Spring 2018

Tracking Pose Using Common Mobile Phone Sensors

Andrew Lee Carlson
Bard College

Tracking Pose Using Common Mobile Phone Sensors

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Andrew Carlson

Annandale-on-Hudson, New York
May, 2018

Abstract

The original goal of this project was to develop an accurate "tape measure" application for Android based mobile phones. The main challenge for such applications is to construct one method that ensures a given Android device can, with accuracy and precision, estimate its own position and movement.

The author developed an application in the Unity 3D game environment in order to understand the challenges of such estimation. This required translation of accelerometer and gyroscope data to the Unity platform. Efficient computation required easy translation from right-handed coordinates to left-handed coordinates. Moreover, it is necessary to design and implement calibration procedures in order to lessen the impact of variation in Android hardware between devices. Once the device has been calibrated, the accelerometer and gyroscope data is synthesized to create an accurate model of the device's position in space.

Contents

Dedication

To my friends and family, for their love and support.

Acknowledgments

I would like to thank my primary advisors for this project, Keith O'Hara in the first semester and Robert McGrail in the second semester. They have been very helpful in pointing me towards valuable resources, and offering advice on which direction the project could move in as it progressed over time. I learned a lot in the course of this project, and these two were very supportive in helping me work through new concepts.

I also need to thank my family, who have always supported and cared for me. I want to thank my girlfriend Jewel, for being my best friend and a great source of support during my years at Bard. I also want to thank my music teachers Ray and Sue Sidoti, who I consider to be family.

Thanks to my friends at Bard, who have greatly enriched my experience here. Finally, thanks to Alexzandra Morris for helping me with this project as it approached completion.

1
Introduction

As computer hardware has become more powerful, engineers have been able to fit higher quality computers into smaller and sleeker form factors. This has led to the rise in popular usage of mobile devices such as smart phones and tablets. Because of their handheld nature, engineers have also added small sensors to these computers, allowing users to interact with them in new and exciting ways. Android devices in particular have become widespread because the open source operating system allows itself to be run on a variety of devices with different hardware. The goal of this project is to develop an application for Android devices which tracks the device's position and orientation in space, or pose, using two sensors commonly found in recently made mobile devices; the accelerometer and gyroscope.

Most mobile devices released within the past few years include a built in three axis accelerometer and three axis gyroscope. An accelerometer is a sensor which measures the forces exerted on it, and a gyroscope measures its rotational velocity. The raw data from these sensors can be processed and fused to produce a mathematical model of the device's position and orientation in space.

The application described in this work was developed in the Unity 3D game engine. This choice was made because the Unity game environment comes with a variety of tools that facilitate the modeling of objects in three dimensional space. Another potential benefit is Unity's consistency across platforms; once this application is developed for Android devices, it could be easily adapted to work on other devices which have the same sensors and support the Unity environment, such as iOS devices.

1.1 Related Work

Previously at Bard College, another Computer Science project by Blagoy Yordanov Kaloferov addressed a similar problem in 2013. The project, "Recreating the Trajectory of a Golf Swing Using a Microelectromechanical System," used a custom device made with an arduino, gyroscope, accelerometer, and a magnetometer to measure the path of a golf club as it is swung by a golf player. In contrast, for this project we will be using commercially available hardware with no assembly or modification needed by the end user. Additionally, the nature of the motion will be different; a golf swing is a short, quick motion, but for our test we will be moving the device in a slow and careful manner, which may affect how well the sensors are able to detect changes in the device's motion.

1.2 Summary

The rest of the paper is organized as follows. Chapter 2 introduces the main challenges in measuring pose using our chosen sensors. Chapter 3 explains the mathematical models that will be used to model and update the devices pose. Chapter 4 details the specifications of the devices used in testing, as well as an overview of the Unity game environment. Chapter 5 explains how the mathematical model from Chapter 3 is adapted to be implemented in Unity. Finally, Chapter 6 contains concluding thoughts, and suggestions for future work.

2
Challenges in Measuring Pose

Although hardware varies by device, many mobile devices now have some sensors in common. Typical sensors include gyroscopes, accelerometers, cameras, proximity sensors, and compasses, to name a few. For this project, we will be examining how precisely a user might be able to interact with such a device simply by moving it, so we will be using the gyroscope and accelerometer sensors. We will create an application which will closely track and model the movement of the device in three dimensions, and then compare the path that the device traveled with the path calculated in the model.

2.1 Modeling Pose in Three Dimensions

In order to analyze the accuracy and precision of the application, we will need an efficient way to mathematically represent pose in three dimensions. A pose is an object's position and orientation in relation to a reference frame. For our purposes, the mathematical representation we choose will need to be able to quickly combine a series of incremental changes in pose, in order to calculate the device's current pose.

2.2 Calibration

When processing the accelerometer data, the data will have to be integrated twice in order to calculate the device's displacement from its original position. A consequence of this process is that any small errors in the accelerometer data will be exaggerated after integration. To deal with this problem, we will have to come up with a simple calibration procedure which reduces error as much as possible, while still being simple enough to carry out that a user would be able to perform it consistently and with little or no extra equipment.

2.3 Testing

To test the accuracy of the model, we will move the device running the application along a predetermined path. Then, we will graph both the path of the device and the path calculated by our model, calculate the difference between the two paths.

3
The Mathematics of Graphics

3.1 Representing Pose in Unity

In Unity 3D, every scene is a separate three dimensional virtual space. In this virtual space, GameObjects are the base class for all entities that exist in this space. Every GameObject has an associated Transform object, which contains information relating to the position, rotation, and scale of the object relative to both the world space and local space. World space refers to the global coordinate system associated with every scene, and local space refers to the local coordinate system of an object's parent GameObject. Position in both of these coordinate systems is represented as a set of Cartesian coordinates on the x, y, and z axes. Orientation is represented as a quaternion, which is a complex number commonly used to represent rotations.

3.1.1 Quaternions

Quaternions are not the most intuitive representation of rotation, especially when compared to alternatives such as Euler angles. However, they have some mathematical properties which are very useful for manipulating rotational data.

