ROCO503 IMU Report

Demetrius Zaibo	Tom Queen	Daniel Gregory-Turner
10324064	10286169	10451001

Introduction:

This report details solutions to the tasks set out in the "ROCO503 IMU Coursework 2017-18" document.
Each task is broken down into four sections:

- Problem statement – Outlines the task boundaries and interpretation
- Hypothesis – Provides a theoretical assessment of the problem and the expected results of the task
- Methodology – Details the step-by-step procedures performed to collect the data for analysis
- Results – Assesses the collected data, comparing and contrasting where appropriate

Finally, the report is concluded with a summary of the tasks performed by each team member. The two appendices include a breakdown of main algorithms used to assess data, and the code itself.

Task 1: Noise Analysis

Problem Statement:

- Quantification of noise present within the Phidget 21 Spatial 3/3/3 1044 along all 3 axes for the accelerometer and gyroscope sensors.
- Identification of dominant frequency bands causing aforementioned noise.
- Demonstrate the effects of filtering with respect to dead-reckoned positional data, contrasting against the stationary ground truth.

Theoretical Assessment:

MEMS accelerometers and MEMS gyroscopes are subject to various sources of error. A non-exhaustive list of such errors being:

- Manufacturing quality, manifesting as systematic errors such as a DC bias; non-uniform scaling between sensors; non-linearity in measurements and susceptibility to other error sources
- Flicker noise, or bias random walk, acting as a low frequency component
- Thermo-mechanical white noise, providing high frequency components
- Temperature & pressure, causing numerous mechanical, geometrical and fluid dynamic properties to change; can be represented as a DC or low frequency offset, and modelled as a polynomial function
- Power supply noise can cause artefacts in the signal across the frequency spectrum; depending on the severity, such as electro-static discharges or power surges, it may also cause damage to the sensor
- The Earth, causing effects such as a DC offset within gyroscope as the Earth's rotation is measured; scaling issues across large distances due to the non-uniform gravitational pull; additional noise depending on how much cosmic radiation is absorbed by the atmosphere or focussed by the Earth's magnetic field

The IMU used for these experiments also happens to be a digital IMU, meaning the data has been discretised. This results in two more sources of error, specifically the quantisation error of the signal and the quantisation error of the timestamp for that signal.

Null Hypotheses:

- The frequency response for a stationary system will comprise solely of low frequency or DC components within all three axes for both the accelerometer and gyroscope sensors
- Filtering the raw data will provide no significant improvement in system performance

Methodology:

For each accelerometer axis:

- Align the accelerometer axis with gravity and clamp the IMU to a sturdy table
- Collect data for 2 minutes and 30 seconds and discard the first 30 seconds worth of data using a shielded cable for data transfer

Using the collected data, for each accelerometer and gyroscope axis:

- Average all of the data to approximate the DC bias
- Generate a standard deviation for all of the data to quantify the spread of the noise
- Perform a Fourier transform and identify any major frequency peaks for filtering
    - Compare filtered data with unfiltered data

Results:

Figure 1 is an example set of Fourier transforms for each sensor throughout a stationary IMU test. As can be readily seen, much of the noise was distributed across all frequencies, slightly attenuating with higher frequencies (aka, pink noise). Also viewable is interference from the power supply at 50Hz, along with several other spikes around the 75Hz range. Two further peaks occur at around 23Hz and 32Hz within the accelerometers y-axis. Finally, all sensors have a large DC noise component.

Based on Figure 1 and the other frequency domain plots gathered, it was estimated that a 10Hz low pass filter should be applied to the accelerometers, and a 20Hz-30Hz bandpass applied to the gyroscope values. Several filters were tested (Chebyshev, Inverse Chebyshev, Bessel, and Butterworth); however, a Butterworth filter of order 4 was selected in the end as it appeared to provide the best overall results. Figure 2 shows the results of these filters, including the reduction in noise standard deviation and bias removal for gyroscopic data.

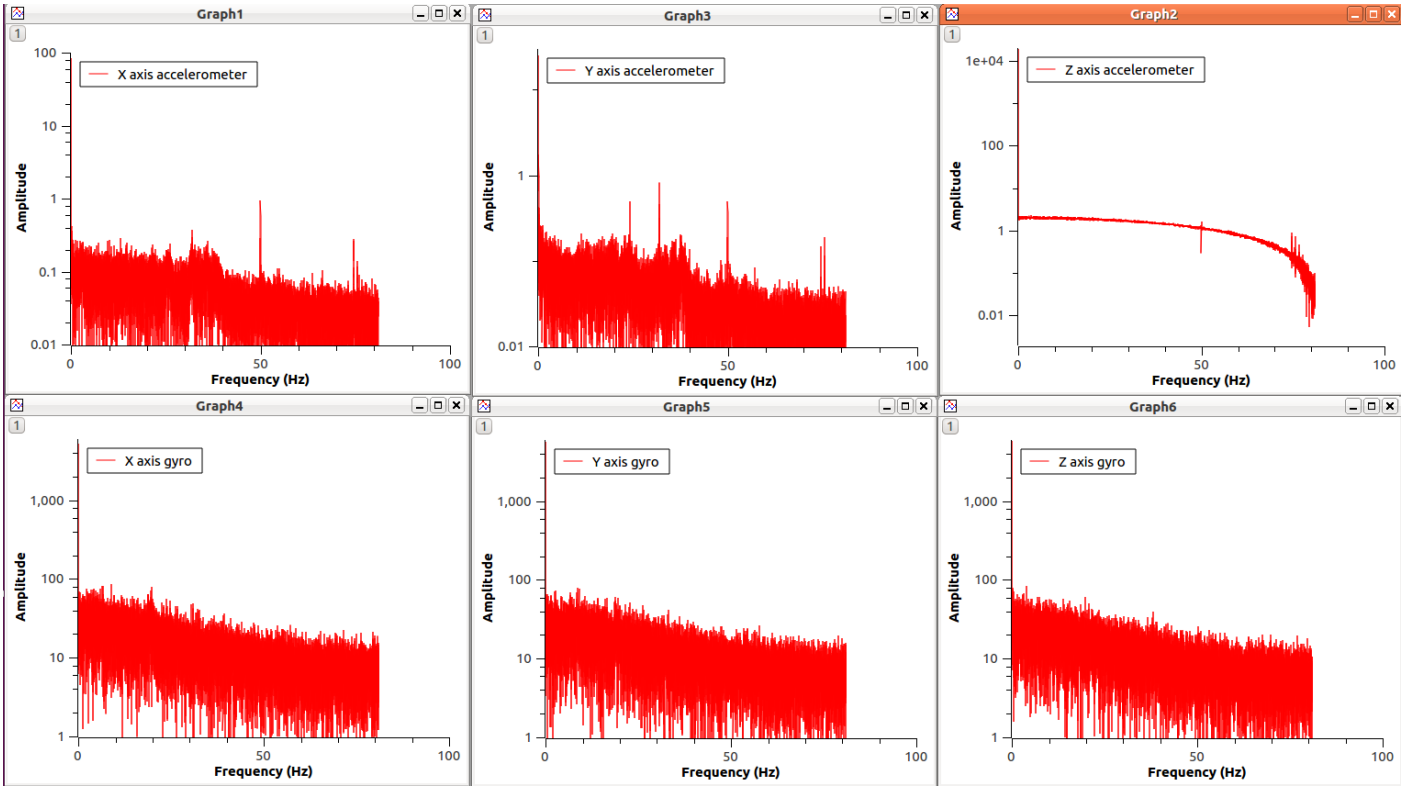

Figure 1: Frequency analysis of IMU sensors. Top row shows accelerometer X, Y & Z axes. Bottom row shows gyroscope X, Y & Z axes.

Task 2: Filtering Effects

Problem Statement:

- Observe differences in dead-reckoned positional data with respect to known motion patterns, specifically:
    - Pendulum motion with known mass and length
    - Vertical motion up and down
    - Z-based motion along a horizontal plane, aided with a smooth surface
- Compare and contrast aforementioned positional data for filtered and unfiltered datasets

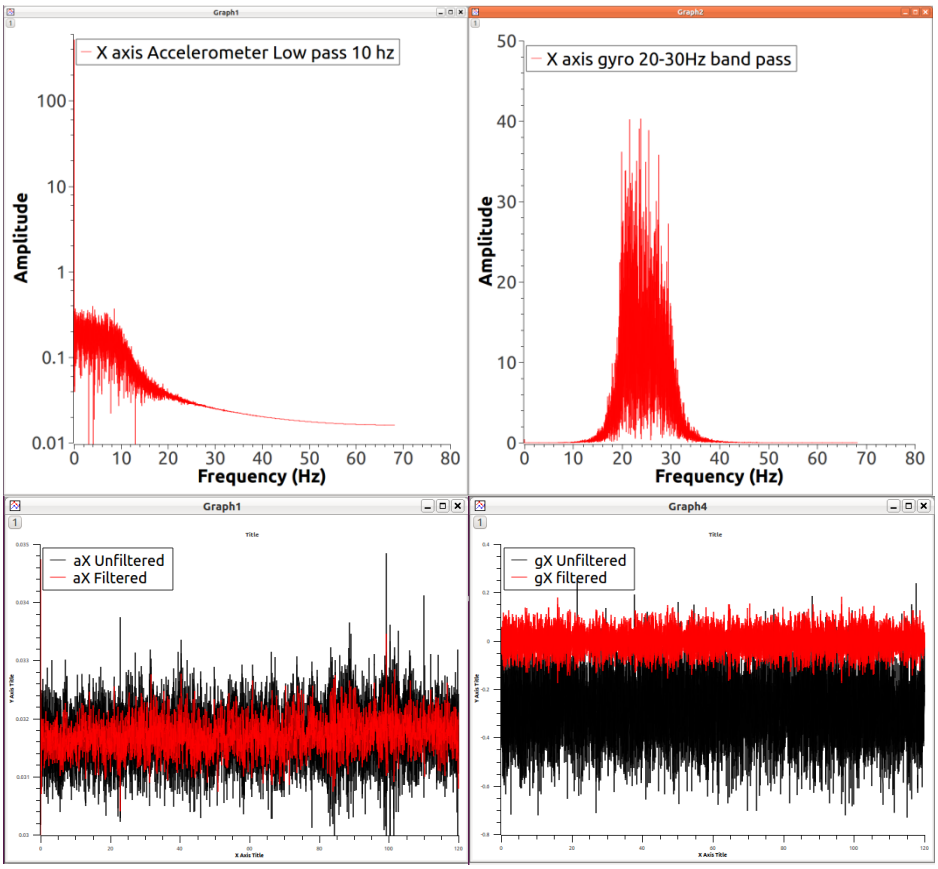

Figure 2: Left: X-axis accelerometer with 10Hz low pass filter. Right: X-axis gyroscope with 20-30Hz bandpass filter.

