ECE 498: IMU 3D Dead Reckoning

Paul Pok Hym Ng[a]
[a]ECE, University of Illinois at Urbana-Champaign, ppng2@illinois.edu

Abstract

The goal of this project is to perform 3D dead reckoning using phone IMU sensors. These sensors include the accelerometer, gyroscope and magnetometer. Using only these three inputs the output of the algorithm should be a 3D plot of the trajectory of the device through 3D space.

Keywords: IMU, Dead Reckoning, Trajectory,

1 Introduction

This paper shows our algorithm, implementation, and methodology when developing our ideas. We exploit the following core idea to reach the goal of 3D dead reckoning.

In elementary physics and math we learn that integrations and derivatives represent the total distance and rate of change that an object or subject is moving at. Therefore the core idea is to do the same but with IMU sensors located on the phone.

A phone has accelerometer, gyroscope and magnetic field sensors that allow us to achieve this goal. And from that it is possible to double integrate acceleration to obtain the distance.

This is in fact an oversimplification of how we achieved our goal. We also need to account for the direction the object is traveling through space and correctly adjust the orientation when it changes. Thus it is necessary to use the other sensors mentioned above. In doing so we have the major components all assembled. Orientation and distance. From there we only need to plot the location.

The current system is not set up as a real time system, but it is possible to use the current system with a few tweaks to fit the needs of a real time system. Namely the following modifications must be made. Firstly, distance calculations should be made at the same time an orientation change is made. Secondly, we need to add routines to reset orientation based upon the status of the sensors (magnitudes). [5] For example, in A^3 which reset orientation using accelerometer and magnetometer if the gyroscope reads values close to 0.

There are multiple applications for such a technology and we shall mention a few below.

1. **Indoor Localization**: With dead reckoning we could effectively plot out where a user is in an indoor environment without the aid of GPS. This allows one to offer location based services indoor at a high accuracy and granularity where it would not have been able to be done before.
2. **Automotive Applications**: Dead reckoning is currently deployed in automotive systems to overcome the limitations of GPS/GNSS. A case where such technology may be applied is to update the map of a user's vehicle when they are in a tunnel or in a parking lot.
3. **Games**: Dead reckoning is used to predict the location of a player in online games. This is done mostly due to the nature of networked connections which may drop/lose packets at any point in time. This is especially true as most games use UDP as re-transmissions of past information is undesirable. Therefore to make up for lost packets, dead reckoning is used.

2 Problems with MEMS Sensors

Due to the nature of construction of MEMS sensors being cheap they lead themselves to inaccuracies. These manifest themselves in multiple ways. MEMS sensors

are based on a non-pendulous design and thus cross-axis coupling and pendulous error from vibrations are insignificant. This is further supported by the fact that in fact most MEMS accelerometers are 3 single-axis accelerometers and thus lead to low cross-axis coupling. This gives us the following equation for a single measurement along any axis for an accelerometer [4]

$$\hat{a_x} = a_x + S_x a_x + B_f + n_x$$

Where $\hat{a_x}$ represents the estimated value (returned by sensor), a_x represents the actual value, $S_x a_x$ a polynomial scaled error term to include non-linear artifacts, B_f a DC bias, and n_x representing random noise.

Similarly we can draw a similar conclusion for gyroscope input.

$$\hat{\omega_x} = \omega_x + S_x \omega_x + B_f + n_x$$

Once again we ignore cross coupling artifacts and in addition only include the *first order* scaling factor S_x

What does this all mean? This means that in the real world we should ideally do some form of pre-processing or compensation in our data to help mitigate these effects. There is must literature on modeling and mitigating such effects and is discussed in [4] [3].

3 Data Collection

While we were provided data to test with, we thought it would be prudent to collect our own data and test some of our own test cases.