Quaternion Structure

Quaternions are complex numbers, represented as $\dot{q} = w + ix + jy + kz$, where $i^2 = j^2 = k^2 = -1$, and w, x, y, and z are all real numbers. An axis-angle, by comparison, represents rotation as a rotation angle θ around a vector $v = (x_a, y_a, z_a)$. The relation between a quaternion and an axis-angle rotation, (v, θ) representing the same rotation is $\dot{q} = (w_q, x_q, y_q, z_q) = (\cos\frac{\theta}{2}, x_a \sin\frac{\theta}{2}, y_a \sin\frac{\theta}{2}, z_a \sin\frac{\theta}{2})$ [2].

Quaternion Product

The Hamilton product of two quaternions, denoted as $\dot{q}_1 \otimes \dot{q}_2$, represents two rotations being applied in succession. A product of two quaternions can be calculated as such:

$$\dot{q}_1 \otimes \dot{q}_2 = \begin{bmatrix} w_1 & x_1 & y_1 & z_1 \\ -x_1 & w_1 & -z_1 & y_1 \\ -y_1 & z_1 & w_1 & -x_1 \\ -z_1 & -y_1 & x_1 & w_1 \end{bmatrix} \begin{bmatrix} w_2 \\ x_2 \\ y_2 \\ z_2 \end{bmatrix} \tag{3.1.1}$$

An important property to note of quaternion products is that they are non commutative, meaning that $\dot{q}_1 \otimes \dot{q}_2$ will yield a different result from $\dot{q}_2 \otimes \dot{q}_1$ [1].

Quaternion Conjugate

Every quaternion also has a conjugate, denoted as $\ominus\dot{q} \mapsto \dot{q}^{-1}$. The conjugate of a quaternion is defined as $\dot{q}^{-1} = (w, -x, -y, -z)$. The conjugate can be thought of as the reversed rotation of a quaternion. The product of a quaternion and its conjugate $\dot{q} \otimes \dot{q}^{-1}$ will result in no rotation, because we are essentially applying a rotation and subsequently undoing that same rotation.

Benefits of Quaternions

There are two ways of interacting with rotations that are supported by Unity; quaternions and Euler angles. However, even though a user may manipulate an object's rotation as an Euler angle, an object's rotation is alway stored internally as a quaternion, because they are more computationally efficient than using Euler angles. Additionally, Euler angle

representations of rotation suffer from the singularity problem, similar to the gimbal lock problem encountered by mechanical gyroscopes.

An Euler angle represents a three dimensional rotation as a series of two dimensional rotations around an object's x, y, and z axes, which we can represent as the orthonormal rotational matrices R_x, R_y, and R_z respectively. We can define an Euler angle as such:

$$R = R_z(\psi)R_x(\phi)R_y(\theta) \tag{3.1.2}$$

$$= \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

It is important to note that just like the quaternion product, these rotations are non-commutative. In Unity, an Euler angle will first rotate ψ degrees around the z axis, ϕ degrees around the x axis, and then θ degrees around the y axis [4]. Given the rotation matrices for two Euler angles, we can multiply the matrices to apply two rotations in sequence ${}^{B}R = {}^{A}R\,{}^{A}_{B}R$, similar to the Hamilton product of two quaternions. However, combining rotations in this way is more computationally intensive than if we represented the same rotations as quaternions; the product of two Euler angle rotational matrices takes 27 multiplications and 18 additions to compute, while the product of two quaternions takes 16 multiplications and 12 additions [1].

The other problem with Euler angles is that they can run into the singularity phenomenon. Singularity occurs when the middle rotation is an odd multiple of 90°. For example, consider the case where $\phi = 90°$ for some rotational matrix $R = R_z(\psi)R_x(\phi)R_y(\theta)$ which represents the rotation from frame A to frame B. In this case, we can apply the identity [1]:

$$R_x^T(90°)R_z(\vartheta)R_x(90°) \equiv R_y(\vartheta) \tag{3.1.3}$$

$$R_z(\vartheta)R_x(90°) \equiv R_x(90°)R_y(\vartheta)$$

In this case, R_x^T represents the transpose of a rotational matrix, such that ${}^{A}_{B}R_x^T = {}^{B}_{A}R_x$.

Substituting this identity into R, we get

$$R = R_x(90^\circ)R_y(\psi)R_y(\phi) = R_x(90^\circ)R_y(\psi + \phi) \tag{3.1.4}$$

The effect of this is that we lose a degree of freedom in our rotation, because rotations around the z and y axes both rotate around the same axis in this instance.

3.1.2 World Space and Local Space

In Unity 3D, the global coordinate system is referred to as world space. In this coordinate system, the x and z axes are both parallel to the ground and perpendicular to each other, and the y axis extends upward perpendicular to both the x and z axes. Every GameObject's Transform object has a position and orientation relative to world space, stored as `transform.position` and `transform.rotation` respectively.

Each GameObject also has a local coordinate system, which is referred to as that object's local space. In the local coordinate system, there are still three perpendicular axes, x y and z, but the origin of this coordinate system is located at the object's position in world space. Additionally, the coordinate system is rotated relative to world space by the rotation stored in `transform.rotation`.

A GameObject can have another GameObject set as its parent. Suppose we have two objects A and B, such that A is the parent of B, denoted here as $B \in A$. In this case, the pose of B in world space is the composition of the pose A in world space, and the pose of B in the local coordinate system of A, such that ${}^W_B P = {}^W_A P \oplus {}^A_B P$. Whenever an object's position is altered, it's local position is updated to match the new global position, and vice versa. The same applies to an object's rotation and local rotation. Additionally, if A changes its pose, the local pose of B, ${}^A_B P$, will remain the same, but its pose in world space ${}^W_B P$ will be recalculated with the new pose of A.

By default, if no parent is set for a GameObject, then the object's pose in local space is the same as its pose in world space. Additionally, while each GameObject can only have

one parent, there is no limit on the number of child objects any one GameObject may possess.

