
Sponsored by: NASA Dryden Flight Research Center

June 7, 2013

Cal Poly Senior Project, SAS Team

Alex F. Graebe
agraebe@calpoly.edu

Bridgette S. Kimball
bskimbal@calpoly.edu

Drew T. LaVoise
alavoise@calpoly.edu
Table of Contents

List of Figures ................................................................................................................................................ 3
List of Tables ................................................................................................................................................. 5
List of Equations ............................................................................................................................................ 5
Abstract ......................................................................................................................................................... 7
Introduction .................................................................................................................................................. 8
Background ................................................................................................................................................... 8
Objectives ..................................................................................................................................................... 9
Method of Approach ................................................................................................................................... 12
  Step 1. Generate Concepts and Requirements ........................................................................................ 12
  Step 2a. Test Phones in Experiments ....................................................................................................... 12
  Step 2b. Algorithm Development ............................................................................................................ 12
  Step 3. Validate Data ............................................................................................................................... 13
  Step 4. Evaluate Results .......................................................................................................................... 13
  Step 5. Test Phones/Algorithm ................................................................................................................ 13
Design Development ................................................................................................................................... 13
  Description of Algorithm Concepts ......................................................................................................... 13
  Algorithm Design Selection ..................................................................................................................... 14
  Description of Mounting Fixtures ........................................................................................................... 15
  Mounting Fixture Analysis ....................................................................................................................... 17
  Mounting Fixture Selection ....................................................................................................................... 17
Methods of Attitude Estimation .................................................................................................................. 18
  Method #1: Components of gravitational acceleration ........................................................................ 18
  Method #2: Flight path in 3D space from GPS ..................................................................................... 18
  Method #3: Flight path from change in heading .................................................................................. 19
Final Design ................................................................................................................................................. 20
  Description of Selected Mounting Fixture .............................................................................................. 20
  Description of the Software Layout ......................................................................................................... 20
    1. Sensor Test (JAVA) ...................................................................................................................... 20
    2. Plot SAS (MATLAB) .................................................................................................................... 26
    3. Plot Test Flight (MATLAB) .......................................................................................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Verification and Testing</td>
<td>37</td>
</tr>
<tr>
<td>Testing</td>
<td>37</td>
</tr>
<tr>
<td>Using the Data Collection App</td>
<td>38</td>
</tr>
<tr>
<td>Installation:</td>
<td>38</td>
</tr>
<tr>
<td>Running the App:</td>
<td>39</td>
</tr>
<tr>
<td>Retrieving the Files:</td>
<td>39</td>
</tr>
<tr>
<td>Results</td>
<td>41</td>
</tr>
<tr>
<td>Inertial Lab</td>
<td>41</td>
</tr>
<tr>
<td>Ideal Data</td>
<td>41</td>
</tr>
<tr>
<td>Flight Test</td>
<td>43</td>
</tr>
<tr>
<td>Management Plan</td>
<td>46</td>
</tr>
<tr>
<td>Conclusions and Recommendations</td>
<td>47</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>48</td>
</tr>
<tr>
<td>References</td>
<td>48</td>
</tr>
<tr>
<td>Appendices</td>
<td>49</td>
</tr>
<tr>
<td>Appendix A. Project Timeline</td>
<td>49</td>
</tr>
<tr>
<td>Appendix B. Quality Function Development</td>
<td>50</td>
</tr>
<tr>
<td>Appendix C. Software Architecture Models</td>
<td>51</td>
</tr>
<tr>
<td>Appendix D: Gantt Chart</td>
<td>56</td>
</tr>
<tr>
<td>Appendix E: Fixture Testing Procedure</td>
<td>57</td>
</tr>
<tr>
<td>Objective:</td>
<td>57</td>
</tr>
<tr>
<td>Setup:</td>
<td>57</td>
</tr>
<tr>
<td>Procedure:</td>
<td>57</td>
</tr>
<tr>
<td>Experimental Results:</td>
<td>60</td>
</tr>
<tr>
<td>Appendix F: Test Point Matrix</td>
<td>61</td>
</tr>
<tr>
<td>Appendix G: Design Verification Plan</td>
<td>63</td>
</tr>
<tr>
<td>Appendix H: Java Software Code</td>
<td>64</td>
</tr>
<tr>
<td>Appendix I: Matlab Software</td>
<td>80</td>
</tr>
<tr>
<td>Plot SAS Flight Parameters (plotSAS)</td>
<td>80</td>
</tr>
<tr>
<td>Flight Test Point to File Name (flightTestPoint2Filename)</td>
<td>86</td>
</tr>
<tr>
<td>Import G1000 File (importG1000File)</td>
<td>92</td>
</tr>
</tbody>
</table>
List of Figures
Figure 1. Methodology Flow Chart ................................................................. 12
Figure 2. ECEF Coordinate System .................................................................. 23
Figure 3. Data Collection Application Start-up Screen ..................................... 40
Figure 4. Data Collection Application Recording Screen .................................. 40
Figure 5. Bank calculated from accelerometer and gyroscope data for inertial lab. .......... 41
Figure 6. Radius of curvature, normal acceleration, calculated bank, and flight path for ideal data. ..... 42
Figure 7. Radius of curvature, normal acceleration, calculated bank, and flight path for ideal data with 10 ft random noise. .... 42
Figure 8. Comparison of phone and truth source bank for test point 24 .................. 43
Figure 9. Comparison of phone and truth source pitch for test point 24..........................43
Figure 10. Comparison of phone and truth source barometric altitude for test point 24. ...........44
Figure 11. Comparison of phone and truth source GPS altitude for test point 15....................45
Figure B1. House of Quality. ......................................................................................................50
Figure C1. Top Level Software Architecture for Model 1. .................................................................51
Figure C2. Top Level Software Architecture for Model 2. .................................................................52
Figure C3. Top Level Software Architecture for Model 3. .................................................................45
Figure C4. Top Level Software Architecture for Final Design. .........................................................54
Figure C5. Top Level Software Architecture for Final Design (Health Monitoring). .................45
Figure D1. Gantt chart for SAS project October 11, 2012 to May 31, 2013. .................................56
Figure E1. Setup for Vibration Testing of Phone Mounts. .................................................................57
Figure E2. Laptop with RT Pro and Shake Table.............................................................................58
Figure E3. Arkon Mount Z-Axis Test Setup. .....................................................................................58
Figure E4.iOttie X-Axis Test Setup.................................................................................................59
Figure E5. Velcro X-Axis Test Setup.................................................................................................59
Figure E6. Amplitude Ratio of Fixtures Mounted in X-Axis............................................................60
Figure E7. Amplitude Ratio of Fixtures Mounted in Z-Axis............................................................60
List of Tables
Table 1. Sensor Evaluation Formal Engineering Requirements ................................................................. 10
Table 2. Explanation of Engineering Requirements and Methods of Measuring ........................................ 11
Table 3. Algorithm Decision Matrix ............................................................................................................. 15
Table 4. Mounting Fixture Decision Matrix .................................................................................................. 18
Table 5. Health Monitoring Limitations ....................................................................................................... 24
Table 6. List of Available G1000 Parameters ................................................................................................. 28
Table 7. List of Sensor Parameters from Cellphone ....................................................................................... 29
Table 8. List of GPS Parameters from Cellphone ........................................................................................... 29
Table 9. List of Measured/Calculated Flight Parameters .............................................................................. 35
Table 10. List of Parameters Supplied to Data Analysis Functions ................................................................. 36
Table 11. Visual inspection of graphs comparing G1000 data to phone data for three test points .............. 45
Table A1. List of Major Milestones. .............................................................................................................. 45
Table F1. Matrix of Test Points for May 3, 2013 Flight Testing .................................................................. 61
Table G1. Design Verification Plan for Fixture and Algorithm. .................................................................... 63

List of Equations
(1): Moving Average Filter .......................................................................................................................... 20
(2): Pitch from Accelerometers .................................................................................................................. 21
(3): Pitch from Gyroscopes ........................................................................................................................... 21
(4): Pitch (Method #1) ................................................................................................................................ 21
(5): Bank from Accelerometers ..................................................................................................................... 21
(6): Bank from Gyroscopes ............................................................................................................................ 21
(7): Bank (Method #1) .................................................................................................................................. 21
(8): Raw Climb Rate ..................................................................................................................................... 22
(9): Filtered Climb Rate .................................................................................................................................. 22
(10): ECEF Conversion ................................................................................................................................... 22
(11): Raw Ground Speed ............................................................................................................................... 23
(12): Filtered Ground Speed ......................................................................................................................... 23
(13): Parameters for Radius of Curvature (Method #2) ................................................................................ 30
(14): Radius of Curvature (Method #2) ......................................................................................................... 30
(15): Normal Acceleration ............................................................................................................................. 30
(16): Tangential Acceleration ......................................................................................................................... 30
(17): Flight Path from ECEF .......................................................................................................................... 31
(18): Flight Path Unit Vector .......................................................................................................................... 31
(19): Vector Normal to Flight Path ................................................................................................................ 31
(20): Vector Tangent to Flight Path ............................................................................................................... 31
(21): Bank (Method #2) .................................................................................................................................. 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ideal Flight Data</td>
<td>32</td>
</tr>
<tr>
<td>Orientation of Aircraft</td>
<td>32</td>
</tr>
<tr>
<td>Change in Heading (Method #3)</td>
<td>33</td>
</tr>
<tr>
<td>Radius of Curvature (Method #3)</td>
<td>33</td>
</tr>
<tr>
<td>Distance Travelled</td>
<td>37</td>
</tr>
</tbody>
</table>
Abstract
Following the development of Aircraft Collision Avoidance Technology (ACAT) by the National Aeronautics and Space Administration (NASA), a need arose to transition the life-saving technology to aid the general aviation community. Considering the realistic cost of implementation, it was decided that the technology should be adapted to function on any smartphone, using that device as an end-to-end solution to sense, process, and alert the pilot to imminent threats. In September of 2012, the SAS (Sense and Survive) Senior Project Team at California Polytechnic University (Cal Poly), San Luis Obispo was assigned the task of using smartphone technology to accurately sense the flight environment and supply usable data to the collision avoidance algorithm. This report outlines the ideation, design selection, and development that the team of students progressed through in the process of determining the feasibility of using a mobile smartphone device to determine attitude and position as algorithm input parameters. Furthermore, this report includes all plans for design verification and product implementation with regards to this team’s task. It was the conclusion of this team that due to the sensitivity of the phone’s internal sensors to noise, complimentary and simple low-pass filtering methods are insufficient in terms of determining accurate attitude. However, it is possible to achieve accurate position using GPS sensors. With better filtered GPS data, it could be possible to determine attitude based on flight path. The final product of this process and any results produced by the SAS Senior Project Team was presented at the Senior Project Exposition at Cal Poly on May 30, 2013. Everything produced and all intellectual property relating to this project is the property of NASA, and the members of the ACAT team will be responsible for any further development and implementation of this technology.
Introduction
The Advanced Collision Avoidance Algorithm developed by NASA has the capacity to save the lives of many pilots by avoiding imminent ground collision threats. The software has already been proven to work for the F-16 and for small unmanned air vehicles (UAV). As this software transitions into general aviation, the need has arisen to provide the collision avoidance algorithm with accurate input data such as attitude and position. There currently exist a number of inertial navigation systems that can accurately provide aircraft attitude and position, but these systems are all very expensive and not readily available to the average general aviation pilot. Thus, the goal of this team is to provide accurate and timely input data to the algorithm using only the smartphones carried by many in the general aviation community. The CTP Avoidance team (another Senior Project team sponsored by NASA at California Polytechnic University, San Luis Obispo) was responsible for developing the algorithm for general aviation, and the objective of this SAS team was to implement their smartphone sensing module into their general aviation collision avoidance software. This project is being sponsored by NASA, and upon completion of this project they are in charge of controlling further development and implementation. Successfully achieving the objectives of this project has the potential to save the lives of hundreds of pilots.

Background
For years, NASA has been developing a system to avoid imminent ground collision, which they call Advanced Collision Avoidance Technology (ACAT). The software takes a set of input parameters, and uses them to determine the projected paths of an aircraft. It then compares these projected paths against a GPS database, and for any cases of an immediate ground collision threat, commands the autopilot to fly the aircraft away from the ground threat. The ACAT team started developing the software for an F-16 fighter jet in 2004. The software was originally coded in C++ and used the F-16 inertial navigation system to provide input parameters to the algorithm. Considering the high velocity and climbing power for this military aircraft, the only path projected for the aircraft was a straight fly-up maneuver. Testing for the F-16 model was completed in August 2010, and is now in the process of being implemented into existing fighter aircraft.

Building on this technology, the ACAT team soon began work on developing this software for the Small Unmanned Air Vehicle (UAV) model. The goal was to prove that the technology developed for a multi-million dollar military aircraft could be transferred to a more generalized aircraft. Before moving the technology into the realm of general aviation, the technology was adapted for a small remotely-controlled plane. At that point in time, there were several existing systems which did not perform ground collision avoidance, but did contain various technologies that contributed to the process. Some of these included avionics and flight control systems such as the Garmin 1000. This avionics system has the capability to sense environment (GPS, aircraft attitude, etc.), and also to control the aircraft through an autopilot capability. Also existing on the market are several data acquisition units, such as the Piccolo, which has the capacity to collect data for GPS and aircraft attitude. Utilizing these existing data
acquisition technologies for general aviation, the ACAT team began the ideation process of combining data acquisition technology with the collision avoidance algorithm.

Soon after, the software code was converted to Java as part of the adaptive process, and a Piccolo data acquisition unit was chosen as the method of obtaining the input parameters for the algorithm. Since the algorithm was to be used for an entirely different class of aircraft with much lower speeds and limited climbing power, the algorithm was redesigned to calculate three projected paths for the aircraft: right, left, and straight. Successful testing was completed for the Small UAV in the summer of 2012. A vision was then projected for how this software could reach the general aviation community. The CTP Avoidance team was assigned to the role of adjusting the software for the general aviation model, and the SAS team was assigned to the role of providing accurate and timely input parameters to the algorithm using the native sensors on a smartphone device. Combined together, the goal is that these teams may generate the potential for the software to one day be implemented on a smartphone, using the device as the end-to-end solution for sensing, computing, and running the algorithm.

Objectives
The goal of this project is to provide accurate data to the ACAT system through phone and low-cost sensor integration. The sensors require testing to show that they can be used to accurately determine the state of an aircraft including attitude, latitude, longitude, and altitude sufficient to support trajectory estimation. In order for the ACAT system to sense a collision threat the phone sensors will work with Digital Terrain Elevation Data (DTED) to build a model of the terrain including elevation, foliage, and man-made obstacles. Once a threat has been determined, the sensors will provide the software with the information to predict evasion trajectories relative to projected threats.

A House of Quality has been developed to define customer and engineering requirements, benchmarked competitors, and set targets. The customers were determined to be the NASA ACAT team, General Aviation Pilots, CTP Avoidance Team, the Federal Aviation Administration (FAA), and future senior project teams. Because each customer has interest in different aspects of the project, each customer requirement has been ranked in order of its importance to the project.

The CTP Avoidance, NASA ACAT, and future senior project teams require a modular software architecture that will not interfere with current algorithms. This will support the preparation for future development of the software. General Aviation (GA) pilots as well as the FAA require precise attitude and position data in order to have confidence in the system. A prompt refresh/scan rate will ensure that figures are up to date and true. In order to make the system readily available to GA pilots, the software will be useable on most phones with an Android™ operating system. Because of the closed programming platform of the iPhone, the algorithm will not be developed for use with this system at this time. The compatibility will be determined by the variety of sensors on the phone, specifically whether or not they contain gyroscopes and accelerometers. Once a pilot has a device that is compatible with the software, he or she will be able to make use of the program on whatever aircraft they are flying. The device and aircraft versatility, along with low cost, will put this system ahead of the
The benchmarks were determined to be the Piccolo Data Acquisition Unit and the Inertial Navigation System currently in use on the F-16 aircrafts. From the House of Quality it can be seen that while both benchmarks are fast and accurate, they are compatible with less aircrafts and more expensive than the proposed solution.

Developing customer requirements lead to the formation of formal engineering specifications that are quantifiable and testable. Correlations were established between the requirements on a strong, medium, small, and no correlation scale. A risk level of high (H) was set for attitude, position, and altitude accuracy because failure to comply with these parameters would present a safety risk for pilots.

<table>
<thead>
<tr>
<th>Spec. #</th>
<th>Parameter Description</th>
<th>Requirement or Target (units)</th>
<th>Tolerance</th>
<th>Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Attitude Accuracy</td>
<td>3 deg. 5 deg.</td>
<td>Min</td>
<td>H</td>
</tr>
<tr>
<td>2</td>
<td>Position Accuracy (Lat/Long)</td>
<td>10 ft 50 ft</td>
<td>Min</td>
<td>H</td>
</tr>
<tr>
<td>3</td>
<td>Altitude Accuracy</td>
<td>10 ft MSL 50 ft MSL</td>
<td>Min</td>
<td>H</td>
</tr>
<tr>
<td>4</td>
<td>Refresh/Scan Rate</td>
<td>200 ms 500 ms</td>
<td>Max</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>Device Compatibility</td>
<td>90% 68%</td>
<td>Min</td>
<td>L</td>
</tr>
<tr>
<td>6</td>
<td>Aircraft Compatibility</td>
<td>100% 100%</td>
<td>Min</td>
<td>L</td>
</tr>
<tr>
<td>7</td>
<td>Repeatability</td>
<td>99 % 95%</td>
<td>Min</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>GPS Availability</td>
<td>4 Satellites 3 Satellites</td>
<td>Min</td>
<td>M</td>
</tr>
<tr>
<td>9</td>
<td>Calibration Time</td>
<td>30 sec 60 sec</td>
<td>Max</td>
<td>L</td>
</tr>
</tbody>
</table>

Note:
L- Low
M-Medium
H- High

1 These targets are nominal values under good operating environment
2 The target goals go beyond the minimum requirement for that parameter, and estimate what could be possible
3 The target threshold outlines the minimum baseline for what is required for the algorithm to accurately process the input data.
<table>
<thead>
<tr>
<th>Spec. #</th>
<th>Parameter Name</th>
<th>Parameter Description</th>
<th>Method of Measuring Targets</th>
<th>Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Attitude Accuracy</td>
<td>The range of rotational uncertainties for heading, bank, and pitch</td>
<td>Provided by NASA based on previously determined parameters</td>
<td>T,S</td>
</tr>
<tr>
<td>2</td>
<td>Position Accuracy</td>
<td>The range of horizontal uncertainty required to run the algorithm</td>
<td>Provided by NASA based on previously determined parameters</td>
<td>T,S</td>
</tr>
<tr>
<td>3</td>
<td>Altitude Accuracy</td>
<td>The range of vertical uncertainty required to run the algorithm</td>
<td>Provided by NASA based on previously determined parameters</td>
<td>T,S</td>
</tr>
<tr>
<td>4</td>
<td>Refresh/Scan Rate</td>
<td>The time taken to send updated data from the sensor class to the rest of the algorithm</td>
<td>Based upon current avoidance algorithm’s processing time, as well as known data for smartphone scan rate</td>
<td>A,I</td>
</tr>
<tr>
<td>5</td>
<td>Device Compatibility</td>
<td>The percentage of the smartphone market that will be able to run the application</td>
<td>Approximately 68% of the smartphone market is Android, and Android-specific code (Java) would cover all these phones. The value of 90% is an approximation of extending code to the Apple market (coded in Objective C)</td>
<td>S,I</td>
</tr>
<tr>
<td>6</td>
<td>Aircraft Compatibility</td>
<td>The percentage of aircraft that can use the smartphone application to perform collision avoidance</td>
<td>Compatibility should be purely based on the smartphone device, not on the aircraft that carries it</td>
<td>S,I</td>
</tr>
<tr>
<td>7</td>
<td>Repeatability</td>
<td>The ability to reproduce numbers within a small uncertainty from one run to the next</td>
<td>99% requires a high level of consistency, while remaining realistic to the possibility of sensor or processing errors</td>
<td>A,T</td>
</tr>
<tr>
<td>8</td>
<td>GPS Availability</td>
<td>The number of satellites that have an established link with the smartphone at any one time</td>
<td>A minimum of 3 Satellites is required to triangulate and get an accurate position, but 4 is desired to get a signal better than the bare minimum</td>
<td>A,T</td>
</tr>
<tr>
<td>9</td>
<td>Calibration Time</td>
<td>The amount of time the pilot must take to calibrate the smartphone to his specific attitude/position and aircraft</td>
<td>Speaking with pilots to determine their preferences</td>
<td>A,T</td>
</tr>
</tbody>
</table>

Note:
A- Analysis
S-Similarity to Existing Designs
I- Inspection
T- Test
Method of Approach

The flow chart shown in Figure 1 represents the process that was followed in order to complete this project and develop the best solutions. This section will discuss the details of each step but will leave the timeliness to another proceeding section.

Step 1. Generate Concepts and Requirements
This was the brainstorming phase for multiple aspects of the project. The following areas required brainstorming: test methods, algorithm structure, phone sensor usage, and what truth sources to use or create. Once ideas for each of these areas were developed, requirements were created in order to give guidelines for experiments and evaluation. This step had to be creative yet well focused on achieving practical ideas.

Step 2a. Test Phones in Experiments
Next, many experiments were run to get as much data from the phone/sensor as possible. These experiments had the goal of determining the accuracy of our position and attitude. In order to do this, the experiments needed to accurately simulate flight and have a valid truth source to compare against. Experimental data from a test flight and from experiments using a rate table already existed and were examined and evaluated.

Step 2b. Algorithm Development
Running parallel with the experiments was the algorithm development. The algorithm takes the data from the phone sensor acquired during the experiment and transfers the flight parameters (latitude, longitude, altitude, roll, pitch, yaw, etc...) that the algorithm team needs to run their program. This phase of the project was done at the same time as the experiments in order to be able to sense what kind of data the phone is giving and redefine the experiments if necessary. With a fast and efficient algorithm, the experimental data is much easier to decipher.

Figure 1. Methodology Flow Chart
**Step 3. Validate Data**

For each experiment, there were different sets of data based on the constraints of the procedure, and each experiment had its own truth source to rely on. This included vibrational testing, testing parameters by inspection, and validating functions in Matlab. To make sure that the experiment was useful, we compared the phone’s results with the truth source to make accurate assumptions about the resulting errors. Thus, having a very accurate truth source was absolutely essential to this process.