The data was collected using an android phone namely the, Xiaomi Mi Mix 2S. Below is the methodology used for collecting the data. It uses Android library functions however the pseudo code is provided

```
function onSensorChanged(){
    if(sensor == ACCELEROMETER){
        save_time(); update_new_acc_data();
        update_old_gyro_data();
        update_old_mag_data();
    } else if(Sensor == GYROSCOPE){
        save_time(); update_new_gyro_data();
        update_old_acc_data();
        update_old_mag_data();
    } else{
        save_time(); update__new_mag_data();
        update_old_gyro_data();
        update_old_acc_data();
    }
}
```

```
while(ButtonNotPressed){
    onSensorChanged()
}
writeDataToFile(data)
```

In the above, `onSensorChanged` will update data arrays that store the current session's data in arrays depending on the sensor data recieved. In addition it will also save a timestamp (in nanoseconds) that it received the data. Upon a on-screen button being pressed we save the data to a file local to the device. It will be output in CSV format and has the following output style.

```
time,aX,aY,aZ,wX,wY,wZ,mX,mY,mZ
```

This methodology however is flawed even if it works relatively well. Because accelerometers and the gyroscopes sample at different rates and accelerometers sample much more consistently, sometimes a timestamp difference (between two consecutive samples) that is negative occurs. This appears to be because the `onSensorChanged()` function seems to act like a priority queue and thus it is possible for certain sensors to return values before others even if they occurred before/after.

While this problem was not resolved we were able to mitigate this and actually obtain a reasonable result despite this. The solution was to average the rate at which data was collected, in essence we averaged the difference in the timestamps between all samples and took that as the rate data was sampled. This is not the ideal solution as the best solution is generate different timestamps for each different sensor update and then pick the data point which corresponds with the closest timestamp difference.

We will discuss the algorithm later but here is an image of a B that we collected data for and plotted.

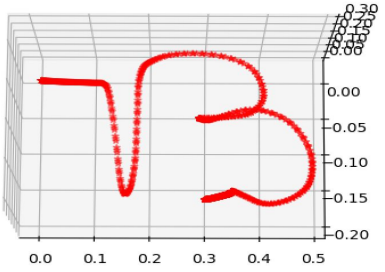

Figure 1: Self B

This was not a problem we encountered with the provided data.

4 Quaternions and Why Use Them

Quaternions are another way of representing rotations. Originally rotations were represented using rotation matrices and Euler angles. However these serve as an alternative that is better.

Compared to rotation matrices, they are more compact and avoid gimbal lock.

Firstly, a rotation matrix is a $3x3$ matrix representing nine floating point integers. A quaternion is represented by 4 numbers

$$a + bi + cj + dk$$

where i, j, k are the fundamental quaternion units. This means a space savings in memory critical systems. Moreover the number of computations of vector rotating operations is less meaning regardless of the system representation or base vector/matrix multiplication algorithm used there are less operations to be done [1].

Rotation matrices can experience a phenomenon known as gimbal lock where pitch and yaw can represent the same motion. In certain applications especially aerospace this could be deadly.

5 Basic Quaternion Usage

As discussed above, a quaternion can be represented via 4 numbers.

$$a + bi + cj + dk$$

Quaternion rotations are done via multiplying two quaternions together. Note that quaternion multiplication is not associative. The quaternion multiplication results in a quaternion that is perpendicular in the perpendicular direction.

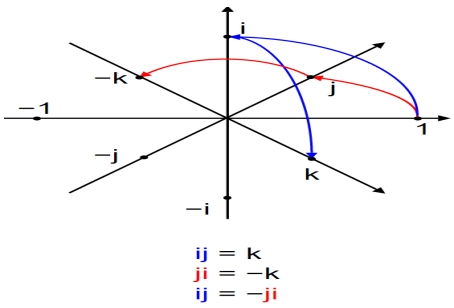

Figure 2: Quaternion Rotation Results (From Wikipedia)

Note: A eye matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

is the equivalent of the quaternion vector

$$\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

6 Calulation of Orientation with Quaternions

We took inspiration from the scikit-kinematics library [2] and their method of calculation of quaternions to represent rotations. It is done with $\omega, q_0, time_array$ as input. ω represents the angular velocity, q_0 represents the original orientation in quaternion form. $time_array$ represents the times stamps provided with the data. This is is done because the rate at which data was collected from the phone varies due to the library function `onAccuracyChanged()`.

The calculation uses the following formulas to calculate $\Delta\vec{q}$. This will ultimately reflect the change in orientation for one time slice. To obtain the orientation at any point in time we need to multiply together all the previous Δq to obtain the current orientation.

$$\Delta\vec{q}_i = \frac{\vec{\omega}(t_i)}{|\vec{\omega}(t_i)|} sin(\frac{|\vec{\omega}(t_i)|\Delta t}{2})$$

and $q(t)$ current orientation represented by

$$q(t) = q(t_0)\Delta q(t_1)\Delta q(t_2)...$$

This now gives us the orientations we need to use in the final position calculation.

