + (1-\alpha)\,\text{atan2}\left(\widetilde{a}_x, \widetilde{a}_y\right) \tag{9}$$

As illustrated in Figure 3, the complementary filter gives us the best of both worlds – no noise and no drift. The blending weight α is defined by the user and can be tuned to particular sensor characteristics.

4 Estimating Pitch and Roll from a 3-axis Accelerometer

Next, we leave flatland and try to understand orientation tracking in 3D. As a first step, let us look more closely at only the accelerometer in this section. Assume we have a 3-axis accelerometer that measures $\widetilde{\mathbf{a}} = (\widetilde{a}_x, \widetilde{a}_y, \widetilde{a}_z)$.

When working with rotations in 3D, we have to define which representation for rotations we will use. In this section, it makes sense to work with Euler angles because they are easy to understand. To this end, we define three angles $\theta_x, \theta_y, \theta_z$ that represent rotations around the respective axis. We call these angles yaw, pitch, and roll for rotations around the y, z, and x axis, respectively (see Figure 4). In addition to the angles, we also need to define in which order

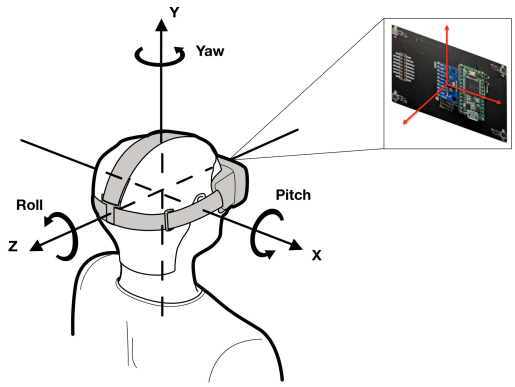

Figure 4: *Illustration of the body or sensor frame as well as yaw, pitch, and roll angles. (Image of head reproduced from Oculus VR). Note that this coordinate system assumes that the VRduino is mounted on the HMD such that the Teensy and the IMU are facing the viewer. In practice, it may be more convenient to mount it the other way around with double-sided tape. Please see Section 5.4 for details on the required coordinate transformations in that case.*

these are applied, because rotations are generally *not commutative*. We will use the order yaw, pitch, then roll, such that the rotation from world (or inertial) frame to the body (or sensor) frame is $\mathbf{R} = \mathbf{R}_z\left(-\theta_z\right)\mathbf{R}_x\left(-\theta_x\right)\mathbf{R}_y\left(-\theta_y\right)$.

Similar to Section 3.2, we make the assumption that, on average, the accelerometers measure the gravity vector, which points up, in the local body (sensor) frame. Again, this assumption is easily violated by sensor noise or forces acting on the sensor, but we ignore that for now. Thus, we write

$$\begin{aligned}\widehat{\mathbf{a}} = \frac{\widetilde{\mathbf{a}}}{\|\widetilde{\mathbf{a}}\|_2} &= \underbrace{\begin{pmatrix}\cos\left(-\theta_z\right) & -\sin\left(-\theta_z\right) & 0\\ \sin\left(-\theta_z\right) & \cos\left(-\theta_z\right) & 0\\ 0 & 0 & 1\end{pmatrix}}_{\mathbf{R}_z(-\theta_z)}\underbrace{\begin{pmatrix}1 & 0 & 0\\ 0 & \cos\left(-\theta_x\right) & -\sin\left(-\theta_x\right)\\ 0 & \sin\left(-\theta_x\right) & \cos\left(-\theta_x\right)\end{pmatrix}}_{\mathbf{R}_x(-\theta_x)}\underbrace{\begin{pmatrix}\cos\left(-\theta_y\right) & 0 & \sin\left(-\theta_y\right)\\ 0 & 1 & 0\\ -\sin\left(-\theta_y\right) & 0 & \cos\left(-\theta_y\right)\end{pmatrix}}_{\mathbf{R}_y(-\theta_y)}\begin{pmatrix}0\\1\\0\end{pmatrix}\\ &= \begin{pmatrix}-\cos\left(-\theta_x\right)\sin\left(-\theta_z\right)\\ \cos\left(-\theta_x\right)\cos\left(-\theta_z\right)\\ \sin\left(-\theta_x\right)\end{pmatrix}\end{aligned} \tag{10}$$

Here, $\widehat{\mathbf{a}}$ is the normalized vector of accelerometer measurements $\widetilde{\mathbf{a}}$ and the vector $(0, 1, 0)$ is the "up vector" in world coordinates, which is rotated into the body frame of the sensors.

