PointBlank Report

B04901036 and B04901165

ES-LAB, NTUEE

1 Introduction

1.1 Overview

PointBlank is intended to be an advanced version of a presentation pointer. It utilizes a gyroscope combined with an accelerometer rather than visual light markers to determine the pointer's orientation. In addition, it has the ability to act as a magnifying glass or simulate a mouse, expanding the capability of presentation pointers.

1.2 Components

- Raspberry Pi 3(with Raspbian as OS)
- Arduino Uno R3
- MPU9250 (Gyro + Accelerometer + Compass)
- Analog five button keyboard module (such as the one shown in the picture)

- Power bank
- A USB cable with switch (connecting Raspberry Pi to the Power Bank)

1.3 Usage

Please refer to `https://github.com/NTUEE-ESLab/2018Fall-PointBlank`

2 Graphical User Interface

To achieve the basic goal of drawing a circle around the cursor, we used PyQt, a binding of the GUI toolkit Qt in Python. Qt has a well-written and detailed documentation on its website [1] .

The first step is to create a full screen transparent window that does not take any inputs from mouse and keyboard, i.e., the user inputs fall through to the next topmost window. With transparency of both visual and input, the window can act as a canvas to be drawn on without interfering with user experience.

The key is to set the flags *WindowTransparentForInput*, *FramelessWindowHint*, *WindowStaysOnTopHint* and *X11BypassWindowManagerHint* when creating the window.

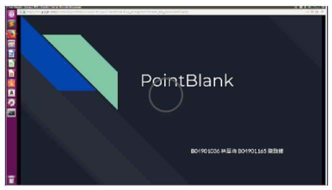
2.1 Rendering

Once we have the canvas in place, we can start to draw the desired accentuation on it. We provide three ways to emphasize the point of attention, dubbed "laser", "highlight" and "magnify". The class *QPainter* is most useful when rendering shapes, text and images.

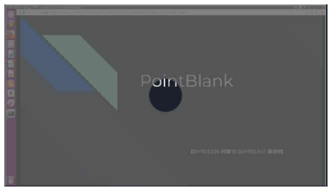
Laser Simply call *QPainter.drawEllipse()* to draw a circle at the desired location while specifying the color and width of the painter's pen.

Highlight First, draw a fill the screen with color by calling *QPainter.fillRect()*. Then use *QPainter.drawEllipse()* to fill a circle with color (0,0,0,0), i.e., full transparent.

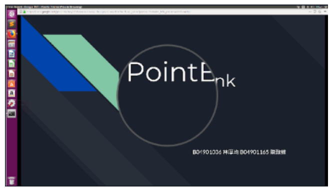
Magnify Take a screenshot by calling *QScreen.grabWindow()* the instant the activation button is pressed, then use *QPixmap.scaled()* to magnify the image. Finally, show the magnified part of the image with the desired outline by calling *setClipRegion(), translate()* and *drawPixmap()* of *QPainter*.



(a) Laser mode



(b) Highlight mode



(c) Magnify mode

Fig. 1: Presentation Modes of PointBlank

2.2 Keybaord/Mouse Input Simulation

For the presentation modes (laser, highlight, magnify), PointBlank allows control of the slides by simulating UP and DOWN of the keyboard. In mouse mode, PointBlank does not draw additional shapes on the screen, but rather simulates the function of a normal mouse with left and right clicks. These functions are implemented by the python library *pyautogui*.

2.3 Customization

Customization of PointBlank exists as a seperate application, which is also implemented with Qt. It allows users to change the size, width and color of the circle, as well as the degree of magnification. The user can instantly preview the changes. The main application must be restarted for the settings to take effect.

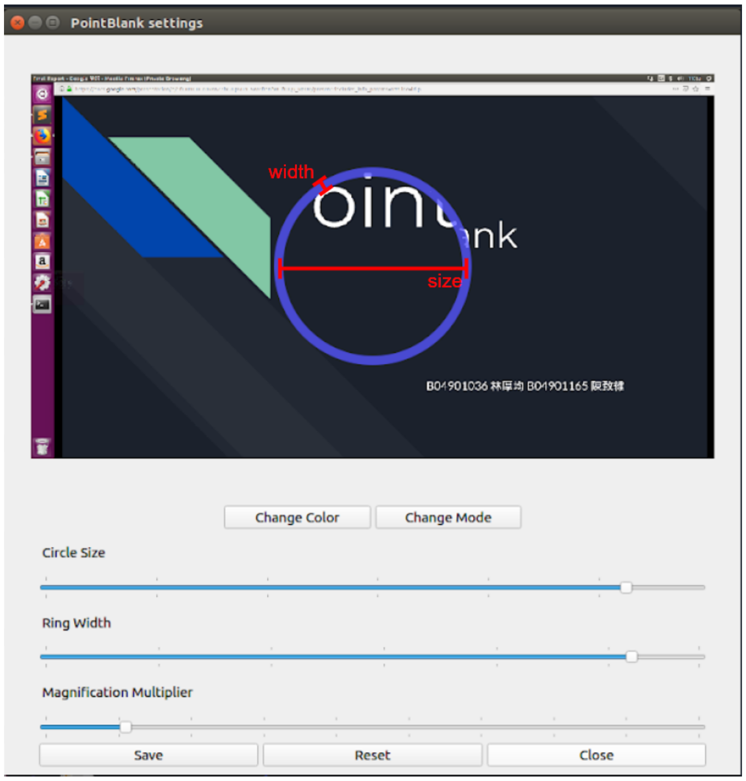


Fig. 2: Customization application

3 Sensors and Communications

The overall structure of PointBlank and communications between its components are shown in figure 3. Raspberry Pi computes the orientation of the pointer using measurements from MPU9250 and transmits the processed data to the PC, along with the state of the button module.