**Step 4. Evaluate Results**

With honest data from the experiments, the usefulness of the results was evaluated. This was a matter of whether or not the errors and uncertainties in the calculated parameters were within the defined tolerances specified in the requirements. Once the project reaches the step of meeting accuracy requirements in a consistent manner, the results can be integrated into the algorithm, and eventually the phones/algorithm can be tested in real world scenarios.

**Step 5. Test Phones/Algorithm**

Finally, it was time to see if our experiments accurately model real flight scenarios. The phone was taken into a plane and tested once again against a truth source. We then compared the difference between our results and the truth source against the engineering requirements outlined in Table 1. Since the data did not meet the requirements on the first attempt, the process of concept generation was repeated iteratively to seek alternative solutions. The results that fell outside of our requirements were documented and error sources were recorded. This way, the next senior project group can pick up where we left off.

**Design Development**

**Description of Algorithm Concepts**

Before actually writing code to meet the requirements of this project, we needed to detail the top-level software architecture which the code would follow. The first design that we came up with is the “Single Module with Subclasses” model (Model 1), which is displayed in Figure C1. For this concept, all variables are defined and raw data is imported into the “Sensor” class. Within this class is a “Data Conditioning” subclass, which contains everything for turning the input data into usable output parameters. Within the “Data Conditioning” subclass exist two further sub-modules: “Attitude” and “Position”. The primary variables for attitude (heading, bank, and pitch) are computed within the “Attitude” sub-module, while the primary variables for position (latitude, longitude, and altitude) are computed within the “Position” sub-module. After these parameters are computed, they are forced through a health monitoring stage within the “Data conditioning” subclass which determines whether or not these parameters are valid. If the values are determined to be healthy, they proceed onward, but if they are unhealthy, they need to be estimated. If this estimation has a low level of uncertainty, it will proceed onward, but if it has a high level of uncertainty, the parameter will be assigned a “null” value. Assuming it makes it through health monitoring, all secondary values will be calculated, as well as the uncertainty associated with both the
primary and secondary values. At this point, the values will be assigned to their respective variables, and placed in the “Sensor” class where they can be retrieved by the algorithm.

The second software architecture that we came up with is the “Single Module with Parallel Conditioning” model (Model 2), which is displayed in Figure C2. Similar to our first model, this software architecture accepts a set of input values, conditions and evaluates them, and outputs them to be used by the algorithm. However, the method in which it achieves this objective is slightly different. In Model 2, we once again contain everything within a “Sensor” class, and begin with defining the variables, inputting raw data from the phone, and setting user inputs. But unlike the first model, this software design performs health monitoring on the raw data before it is conditioned. Since the raw data will be noisy, the health monitoring needs to be slightly more relaxed, but would also avoid performing unnecessary data conditioning on unhealthy data. Next, considering that the data passes the health monitoring stage, it would proceed into the data conditioning subclass. However, unlike Model 1, there are no primary and secondary values, but instead, all of the required parameters are computed in parallel. Once these parameters are computed, the uncertainty is determined for each of these. Finally, once each of these values and their associated uncertainties are computed within data conditioning, the conditioned data is outputted and assigned to each variable in the “Sensor” class.

The third software architecture is the “Tri-Module Flow” model (Model 3), which is shown in Figure C3. This software architecture is quite different from the other two in the way it is structured. Instead of being entirely contained in one module, it is contained in three main modules: “Attitude”, “Position”, and “Sensor”. In this architecture, these modules run separately but in parallel, as variables are defined and raw data is inputted into either the “Attitude” or the “Position” module. This continues as the data is conditioned separately in the respective modules. Similar to Model 1, the primary variables for attitude are calculated within the “Attitude” module, and the primary variables for position are calculated within the “Position” module. After the data conditioning, uncertainties are determined within each module, and health monitoring is performed to determine whether or not the primary variables are valid. After the health monitoring stage, the primary variables are set within each of the respective classes, and then inputted into the “Sensor” module. Inside the “Sensor” module, the primary parameters that are pulled in get used to compute the secondary variables and their associated uncertainties. Finally, the secondary variables are set and assigned to their variables within the “Sensor” class. When the algorithm requests these parameters, it will need to pull the primary variables from either the “Attitude” or “Position” module, and pull the secondary variables from the “Sensor” module.

Algorithm Design Selection
After developing these concepts for the top-level software architecture, we had to decide on one conceptual model that would mostly accurately meet our objectives and requirements. The main attributes our architecture needed to possess was efficiency, modularity, and the ability to interface with the current algorithm smoothly. Based on these specific attributes, we developed a simple design matrix to decide which architecture our software is going to follow.
Each model was scored on a range of 0-5 for each attribute which was weighted according to importance for meeting the project requirements. The first attribute chosen was efficiency which is referring to minimizing the time that our algorithm can take in raw data and return wanted variables with associated uncertainties. This attribute is very important to the whole project because the main algorithm can only refresh as fast as the sensor data can. The next attribute these models were scored on was modularity. Modularity refers to the ability to isolate parts of the code in order to locate errors, allow versatility in devices and aircraft, and keep the code compact and centrally located. The best way to complete this is with sub-modules rather than many large modules which can add complication. Modularity and efficiency are equally needed attributes to succeed and thus are both weighted at 40% of the entire attribute spectrum. The last attribute these models were scored on was the ability to interface with the current algorithm. We are adding our sensor architecture into a much larger project with a previously defined architecture. It is imperative that our module fits into this structure smoothly without interference but it is something that can be worked around if necessary, which is why it received a 20% weighting.

Once the attributes and weighting scheme were chosen, each module was ranked and the total score for each was added up. Model one scored the highest with 4.2 points, which is distinguishably higher than model 2 and 3 which scored a 3.4 and 2.8 respectively. Model one received high marks in all attributes. It provides a structure that allows efficiency and modularity by containing all calculations within sub-modules of a single sensor module. Also, it lists all outputs within this single class and can be passed into the existing algorithm with ease.

### Table 3. Algorithm Decision Matrix

<table>
<thead>
<tr>
<th>Weighting Factor</th>
<th>Single Module with Subclasses</th>
<th>Single Module with Parallel Conditioning</th>
<th>Tri-Module Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficiency</td>
<td>0.4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Modularity</td>
<td>0.4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Algorithm Interface</td>
<td>0.2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td><strong>4.2</strong></td>
<td><strong>3.4</strong></td>
<td><strong>2.8</strong></td>
</tr>
</tbody>
</table>

### Description of Mounting Fixtures

There are many types of universal phone mounting fixtures available on the market ranging in price from two dollars to upwards of fifty dollars. The majority of options include a suction cup for adhesion to the windshield or dashboard, an adjustable arm, and a clamping mechanism to hold the phone in place. In order to test multiple designs, four unique mounting setups were chosen to be tested. Cost, customer ratings, and features were taken into account while deciding on which products to purchase. Pictures and specifications of these products can be viewed in Figures. The Arkon Mega Grip mount has the typical features described above and had good ratings and reviews for stability and durability on Amazon. The iOttie Easy Flex was picked because rather than a long arm for adjusting the position of the phone it tilts the phone about one axis and rotates the base. This was viewed positively because it could
reduce oscillation and the moment at the suction cup. The third mount is a small tube with suction cups at both ends made of PVC called the Barnacle. The unique design and small price of $1.95 made this product a prime candidate for testing. In addition to the three purchased mounts, Velcro was chosen to for testing because of its small size and low cost.

Figure 2. Arkon Mega Grip Windshield & Dash Smartphone Mount,
Price: $15.89
Product Dimensions: 11 x 6 x 3 inches ; 7.5 ounces

Figure 3. iOttie Easy Flex Universal Car Mount
Price: $12.99
Product Dimensions: 6.1 ounces
Mounting Fixture Analysis
The phone mounts were tested and evaluated based on their responses to sinusoidal vibration. Each mount was fastened to a shake table and subjected to a logarithmic sine sweep test beginning at 5 Hz and ending at 2000 Hz as per MIL-STD 801F. The tests were repeated in the X and Z axes. The amplitude ratio was graphed against frequency, and the peak amplitude ratio as well as the average over the range of frequencies was recorded and compared for each fixture. These graphs as well as the detailed testing procedure, setup, and photos can be seen in Appendix E.

Mounting Fixture Selection
To determine which mounting setup was the best, the decision matrix shown in Table 4 below was developed. Upon initial inspection of the Barnacle mount it was determined that with very little force the mount would shake too much to safely keep the phone in place. In an effort to not damage any equipment the Barnacle was rejected from testing. For the remaining three fixtures, amplitude ratio data from tests described above were compared. For the Arkon Mega Grip mount, the peak amplitude ratios were 17.2 dB and 16.6 dB in the x and z axes respectively. The iOttie performed slightly better with peaks of 16.9 dB and 6.5 dB while the Velcro showed the lowest ratios of 4.1 dB and 10.23dB. The average amplitude ratios were also calculated and the iOttie and Velcro mounts had the lowest averages.

Since the phone is to be mounted in the cockpit of the aircraft where space is limited, adjustability and size were important in selecting an appropriate fixture. The pilot will want to be able to use the phone and see the screen to receive alerts. A fixture that is easily adjusted gives the pilot freedom to mount the phone where he or she pleases. Ease of installation was also taken into consideration. Velcro is easy to apply but requires the pilot to have Velcro attached to their phone at all times, which may not be
convenient. The suction cup on the iOttie mount had an adhesive film which made it more difficult to remove and reposition than the Arkon which had a locking suction cup.

Reliability scores were based on visual inspection during testing. During testing it was observed that because the accelerometer was mounted to the fixture rather than the phone itself, the data collected may not completely capture the effect of the vibration. In the case of the iOttie, the fixture stayed rigidly mounted, but the phone vibrated within the fixture. The Arkon however had the most secure and stable clamping mechanism to hold the phone in place.

Table 4. Mounting Fixture Decision Matrix

<table>
<thead>
<tr>
<th>Weighting Factor</th>
<th>Arkon</th>
<th>iOttie</th>
<th>Velcro</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak Amplitude Ratio</td>
<td>0.15</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Average Amplitude Ratio</td>
<td>0.15</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Adjustability</td>
<td>0.2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Size</td>
<td>0.2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Reliability</td>
<td>0.2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Installation</td>
<td>0.1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3.7</strong></td>
<td><strong>3.5</strong></td>
<td><strong>3.65</strong></td>
</tr>
</tbody>
</table>

Methods of Attitude Estimation

Method #1: Components of gravitational acceleration
The first method that was tested was to simply estimate the attitude of the aircraft using the gravitational acceleration components along each axis of the phone. Since gravity is always 9.81 m/s² towards the center of the Earth, the components of gravity along each axis could be measured to quantify the attitude angles of the aircraft.

Since the majority of general aviation flight is with aircraft that move at relatively slow velocities and do not move in a very aggressive manner, this method depends on the expectation that normal accelerations during a bank or pitch maneuver are small enough to be neglected. Furthermore, this method is very simple and requires very little computing power or delay.

Method #2: Flight path in 3D space from GPS
This method builds on the previous method, but includes normal acceleration by integrating both the accelerometer and GPS sensors. In three-dimensional space, three GPS points can be used to define a circular path in terms of an Earth-Centered Earth-Fixed (ECEF) coordinate system. Based on the equation of this circle in three-dimensional space, it is possible to determine the radius of curvature for the aircraft’s flight path. Concurrently, the aircraft’s velocity is possible to calculate based on the aircraft’s change in distance over a known time interval. Then, the aircraft’s normal acceleration can be
calculated based on the velocity and radius of curvature at any given time, and the tangential acceleration can be calculated based on the change in velocity over time. Since the equation of the circle is known, the direction of the normal vector and tangential vector (in ECEF coordinates) along which this acceleration acts is known as well. Combining together the normal and tangential acceleration, it is possible to finally identify the direction and magnitude of the load acceleration acting on the aircraft at any given time.

Now that all the accelerations acting on the aircraft are known (gravitational acceleration, load acceleration), all accelerations acting on the aircraft can be balanced and compared against the acceleration measured by the phone’s accelerometer sensors. However, since the acceleration measured by the phone is in the aircraft’s coordinate system, rotation matrices are needed to get all of the accelerations into the same coordinate system. This matrix transform involves the attitude angles of the aircraft and since the load acceleration and gravitational acceleration along the phone’s X, Y, and Z axis are all dependent upon the attitude of the aircraft, a force balance yields six equations and six unknowns (attitude angles and gravitational acceleration along each axis). So finally, this set of equations can be iteratively solved for the bank, pitch, and yaw angles that satisfy the system of nonlinear equations.

It should also be noted that since GPS data is collected about once per second and previous GPS data is used, this method can cause a certain amount of lag in determining attitude. However, it is expected that since only three GPS points are used at a time, this small amount of lag should be negligible.

**Method #3: Flight path from change in heading**

The final method considered to determine the attitude of the aircraft is very similar to the second method, but uses a change in heading in addition to GPS data to calculate normal acceleration. The heading of the aircraft is acquired by integrating the data measured by the gyroscopes and magnetometer. Assuming a level bank, two GPS points define an arc in space. For this arc, the change in heading from one point to the next is equal to the angular portion of the arc. Then from GPS, the distance between the two points is known and for a given time interval the velocity is known as well. Based on the angular portion of the arc and the distance between the two points on the arc, the radius of curvature for the flight path can be determined. From here, the velocity and radius of curvature can be used to compute the normal acceleration, and the attitude angles are a function of the normal acceleration.

It is important to note that this method does not necessarily require accurate heading, but requires an accurate and consistent change in heading. However, even an accurate change in heading can often be difficult to achieve since the gyroscopes are very sensitive to vibration and the magnetometer is very sensitive to the presence of metal and magnets (which can be quite common in an aircraft cockpit). In order for this method to work, the heading data needs to be consistent and reliable. One benefit of this method is that since heading data is retrieved every 20 ms and GPS is retrieved only once per second, a large amount of heading data can be averaged in between each GPS data point to acquire a more accurate measurement of heading.
Final Design

Description of Selected Mounting Fixture
The Arkon Mega Grip mounting fixture was selected based on the decision matrix described above. The margin was small between the three mounts but the sturdy and highly adjustable design of the Arkon fixture and the ease of installation made it the best option.

Description of the Software Layout
In the course of designing and testing the algorithm, three separate programs were used to complete different tasks. The first, was coded in JAVA while the others were completed using MATLAB. The following sections will outline the purpose of each program, as well as detail all the functions used within each program.

1. Sensor Test (JAVA)
Sensor Test is the main program that the phone uses to compute all of the needed parameters. This program follows the form of the flow chart detailed earlier in the report and is used to feed data to the collision avoidance algorithm. This code controls the interface of the phone and was the program that performed all calculations during test flights. Simulation testing and function verification was not completed within this program itself as it took too much time between runs to be used effectively.

Sensor Test Functions Explained

Filter Raw
This function is used to filter the following raw sensor values: accelerometer x-axis, accelerometer y-axis, accelerometer z-axis, gyroscope x-axis, gyroscope y-axis, and gyroscope z-axis. It takes each of these variables and subjects them to a moving average filter. The following equation is implemented:

\[
\text{Filtered Value} = (\text{Current Raw Value} \times k\text{Filter}) + (\text{Last Filtered Value} \times (1 - k\text{Filter}))
\] (1)

The constant kFilter is the weighting factor for the moving average filter. It is a number between 0 and 1 that controls how much the last filtered values affect the current one. The closer kFilter is to 1, the more effect the current raw value will have on the new filtered value. The value used in this function is .05 which means that the new value only accounts for 5% of the new filtered value. This filtering method will reduce the effects of random spikes in the sensor data. These new values for the accelerometer and gyroscope are used in all later functions that require them.

Calc Pitch
The purpose of this function is to calculate the pitch of the aircraft using a complimentary filter. A complimentary filter incorporates two different types of filters for two different sensors. In this
function a low pass filter is used on the accelerometer values and a high pass filter is used on integrated gyroscope values. The low pass filter allows the accelerometer to be used for steady state values while the high pass filter on the gyroscope accounts for dynamic movements.

The way pitch is calculated using the accelerometers is as follows:

\[ \text{Pitch accel} = -\tan^{-1}\left(\frac{A_z}{\sqrt{A_y^2 + A_x^2}}\right) \]  

(2)

To get the change in pitch using the integrated gyroscope values the equation is:

\[ \text{Pitch gyro} = \text{sensor rate} \times [\text{current gyro y} + 1/2 \times (\text{last gyro y} - \text{current gyro y})] \]  

(3)

where sensor rate is how fast the sensor is updating, which is set at 20 ms for all sensors.

Once these two forms of pitch have been created they are combined with a complimentary filter as shown to get the filtered pitch value:

\[ \text{Pitch} = (0.9) \times (\text{last pitch value} + \text{pitch gyro}) + (0.1) \times (\text{pitch accel}) \]  

(4)

**Calc Bank**

The logic is the same as that of the calc pitch function except that the equations are slightly different. The way bank is calculated using the accelerometers is as follows:

\[ \text{Bank accel} = -\tan^{-1}\left(\frac{A_y}{\sqrt{A_x^2 + A_z^2}}\right) \]  

(5)

To get the change in bank using the integrated gyroscope values the equation is:

\[ \text{Bank gyro} = \text{sensor rate} \times [\text{current gyro x} + 1/2 \times (\text{last gyro x} - \text{current gyro x})] \]  

(6)

where sensor rate is how fast the sensor is updating, set to 20 ms for all sensors.

Once these two forms of pitch have been created they are combined with a complimentary filter as shown to get the filtered bank value:

\[ \text{Bank} = (0.9) \times (\text{last bank value} + \text{bank gyro}) + (0.1) \times (\text{bank accel}) \]  

(7)
 Heading

There is not an actual function called out to calculate heading. It is calculated in the method “On Sensor Changed” by using the following Sensor Manager Functions:

\[
\begin{align*}
\text{SensorManager.getRotationMatrix();} \\
\text{SensorManager.remapCoordinateSystem();} \\
\text{SensorManager.getOrientation();}
\end{align*}
\]

This series of functions calculates the orientation of the phone through rotation matrices and takes into account if the phone is in portrait or landscape. To read further on the above functions, please refer to: [http://developer.android.com/reference/android/hardware/SensorManager.html](http://developer.android.com/reference/android/hardware/SensorManager.html)

Calc Climb Rate

This function takes the derivative of the change in altitude based only on GPS. Then it applies a moving average filter to the climb rate to even out spikes in the altitude data. The following equation calculates raw climb rate:

\[
\text{Raw Climb Rate} = \frac{(\text{current altitude} - \text{last altitude})}{\text{delta time}}
\]

where delta time is the time change from the last GPS point to the current one.

Next, the raw climb rate is subjected to a moving average filter with a filtering constant, kFilter, of 0.1. The equation is as follows:

\[
\text{Filtered Climb Rate} = \frac{(\text{Current Raw Climb Rate} \times k\text{Filter}) + (\text{Last Filtered Climb Rate} \times (1 - k\text{Filter}))}{\text{delta time}}
\]

Convert to ECEF

This takes the following inputs (lat, lon, altitude) and converts it into x, y and z values in ECEF coordinates which is measured as feet from the center of the Earth. See picture below for coordinate directions.

This coordinate system is useful for the calculation of ground speed. The following equations outline the process of converting a geoposition (lat,lon,alt) to an ECEF position (x,y,z) in feet:

\[
\begin{align*}
x &= (aC + \text{altitudeGPS}) \times \cos(\text{lat}) \times \cos(\text{lon}) \\
y &= (aC + \text{altitudeGPS}) \times \cos(\text{lat}) \times \sin(\text{lon}) \\
z &= (aS + \text{altitudeGPS}) \times \sin(\text{lat})
\end{align*}
\]
\[ C = \frac{1}{\sqrt{\cos^2(lat) + (1 - f)^2 \cdot \sin^2(lat)}} \]  

(10d)

\[ S = (1 - f)^2 \cdot C \]  

(10e)

\[ a = 20925647 \text{ ft (earth’s radius in WGS84)} \]

\[ f = \frac{1}{298.257224} \text{ (flattening parameter)} \]

where altitude is in feet and lat/lon are in radians.

**Calc Ground Speed**

Now that each point of flight has been calculated as an x,y and z value, the ground speed can be computed using a simple 3-D distance formula over the time elapsed between these points. The following formula will compute ground speed in feet per second:

\[ \text{Ground Speed} = \frac{\sqrt{(x(1) - x(2))^2 + (y(1) - y(2))^2 + (z(1) - z(2))^2}}{t(1) - t(2)} \]  

(11)

where 1 is the current location and 2 is the last location.

To get into knots, multiply the computed ground speed in fps by conversion factor of 0.59248.

The ground speed is also subjected to a moving average filter with the averaging constant, kFilter, equal to 0.3. This is the moving average filter:

\[ \text{Filtered Ground Speed} = (\text{Current Raw Ground speed} \times k\text{Filter}) + (\text{Last Filtered Ground Speed} \times (1 - k\text{Filter})) \]  

(12)

**Calc Alt Baro**

This function is used to calculate the altitude from the barometer sensor if the phone has this sensor installed. Currently this altitude is never used in other functions because the GPS altitude is being implemented. This altitude is computed in meters from sea level based on the pressure changes felt by the sensor. If no barometer exists, this variable is set to zero. The following built in android method is used to calculate the altitude:

\[ \text{Altitude Baro} = \text{SystemManager. getAltitude(pressure @ sea level, current baro pressure value)} \]

**Calibrate**

The calibrate function is called when the calibrate button is pressed on the app. Once this button has been hit once the button is disabled which only allows the user to calibrate one time once the
app is running. This function will offset the bank and pitch of the phone to zero wherever the user is holding the phone. For testing, this button would be pressed once the phone is in the mount and not going to be moved. It does this by setting the current measured bank and pitch as the offset. These offsets are subtracted from the new calculated bank and pitch values, allowing the point at which the calibrate button was pressed to be the new datum.