One very important insight of Equation 10 is that the accelerometer measurements are not dependent on yaw (i.e., θ_y) at all. This is intuitive, because we could rotate the VRduino around the y axis without affecting the gravity vector pointing up. This implies that accelerometer measurements alone are insufficient to estimate yaw. Nevertheless, we can still estimate pitch and roll. Together, these angles are called *tilt*.

Pitch It would seem obvious to directly use $\widehat{a}_z$ to calculate $\theta_x = -\sin^{-1}\left(\widehat{a}_z\right)$. However, this is ambiguous because multiple different values of θ_x would actually result in the same measurement $\widehat{a}_z$. Instead, we use the following formulation:

$$\frac{\widehat{a}_z}{\sqrt{\widehat{a}_x^2 + \widehat{a}_y^2}} = \frac{\sin\left(-\theta_x\right)}{\sqrt{\cos^2\left(-\theta_x\right)\left(\sin^2\left(-\theta_z\right) + \cos^2\left(-\theta_z\right)\right)}} = \tan\left(-\theta_x\right) \quad \Rightarrow \quad \theta_x = -\text{atan2}\left(\widehat{a}_z, \text{sign}\left(\widehat{a}_y\right)\sqrt{\widehat{a}_x^2 + \widehat{a}_y^2}\right) \tag{11}$$

This gives us an unambiguous estimate of the pitch angle. We use the insight that $\sin^2\left(-\theta_z\right) + \cos^2\left(-\theta_z\right) = 1$ and we require the sign of $\widehat{a}_y$ in the second argument of *atan2* to make sure the estimated angle is valid over the full range $[-\pi, \pi]$ and not just over half of it.

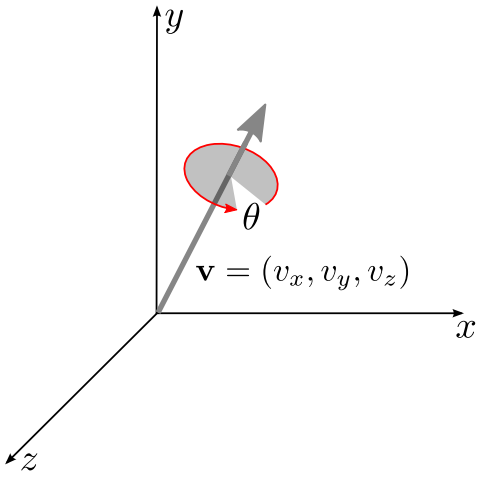

Figure 5: *Axis-angle representation for rotations. Instead of representing a 3D rotation using a sequence of rotations around the individual coordinate axes, as Euler angles do, the axis-angle representation uses a normalized vector* $\mathbf{v}$ *around which the rotation is defined by some angle* θ*. Although the axis-angle representation requires 4 values to define a rotation, instead of 3, it is preferred for many applications in computer graphics, vision, and virtual reality.*

Roll Similarly, we calculate roll as

$$\frac{\widehat{a}_x}{\widehat{a}_y} = -\frac{\sin\left(-\theta_z\right)}{\cos\left(-\theta_z\right)} = -\tan\left(-\theta_z\right) \qquad \Rightarrow \qquad \theta_z = -\tan^{-1}\left(-\frac{\widehat{a}_x}{\widehat{a}_y}\right) = -\text{atan2}\left(-\widehat{a}_x, \widehat{a}_y\right) \tag{12}$$

Again, we want to use the function *atan2* whenever possible.

5 Orientation Tracking with Quaternions

Unfortunately, Euler angles usually do not work for orientation tracking in 3D. In this week's assignment, you will explore the reasons for that in more detail.