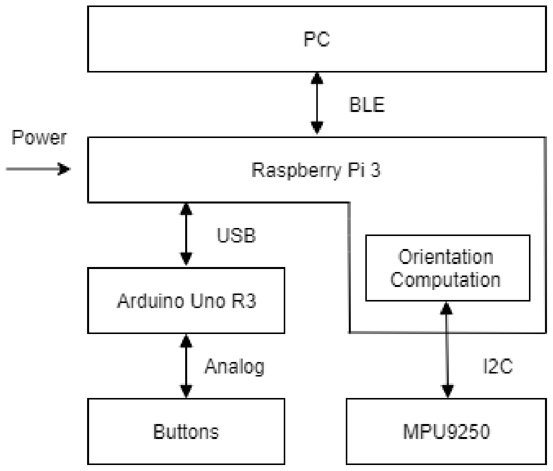


Fig. 3: Structure of PointBlank.

3.1 Sensors

MPU9250 To determine the orientation of the pointer, we used MPU9250, a multi-chip module with a gyroscope, an accelerometer, and a compass built into it. We did not use the compass in this project because it is subject to interference. Furthermore, the gyroscope and the accelerometer combined satisfy our needs.

The output formats of both the gyroscope and the accelerometer consist of three 16-bit integers, representing the value of angular velocity and acceleration along the x, y, and z axes. There are multiple configurations for the full scale range and the bandwidth of the built-in low pass filter. Our setting is shown below:

- gyroscope
  - full scale range: $\pm$ 2000 deg/s
  - LPF bandwidth: 5 Hz
- accelerometer
  - full scale range: $\pm$ 16 g
  - LPF bandwidth: 5 Hz

We activate MPU9250 when the activation button is pressed and collect the data from its gyroscope and accelerometer at roughly the rate of 500 Hz. For the rest of the time, MPU9250 is in sleep mode in order to save power.

Please refer to the datasheet [3] for more information on MPU9250.

Buttons The button module that we used has five buttons connected to an analog output pin. The voltage output is 5V when no buttons are pressed, and 0V, 0.15V, 0.43V, 0.82V and 1.74V when each of the buttons is pressed in turn.

Unfortunately, Raspberry Pi does not have analog inputs, so we had to let Arduino Uno serve as an A/D device. However, Arduino Uno plays another role here. We do not simply pass the digitized voltage level directly to Raspberry Pi. Arduino Uno reads the buttons' output at a rate of 20 Hz, but only sends a signal to Raspberry Pi when it detects a change of state, i.e., when a button is pressed or released. In this sense, Raspberry Pi receives a signal very similar to an interrupt, as is what we normally do with a digital button.

3.2 Bluetooth Low Energy

The communication between PointBlank and the application on PC is implemented by Bluetooth Low Energy. The application receives two types of data:

– Two floating point numbers ranging between 0 and 1, indicating the relative position of the projection of PointBlank on screen.
– An integer indicating which button is pressed (or released).

The data are transmitted whenever their values are changed, which is implemented by notifications, a feature of Bluetooth Low Energy. The Position data is updated at around 50 Hz when the pointer is active, and the Button data is sent every time the button module's state has changed.

Central For the application on PC, we used Qt for Bluetooth functionality in order to have better compatibility with the GUI. Qt provides *QBluetoothDeviceDiscoveryAgent* to search for nearby devices, and *QLowEnergyController* for connection to the device. From there, *QLowEnergyService* and *QLowEnergyCharacteristic* provide access to read, write and turn on notifications of characteristics.

Peripheral BlueZ has defined classes of Advertisement, Service, Characteristic and Descriptor in its example codes, which can be found on BlueZ's website [2]. We inherited the classes and filled in the necessary values such as UUIDs and behaviors of read/write operations. Then it is a simple matter of registering those classes to the underlying main structure, in this case dbus and BlueZ's manager.

4 Orientation

4.1 Representation

The data from the sensors are measured with respect to the sensor frame, but what we need is the orientation of pointer in the global coordinate system, or the earth frame. To convert the raw data into the information we need, we have to select an effective representation to describe different coordinate systems and the relations between them. The representation is also required to be convenient for subsequent data processing such as data fusion of different sensors, orientation computation and noise filtering. We will now introduce the concept of three-dimensional rotation and various notations.