3.2 Translating Sensor Data to Model Movement

3.2.1 Combining Poses

For this project we will have a constant stream of data from the gyroscope and accelerometer, representing incremental changes in pose in the local coordinate system. We will have to continually incorporate this data to update the world position of the device every frame. We can represent a pose P as $P = (t, \dot{q})$, where t is the position of the object represented as a set of Cartesian coordinates, and $\dot{q}$ is the orientation of the object relative to the origin, represented as a quaternion. For each frame, we can use the following equation to calculate the new position of the device [1]:

$$P = P_1 \oplus P_2 = (t_1 + (t_2 * \dot{q}_1), \dot{q}_1 \otimes \dot{q}_2) \tag{3.2.1}$$

Where P_1 is the pose of the device in world space calculated in the previous frame, and P_2 is the change in pose of the device in its local coordinate system. In this application, P_1 will be the pose in world space in the previous frame, and P_2 will be the change in pose in local space between P_1 and the current frame, and P is the updated pose in world space in the current frame. This means that $(t_2 * \dot{q}_1)$ represents the change in displacement in world space.

3.2.2 Calculating Change in Displacement

In order to calculate the change in displacement since the last frame, first we calculate the velocity of the device by integrating the device's acceleration:

$$\vec{v}(n) = \int_0^n \vec{a}(n)\mathrm{d}t \tag{3.2.2}$$

Where n is the number of samples taken by the accelerometer up to this point. However, since the device will be changing orientation, we need to calculate the velocity in world space; otherwise, if the device rotates during movement, it will carry over its previous velocity on its local axes, making the model move in a different direction than the device.

One way to approximate velocity is the equation:

$$\vec{v}(n) = v(n-1) + \vec{dv_W}(n) \tag{3.2.3}$$

Where $v(n-1)$ is the velocity in the previous frame, and $\vec{dv_W}$ is the change in velocity in world space. We can calculate the vector representing the local change in velocity, $\vec{dv_L}$, by simply multiplying $\vec{a}(n)$ by the time since the last frame dt. Once we have $\vec{dv_L}$, we have to rotate the vector by the devices orientation in order to find the change in velocity in world space, $\vec{dv_W}$. Then, we can simply add $\vec{dv_W}$ to the velocity vector $\vec{v}(n-1)$ from the previous frame to calculate $\vec{v}(n)$.

Once we have the velocity in world space, we simply need to multiply that velocity by the time since the last frame dt, to find the change in displacement in world space, $\vec{d}$. This change in displacement approximates $(t_2 * \dot{q}_1)$ from our pose composition equation 3.2.1.

4
Technology

4.1 Unity

Unity is a cross-platform game engine developed by Unity Technologies. A game engine is a software development environment designed to facilitate the construction of video games. Aside from video games, Unity is also used to create simulations on computers, consoles, and mobile devices. In Unity, a developer can create a Scene, which is a three dimensional virtual space, and import 3D models of objects into the scene. Once a model, or `gameObject`, is in the Scene, a script can be attached to the object. Then, the script can access user input, and affect attributes of the `gameObject` it is attached to, as well as other `gameObjects` within the same Scene. Scripts are written in C# or in Unityscript, a variation of Javascript. In this project, scripts are written exclusively in C#.

4.2 Device Used

The following table details the technical specifications of the hardware for the device used during the tests. [3]

Table 4.2.1: Nexus 5 Specifications.

Screen	4.95” 1920 x 1080 display (445 ppi) Full HD IPS Corning Gorilla Glass 3
Size	69.17 x 137.84 x 8.59mm
Weight	4.59 ounces (130g)
Cameras	1.3 MP front facing 8 MP rear facing with Optical Image Stabilization
Memory	16GB or 32GB (actual formatted capacity will be less) 2GB RAM
Processing	CPU: Qualcomm Snapdragon? 800, 2.26GHz GPU: Adreno 330, 450MHz
Sensors	GPS Gyroscope Accelerometer Compass Proximity/Ambient Light Pressure Hall Effect
Battery	2,300 mAh non-removable battery Standby time: up to 300 hours Talk time: up to 17 hours Internet use time: up to 8.5 hours on Wi-Fi; up to 7 hours on LTE Wireless Charging built-in
OS	Android 6.0.1

5
Implementation

The following chapter details how the mathematical models used to represent pose were adapted for implementation in the Unity3d environment. The final update method can be found in the Appendix in section 8.1.1.

5.1 Quaternion Conversion

In Unity, the data output from the accelerometer and gyroscope provides us the orientation and acceleration of the device in a right handed coordinate system; that is, a coordinate system where the x axis extends to the right of the screen, the y axis extends upward from the top of the screen, and the z axis extends outward perpendicular to the screen. This is a problem, because in Unity everything is modeled using a left handed coordinate system, meaning that the direction of the y and z axes are swapped from what it would in a right handed coordinate system. The simplest way to correct this is to invert the z component, and switch the y and z components in the quaternions and vectors returned by the gyroscope and accelerometer.

5.2 Calculating the Acceleration Vector

In Unity, we have access to an array of AccelerationEvent objects from Input.accelerationEvents. This array holds every acceleration measurement since the last frame, as well as the time since the last measurement for each acceleration vector in the array. We can calculate the average acceleration in the local coordinate system since the last frame using an iterative loop, found in the Appendix in section 8.1.2.

This calculates the average acceleration measured by the device since the last frame. We use an average here because it helps to lessen the impact of sudden jumps in the accelerometer readings.

5.3 Calibration

Commercial IMU (Inertial Measurement Unit) sensors are often not calibrated very well by default, so the first step will be to ensure that the sensor readings are as accurate as possible. Fortunately, Unity provides us with methods that already integrate gyroscope measurements with a reasonable degree of accuracy. With the accelerometer, we are not so lucky; we only have the raw accelerometer data, so we will have to account for errors such as gain and bias, as well as remove the gravity forces from the acceleration vector.

5.3.1 Bias and Gain

We can account for errors in raw sensor readings by using the equation

$$c = Ku - b \tag{5.3.1}$$

where c is the calibrated sensor reading, determined by the sensor gain K, the uncalibrated reading u, and the sensor bias b [7]. Sensor gain is a sensitivity variable, so changing the gain variable adjusts the sensitivity of the sensor readings. Sensor bias is a constant variable representing any constant offset in the uncalibrated sensor readings. However, for

the purposes of calibration, I will use a slightly different form of this equation. If we let $b = Ki$, then we can rewrite equation 5.3.1 as $c = Ku - Ki$. This will be useful because when we calibrate the sensor, we can solve for K and i individually much more easily than we could solve for K and b.