Instead of Euler angles, we will work with quaternions. Some of the math in this section is adopted from [LaValle et al. 2014] and also Chapters 9.1 and 9.2 of [LaValle 2016]. A quaternion $q = q_w + iq_x + jq_y + kq_z$ is defined by 4 coefficients: a scalar q_w and a vector part q_x, q_y, q_z. The fundamental quaternion units i, j, k are comparable to the imaginary part of complex numbers, but there are three different fundamental units for each quaternion. Moreover, there are two types of quaternions: quaternions representing a rotation and vector quaternions.

Think about a rotation quaternion as a wrapper for the axis-angle representation of a rotation (see Fig. 5). The benefit of using a quaternion-based axis-angle representation is that there is a well-defined set of operations for quaternions that allow us to add or multiply them, to interpolate them, and to convert them directly to 3×3 rotation matrices (see Appendix A). A rotation quaternion has unit length $\|q\| = \sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2} = 1$ and it can be constructed from a rotation of θ radians around an axis $\mathbf{v}$ as

$$q\left(\theta, \mathbf{v}\right) = \underbrace{\cos\left(\frac{\theta}{2}\right)}_{q_w} + i\underbrace{v_x \sin\left(\frac{\theta}{2}\right)}_{q_x} + j\underbrace{v_y \sin\left(\frac{\theta}{2}\right)}_{q_y} + k\underbrace{v_z \sin\left(\frac{\theta}{2}\right)}_{q_z} \tag{13}$$

A vector quaternion is not constrained to unit length and it usually represents a 3D point or a 3D vector $\mathbf{u} = (u_x, u_y, u_z)$. The scalar part of a vector quaternion is always zero:

$$q_{\mathbf{u}} = 0 + iu_x + ju_y + ku_z \tag{14}$$

Given a rotation quaternion q and a vector quaternion $q_{\mathbf{u}}$, we can rotate the point or vector described by the latter as

$$q'_{\mathbf{u}} = q\, q_{\mathbf{u}}\, q^{-1}, \tag{15}$$

where $q'_{\mathbf{u}}$ is the rotated vector quaternion. This operation is equivalent to multiplying the 3×3 rotation matrix corresponding to q with $\mathbf{u}$. Concatenating several rotations is done using the quaternion product $q = q_2 q_1$ and written as

$$q'_{\mathbf{u}} = q_2\, q_1\, q_{\mathbf{u}}\, q_1^{-1}\, q_2^{-1} \tag{16}$$

In Appendix A, you can find a detailed summary and definitions of the most important quaternion operations, formulas for converting quaternions to other rotation representations, including matrices and axis-angle representations.

5.1 Integrating 3-axis Gyro Measurements

Given the output of a 3-axis gyro $\widetilde{\boldsymbol{\omega}} = (\widetilde{\omega}_x, \widetilde{\omega}_y, \widetilde{\omega}_z)$, we can determine the normalized axis of this rotation as $\frac{\widetilde{\boldsymbol{\omega}}}{\|\widetilde{\boldsymbol{\omega}}\|}$ and the angle of rotation (in radians) as $\Delta t \, \|\widetilde{\boldsymbol{\omega}}\|$, where Δt is the time step. Using Equation 13, we convert this axis-angle representation to a rotation quaternion as

$$q_{\Delta} = q\left(\Delta t \, \|\widetilde{\boldsymbol{\omega}}\|, \frac{\widetilde{\boldsymbol{\omega}}}{\|\widetilde{\boldsymbol{\omega}}\|}\right) \tag{17}$$

Here, q_{Δ} represents the instantaneous rotation from the local sensor frame at current time step to the local sensor frame at the last time step.

Due to the fact that we continuously take measurements with the gyro and our goal is to integrate each instantaneous rotation to the desired orientation of the device in world coordinates, the forward model of the gyro combines all instantaneous rotations as

$$q_{\boldsymbol{\omega}}^{(t+\Delta t)} = q^{(t)} q_{\Delta} \tag{18}$$

Here, $q^{(t)}$ is the set of accumulated rotations from all previous time steps and q_{Δ} is the instantaneous rotation estimated from the current set of gyro measurements. Start with the quaternion $q^{(0)} = 1 + i0 + j0 + k0$ for initialization. Using Equation 15, a vector quaternion $q_{\mathbf{u}}$ can then be rotated from body to world frame as

$$q_{\mathbf{u}}^{(world)} = q_{\boldsymbol{\omega}}^{(t+\Delta t)}\, q_{\mathbf{u}}^{(body)}\, q_{\boldsymbol{\omega}}^{(t+\Delta t)^{-1}} = q^{(t)}\, q_{\Delta}\, q_{\mathbf{u}}^{(body)}\, q_{\Delta}^{-1}\, q^{(t)^{-1}} \tag{19}$$