Vector representation A vector has different components in each axis in different reference frames. We denote the unit vector of x, y, and z axis by $\mathbf{u_x}$, $\mathbf{u_y}$, and $\mathbf{u_z}$ respectively. Thus a vector in different frames, for example the sensor frame S and the earth frame E, has different forms

$$\mathbf{v} = v_{x_S}\mathbf{u}_{x_S} + v_{y_S}\mathbf{u}_{y_S} + v_{z_S}\mathbf{u}_{z_S} = v_{x_E}\mathbf{u}_{x_E} + v_{y_E}\mathbf{u}_{y_E} + v_{z_E}\mathbf{u}_{z_E}$$

or we can denote the vector as

$$[\mathbf{v}]_S = \begin{bmatrix} v_{x_S} \\ v_{y_S} \\ v_{z_S} \end{bmatrix}, \quad [\mathbf{v}]_E = \begin{bmatrix} v_{x_E} \\ v_{y_E} \\ v_{z_E} \end{bmatrix}$$

Vector transformation We can find the components of a vector in the earth frame in terms of those in the sensor frame

$$\begin{aligned} [\mathbf{v}]_E = \begin{bmatrix} \mathbf{v}\cdot\mathbf{u}_{x_E} \\ \mathbf{v}\cdot\mathbf{u}_{y_E} \\ \mathbf{v}\cdot\mathbf{u}_{z_E} \end{bmatrix} &= \begin{bmatrix} (v_{x_S}\mathbf{u}_{x_S} + v_{y_S}\mathbf{u}_{y_S} + v_{z_S}\mathbf{u}_{z_S})\cdot\mathbf{u}_{x_E} \\ (v_{x_S}\mathbf{u}_{x_S} + v_{y_S}\mathbf{u}_{y_S} + v_{z_S}\mathbf{u}_{z_S})\cdot\mathbf{u}_{y_E} \\ (v_{x_S}\mathbf{u}_{x_S} + v_{y_S}\mathbf{u}_{y_S} + v_{z_S}\mathbf{u}_{z_S})\cdot\mathbf{u}_{z_E} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{u}_{x_S}\cdot\mathbf{u}_{x_E} & \mathbf{u}_{y_S}\cdot\mathbf{u}_{x_E} & \mathbf{u}_{z_S}\cdot\mathbf{u}_{x_E} \\ \mathbf{u}_{x_S}\cdot\mathbf{u}_{y_E} & \mathbf{u}_{y_S}\cdot\mathbf{u}_{y_E} & \mathbf{u}_{z_S}\cdot\mathbf{u}_{y_E} \\ \mathbf{u}_{x_S}\cdot\mathbf{u}_{z_E} & \mathbf{u}_{y_S}\cdot\mathbf{u}_{z_E} & \mathbf{u}_{z_S}\cdot\mathbf{u}_{z_E} \end{bmatrix} \begin{bmatrix} v_{x_S} \\ v_{y_S} \\ v_{z_S} \end{bmatrix} \\ &= [\mathbf{R}]_E^S \, [\mathbf{v}]_S \end{aligned}$$

where $[\mathbf{R}]_E^S$ is a rotation matrix transforming a vector from frame S to frame E. It has the following properties:

$$\begin{aligned} [\mathbf{R}]_C^A &= [\mathbf{R}]_C^B \, [\mathbf{R}]_B^A \\ [\mathbf{R}]_A^B &= ([\mathbf{R}]_B^A)^{-1} = ([\mathbf{R}]_B^A)^T \end{aligned}$$

Rotation matrix Notice that

$$[\mathbf{v}]_E = \begin{bmatrix} \mathbf{u}_{x_S} \cdot \mathbf{u}_{x_E} & \mathbf{u}_{y_S} \cdot \mathbf{u}_{x_E} & \mathbf{u}_{z_S} \cdot \mathbf{u}_{x_E} \\ \mathbf{u}_{x_S} \cdot \mathbf{u}_{y_E} & \mathbf{u}_{y_S} \cdot \mathbf{u}_{y_E} & \mathbf{u}_{z_S} \cdot \mathbf{u}_{y_E} \\ \mathbf{u}_{x_S} \cdot \mathbf{u}_{z_E} & \mathbf{u}_{y_S} \cdot \mathbf{u}_{z_E} & \mathbf{u}_{z_S} \cdot \mathbf{u}_{z_E} \end{bmatrix} \begin{bmatrix} v_{x_S} \\ v_{y_S} \\ v_{z_S} \end{bmatrix}$$
$$= \left[[\mathbf{u}_{x_S}]_E \ [\mathbf{u}_{y_S}]_E \ [\mathbf{u}_{z_S}]_E \right] [\mathbf{v}]_S$$

which means that by keeping track of the three axes of the sensor frame in the earth frame, the rotation matrix for vector transformation $[\mathbf{R}]_E^S$ can be constructed directly. By the same logic, $[\mathbf{R}]_E^S$ itself can be interpreted as the orientation of frame S relative to frame E.

However, this method suffers from noise and numerical error: it is not easy to ensure that the three axes are still perpendicular to each other after a series of updates. This problem arises from the fact that maintaining the three axes has 9 degrees of freedom but the 3D rotation itself has only 3 degrees of freedom, so the 9 parameters are actually dependent on each other.

Roll-Pitch-Yaw (Euler Angles) This method decomposes the 3D rotation to exactly three rotations: roll angle θ, pitch angle ϕ, and yaw angle ψ, rotating about the x, y, and z axes respectively.