Theoretical Assessment:

Task 1 provided insight into how noise filtering affects system reliability for a stationary setup. However, IMU's are often used for mobile systems, such as within smartphones and other such tracking devices. To begin assessing the efficacy of the filters it is necessary to analyse the system for constrained motions.

Each of the three motions provides a test bed for approximating the direction of gravity. When averaged over a large enough timeframe, all three setups can be modelled as a stationary point. As such, when compared the frequency responses to task one, any additional frequency components will be related to desirable motion to be detected.

Null Hypotheses:

- Frequency components of the motion data will not be readily detectable
- Filtered data will not drift from the raw data

Methodology:

Pendulum:

The setup in figure 3 is comprised of a single axis rotary joint levelled such that no translational movements occur whilst it is swinging. Additional mass was added such that the pendulum section weighs 500 grams. The centre of mass is approximately 25 centimetres below the hinge joint. The system was left to swing until it stopped.

Figure 3: Pendulum setup

Vertical Motion:

To ensure a fixed axial motion up and down the IMU was connected to a pillar drill. The drill was then manually moved up and down at regular intervals.

Z-Shaped Horizontal Movement:

Masking tape was used to outline the Z-shape to be moved over. The IMU was, whilst secured to solid cuboidal object, translated by hand over the masking tape for 2 minutes.

Data Collection & Analysis:

For all three setups described above, and for each accelerometer axis:

- Align the selected accelerometer axis with gravity whilst securing the IMU to the setup appropriately
- Connect a flexible, shielded cable between the IMU and computer; the flexibility is used to limit any adverse effects caused to the system model by the cable
- Calibrate the IMU before running the experiment and collecting data

For all of the data collected along each axis:

- Average all of the data to determine expected orientation and compare DC bias offset terms
- Perform a Fourier transform and identify any major frequency peaks for filtering
    - Compare filtered data with unfiltered data
- Observe dead-reckoned position, velocity and orientation over time, comparing filtered to unfiltered data

Results:

Of the three experiments, the pendulum swing dataset was the only one which produced significantly different results when compared to the stationary test. Figure 4 shows a Fourier analysis of the pendulum swing experiment results without any filtering. From this it is determined that the null hypothesis is true for the Z-pattern and vertical motion cases.

Figure 5 shows the raw data verses filtered data for dead reckoned acceleration, velocity and position. Filtered acceleration appears to have drift frequencies removed, but fails to resolve the overall bias. From this, the integration in the velocity and position rapidly adds up leading to an unusable system.

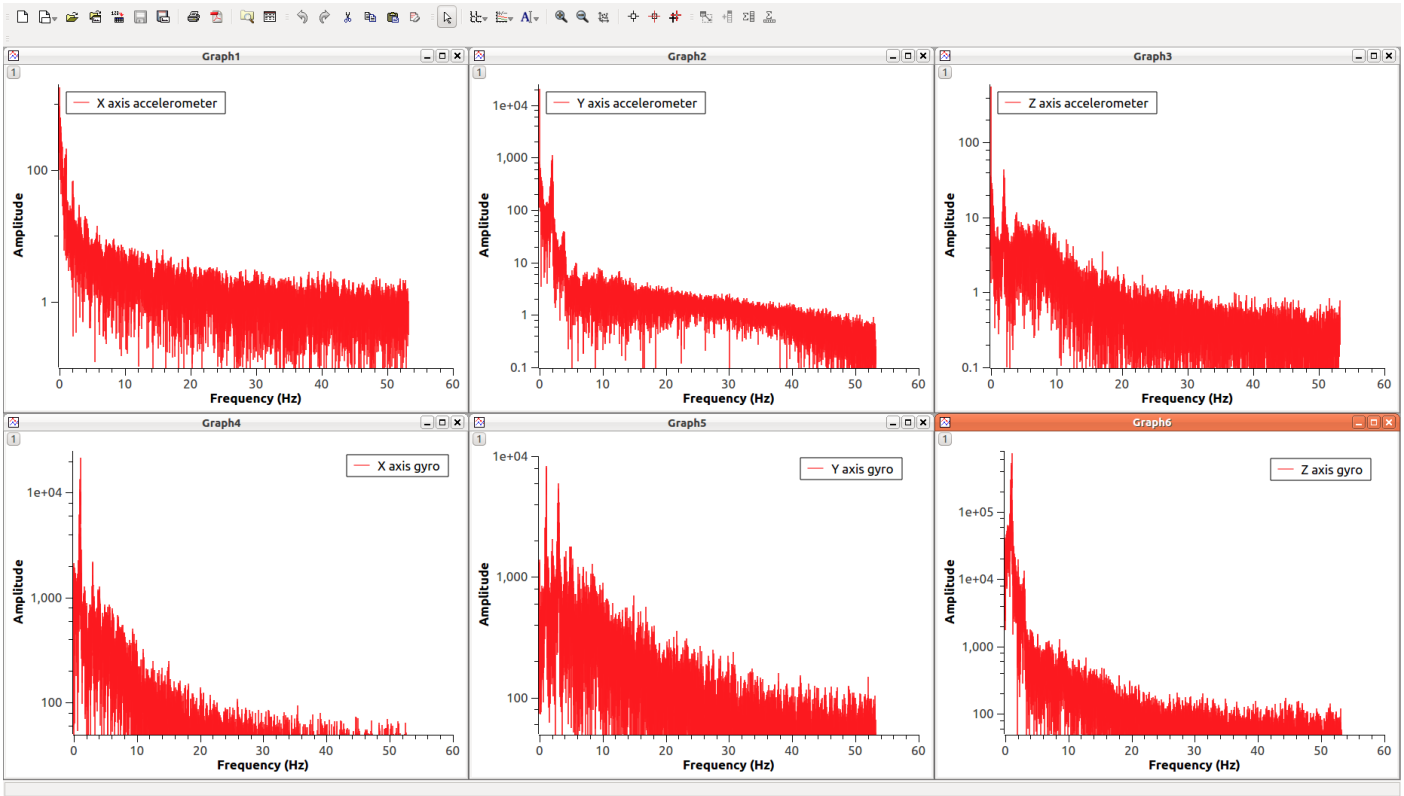

Figure 4: Fourier analysis of pendulum data

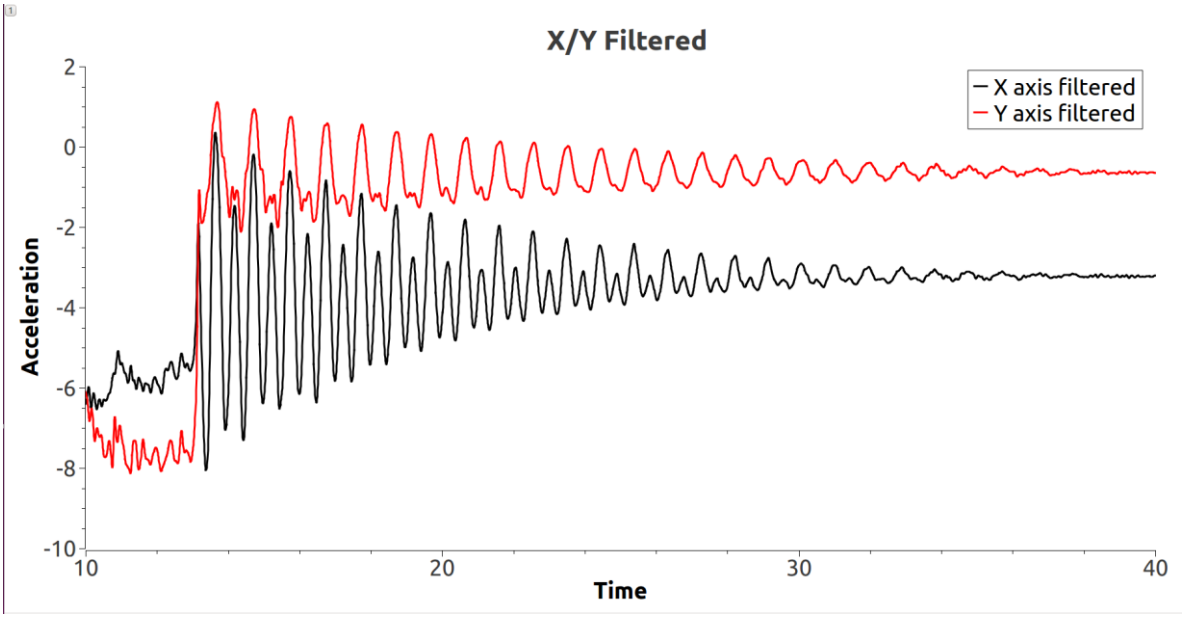

Figure 5: X-axis and Y-axis for a pendulum test against time.

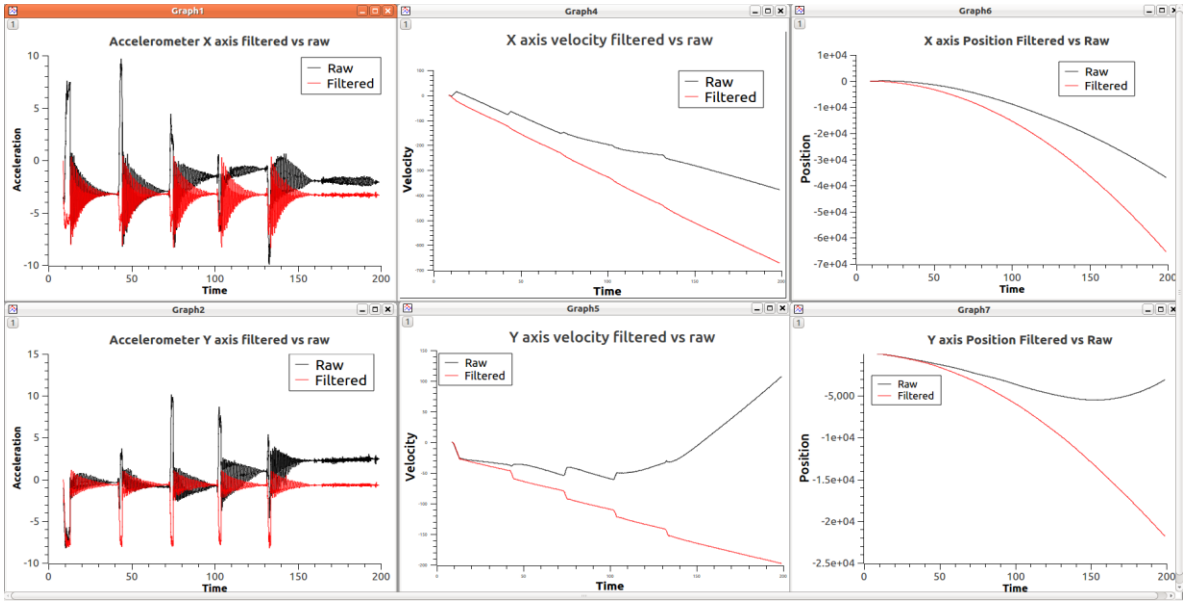

Figure 6: X & Y-axis raw verses filtered dead reckoned data.