Note that for a dead reckoning approach, i.e. orientation tracking using only gyros, $q^{(t)} = q_{\boldsymbol{\omega}}^{(t)}$ but if a sensor fusion approach is used, we want to use the best available estimate, so $q^{(t)}$ would be the orientation estimated by the sensor fusion approach at the last time step.

5.2 Estimating Tilt with the Accelerometer

We already discussed tilt, i.e. pitch and roll, in the last section, but let us outline how to estimate it with quaternions. First, we represent the measured accelerometer values $\widetilde{\mathbf{a}} = (\widetilde{a}_x, \widetilde{a}_y, \widetilde{a}_z)$ as a vector quaternion in the body frame

$$q_{\mathbf{a}}^{(body)} = 0 + i\widetilde{a}_x + j\widetilde{a}_y + k\widetilde{a}_z \tag{20}$$

As usual, we assume that the accelerometer vector exclusively measures gravity, which means it should point up in world coordinates. However, we measured the acceleration in local sensor coordinates, so we need to rotate it from the body to the world frame. Unfortunately, we do not know that rotation because it is the very quantity we are trying to estimate. Intuitively, we could define the up vector as $q_{up}^{(world)} = 0 + i0 + j1 + k0$ and compute the rotation quaternion that would rotate $q_{\mathbf{a}}^{(body)}$ to $q_{up}^{(world)}$. This would make sense if we only have an accelerometer, but the goal of gyro and accelerometer sensor fusion here is slightly different: we want to correct the tilt error of the gyro measurements, or at least the pitch and roll component of it, using the accelerometer. Therefore, we use the rotation quaternion estimated by the gyro to rotate the accelerometer values into world space as

$$q_{\mathbf{a}}^{(world)} = q_{\boldsymbol{\omega}}^{(t+\Delta t)}\, q_{\mathbf{a}}^{(body)}\, q_{\boldsymbol{\omega}}^{(t+\Delta t)^{-1}} \tag{21}$$

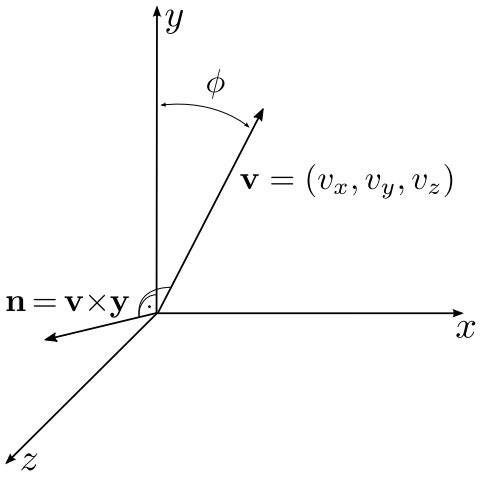

Figure 6: *Tilt correction quaternion. Given the vector quaternion of the accelerometer in world space* $q_{\mathbf{a}}^{(world)}$ *along with its normalized vector component* $\mathbf{v}$*, we can calculate a rotation axis* $\mathbf{n}$ *that is orthogonal to both the* y *axis and* $\mathbf{v}$ *via their cross product. The angle between the* y *axis and* $\mathbf{v}$ *is computed using their dot product* $\mathbf{y} \cdot \mathbf{v} = cos\,(\phi)$*.*

In the ideal case scenario, $q_{\mathbf{a}}^{(world)}$ would now point up, i.e. $q_{\mathbf{a}}^{(world)} = q_{up}^{(world)}$, but due to the gyro drift, sensor noise, and the fact that the the starting condition of $q^{(0)}$ may have not been correct, this is probably not going to be the case in practice.