**Health Monitoring**

This function is used to monitor the health of the sensors and the algorithm itself during the flight. It creates a Boolean called isHealthy which is false if a computed variable falls out of a predetermined “healthy” range. These ranges are rather large currently and need to be more constricting as the app gets further in development. The following limitations are currently in place:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Max</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heading</td>
<td>180deg</td>
<td>-180deg</td>
</tr>
<tr>
<td>Bank</td>
<td>90 deg</td>
<td>-90 deg</td>
</tr>
<tr>
<td>Pitch</td>
<td>90 deg</td>
<td>-90 deg</td>
</tr>
<tr>
<td>Latitude</td>
<td>90deg</td>
<td>-90deg</td>
</tr>
<tr>
<td>Longitude</td>
<td>180deg</td>
<td>-180deg</td>
</tr>
<tr>
<td>Altitude</td>
<td>18000 ft</td>
<td>0 ft</td>
</tr>
<tr>
<td>Roll Rate</td>
<td>45 rad/s</td>
<td>N/A</td>
</tr>
<tr>
<td>Climb Rate</td>
<td>50 ft/s</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Write CSV Header Raw Sensor**

This function is called every time the “start algorithm” button is pressed to signify a new file needs to be created for a new run. This function creates the header for the file that contains the raw sensor values for: Accelerometer (x,y,z), Gyroscope(x,y,z), Magnetometer (x,y,z), Orientation (x,y,z), and the barometric pressure reading. A column with each of these variables is created in the file and ready to have data recorded to it. The file created is saved to the SD card within the phone. If there is no SD card available it is saved to internal storage of phone. Each file name has the following format yyyy_MM_dd_hh_mm_ss-phoneSensorRaw.csv where the time refers to the phones internal time when the file is created.

**Write CSV Raw Sensor**

This function is called every time a new data set is computed (every 20 ms) and it fills in the columns that the “Write CSV Header Raw Sensor” has preset with the most current sensor data. This function will continue to be called until the “stop algorithm” button is pressed and data is no longer being recorded.
**Write CSV Header Filter Sensor**

This function is called every time the “start algorithm” button is pressed to signify a new file needs to be created for a new run. This function creates the header for the file that contains the filtered sensor values for: Accelerometer \((x,y,z)\) and Gyroscope \((x,y,z)\). A column with each of these variables is created in the file and ready to have data recorded to it. The file created is saved to the phones SD card. If there is no SD card available it is saved to internal storage of phone. Each file name has the following format `yyyy_MM_dd_hh_mm_ss-phoneSensorFiltered.csv` where the time refers to the phones internal time when the file is created.

**Write CSV Filter Sensor**

This function is called every time a new data set is computed (every 20 ms) and it fills in the columns that the “Write CSV Header Filter Sensor” has preset with the most current filtered sensor data. This function will continue to be called until the “stop algorithm” button is pressed and data is no longer being recorded.

**Write CSV Header Parameters**

This function is called every time the “start algorithm” button is pressed to signify a new file needs to be created for a new run. This function creates the header for the file that contains the computed flight parameters: GPS time(ms), Lat(deg), Lon(deg), Altitude GPS (ft), Altitude Baro (ft), Bank(deg), Pitch(deg), Heading(deg), Ground Speed (kts), Climb Rate (fps), Roll Rate (deg/s), and Health Status (T/F). A column with each of these variables is created in the file and ready to have data recorded to it. The file created is saved to the phones SD card. If there is no SD card available it is saved to internal storage of phone. Each file name has the following format `yyyy_MM_dd_hh_mm_ss-flightParameters.csv` where the time refers to the phones internal time when the file is created.

**Write CSV Parameters**

This function is called every time a new data set is computed (every 20 ms) and it fills in the columns that the “Write CSV Header Raw Sensor” has preset with the most current flight parameter data. Since this file contains values that have varying sample rates, the data based on GPS will hold the same value for around one second until GPS is updated, while parameters based on the phones sensors will change every step (20 ms). This function will continue to be called until the “stop algorithm” button is pressed and data is no longer being recorded.

**Round**

There are two rounding functions that exist in the algorithm, one for doubles and one for floats. These functions have the input of the number being rounded, and the amount of decimal places to round that number to. The output is a number that is rounded to the appropriate number of decimals inputted. These functions are mainly used to cut down on the number of decimal places displayed on the screen so it is more visually appealing.
**Do Calcs**

This is one of two threads running in the algorithm and it is where most of the functions are called. It controls the scan rate and allows for new data lines to be measured, computed and recorded. The following functions are called in this thread:

- `writeCSVHeaderRawSensor`
- `writeCSVHeaderFilterSensor`
- `writeCSVHeaderParameters`
  - `filterRaw`
  - `calcAltBaro`
  - `calcBank`
  - `calcPitch`
  - `convertToEcef(lat,lon,altGps[0])`
  - `calcGroundSpeed`
  - `calcClimbRate`
  - `calcRollRate`
- `writeCSVRawSensor`
- `writeCSVFilterSensor`
- `writeCSVParameters`

“Do Calcs” is also where the pitch and bank are corrected with the offset computed with the calibrate function. Lastly, this function converts the pitch, bank, and yaw to degrees before recording or displaying the data.

**Update View**

The purpose of this thread is to control what the user views on the display of the phone. It updates the screen every 200 ms for the following display outputs: Pitch, Bank, Yaw, Lat, Yaw, Lat, Lon, Altitude, Ground Speed, Roll Rate, and Climb Rate.

2. **Plot SAS (MATLAB)**

Plot SAS was the program used to test and verify most functions used in Sensor Test. The functions were translated from JAVA to MATLAB and sensor data was imported from the CSV files recorded during the first test flight in the summer of 2012 by NASA. These test points were run through the various functions needed to compute all the necessary flight parameters and plots were created. These plots were an effective way to compare the Sensor Test functions to expected results from the G1000 truth source data from the same test flight. Changes in the functions were first made in this program and the changes were then re-translated back into the Sensor Test program once they had been verified. Note that verification was also shown by using excel to compare the results of the functions to the inertial lab data taken in the summer of 2012 at NASA Dryden. The following functions were tested and verified using Plot SAS: Filter Raw, Calc Pitch, Calc Bank, Calc Climb Rate,
Convert to ECEF, and Calc Ground Speed. To get a better explanation of these functions refer to the above Sensor Test Functions Explained section.

Attitude Estimation Method 2 was also developed and tested with Plot SAS where a set of ideal data was created to test the most simple case before moving on to the noisy test flight data.

Plot SAS Functions Explained

Main Class

The main class for this program is called plotSAS.m and its purpose is to import all of the data from three sources (G1000, phone sensor, phone GPS). It takes this data and passes it into the functions being verified. These computed outputs from the functions are then sent back to plotSAS and they are plotted against truth sources to compare the validity of the function. To make sure all of the different types of variables could be plotted against each other, unique time vectors were created for each data set. Top plot more variables follow the form in plotSAS and be sure to use the appropriate time vector. Below details some of the functions used within plotSAS.

Flight Test Point 2 File Name (FlightTestPoint2FileName)

This function allows the user to input a certain test point for the flight test occurring in the summer of 2012 and links it with the data files associated with that test run. These data files include the G100 data, phone raw sensor data, phone GPS data, and phone computed flight parameters data. For this flight test there were only test points 1 – 27 and any other input will result in an error. The input is found in the plot SAS m-file.

Import G1000 File (ImportG1000File)

This function is called to import the data from the G1000 output file. The input to this function is a filename which is acquired by the function flight test point 2 file name and the output is a data structure containing numerous truth source variables for a single test point. The returned data file is organized so that each data.G1000Variable links to an entire column from the CSV G1000 file. The following parameters are available from this function:
### Table 6. List of Available G1000 Parameters

<table>
<thead>
<tr>
<th>Parameter (column in file)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date of test flight as a string in yy-mm-dd format (1)</td>
</tr>
<tr>
<td>time</td>
<td>Local time as a string in hh-mm-ss format (2)</td>
</tr>
<tr>
<td>utcOffset</td>
<td>The offset from local time to UTC time as a string in hh-mm format (3)</td>
</tr>
<tr>
<td>lat</td>
<td>Latitude in deg (4)</td>
</tr>
<tr>
<td>long</td>
<td>Longitude in deg (5)</td>
</tr>
<tr>
<td>alt_baro</td>
<td>Altitude from Barometer in feet (6)</td>
</tr>
<tr>
<td>alt_msl</td>
<td>Altitude from GPS in ft MSL (7)</td>
</tr>
<tr>
<td>alt_gps</td>
<td>Altitude from GPS in feet (8)</td>
</tr>
<tr>
<td>ias</td>
<td>Indicated Air Speed in knots (9)</td>
</tr>
<tr>
<td>tas</td>
<td>True Air Speed in knots (10)</td>
</tr>
<tr>
<td>gndSpd</td>
<td>Ground Speed in knots (11)</td>
</tr>
<tr>
<td>vertSpd</td>
<td>Vertical Speed in ft/min (12)</td>
</tr>
<tr>
<td>vertSpdGps</td>
<td>Vertical Speed based only on GPS in ft/min (13)</td>
</tr>
<tr>
<td>pitch</td>
<td>Pitch Angle in deg (14)</td>
</tr>
<tr>
<td>roll</td>
<td>Bank Angle in deg (15)</td>
</tr>
<tr>
<td>rpm</td>
<td>Engine RPM in rev/min (16)</td>
</tr>
<tr>
<td>latAccel</td>
<td>Lateral Acceleration in g’s (17)</td>
</tr>
<tr>
<td>normAccel</td>
<td>Normal Acceleration in g’s (18)</td>
</tr>
<tr>
<td>heading</td>
<td>Heading in deg (19)</td>
</tr>
<tr>
<td>gndTrack</td>
<td>Ground Track in deg (20)</td>
</tr>
<tr>
<td>fuelLeft</td>
<td>Fuel in left wing (21)</td>
</tr>
<tr>
<td>fuelRight</td>
<td>Fuel in right wing (22)</td>
</tr>
<tr>
<td>airTemp</td>
<td>Outside Air Temp in deg C (23)</td>
</tr>
<tr>
<td>wndSpd</td>
<td>Wind Speed in knots (24)</td>
</tr>
<tr>
<td>wndDir</td>
<td>Wind Direction in deg (25)</td>
</tr>
<tr>
<td>magVar</td>
<td>Magnetic Variance in deg (26)</td>
</tr>
<tr>
<td>baroPress</td>
<td>Barometric Pressure in inches of Mercury (27)</td>
</tr>
<tr>
<td>flyupBool</td>
<td>Fly up bool in (T/F) (28)</td>
</tr>
</tbody>
</table>

**Import Cell Phone Sensor File (ImportCellPhoneSensorFile)**

This function is called to import the data from the cell phone sensor output file. The input to this function is a filename which is acquired by the function `flight test point 2 file name` and the output is a data structure containing numerous sensor variables for a single test point. The returned data file is organized so that each `data.SensorValue` links to an entire column from the CSV cell phone sensor file. The following parameters are available from this function:
Table 7. List of Sensor Parameters from Cellphone

<table>
<thead>
<tr>
<th>Parameter (column in file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data, h</td>
</tr>
<tr>
<td>m</td>
</tr>
<tr>
<td>s</td>
</tr>
<tr>
<td>a_x</td>
</tr>
<tr>
<td>a_y</td>
</tr>
<tr>
<td>a_z</td>
</tr>
<tr>
<td>g_x</td>
</tr>
<tr>
<td>g_y</td>
</tr>
<tr>
<td>g_z</td>
</tr>
<tr>
<td>m_x</td>
</tr>
<tr>
<td>m_y</td>
</tr>
<tr>
<td>m_z</td>
</tr>
<tr>
<td>o_x</td>
</tr>
<tr>
<td>o_y</td>
</tr>
<tr>
<td>o_z</td>
</tr>
<tr>
<td>time_synch</td>
</tr>
<tr>
<td>time</td>
</tr>
</tbody>
</table>

Import Cell Phone GPS File (ImportCellPhoneGPSFile)

This function is called to import the data from the cell phone GPS output file. The input to this function is a filename which is acquired by the function flight test point 2 file name and the output is a data structure containing numerous GPS variables for a single test point. The returned data file is organized so that each data.GpsValue links to an entire column from the CSV cell phone GPS file. The following parameters are available from this function:

Table 8. List of GPS Parameters from Cellphone

<table>
<thead>
<tr>
<th>Parameter (column in file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data, H</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>Lat</td>
</tr>
<tr>
<td>Lon</td>
</tr>
<tr>
<td>alt</td>
</tr>
<tr>
<td>vel</td>
</tr>
<tr>
<td>bearing</td>
</tr>
<tr>
<td>time_synch</td>
</tr>
<tr>
<td>time</td>
</tr>
</tbody>
</table>
Calculate Radius of Curvature Method 2 (calcRadCurvature)

The overall goal of method 2 was to use the radius of curvature and velocity to calculate normal acceleration. This function calculates the radius of curvature of the flight path using three GPS points. The radius of a circle circumscribing a triangle is a function of the area of the triangle and lengths of its sides. To find the area of the triangle defined by three points in three dimensional space, Heron’s formula was used. The formula states that for a triangle with lengths a, b, and c, semi perimeter, s, and area, A:

\[
\text{Area} = \sqrt{s \cdot (s - a) \cdot (s - b) \cdot (s - c)}
\]

\[
\text{where } s = \frac{a + b + c}{2}
\]

The lengths a, b, and c of the triangle were found using the distance formula. These lengths and the area found from the above formula were then used to calculate a radius of curvature, r, with the following equation:

\[
\text{radius of curvature} = \frac{a \cdot b \cdot c}{4 \cdot \text{Area}}
\]

Calculate Load Acceleration (calcLoadAccel)

This function uses ground speed in feet per second, time in seconds, and radius of curvature in feet. The output is the acceleration of the aircraft due the normal and tangential components in feet per second squared. Normal acceleration was calculated using ground speed and radius of curvature from calcRadCurvature using the following formula:

\[
\text{accel}_n = \frac{\text{ground speed}^2}{\text{radius of curvature}}
\]

The tangential acceleration was calculated by the change in ground speed over time.

\[
\text{accel}_t = \frac{\text{ground speed}_2 - \text{ground speed}_1}{\Delta \text{time}}
\]

The acceleration resulting from normal and tangential components was then combined to give a total load acceleration in each axis.

Calculate Normal Acceleration Vector (calcNormVector)

The purpose of this function is to output a unit vector from the last point in the direction of the center of the circle and use this to project the acceleration calculated by calcLoadAccel in the
correct direction. The function does this by using the GPS points used to calculate the radius of curvature to solve the equation for a circle.

\[ 0 = (x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - r^2 \]  \hspace{1cm} (17)

Where \( c_x, c_y, \) and \( c_z \) represent the center of the circle in the x, y, and z directions respectively. This equation has three unknowns, but by using the last three data points, fsolve can solve the system with three equations. With the center of the circle known, a unit vector can be determined by dividing the vector from the gps point to the center of the circle by the magnitude (radius).

\[ \text{unit vector} = \frac{1}{r} \times [(x - c_x)(y - c_y)(z - c_z)] \]  \hspace{1cm} (18)

By multiplying the unit vector by the normal acceleration, it is projected in the direction of the center of the circle.

\[ \text{normal vector} = \text{normal acceleration magnitude} \times \text{< unit vector >} \]  \hspace{1cm} (19)

**Calculate Tangential Acceleration Vector (calcTanVector)**

This function projects the tangential acceleration in the direction tangent to the flight’s path. It uses three GPS points, the normal vector calculated in calcNormVector, and load acceleration. This unit vector is calculated differently than the normal acceleration unit vector. First, two vectors are defined, \( v1 \) and \( v2 \) which are the vectors directing from the third GPS point to the first and second respectively. The cross product of \( v2 \) and \( v1 \) is taken to get a vector that is perpendicular to the plane defined by the flight path. This vector is then crossed with the normal vector to get the vector perpendicular to the normal acceleration and in the same plane as the flight path. This process shown by the following equations:

\[ v1 = \langle x3, y3, z3 \rangle \rightarrow \langle x1, y2, y3 \rangle \]  \hspace{1cm} (20a)

\[ v2 = \langle x3, y3, z3 \rangle \rightarrow \langle x2, y2, z2 \rangle \]  \hspace{1cm} (20b)

\[ \text{cross product one} = v2 \times v1 \]  \hspace{1cm} (20c)

\[ \text{cross product two} = \text{cross product one} \times \text{normal vector} \]  \hspace{1cm} (20d)

The unit vector tangent to the flight path is found by dividing the “cross product two” by its magnitude. This vector is then multiplied by the magnitude of the tangential acceleration to find the magnitude and direction of tangential acceleration.
**Calculate Bank Method 2 (calcBank2)**

In methods two and three, acceleration due to gravity and normal acceleration are used to calculate bank with a force balance.

\[
\text{bank} = \tan^{-1}\left( \frac{\text{Ground speed}^2}{\text{radius of curvature} \times \text{gravitational acceleration}} \right)
\]  
(21)

**Ideal Flight Parameters (SampleFlight and SampleFlightxz)**

These files create sets of data containing GPS points, time, and ground speed. The purpose of these ideal data sets was to test method two with data containing no error, and then add random noise to see how much the bank calculations would be effected. SampleFlight and SampleFlightxz create perfectly circular flight paths which travel in only two directions using the equation for a circle. X points were generated at equal intervals for the diameter of the circle and Y points were found using the equation for a circle given the radius and center.

\[
y = c_y + \sqrt{\text{radius}^2 - (x - c_x)^2}
\]  
(22)

SampleFlightxz was created after SampleFlight to test to verify that the method could be used in the \(z\) direction as well.

**Calculate Orientation (calcOrientation)**

This function calculates the orientation of the plane in the phone’s coordinates using the tangent and normal acceleration vectors and the acceleration measured with accelerometers. First, the total load acceleration is calculated in ECEF coordinates. The total load acceleration is reoriented for the direction of the phone’s axes with a rotation matrix. Because sensors update fifty times a second and GPS updates roughly once a second, a matrix is created to hold the GPS values for all sensor values until updated. The Matlab function fsolve is used to solve a system of six unknowns and six equations. For bank, \(\theta\), pitch, \(\phi\), and yaw \(\psi\), a rotation matrix is used to rotate the acceleration matrix in the plane’s coordinates to the coordinates of measured acceleration. This rotated load acceleration matrix is equal to the measured acceleration minus the gravity matrix. The flow of equations is as follows:

\[
A_{\text{tanECEF}} + A_{\text{normECEF}} = A_{\text{totalECEF}}
\]  
(23a)

\[
A_{\text{ECEF}} \cdot \begin{bmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} = A_{\text{plane}}
\]  
(23b)

\[
A_{\text{plane}} \cdot R_{3x3} = A_{\text{rotated}}
\]  
(23c)
\[ R_{3x3} = \begin{bmatrix} 
\cos(\theta) \cos(\psi) & \cos(\theta) \sin(\psi) & -\sin(\theta) \\
\sin(\phi)\sin(\theta)\cos(\psi) - \cos(\phi)\sin(\psi) & \cos(\phi)\cos(\psi) + \sin(\phi)\sin(\theta)\sin(\psi) & -\sin(\phi)\cos(\theta) \\
\sin(\phi)\sin(\theta)\sin(\psi) + \cos(\phi)\cos(\psi) & \cos(\phi)\sin(\psi) - \sin(\phi)\cos(\psi) & \cos(\phi)\cos(\theta) 
\end{bmatrix} \] (23d)

\[ A_{\text{rotated}} = A_{\text{measured}} - \mathbf{G}_{\text{phone}} \] (23e)

\[ A_{\text{measured}} = [\text{accelerometer } x, \text{accelerometer } y, \text{accelerometer } z] \] (23f)

\[ \mathbf{G}_{\text{phone}} = [g_x, g_y, g_z] \] (23g)

\[ g_x = \tan(\psi) \left( \sqrt{g_y^2 + g_z^2} \right) \] (23h)

\[ g_y = -\tan(\phi) \left( \sqrt{g_x^2 + g_z^2} \right) \] (23i)

\[ g_z = \tan(\theta) \left( \sqrt{g_x^2 + g_y^2} \right) \] (23j)

The matlab function fsolve is used to solve the system of 6 equations and 6 unknowns.

**Calculate Radius of Curvature Method 3 (calcRadCurvature3)**

Method three uses a different approach to find the radius of curvature from method two. This method relates the change in heading angle to a portion of a circle describing the flight path.

\[ \text{heading}_2 - \text{heading}_1 = \alpha \] (24)

Next, the distance traveled between points one and two was calculated using calcDistance. The radius of curvature is then calculated by relating the distance traveled as a portion of the circumference to the angle alpha as a portion of three hundred and sixty degrees.

\[ \text{radius} = \frac{\text{distance traveled}}{\alpha} \] (25)

3. **Plot Test Flight (MATLAB)**

Plot Test Flight was the program developed to import the data from the test flight on May 3rd, 2013. Once again, MATLAB was used due to its ease of plotting results and the ability to see the results of changing pieces of code quickly. This tool was also used to test attitude estimation method 3 so that the data used would be the most current.
**Plot Test Flight Functions Explained**

**Main Class**

The top level class for comparing the results for test flight #2 is called `plotTestFlight`. It uses the below functions to import the phone and truth source data and plot corresponding parameters against one another for every test point performed in the flight. In order to match the time vectors up, two time arrays were created for each data set (tReal for G1000 data and tfp for the phone data). With all the data imported and the time vectors created, parameters from the phone and G1000 are plotted against each other to compare differences. The following parameters are currently being compared in this class: Pitch, Bank, Heading, Latitude, Longitude, Altitude Baro, Altitude GPS, and Ground Speed.

**Flight Test Point To File Name (FlightTestPointToFileName)**

This function allows the user to input a certain test point for the flight test occurring on May 3rd 2013 and links it with the data files associated with that test run. These data files include the G100 data, phone raw sensor data, phone filtered sensor data, and phone computed flight parameters data. For this flight test there were only test points 1 – 24 and any other input will result in an error. The input is found in the `plot flight test` m-file.