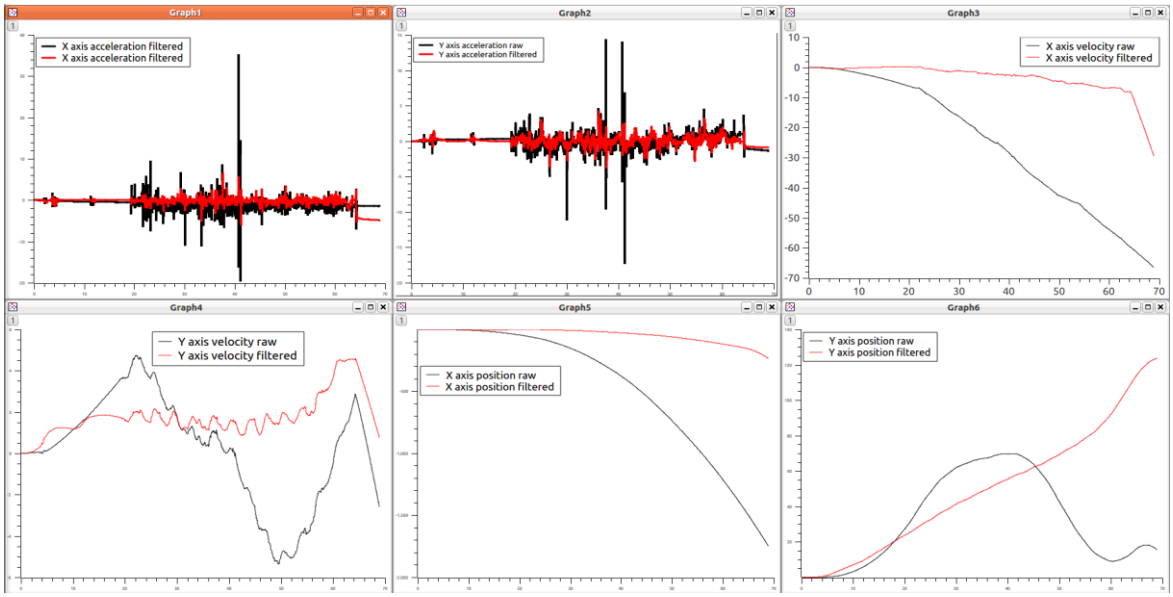

Figure 7: Acceleration, velocity and position dead reckoned data for the vertical test, showing the effects of filtering.

Task 3: Comparison with Ground Truth Data

Problem Statement:

- Perform Task 2 measurements again whilst capturing ground truth data
- Compare and contrast dead-reckoned positional data for filtered and ground-truth datasets

Theoretical Assessment:

Task 2 provided an assessment of the noise characteristics within a moving system, and a comparison of the effects filtering has on the dead-reckoned data. However, it is difficult to properly assess that data without also having a ground-truth to compare against. From this it is possible to determine the rate of change of system error over time.

Null Hypotheses:

- The system error for the dead reckoned data is negligible
- The system error will either stay the same or decrease over time

Methodology:

For each of the setups described in task 2, a coloured marker was added at an appropriate location. The experiments should then be repeated and filmed, using a camera with a fast shutter speed to reduce motion blur. This camera feed should then be used in an object tracking software package, providing a ground-truth measurement of the actual motion of the IMU.

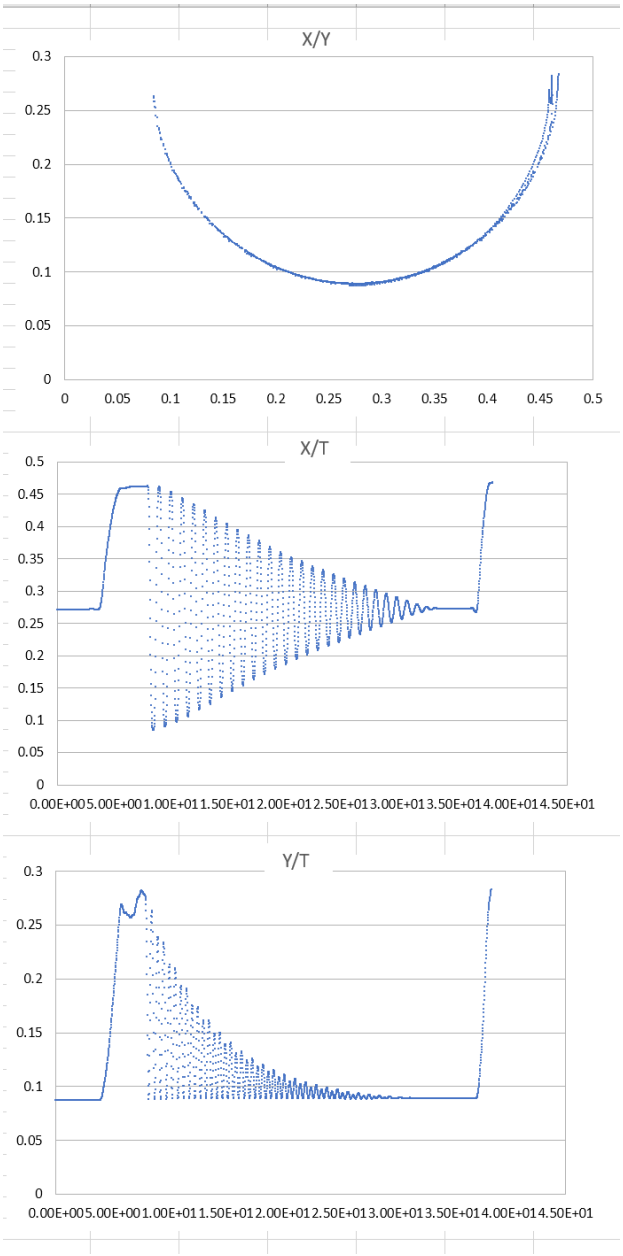

Figure 8 - Ground truth data for swinging pendulum

Camera-based positions to IMU-based positions should then be synchronised, and the camera-based data interpolated appropriately. A comparison between camera and IMU based positions should then be made, and a measure of error over time established.

Results:

Even when using filters focussed solely on the types of motions being produced, it was not possible to accurately track the position of the IMU over time. Filters provide significant improvement over raw data; however, the results are still inadequate for path reconstruction. Using figure 8 as a guideline, a comparison with the information in figure 5 can be made. The pattern of motion is sinusoidal, as such the second derivative should be the inverse of the middle graph in figure 8, centred around zero. What can be readily seen is how there is an attenuation every other swing cycle. This implies the acquired signal fails to describe the motion correctly. This may be caused by a sensor mis-alignment or 1-sided cusioning of the IMU, dampening effects, or that data could have been lost during the resampling phase.

Due to similar such errors, direct positional comparisons could not be made with the ground truth datum beyond noting that there are significant amounts of error being integrated.

Task 4: Complementary Filter

Problem Statement:

- Implement a complimentary filter as set out in the coursework specification document
- Perform Task 3 measurements again with application of the complementary filter
- Compare and contrast dead-reckoned positional data for filtered; complementary filtered and ground-truth datasets

Theoretical Assessment:

It is well known that gyroscopes suffer from significant bias drift, or angle random walk, over time. These artefacts are typically low frequency, however. Accelerometers on the other hand are very sensitive to sudden movements, whilst being under the constant influence of gravity. Both sensor sets can also provide a measure of the IMU's orientation, with the notable exception of rotations about the z-axis (yaw) for the accelerometer datum.

Thus, a sensor fusion between the low-frequency components of the accelerometers measure, and high frequency components of the gyroscopes measure, can provide an overall more accurate representation of the IMU's orientation. This allows for an improved separation of the gravity component and, hence, improved system performance.

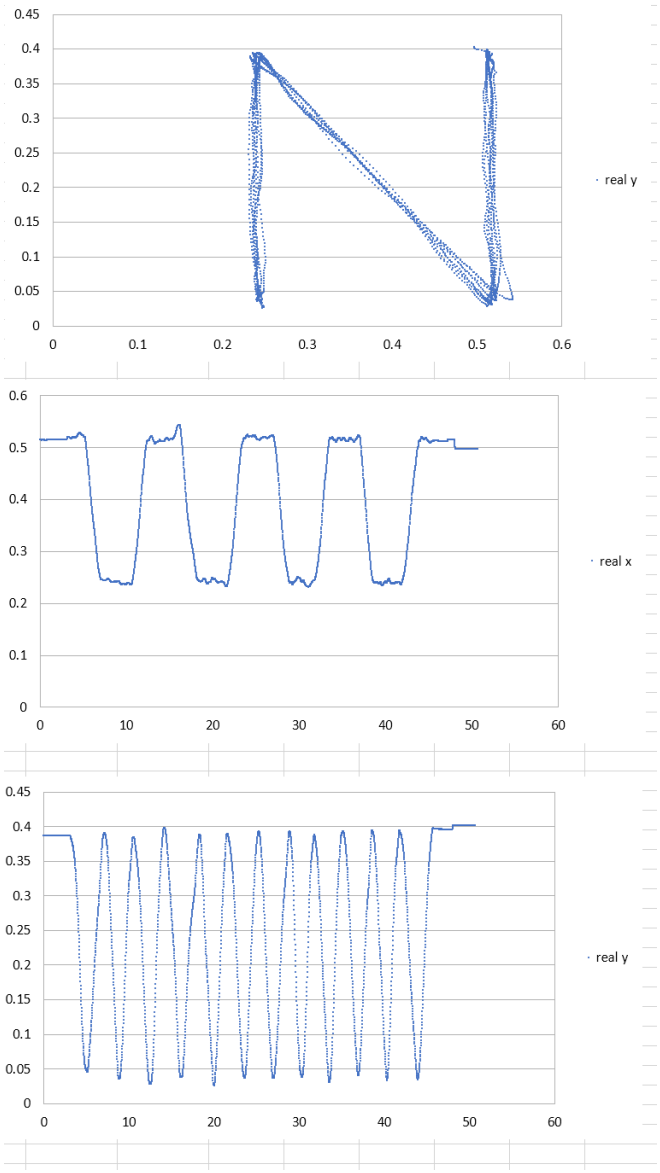

Figure 9 - Ground truth data for Z pattern

Null Hypothesis:

- The complementary filter will show no significant improvement in the system error or its rate of change

Methodology:

The experiments of task 3 should be repeated once more, with application of a complimentary filter to the filtered data. Comparisons between the ground truth, filtered and complementary filtered data should then be assessed.

Results:

Comparing the complementary filter to the filtered only and ground truth positions shows only a slight improvement. Bias random walk is reduced, but not entirely removed; because of this the velocity and positional information had significant integrated drift as shown in Figure 11. Comparing the orientations on the other hand shows significant improvement, as shown in Figure 10. Although the signal appears attenuated and inverted, its bias random walk has been completely removed. The attenuation is from a non-optimal selection of beta. Specifically, the weighted effects of the gyroscope are too low, resulting in the faster changes in orientation being missed.