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}, \ \mathbf{R}_y(\phi) = \begin{bmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 0 & 0 \\ \sin\phi & 0 & \cos\phi \end{bmatrix},$$
$$\mathbf{R}_z(\psi) = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

and the final rotation matrix are constructed by applying these three rotations in a specific order:

$$[\mathbf{R}]_E^S = \mathbf{R}_x(\theta)\mathbf{R}_y(\phi)\mathbf{R}_z(\psi)$$

It is easy to see that the number of parameters meets the intrinsic degrees of freedom of 3D rotation. However, this representation has another problem, called gimbal lock. Gimbal lock happens whenever magnitude of the second rotation ($\mathbf{R}_y$) is 90 degrees, causing the axis of the third rotation (x axis) to be parellel with the axis of the first rotation (z axis), and thus makes the system lose one degree of freedom. Therefore, this representation actually does not always have three degrees of freedom.

Quarternion

Theorem 1 (Euler Rotation Theorem). *Any sequence of 3D rotations about a fixed point is equivalent to a single rotation by angle θ along some axis $\boldsymbol{r}$ running through the fixed point.*

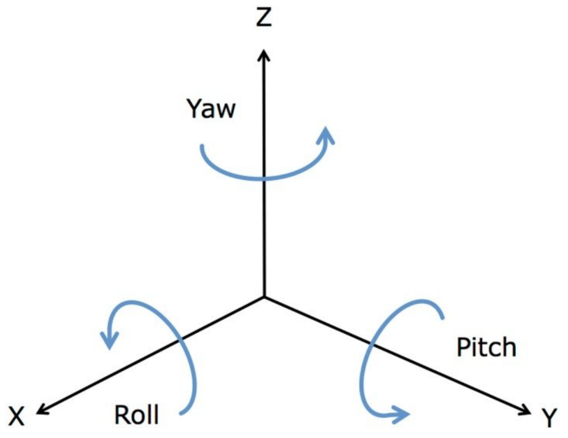


Fig. 4: Roll-pitch-yaw (Euler angles)

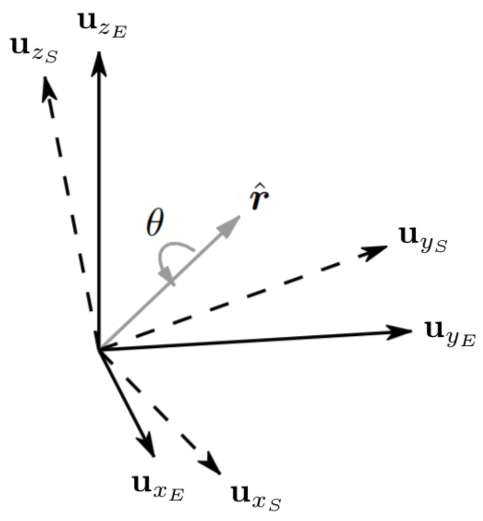


Fig. 5: Euler rotation theorem

According to the Euler rotation theorem, any 3D rotation can be represented by four real numbers θ, r_x, r_y, r_z, where $\mathbf{r} = (r_x, r_y, r_z)$ is a unit vector represented the rotation axis in Euler rotation theorem. The usage of quarternions provide an indirect encoding:

$$[\hat{\mathbf{q}}]_B^A = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix} = \begin{bmatrix} \cos\frac{\theta}{2} & -r_x \sin\frac{\theta}{2} & -r_y \sin\frac{\theta}{2} & -r_z \sin\frac{\theta}{2} \end{bmatrix}, \qquad (1)$$

Originally, the quarternions are an extension of complex numbers: they have a real part and three imaginary parts **i**, **j**, **k**. Their general form, conjugation, norm, and normalization are

$$\mathbf{q} = a + b\,\mathbf{i} + c\,\mathbf{j} + d\,\mathbf{k} \tag{2}$$
$$\mathbf{q}^* = a - b\,\mathbf{i} - c\,\mathbf{j} - d\,\mathbf{k} \tag{3}$$
$$||\mathbf{q}|| = \sqrt{a^2 + b^2 + c^2 + d^2} \tag{4}$$
$$\hat{\mathbf{q}} = \frac{\mathbf{q}}{||\mathbf{q}||} \tag{5}$$

and the three imaginary units have following multiplication rule:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1 \tag{6}$$

Detailed multiplication table is in table 1.

x	1	i	j	k
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

Table 1: Quarternion multiplication.