However there is some problem with this methodology. This is essentially "integrating" the gyroscope to obtain the current orientation. This proves problematic because of the problems with MEMS sensors mentioned above and as such this calculation will eventually diverge. To help mitigate this drift we do two things. Firstly, we calculate two different sets of quaternions (one discussed above, second discussed below) and merge them together remembering to normalize the vector after calculation. Secondly we scale our results by a scalar value to help mitigate this drift.

7 Calculation of Orientation with Madgwick Quaternions

Why use Madgwick's implementation? One thing that this algorithm is very good at is having good

accurate quaternion calculations at a wide range of sampling rates ranging from anything from 10Hz. With this calculation we can be more sure that the error introduced via the quaternion calculation derives itself from either the data/drift and not the algorithm which was unsuited to calculation over such a wide variation in sampling rates. In fact it was documented on the scikit-kinematics documentation [2] that the quaternion calculation method mentioned above actually requires a "high enough" sampling rate. Given that their default setting is a 100Hz it is fair to assume that this is the absolute minimum that that algorithm is reasonably accurate for.

The idea of the algorithm is as follows. A quaternion representation of orientation can be reformated as an optimization problem. Given the sensor readings $\hat{s}_S$ in the sensor frame which define a direction, a reference direction in the earth frame $\hat{d}_E$, and a starting orientation $\hat{q}_{SE}$ the orientation of earth relative to the sensor the optimization problem is as follows. Note, $\hat{d}_E$ and $\hat{s}_S$ have 0 for their first vector value and are of the same shape as the quaternion 1 by 4. Also note $\hat{q}^*$ is the conjugate quaternion and reverses the frame of reference.

$$\min_{\hat{q}_{SE} \in \mathbb{R}^4} f(\hat{q}_{SE}, d_E, s_S) = \hat{q}^*_{SE} \times \hat{d}_E \times \hat{q}_{SE} - \hat{s}_S$$

With the following definition defining the quaternion required to rotate v_A into frame B

$$v_B = q_{AB} \times v_A \times q^*_{AB}$$

we can see the above objective function reverses that and minimizes the difference between the estimated next quaternion update minus the measured direction in the sensor frame should be minimized. Essentially giving us the best orientation that matches the sensor readings. The author Madgwick then optimized this function via gradient descent (as it is fast enough for real time implementation. This gives us the formulation for the next update step for a quaternion. Namely the following equation.

$$q_{SE,t} = \hat{q}_{SE,t-1} - \mu_t \frac{\nabla f}{||\nabla f||}$$

Where the objective function is defined as where J is the Jacobian matrix of f.

$$\nabla f(\hat{q}_{SE}, d_E, s_S) = J^T(\hat{q}_{SE}, d_E) f(\hat{q}_{SE}, d_E, s_S)$$

With this we obtain a way to get the quaternion at every time step.

8 Fusing Quaternions

In order to maintain the correct magnitude (and as such the correct amount of rotation), we scaled one quaternion by a value $v \leq 1$ and scaled the other by $1 - v$. Afterwards we normalized the quaternion back to the correct magnitude.

9 Algorithm

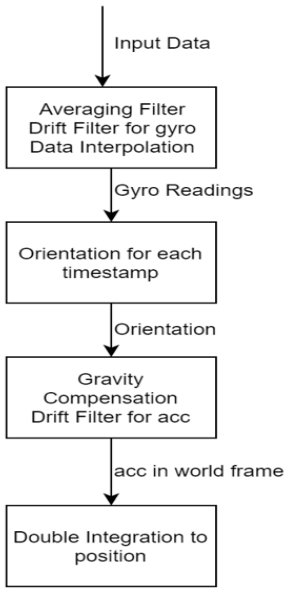

Figure 3: Algorithm Flowchart

The idea is quite simple. As one can see from the flow chart given the input we use an averaging filter to help compensate for some of the noise that exists in the input filter. Secondly, if the data input are not all of the same length we interpolate each sensor data array to have the same length as the longest one. While this is not ideal as we are "guessing" inputs, this is required for the calculation of quaternions as we need some data for each timestamp.

Then we calculate the orientation using the method described above with both the scikit-kinematics method [2] and the Madgwick method [3].