5.3.2 Calibration Method

Given that the average user does not have access to machines that can move their phones at a consistent velocity, we will try to base our calibration method off of a series of stationary poses that the user can position their phone in to gather the parameters for our calibration equation. For each of the three axes of the accelerometer, we will take the following steps.

When the accelerometer experiences no acceleration, such as when sitting stationary with the given axis parallel with the ground, we can easily solve for i:

$$0 = Ku - Ki$$

$$Ki = Ku$$

$$i = u \tag{5.3.2}$$

Then, we can solve for K by positioning the phone so that the axis is now perpendicular to the ground. The accelerometer output is in G force units, where one G force is the force exerted by earth's gravity, so the calibrated output should be -1 G force.

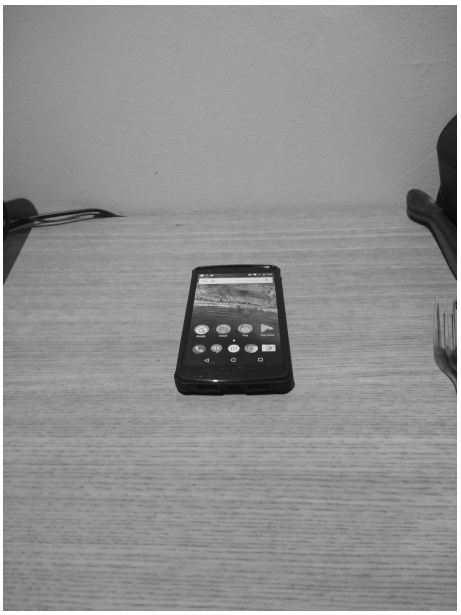
$$-1 = Ku - Ki$$

$$-1 = K(u - i)$$

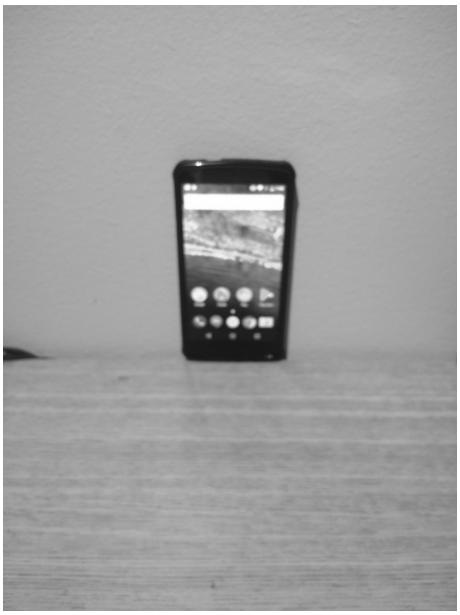
$$K = -1/(u - i) \tag{5.3.3}$$

5.3.3 Poses

The axes relative to the phone are aligned so that if the screen is upright and facing you, the x axis points directly to the right, the y axis points directly up, and the z axis points out of the screen perpendicular to the plane of the screen. For each pose in the calibration process, the calibration program will run for ten seconds, summing all of the acceleration vectors during that time, and then dividing that by the number of samples taken to get the average acceleration recorded on each axis. Using this average, we can calculate the K and i variables of our calibration equation.

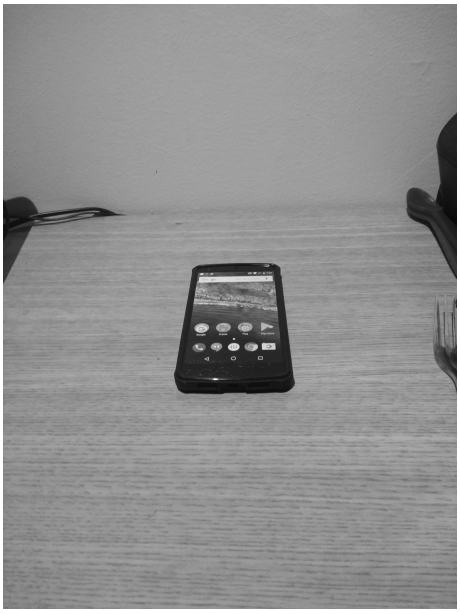


(a) In the first pose, the phone is placed laying down face up on a flat horizontal surface, such as a floor or level table. Using the acceleration vector, $\vec{a}$, of the average acceleration from this sample, we can calculate the i variable for the x and y axes, since both axes are perpendicular to the force of gravity in this position, and therefore should experience no acceleration.



(b) In the next pose, the phone is positioned "standing up," meaning that the phone is positioned such that it rests on the bottom of the phone screen, with the screen perpendicular to the ground. A wall or heavy block would be ideal for this pose. This is the most unstable of all the poses, so care must be taken to ensure that the phone is as vertical as possible. Using the average acceleration $\vec{a}$ of this sample, we can calculate the i variable for the z axis, since it is perpendicular to the force of gravity in this position, and therefore should experience no acceleration, and we can calculate K for the y axis, as it should be experiencing -1 G.

(a) The third pose is similar to the second, but instead of resting on the bottom of the phone screen, the phone is resting on the left side of the screen, again with the screen perpendicular to the ground. Similarly, care should be taken to ensure that the phone is stable and as vertical as possible. Using the average acceleration $\vec{a}$ of this sample, we can calculate the K variable for the x axis, as it should be experiencing -1 G.



(b) The last pose is the same as the first. This time, since we have the i variable for the z axis, we can now calculate the K variable for the z axis.

Figure 5.3.2: Calibration Steps

5.4 Removing Gravity

To estimate the position of the device, we first have to remove the acceleration due to gravity from the accelerometer data. The accelerometer returns a vector in local space, but we can use two empty GameObjects, `gravPoint` and `gravOrigin`, to simultaneously model the vector in local space and world space. In the method `RemoveGrav(Vector3 accelInput)` from Appendix section 8.1.3, `gravOrigin` is set as the parent of `gravPoint`, and `gravOrigin` is set to a position of $(0, 0, 0)$ in world space. By setting the local position of `gravPoint` to the acceleration vector, and setting the orientation of `gravOrigin` equal to the device's current orientation, we effectively rotate the acceleration vector so that the position of `gravPoint` in world space is now equal to the acceleration vector in world space. Then, we can simply add 1 to the y component of the vector (gravity exerts a force of -1G on the global y axis), so that the local position and global position of `gravPoint` represents the acceleration vector in local space and world space, respectively, without the influence of gravity.