Thus, our aim is to compute a rotation quaternion q_t that corrects for the tilt error. This is also illustrated in Figure 6. The tilt correction quaternion should satisfy

$$q_{up}^{(world)} = q_t \, q_{\mathbf{a}}^{(world)} \, q_t^{-1} \tag{22}$$

There are several ways one could go about calculating q_t. One intuitive way is to calculate its normalized rotation axis $\mathbf{n} = (n_x, n_y, n_z)$ and its rotation angle ϕ first and then convert that into a rotation quaternion. We can do that by calculating $\mathbf{n}$ as the vector that is orthogonal to the vector components of both $q_{up}^{(world)}$ and $q_{\mathbf{a}}^{(world)}$ via their cross product and ϕ as the angle between these two vector components (see Fig. 6). Remember that the angle between two vectors is calculated as their dot product, i.e. $\mathbf{v}_1 \cdot \mathbf{v}_2 = \|\mathbf{v}_1\| \, \|\mathbf{v}_2\| \cos(\phi)$. Therefore

$$q_t = q\left(\phi, \frac{\mathbf{n}}{\|\mathbf{n}\|}\right), \qquad \mathbf{n} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -v_z \\ 0 \\ v_x \end{pmatrix}, \qquad \phi = \cos^{-1}(v_y) \tag{23}$$

Here, $(v_x, v_y, v_z) = \left(q_{\mathbf{a}_x}{}^{(world)}, q_{\mathbf{a}_y}{}^{(world)}, q_{\mathbf{a}_z}{}^{(world)}\right) / \left\| q_{\mathbf{a}}^{(world)} \right\|$ is the normalized vector component of $q_{\mathbf{a}}^{(world)}$.

5.3 Tilt Correction with the Complementary Filter

As discussed in Section 3.3, the complementary filter is a linear interpolation between the orientation estimated by the gyro and that estimated by the accelerometer. In quaternion operations, this idea can be mathematically expressed by a rotation from body space to world space, as estimated by the gyro, and then rotating some more along the tilt correction quaternion q_t

$$q_c^{(t+\Delta t)} = q\left((1-\alpha)\,\phi, \frac{\mathbf{n}}{\|\mathbf{n}\|}\right) q_{\boldsymbol{\omega}}^{(t+\Delta t)} \tag{24}$$

Here, $q_c^{(t+\Delta t)}$ is the rotation quaternion that includes the estimated gyro rotation as well as the tilt correction from the accelerometer. The user-defined parameter $0 \leq \alpha \leq 1$ defines how aggressively tilt correction is being applied. Thus, a point or vector quaternion in the local body frame of the VRduino is rotated into world space as $q_{\mathbf{u}}^{(world)} = q_c^{(t+\Delta t)} \, q_{\mathbf{u}}^{(body)} \, q_c^{(t+\Delta t)^{-1}}$.

Note that Equation 18 outlined the gyro quaternion update as $q_{\boldsymbol{\omega}}^{(t+\Delta t)} = q^{(t)} q_{\Delta}$. If we only have a gyro at our disposal, we would use $q^{(t)} = q_{\boldsymbol{\omega}}^{(t)}$. When working with a complementary filter, however, we actually have a better estimate of the orientation at the last time step from the complementary filter: $q_c^{(t)}$. Therefore, we will be using $q^{(t)} = q_c^{(t)}$ in the gyro update.

5.4 Integration into the Graphics Pipeline

For best performance, the gyro and accelerometer values are read on the microcontroller (e.g., the Teensy on the VRduino) and all of the above calculations are done there as well. The resulting rotation quaternion q_c is then streamed, for example via serial USB, to a host computer that implements the rendering routines. Rounding errors in the serial data transmission may require the quaternion to be re-normalized after transmission. Remember that only normalized quaternions represent rotations, even slight rounding errors may result in this assumption to be violated and you may get unexpected results if you ignore that.

5.4.1 Visualizing Quaternion Rotation with a Test Object

The easiest way to verify that your quaternion-based orientation tracker is working correctly is to render a test object, such as coordinate axes, and rotate them in real-time with your quaternions streamed from the VRduino. Algorithm 1 outlines pseudo code for the rendering part of this. The quaternion is updated and re-normalized from the incoming serial stream, it is converted to the angle and axis representation (see Eq. 36), and then the *glRotate* function, or a suitable alternative, can be directly used with this angle and axis to create an appropriate rotation. After that the test object is rendered. Figure 7 shows the output of rendering such a test object.