**Import Cell Phone Flight Parameters File (ImportCellPhoneFlightParametersFile)**

The purpose of this function is to import the CSV file for the flight parameters for the specific test point given. The input to this function is a file name which can be acquired by the function `flight test point to file name` and the output is a data structure containing all of the computed flight parameters in that file. The returned data file is organized so that each data `flightParameter` links to an entire column from the CSV flight parameter file. The following parameters are available from this function:
Table 9. List of Measured/Calculated Flight Parameters

<table>
<thead>
<tr>
<th>data_column</th>
<th>Parameter (column in file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeGps</td>
<td>GPS time in ms (1)</td>
</tr>
<tr>
<td>Lat</td>
<td>Latitude in deg (2)</td>
</tr>
<tr>
<td>Lon</td>
<td>Longitude in deg (3)</td>
</tr>
<tr>
<td>altGps*</td>
<td>Altitude from GPS in feet (4)</td>
</tr>
<tr>
<td>altBaro*</td>
<td>Altitude from Barometer in feet (5)</td>
</tr>
<tr>
<td>Bank</td>
<td>Bank Angle in deg (6)</td>
</tr>
<tr>
<td>Pitch</td>
<td>Pitch Angle in deg (7)</td>
</tr>
<tr>
<td>Yaw</td>
<td>Heading in deg (8)</td>
</tr>
<tr>
<td>gndSpd</td>
<td>Ground Speed in knots (9)</td>
</tr>
<tr>
<td>climbRate</td>
<td>Climb Rate in ft/s (10)</td>
</tr>
<tr>
<td>rollRate</td>
<td>Roll Rate in deg/s (11)</td>
</tr>
<tr>
<td>H</td>
<td>GPS Hour (12)</td>
</tr>
<tr>
<td>M</td>
<td>GPS Minutes (13)</td>
</tr>
<tr>
<td>S</td>
<td>GPS Seconds (14)</td>
</tr>
<tr>
<td>Time**</td>
<td>Time into the test point in seconds (15)</td>
</tr>
</tbody>
</table>

*Note: Units were converted from m to ft the data file in order to plot correctly

**Note: This time vector is calculated off of the G1000 time in order to line up test points and will start at 0 for every test point.

**Import G1000 File 2 (ImportG1000File2)**

This function is called to import the data from the G1000 output file. The input to this function is a filename which is acquired by the function flight test point to file name and the output is a data structure containing numerous truth source variables for a single test point. The returned data file is organized so that each data. G1000Variable links to an entire column from the CSV G1000 file. The following parameters are available from this function:
Table 10. List of Parameters Supplied to Data Analysis Functions

<table>
<thead>
<tr>
<th>Parameter (column in file)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date of test flight as a string in yy-mm-dd format (1)</td>
</tr>
<tr>
<td>time</td>
<td>Local time as a string in hh-mm-ss format (2)</td>
</tr>
<tr>
<td>utcOffset</td>
<td>The offset from local time to UTC time as a string in hh-mm format (3)</td>
</tr>
<tr>
<td>lat</td>
<td>Latitude in deg (4)</td>
</tr>
<tr>
<td>lon</td>
<td>Longitude in deg (5)</td>
</tr>
<tr>
<td>alt_baro</td>
<td>Altitude from Barometer in feet (6)</td>
</tr>
<tr>
<td>baroP</td>
<td>Barometric Pressure in inches of Mercury (7)</td>
</tr>
<tr>
<td>alt_msl</td>
<td>Altitude GPS in ft from MSL (8)</td>
</tr>
<tr>
<td>oat</td>
<td>Outside Air Temperature in deg C (9)</td>
</tr>
<tr>
<td>ias</td>
<td>Indicated Air Speed in knots (10)</td>
</tr>
<tr>
<td>gndSpd</td>
<td>Ground Speed in knots (11)</td>
</tr>
<tr>
<td>vertSpd</td>
<td>Vertical Speed in ft/min (12)</td>
</tr>
<tr>
<td>pitch</td>
<td>Pitch Angle in deg (13)</td>
</tr>
<tr>
<td>roll</td>
<td>Bank Angle in deg (14)</td>
</tr>
<tr>
<td>latAccel</td>
<td>Lateral Acceleration in g’s (15)</td>
</tr>
<tr>
<td>normAccel</td>
<td>Normal Acceleration in g’s (16)</td>
</tr>
<tr>
<td>heading</td>
<td>Heading in deg (17)</td>
</tr>
<tr>
<td>gndTrack</td>
<td>Ground Track in deg (18)</td>
</tr>
<tr>
<td>alt_gps</td>
<td>Altitude from GPS in feet (19)</td>
</tr>
<tr>
<td>tas</td>
<td>True Air Speed in knots (20)</td>
</tr>
<tr>
<td>crs</td>
<td>Course Set in deg (21)</td>
</tr>
<tr>
<td>magVar</td>
<td>Magnetic Variance in deg (22)</td>
</tr>
<tr>
<td>vertSpdGps</td>
<td>Vertical Speed based only on GPS in ft/min (23)</td>
</tr>
</tbody>
</table>

**Filter Heading (FilterHeading)**

This function subjects the phone heading to a moving average filter with a averaging constant of 0.05. To read more about moving average filters refer to the Filter Raw function in the above Sensor Test Functions Explained section.

**Calculate Average Heading (calcAvgHeading)**

Since heading data is recorded 50 more times than GPS data, there needs to be a way to take the average heading over those 50 data points. Essentially, this is what the purpose of this function is. It takes the sum of each set of 50 data points and then divides this sum by 50 (the number of points). The resulting average heading is placed where the 50th data point occurs and this process is repeated until there are less than 50 data points left. This function will cut off the beginning and end of the heading data but, for long runs, this will not be an issue.
**Calculate Distance (calcDistance)**

The goal of this function is to calculate a 3-D distance traveled in feet from one GPS point to the next. It takes an input of ECEF \((x,y,z)\) positions and performs the distance formula from the last point to the current point. The following equation is used:

\[
Distance = \sqrt{(x(1) - x(2))^2 + (y(1) - y(2))^2 + (z(1) - z(2))^2}
\]  

(26)

**Calculate ECEF (calcECEF2)**

This function is explained in more detail in the above Sensor Test Functions Explained section. The reason why it is also used in the Plot Flight Test MATLAB file is to correct for the scan rate differences in altitude from the barometer and other GPS data, 20 ms to 1 s respectively. This error caused the ground speed to update for every barometer change even if GPS didn’t change which made ground speed much smaller than it should have been. Here the inputs are latitude, longitude, and altitude from GPS and the output is \((x,y,z)\) in feet from the center of the earth.

**Calculate Ground Speed (calcGroundSpeed 2)**

Using the corrected ECEF data, this function computes the corrected ground speed only using changes in GPS. To get a more detailed explanation of this function please refer to the above Sensor Test Functions Explained section. This function has the inputs of \((x,y,z)\) and outputs ground speed in knots.

**Calculate Radius of Curvature Method 3 (calcRadCurvature3)**

This function calculates the radius of curvature based on change in heading and distance traveled over time. The function is explained in detail in Plot SAS.

**Calculate Bank Method 2 (calcBank2)**

This function calculates bank by taking the inverse tangent of normal acceleration divided by gravitational acceleration. See calcBank2 in plotSAS functions above for detailed equations.

**Design Verification and Testing**

**Testing**

Design verification was part of an iterative process of code development for this project. Each of the functions and methods of calculating the necessary parameters were validated with data from ideal, controlled, and actual test environment. The design verification plan can be seen in Appendix G which includes the specification, test description, acceptance criteria, and testing schedule.
The first step in design verification was to use a controlled environment to evaluate the capabilities of the phone’s sensors and the algorithm’s ability to derive the necessary parameters from them. The controlled environment testing involved the phone being attached to a rate table which rotated about one axis at increasing degrees of bank. The accelerometer data was retrieved from the phone and used to validate Method 1 as an approach to get bank from the components of gravity in each direction. Because the angles were changed steadily and there was only one axis of rotation, the test did not simulate completely the acceleration felt when on an airplane but was good for a first step towards validating equations.

Ideal data sets were created in Matlab to test the algorithm and make sure that it was the data collected by the phone which was causing any errors in the results, as opposed to errors within the code. The ideal data sets included GPS coordinates, time, and velocity for a perfectly circular flight path. These parameters were used to calculate the bank based on normal acceleration and gravity. Once the equations were validated with a perfect flight path, error was introduced in the form of the random noise function in Matlab. The magnitude of noise was varied from a few inches to a few feet to test how sensitive the method was to error in GPS data.

The next step in design verification was to use data collected on a cell phone during real flights. Twenty-seven fly-up maneuvers were tested in the summer of 2012. These maneuvers consisted of a starting climb rate and bank, an entry procedure, and a target bank. When the pilot began each test point a passenger noted the start and end time, altitude, and indicated air speed. The phone’s sensor data was stored on the phone to be conditioned after. To test the accuracy of all values calculated, the flight parameters were also captured by a Garmin G1000® which was used as a truth source.

A second round of test flights was deemed necessary to get a better idea of what the sensors and algorithm could do. 24 flight cards were created to reenact several flight scenarios. In this test flight, the parameters required by the collision avoidance algorithm were calculated in real time. A matrix of test points including starting climb rate and bank, entry procedure, and target bank can be found in Appendix F. The test points were also categorized by whether they included a fly-up to reach target bank, since all twenty-seven test points from the previous test flight included a fly-up at the end. Several maneuvers were executed without changing elevation to get a simpler set of data for analysis.

Using the Data Collection App

Installation:
1) Connect the phone to the computer via a micro-USB to USB connection
2) Make sure the phone is mounted as “charge only” and not a disk drive
3) Run Sensor Test through Eclipse
4) The Sensor Test App should now be installed on the phone. An icon within the app menu that reads “Sensor Test” will now appear.
5) The installation only has to be performed the first time the app is being placed on the phone.
Running the App:
1) Choose the Sensor Test App from the App menu
2) A start up screen seen in Figure 6 will appear with empty data columns (#3a,#4a,#5a)
3) To properly orient the phone make sure that the buttons of the phone are to the left when the phone is held in the landscape position
4) Place the phone onto a sturdy mount and move to desired position
5) Press the calibrate button (#2) to calibrate this new phone orientation as the datum. This button can only be pressed one time while the app is in use.
6) Make sure the banner reads Ready and is green (#6a)
7) When ready, press the Start Algorithm button (#1) to run the algorithm and start computing and recording all parameters. The screen will now look like Figure 7.
8) The data columns will now fill in with computed values (#3b, #4b, #5b) and update five times per second. Note that a GPS link may take a few seconds to be initiated if it can create a link at all. Due to this, the data depending on GPS will read zero until a link is made.
9) The banner will now turn red and read Recording (#6b)
10) To stop the algorithm from computing values, press the stop algorithm button (#1b)
11) To perform more runs, repeat steps 6-10
12) Exit the app when finished

Retrieving the Files:
1) Connect the phone to the computer using a micro-USB to USB connection
2) Mount the phone as a disk drive
3) Locate the files on either the SD card drive or the internal storage of the phone
4) Save files to computer
5) Disconnect the phone
Figure 6. Data Collection Application Start-up Screen

Figure 7. Data Collection Application Recording Screen
Results

Inertial Lab
As stated above, the purpose of the inertial lab was to verify equations used for bank estimation using the phone’s internal sensors in a controlled environment. The actual angle recorded by the rate table was plotted against the angle calculated using the components of gravity in each axis. Figure 8 shows that the calculated and truth source bank angles are very close, with a small amount of error when the phone changes direction at the peaks. This is consistent with later conclusions that the acceleration felt while turning will make it difficult to calculate bank from components of gravity.

![Figure 8. Bank calculated from accelerometer and gyroscope data for inertial lab.](image)

Ideal Data
The results of using ideal data to use method two of bank estimation with GPS points can be seen figures 6&7. The first set of graphs shows the results of a perfect flight path with no noise in the GPS signal. The path, seen at the bottom right corner of the figure, is a semi-circle with a constant radius of 2000 feet. At a designated ground speed of 100 kts, the normal acceleration equals 14.1 ft/s², and the required bank angle for the maneuver is 23.6°. The ideal graphs show that the correct values have been calculated, supporting the use of this method of bank estimation. With a few inches of added noise, the bank calculated varied between 20 and 27 degrees, which meets the threshold requirement of 5 degrees. However, when ten feet of noise (which is within the accuracy of GPS data collected in real flight) is added to the ideal flight path, the calculated parameters are distorted and bank varies from 5 to 50 degrees, which is not within the threshold values.
Figure 9. Radius of curvature, normal acceleration, calculated bank, and flight path for ideal data.

Figure 10. Radius of curvature, normal acceleration, calculated bank, and flight path for ideal data with 10 ft random noise.
Flight Test
The results of flight testing were evaluated at the various test points and three points were chosen to be put into the table below describing the accuracy of flight parameters calculated in the phone with the G1000 data. The three test points 1, 15, and 24, are straight and level, diving turn with bank changing from 30 to -30 degrees, and a climbing turn from 0 to 30 degree bank respectively. Accuracy was evaluated using the Matlab generated plots seen in Appendix J. Attitude was difficult to evaluate by visual inspection for test points 15 and 24 because as seen in Figures 11 and 12, they did not appear to follow the trend of the actual data.

![Figure 11. Comparison of phone and truth source bank for test point 24.](image1)

![Figure 12. Comparison of phone and truth source pitch for test point 24.](image2)
One explanation investigated was that the times were not matching up between the phone sensors and G1000, distorting the plots. However, barometric altitude follows the correct trend and has a constant deviation from the truth source around 100 feet. Since the barometric altitude has the same time stamp as the other sensors, it appears that the times are relatively well matched. This evidence is shown in Figure 13.

![Figure 13. Comparison of phone and truth source barometric altitude for test point 24.](image)

Unlike data collected from sensors, parameters calculated from GPS data were steady and followed the trend of the truth source data. Latitude and longitude matched within 0.001 degrees for every test point. Altitude calculated with GPS data was less accurate for the test points with rapid altitude changes. This is believed to be a result of internal filtering within the phone. Because the phone is designed to stay on the ground, rapid changes in altitude are not expected. Taking a closer look at Figure 14, the G1000 data shows an increase from 6250 ft to above 6550 ft in about ten seconds. This climb rate of 1800 ft/min is not expected on the ground, and it appears that the phone’s GPS has filtered out both of the drastic changes in altitude.

A complete table of the results for all design verification testing can be seen in Appendix G. Some test parameters such as the sensor scan rate and fixture vibration characteristics were able to meet the acceptance criteria. However, the tests regarding attitude and altitude had considerable error and did not meet the acceptance criteria. Further design modifications should be researched in order to reduce the error of these readings in an attempt to meet the tolerances required for the collision avoidance algorithm.
Figure 14. Comparison of phone and truth source GPS altitude for test point 15.

Table 11. Visual inspection of graphs comparing G1000 data to phone data for three test points.

<table>
<thead>
<tr>
<th>Test Point</th>
<th>1: Straight and Level</th>
<th>15: dive bank 30 to -30 degree bank</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pitch</strong></td>
<td>Between +5 and -17 degrees, average -7 degrees, noise level very high</td>
<td>Does not follow trend</td>
</tr>
<tr>
<td><strong>Bank</strong></td>
<td>Between +13 and -5 degrees, average 6 degrees, noise level very high</td>
<td>Does not follow trend</td>
</tr>
<tr>
<td><strong>Heading</strong></td>
<td>50 degrees less than G1000, noise level low</td>
<td>Does not follow trend</td>
</tr>
<tr>
<td><strong>Latitude</strong></td>
<td>0.001 degrees, noise level low</td>
<td>0.001 degrees, noise level low</td>
</tr>
<tr>
<td><strong>Longitude</strong></td>
<td>0.001 degrees, noise level low</td>
<td>0.0001 degrees, noise level low</td>
</tr>
<tr>
<td><strong>Altitude Baro</strong></td>
<td>100 ft, noise level high</td>
<td>100 ft, noise level moderate</td>
</tr>
<tr>
<td><strong>Altitude GPS</strong></td>
<td>Between -40 and -60 ft, average -50 ft, noise level moderate</td>
<td>Does not follow trend</td>
</tr>
<tr>
<td><strong>Ground Speed</strong></td>
<td>Between +4 and -3 kts, average +3 kts, noise level moderate</td>
<td>Between +50 and +20 kt, noise level high</td>
</tr>
</tbody>
</table>

*Note: First ten seconds of ground speed due to calibration error*
Management Plan

Our team consisted of three members who split up tasks according to the correlating roles. All instrumentation and creating/finding truth sources for experiments were under Bridgette’s control. Alex took on the role of being phone specialist which included internal systems (sensors, etc...) as well as the phone’s programming mainframe. Drew was in control of the algorithm development and conditioning of parameters to be given to the CTP Avoidance team. All other responsibilities such as documentation, test approach generation, selecting a mounting fixture, and carrying out experiments were done by all members collaboratively. Deadlines were taken seriously and professionally by all members of the team. Table A1 in Appendix A details the major milestones and due dates for the span of the project.

To keep the project moving forward and ensure that major milestones were achieved on time, the team developed a Gannt chart which can be seen in Appendix D specifying when major tasks would be initiated and completed. The start and finish dates for each task were selected keeping in mind that many have predecessors which had to be completed prior to their start date, and tasks that would be accomplished in parallel.

The initial step in the project was the design of the fixture and algorithm. The phase was set to last 59 days, from October 23rd to January 11th. Brainstorming and researching existing products lead to the development of design matrixes which served as the guide to choosing the appropriate models. The critical part of the design phase, creating a detailed flow chart, took place in December and January and served as the outline for building the algorithm code.

Once the design phase was completed and a suitable flow chart and fixture were selected the build period started. Because the fixture was to be a purchased product, the “build” consisted of ordering the various products for testing. The algorithm code was projected to be completed by February 14, allowing plenty of time for testing of the phones scheduled to commence that week. As the code was developed, new methods had to be tested and implemented because of discoveries that arose about the accuracy of the phone’s sensors and the algorithm’s sensitivity to error. Because of this the “building” phase of the project continued into May.

Testing was scheduled to span 80 days from January to April. The first phase of testing was of the fixtures to identify which had the least effect on the sensor data from the phone. This testing was completed on time and the fixture was selected and purchased for use in the flight testing. Analysis of data from flight tests took place throughout the building of the algorithm to validate equations. The continual testing of the software extended the “testing” phase into May as well.

The report was written in stages throughout the project. This final report is a combination of the project proposal, conceptual design, and critical design reports. Each report will was accompanied by presentations in class followed by a presentation to the sponsor.
Conclusions and Recommendations

After testing the software developed and analyzing the data from these tests, significant progress was made in the field of determining attitude from smartphone sensors. However, all of the engineering requirements outlined were not able to be met. Most specifically, the minimum threshold for attitude angles was unachievable based on the methods and functions we used. This is not to say that it is impossible to do, but in order to achieve this goal, further research and testing will be required.

One of the largest factors that made accurate attitude so difficult to achieve was the significant noise in the data due to aircraft vibration. The noise was significant enough that it was difficult to detect any trend in the data. Accelerometers and gyroscopic sensors in the phone are extremely sensitive to vibration and typically had a band of noise that spanned ±3 m/s². Data tests conducted in a controlled environment on a rate/tilt table was very accurate, but when the same technology was moved into flight testing, the functions were no longer accurate. It is the strong recommendation of this team that a fixture should be researched that can be used as a mechanical damper. Based on the observations of this project, electronic filtering will only do so much until a mechanical filter is required. Although this would detract from the phone’s capability as a stand-alone device, this team feels it is a possibility that should be investigated.

Based on our testing and data analysis, a low-pass filter and a complimentary filter both contribute significantly to reducing the noise of the data and focusing in on a trend. However, even the best filter method we used (a complimentary filter integrating accelerometers and gyroscopes), did not adequately filter the data. Also, for many of the methods and functions we tested, they were extremely sensitive to noise, and a small amount of random noise caused an error in the attitude calculations. Thus, this team suggests running the data through a Kalman filter in the future. If the filtering method has some knowledge of how to model the system it is filtering (such as with a Kalman filter), the quality of the data should be expected to increase dramatically.

Regarding the capabilities of the phone sensors to determine the aircraft’s position, our testing and analysis found that the GPS sensors are very accurate at determining latitude and longitude. The GPS data retrieved by the phone coincided almost precisely with the GPS data retrieved by the G1000 (the truth source). However, altitude data from the phone was relatively inaccurate and lagged behind the truth source creating an error of 50 – 100 ft. It is possible that this is due to filtering built in by the phone manufacturer, since the phones are primarily intended to only be used for navigation on the ground. During this project our team attempted to contact the phone manufacturer to investigate this issue, but we were unsuccessful. Moving forward, the phone manufacturing companies should be contacted to better understand this issue.
Acknowledgments

This team would like to thank Dr. Charles Birdsong for his advice and guidance as the faculty advisor of this project. For the duration of the project at California Polytechnic State University, San Luis Obispo, Dr. Birdsong used his background in data filtering and collision avoidance to provide considerable assistance to the task of smartphone sensor evaluation.

We would also like to thank the NASA organization and the NASA employees involved in the ACAT project. Led by Mr. Mark Skoog, other employees that contributed to the project include Ms. Dena Gruca, Mr. Frederick Wright, Mr. Ben Pearson, Mr. Lloyd Hook, and Mr. Doyle Janzen. This team of individuals assisted this team in all of their respective fields including overall project supervision and software development. Additionally, Mr. Skoog and Mr. Janzen conducted all flight testing required to verify the software that was produced.

Finally, we would like to thank California Polytechnic State University, and specifically the Mechanical Engineering Department for facilitating the project and creating the opportunity to work with NASA to further research this technology that has the potential to save so many lives. We look forward to seeing the progress made with this technology in the future, and wish the best for NASA and the future teams working on this project.

References


Appendices

Appendix A. Project Timeline

Table A1. List of Major Milestones

<table>
<thead>
<tr>
<th>Milestone</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Proposal</td>
<td>18-Oct</td>
<td>Outlines initial project requirements and goals</td>
</tr>
<tr>
<td>Conceptual Model</td>
<td>8-Nov</td>
<td>Model Fixture and Algorithm Flowchart</td>
</tr>
<tr>
<td>Conceptual Design Review (In Class)</td>
<td>27-Nov</td>
<td>Proposed top few solutions and schedule; Get feedback from students and advisor</td>
</tr>
<tr>
<td>Conceptual Design Review (With Sponsor)</td>
<td>30-Nov</td>
<td>Present solution and schedule to sponsor and receive input before proceeding to build fixture and test sensors</td>
</tr>
<tr>
<td>Critical Design Review</td>
<td>7-Feb</td>
<td>Final solution presented</td>
</tr>
<tr>
<td>Project Update Memo</td>
<td>12-Apr</td>
<td>Sponsor Update on Progress of Team</td>
</tr>
<tr>
<td>Senior Design Expo</td>
<td>30-May</td>
<td>Event for all Senior projects to present using a display/demo</td>
</tr>
<tr>
<td>Final Report Complete</td>
<td>7-Jun</td>
<td>All details of project compiled into one formal document</td>
</tr>
</tbody>
</table>
Appendix B. Quality Function Development

Figure B1. House of Quality
Appendix C. Software Architecture Models

Figure C1. Top Level Software Architecture for Model 1 (Single Module with Subclasses)
Figure C2. Top Level Software Architecture for Model 2 (Single Module with Parallel Conditioning)
Figure C3. Top Level Software Architecture for Model 3 (Tri-Module Flow)
Figure C4. Top Level Software Architecture for Final Design (Sensor Module)
Health Monitoring
Class

Figure C5. Top level Software Architecture for Final Design (Health Monitoring Module)
Appendix E: Fixture Testing Procedure

Objective:
1) Evaluate dampening properties of various mounting fixtures and positions based on amplitude ratio.
2) Evaluate under what vibration environments the phone mounts stay affixed to their intended location.