Although quarternions inherit some structure from complex numbers, their multiplication loses commutativity due to the additional imaginary parts. This makes them more similar to matrices instead of complex numbers in terms of algebraic operations. In the following paragraph, we will give the multiplication of quarternions a different notation to distinguish it from commutative multiplication:

$$\mathbf{q_1} \otimes \mathbf{q_2} \tag{7}$$

The algebraic similarity between quarternions and matrices is not just an analogy. In fact, because the unit quarternions $\hat{\mathbf{q}}$ has a multiplication structure which is isomorphic to 3D rotation group (SO(3)), it has the ability to describe 3D rotation completely, and its rotation arithmetic is identical to that in matrix representation. Like the matrix representation, $[\hat{\mathbf{q}}]^A_B$ itself can be interpreted as the orientation of frame A relative to frame B. Consecutive rotations are simply multiplications, and the reversed rotation is the conjugate quarternion.

$$[\hat{\mathbf{q}}]^A_C = [\hat{\mathbf{q}}]^B_C \otimes [\hat{\mathbf{q}}]^A_B \tag{8}$$
$$[\hat{\mathbf{q}}]^B_A = \left([\hat{\mathbf{q}}]^A_B\right)^{-1} = \left([\hat{\mathbf{q}}]^A_B\right)^* \tag{9}$$

A quarternion contains a scalar part, a, and a vector part, $bi + cj + dk$. The vector part can be interpreted as a 3D vector, and therefore the algebraic operations of the quarternions can also represent 3D vector arithmetic. Representing a 3D vector as a quarternion and transforming it from frame S to frame E with another rotation quarternion $\mathbf{q}_E^S$ can be written as

$$[\mathbf{v}]_S = \left[0\ v_{x_S}\ v_{y_S}\ v_{y_S}\right] = v_{x_S}\mathbf{i} + v_{y_S}\mathbf{j} + v_{z_S}\mathbf{k} \tag{10}$$

$$[\mathbf{v}]_E = [\hat{\mathbf{q}}]_E^S \otimes [\mathbf{v}]_S \otimes \left([\hat{\mathbf{q}}]_E^S\right)^* \tag{11}$$

Furthermore, the inner product and outer product can be defined as

$$\mathbf{u} \cdot \mathbf{v} = \frac{1}{2}\left(\mathbf{u} \otimes \mathbf{v}^* + \mathbf{v} \otimes \mathbf{u}^*\right) \tag{12}$$

$$\mathbf{u} \times \mathbf{v} = \frac{1}{2}\left(\mathbf{u} \otimes \mathbf{v} - \mathbf{v}^* \otimes \mathbf{u}^*\right) \tag{13}$$

In summary, the quarternions integrates 3D rotation and 3D vector operations in a single number system. Most importantly, unit quarternions has the same degree of freedom as 3D rotation, and is proved to be fully capable of representing 3D rotation. Quarternion representation satisfies all our needs, and simplifies the computation of vector and matrix arithmetic as well. Therefore, we adopt it for our project.

4.2 6-axis Data Fusion

The measurement data from the gyroscope and the accelerometer have to be combined to determine the orientation of pointer with higher accuracy. The reason is that we need to integrate angular velocity to obtain rotation angle, and error accumulates in the process of integration. Thus the gyroscope only performs well within a short period, but drifts away from the correct orientation in the long run. On the other hand, the accelerometer is subject to hand movement, so its measurement is not accurate in the short term but is reliable for longer periods.

Madgwick orientation filter is an open-source data fusion method using quartenion representation [4]. Its 6-axis version updates the orientation of the device using both gyroscope and accelerometer readings. The filter keeps track of the orientation of the sensor frame relative to the earth frame, $[\hat{\mathbf{q}}]_E^S$. Here we introduce the update steps of both sensors in Madgwick filter.

Orientation from Gyroscope The derivative of $[\hat{\mathbf{q}}]_E^S$ can be calculated from

$$\frac{d}{dt}[\mathbf{q}]_E^S = \frac{1}{2}[\hat{\mathbf{q}}]_E^S \otimes [\omega]_S \tag{14}$$

Thus the difference between orientations according to the gyroscope $[\varDelta\mathbf{q}_\omega]_E^S$ is written as

$$[\varDelta\mathbf{q}_\omega]_E^S = \frac{1}{2}[\hat{\mathbf{q}}]_E^S \otimes [\omega]_S \cdot \varDelta t \tag{15}$$

$$[\hat{\mathbf{q}'}_\omega]_E^S = [\hat{\mathbf{q}}_\omega]_E^S + [\varDelta\mathbf{q}_\omega]_E^S \tag{16}$$

Orientation from Accelerometer The update from accelerometer tries to minimize the gap between the acceleration direction observed by the sensor $[\hat{\mathbf{a}}]_S$ and the estimation of the gravity direction $[\hat{\mathbf{g}}]_S$ in the sensor frame. The estimation is computed by rotating the fixed gravity direction $[\hat{\mathbf{g}}]_E$ in the earth frame to the sensor frame with some orientation quarternion $[\hat{\mathbf{q}}]_E^S$. So this problem is equivalent to an optimization problem

$$\min_{[\mathbf{q}]_E^S \in \mathbb{R}^4} \mathbf{f}\left([\mathbf{q}]_E^S, [\mathbf{g}]_E, [\mathbf{a}]_S\right) \tag{17}$$

$$\begin{aligned} \mathbf{f}\left([\hat{\mathbf{q}}]_E^S, [\hat{\mathbf{g}}]_E, [\hat{\mathbf{a}}]_S\right) &= [\hat{\mathbf{q}}]_S^E \otimes [\hat{\mathbf{g}}]_E \otimes \left([\hat{\mathbf{q}}]_S^E\right)^* - [\hat{\mathbf{a}}]_S & (18)\\ &= \left([\hat{\mathbf{q}}]_E^S\right)^* \otimes [\hat{\mathbf{g}}]_E \otimes [\hat{\mathbf{q}}]_E^S - [\mathbf{a}]_S & (19) \end{aligned}$$

where $[\mathbf{g}]_E$ is set to be

$$[\hat{\mathbf{g}}]_E = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \tag{20}$$