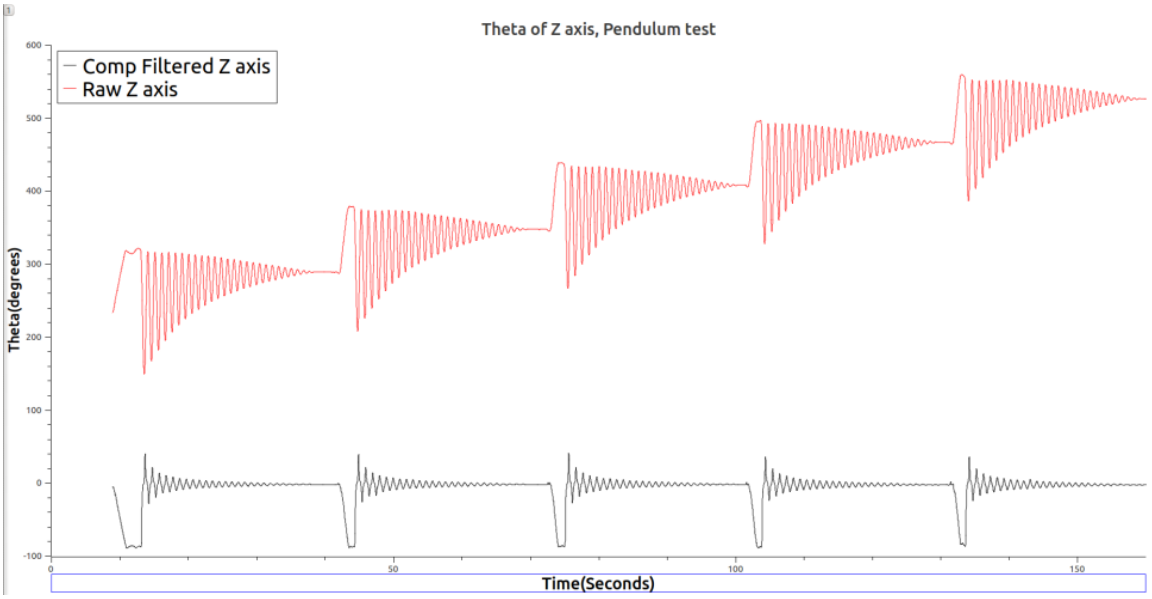

Figure 10: Z-axis orientation for the pendulum test, effect of the complementary filter.

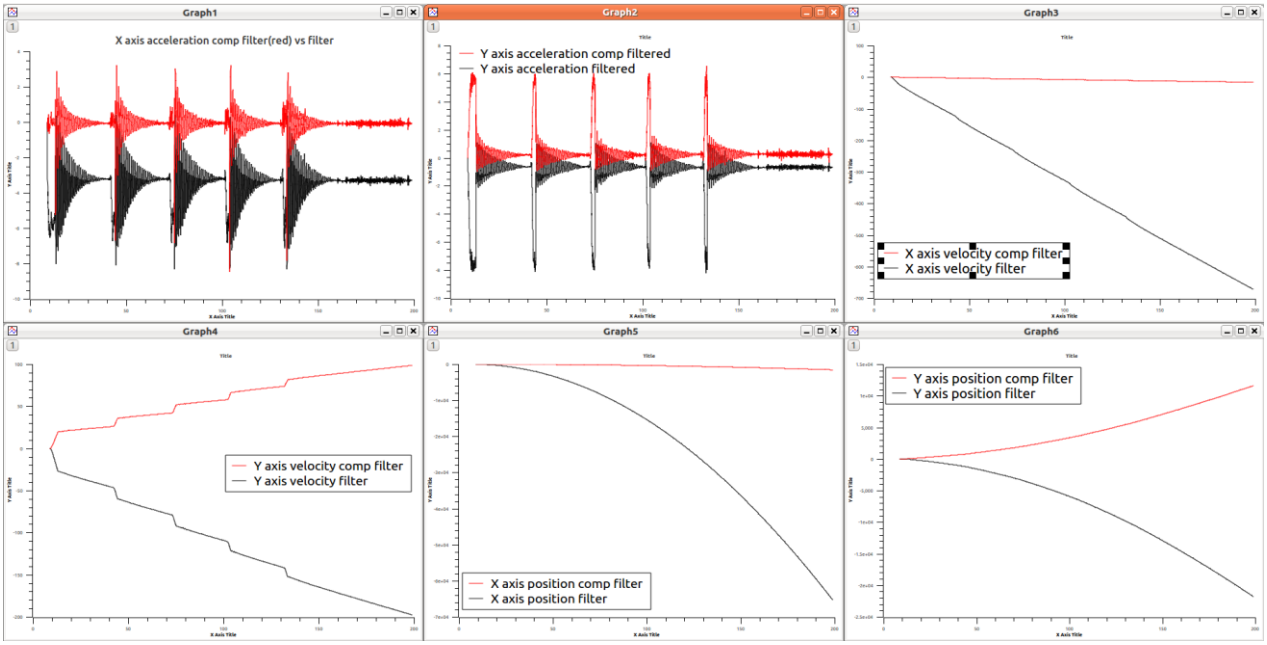

Figure 11: Pendulum data, filtered verses complementary filtered data.

Task 5: Extended Assessments

Problem Statement:

- Assess the performance of the system for dead-reckoning the position of the IMU over large distances
- Assess efficacy of alternative complementary filter setups

Theoretical Assessment:

The previous tasks have honed in towards an optimal system setup for dead reckoning with the IMU. However, previous experiments have been limited to a localised position. As such, to more fully test the capabilities of the system experiments over larger distances need to be performed.

Integration is a necessary aspect of the dead reckoning process with IMU's. Hence, measurement errors and noise will continually be integrated, or double integrated, over time, causing a gradual yet significant increase in positional error.

Null Hypotheses:

- The system will not be able to reliably dead-reckon under varying orientations for any extended period of time
- The system will not be able to reliably dead-reckon over large distances for any extended period of time
- The system will not be able to reliably dead-reckon in a volumetric, 3D space for any extended period of time

Methodology:

Large Distance Movements:

The tiles within the Smeaton 303 lab are precisely 0.5x0.5 metres in size. Using these as an accurate measure of distance, the IMU should be secured to a mobile platform and moved over a pre-set path. This path should include translations in two axes whilst fixing the orientation around the Z-axis.

Data Collection & Analysis:

For all three setups described above:

- Align a selected accelerometer axis with gravity whilst securing the IMU appropriately
- Connect a flexible, shielded cable between the IMU and computer
- Calibrate the IMU before running the experiment and collecting data

For all of the data collected:

- Known positions / paths should be plotted
- IMU-based positions should be plotted for the complementary filtered data to assess performance over each of the setups

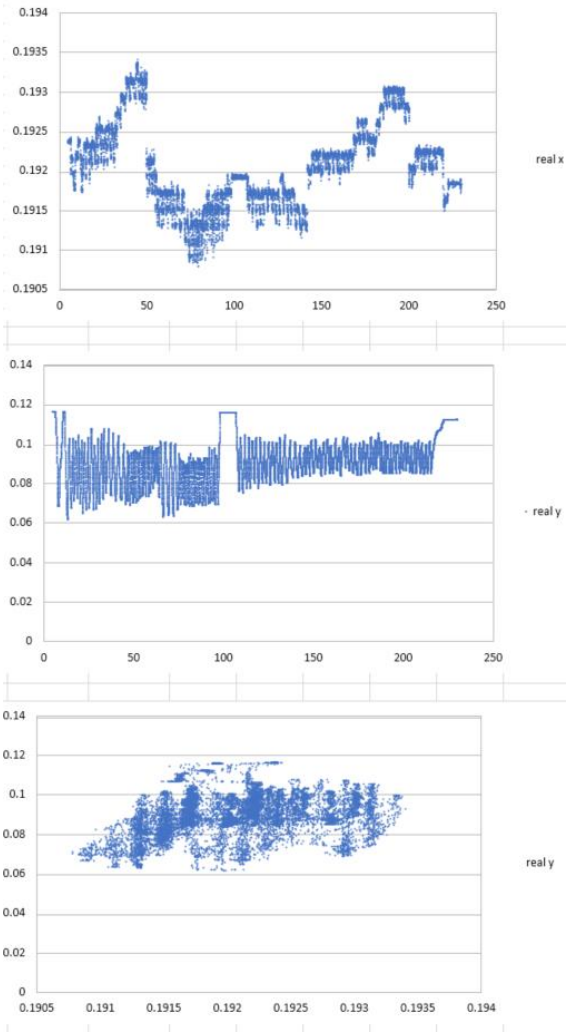

Figure 12: Ground truth data for the vertical motion experiment.

Results:

After analysing the data collected for the large motion test, it was not possible to determine the path taken, nor see any appreciable points of motion change. The complementary filter is designed to improve orientation about the roll and pitch axes. However, without the addition of a magnetometer, or other such sensor, resolving drifts in the yaw-axis is infeasible without setup constraints. Attempts were made to improve the system response via modifying the beta and filter parameters, however, this achieved very little.

Contribution:

Tom was responsible for developing the auto-calibration code, as well as the command line interface; data resampling script and the object class which allowed the Phidget to be graphed live. He also performed data collection and constructed the test apparatus for tasks 1; 2; 3 and 4.

Daniel's primary role in this project was experiment design, data collection and analysis. He produced all Fourier transforms, graphs and images used in the report. Dan also performed some of the data collection experiments and aided in the development of the mathematical model which is the core of the developed software.

Demetrius developed the filter, dead reckoning and complementary filter software, as well as the mathematical models involved in the system. Demetrius also took charge of the report writing, and performed critical analysis of each of the results.

Appendix A: Raw Data Processing

Throughout all of the experiments, dead reckoning was performed to acquire velocity and positional information from the data. This was done using the Python programming language, along with several standard libraries (including Numpy, SciPy, and PyQtGraph). Figures 3, 4 & 5 break down the underlying processes into a flow chart format.

It was noticed that, at higher IMU sample rates (~4ms), the IMU sometimes skipped data points. The typical delay between incoming timestamped data was 4ms, but approximately one in eight samples was either missed or was not sent by the IMU. These discontinuities resulted in large errors in data filtering and analysis (especially Fourier transforms). To remedy this, a MatLab script was developed to resample the data. The script starts by reading a CSV file saved by the python program. It takes in the non-uniform list of sample times and the corresponding data values and resamples them, linearly interpolating where appropriate in order to create a uniform dataset. The MatLab script then calculates the new sample rate and saves this in the filename of the resampled data.

Figure 13 (left) shows the main steps performed. Orientation initialisation is required to approximate the initial orientation conditions of the IMU using the same function as the complementary filter. Bias offsets and rescaling are performed using calibration data from a stationary setup.