Setup:

Figure E15. Setup for Vibration Testing of Phone Mounts

1) Mount fixture and phone to base on shake table.
2) Wire Dactron Signal Analyzer and shake table components as shown in Figure E1.
3) Attach accelerometers to base and phone with channel one to signal analyzer as input.

Procedure:
1) Apply sine sweep from 5 Hz to 2000 Hz at a logarithmic rate of 6 octaves per minute up and down.
2) Observe reaction of phone and mount to excitation including how much the phone moves within the fixture and how much the fixture oscillates.
3) Plot the transfer function vs. frequency and note the magnitude and location of any peaks.
4) Repeat 1-4 for x and z directions.
5) Repeat all above for each fixture.
Test Photos:

Figure E2. Laptop with RT Pro and Shake Table.

Figure E3. Arkon Mount Z-Axis Test Setup
Figure E4. iOttie X-Axis Test Setup

Figure E5. Velcro X-Axis Test Setup
Experimental Results:

Figure E6. Amplitude Ratio of Fixtures Mounted in X-Axis

Figure E7. Amplitude Ratio of Fixtures Mounted in Z-Axis
## Appendix F: Test Point Matrix

### Table F112. Matrix of Test Points for May 3, 2013 Flight Testing

<table>
<thead>
<tr>
<th>Card Number</th>
<th>Start Climb Rate</th>
<th>Start Bank</th>
<th>Entry Proc.</th>
<th>Target Bank</th>
<th>Flyup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>S and L</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-30</td>
<td>90 deg heading change</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>30</td>
<td>90 deg heading change</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>-30</td>
<td>90 deg heading change</td>
<td>30</td>
<td>n</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>30</td>
<td>90 deg heading change</td>
<td>-30</td>
<td>n</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>-30</td>
<td>90 deg heading change</td>
<td>0</td>
<td>y</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>30</td>
<td>90 deg heading change</td>
<td>0</td>
<td>y</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>-30</td>
<td>90 deg heading change</td>
<td>30</td>
<td>y</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>30</td>
<td>90 deg heading change</td>
<td>-30</td>
<td>y</td>
</tr>
<tr>
<td>10</td>
<td>-500 fpm</td>
<td>0</td>
<td>500 fpm dive</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>11</td>
<td>-500 fpm</td>
<td>0</td>
<td>500 fpm dive</td>
<td>0</td>
<td>y</td>
</tr>
<tr>
<td>12</td>
<td>-500 fpm</td>
<td>-30</td>
<td>500 fpm dive</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td>-500 fpm</td>
<td>30</td>
<td>500 fpm dive</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>---</td>
<td>----------</td>
<td>-----</td>
<td>--------------</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>14</td>
<td>-500 fpm</td>
<td>-30</td>
<td>500 fpm dive</td>
<td>30</td>
<td>y</td>
</tr>
<tr>
<td>15</td>
<td>-500 fpm</td>
<td>30</td>
<td>500 fpm dive</td>
<td>-30</td>
<td>y</td>
</tr>
<tr>
<td>16</td>
<td>+500 fpm</td>
<td>0</td>
<td>500 fpm climb</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>17</td>
<td>+500 fpm</td>
<td>-30</td>
<td>500 fpm climb</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>18</td>
<td>+500 fpm</td>
<td>30</td>
<td>500 fpm climb</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>19</td>
<td>+500 fpm</td>
<td>-30</td>
<td>500 fpm climb</td>
<td>30</td>
<td>y</td>
</tr>
<tr>
<td>20</td>
<td>+500 fpm</td>
<td>30</td>
<td>500 fpm climb</td>
<td>-30</td>
<td>y</td>
</tr>
<tr>
<td>21</td>
<td>-500 fpm</td>
<td>0</td>
<td>500 fpm dive</td>
<td>-30</td>
<td>y</td>
</tr>
<tr>
<td>22</td>
<td>-500 fpm</td>
<td>0</td>
<td>500 fpm dive</td>
<td>30</td>
<td>y</td>
</tr>
<tr>
<td>23</td>
<td>+500 fpm</td>
<td>0</td>
<td>500 fpm climb</td>
<td>-30</td>
<td>y</td>
</tr>
<tr>
<td>24</td>
<td>+500 fpm</td>
<td>0</td>
<td>500 fpm climb</td>
<td>30</td>
<td>y</td>
</tr>
</tbody>
</table>
### Table G1. Design Verification Plan for Fixture and Algorithm

<table>
<thead>
<tr>
<th>Item No</th>
<th>Specification or Clause Reference</th>
<th>Test Description</th>
<th>Acceptance Criteria</th>
<th>Test Stage</th>
<th>Quantity Pass: Quantity Fail</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Altitude Accuracy</td>
<td>Compare measured data with truth source</td>
<td>5 degrees max</td>
<td>AG PV 24 C 2/28/2013 4/12/2013</td>
<td>Unable to meet acceptance criteria</td>
<td>all</td>
</tr>
<tr>
<td>2</td>
<td>Position Accuracy</td>
<td>Compute measured data with truth source</td>
<td>5 degrees max</td>
<td>AG PV 24 C 2/28/2013 4/12/2013</td>
<td>All test points were able to be analyzed</td>
<td>all</td>
</tr>
<tr>
<td>3</td>
<td>Refresh/Scan Rate</td>
<td>Simulate in-flight vibration and observe effect on accelerometer data</td>
<td>500 ms max</td>
<td>DL PV 24 C 2/28/2013 4/12/2013</td>
<td>All test points met all above criteria</td>
<td>all</td>
</tr>
<tr>
<td>4</td>
<td>Reproducibility</td>
<td>Simulation using existing test flight data</td>
<td>95% min</td>
<td>DL PV 24 C 2/28/2013 4/12/2013</td>
<td>Lowest peak amplitude ratio: Velcro, Lowest average: iOttie</td>
<td>all</td>
</tr>
</tbody>
</table>

**TEST PLAN**

**TEST REPORT**

**NOTES**
Appendix H: Java Software Code

package com.example.sensor.test;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.utilTimeZone;
import android.annotation.TargetApi;
import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.pm.PackageManager;
import android.graphics.Color;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.os.Bundle;
import android.os.Environment;
import android.os.Handler;
import android.support.v4.content.LocalBroadcastManager;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
import android.widget.ToggleButton;

public class MainActivity extends Activity implements SensorEventListener, OnClickListener {

    /**
     * ----------------------------------------Class/Instance Variables------------------------------------------*
     */

public static final boolean D = true; // Debug variable. Turns on/off logging in every activity
private static final int dp = 2; // How many decimal places shown on screen
private static final String TAG = "MainActivity"; // Tag for debug logs
private boolean isHealthy; // bool to say if a parameter is healthy or not
private boolean hasWrittenHeader; // bool to say if the headers for the output files have been written yet
private boolean hasBaro; // bool to see if phone has barometer sensor
private boolean hasCalibrated; // bool to see if a calibration has occurred
private BufferedWriter rawSensorWriter = null; // writes data for raw sensor outputs
private BufferedWriter filteredSensorWriter = null; // writes data for filtered sensor outputs
private BufferedWriter flightWriter = null; // writes data for computed flight parameters
private Button calibrateButton; // button to calibrate phone
private double lat; // Aircraft Latitude (deg)
private double lon; // Aircraft Longitude (deg)
private double[] altGps = new double[2]; // aircraft altitude based only on gps (ft msl)
private double[] altBaro = new double[2]; // aircraft altitude based only on barometric sensor (ft msl)
private double[] altitude = new double[2]; // Current and last known altitude (ft msl)
private double x = new double[2]; // X pos of Aircraft in ECEF coords(ft)
private double y = new double[2]; // Y pos of Aircraft in ECEF coords(ft)
private double z = new double[2]; // Z pos of Aircraft in ECEF coords(ft)
private double[] groundSpeed = new double[2]; // Groundspeed of aircraft (kts)
private double[] climbRate = new double[2]; // change in altitude per change in time (m/s)
private double rollRate; // change in bank angle per change in time (deg/s)
private float[] accel = new float[3]; // Filtered Accelerometer vector (m/s^2)
private float[] gyro = new float[3]; // Filtered Gyroscope values
private float[] magnet = new float[3]; // Magnetometer values
private float[] orient = new float[3]; // Orientation values
private float[] accelRaw = new float[3]; // Unfiltered Accelerometer vector
private float[] gyroRaw = new float[3]; // Unfiltered Gyroscope values
float[] accelZfilter = new float[2]; // Filtered accelerometer z value
float[] accelYfilter = new float[2]; // Filtered accelerometer y value
float[] accelXfilter = new float[2]; // Filtered accelerometer x value
float[] gyroZfilter = new float[2]; // Filtered gyroscope z value
float[] gyroYfilter = new float[2]; // Filtered gyroscope y value
float[] gyroXfilter = new float[2]; // Filtered gyroscope x value
private float[] magnetRaw = new float[3]; // Raw Magnetometer values
private float[] orientRaw = new float[3]; // Raw Orientation values
private float[] baro = new float[1]; // barometric sensor output (Pa)
private float[] pitch = new float[2]; // Aircraft pitch in deg
private float[] bank = new float[2]; // Aircraft bank angle in deg
private float yaw; // Yaw of the aircraft in deg
public float offsetBank; // initial bank used for calibration
public float offsetPitch; // initial pitch used for calibration
public float offsetYaw; // initial heading used for calibration
private float[] Rot = new float[16]; // Initial rotation Matrix used for heading calculation
private float[] outR = new float[16]; // Output Rotation matrix for heading
private float[] I = new float[16]; // Inclination matrix used to determine orientation
private float[] values = new float[3]; // values of orientation after rotation has been completed
private Handler handler = new Handler();
private int sensorRate = 20; // Rate of new data being pushed (ms)
    private int refreshRate = 200; // How often the sensors refresh on screen(ms)
    private LocationManager locationManager; // manages gps
private LocationListener locListener; // updates gps data
    private long[] gpsTime=new long[2]; // gps time in ms
    private long deltaTime; // time step between gps data points
    private Sensor accelerometer; // accelerometer sensor
    private Sensor gyroscope; // gyroscope sensor
private Sensor magnetometer; // magnetometer sensor
private Sensor barometric; // barometric sensor
private Sensor orientation; // Verizon's own way of determining attitude
private SensorManager sensorManager; // manages all above sensors
private SimpleDateFormat timeDisplayFormatter; // sets the time format for naming output files
private String fileName; // name of each output file
    private StringBuilder sensorBuilder1 = new StringBuilder();
    private StringBuilder sensorBuilder2 = new StringBuilder();
    private StringBuilder sensorBuilder3 = new StringBuilder();
    private TextView sensorView1; // first column text view on phone
    private TextView sensorView2; // second column text view on phone
    private TextView sensorView3; // third column text view on phone
    private TextView recordingView; //
    private ToggleButton recordButton; // "start" button for application's recording service

    @Override
    public void onCreate(Bundle savedInstanceState) {
setContentView(R.layout.activity_main);
if(D) Log.d(TAG, "onCreate");
}
@SuppressWarnings("deprecation")
public void onStart() {
super.onStart();
if(D) Log.d(TAG, "onStart");
}
// Set views
sensorView1 = (TextView) findViewById(R.id.sensorView1);
sensorView2 = (TextView) findViewById(R.id.sensorView2);
sensorView3 = (TextView) findViewById(R.id.sensorView3);
recordingView = (TextView) findViewById(R.id.recordingView);
recordingView.setBackgroundColor(Color.GREEN);
recordingView.setTextColor(Color.WHITE);

// Set buttons
recordButton = (ToggleButton)findViewById(R.id.recordButton);
recordButton.setOnClickListener(this);
calibrateButton = (Button)findViewById(R.id.calibrateButton);
calibrateButton.setOnClickListener(this);

// Set sensors
sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
gyroscope = sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
magnetometer = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
orientation = sensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION);
locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
PackageManager pm = this.getPackageManager();
hasBaro = pm.hasSystemFeature(PackageManager.FEATURE_SENSOR_BAROMETER);
if(hasBaro == true) {
    barometric = sensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE);
sensorManager.registerListener(this, barometric, SensorManager.SENSOR_DELAY_FASTEST);
}

// Start listeners
sensorManager.registerListener(this, accelerometer, SensorManager.SENSOR_DELAY_FASTEST);
sensorManager.registerListener(this, gyroscope, SensorManager.SENSOR_DELAY_FASTEST);
sensorManager.registerListener(this, magnetometer, SensorManager.SENSOR_DELAY_FASTEST);
sensorManager.registerListener(this, orientation, SensorManager.SENSOR_DELAY_FASTEST);

locListener = new MyLocationListener();

// This is a listener that listens for a broadcast from RecordDataService that gets sent
// if the gps service provider is disabled.
LocalBroadcastManager.getInstance(this).registerReceiver(mMessageReceiver, new IntentFilter("gpsIsOff"));

locationManager.requestLocationUpdates("gps", 0, 0, locListener);

// Define time formats here so it doesn't have to do it every 'refreshRate'
timeDisplayFormatter = new SimpleDateFormat("hh:mm:ss.SSSa");
timeDisplayFormatter.setTimeZone(TimeZone.getDefault());
// run doCalcs thread from onStart in order to be able to calibrate prior to recording
handler.post(doCalcs);
}
protected void onResume() {
    super.onResume();
    if(D) Log.d(TAG, "onResume");
}
protected void onPause() {
    super.onPause();
    if(D) Log.d(TAG, "onPause");
    // Stop all screen refreshes from running
    handler.removeCallbacksAndMessages(null);
}
protected void onStop() {
    super.onStop();
    if(D) Log.d(TAG, "onStop");
}
protected void onDestroy() {
    super.onDestroy();
    if(D) Log.d(TAG, "onDestroy");

    sensorManager.unregisterListener(this);
    handler.removeCallbacks(updateView);
    handler.removeCallbacks(doCalcs);
    locationManager.removeUpdates(locListener);
    LocalBroadcastManager.getInstance(this).unregisterReceiver(mMessageReceiver);
}

/*
* xxxtttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
SimpleDateFormat fileFormatter = new SimpleDateFormat("yyyy_MM_dd
HH_mm_ss");
fileFormatter.setTimeZone(TimeZone.getDefault());
Date now = new Date();
fileName = fileFormatter.format(now);
hasWrittenHeader = false;

// Begin running the thread with the algorithm
handler.post(doCalcs);
handler.post(updateView);

} else {
    recordingView.setBackgroundColor(Color.GREEN);
    recordingView.setText("Ready");

    // Stop the algorithm thread
    handler.removeCallbacksAndMessages(null);

    try {
        // initiate the writers for writing to csv files
        rawSensorWriter.close();
        filteredSensorWriter.close();
        flightWriter.close();
    } catch (Exception e) {
        if (D) Log.e(TAG, e.toString());
    }
}

break;

case R.id.calibrateButton:
    // only allows calibration to occur the first time the button is pressed
    if (hasCalibrated != true) {
        calibrate();
    }
    break;
} //switch
} //onClick

/
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
* -------------------------------------------------------------Listeners---------------------------------------------------------------------------*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
*/
// Brings in the raw sensor values from the phone
@SuppressWarnings("deprecation")
public void onSensorChanged(SensorEvent event) {
if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
    System.arraycopy(event.values, 0, accel, 0, 3);
    System.arraycopy(event.values, 0, accelRaw, 0, 3);
} else if (event.sensor.getType() == Sensor.TYPE_GYROSCOPE) {
    System.arraycopy(event.values, 0, gyro, 0, 3);
    System.arraycopy(event.values, 0, gyroRaw, 0, 3);
} else if (event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD) {
    System.arraycopy(event.values, 0, magnet, 0, 3);
    System.arraycopy(event.values, 0, magnetRaw, 0, 3);
} else if (event.sensor.getType() == Sensor.TYPE_ORIENTATION) {
    System.arraycopy(event.values, 0, orient, 0, 3);
    System.arraycopy(event.values, 0, orientRaw, 0, 3);
} else if (event.sensor.getType() == Sensor.TYPE_PRESSURE) {
    System.arraycopy(event.values, 0, baro, 0, 1);
}

// calculates the orientation of the phone based on any physical orientation and screen layout
if (magnet != null && accel != null) {
    SensorManager.getRotationMatrix(Rot, I, accel, magnet);

    // Correct if screen is in Landscape
    SensorManager.remapCoordinateSystem(Rot, SensorManager.AXIS_X, SensorManager.AXIS_Z, outR);

    SensorManager.getOrientation(outR, values);
}
}

public void onAccuracyChanged(Sensor sensor, int accuracy) {
}

private BroadcastReceiver mMessageReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Ignore for now
    }
};

// Brings in the GPS data from the cell phone
private final class MyLocationListener implements LocationListener {
    public void onLocationChanged(Location location) {
        // Location data processing
    }
};

// This will be called whenever an Intent with an action named "gpsIsOff" is broadcasted.
// The intent comes from RecordDataService when the gps provider is disabled
private BroadcastRecevier mMessageReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Ignore for now
    }
};
// Called when the listener is notified with a location update from the GPS
lon = location.getLongitude();
l = location.getLatitude();
altGps[0] = location.getAltitude();
gpsTime[1]=gpsTime[0];
gpsTime[0] = location.getTime();
deltaTime = (gpsTime[0]-gpsTime[1])/1000;
}
public void onProviderDisabled(String provider) {
}
public void onProviderEnabled(String provider) {
}
public void onStatusChanged(String provider, int status, Bundle extras) {
}
} //MyLocationListener

/*/  * This method rounds a number to a given number of decimal places  */
private float round(float value, int decimals) {
  if (decimals == 0) {
    float answer = Math.round(value);
    return answer;
  }
}
  } else {
    float temp = 10.0f;
    for (int i = 1; i < decimals; i++) {
      temp *= 10.0f;
    }
    float answer = Math.round(value * temp)/temp;
    return answer;
  }
}

/**
 * This method rounds a number to a given number of decimal places
 */
private float round(double value, int decimals) {
  if (decimals == 0) {
    } else {
      float temp = 10.0f;
      for (int i = 1; i < decimals; i++) {
        temp *= 10.0f;
      }
      float answer = Math.round(value * temp)/temp;
      return answer;
    }
}
float answer = Math.round(value);
return answer;

} else {
float temp = 10.0f;
   for (int i = 1; i < decimals; i++) {
      temp *= 10.0f;
   }
float answer = Math.round(value * temp)/temp;
return answer;
}

// This function filters all accelerometer and gyro data using a moving average filter
private void filterRaw(){
   float kFilter = 0.05f;

   accelXfilter[1]=accelXfilter[0];
   accelXfilter[0] = (float) ((accelRaw[0]*kFilter)+(accelXfilter[1]*(1.0-kFilter)));
   accel[0]=accelXfilter[0];

   accelYfilter[1]=accelYfilter[0];
   accelYfilter[0] = (float) ((accelRaw[1]*kFilter)+(accelYfilter[1]*(1.0-kFilter)));
   accel[1]=accelYfilter[0];

   accelZfilter[1]=accelZfilter[0];
   accelZfilter[0] = (float) ((accelRaw[2]*kFilter)+(accelZfilter[1]*(1.0-kFilter)));
   accel[2]=accelZfilter[0];

   gyroXfilter[1]=gyroXfilter[0];
   gyroXfilter[0] = (float) ((gyroRaw[0]*kFilter)+(gyroXfilter[1]*(1.0-kFilter)));
   gyro[0]=gyroXfilter[0];

   gyroYfilter[1]=gyroYfilter[0];
   gyroYfilter[0] = (float) ((gyroRaw[1]*kFilter)+(gyroYfilter[1]*(1.0-kFilter)));
   gyro[1]=gyroYfilter[0];
gyroZfilter[1]=gyroZfilter[0];
gyroZfilter[0] = (float) ((gyroRaw[2]*kFilter)+(gyroZfilter[1]*(1.0-kFilter)));
gyro[2]=gyroZfilter[0];

// This will set the offsets of the phone in the initial orientation and calibrate sensors
public void calibrate(){
    offsetBank = (float) (bank[0]*Math.PI/180);
    offsetPitch = (float) (pitch[0]*Math.PI/180);
    hasCalibrated = true;
}

// Calculates the filtered pitch angle in radians
private void calcPitch() {
    float[] gyroPitch = new float[2]; // integrated gyro pitch angle
    gyroPitch[1]=gyroPitch[0]; // creates an array of the current value and prior value
    gyroPitch[0]= gyro[1]; // creates an array of the current value and prior value

    // gets pitch using only accelerometers
    float accelPitch =(float) Math.atan2(accel[2], Math.sqrt(accel[1]*accel[1]+accel[0]*accel[0]));

    // gets pitch using integrated gyroscopes
    float integratedGyro = (float) (sensorRate/1000 *gyroPitch[0] + .5 *sensorRate/1000 * (gyroPitch[1]-gyroPitch[0]));
    //Gets pitch based on a complimentary filter with accel and gyro sensors
    pitch[0] = (float) ((0.9)*(pitch[1]+integratedGyro)+(0.1)*accelPitch);
}

// Calculates the filtered bank angle in radians
private void calcBank() {
    float[] gyroBank = new float[2]; // integrated gyro bank angle
    gyroBank[1]=gyroBank[0]; // creates an array of the current value and prior value
    gyroBank[0]= gyro[0]; // creates an array of the current value and prior value

    // gets bank using only accelerometers
    float accelBank = (float) - Math.atan2(accel[1],Math.sqrt(accel[2]*accel[2]+accel[0]*accel[0]));