Madgwick filter utilizes the gradient descent algorithm to solve this optimization problem. The difference between orientations according to the accelerometer $[\Delta\mathbf{q}_\nabla]_E^S$ can be solved as

$$\begin{aligned} [\Delta\mathbf{q}_\nabla]_E^S &= -\gamma \frac{\nabla\mathbf{f}}{||\nabla\mathbf{f}||} & (21)\\ [\hat{\mathbf{q}'}_\nabla]_E^S &= [\hat{\mathbf{q}}_\nabla]_E^S + [\Delta\mathbf{q}_\nabla]_E^S & (22) \end{aligned}$$

with a learning rate $\gamma = \beta\Delta t$ depending on the sample rate Δt.

Combination The combined difference $[\Delta\mathbf{q}]_S^E$ is a simple addition of the difference from both of the sensors:

$$\begin{aligned} [\Delta\mathbf{q}]_E^S &= [\Delta\mathbf{q}_\omega]_E^S + [\Delta\mathbf{q}_\nabla]_E^S & (23)\\ &= \left(\frac{1}{2}[\hat{\mathbf{q}}]_E^S \otimes [\omega]_S - \beta\frac{\nabla\mathbf{f}}{||\nabla\mathbf{f}||}\right) \cdot \Delta t & (24)\\ [\mathbf{q}']_E^S &= [\hat{\mathbf{q}}]_E^S + [\Delta\mathbf{q}]_E^S & (25) \end{aligned}$$

In the last step, $[\mathbf{q}']_E^S$ is not guaranteed to be a valid orientation quarternion: it might not be a unit quarternion. This is the same problem that happened with rotation matrix representation, as the number of parameters is larger than the intrinsic degrees of freedom of 3D rotation. However, normalization easily solves this problem in the quarternion representation.

$$[\hat{\mathbf{q}'}]_E^S = \frac{[\mathbf{q}']_E^S}{||[\mathbf{q}']_E^S||} \tag{26}$$

This step completes the update process in Madgwick filter. In addition, the author of the filter suggests that $0.03 < \beta < 0.1$ performs best in most cases.

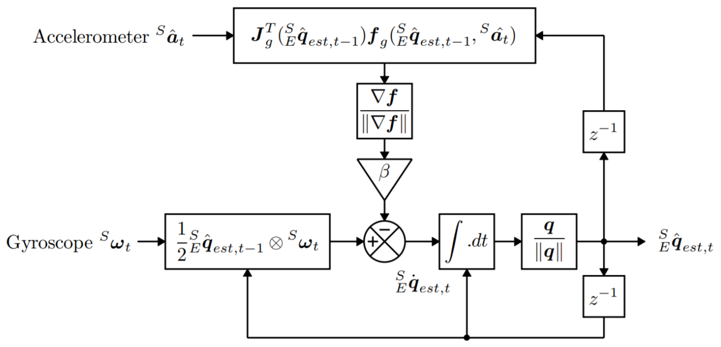

Accelerometer ${}^S\hat{\boldsymbol{a}}_t$
$\boldsymbol{J}_g^T({}^S_E\hat{\boldsymbol{q}}_{est,t-1})\boldsymbol{f}_g({}^S_E\hat{\boldsymbol{q}}_{est,t-1}, {}^S\hat{\boldsymbol{a}}_t)$
$\frac{\nabla \boldsymbol{f}}{\|\nabla \boldsymbol{f}\|}$
β
z^{-1}
Gyroscope ${}^S\boldsymbol{\omega}_t$
$\frac{1}{2}{}^S_E\hat{\boldsymbol{q}}_{est,t-1} \otimes {}^S\boldsymbol{\omega}_t$
$-$
$+$
$\int .dt$
$\frac{\boldsymbol{q}}{\|\boldsymbol{q}\|}$
${}^S_E\hat{\boldsymbol{q}}_{est,t}$
${}^S_E\dot{\boldsymbol{q}}_{est,t}$
z^{-1}

Fig. 6: Summary of Madgwick 6-axis orientation filter.

4.3 Initial Orientation

Madgwick filter provides a simple way to update the orientation of the pointer, but the filter cannot determine the orientation at the beginning. It also assumes that the sensor's z axis is vertical and pointing towards the ground. If this condition is not satisfied, the pointer experiences a strange drifting at the beginning, so we need to initialize the orientation of the sensor properly. Additionally, usage of the pointer should not depend on the pointer's initial direction. Instead, the information that matter are the yaw and pitch angles relative to the initial direction. The roll angle should have no effect on the direction of the pointer.