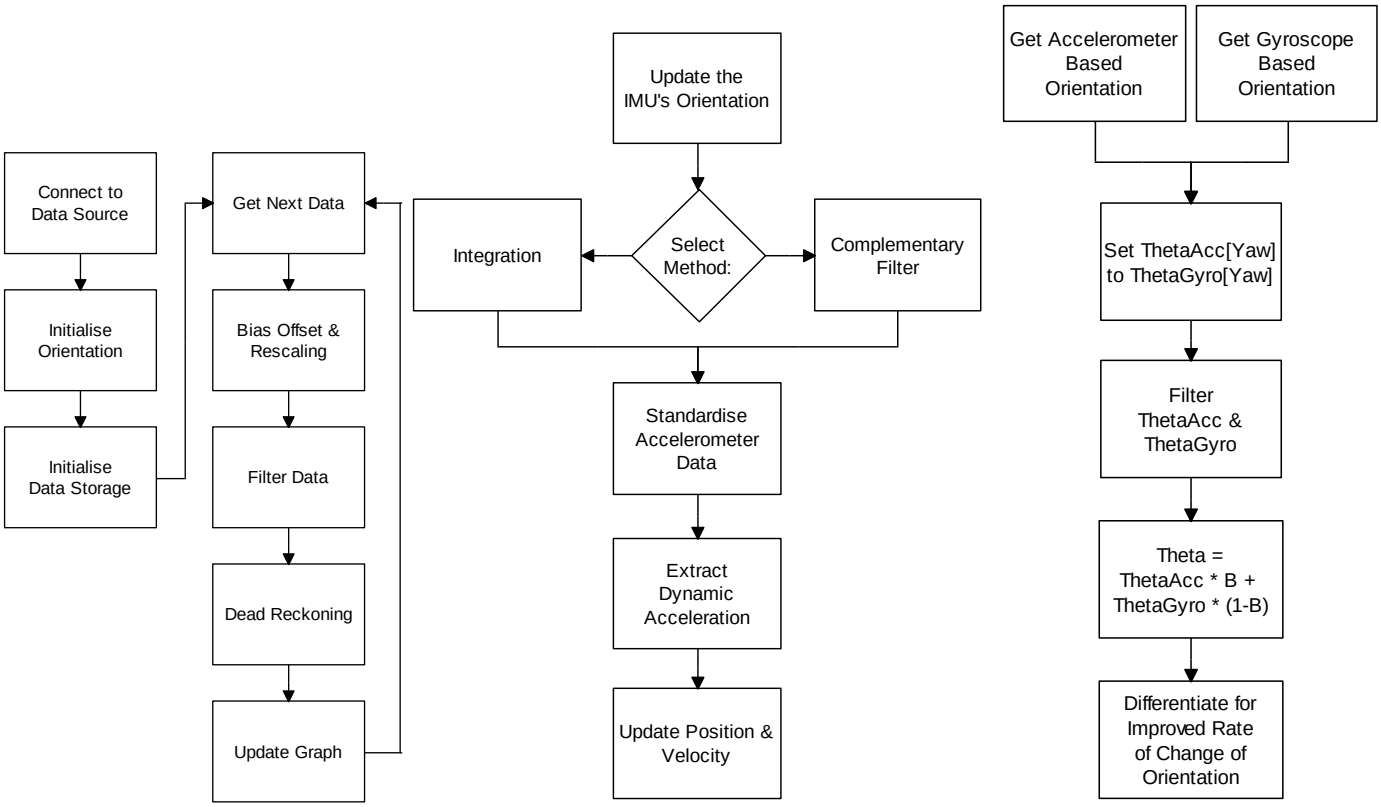

Figure 13: Flow diagrams of the overall algorithm (left), dead reckoning algorithm (middle), and complementary filter algorithm (right).

For Figure 13 (middle) the IMU orientation is updated by either performing integration, via $\theta = \omega * \Delta t$ (for tasks 1, 2 & 3), or through application of a complementary filter (for tasks 4 & 5). Using the most recently calculated Euler angle a direction cosine matrix is produced to rotate the accelerometer values such that gravity is orientated with the z-axis. Position is then calculated using $s = u * \Delta t + \frac{1}{2} * a * \Delta t^2$; followed by the IMU's current velocity via $v \mathrel{+}= a * \Delta t$. This was performed along all axes.

To calculate the orientation using the Figure 13 (right) a metric of the IMU's orientation needs to be calculated from the accelerometer data. This is done using the following maths:

$$acc\theta_x = \tan^{-1}\left({acc_y}/{acc_z}\right)$$

$$acc\theta_y = \tan^{-1}\left(\frac{-acc_x}{\sqrt[2]{acc_y^2 + acc_z^2}}\right)$$

$$acc\theta_z = gyro\theta_z$$

$acc\theta_y$ uses a different formula to $acc\theta_x$ because the angle needs to be with respect to the orientation of the IMU after performing the X-axis rotation.

To aid in development and use of the python programmes developed for this coursework, a user command line interface was developed. Calling the Main.py program with the -h flag will print out a list of arguments that can be fed to the program. These arguments allow triggering calibration sequences, specifying which offline dataset to load or whether to use the program in live mode, as well as which data to output or save to a file.

Calibration can be performed by passing the -c flag to the program. The user can then specify the duration of the calibration cycle (using -d), as well as how much initial data to discard (-s). Once calibration data is gathered the program will automatically calculate and update the bias offsets and scaling factors, save them to an external file for application to future executions of the program.

A live data mode has also been implemented by creating an IMU object class from the phidget21 test code. Live mode graphs incoming data from the IMU with different graphs selectable. Currently implemented graphs include acceleration, velocity, position, angular velocity and angular orientation against time.

Appendix B: Python code listings

For more information beyond the code, the reader is referred to https://github.com/demzai/ROCO503x.