5.5 Velocity Threshold

Even after calibration, the accelerometer is still prone to small errors when stationary, which can lead to a nontrivial amount of drift when the device is left stationary. To address this issue, we apply a threshold to the calculated local velocity vector dv_L, as detailed in the method 8.1.4 in the Appendix, such that if the magnitude of any one component of the vector falls below the threshold amount, that component will be set to zero. Here, we use $0.001G * s$ as our threshold, so we will discount any acceleration less than the threshold. The unit $G * s$ means G forces times time in seconds, which we use here because the threshold is applied before the conversion from $G * s$ to m/s.

5.6 Velocity Window

One problem encountered during development of this application was that the device would not detect sharp decelerations when the device stopped moving; that is, it could detect when the device started moving, and when the device moved in a different direction, but when the device stopped moving, it could not detect a sharp acceleration that would bring the velocity back to zero. Because of this, the model would continue to move at a constant velocity even when the device was brought to a standstill.

The compromise to this problem was to use a moving window, using the last 64 calculated dv_W values to calculate the current velocity, resulting in the following approximation of the velocity equation 3.2.3:

$$\vec{v}(n) = \sum_{i=(n-64)}^{n} d\vec{v_W}(n) \tag{5.6.1}$$

This compromise allowed the program to integrate acceleration over short periods of time, enough for most quick short movements a user might make within the range of their own reach, and it will cause the velocity to regress to zero when the accelerometer no longer indicates movement in any direction, such as when lying on a table or floor after having been moved around by the user.

In the application, the window and the integration was maintained using a `Vector3` object called `velocity` to hold the integrated velocity, and an array of `Vector3` objects called `vWindow` to hold the past 64 values of dv_W. Both `velocity` and all the `Vector3` objects in `vWindow[]` are initialized to `(0,0,0)` at the beginning of each measurement. The velocity threshold method is listed in Appendix section 8.1.5.

6
Application Evaluation

6.1 Experiment

For the experiment, we moved the device in a half-circle arc, of radius 0.5m. A picture of the path that the device traced can be found in figure 6.1.1. In the tests, we moved the device along the path starting at the left end of the semicircle and ending at the right end, all the while keeping the device's local x axis as parallel as possible to the tangent of the path at any given point. As the device is moved, the application writes the following information to a text file in every frame: the device's position on the global x and z axes, as well as its orientation around the y axis. We only log the position in the x and z axes and the orientation around the y axis, because in the test the device does not move along the y axis, nor does it rotate around the x or z axes. Once we have this information, we determine the accuracy of the trial by comparing the final position vector calculated by the model with the vector representing the device's real position in relation to its starting point.

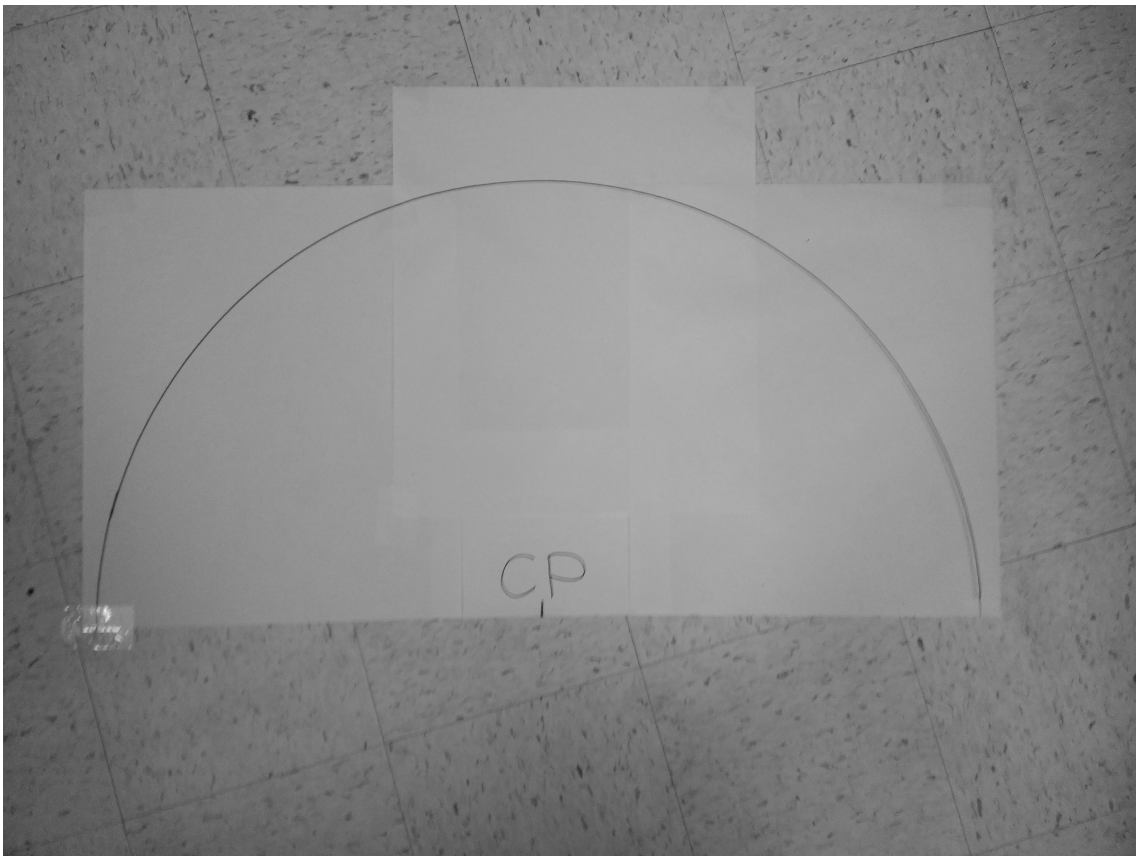


Figure 6.1.1: The path that the device followed during testing.