Algorithm 1 Render Test Object to Verify Quaternion Rotation

```
1: double q[4];
2: updateQuaternionFromSerialStream(q);                          // get quaternion from serial stream
3: float angle, axisX, axisY, axisZ;
4: quatToAxisAngle(q[0], q[1], q[2], q[3], angle, axisX, axisY, axisZ);   // convert quaternion to axis and angle
5: glRotatef(angle, axisX, axisY, axisZ);                        // rotate via axis-angle representation
6: drawTestObject();                                             // draw the target object
```

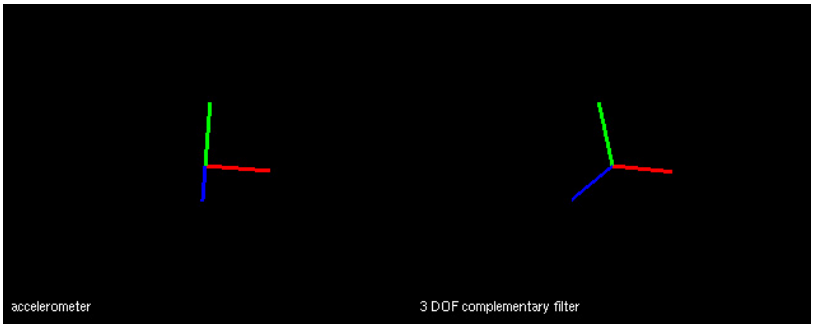

Figure 7: *Coordinate axis rendered with OpenGL using the estimated orientation streamed from the VRduino. Using only the accelerometer (left image), only pitch and roll can be estimated but yaw, i.e. rotations around the y-axis, are missing. Using the quaternion-based complementary filter (right image), the rotation around all 3 axis is tracked accurately.*

5.4.2 Using Quaternion Rotation to Control the Viewer in VR

Rotating a rendered object with a physical object, like the VRduino, is really fun. However, in many cases what we really want is to control the camera in the virtual environment and map the user's head orientation to it. To map the HMD to the virtual camera, we will use OpenGL's convention of the camera/HMD looking down the negative negative z-axis, as illustrated in Figure 4.

An intuitive solution for this is to set the view matrix to the inverse rotation of what we used in Section 5.4.1. Instead of rotating the test object in the world along with the VRduino, this approach would rotate the world around the viewer in the inverse manner (which is the right thing to do). Applying the inverse rotation of a quaternion is as simple as flipping the sign of the angle in its angle-axis representation. So the command *glRotatef(-angle, axisX, axisY, axisZ)* is the inverse rotation of *glRotatef(angle, axisX, axisY, axisZ)*. You could also convert the quaternion to a 4×4 rotation matrix and invert that or do other conversions, but it may end up being less efficient than simply flipping the sign of the angle. You can find more details on this in Appendix A.

In practice, we need to take care of a few more things here. First, we often use stereo rendering so it may not be entirely clear what the order of the operations is here. Second, we may have to turn the VRduino around to be able to mount it on the HMD, in which case we have to account for that transformation explicitly in the rendering code (otherwise, the rotations are messed up). Third, the center of rotation of your head is usually not where the IMU is, so we we may want to add constraints of the user's anatomy to the rendering pipeline. But let's do this step by step.

Stereo Rendering with IMU Being the Center of Rotation Let us consider the first case where the VRduino is mounted (or held) on the HMD as shown in Figure 8 (left). It is a bit awkward to hold the VRduino in front of the HMD, but at least the coordinates systems of the HMD and the VRduino match (see Figs. 1 and 4). We also ignore the fact that our head does not actually rotate around the IMU for now. However, we do want to include stereo rendering by setting the appropriate view and projection matrices for each of the left and right view.

The sequence of necessary steps is outlined by Algorithm 2. What is important to note is that we want to render the scene first, then set the view matrix (e.g., by using the *lookAt* function), then include the rotation of the quaternion streamed from the VRduino as the rotation around its axis by the negative angle (i.e. inverse rotation as discussed above).