We compensate for gravity in the acceleration by subtracting the average of the first few samples in the input data. However this notion is based upon the assumption that the object starts from a **standstill**. If the user is not standing still then this subtraction will fail and induce heavy bias into the calculation. In addition we also take the mean of the newly gravity compensated acceleration, scale it by a value (we use 0.7) and subtract it to compensate for drift.

Finally we perform double integration on the acceleration and apply the orientation change every time to obtain our position vector.

10 Results

Here are some images of the results that we had.

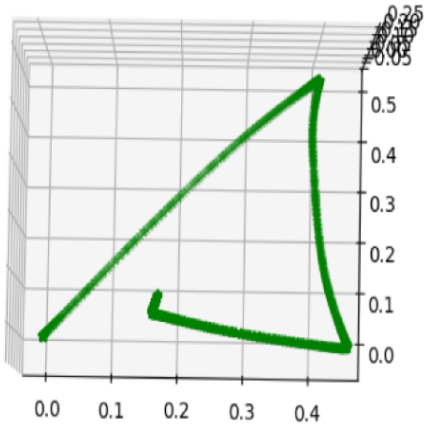

Figure 4: 2D Triangle: 66.5cm x 56.5cm x 35.5cm

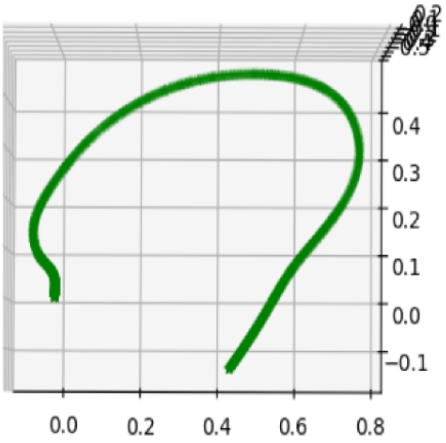

Figure 5: 3D Circle: r = 27

11 Problems

We experience exponential blowup in the Z direction. And after weeks, and much tinkering we are still unable to explain why such a problem exists. Please see figure 6 and 7 for an example of this. It is obvious that the Z direction calculated should not have traveled 0.8 meters as we are actually static in the Z axis.

In addition we have had both in our own collected data, and provided data a tendency to develop a "tail" at the end of the data. If one observes figure 2 we can clearly see a tail at the closing of the B.

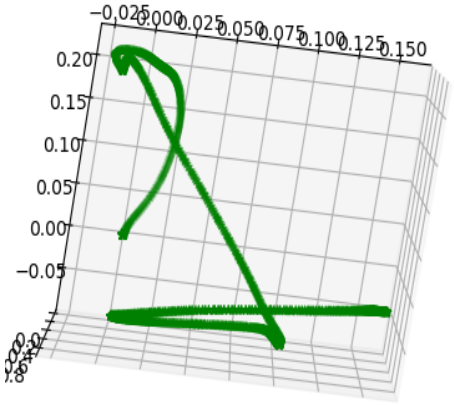

Figure 6: 2D 1

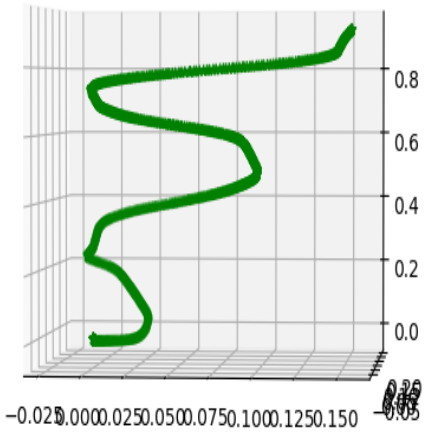

Figure 7: 2D 1 Viewed on Z axis

12 Conclusion

3D IMU Dead Reckoning is still an open problem and even though current implementations are not perfect have many applications in the real world.

This specific implementation while flawed, has promise and reasonable results especially in the 2D space. In addition with a bit of tweaking namely, moving the distance calculation into the orientation calculation phase.

References

[1] D. Eberly, Rotation representations and performance issues.

[2] T. Haslwanter, scikit-kinematics.

[3] S. O. Madgwick, An efficient orientation filter for inertial and intertial/magnetic sensor arrays.

[4] M. Park, Y. Gao, Error and performance analysis of mems-based inertial sensors with a low-cost gps receiver.

[5] P. Zhou, G. Shen, Use it free: Instantly knowing your phone attitude application to approximate reasoning.