6.2 Results and Analysis

The experiment was run a total of 16 times, and the data is organized in the following tables. Table 6.2.1 holds the final displacement from the starting point, in world space coordinates, calculated by the application. Table 6.2.2 holds initial orientation of the device around the y axis θ, and the final displacement of the phone in real space. The final position in real space should be negative one meter in the local z axis of the device from the starting position. To calculate the final position in real space, we use the equation $(x, z) = (\cos(\theta + \pi), \sin(\theta + \pi))$. Table 6.2.3 holds the percent error calculated in each trial for each axis, as well as the average percent error for both axes in each trial. The percent error for a trial was calculated as $E = |\frac{(s_r - s_m)}{s_r}|$, where s_r is the displacement on the given axis in real space, and s_r is the displacement on the given axis calculated by the model. Figures 6.2.1 through 6.2.3 depict histograms of the error calculated in each axis for the 16 trials that were conducted using the methodology from section 6.1. Figure 6.2.4 depicts the error calculated in each axis, as well as the average error for each trial in a column chart.

Table 6.2.1: Model Final Displacement Per Trial.
This vector represents the final position of the device as calculated by the application.

Trial	**Model Final X Position (m)**	**Model Final Z Position (m)**
1	-0.8489108	0.4057785
2	-0.2229519	0.2225117
3	-0.3422129	0.5514016
4	-0.5277649	0.7251852
5	-1.032179	-0.01985528
6	-0.9716508	0.85658
7	-0.2344769	-0.04385906
8	0.0535818	-0.1542997
9	-0.3942275	-0.0961595
10	-0.3441142	0.4647673
11	-0.4832019	-0.2587929
12	-0.4089997	-0.1077248
13	-0.406965	-0.2025759
14	0.4144926	-0.2419026
15	-0.4653716	-0.3411021
16	0.2594578	0.7042051

Table 6.2.2: Final Displacement Per Trial.
This vector represents the final position of the device as measured manually in the experiment

Trial	**Model beginning θ (rad)**	**Final X Position (m)**	**Final Z Position (m)**
1	-1.274081627	-0.29238005	0.9563022045
2	-1.251439471	-0.3139560004	0.9494375334
3	-1.287847039	-0.2791888715	0.9602362074
4	-1.232874404	-0.3315272568	0.9434456412
5	-1.276785142	-0.2897936073	0.9570891626
6	-1.269735757	-0.2965332408	0.9550225322
7	-1.220332468	-0.3433335072	0.9392135555
8	-1.240377574	-0.3244391577	0.9459065667
9	-1.289742467	-0.2773683129	0.9607636645
10	-1.272643476	-0.2937550545	0.9558807289
11	-1.262295419	-0.3036306585	0.9527898106
12	-1.275542468	-0.2909827335	0.9567283046
13	-1.254710218	-0.3108489567	0.9504593238
14	-1.228266734	-0.3358708076	0.9419080638
15	-1.26852799	-0.2976864698	0.9546636925
16	-1.230711941	-0.3335666463	0.9427265205

Table 6.2.3: Percent Error by Trial.
The error calculated between the application model's final position and the device's actual final position

Trial	**X Axis % Error**	**Z Axis % Error**	**Average % Error**
1	190.3449807	57.56796355	123.9564721
2	28.98625931	76.56383994	52.77504963
3	22.57397587	42.57646236	32.57521912
4	59.19200885	23.134395	41.16320192
5	256.1772841	102.0745486	179.1259163
6	227.6701112	10.30787535	118.9889933
7	31.70579186	104.6697644	68.18777812
8	116.5152075	116.3123616	116.4137846
9	42.13141216	110.0086529	76.07003251
10	17.14324392	51.37810755	34.26067574
11	59.14134047	127.161594	93.15146723
12	40.55806511	111.2597066	75.90888585
13	30.92049733	121.3134739	76.11698562
14	223.4083435	125.682188	174.5452658
15	56.32944296	135.7300799	96.02976143
16	177.7828967	25.30123162	101.5420642

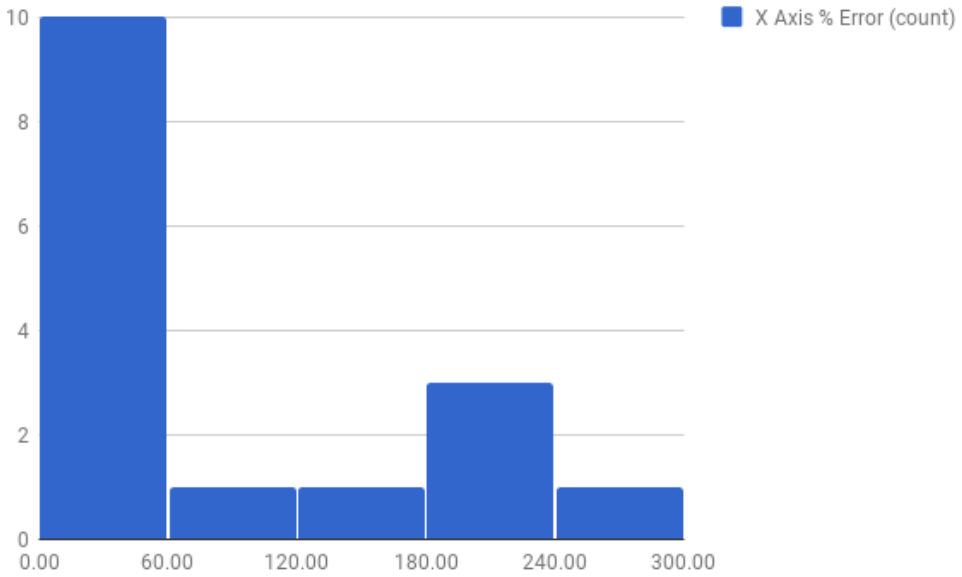


Figure 6.2.1: Histogram of the X axis percent error for 16 trials: a visual representation of columns 1 and 2 from Table 6.2.3
The average percent error in the X axis is 98.78630385%