Main.py:

```
#####################################################
################### DEPENDENCIES ####################
#####################################################
import numpy as np
import graph as gr
import filter as fl
from pyqtgraph.Qt import QtCore
import time
import dead_reckoning as dr
import sys, signal, os
import argument_parser
import pickle
clp = argument_parser.commandline_argument_parser()
args = clp.parser.parse_args()
argument_parser.validate(args)



#####################################################
################## GLOBAL CONSTANTS #################
#####################################################
inputType = args.dataSource
print "Input type:",inputType
if (inputType == "live"):
    import Spatial_simple_cl as spatialC
#fileLocale = "IMU_Stationary.txt"
fileLocale = args.fileLocation
print "File location:",fileLocale

sleepTime = 0.01
numSamplesMax = 100
graphWindow = gr.newWindow("Graphs", 640, 480)
graphChart = gr.newGraph(graphWindow, "Test")
graphAccX = gr.newPlot(graphChart, [], [], 'r', None, None, 3, 'o')
graphAccY = gr.newPlot(graphChart, [], [], 'g', None, None, 3, 'o')
graphAccZ = gr.newPlot(graphChart, [], [], 'b', None, None, 3, 'o')
updateEvery = 10
cutoffFrequency = [20, 5]  # [Accel Low Pass, Gyro High Pass]



#####################################################
################## GLOBAL VARIABLES #################
#####################################################
dataFile = None
listRaw = []
listRawDR = []
listFiltered = []
listFilteredDR = []
count = 0
calibCount = 0
dcm = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])  # @todo Incorrect initial
orientation!

#####################################################
############### CALIBRATION CONSTANTS ###############
#####################################################
calGyroOffset  = [-0.35, -0.3, -0.45]
calAccelOffset = [0.0, 0.0, 0.0]
calAccelScale  = [1.0, 1.0, 1.0]
calV = [0,0,0]
```

```
#####################################################
################## CUSTOM FUNCTIONS #################
#####################################################
#Apply scaling and offset factors
# def demiApplyCalibration(data, calGyroOffset, calAccelOffset, calAccelScale):
#     temp = data*1
#     for i in range(0, 3):
#         # Offset then scale accelerometer values
#         temp[i+1] -= calAccelOffset[i]
#         temp[i+1] /= calAccelScale[i]
#
#         # Offset the gyroscope values
#         temp[i+4] -= calGyroOffset[i]
#     return temp

def demiApplyCalibration(data, calGyroOffset, calAccelOffset, calAccelScale):
    temp = data*1
    for i in range(0, 3):
        # Offset then scale accelerometer values
        data[i+1] -= calAccelOffset[i]
        data[i+1] /= calAccelScale[i]

        # Offset the gyroscope values
        data[i+4] -= calGyroOffset[i]
    return data


def read_calibration_file():
    file = open('calibration.txt', 'rb')
    calibration_variables = pickle.load(file)
    file.close()
    return calibration_variables




def collect_calibration(data):
    global start, calibCount, calV, args
    time_esapsed = time.time() - start
    print "start time", args.startTime
    print "end time", args.startTime + args.durationTime
    if (time_esapsed > args.startTime):
        calibCount += 1
        calGyroOffset[0] += data[4]
        calGyroOffset[1] += data[5]
        calGyroOffset[2] += data[6]
    if (time_esapsed > args.startTime + args.durationTime):
        calGyroOffset[0] = calGyroOffset[0] / calibCount
        calGyroOffset[1] = calGyroOffset[1] / calibCount
        calGyroOffset[2] = calGyroOffset[2] / calibCount
        print "Vx", calGyroOffset[0]
        print "Vy", calGyroOffset[1]
        print "Vz", calGyroOffset[2]
        pickle.dump(calGyroOffset, open("calibration.txt", "wb"))
        close_nicely()




def read_calibration_file_accel():
    file = open('calibrationAccel.txt', 'rb')
    calibration_variables = pickle.load(file)
    file.close()
```

```
    return calibration_variables



def generate_accel_offsets(calibration_variables_accel):
    offsets = [0.0, 0.0, 0.0]
    for i in range(0, 3):
        offsets[i] = (calibration_variables_accel[i] + calibration_variables_accel[i +
3]) / 2
    return offsets



def generate_accel_scalars(offset, data):
    output = [0.0, 0.0, 0.0]
    for i in range(0, 3):
        output[i] = data[i] - offset[i]
    return output



    print "start time", args.startTime
    print "end time", args.startTime + args.durationTime



printflag = True
# Retrieve the next input of raw data
def getNextData():
    """
    # Extract the next piece of data
    :param type: Determines whether the function tries to read from a file or directly
from an IMU
    :return:
    """
    global inputType, imuObj, calGyroOffset, calAccelOffset, calAccelScale, args,
start, printflag
    time_esapsed = time.time() - start
    if (args.use_time)and(printflag):
        print "start time", args.startTime
        print "end time", args.startTime + args.durationTime
        print "waiting", args.startTime, "seconds to start..."
        printflag = False
    if (inputType == "file"):
        # Considered an array of chars, so [:-1] removes the last character
        data = dataFile.readline()[:-1]

    elif (inputType == "live"):
        while (imuObj.dataReady == False):
            pass
        if (imuObj.dataReady):
            data = imuObj.getData()
            if not args.calibrate:
                pass
                data = demiApplyCalibration(data, calGyroOffset, calAccelOffset,
calAccelScale)
    else:
        print("Error: invalid input type specified. Please set either \"file\" or
\"live\"")
    # Try to convert the data into an array of floats
    # This should only fail if the package is corrupt
    if (inputType == "file"):
        try:
            data = [float(i) for i in data.split(",")]
        except:
            data = None
```

```
    if (args.calibrate):
        collect_calibration(data)

    if (args.use_time):
        if (time_esapsed > args.startTime):
            print(data)
        if (time_esapsed > args.startTime + args.durationTime):
            close_nicely()
    else:
        print(data)
        return data
    return data


# Extract a single column and return it as a python list
def getCol(data, column):
    return np.array(data)[:, column].tolist()


# Normalise a given list to a given size
def limitSize(data, maxLength=numSamplesMax):
    returnVal = data
    if (returnVal[-1] == None):
        returnVal = returnVal[:-1]
    if (returnVal.__len__() >= maxLength):
        returnVal = returnVal[-maxLength:]
    return returnVal

def close_nicely():
    global imuObj
    # close down sockets before exiting
    if (inputType == "live"):
        imuObj.stopIMU()
    os.system('stty sane')
    sys.exit(0)

def handle_ctrl_c(signal, frame):
    close_nicely()
    #sys.exit(130) # 130 is standard exit code for ctrl-c



##################################################
############ INITIALIZATION FUNCTION(S) ############
##################################################
def init():
    #Initialize start time variable, IMU object and any input files
    global start, calibration_variables_gyro, calibration_variables_accel, dataFile,
fileLocale, imuObj
    global calGyroOffset, calAccelScale, calAccelOffset

    #If we're not in calibration mode, apply the last calibration values
    if not args.calibrate:
        calGyroOffset = read_calibration_file()
        calibration_variables_accel = read_calibration_file_accel()
        calAccelOffset = generate_accel_offsets(calibration_variables_accel)  #
generate offsets
        calAccelScale = generate_accel_scalars(calAccelOffset,
calibration_variables_accel)

        print "Applying calibration from file. If you wish to re-run calibration,
please call this program with -c or --calibrate."
        print "Please press enter when ready."
        # Press enter to continue
        #chr = sys.stdin.read(1)
    #If we are in calibration mode...
```

```
    else:
        duration = args.durationTime
        print "Entering calibration mode. Please try and isolate the IMU from noise and
vibrations."
        print "Calibration data will be saved to calibration.txt in the local
directory, and will be automatically applied the next time you run this program without
the calibration flag."
        if duration == 1:
            print "Calibration will start at t =", args.startTime, "seconds and finish
at t =", args.startTime + args.durationTime, "seconds, covering a duration of",
duration, "second."
        else:
            print "Calibration will start at t =", args.startTime, "seconds and finish
at t =", args.startTime + args.durationTime, "seconds, covering a duration of",
duration, "seconds."
        print "Please press enter when ready."
        #Press enter to continue
        chr = sys.stdin.read(1)


    #This line tells python to call the 'handle_ctrl_c' function if control+c is
pressed (allows us to exit nicely)
    signal.signal(signal.SIGINT, handle_ctrl_c)

    ###
    # Open the data file or connect to the IMU
    ###

    if (inputType == "file"):
        dataFile = open(fileLocale, "r")
    elif (inputType == "live"):
        imuObj = spatialC.IMU('some')
        time.sleep(1)
    # Remember start time so that we can calculate durations later
    start = time.time()
    ###
    # Initialise the lists
    ###
    global listRaw, listCrude, listFiltered
    # [time, ax,  ay,  az,  gx,  gy,  gz]
    listRaw.append(getNextData())

    # [time, ax,  ay,  az,  vx,  vy,  vz,  px,  py,  pz,
    #  gx,  gy,  gz,  tx,  ty,  tz,  qw,  qx,  qy,  qz]
    tempList = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 0.0, 0.0]
    listRawDR.append(dr.doDeadReckoning(tempList, listRaw[0]))
    listRawDR[0] = [listRawDR[0][0] - 0.004] + \
                   listRawDR[0][1:4] + [0.0, 0.0, 0.0, 0.0, 0.0, 0.0] + \
                   listRawDR[0][4:7] + [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]

    listFiltered.append(listRaw[0])
    listFilteredDR.append(listRawDR[0])

    return


#####################################################
#################### MAIN CODE ######################
#####################################################
def main():
    global listRaw, count, sleepTime, inputType, listFiltered
    count += 1

    # Get raw data
    nextData = getNextData()
    listRaw.append(nextData)
```

```
    listRaw = limitSize(listRaw)

    # Get filtered data
    if (nextData != None):
        global listRawDR, listFilteredDR, listFiltered
        listFiltered.append([listRaw[-1][0]] +
                            fl.filterData(listRaw, [1, 2, 3], ['bessel', 'low',
cutoffFrequency[0], 4]) +
                            fl.filterData(listRaw, [4, 5, 6], ['bessel', 'high',
cutoffFrequency[1], 4])
                            )

        listFiltered = limitSize(listFiltered)

        # Get dead reckoned data
        listRawDR.append(dr.doDeadReckoning(listRawDR[-1], listRaw[-1]))
        listRawDR = limitSize(listRawDR)
        listFilteredDR.append(dr.doDeadReckoning(listFilteredDR[-1], listFiltered[-1]))
        listFilteredDR = limitSize(listFilteredDR)

        # listRawDR:
        #  [time, ax,  ay,  az,  vx,  vy,  vz,  px,  py,  pz,
        #   gx,  gy,  gz,  tx,  ty,  tz,  qw,  qx,  qy,  qz]
        """
        set triplet to:
        acceleration = 0
        velocity = 1
        position = 2
        angular velocity = 3
        angular position = 4
        quaternion = 5
        """

    if args.calibrate:
        triplet=3
    else:
        triplet = args.triplet

    # Plot data if appropriate
    if (count == updateEvery):
        timeCol = getCol(listFilteredDR, 0)
        #gr.updatePlot(graphAccX, getCol(listFiltered, 1 + 3 * triplet), timeCol)
        #gr.updatePlot(graphAccY, getCol(listFiltered, 2 + 3 * triplet), timeCol)
        #gr.updatePlot(graphAccZ, getCol(listFiltered, 3 + 3 * triplet), timeCol)
        gr.updatePlot(graphAccX, getCol(listRaw, 1 + 3 * triplet), timeCol)
        gr.updatePlot(graphAccY, getCol(listFiltered, 1 + 3 * triplet), timeCol)
        #gr.updatePlot(graphAccZ, getCol(listRaw, 3 + 3 * triplet), timeCol)
    count = count % updateEvery
    if (inputType == 'file'):
        time.sleep(sleepTime)
    if (inputType == 'live'):
        time.sleep(sleepTime)
    # gr.newPlot(graphChart, getCol(listRaw, 1)[-3:-1], getCol(listRaw, 0)[-3:-1], 'g',
'r', 'b', 5, 'o')



# Cannot put into a function because QtCore.QTimer() is a dick
# Initialize the system
init()

# Update the graph data
timer = QtCore.QTimer()
timer.timeout.connect(main)
timer.start(0)
```

Filter.py:

```
# @todo create complementary filter

#####################################################
################### DEPENDENCIES ####################
#####################################################
from scipy.signal import butter, cheby1, cheby2, lfilter, bessel
import numpy as np


#####################################################
################## GLOBAL CONSTANTS #################
#####################################################
imuSampleFrequency = 136
chebyRipple = 1 #dB


# IMU sampling frequency is 62.5Hz
# Create a butterworth filter & apply it to provided data
def butter_filter(data, order, cutoff, type, fSample):
    nyq = 0.5 * fSample
    p = butter(order, cutoff/nyq, btype=type)
    return lfilter(p[0], p[1], data)

def bessel_filter(data, order, cutoff, type, fSample):
    nyq = 0.5 * fSample
    p = bessel(order, cutoff/nyq, btype=type)
    return lfilter(p[0], p[1], data)


# Create a chebyshev top-wobble filter & apply it to provided data
def cheby_top_filter(data, order, cutoff, type, fSample):
    nyq = 0.5 * fSample
    p = cheby1(order, chebyRipple, cutoff/nyq, btype=type)
    return lfilter(p[0], p[1], data)


# Create a chebyshev bottom-wobble filter & apply it to provided data
def cheby_bottom_filter(data, order, cutoff, type, fSample):
    nyq = 0.5 * fSample
    p = cheby2(order, chebyRipple, cutoff/nyq, btype=type)
    return lfilter(p[0], p[1], data)


# Select a filter & perform it
def doFilter(data, filterSelect, filterType, cutoffFrequency, order, sampleFrequency):
    if(filterSelect == 'butter'):
        return butter_filter(data, order, cutoffFrequency,
                             filterType, sampleFrequency)
    if(filterSelect == 'bessel'):
        return bessel_filter(data, order, cutoffFrequency,
                             filterType, sampleFrequency)
    elif(filterSelect == 'chebyT'):
        return cheby_top_filter(data, order, cutoffFrequency,
                                filterType, sampleFrequency)
    elif(filterSelect == 'chebyB'):
        return cheby_bottom_filter(data, order, cutoffFrequency,
                                   filterType, sampleFrequency)
    else:
        return [0]


# Extract a single column of data
def getCol(data, column):
    return np.array(data)[:, column].tolist()
```

```
# Filter a data set
# filterX = [filterSelect, filterType, cutoffFrequency(ies), order]
def filterData(data, columns, filter, sampleFrequency=imuSampleFrequency):
    numColumns = len(columns)
    returnVal = []
    for column in range(0, numColumns):
        returnVal += [doFilter(getCol(data, columns[column]), filter[0],
                               filter[1], np.array(filter[2]),
                               filter[3], sampleFrequency)[-1]]
    return returnVal
```