To achieve these goals, not only should we record the initial orientation of the sensor frame relative to the earth frame $[\mathbf{q}_0]_E^S$, but also a beginning body frame B, which consists of three axes pointing towards the pointer's initial direction $\mathbf{p}_0$, the horizontal direction $\mathbf{h}_0$ perpendicular to both $\mathbf{p}_0$ and gravity $\mathbf{g}$, and the vertical direction $\mathbf{v}_0$ perpendicular to both $\mathbf{p}_0$ and $\mathbf{h}_0$. Unlike the sensor frame S, B should be independent of the roll angle. After defining B, we can transform subsequent measurements of the pointer direction to that frame.

In our settings, we chose $\mathbf{h}_0$ as the x axis, $\mathbf{p}_0$ as the y axis, and $\mathbf{v}_0$ as the z axis for B. This is convenient because the y axis of the sensor is parallel to $\mathbf{p}_0$, so the y axes of S and B are aligned at the beginning. We chose $\mathbf{h}_0$ as the x axis so that the x axes of B and E are aligned as well.

Following the above discussion, our initialization can be divided into two steps: a rotation along the y axis from the sensor frame S to the beginning frame B, and a rotation along the x axis from the beginning frame B to the earth frame E. Given the measurement of the accelerometer in quarternion form,

$$[\mathbf{a}]_S = \begin{bmatrix} 0 & a_x & a_y & a_z \end{bmatrix} \tag{27}$$

the first rotation angle ϕ along y axis is

$$\cos\phi = \frac{a_z}{\sqrt{a_x^2 + a_z^2}}, \quad \sin\phi = \frac{a_x}{\sqrt{a_x^2 + a_z^2}} \tag{28}$$

and the orientation quarternion of the sensor frame S relative to the beginning frame B is

$$[\mathbf{q}]_B^S = \begin{bmatrix}\cos\frac{\phi}{2} & 0 & -\sin\frac{\phi}{2} & 0\end{bmatrix} \tag{29}$$

$$= \begin{bmatrix}\sqrt{\frac{1+\cos\phi}{2}} & 0 & -\text{sign}(a_x)\sqrt{\frac{1-\cos\phi}{2}} & 0\end{bmatrix} \tag{30}$$

Applying the transformation from S to B on acceleration $[\mathbf{a}]_S$, we can get

$$[\mathbf{a}]_B = \begin{bmatrix}0 & a_x' & a_y' & a_z'\end{bmatrix} = [\mathbf{q}]_B^S \otimes [\mathbf{a}]_S \otimes \left([\mathbf{q}]_B^S\right)^* \tag{31}$$

then the second rotation angle θ along x axis is

$$\cos\theta = \frac{a_z'}{\sqrt{(a_y')^2 + (a_z')^2}}, \quad \sin\theta = \frac{-a_y'}{\sqrt{(a_y')^2 + (a_z')^2}} \tag{32}$$

and the orientation quarternion of the beginning frame B relative to the earth frame E is

$$[\mathbf{q}]_E^B = \begin{bmatrix}\cos\frac{\theta}{2} & 0 & -\sin\frac{\theta}{2} & 0\end{bmatrix} \tag{33}$$

$$= \begin{bmatrix}\sqrt{\frac{1+\cos\theta}{2}} & 0 & \text{sign}(a_y')\sqrt{\frac{1-\cos\theta}{2}} & 0\end{bmatrix} \tag{34}$$

Finally, the estimation of the initial orientation quarternion of the sensor frame S relative to the earth frame E is given as

$$[\mathbf{q}_0]_E^S = [\mathbf{q}]_E^B \otimes [\mathbf{q}]_B^S \tag{35}$$

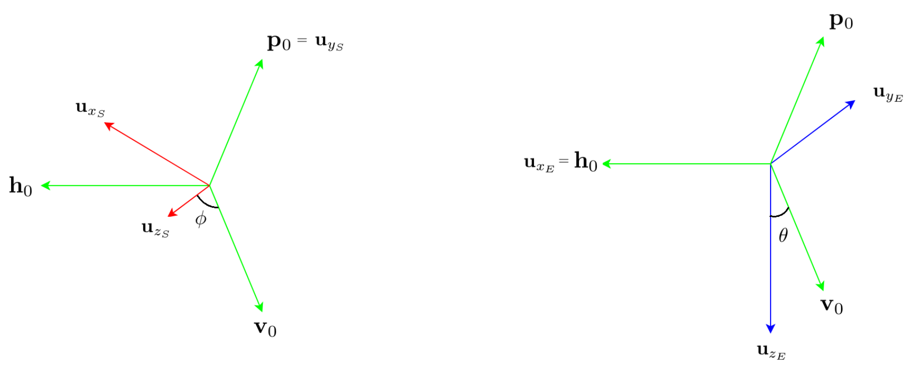
The current orientation of the sensor frame relative to the earth frame $[\mathbf{q}]_E^S$ will be updated by the Madgwick filter at preset intervals, and the new direction of the pointer $\mathbf{p}$ measured in the beginning frame B is

$$[\mathbf{p}]_B = [\mathbf{q}]_B^S \otimes [\mathbf{p}]_S \left([\mathbf{q}]_B^S\right)^* \tag{36}$$

where

$$[\mathbf{q}]_B^S = \left([\mathbf{q}]_E^B\right)^* * [\mathbf{q}]_E^S \tag{37}$$

$$[\mathbf{p}]_S = \begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix} \tag{38}$$



(a) S to B (b) B to E

Fig. 7: The two steps of orientation initialization.