    // gets bank using integrated gyroscopes
    float integratedGyro = (float) (sensorRate/1000 *gyroBank[0] + .5 * sensorRate/1000 * (gyroBank[1]-gyroBank[0]));
    //Gets bank based on a complimentary filter with accel and gyro sensors

bank[0] = (float) ((0.9)*(bank[1]+integratedGyro)+(0.1)*accelBank);
}

// Climb rate solely as a function of altitude and time
private void calcClimbRate(){
    if(altGps[0] != altGps[1]){ // Climb rate solely as a function of altitude and time
        climbRate[1]=climbRate[0];
        double climbRateF = (altGps[0]-altGps[1])/deltaTime;
        climbRate[0]=(0.1)*climbRateF+(0.9)*climbRate[1];
    }
}

//Conversion from a geoposition (lat,lon,alt) to an ECEF reference frame(x,y,Z) (ft)
private void convertToEcef(double lat, double lon, double alt){
    x[1]=x[0]; // creates an array of the current value and prior value
    y[1]=y[0]; // creates an array of the current value and prior value
    z[1]=z[0]; // creates an array of the current value and prior value
    int a = 20925647 ; //earths radius in ft (WGS84)
    double f = 1/298.257224; //flattening param
    double c = 1/(Math.sqrt((Math.pow(Math.cos(lat), 2))+(1-f)*(1-f)*Math.pow(Math.sin(lat), 2))));
    double s = (1-f)*(1-f)*c;
    x[0] = (a*c+alt*3.2808)*Math.cos(lat*Math.PI/180)*Math.cos(lon*Math.PI/180);
    y[0] = (a*c+alt*3.2808)*Math.cos(lat*Math.PI/180)*Math.sin(lon*Math.PI/180);
    z[0] = (a*s+alt*3.2808)*Math.sin(lat*Math.PI/180);
}

// Calculates the derivative of the bank angle
private void calcRollRate(){
    rollRate = (180.0/Math.PI)*((bank[0]-bank[1])/(double)sensorRate/1000)); //Derivative of bank
}

// Calculates the groundspeed of aircraft using 3D dist formula per time step
private void calcGroundSpeed(){
    //Dist Formula/ delta time
    double kFilter = 0.30; //Moving average filtering constant
    double[] groundSpeedF = new double[2]; // Filtered ground speed of aircraft(fps)
    groundSpeedF[1]=groundSpeedF[0];
    groundSpeedF[0] = Math.sqrt((x[0]-x[1])*(x[0]-x[1])+(y[0]-y[1])*(y[0]-y[1])+(z[0]-z[1])*(z[0]-z[1]))/deltaTime;
    groundSpeed[1]=groundSpeed[0];
    if (groundSpeedF[0]!= 0){
        //uses a moving avg filter and converts from ft/s to kts
groundSpeed[0]=0.59248*(groundSpeedF[0]*kFilter+groundSpeed[1]*(1-kFilter));

else{
    groundSpeed[0]=groundSpeed[0];
}
}

// This function will calculate the altitude based on a pressure
// differential from sea level if phone has barometric sensor
@TargetApi(9)
private void calcAltBaro(){
    float p0 = SensorManager.PRESSURE_STANDARD_ATMOSPHERE;

    if (hasBaro == true)
        altBaro[0] = SensorManager.getAltitude(p0, baro[0]);
    
} else {
    altBaro[0] = 0;
}
}

// Writes header for raw sensor data file
private void writeCSVHeaderRawSensor(){
    // Write the sensor csv header to a new BufferedWriter
    try {
        rawSensorWriter = new BufferedWriter(new FileWriter(new
            File(Environment.getExternalStorageDirectory(),fileName + "-phoneSensorRaw.csv")));
        rawSensorWriter.write("A_x,A_y,A_z,G_x,G_y,G_z,M_x,M_y,M_z,O_x,O_y,O_z,Baro");
        rawSensorWriter.newLine();
    } catch (Exception e) {
        if(D) Log.e( "error", e.toString());
    }
}

// Writes data for raw sensor output
private void writeCSVRawSensor(){
    try {
            magnetRaw[0] + "," +
}
rawSensorWriter.newLine();
}
}

// Writes header for filtered sensor data file
private void writeCSVHeaderFilterSensor(){
    // Write the sensor csv header to a new BufferedWriter
    try {
        filteredSensorWriter = new BufferedWriter(new FileWriter(new
        File(Environment.getExternalStorageDirectory(), fileName + "-phoneSensorFiltered.csv"));
        filteredSensorWriter.write("A_x,A_y,A_z,G_x,G_y,G_z");
        filteredSensorWriter.newLine();
    } catch (Exception e) {
        if(D) Log.e("error", e.toString());
    }
}

// Writes data for filtered sensor output
private void writeCSVFilterSensor(){
    try {
        gyro[0] + "," + gyro[1] + "," + gyro[2]);
        filteredSensorWriter.newLine();
    } catch (Exception e) {
        if(D) Log.e("error", e.toString());
    }
}

// Writes header for computed flight parameters file
private void writeCSVHeaderParameters(){
    // Write the sensor csv header to a new BufferedWriter
    try {
        flightWriter = new BufferedWriter(new FileWriter(new
        File(Environment.getExternalStorageDirectory(), fileName + "-flightParameters.csv")));

        flightWriter.write("Time(ms),Lat(deg),Lon(deg),AltGps(m),AltBaro(m),Bank(deg),Pitch(deg),Yaw(deg),Gnd
        Spd(kts),ClimbRate(m/s),RollRate(deg/s),Healthy");
        flightWriter.newLine();
    } catch (Exception e) {
        if(D) Log.e("error", e.toString());
    }
}
// Writes data for flight parameter outputs
private void writeCSVParameters(){
  try {
    flightWriter.write(gpsTime[0] + "," + lat + "," + lon + "," +
    altGps[0] + "," + altBaro[0] + "," + bank[0] + "," + pitch[0] +
    "," +
    yaw + "," + groundSpeed[0] + "," + climbRate + "," + rollRate + "," +
    isHealthy);
    flightWriter.newLine();
  } catch (Exception e) {
    if(D) Log.e(TAG, e.toString());
  }
}

// Sets the ranges of healthy variables and reports if these variables pass or not
private void healthMonitoring(){
  isHealthy = true;
  if(yaw<-180f || yaw>180f){
    isHealthy = false;
  }
  if(bank[0]<-90f || bank[0]>90f){
    isHealthy = false;
  }
  if(pitch[0]<-90f || pitch[0]>90f){
    isHealthy = false;
  }
  if(lat<-90f || lat>90f){
    isHealthy = false;
  }
  if(lon<-180f || lon>180f){
    isHealthy = false;
  }
  if(altitude[0]< 0d || altitude[0]> 18000d){
    isHealthy = false;
  }
  if(rollRate> 45){     // deg/s
    isHealthy = false;
  }
  if(climbRate[0]> 50){     // ft/s
isHealthy = false;
}

/*
* This thread runs the sensor algorithm
*/
Thread doCalcs = new Thread(new Runnable() {
  public void run() {
    handler.postDelayed(this, sensorRate);

    if(hasWrittenHeader==false){
      writeCSVHeaderRawSensor();
      writeCSVHeaderFilterSensor();
      writeCSVHeaderParameters();
      hasWrittenHeader = true;
    }
  }

  //Algorithm Methods
  filterRaw();
  calcAltBaro();
  calcBank();
  calcPitch();
  convertToEcef(lat,lon,altGps[0]);
  calcGroundSpeed();
  calcClimbRate();
  calcRollRate();

  // Sets attitude equal to the calculated value minus the calibrated offset in deg
  pitch[1]=pitch[0];
  pitch[0] = (float) ((pitch[0]-offsetPitch)*180/Math.PI);
  bank[1]=bank[0]; // creates an array of the current value and prior value
  bank[0]  = (float) ((bank[0]-offsetBank)*180/Math.PI);
  yaw = (float) (values[0]*180/Math.PI);
  // runs a health monitoring check on the calculated values
healthMonitoring();

// Writes Data
writeCSVRawSensor();
writeCSVFilterSensor();
writeCSVParameters();

} // run
});
/**
 * This thread updates the text view on the phone's screen
 */
Thread updateView = new Thread(new Runnable() {
    public void run() {
        // repeat as long as the app is running
        handler.postDelayed(this, refreshRate);

        // update the screen with new info
        sensorBuilder1.setLength(0);
        sensorBuilder1.append("Pitch: " + round(pitch[0],dp) + "]n\n Roll: " + round(bank[0],dp) + "]n\n Yaw: " +
        round(yaw,dp) + "]n\n");
        // Update sensorView1 in the UI thread
        sensorView1.post(new Runnable() {
            public void run() {
                sensorView1.setText(sensorBuilder1.toString());
            }
        });

        // update the screen with new info
        sensorBuilder2.setLength(0);
        sensorBuilder2.append("Lat: " + round(lat,dp) + "]n\n Lon: " + round(lon,dp) + "]n\n Alt: " +
        round(altGps[0],dp) + "]n\n");
        // Update sensorView2 in the UI thread
        sensorView2.post(new Runnable() {
            public void run() {
                sensorView2.setText(sensorBuilder2.toString());
            }
        });

        // update the screen with new info
        sensorBuilder3.setLength(0);
        sensorBuilder3.append("GndSpd: " + round(groundSpeed[0],dp) + "]n\n RollRate: " + round(rollRate,dp) +
        "]n\n ClmRate: " + round(climbRate[0],dp) + "]n\n");
        // Update sensorView3 in the UI thread
        sensorView3.post(new Runnable() {
            public void run() {
                sensorView3.setText(sensorBuilder3.toString());
            }
        });
Appendix I: Matlab Software

Plot SAS Flight Parameters (plotSAS)

% function [figureName, figureHandle] = plotSAS(testPoints)
% This function plots climb rate, bank angle, airspeed, and engine rpm vs
% time for the test point(s) desired
%
% Input: an integer to plot a single test point
%        an array of integers to plot multiple test points
%
% Output: figureName = a string containing the title used at the top of the
%        figure box and the graph itself
%        figureHandle = an array containing the figure handle. Useful if
%        using the function saveppt2() to create a Powerpoint
%
% Functions required:
% - flightTestPoint2Filename()
% - importCellPhoneSensorFile()
%
% Paths required:
addpath(genpath('Test Data Files'));
addpath('Matlab Functions');
addpath('SAS Functions');

% This is where the test point of the flight is chosen (9-27)
testpoint=9;
% Turns the test point input into the test point file
filename=flightTestPoint2Filename(testpoint);
% Imports the data collected from the cell phone
data= importCellPhoneSensorFile(filename.chargeSensor{1});
% Imports the truth source data from the G1000
dataReal = importG1000File(filename.g1000{1});
% Imports cell phone GPS file
dataGps = importCellPhoneGpsFile(filename.chargeGps{1});
% Defines x,y,z accelerometer data as rawData
rawData = [data.a_x,data.a_y,data.a_y];

% Runs low pass filter on cell phone data
filteredValues=lowPassFilter(data);
%% Calculate parameters from filtered values

% Pitch
pitchF = calcPitch(filteredValues, false, data.time);
pitchComp = calcPitch(filteredValues, true, data.time);

% Bank Filtered
bankF = calcBank(filteredValues, false, data.time);
bankComp = calcBank(filteredValues, true, data.time);

% Pitch Raw
pitchR = calcPitch(rawData, false, data.time);

% Bank Raw
bankR = calcBank(rawData, false, data.time);

% Truth bank and pitch
realBank = dataReal.roll;
realPitch = dataReal.pitch;

% Roll and climb rates
rollRateF = calcRollRate(bankF, data.time);
climbRate = calcClimbRate(dataGps.alt, dataGps.time);

% ECEF coordinates from WGS84
ecef = calcEcef(dataGps);

% Calculate groundspeed from cell phone GPS
groundSpeedRaw = calcGroundSpeed(ecef, dataGps.time, false);
groundSpeedFiltered = calcGroundSpeed(ecef, dataGps.time, true);

%% "Method 2" calculating radius of curvature from GPS points
rad_curvatureRaw = calcRadCurvature(ecef, false);
radiusCurvatureFiltered = calcRadCurvature(ecef, true);

load_accelRaw = calcLoadAccel(groundSpeedFiltered, dataGps.time, rad_curvatureFiltered, false);
load_accelFiltered = calcLoadAccel(groundSpeedFiltered, dataGps.time, rad_curvatureFiltered, true);

norm_Vector = calcNormVector(ecef, rad_curvatureFiltered, load_accelFiltered);
tan_Vector = calcTanVector(ecef, norm_Vector, load_accelFiltered);

bank2 = calcBank2(groundSpeedFiltered, rad_curvatureFiltered);

% orientation = calcOrientation(tan_Vector, norm_Vector, data, dataGps);

%% Normalizing time for plots
tReal = 1:1:length(realBank);
rollRateReal = calcRollRate(dataReal.roll, tReal);

for i=2:length(data.time)
    t(i) = data.time(i);
    dt = data.time(i) - data.time(i-1);
end
for i=2:length(dataGps.time)
    tGps(i) = dataGps.time(i);
end
%% Plots from specified test point
%Pitch Comparison
figure (1)
plot(t,pitchR,t,pitchF,t,pitchComp,tReal,realPitch);
title('Comparison of Pitch')
xlabel('time(s)')
ylabel('pitch(deg)')
legend('Raw','Accel Only','Accel and Gyro','G1000')
%Bank Comparison
figure (2)
plot(t,bankR,t,bankF,t,bankComp,tReal,realBank,tGps,bank2);
title('Comparison of Bank')
xlabel('time(s)')
ylabel('bank(deg)')
legend('Raw','Accel Only','Accel and Gyro','G1000')
%Accelerometer raw and Filtered
figure(3)
subplot(3,2,1) , plot(t,data.a_x)
title('Accelerometer X')
xlabel('time(s)')
ylabel('acceleration (m/s)')
subplot(3,2,3) , plot(t,data.a_y)
title('Accelerometer Y')
xlabel('time(s)')
ylabel('acceleration (m/s)')
subplot(3,2,5) , plot(t,data.a_z)
title('Accelerometer Z')
xlabel('time(s)')
ylabel('acceleration (m/s)')
subplot(3,2,2) , plot(t,filteredValues(:,1))
title('Filtered Accelerometer X')
xlabel('time(s)')
ylabel('acceleration (m/s)')
subplot(3,2,4) , plot(t,filteredValues(:,2))
title('Filtered Accelerometer Y')
xlabel('time(s)')
ylabel('acceleration (m/s)')
subplot(3,2,6) , plot(t,filteredValues(:,3))
title('Filtered Accelerometer Z')
xlabel('time(s)')
ylabel('acceleration (m/s)')
%Groundspeed Comparison
figure(4)
plot(tGps,groundSpeedRaw,tGps,groundSpeedFiltered,tReal,dataReal.gndSpd);
title('Aircraft Ground Speed')
xlabel('Time (s)')
ylabel('Ground Speed (ft/s)')
legend('PhoneRaw','PhoneFiltered','G1000')
% Subplots with coordinates in ECEF
figure(5)
subplot(3,1,1),plot(tGps,ecef(:,1))
title('ECEF X Coordinate')
xlabel('Time (s)');
ylabel('Distance to Earth Center along X-axis (ft)')
subplot(3,1,2),plot(tGps,ecef(:,2));
title('ECEF Y Coordinate');
xlabel('Time (s)');
ylabel('Distance to Earth Center along Y-axis (ft)')
subplot(3,1,3),plot(tGps,ecef(:,3));
title('ECEF Z coordinate');
xlabel('Time (s)');
ylabel('Distance to Earth Center along Z-axis (ft)')
%Roll rate comparison
figure(6)
plot(t(5:end),rollRateF(5:end,1),tReal,rollRateReal(:,1),'r');
title('Comparison of Roll Rate')
xlabel('time(s)')
ylabel('roll rate(deg/s)')
legend('From Phone','From G1000')
%Altitude Comparison
figure(7)
plot(tGps,dataGps.alt,'*',tReal,dataReal.alt_gps,'+',tReal,dataReal.alt_baro);
title('Comparison of Altitude')
xlabel('time(s)')
ylabel('altitude(ft)')
legend('Phone GPS','G1000 GPS','G1000 Baro')
%Subplot of latitude comparison
figure(8)
subplot(2,1,1),plot(tGps,dataGps.lat,'*',tReal,dataReal.lat);
title('Comparison of Latitude')
xlabel('time(s)')
ylabel('latitude(deg)')
legend('Phone GPS','G1000 GPS')
subplot(2,1,2),plot(tGps,dataGps.long,'*',tReal,dataReal.long);
title('Comparison of Latitude')
xlabel('time(s)')
ylabel('latitude(deg)')
legend('Phone GPS','G1000 GPS')
%Heading from G1000
figure(9)
plot(tReal,dataReal.heading)
title('Heading')
xlabel('time (s)')
ylabel('heading (deg)')
% Radius of curvature raw and filtered
figure(10)
plot(tGps,rad_curvatureRaw,tGps,rad_curvatureFiltered)
title('Radius of Curvature')
xlabel('time (s)')
ylabel('Radius of Curvature (ft)')
legend('Raw data','Filtered data')
%Bank calculated with method 2
figure(11)
plot(tGps,bank2)
title('Bank calculated from normal acceleration')
xlabel('time (s)')
ylabel('Bank (deg)')
%% Ideal Data

SampleGps = SampleFlight;
rad_curvatureTest = calcRadCurvature(SampleGps, false);
SampleTime = SampleGps(:,4);
TestGround_Speed = SampleGps(:,5);

Testload_accel = calcLoadAccel(TestGround_Speed, SampleTime, rad_curvatureTest, false);
Testnorm_Vector = calcNormVector(SampleGps, rad_curvatureTest, Testload_accel);
TestTanVector = calcTanVector(SampleGps, Testnorm_Vector, Testload_accel);

TestBank = calcBank2(TestGround_Speed, rad_curvatureTest);

SampleGpsxz = SampleFlightxz;
rad_curvatureTestxz = calcRadCurvature(SampleGpsxz, false);
SampleTimexz = SampleGpsxz(:,4);
TestGround_Speedxz = SampleGpsxz(:,5);

Testload_accelxz = calcLoadAccel(TestGround_Speedxz, SampleTimexz, rad_curvatureTestxz, false);
Testnorm_Vectorxz = calcNormVector(SampleGpsxz, rad_curvatureTestxz, Testload_accelxz);
TestTanVectorxz = calcTanVector(SampleGpsxz, Testnorm_Vectorxz, Testload_accelxz);

%% Ideal Data Plots

figure(24)
subplot(2,2,1)
plot(SampleTimexz, rad_curvatureTestxz)
title('Test Radius of Curvature')
xlabel('time (s)')
ylabel('Radius of Curvature (ft)')

subplot(2,2,2)
plot(SampleTimexz, Testload_accelxz(:,1))
title('Normal Acceleration Magnitude')
xlabel('time (s)')
ylabel('Acceleration (ft/s^2)')

subplot(2,2,3)
plot(SampleTimexz, Testload_accelxz(:,3))
title('Tangential Acceleration Magnitude')
xlabel('time (s)')
ylabel('Acceleration (ft/s^2)')

subplot(2,2,4)
plot(Testnorm_Vectorxz(:,1), Testnorm_Vectorxz(:,3));
xlabel('x Acceleration (ft/s^2)')
ylabel('z Acceleration (ft/s^2)')
subplot(2,2,4)
plot(SampleGpsxz(:,1),SampleGpsxz(:,3))
title ('Flight Path at Constant 2000 ft Radius');
xlabel('x (ft)');
ylabel('z (ft)');

figure(25)
plot3(Testnorm_Vectorxz(:,1),Testnorm_Vectorxz(:,2),Testnorm_Vectorxz(:,3))
title('Normal Acceleration Vector')
xlabel('x Acceleration (ft/s^2)')
ylabel('y Acceleration (ft/s^2)')
zlabel('z Acceleration (ft/s^2)')
ylim([-10 10])
grid on

figure(20)
subplot(2,2,1)
plot(SampleTime,rad_curvatureTest)
title ('Test Radius of Curvature')
xlabel('time (s)')
ylabel('Radius of Curvature (ft)')
xlim([5 40])
ylim([0 3500])

subplot(2,2,2)
plot(SampleTime,Testload_accel(:,1))
title ('Normal Acceleration Magnitude')
xlabel('time (s)')
ylabel('Acceleration (ft/s^2)')
xlim([5 40])
ylim([0 50])

subplot(2,2,3)
plot(SampleTime,TestBank)
title ('Calculated Bank')
xlabel('time (s)')
ylabel('Bank Angle (deg)')
xlim([5 40])
ylim([0 90])

subplot(2,2,4)
plot(SampleGps(:,1),SampleGps(:,2))
title ('Flight Path at Constant 2000 ft Radius');
xlabel('x (ft)');
ylabel('y (ft)');

figure(21)
plot(SampleTime,TestBank)
title ('Calculated Bank, Maximum 10 ft Noise')
xlabel('time(s)');
ylabel('Bank (deg)');

figure(22)
plot3(Testnorm_Vector(:,1),Testnorm_Vector(:,2),Testnorm_Vector(:,3))
title('Normal Acceleration Vector')
xlabel('x Acceleration (ft/s^2)')
ylabel('y Acceleration (ft/s^2)')
zlabel('z Acceleration (ft/s^2)')
zlim([-10 10])
grid on

figure(15) % Load Acceleration in X (normal) Plots
plot(tGps,load_accelRaw(:,1),tGps,load_accelFiltered(:,1))
title('Normal Load Acceleration')
xlabel('time (s)')
ylabel('Acceleration (ft/s^2)')
legend('X-accel Raw','X-accel Filtered')

figure(16) % Load Acceleration in Z (tangential) Plots
plot(tGps,load_accelRaw(:,3),tGps,load_accelFiltered(:,3))
title('Tangential Load Acceleration')
xlabel('time (s)')
ylabel('Acceleration (ft/s^2)')
legend('Z-accel Raw','Z-accel Filtered')

figure(17) % Norm Vector Magnitudes
plot(tGps,norm_Vector(:,1),tGps,norm_Vector(:,2),tGps,norm_Vector(:,3))
title('Norm Vector Magnitudes')
xlabel('Time (s)')
ylabel('Magnitude of Norm Accel Vector (ft/s^2)')
legend('Norm Vector in X','Norm Vector in Y','Norm Vector in Z')

figure(18) % Tan Vector Magnitudes
plot(tGps,tan_Vector(:,1),tGps,tan_Vector(:,2),tGps,tan_Vector(:,3))
title('Tan Vector Magnitudes')
xlabel('Time (s)')
ylabel('Magnitude of Tan Accel Vector (ft/s^2)')
legend('Tan Vector in X','Tan Vector in Y','Tan Vector in Z')

figure(19) % Bank and Pitch Angles
plot(tGps,orientation(:,1),tGps,orientation(:,2),tGps,orientation(:,3))
title('Orientation Angles')
xlabel ('Time (s)')
ylabel ('Orientation Angles')
legend('Bank','Pitch','Yaw')

Flight Test Point to File Name (flightTestPoint2Filename)
function [filename] = flightTestPoint2Filename(testPoint)
This function takes the test point (as an integer) and outputs a structure containing fields of cells (arrays, if applicable) containing the string of the filename for the given test point. This string can then be used as an input for the importG1000Data() function.