argument_parser.py:

```
import argparse, sys

class SmartFormatter(argparse.HelpFormatter):

    def _split_lines(self, text, width):
        if text.startswith('R|'):
            return text[2:].splitlines()
        # this is the RawTextHelpFormatter._split_lines
        return argparse.HelpFormatter._split_lines(self, text, width)

def validate(args):
    if (args.calibrate or args.accelCalibrate):
        if args.durationTime < 5:
            if args.durationTime == 1:
                print"Duration is only", args.durationTime, "second"
            else:
                print"Duration is only", args.durationTime, "seconds"
            print "It is recommended that you re-run this script with a longer
duration. Calibration data generated on this run may not be very useful"
            print "Press enter to continue, or ctrl+c to quit"
            chr = sys.stdin.read(1)

        if args.startTime < 4:
            if args.startTime == 1:
                print "You have requested calibration to start at", args.startTime,
"second"
            else:
                print "You have requested calibration to start at", args.startTime,
"seconds"
            print "It is recommended that you choose a start time of at least four
seconds. Calibration data generated on this run will be inaccurate"
            print "Press enter to continue, or ctrl+c to quit"
            chr = sys.stdin.read(1)


class commandline_argument_parser(object):
    class durationAction(argparse.Action):
        def __call__(self, parser, namespace, values, option_string=None):
            if values < 5:
                if values == 1:
                    print"Duration is only", values, "second"
                else:
                    print"Duration is only", values, "seconds"
                print "It is recommended that you re-run this script with a longer
duration. Calibration data generated on this run may not be very useful"
                print "Press enter to continue, or ctrl+c to quit"
                chr = sys.stdin.read(1)
            setattr(namespace, self.dest, values)


    class startTimeAction(argparse.Action):
        def __call__(self, parser, namespace, values, option_string=None):
            if values < 4:
                if values == 1:
                    print "You have requested calibration to start at", values,
"second"
                else:
                    print "You have requested calibration to start at", values,
"seconds"
                print "It is recommended that you choose a start time of at least four
seconds. Calibration data generated on this run will be inaccurate"
                print "Press enter to continue, or ctrl+c to quit"
                chr = sys.stdin.read(1)
            setattr(namespace, self.dest, values)
```

```
    def __init__(self):
        self.parser = argparse.ArgumentParser(description='IMU filtering program for
ROCO503. All command-line arguments are optional.', formatter_class=SmartFormatter)

        self.parser.add_argument('-l', '--live', metavar='Data source',
action='store_const', const="live", default='file', dest='dataSource',
                                 help='R|call this flag to tell %(prog)s to use live data
\ninstead of a file\ndefault=%(default)s')


        self.parser.add_argument('-c', '--calibrate', action='store_const', const=True,
default=False, dest='calibrate',
                                 help='call this flag to perform calibration of the
IMU\ndefault=%(default)s')

        self.parser.add_argument('-t', '--time', action='store_const', const=True,
default  =False, dest='use_time',
                                       help='call this flag to enable start and end
times\ndefault=%(default)s')


        self.parser.add_argument('-a', '--accelCalibrate', type=int, default=0,
metavar='\b', dest='accelCalibrate',
                                 help="R|call this flag to perform calibration of the
accelerometer on the IMU\n"
                                 "1 = X\n"
                                 "2 = Y\n"
                                 "3 = Z\n"
                                 "4 = -X\n"
                                 "5 = -Y\n"
                                 "6 = -Z\n"
                                 "0 = don't calibrate\n"
                                 "default=%(default)s")
        self.parser.add_argument('-s', '--start', nargs='?', type=int, default=4,
dest='startTime', help='Please enter an integer start time for the calibration routine
in seconds')
        #self.parser.add_argument('-e', '--end', nargs='?', type=int,
action=self.endTimeAction, default=10, dest='endTime', help='Please enter an integer
end time for the calibration routine in seconds')
        self.parser.add_argument('-d', '--duration', nargs='?', type=int, default=10,
dest='durationTime', help='Please enter an integer end time for the calibration routine
in seconds')
        self.parser.add_argument('-g', '--graph', metavar='\b', dest='triplet',
default='0', type=int, action='store',
                 help="R|    choose what data you would like to graph:\n"
                 "acceleration = 0\n"
                 "velocity = 1\n"
                 "position = 2\n"
                 "angular velocity = 3\n"
                 "angular position = 4\n"
                 "quaternion = 5\ndefault=%(default)s")

        self.parser.add_argument('-f', '--file', nargs='?',
default="IMU_Stationary.txt", const="IMU_Stationary.txt", dest='fileLocation',
                                       help="R|use this to set the file location, either by
\nspecifying a local file"
                                            "(e.g. data.txt) or a full path \n(e.g.
/home/user/data.txt)\n(not tested"
                                            "on windows yet) \ndefault=%(default)s'")



"""
data needed:
    live or not?                    ###
    thing to graph?                 ###
```

```
    data file location (optional)   ###
    calibrate vel (seperate file?)  ###
    calibrate accel (seperate file?)#
"""
```

Dead_reckoning.py:

```
import numpy as np
import quaternion as qt
from math import *


# Convert degrees to radians
def d2r(angle):
    return angle * pi / 180


# Generate a rotation matrix from the roll pitch & yaw values
def getRotationMatrix(x, y, z):
    rotX = [[1, 0, 0], [0, cos(d2r(x)), -sin(d2r(x))], [0, sin(d2r(x)), cos(d2r(x))]]
    rotY = [[cos(d2r(y)), 0, sin(d2r(y))], [0, 1, 0], [-sin(d2r(y)), 0, cos(d2r(y))]]
    rotZ = [[cos(d2r(z)), -sin(d2r(z)), 0], [sin(d2r(z)), cos(d2r(z)), 0], [0, 0, 1]]
    rot = np.matrix(rotX) * np.matrix(rotY) * np.matrix(rotZ)
    return rot.tolist()


# Update the orientation estimation
def integrateGyro(prevOrientation, gyroData, delTime):
    returnList = []
    for i in range(0, 3):
        returnList += [prevOrientation[i] + gyroData[i] * delTime]  # Theta += Gyro *
time
    return returnList


# Perform dead reckoning on the provided raw data
def doDeadReckoning(prevComplete, raw):
    # Ensure that prevComplete isn't modified anywhere by copying the data over
beforehand
    delTime = raw[0] - prevComplete[0]
    complete = prevComplete * 1  # Deep copy items over

    # Update the time
    complete[0] = raw[0]

    # Update the Euler orientation
    orientation = integrateGyro(prevComplete[13:16], raw[4:7], delTime)
    for i in range(0, 3):
        complete[i + 10] = raw[i + 4]
        complete[i + 13] = orientation[i]

    # UPDATE THE QUATERNION ORIENTATION
    # Small angle Euler angles do not work over time
    # Quaternions do work, and are safer as they don't suffer from gimbal lock
    # Choose between the 2 methods below:

    # global dcm
    # dcm = np.array(np.matrix(dcm) *
    #                np.matrix(getRotationMatrix(raw[4], raw[5], raw[6])))

    gyroQuat = qt.euler_to_quat(raw[4:7])
    currQuat = [complete[16], complete[17], complete[18], complete[19]]
    complete[16], complete[17], complete[18], complete[19] = qt.q_mult(currQuat,
gyroQuat)
    currQuat = [complete[16], complete[17], complete[18], complete[19]]
    dcm = np.array(qt.quat_to_dcm(currQuat))

    # Re-orientate the accelerometer values based on the IMU orientation
    acc = np.array([raw[1], raw[2], raw[3]]).transpose()
    acc = dcm.dot(acc) * 9.81

    # Update the acceleration, velocity & position info
    for i in range(0, 3):
```

```
        # Acceleration
        complete[i + 1] = acc[i]
        # Subtract gravity
        if (i == 2):
            acc[i] -= 9.81
        # Position += u*t + 0.5*a*t^2
        complete[i + 7] += (complete[i + 4] + 0.5 * acc[i] * delTime) * delTime
        # Velocity += a*t
        complete[i + 4] += acc[i] * delTime

    return complete
```

graph.py:

```
import numpy as np
import pyqtgraph as pg


# Create a new window to store graphs in
def newWindow(titleText, xSize, ySize):
    # Create the window
    win = pg.GraphicsWindow(title=titleText)
    # Ensure the title is set
    win.setWindowTitle(titleText)
    # Preset the window size
    win.resize(xSize, ySize)
    # Enable antialiasing for prettier plots
    pg.setConfigOptions(antialias=True)
    # Return the window
    return win


# Create a new graph within a given window
def newGraph(window, plotTitle=None, xLabel=None, xUnits=None, yLabel=None,
yUnits=None,
             xLog=False, yLog=False, showGrid=True, newRow=False):
    # Create a new graph, also creating a new row if so desired
    if (newRow == True):
        window.nextRow()
    graph = window.addPlot(title=plotTitle)

    # Setup grid & labels
    graph.showGrid(x=showGrid, y=showGrid)
    graph.setLogMode(x=xLog, y=yLog)
    graph.setLabel('bottom', xLabel, units=xUnits)
    graph.setLabel('left'  , yLabel, units=yUnits)

    # Return the graph handle
    return graph


# Add a new plot into a given graph
def newPlot(graph, yData=[], xData=[],
            lineColour='w', markInnerColour=None, markOuterColour=None,
            markSize=3, markShape='o'):

    # Resolve the no-xData condition
    if (yData == []):
        plt = graph.plot(pen=lineColour,                # Graph line colour
                         symbolBrush=markInnerColour,   # Data point dot colour
                         symbolPen=markOuterColour,     # Data point dot outline colour
                         symbolSize=markSize,           # Data point dot size
                         symbol=markShape)              # Data point dot shape
    elif (xData == []):
        plt = graph.plot(np.array(yData),               # Y-axis data
                         pen=lineColour,                # Graph line colour
                         symbolBrush=markInnerColour,   # Data point dot colour
                         symbolPen=markOuterColour,     # Data point dot outline colour
                         symbolSize=markSize,           # Data point dot size
                         symbol=markShape)              # Data point dot shape
    else:
        plt = graph.plot(np.array(xData),               # X-axis data
                         np.array(yData),               # Y-axis data
                         pen=lineColour,                # Graph line colour
                         symbolBrush=markInnerColour,   # Data point dot colour
                         symbolPen=markOuterColour,     # Data point dot outline colour
                         symbolSize=markSize,           # Data point dot size
                         symbol=markShape)              # Data point dot shape

    # Return the plot data handle
```

```
    return plt


# Update the data used in a given plot
def updatePlot(plotHandle, yData, xData=[]):
    if (xData == []):
        plotHandle.setData(np.array(yData))
    else:
        plotHandle.setData(np.array(xData), np.array(yData))
    return
```