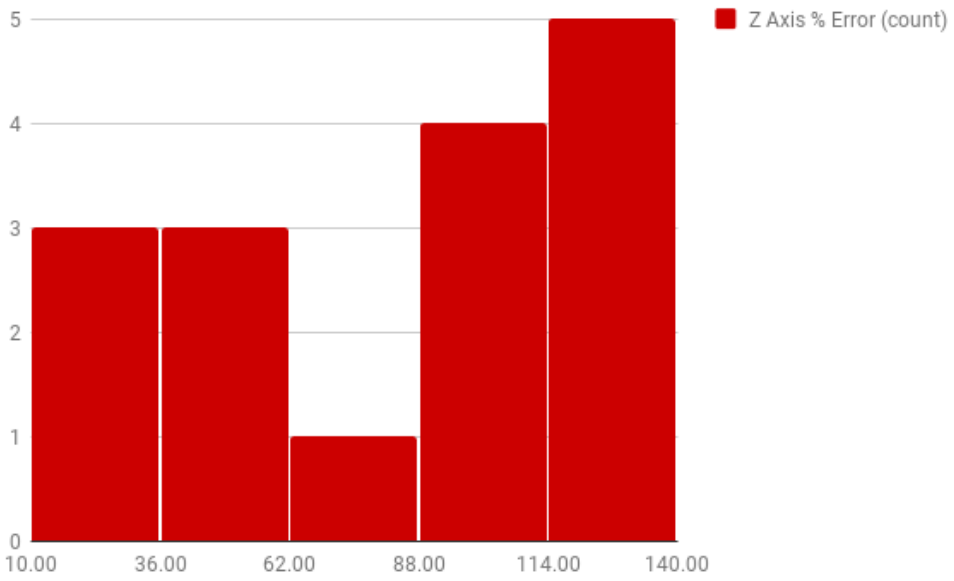


Figure 6.2.2: Histogram of the Z axis percent error for 16 trials: a visual representation of columns 1 and 2 from Table 6.2.3
The average percent error in the Z axis is 83.81514033%

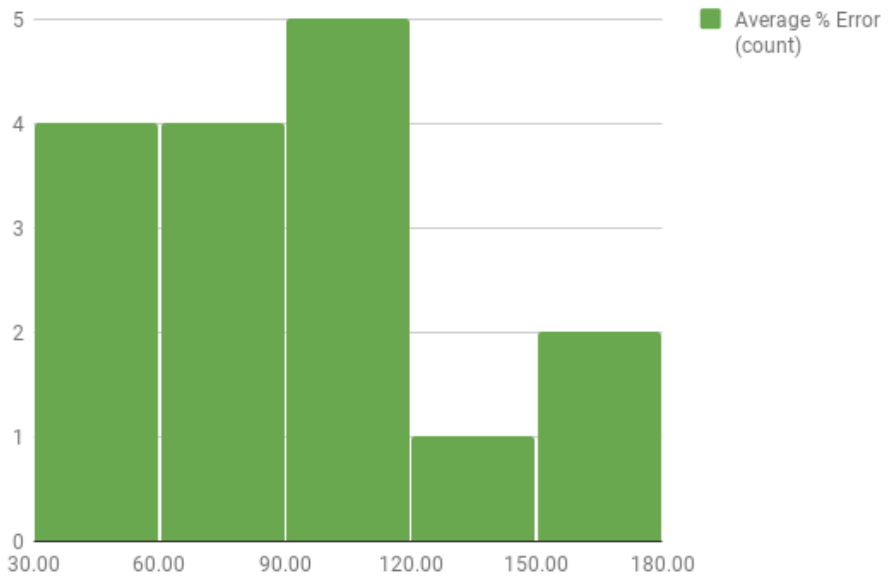


Figure 6.2.3: Histogram of the average error in both axes for 16 trials: a visual representation of columns 1 and 2 from Table 6.2.3
The average percent error for both axes is 91.30072209%

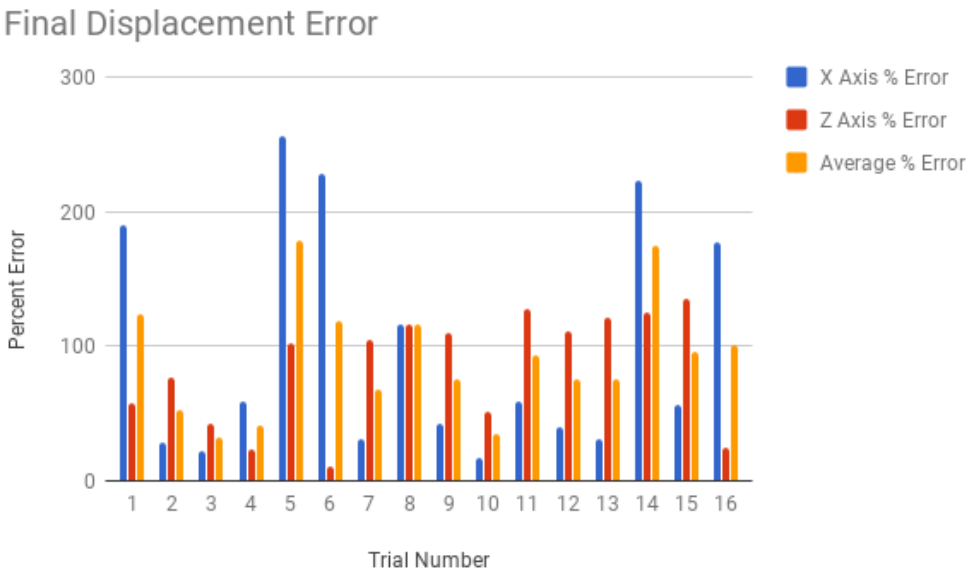


Figure 6.2.4: Error calculated in each trial. Average error is the average of the error in the X and Z axes for that particular trial.

6.3 Discussion

The application is able to calculate the final displacement of the device with an average percent error of 91.3%. With an average error so close to 100%, the position tracking in this application is not very useful in practice. Part of the problem comes from the fact that after the device is brought to rest, the application predicts that the device is still moving in a similar motion for a moment after it is brought to rest, and so the model continues to move in an arc even after the device stops moving. Another limitation is the hardware itself; during testing, it was discovered that the device would not pick up on sharp changes in acceleration well. It was for this reason that we had to implement the velocity window to bring the velocity vector back to zero once the device stopped moving, because the device would not detect the sharp change in acceleration as the device was brought to a standstill. The sensors are not reliable then at very high accelerations, but at low accelerations small errors have a larger proportional impact on any estimates of position derived from the sensor data, especially when the sensor readings rise just above the threshold we impose on the raw data. Having unreliable readings at both high and low acceleration values makes it very difficult to obtain good data for position estimation, especially since such estimation requires double integration of the accelerometer values, compounding any errors in the sensor readings.