Input: testPoint = a string containing the test point

Output: filename = a string containing the filename of the csv file you wish to plot

The cases that reference "GpsOld" refer to the phone gps files before alteration. The cases with "Gps" that reference a file ending in "Alter" contain values that are time correlated to start with the G1000 file.

Limitations:
If adding a new field (i.e. filename.extra{1}) you must add it to every testPoint case, regardless of whether a filename exists for each test point. Adding filename.[extra field] = ''; to test points without a file will suffice. This error becomes an issue when comparing multiple test points using the compare functions.

switch testPoint
  case 1
    filename.g1000{1} = 'g1000_tp01.csv';
    filename.chargeGpsOld{1} = '';
    filename.chargeGps{1} = 'charge_tp01_g.csv';
    filename.chargeSensor{1} = 'charge_tp01_s.csv';
    filename.x2GpsOld{1} = '';
    filename.x2Gps{1} = 'x2_tp01_g.csv';
    filename.x2Sensor{1} = 'x2_tp01_s.csv';
  case 2
    filename.g1000{1} = 'g1000_tp02.csv';
    filename.chargeGpsOld{1} = '';
    filename.chargeGps{1} = 'charge_tp02_g.csv';
    filename.chargeSensor{1} = 'charge_tp02_s.csv';
    filename.x2GpsOld{1} = '';
    filename.x2Gps{1} = 'x2_tp02_g.csv';
    filename.x2Sensor{1} = 'x2_tp02_s.csv';
  case {3 3.1}
    % Best run
    filename.g1000{1} = 'g1000_tp03_2.csv';
    filename.chargeGpsOld{1} = '';
    filename.chargeGps{1} = 'charge_tp03_2g.csv';
    filename.chargeSensor{1} = 'charge_tp03_2s.csv';
    filename.x2GpsOld{1} = '';
    filename.x2Gps{1} = 'x2_tp03_2g.csv';
    filename.x2Sensor{1} = 'x2_tp03_2s.csv';
  case 3.2

% Other run
filename.g1000{1} = 'g1000_tp03_1.csv';
filename.chargeGpsOld{1} = '';  
filename.chargeGps{1} = 'charge_tp03_1.csv';
filename.chargeSensor{1} = 'charge_tp03_1s.csv';
filename.x2GpsOld{1} = '';  
filename.x2Gps{1} = 'x2_tp03_1g.csv';
filename.x2Sensor{1} = 'x2_tp03_1s.csv';

\textit{case 4}
filename.g1000{1} = 'g1000_tp04.csv';
filename.chargeGpsOld{1} = 'charge_tp04_g.csv';
filename.chargeGps{1} = 'charge_tp04_gAlter.csv';
filename.chargeSensor{1} = 'charge_tp04_s.csv';
filename.x2GpsOld{1} = 'x2_tp04_g.csv';
filename.x2Gps{1} = 'x2_tp04_gAlter.csv';
filename.x2Sensor{1} = 'x2_tp04_s.csv';

\textit{case 5}
filename.g1000{1} = 'g1000_tp05.csv';
filename.chargeGpsOld{1} = 'charge_tp05_g.csv';
filename.chargeGps{1} = 'charge_tp05_gAlter.csv';
filename.chargeSensor{1} = 'charge_tp05_s.csv';
filename.x2GpsOld{1} = 'x2_tp05_g.csv';
filename.x2Gps{1} = 'x2_tp05_gAlter.csv';
filename.x2Sensor{1} = 'x2_tp05_s.csv';

\textit{case 6}
filename.g1000{1} = 'g1000_tp06.csv';
filename.chargeGpsOld{1} = 'charge_tp06_g.csv';
filename.chargeGps{1} = 'charge_tp06_gAlter.csv';
filename.chargeSensor{1} = 'charge_tp06_s.csv';
filename.x2GpsOld{1} = 'x2_tp06_g.csv';
filename.x2Gps{1} = 'x2_tp06_gAlter.csv';
filename.x2Sensor{1} = 'x2_tp06_s.csv';

\textit{case 7}
filename.g1000{1} = 'g1000_tp07.csv';
filename.chargeGpsOld{1} = '';  
filename.chargeGps{1} = 'charge_tp07_g.csv';
filename.chargeSensor{1} = 'charge_tp07_s.csv';
filename.x2GpsOld{1} = '';  
filename.x2Gps{1} = 'x2_tp07_g.csv';
filename.x2Sensor{1} = 'x2_tp07_s.csv';

\textit{case 8}
filename.g1000{1} = 'g1000_tp08.csv';
filename.chargeGpsOld{1} = '';  
filename.chargeGps{1} = 'charge_tp08_g.csv';
filename.chargeSensor{1} = 'charge_tp08_s.csv';
filename.x2GpsOld{1} = '';  
filename.x2Gps{1} = 'x2_tp08_g.csv';
filename.x2Sensor{1} = 'x2_tp08_s.csv';
case 9
filename.g1000{1} = 'g1000_tp09.csv';
filename.chargeGpsOld{1} = 'charge_tp09_g.csv';
filename.chargeGps{1} = 'charge_tp09_gAlter.csv';
filename.chargeSensor{1} = 'charge_tp09_s.csv';
filename.x2GpsOld{1} = 'x2_tp09_g.csv';
filename.x2Gps{1} = 'x2_tp09_gAlter.csv';
filename.x2Sensor{1} = 'x2_tp09_s.csv';

case 10
filename.g1000{1} = 'g1000_tp10.csv';
filename.chargeGpsOld{1} = 'charge_tp10_g.csv';
filename.chargeGps{1} = 'charge_tp10_gAlter.csv';
filename.chargeSensor{1} = 'charge_tp10_s.csv';
filename.x2GpsOld{1} = 'x2_tp10_g.csv';
filename.x2Gps{1} = 'x2_tp10_gAlter.csv';
filename.x2Sensor{1} = 'x2_tp10_s.csv';

case 11
filename.g1000{1} = 'g1000_tp11.csv';
filename.chargeGpsOld{1} = 'charge_tp11_g.csv';
filename.chargeGps{1} = 'charge_tp11_gAlter.csv';
filename.chargeSensor{1} = 'charge_tp11_s.csv';
filename.x2GpsOld{1} = 'x2_tp11_g.csv';
filename.x2Gps{1} = 'x2_tp11_gAlter.csv';
filename.x2Sensor{1} = 'x2_tp11_s.csv';

case 12
filename.g1000{1} = 'g1000_tp12.csv';
filename.chargeGpsOld{1} = 'charge_tp12_g.csv';
filename.chargeGps{1} = 'charge_tp12_gAlter.csv';
filename.chargeSensor{1} = 'charge_tp12_s.csv';
filename.x2GpsOld{1} = 'x2_tp12_g.csv';
filename.x2Gps{1} = 'x2_tp12_gAlter.csv';
filename.x2Sensor{1} = 'x2_tp12_s.csv';

case 13
filename.g1000{1} = 'g1000_tp13.csv';
filename.chargeGpsOld{1} = 'charge_tp13_g.csv';
filename.chargeGps{1} = 'charge_tp13_gAlter.csv';
filename.chargeSensor{1} = 'charge_tp13_s.csv';
filename.x2GpsOld{1} = 'x2_tp13_g.csv';
filename.x2Gps{1} = 'x2_tp13_gAlter.csv';
filename.x2Sensor{1} = 'x2_tp13_s.csv';

case 14
filename.g1000{1} = 'g1000_tp14.csv';
filename.chargeGpsOld{1} = 'charge_tp14_g.csv';
filename.chargeGps{1} = 'charge_tp14_gAlter.csv';
filename.chargeSensor{1} = 'charge_tp14_s.csv';
filename.x2GpsOld{1} = 'x2_tp14_g.csv';
filename.x2Gps{1} = 'x2_tp14_gAlter.csv';
filename.x2Sensor{1} = 'x2_tp14_s.csv';
case 15
    filename.g1000{1} = 'g1000_tp15.csv';
    filename.chargeGpsOld{1} = 'charge_tp15_g.csv';
    filename.chargeGps{1} = 'charge_tp15_gAlter.csv';
    filename.chargeSensor{1} = 'charge_tp15_s.csv';
    filename.x2GpsOld{1} = 'x2_tp15_g.csv';
    filename.x2Gps{1} = 'x2_tp15_gAlter.csv';
    filename.x2Sensor{1} = 'x2_tp15_s.csv';

case 16
    filename.g1000{1} = 'g1000_tp16.csv';
    filename.chargeGpsOld{1} = 'charge_tp16_g.csv';
    filename.chargeGps{1} = 'charge_tp16_gAlter.csv';
    filename.chargeSensor{1} = 'charge_tp16_s.csv';
    filename.x2GpsOld{1} = 'x2_tp16_g.csv';
    filename.x2Gps{1} = 'x2_tp16_gAlter.csv';
    filename.x2Sensor{1} = 'x2_tp16_s.csv';

case 17
    filename.g1000{1} = 'g1000_tp17.csv';
    filename.chargeGpsOld{1} = 'charge_tp17_g.csv';
    filename.chargeGps{1} = 'charge_tp17_gAlter.csv';
    filename.chargeSensor{1} = 'charge_tp17_s.csv';
    filename.x2GpsOld{1} = 'x2_tp17_g.csv';
    filename.x2Gps{1} = 'x2_tp17_gAlter.csv';
    filename.x2Sensor{1} = 'x2_tp17_s.csv';

case 18
    filename.g1000{1} = 'g1000_tp18.csv';
    filename.chargeGpsOld{1} = 'charge_tp18_g.csv';
    filename.chargeGps{1} = 'charge_tp18_gAlter.csv';
    filename.chargeSensor{1} = 'charge_tp18_s.csv';
    filename.x2GpsOld{1} = '';  
    filename.x2Gps{1} = 'x2_tp18_g.csv';
    filename.x2Sensor{1} = 'x2_tp18_s.csv';

case 19
    filename.g1000{1} = 'g1000_tp19.csv';
    filename.chargeGpsOld{1} = 'charge_tp19_g.csv';
    filename.chargeGps{1} = 'charge_tp19_gAlter.csv';
    filename.chargeSensor{1} = 'charge_tp19_s.csv';
    filename.x2GpsOld{1} = 'x2_tp19_g.csv';
    filename.x2Gps{1} = 'x2_tp19_gAlter.csv';
    filename.x2Sensor{1} = 'x2_tp19_s.csv';

case 20
    filename.g1000{1} = 'g1000_tp20.csv';
    filename.chargeGpsOld{1} = 'charge_tp20_g.csv';
    filename.chargeGps{1} = 'charge_tp20_gAlter.csv';
    filename.chargeSensor{1} = 'charge_tp20_s.csv';
    filename.x2GpsOld{1} = '';  
    filename.x2Gps{1} = 'x2_tp20_g.csv';
    filename.x2Sensor{1} = 'x2_tp20_s.csv';
case 21
    filename.g1000{1} = 'g1000_tp21.csv';
    filename.chargeGpsOld{1} = 'charge_tp21_g.csv';
    filename.chargeGps{1} = 'charge_tp21_gAlter.csv';
    filename.chargeSensor{1} = 'charge_tp21_s.csv';
    filename.x2GpsOld{1} = '';
    filename.x2Gps{1} = 'x2_tp21_g.csv';
    filename.x2Sensor{1} = 'x2_tp21_s.csv';

case {22 22.1}
% Best run
    filename.g1000{1} = 'g1000_tp22_2.csv';
    filename.chargeGpsOld{1} = 'charge_tp22_2g.csv';
    filename.chargeGps{1} = 'charge_tp22_2gAlter.csv';
    filename.chargeSensor{1} = 'charge_tp22_2s.csv';
    filename.x2GpsOld{1} = '';
    filename.x2Gps{1} = 'x2_tp22_2g.csv';
    filename.x2Sensor{1} = 'x2_tp22_2s.csv';

case 22.2
% Other run
    filename.g1000{1} = 'g1000_tp22_1.csv';
    filename.chargeGpsOld{1} = 'charge_tp22_1g.csv';
    filename.chargeGps{1} = 'charge_tp22_1gAlter.csv';
    filename.chargeSensor{1} = 'charge_tp22_1s.csv';
    filename.x2GpsOld{1} = '';
    filename.x2Gps{1} = 'x2_tp22_1g.csv';
    filename.x2Sensor{1} = 'x2_tp22_1s.csv';

case {23 23.1}
% Best run
    filename.g1000{1} = 'g1000_tp23_2.csv';
    filename.chargeGpsOld{1} = 'charge_tp23_2g.csv';
    filename.chargeGps{1} = 'charge_tp23_2gAlter.csv';
    filename.chargeSensor{1} = 'charge_tp23_2s.csv';
    filename.x2GpsOld{1} = '';
    filename.x2Gps{1} = 'x2_tp23_2g.csv';
    filename.x2Sensor{1} = 'x2_tp23_2s.csv';

case 23.2
% Other run
    filename.g1000{1} = 'g1000_tp23_1.csv';
    filename.chargeGpsOld{1} = 'chargeTp23_1g.csv';
    filename.chargeGps{1} = 'chargeTp23_1gAlter.csv';
    filename.chargeSensor{1} = 'chargeTp23_1s.csv';
    filename.x2GpsOld{1} = '';
    filename.x2Gps{1} = 'x2_tp23_1g.csv';
    filename.x2Sensor{1} = 'x2_tp23_1s.csv';

case 24
    filename.g1000{1} = 'g1000_tp24.csv';
    filename.chargeGpsOld{1} = 'chargeTp24_g.csv';
    filename.chargeGps{1} = 'chargeTp24_gAlter.csv';
    filename.chargeSensor{1} = 'chargeTp24_s.csv';
filename.x2GpsOld{1} = '';  
filename.x2Gps{1} = 'x2_tp24_g.csv';  
filename.x2Sensor{1} = 'x2_tp24_s.csv';

\begin{verbatim}
\textbf{case 25}
filename.g1000{1} = 'g1000_tp25.csv';  
filename.chargeGpsOld{1} = 'charge_tp25_g.csv';  
filename.chargeGps{1} = 'charge_tp25_gAlter.csv';  
filename.chargeSensor{1} = 'charge_tp25_s.csv';  
filename.x2GpsOld{1} = '';  
filename.x2Gps{1} = 'x2_tp25_g.csv';  
filename.x2Sensor{1} = 'x2_tp25_s.csv';
\end{verbatim}

\begin{verbatim}
\textbf{case 26}
filename.g1000{1} = 'g1000_tp26.csv';  
filename.chargeGpsOld{1} = 'charge_tp26_g.csv';  
filename.chargeGps{1} = 'charge_tp26_gAlter.csv';  
filename.chargeSensor{1} = 'charge_tp26_s.csv';  
filename.x2GpsOld{1} = '';  
filename.x2Gps{1} = 'x2_tp26_g.csv';  
filename.x2Sensor{1} = 'x2_tp26_s.csv';
\end{verbatim}

\begin{verbatim}
\textbf{case 27}
filename.g1000{1} = 'g1000_tp27.csv';  
filename.chargeGpsOld{1} = 'charge_tp27_g.csv';  
filename.chargeGps{1} = 'charge_tp27_gAlter.csv';  
filename.chargeSensor{1} = 'charge_tp27_s.csv';  
filename.x2GpsOld{1} = '';  
filename.x2Gps{1} = 'x2_tp27_g.csv';  
filename.x2Sensor{1} = 'x2_tp27_s.csv';
\end{verbatim}

end

\textbf{Import G1000 File (importG1000File)}
\begin{verbatim}
function [data] = importG1000File(filename)
% This function imports the data from the g1000 csv file and outputs a
% structure containing the data of each column.
%
%Troubleshooting:
% 1) Be sure that you have added the path containing the file you wish to
% import in your script file.
% 2) The filename must end in '.csv'.
% 3) If using the string output from the testpoint2FileName() function, be
% sure that the input to this function is "fileName.device{i}" and not
% just "fileName.device".
%
% Import the file
fid = fopen(filename);
\textbf{x} = textscan(fid,'\%s \%s \%s \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f \%f ','delimiter','','headerLines',2);
% 12345678910111213141516171819202122232425262728293031
\end{verbatim}
fclose(fid);

%% Create a structure containing all data
data.date = x(1);
data.time = x(2);
data.utcOffset = x(3);
data.lat = x(4);
data.long = x(5);
data.alt_baro = x(6)/3.2808;%ft
data.alt_msl = x(7);
data.alt_gps = x(8)/3.2808;%ft
data.ias = x(9);
data.tas = x(10);
data.gndSpd = x(11)*1.6878;%fps
data.vertSpd = x(12);
data.vertSpdGps = x(13);
data.pitch = x(14);
data.roll = x(15);
data.rpm = x(16);
data.latAccel = x(17);
data.normAccel = x(18);
data.heading = x(19);
data.gndTrk = x(20);
data.fuelLeft = x(21);
data.fuelRight = x(22);
data.airTemp = x(23);
data.wndSpd = x(24);
data.wndDir = x(25);
data.magVar = x(26);
data.baroPres = x(30);
data.flyupBool = x(31);
end

Import Cell Phone Sensor File (importCellPhoneSensorFile)

function [data] = importCellPhoneSensorFile(filename)
% This function imports the data from the cell phone sensor csv file and
% outputs a structure containing the data of each column.
% Troubleshooting:
% 1) Be sure that you have added the path containing the file you wish to
% import in your script file.
% 2) The filename must end in '.csv'.
% 3) If using the string output from the testpoint2Filename() function, be
% sure that the input to this function is "filename.device(i)" and not
% just "filename.device".

%% Import the file
x = csvread(filename,1,0);

%% Create a structure containing each column of data
data.h = x(:,1);
data.m = x(:,2);
data.s = x(:,3);
data.a_x = x(:,4);
data.a_y = x(:,5);
data.a_z = x(:,6);
data.g_x = x(:,7);
data.g_y = x(:,8);
data.g_z = x(:,9);
data.m_x = x(:,10);
data.m_y = x(:,11);
data.m_z = x(:,12);
data.o_x = x(:,13);
data.o_y = x(:,14);
data.o_z = x(:,15);
data.timeSync = (3600*x(:,1) + 60*x(:,2) + x(:,3));
data.time = (3600*x(:,1) + 60*x(:,2) + x(:,3))-(3600*x(2,1) + 60*x(2,2) + x(2,3));
end

Import Cell Phone GPS File (importCellPhoneGpsFile)

function [data] = importCellPhoneGpsFile(filename)
% This function imports the data from the cell phone gps csv file and
% outputs a structure containing the data of each column.
% Troubleshooting:
% 1) Be sure that you have added the path containing the file you wish to
%    import in your script file.
% 2) The filename must end in '.csv'.
% 3) If using the string output from the testpoint2Filename() function, be
%    sure that the input to this function is "filename.device(i)" and not
%    just "filename.device".