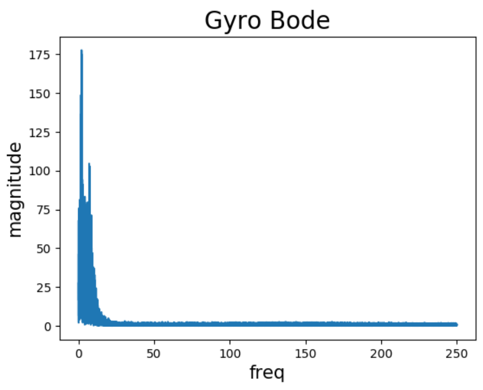


Fig. 8: Frequency magnitude of gyro output data

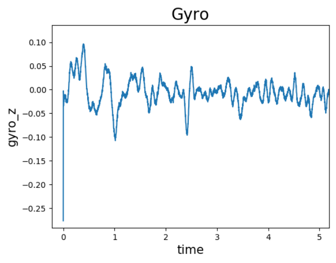
4.4 Vibration Reduction

Vibration of the pointer has two causes: the natural quiver of the hand at low frequencies ($1 \sim 10$Hz), and white noise from electrical circuits present across the whole spectrum. To reduce vibration, we need to apply some sort of filter. However, there exists a trade-off between filter bandwidth and the device response time: lower bandwidth leads to a larger delay.

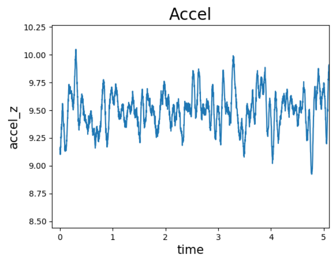
We found that a first order IIR filter nicely solves the problem.

$$\text{output}[n] = (1 - \alpha) \times \text{input}[n] + \alpha \times \text{output}[n-1] \tag{39}$$

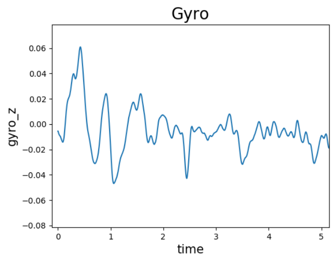
This filter has only one parameter to tune. With proper discount parameter $\alpha = 0.98$, applying this filter on both the gyroscope and the accelerometer reduces vibration up to 70% and creates a negligible 0.1 second delay.



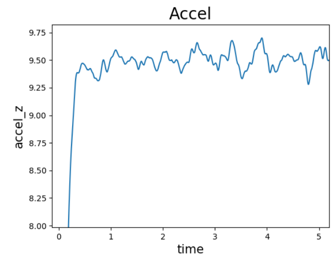
(a) Raw gyro data



(b) Raw accel data



(c) Filtered gyro data



(d) Filtered accel data

Fig. 9: Performance of first order IIR filter on both gyroscope and accelerometer outputs.

4.5 Projection

The final step is to convert the direction of the pointer to a position on the screen. With some testing, we found that it is most comfortable for users when mapping rotations of $\sim \ \pm 20^\circ$ to positions at the edges of the screen. The position is encoded as two real numbers between 0 and 1, with the top-left corner being (0,0), and the bottom-right corner being (1,1). We assume that the pointer is pointed towards the center of the screen when it is activated, so we initialize the position on the screen to

$$\mathbf{s}_0 = (0.5, 0.5) \tag{40}$$

If the pointer is active and its direction relative to the beginning frame is

$$[\mathbf{p}]_B = \begin{bmatrix} 0 \ p_x \ p_y \ p_z \end{bmatrix}, \tag{41}$$

the updated position on screen is

$$\mathbf{s}_0 = (-1.5p_x + 0.5, 1.5p_z + 0.5) \tag{42}$$

If the new position is out of bound, it stays at the screen boundary. To reduce power consumption, the position is updated for every 10 measurements of MPU9250, so the refresh rate is around 50 Hz.

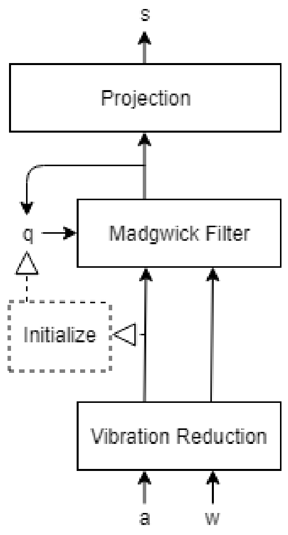


Fig. 10: Summary of orientation computation. Its input is the acceleration measurement $[\mathbf{a}]_S$ and the gyroscope measurement $[\omega]_S$ in the sensor frame. The computation output is a position on the screen.

References

1. Qt Documents, `http://doc.qt.io/`.
2. BlueZ Example Codes, `http://www.bluez.org/`.
3. MPU-9250 Datasheet, `http://www.invensense.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf`.
4. Madgwick, Harrison, Vaidyanathan: Estimation of IMU and MARG orientation using a gradient descent algorithm. In: 9th IEEE International Conference on Rehabilitation Robotics (2011)