PhidgetLibrary.py:

```
import threading
from ctypes import *
import sys

class PhidgetLibrary:
    __dll = None
    @staticmethod
    def getDll():
        if PhidgetLibrary.__dll is None:
            if sys.platform == 'win32':
                PhidgetLibrary.__dll =
windll.LoadLibrary("C:\\Users\\Aorus\\Documents\\Uni\\ROCO503_IMU\\VCpp\\Release\\phidg
et21.dll")
            elif sys.platform == 'darwin':
                PhidgetLibrary.__dll =
cdll.LoadLibrary("/Library/Frameworks/Phidget21.framework/Versions/Current/Phidget21")
            elif sys.platform == 'linux2':
                PhidgetLibrary.__dll = cdll.LoadLibrary("libphidget21.so.0")
            else:
                raise RuntimeError("Platform not supported")

        return PhidgetLibrary.__dll
```

quaternion.py:

```
# Reference: https://stackoverflow.com/questions/4870393/rotating-coordinate-system-via-a-quaternion

from math import *


def euler_to_quat(euler):
    """
    # Converts an Euler angle vector into a Quaternion vector
    # Reference: http://www.euclideanspace.com/maths/geometry/rotations/conversions/eulerToQuaternion/index.htm
    :param euler: [bank, heading, attitude] Euler angles
    :rtype: list: [w,x,y,z] Quaternion
    """
    cx = cos((euler[0] / 2)*pi/180)
    cy = cos((euler[1] / 2)*pi/180)
    cz = cos((euler[2] / 2)*pi/180)
    sx = sin((euler[0] / 2)*pi/180)
    sy = sin((euler[1] / 2)*pi/180)
    sz = sin((euler[2] / 2)*pi/180)

    w = cy * cz * cx - sy * sz * sx
    x = sy * sz * cx + cy * cz * sx
    y = sy * cz * cx + cy * sz * sx
    z = cy * sz * cx - sy * cz * sx

    quaternion = [w, x, y, z]
    return quaternion


# @todo double check this function was c&p'd correctly
def quat_to_dcm(quat):
    """
    # Reference: http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToMatrix/index.htm
    :param quat: Quaternion vector
    :return: Discrete Cosine (rotation) Matrix
    """
    q = quat
    # @todo fuck the auto-indenting as the matrix is now unreadable DX
    dcm = [
        [1 - 2 * q[3] * q[3] - 2 * q[2] * q[2], -2 * q[3] * q[0] + 2 * q[2] * q[1], 2 * q[2] * q[0] + 2 * q[3] * q[1]],
        [2 * q[1] * q[2] + 2 * q[0] * q[3], 1 - 2 * q[3] * q[3] - 2 * q[1] * q[1], 2 * q[3] * q[2] - 2 * q[1] * q[0]],
        [2 * q[1] * q[3] - 2 * q[0] * q[2], 2 * q[2] * q[3] + 2 * q[0] * q[1], 1 - 2 * q[2] * q[2] - 2 * q[1] * q[1]]
    ]
    return dcm


def normalize(v, tolerance=0.00001):
    mag2 = sum(n * n for n in v)
    if abs(mag2 - 1.0) > tolerance:
        mag = sqrt(mag2)
        v = tuple(n / mag for n in v)
    return v


def q_mult(q1, q2):
    w1, x1, y1, z1 = q1
    w2, x2, y2, z2 = q2
    w = w1 * w2 - x1 * x2 - y1 * y2 - z1 * z2
    x = w1 * x2 + x1 * w2 + y1 * z2 - z1 * y2
```

```
    y = w1 * y2 + y1 * w2 + z1 * x2 - x1 * z2
    z = w1 * z2 + z1 * w2 + x1 * y2 - y1 * x2
    return w, x, y, z


def q_conjugate(q):
    w, x, y, z = q
    return (w, -x, -y, -z)


def qv_mult(q1, v1):
    q2 = (0.0,) + v1
    return q_mult(q_mult(q1, q2), q_conjugate(q1))[1:]


def axisangle_to_q(v, theta):
    v = normalize(v)
    x, y, z = v
    theta /= 2
    w = cos(theta)
    x = x * sin(theta)
    y = y * sin(theta)
    z = z * sin(theta)
    return w, x, y, z


def q_to_axisangle(q):
    w, v = q[0], q[1:]
    theta = acos(w) * 2.0
    return normalize(v), theta
```

Spatial_simple.py:

```
#!/usr/bin/env python

#Basic imports
from ctypes import *
import sys
#Phidget specific imports
from Phidgets.Phidget import Phidget
from Phidgets.PhidgetException import PhidgetErrorCodes, PhidgetException
from Phidgets.Events.Events import SpatialDataEventArgs, AttachEventArgs,
DetachEventArgs, ErrorEventArgs
from Phidgets.Devices.Spatial import Spatial, SpatialEventData, TimeSpan
from Phidgets.Phidget import PhidgetLogLevel
import time



class IMU(object):

    #global iPrint
    #iPrint = True

    def __init__(self, print_text_in):

        self.dataList = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

        #########self.print_text_in = print_text_in
        if (print_text_in == "print"):
            self.iPrint = True
            self.iDebug = True
        elif (print_text_in == "no_print"):
            self.iPrint = False
            self.iDebug = False
        elif (print_text_in == "some"):
            self.iPrint = False
            self.iDebug = True
        else:
            print("Please say what kind of data you wish to display, call
spatialC.IMU() with either 'print', 'no_print', or 'some'. e.g. imuObj =
spatialC.IMU('some')")
            time.sleep(1)
            exit(1)
        #self.iPrint = iPrint
        #Create an accelerometer object
        try:
            self.spatial = Spatial()
        except RuntimeError as e:
            if (iDebug):
                print("Runtime Exception: %s" % e.details)
                print("Exiting....")
            time.sleep(2)
            exit(1)


            # Main Program Code
        try:
            # logging example, uncomment to generate a log file
            # spatial.enableLogging(PhidgetLogLevel.PHIDGET_LOG_VERBOSE,
"phidgetlog.log")

            self.spatial.setOnAttachHandler(self.SpatialAttached)
            self.spatial.setOnDetachHandler(self.SpatialDetached)
            self.spatial.setOnErrorhandler(self.SpatialError)
            self.spatial.setOnSpatialDataHandler(self.SpatialData)
        except PhidgetException as e:
            if (iDebug):
```

```
                print("Phidget Exception %i: %s" % (e.code, e.details))
                print("Exiting....")
            time.sleep(2)
            exit(1)
        if (self.iDebug):
            print("Opening phidget object....")

        try:
            self.spatial.openPhidget()
        except PhidgetException as e:
            if (self.iDebug):
                print("Phidget Exception %i: %s" % (e.code, e.details))
                print("Exiting....")
            time.sleep(2)
            exit(1)

        if (self.iDebug):
            print("Waiting for attach....")

        try:
            self.spatial.waitForAttach(10000)
        except PhidgetException as e:
            if (self.iDebug):
                print("Phidget Exception %i: %s" % (e.code, e.details))
            try:
                self.spatial.closePhidget()
            except PhidgetException as e:
                if (self.iDebug):
                    print("Phidget Exception %i: %s" % (e.code, e.details))
                    print("Exiting....")
                exit(1)
            if (self.iDebug):
                print("Exiting....")
            time.sleep(2)
            exit(1)
        else:
            self.spatial.setDataRate(4)
            self.DisplayDeviceInfo()
        """
        if (self.iDebug):
            print("Press Enter to quit....")

        # This is how the phidget used to shut down when enter was pressed. Find a way
to re-integrate this!
        chr = sys.stdin.read(1)

        if (self.iDebug):
            print("Closing...")

        try:
            self.spatial.closePhidget()
        except PhidgetException as e:
            if (self.iDebug):
                print("Phidget Exception %i: %s" % (e.code, e.details))
                print("Exiting....")
            time.sleep(2)
            exit(1)

        if (self.iDebug):
            print("Done.")
        """
    #Information Display Function
    def DisplayDeviceInfo(self):
        if (self.iDebug):
            print("|------------|----------------------------------|--------------|----
--------|")
```

```
            print("|- Attached -|              Type              -|- Serial No. -|-
Version -|")
            print("|------------|--------------------------------|--------------|----
--------|")
            print("|- %8s -|- %30s -|- %10d -|- %8d -|" % (self.spatial.isAttached(),
self.spatial.getDeviceName(), self.spatial.getSerialNum(),
self.spatial.getDeviceVersion()))
            print("|------------|--------------------------------|--------------|----
--------|")
            print("Number of Acceleration Axes: %i" %
(self.spatial.getAccelerationAxisCount()))
            print("Number of Gyro Axes: %i" % (self.spatial.getGyroAxisCount()))
            print("Number of Compass Axes: %i" % (self.spatial.getCompassAxisCount()))

    def stopIMU(self):

        if (self.iDebug):
            print("Closing...")

        try:
            self.spatial.closePhidget()
        except PhidgetException as e:
            if (self.iDebug):
                print("Phidget Exception %i: %s" % (e.code, e.details))
                print("Exiting....")
            time.sleep(2)
            exit(1)

        if (self.iDebug):
            print("Done.")

    #Event Handler Callback Functions
    def SpatialAttached(self, e):
        attached = e.device
        if (self.iDebug):
            print("Spatial %i Attached!" % (attached.getSerialNum()))

    def SpatialDetached(self, e):
        detached = e.device
        if (self.iDebug):
            print("Spatial %i Detached!" % (detached.getSerialNum()))

    def SpatialError(self, e):
        try:
            source = e.device
            if (self.iDebug):
                print("Spatial %i: Phidget Error %i: %s" % (source.getSerialNum(),
e.eCode, e.description))
        except PhidgetException as e:
            if (self.iDebug):
                print("Phidget Exception %i: %s" % (e.code, e.details))

    def SpatialData(self, e):
        source = e.device
        if (self.iPrint):
            print("Spatial %i: Amount of data %i" % (source.getSerialNum(),
len(e.spatialData)))
        for index, spatialData in enumerate(e.spatialData):
            if (self.iPrint):
                print("=== Data Set: %i ===" % (index))
            if len(spatialData.Acceleration) > 0:
                self.dataList[1] = spatialData.Acceleration[0]
                self.dataList[2] = spatialData.Acceleration[1]
                self.dataList[3] = spatialData.Acceleration[2]
                if (self.iPrint):
```

```
                    print("Acceleration> x: %6f  y: %6f  z: %6f" %
(spatialData.Acceleration[0], spatialData.Acceleration[1],
spatialData.Acceleration[2]))
            if len(spatialData.AngularRate) > 0:
                self.dataList[4] = spatialData.AngularRate[0]
                self.dataList[5] = spatialData.AngularRate[1]
                self.dataList[6] = spatialData.AngularRate[2]
                if (self.iPrint):
                    print("Angular Rate> x: %6f  y: %6f  z: %6f" %
(spatialData.AngularRate[0], spatialData.AngularRate[1], spatialData.AngularRate[2]))
            if len(spatialData.MagneticField) > 0:
                if (self.iPrint):
                    print("Magnetic Field> x: %6f  y: %6f  z: %6f" %
(spatialData.MagneticField[0], spatialData.MagneticField[1],
spatialData.MagneticField[2]))
            if (self.iPrint):
                print("Time Span> Seconds Elapsed: %i  microseconds since last packet:
%i" % (spatialData.Timestamp.seconds, spatialData.Timestamp.microSeconds))
            seconds = float(spatialData.Timestamp.seconds +
spatialData.Timestamp.microSeconds/1000000.0)
            self.dataList[0] = float(self.truncate(seconds, 5))
        if (self.iPrint):
            print("----------------------------------------")
        self.dataReady = True

    def getData(self):
        if (self.dataReady):
            self.dataReady = False
            return self.dataList



    def truncate(self, f, n):
        '''Truncates/pads a float f to n decimal places without rounding'''
        s = '{}'.format(f)
        if 'e' in s or 'E' in s:
            return '{0:.{1}f}'.format(f, n)
        i, p, d = s.partition('.')
        return '.'.join([i, (d + '0' * n)[:n]])
```

Appendix C: Special Thanks

Special thanks is given to David Jenkins for teaching the module and Martin Stoelen for providing equipment for experiments.