% Import the file
x = csvread(filename,1,0);

% Create a structure containing each column of data
data.h = x(:,1);
data.m = x(:,2);
data.s = x(:,3);
data.lat = x(:,4);
data.long = x(:,5);
data.alt = x(:,6);
data.vel = x(:,7);
data.bearing = x(:,8);
data.timeSync = (3600*x(:,1) + 60*x(:,2) + x(:,3));
data.time = (3600*x(:,1) + 60*x(:,2) + x(:,3))- (3600*x(2,1) + 60*x(2,2) + x(2,3));
end

Calculate Radius of Curvature (calcRadCurvature)

function [rad_curvature] = calcRadCurvature(ecef,filterOn)
% This function calculates the radius of curvature of circle
%circumscribing a triangle using Heron's formula for the area of a triangle.

kFilter = 0.5; % Filtering constant

for k=4:length(ecef)
x1 = ecef(k-2,1);
x2 = ecef(k-1,1);
x3 = ecef(k,1);
y1 = ecef(k-2,2);
y2 = ecef(k-1,2);
y3 = ecef(k,2);
z1 = ecef(k-2,3);
z2 = ecef(k-1,3);
z3 = ecef(k,3);

%Calculate the lengths of triangle formed by 3 GPS points
a=sqrt((x2-x1)^2+(y2-y1)^2+(z2-z1)^2);
b=sqrt((x3-x2)^2+(y3-y2)^2+(z3-z2)^2);
c=sqrt((x3-x1)^2+(y3-y1)^2+(z3-z1)^2);
s=(a+b+c)/2; %Semi-perimeter
A=sqrt(s*(s-a)*(s-b)*(s-c)); %Area
rad_curvature(k,1)=(a*b*c)/(4*A); %Radius of curvature

if (filterOn == true) % Low Pass Filter to smooth data
    if(k<5)
        rad_curvature(k,1) = rad_curvature(k,1);
    else
        rad_curvature(k,1) = rad_curvature(k,1)* kFilter + rad_curvature(k-1,1)* (1-kFilter);
    end
end
end

Calculate Load Acceleration (calcLoadAccel)

function [LoadAccel] =
    calcLoadAccel(Ground_Speed,time,rad_curvature,filterOn)

% This function outputs the load acceleration of the aircraft due to normal and tangential components
kFilter = 0.3; % Filtering Constant

for k = 3:length(Ground_Speed) % Must start at 3 since data calculations include previous data points
% Calculates normal acceleration along x-axis
accel_Norm(k,1) = ((Ground_Speed(k,1))^2)/(rad_curvature(k,1)); % Normal acceleration along X-axis (Orthogonal to plane's path)

% Calculates tangential acceleration along z-axis
accel_Tan(k,1) = (Ground_Speed(k,1)-Ground_Speed(k-1,1))/(time(k,1)-time(k-1,1));

% Total load acceleration along each axis
LoadAccel(k,1) = accel_Norm(k,1); % Accel X
LoadAccel(k,2) = 0; % Accel Y
LoadAccel(k,3) = accel_Tan(k,1); % Accel Z

if (filterOn == true)
    if (k>4)
        LoadAccel(k,1) = LoadAccel(k,1)* kFilter + LoadAccel(k-1,1)* (1-kFilter);
        LoadAccel(k,2) = LoadAccel(k,2)* kFilter + LoadAccel(k-1,2)* (1-kFilter);
        LoadAccel(k,3) = LoadAccel(k,3)* kFilter + LoadAccel(k-1,3)* (1-kFilter);
    end

    % Populates first few values in Load Acceleration vector with first available point
    LoadAccel(1,1) = LoadAccel(3,1);
    LoadAccel(1,2) = LoadAccel(3,2);
    LoadAccel(1,3) = LoadAccel(3,3);

    LoadAccel(2,1) = LoadAccel(3,1);
    LoadAccel(2,2) = LoadAccel(3,2);
    LoadAccel(2,3) = LoadAccel(3,3);
end

end

% Note: Due to plane's path, there is only a normal acceleration along the x-axis, and only a tangential acceleration along the z-axis

Calculate Normal Vector (calcNormVector)

function [norm_Vector] = calcNormVector(ecef,rad_curvature,LoadAccel)

% This function outputs the unit vector normal to the path of the plane

for k=4:length(ecef)
    x1 = ecef(k-2,1);
    x2 = ecef(k-1,1);
    x3 = ecef(k,1);

    % Calculate the normal vector components
    norm_x = x2*rad_curvature(k,1) - x1*rad_curvature(k-1,1);
    norm_y = x1*rad_curvature(k,1) - x2*rad_curvature(k-1,1);
    norm_z = x2 - x1;

    % Normalize the vector
    norm = sqrt(norm_x^2 + norm_y^2 + norm_z^2);
    norm_Vector(k,1) = norm_x/norm;
    norm_Vector(k,2) = norm_y/norm;
    norm_Vector(k,3) = norm_z/norm;
end
x3 = ecef(k,1);
y1 = ecef(k-2,2);
y2 = ecef(k-1,2);
y3 = ecef(k,2);
z1 = ecef(k-2,3);
z2 = ecef(k-1,3);
z3 = ecef(k,3);

r = rad_curvature(k); % Radius of curvature

% Establish 3 points in space from GPS
p1 = [x1,y1,z1];
p2 = [x2,y2,z2];
p3 = [x3,y3,z3];

input = [x1 y1 z1 x2 y2 z2 x3 y3 z3 r];
if k==4
    x0 = [-8000000,-15000000,11800000]; % Initial conditions to solve for
    solution to nonlinear equations
else
    x0= [output(1), output(2), output(3)];
end
output= fsolve(@(output)findCenter(output,input),x0); % Calls
function and uses fsolve to determine center of circle from a system on
nonlinear equations

% Here we should have x,y,z,r
x = output(1,1);
y = output(1,2);
z = output(1,3);
r = rad_curvature(k);

unit_Vector(k,1) = 1/r*(x-p3(1)); % Unit vector in the direction of
circle center (X-component)
unit_Vector(k,2) = 1/r*(y-p3(2)); % Unit vector in the direction of
circle center (Y-component)
unit_Vector(k,3) = 1/r*(z-p3(3)); % Unit vector in the direction of
circle center (Z-component)

norm_Vector(k,1) = LoadAccel(k,1)*unit_Vector(k,1); % X-component
vector
norm_Vector(k,2) = LoadAccel(k,1)*unit_Vector(k,2); % Y-component
vector
norm_Vector(k,3) = LoadAccel(k,1)*unit_Vector(k,3); % Z-component
vector

if (k == 4)
    for j=1:3 % Since there is no data for the first few points, this
    populates the vector with data corresponding to the first available point
    norm_Vector(j,1) = norm_Vector(4,1);
    norm_Vector(j,2) = norm_Vector(4,2);
    norm_Vector(j,3) = norm_Vector(4,3);
end
% Solves for the center of a circle using the equations of a circle for 3 distinct points
% Equations for circle given that center of circle is (x,y,z), and % radius of curvature r

function F = findCenter(output,input)
    F(1)= ((-output(1))+(input(1)))^2+((-output(2))+(input(2)))^2+((-
    output(3))+(input(3)))^2-(input(10))^2;
    F(2)= ((-output(1))+(input(4)))^2+((-output(2))+(input(5)))^2+((-
    output(3))+(input(6)))^2-(input(10))^2;
    F(3)= ((-output(1))+(input(7)))^2+((-output(2))+(input(8)))^2+((-
    output(3))+(input(9)))^2-(input(10))^2;
end

Calculate Tangential Acceleration Vector (calcTanVector)

function [tan_Vector] = calcTanVector(ecef,norm_Vector, LoadAccel)

for k=4:length(ecef)
    x1 = ecef(k-2,1);
    x2 = ecef(k-1,1);
    x3 = ecef(k,1);
    y1 = ecef(k-2,2);
    y2 = ecef(k-1,2);
    y3 = ecef(k,2);
    z1 = ecef(k-2,3);
    z2 = ecef(k-1,3);
    z3 = ecef(k,3);

    % Establish 3 points in space from GPS
    p1 = [x1,y1,z1];
    p2 = [x2,y2,z2];
    p3 = [x3,y3,z3];

    v1 = p3-p1;
    v2 = p3-p2;

    cross_prod = cross(v2,v1);
    cross_prod2 = cross(cross_prod,norm_Vector(k,:));

    mag_cross_prod2 =
    sqrt((cross_prod2(1))^2+(cross_prod2(2))^2+(cross_prod2(3))^2);

    unit_Vector = 1/(mag_cross_prod2) * cross_prod2;

    tan_Vector(k,:) = LoadAccel(k,3) * unit_Vector;
    tan_Vector(1,:) = tan_Vector(4,:);
    tan_Vector(2,:) = tan_Vector(4,:);
    tan_Vector(3,:) = tan_Vector(4,:);
end
**Calculate Bank Method 2 (calcBank2)**

```matlab
function [SampleBank]=calcBank2(GroundSpeed,rad_curv)
g=32.174; %ft/s^2
for i=1:length(GroundSpeed)
    theta(i)=atand((GroundSpeed(i))^2/(rad_curv(i)*g));
end
SampleBank(1,:)=theta;
end
```

**Sample Flight Data (SampleFlight and SampleFlightxz)**

```matlab
function [SampleGps]= SampleFlight

h=-8.71951e6;   %Center of circle latitude (ECEF)
k=-14706253.281; %Center of circle longitude (ECEF)
r=2000; %Radius of circle (ft)
V=168; %ft/s (100 knots)
x1=-8.71951e6 - r; %ECEF latitude start
x2=-8.71951e6 + r;  %ECEF latitude end

x= x1:100:x2;
y=zeros(length(x),1);
z=zeros(length(x),1);   %assume a constant elevation turn
for i=1:length(x)
    y(i)=k+sqrt(r^2-(x(i)-h)^2)+rand;
    xr(i)=x(i)+rand;
end
SampleGps(:,1)=xr;
SampleGps(:,2)=y;
SampleGps(:,3)=z;
t=1;

for ii=1:length(x)
    Vel(ii)=168;
end
V(:,1)=Vel;

for j=2:length(x)
    dS=sqrt((SampleGps(j,1)-SampleGps(j-1,1))^2+(SampleGps(j,2)-SampleGps(j-1,2))^2);
    dt(j)=dS/V(j);
    t(j)=dt(j)+t(j-1);
end
```
SampleGps(:,4)=t;  %Sample Time
SampleGps(:,5)=V;  %Sample GroundSpeed

return
end

function [SampleGps]= SampleFlightxz

h=-8.71951e6;   %Center of circle latitude (ECEF)
k=-14706253.281; %Center of circle longitude (ECEF)
r=2000;  %Radius of circle (ft)
% V=168;  %ft/s (100 knots)
x1=-8.71951e6 - r;  %ECEF latitude start
x2=-8.71951e6 + r;  %ECEF latitude end

x= x1:10:x2;
y=zeros(length(x),1);
z=zeros(length(x),1);   %assume a constant elevation turn
for i=1:length(x)
    z(i)=k+sqrt(r^2-(x(i)-h)^2); %+rand*.1;
xr(i)=x(i); %+rand*.1;
end

SampleGps(:,1)=xr;
SampleGps(:,2)=y;
SampleGps(:,3)=z;
t=1;

for ii=1:length(x)
    Vel(ii)=168;
end
V(:,1)=Vel;

for j=2:length(x)
    dS=sqrt((SampleGps(j,1)-SampleGps(j-1,1))^2 + (SampleGps(j,2)-SampleGps(j-1,2))^2 + (SampleGps(j,2)-SampleGps(j-1,2))^2);
    dt(j)=dS/V(j);
    t(j)=dt(j)+t(j-1);
end

SampleGps(:,4)=t;  %Sample Time
SampleGps(:,5)=V;  %Sample GroundSpeed

return
end
Calculate Orientation (calcOrientation)

function [orientation] = calcOrientation(tan_Vector,norm_Vector,data,dataGps)

%sensorSampRate = (dataGps.timeSync(length(data.Gps),1)-
dataGps.timeSync(1,1))/length(dataGps);

for k=4:length(data.time)

    % Total Acceleration due to norm/tan accelerations in ECEF coordinates
    LoadAccelECEF = tan_Vector + norm_Vector;

    % Total Load Accel reoriented for direction of phone's axes (90 deg rotations from ECEF)
    LoadAccelPlane = LoadAccelECEF * [1 0 -1;
                                      1 0 0;
                                      0 -1 0];

    % Creates a matrix of equivalent length to sensor file that hold GPS point for all sensor values until the GPS point changes
    j=1;
    for m=1:length(data.timeSync)
        if data.timeSync(m,1) >= dataGps.timeSync(j,1)
            j=j+1;
        end
        AccelPlaneAltered(m,1) = LoadAccelPlane(j,1);
        AccelPlaneAltered(m,2) = LoadAccelPlane(j,2);
        AccelPlaneAltered(m,3) = LoadAccelPlane(j,3);
    end

    % Acceleration measured on phone accelerometers
    accelMeasured = [data.a_x data.a_y data.a_z];

    input = [accelMeasured(k,1) accelMeasured(k,2) accelMeasured(k,3)
             AccelPlaneAltered(k,1) AccelPlaneAltered(k,2) AccelPlaneAltered(k,3)];  % Input values in vector form

    if (k==4)
        x0 = [0 10 0 3 4 3]; % Initial conditions to fsolve
    else
        x0 = [output(k-1,:)];
    end

    output(k,:) = fsolve(@(output)findAngles(output,input),x0); % Calls function and uses fsolve to determine angles

    gravComp = [output(1,4) output(1,5) output(1,6)];
    orientation(k,1) = output(k,1);
    orientation(k,2) = output(k,2);
    orientation(k,3) = output(k,3);

end
disp('Done with orient fsolve')
end

function F = findAngles(output,input)
    LoadAccel = [input(4) input(5) input(6)];
    orient = [output(1) output(2) output(3)];
    gravity = [output(4) output(5) output(6)];
    % Rotational Matrix in 3d
    RotMatrix = [cosd(orient(2))*cosd(orient(3))
                 cosd(orient(2))*sind(orient(3)) -sind(orient(3));
                 sind(orient(1))*sind(orient(2))*cosd(orient(3))-
                 cosd(orient(1))*sind(orient(3))
                 cosd(orient(1))*cosd(orient(3))+sind(orient(1))*sind(orient(2))*sind(orient(3))-
                 sind(orient(1))*sind(orient(3))+cosd(orient(1))*sind(orient(2))*sind(orient(3));
                 sind(orient(1))*sind(orient(3))+cosd(orient(1))*sind(orient(2))*cosd(orient(3));
    F(1) = LoadAccel*RotMatrix(:,1) - input(1) + gravity(1);
    F(2) = LoadAccel*RotMatrix(:,2) - input(2) + gravity(2);
    F(3) = LoadAccel*RotMatrix(:,3) - input(3) + gravity(3);
    F(4) = gravity(1) -
    tand(orient(3))*sqrt(((gravity(2))^2)+((gravity(3))^2)); % Yaw equation
    F(5) = gravity(2) +
    tand(orient(1))*sqrt(((gravity(1))^2)+((gravity(3))^2)); % Bank equation
    F(6) = gravity(3) -
    tand(orient(2))*sqrt(((gravity(1))^2)+((gravity(2))^2)); % Pitch equation
end

Plot Test Flight (plotTestFlight)

% This function plots the phone's calculated values vs the G1000 truth
% source for each test point for the test flight on 5/3/13.
% %
% Functions required:
% - flightTestPointToFilename()
% - importCellPhoneFlightParametersFile()
% - importG1000File()
% - calcEcef2()
% - calcGroundSpeed2()

% %
% Paths required:

addpath(genpath('Test2Data'));
addpath('Matlab Functions');
addpath('Flight Test Analysis Functions');

% This is where the test point of the flight is chosen (1-24)
testpoint=1;
% Turns the test point input into the test point file
filename=flightTestPointToFilename(testpoint);
% Imports the data collected from the cell phone
data = importCellPhoneFlightParametersFile(
  filename.nexus4FlightParameters{1});
% Imports the truth source data from the G1000
dataReal = importG1000File2(filename.g1000{1});

% Creates a normalized time vector for the cell phone data
tfp = data.time;
% Creates a normalized time vector for the G1000 data
tReal = 0:1:length(dataReal.lat)-1;

% These next two variables are here to perform a correction for the ground
% speed computed during the flight
ecef = calcEcef2(data);
gndSpeedGps = .59248*calcGroundSpeed2(ecef,true);
for k = 2:length(data.yaw);
  if(data.yaw(k)<0)
    data.yaw(k)  = data.yaw(k)+360.0;
  end
end

%% Functions to calculate pitch using Method 3
headingFiltered = filterHeading(data.yaw,0.05);
tAvgHead = 1:1 : length(headingFiltered)/50;
avgHeading = calcAvgHeading(headingFiltered);
distance = calcDistance(ecef);
rad_curv3=calcRadCurv3(avgHeading,distance);

bank3=calcBank2(gndSpeedGps,rad_curv3);

%% The following figures are comparisons of cell phone data vs G1000 data
% To continue to add plots follow the same form using tfp for flight
% parameters and tReal for G1000 data. To see what other options there are
% to plot see importCellPhoneFlightParametersFile() and importG1000File()
figure (1)
plot(tfp,data.bank,tReal,dataReal.roll);
xlabel('time (s)');
ylabel('bank (deg)')
title ('Bank')
legend('phone','G1000')
figure (2)
plot(tfp,data.pitch,tReal,dataReal.pitch);
xlabel('time (s)');
ylabel('pitch (deg)')
title ('Pitch')
legend('phone','G1000')
figure(3)
plot(tfp,headingFiltered,tReal,dataReal.heading,tAvgHead,avgHeading);
xlabel('time (s)');
ylabel('Heading (deg)')
title('Heading')
legend('average heading','filtered heading')
figure (4)
plot(tfp,data.lat,tReal,dataReal.lat);
xlabel('time (s)');
ylabel('Lat (deg)')
title('Latitude')
legend('phone','G1000')
figure (5)
plot(tfp,data.lon,tReal,dataReal.lon);
xlabel('time (s)');
ylabel('Lon (deg)')
title('Longitude')
legend('phone','G1000')
figure (6)
plot(tfp,data.altBaro,tReal,dataReal.alt_baro);
xlabel('time (s)');
ylabel('altBaro (ft)')
title('Altitude Baro')
legend('phone','G1000')
figure (7)
plot(tfp,data.altGps,tReal,dataReal.alt_gps);
xlabel('time (s)');
ylabel('altGps (ft)')
title('Altitude GPS')
legend('phone','G1000')
figure (8)
plot(tfp,gndSpeedGps,tReal,dataReal.gndSpd);
xlabel('time (s)');
ylabel('Ground speed (kt)')
title('Ground Speed')
legend('phone','G1000')
figure(9)
plot(tAvgHead,rad_curv3)
title('Radius of Curvature from Heading')
figure(10)
plot(tAvgHead,bank3)

Import Cell Phone Flight Parameters File (importCellPhoneFlightParametersFile)

function [data] = importCellPhoneFlightParametersFile(filename)
% This function imports the data from the cell phone flight parameters csv
% file and
% outputs a structure containing the data of each column.
% % Troubleshooting:
% 1) Be sure that you have added the path containing the file you wish to
% import in your script file.
% 2) The filename must end in '.csv'.
% 3) If using the string output from the flightTestPointToFilename() function, be
% sure that the input to this function is "filename.device{i}" and not % just "filename.device".

%% Import the file
x = csvread(filename,1,0);

%% Create a structure containing each column of data
data.timeGps = x(:,1); %ms
data.lat = x(:,2); %deg
data.lon = x(:,3); %deg
data.altGps = x(:,4)*3.2808; % ft MSL
data.altBaro = x(:,5)*3.2808; % ft MSL
data.bank = x(:,6); %deg
data.pitch = x(:,7); %deg
data.yaw = x(:,8); %deg
data.gndSpd = x(:,9); %kts
data.climbRate = x(:,10); %fps
data.rollRate = x(:,11); %deg/s
data.h = x(:,12); %hr
data.min = x(:,13); %min
data.s = x(:,14); %s
data.time = x(:,15)-x(2,15);
end


Import G1000 File 2 (importG1000File2)

function [data] = importG1000File2(filename)
% This function imports the data from the g1000 csv file and outputs a % structure containing the data of each column.
%
% Troubleshooting:
% 1) Be sure that you have added the path containing the file you wish to % import in your script file.
% 2) The filename must end in '.csv'.
% 3) If using the string output from the testpoint2FileName() function, be % sure that the input to this function is "fileName.device{i}" and not % just "fileName.device".

%% Import the file
fid = fopen(filename);
x = textscan(fid,'%s %s %s %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f ','delimiter',',','headerLines',2);
% 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
fclose(fid);

%% Create a structure containing all data
data.date = x{1};
data.time = x{2};
data.utcOffset = x{3};
data.lat = x{4}; %deg
data.lon = x{5}; %deg
data.alt_baro = x(:,6); %ft
data.baroP = x(:,7);
data.alt_msl = x(:,8);
data.oat = x(:,9);
data.ias = x(:,10); %kts
data.gndSpd = x(:,11); %kts
data.vertSpd = x(:,12);
data.pitch = x(:,13); %deg
data.roll = x(:,14); %deg
data.latAccel = x(:,15);
data.normAccel = x(:,16);
data.heading = x(:,17); %deg
data.gndTrk = x(:,18); %deg
data.alt_gps = x(:,19); %ft
data.tas = x(:,20); %kts
data.crs = x(:,21);
data.magVar = x(:,22);
data.vertSpdGps = x(:,23);

end

**Filter Heading (filterHeading)**

```matlab
function [filtered_heading] = filterHeading(heading, kFilter)

filtered_heading(1) = heading(1);
for k=2:length(heading)
    filtered_heading(k) = (heading(k)*kFilter) + (filtered_heading(k-1)*(1.0-kFilter));
end
```

**Calculate Average Heading (calcAvgHeading)**

```matlab
function [avgHeading] = calcAvgHeading(heading)

for k = 1:length(heading)/50
    avgHeading(k) = sum(heading(2+50*(k-1):k*50))/49;
end
end
```

**Calculate Distance (calcDistance)**

```matlab
function [distance] = calcDistance(ecef)

for k=3:length(ecef)
    x1 = ecef(k-1,1);
x2 = ecef(k,1);
y1 = ecef(k-1,2);
y2 = ecef(k,2);
z1 = ecef(k-1,3);
z2 = ecef(k,3);
distance(2) = 0;
if (x1==x2)
    distance(k)=distance(k-1);
else
    distance(k) = (sqrt(((x2-x1)^2)+((y2-y1)^2)+((z2-z1)^2)));
end
end

Calculate ECEF2 (calcECEF2)
function [ecef] = calcECEF2(gps)
% This function turns (lat,lon,alt) into (x,y,z) from the center of the
% earth
% GPS is imported in 3 columns: lat (1), lon (2), alt (3)

f = 1/298.257224; % flattening parameter
a = 20925647; % earth's radius in feet (WGS 84)

for k=2:length(gps.time)
    c = 1/(sqrt((cos(gps.lat(k)))^2 + ((1-f)^2) * (sin(gps.lat(k)))^2));
    s = ((1-f)^2) * c;
    ecef(k,1) = (a*c + gps.altGps(k)*3.2808) * cosd(gps.lat(k)) *
                 cosd(gps.lon(k)); % ECEF X Value
    ecef(k,2) = (a*c + gps.altGps(k)*3.2808) * cosd(gps.lat(k)) *
                 sind(gps.lon(k));% ECEF Y Value
    ecef(k,3) = (a*s + gps.altGps(k)*3.2808) * sind(gps.lat(k)); % ECEF Z
                  Value
end
ecef(1,1) = ecef(2,1);
ecef(1,2) = ecef(2,2);
ecef(1,3) = ecef(2,3);

Calculate Ground Speed 2 (calcGroundSpeed2)
function [groundSpeed] = calcGroundSpeed2(ecef,filterOn)
% This function determines ground speed from the ECEF coordinates

kFilter = 0.3; % Filtering constant

groundSpeed(1,1) = 220;
groundSpeed(2,1) = 220;
for k=3:length(ecef)
    x1 = ecef(k-1,1);
    x2 = ecef(k,1);
    y1 = ecef(k-1,2);
    y2 = ecef(k,2);
    z1 = ecef(k-1,3);
    z2 = ecef(k,3);

    dt=1;
    if(x1==x2)
        groundSpeed(k,1)=groundSpeed(k-1,1);
    else
        groundSpeed(k,1) = (sqrt(((x2-x1)^2)+((y2-y1)^2)+((z2-z1)^2)))/dt;
    end
    if (filterOn == true)
        if(k<4)
            groundSpeed(k,1) = groundSpeed(k,1);
        else
            groundSpeed(k,1) = (groundSpeed(k,1)* kFilter + groundSpeed(k-1,1)*
                                  (1-kFilter));
        end
    end
end

groundSpeed(1,1) = groundSpeed(3,1);
groundSpeed(2,1) = groundSpeed(3,1);

Calculate Radius of Curvature Method 3 (calcRadCurvature3)
function [radcurv] = calcRadCurv3(heading,distance)
for i = 2:length(heading)
    alpha(i)=heading(i)-heading(i-1);
    radcurv(i)=distance(i)/alpha(i);
end
Appendix J: Matlab Plots

Test Point 1:

![Bank Plot](image)

![Pitch Plot](image)
Test Point 15

- Pitch
- Bank
Test Point 24

Bank

Pitch