7
Conclusion and Future Work

In this project an application was developed for Android devices to measure the displacement of a device using internal sensors. This application had rather poor accuracy, drastically limiting its practical use. However, there are a number of ways that this application, or one similar to it, could be modified or improved for future study. One route would be to research and implement a variety of different methods of processing and filtering sensor data, so that the different approaches could be compared side by side. Another area for improvement would be to experiment with different ways of combining position and orientation data to represent a device's pose, to see if this had any effect on the accuracy of the estimated position of the device.

Another way to improve the accuracy would be to account for the location of the accelerometer sensor within the device. The reason that this is important, is that if the gyroscope is sensing that the device is rotating, but the accelerometer detects no acceleration, then the device must be rotating around the point where the accelerometer is located. In order to create a more accurate model of the device's pose, we would want to take into account the dimensions of the device, as well as making the location of the accelerometer

the center of rotation for the model. We had tried to incorporate this information into the model for the application developed in this project, by using the accelerometer and gyroscope sensors to calculate the distance from the accelerometer sensor to an edge of the device by rotating the device around that edge. We tried to calculate these values within the application, because the location of the sensor is not readily available information that the application could obtain from the device, and there are too many devices running Android to keep a comprehensive list containing this information for all devices that could run the application. However, this aspect of the model had to be abandoned, because the calculated distance was not accurate enough to be reliably used to locate the accelerometer within the device. Should another method for modeling pose produce more reliable and accurate results, it may be able to incorporate this information into the model in order to further increase the accuracy of the model.

8
Appendices

8.1 Code Excerpts

8.1.1 Complete Update Method

```
// Update is called once per frame
void Update () {

   if (collect) {
      // Find average acceleration
      Vector3 accel;
      Vector3 accelMean = Vector3.zero;
      int count = 0;
      float dt = 0;
      foreach (AccelerationEvent accEvent in Input.accelerationEvents) {
         accel = accEvent.acceleration;
         accelMean += accel;// * gforce;
         count++;
         dt += accEvent.deltaTime;
      }

      accelMean = accelMean / count;

      // c = K(u - I)
      Vector3 c = Vector3.Scale ((accelMean - I), K);

      // Find local dv vector
      Vector3 dv = RemoveGrav (c) * dt;
      // Apply threshold
      dv = VelocityThreshold (dv) * gforce;
```

```
        // Change local dv vector to global coordinates
        Vector3 gdv = LocalToGlobal (dv);

        // add global gdv to velocity window
        velocity -= vWindow [n];
        vWindow [n] = gdv;
        velocity += vWindow [n];
        n++;
        if (n >= vWindow.Length)
           n = 0;

        // integrate global velocity to get global change in position
        Vector3 deltad = velocity * dt;

        // add change in position to global position
        displacement += deltad;

     }

     // update orientation
     transform.rotation = GetOrientation ();

     // Display current orientation and displacement
     SetDisplacementText (displacement);
     SetOrientationText (transform.rotation.eulerAngles);

     if (collect)
        LogData ();

//   transform.rotation = ConvertRotation(gyro.attitude);
   }
```

8.1.2 Acceleration Vector

```
int count = 0;
Vector3 accelMean = Vector3.zero;
Vector3 accel;
foreach (AccelerationEvent accEvent in Input.accelerationEvents) {
   accel = accEvent.acceleration;
   accelMean += accel;
   count++;
}
accelMean = accelMean / count;
```

8.1.3 Remove Gravity Vector

```
   Vector3 accel;
```

```
  Quaternion pose = GetOrientation ();

  // set gravPoint.transform.localPosition to the vector
  // representing the accelerometer input
  gravPoint.transform.localPosition = new Vector3(-accelInput.x, accelInput.z,
      -accelInput.y);

  // rotate the vector around gravOrigin by the pose
  gravOrigin.transform.rotation = pose;

  // gravPoint world position is now equal to the
  // acceleration in world axes

  // subtract gravity from the world y axis
  // note: accelerometer data is measured in G forces,
  // so gravity will exert a force of -1 G forces on the
  // world y axis
  accel = new Vector3(gravPoint.transform.position.x,
     gravPoint.transform.position.y + 1,
     gravPoint.transform.position.z);
  gravPoint.transform.position = accel;

  // gravPoint.transform.localPosition is now the accelerometer
  // data without gravity
  accel = gravPoint.transform.localPosition;

  return accel;
}
```

8.1.4 Velocity Threshold

```
  public Vector3 VelocityThreshold(Vector3 v) {
     Vector3 vt;
     if (v.x > -0.001 && v.x < 0.001)
        vt.x = 0;
     else
        vt.x = v.x;
     if (v.y > -0.001 && v.y < 0.001)
        vt.y = 0;
     else
        vt.y = v.y;
     if (v.z > -0.001 && v.z < 0.001)
        vt.z = 0;
     else
        vt.z = v.z;
     return vt;
  }
```

8.1.5 Velocity Window

```
// Update is called once per frame
void Update () {

   ...
   velocity -= vWindow [n];
   vWindow [n] = gdv;
   velocity += vWindow [n];
   n++;
   ...

}
```

Bibliography

[1] Peter Corke, *Robotics, Vision and Control: Fundamental Algorithms in Matlab*, Springer, Berlin, 2013.

[2] Steven M. LaValle, *Virtual Reality*, Cambridge University Press, 2017.

[3] Google, *Nexus tech specs* (2018), https://support.google.com/nexus/answer/6102470?hl=en.

[4] Unity Technologies, *Unity - Scripting API* (2018), https://docs.unity3d.com/ScriptReference/index.html.

[5] ———, *Unity User Manual (2017.3)* (2018), https://docs.unity3d.com/Manual/UnityManual.html.

[6] Blagoy Yordanov Kaloferov, *Recreating the Trajectory of a Golf Swing Using a Micro-electromechanical System*, http://digitalcommons.bard.edu/senproj_s2013/48.

[7] Sebastian O.H. Madgwick, *Automated calibration of an accelerometers, magnetometers and gyroscopes - A feasibility study*, http://x-io.co.uk/res/doc/automated_calibration_feasibility_study.pdf.