Algorithm 2 Render Stereo View with Quaternion Rotation

```
1: double q[4];
2: updateQuaternionFromSerialStream(q);                                  //get quaternion from serial stream
3: setProjectionMatrix();
4: float angle, axisX, axisY, axisZ;
5: quatToAxisAngle(q[0], q[1], q[2], q[3], angle, axisX, axisY, axisZ);  // convert quaternion to axis and angle
6: glRotatef(-angle, axisX, axisY, axisZ);                               // rotate via axis-angle representation
7: double ipd = 64.0;                                                    // define interpupillary distance (in mm)
8: setLookAtMatrix( ± ipd/2, 0, 0, ± ipd/2, 0, -1, 0, 1, 0 );            // set view matrix for right/left eye
9: drawScene();
```

A Better Way of Mounting the VRduino on the HMD It is much easier to mount the VRduino on the HMD as shown in Figure 8 (right). Unfortunately, the local coordinate system of the VRduino now does not match that of the HMD anymore. We can account for that by include one additional transformation to Algorithm 2. We basically need to rotate the transformation given by the quaternion by 180°. The easiest way to do that is to simply flip the x and z-coordinates of the axis in the axis-angle representation. So we will replace line 6 of Algorithm 2 by the following statement: *glRotatef(-angle, -axisX, axisY, -axisZ);*.

Head and Neck Model Finally, the head does not actually rotate around the IMU, but around some point that is between the head and the spine. Incorporating this head and neck model is not only important to correct the pitch from being centered on the IMU, but it also gives us a little bit of motion parallax when we roll our head because it actually moves slightly to the left and the right. Assume that the center of rotation is a distance l_h away from the IMU on the z-axis, i.e. half the length of the head, and a distance l_n away from the IMU along the y-axis, i.e. accounting for the neck, then we can incorporate the offset between IMU and center of rotation as

Note that Algorithm 3 assumes that the VRduino is mounted on the HMD using the *convenient* way, shown in

Figure 8: *Two ways of mounting the VRduino on the HMD. The left version is not very convenient to mount whereas the right version allows you to mount the VRduino on the HMD with double-sided tape. However, keep in mind that the local coordinate systems of HMD and VRduino are different. They only match in the left case and are rotated by 180° in the right case; we need to take that into account for the rendering.*

Algorithm 3 Render Stereo View with Quaternion Rotation and Head and Neck Model

```
1:  double q[4];
2:  updateQuaternionFromSerialStream(q);                          //get quaternion from serial stream
3:  setProjectionMatrix();
4:  glTranslatef(0, -l_n, -l_h);
5:  float angle, axisX, axisY, axisZ;
6:  quatToAxisAngle(q[0], q[1], q[2], q[3], angle, axisX, axisY, axisZ);   // convert quaternion to axis and angle
7:  glRotatef(-angle, -axisX, axisY, -axisZ);                     // rotate via axis-angle representation
8:  glTranslatef(0, l_n, l_h);
9:  double ipd = 64.0;                                            // define interpupillary distance (in mm)
10: setLookAtMatrix( ± ipd/2, 0, 0, ± ipd/2, 0, -1, 0, 1, 0 );    // set view matrix for right/left eye
11: drawScene();
```

Figure 8 (right).

6 Appendix A: Quaternion Algebra Reference

This appendix is a summary of quaternion definitions and algebra useful for orientation tracking. A quaternion $q = q_w + iq_x + jq_y + kq_z$ is defined by 4 coefficients: a scalar q_w and a vector part q_x, q_y, q_z. The fundamental quaternion units i, j, k are comparable to the imaginary part of complex numbers, but there are three of them for each quaternion. Each of the fundamental quaternion units is different, but the following relationships hold

$$i^2 = j^2 = k^2 = ijk = -1, \quad ij = -ji = k, \quad ki = -ik = j, \quad jk = -kj = i \tag{25}$$

In general, there are two types of quaternions important for our application: quaternions representing a rotation and vector quaternions.

A valid rotation quaternion has unit length, i.e.

$$\|q\| = \sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2} = 1 \tag{26}$$

and the quaterions q and $-q$ represent the same rotation.

A vector quaternion representing a 3D point or vector $\mathbf{u} = (u_x, u_y, u_z)$ can have an arbitrary length but its scalar part is always zero

$$q_{\mathbf{u}} = 0 + iu_x + ju_y + ku_z \tag{27}$$

The conjugate of a quaternion is

$$q^* = q_w - iq_x - jq_y - kq_z \tag{28}$$

and its inverse

$$q^{-1} = \frac{q^*}{\|q\|^2} \tag{29}$$

Similar to polynomials, two quaternion q and p are added as

$$q + p = (q_w + p_w) + i\,(q_x + p_x) + j\,(q_y + p_y) + k\,(q_z + p_z) \tag{30}$$

and multiplied as

$$\begin{aligned} q\,p &= (q_w + iq_x + jq_y + kq_z)\,(p_w + ip_x + jp_y + kp_z) && (31)\\ &= (q_w p_w - q_x p_x - q_y p_y - q_z p_z) \\ &\quad + i\,(q_w p_x + q_x p_w + q_y p_z - q_z p_y) \\ &\quad + j\,(q_w p_y - q_x p_z + q_y p_w + q_z p_x) \\ &\quad + k\,(q_w p_z + q_x p_y - q_y p_x + q_z p_w) && (32) \end{aligned}$$

Rotations with Quaternions A quaternion q can be converted to a 3×3 rotation matrix as

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} = \begin{pmatrix} q_w^2 + q_x^2 - q_y^2 - q_z^2 & 2q_xq_y - 2q_wq_z & 2q_xq_z + 2q_wq_y \\ 2q_xq_y + 2q_wq_z & q_w^2 - q_x^2 + q_y^2 - q_z^2 & 2q_yq_z - 2q_wq_x \\ 2q_xq_z - 2q_wq_y & 2q_yq_z + 2q_wq_x & q_w^2 - q_x^2 - q_y^2 + q_z^2 \end{pmatrix} \tag{33}$$

Similarly, a 3×3 rotation matrix is converted to a quaterion as

$$q_w = \frac{\sqrt{1 + r_{11} + r_{22} + r_{33}}}{2}, \qquad q_x = \frac{r_{32} - r_{23}}{4q_w}, \qquad q_y = \frac{r_{13} - r_{31}}{4q_w}, \qquad q_z = \frac{r_{21} - r_{12}}{4q_w} \tag{34}$$

Due to numerical precision of these operations, it may be necessary to re-normalize the resulting quaterion to make sure it has unit length.

A rotation of θ radians around a normalized axis $\mathbf{v}$ is

$$q\left(\theta, \mathbf{v}\right) = \underbrace{\cos\left(\frac{\theta}{2}\right)}_{q_w} + i\underbrace{v_x \sin\left(\frac{\theta}{2}\right)}_{q_x} + j\underbrace{v_y \sin\left(\frac{\theta}{2}\right)}_{q_y} + k\underbrace{v_z \sin\left(\frac{\theta}{2}\right)}_{q_z} \tag{35}$$

and the axis-angle representation for a rotation is extracted from a rotation quaternion as

$$\theta = 2\cos^{-1}\left(q_w\right), \quad v_x = \frac{q_x}{\sqrt{1-q_w^2}}, \quad v_y = \frac{q_y}{\sqrt{1-q_w^2}}, \quad v_z = \frac{q_z}{\sqrt{1-q_w^2}} \tag{36}$$

Given a 3D point or vector, its corresponding vector quaternion $q_{\mathbf{u}}$ is transformed by the rotation quaternion q as

$$q'_{\mathbf{u}} = q\, q_{\mathbf{u}}\, q^{-1} \tag{37}$$

and the inverse rotation is

$$q_{\mathbf{u}} = q^{-1}\, q'_{\mathbf{u}}\, q \tag{38}$$

Successive rotations by several rotation quaternions as $q_1, q_2, \ldots$ is expressed as

$$q'_{\mathbf{u}} = \ldots\, q_2\, q_1\, q_{\mathbf{u}}\, q_1^{-1}\, q_2^{-1}\, \ldots \tag{39}$$

References

LaValle, S. M., Yershova, A., Katsev, M., and Antonov, M. 2014. Head tracking for the oculus rift. In *IEEE International Conference on Robotics and Automation (ICRA)*, 187–194.

LaValle, S. 2016. *Virtual Reality*. Cambridge University Press.