GPS + Inertial Sensor Fusion

Group Members:
Aleksey Lykov, William Tarpley, Anton Volkov

Advisors:
Dr. In Soo Ahn, Dr. Yufeng Lu

Bradley University ECE Department

May 9, 2014

Table of Contents

Abstract

An inertial navigation system (INS) or inertial measurement unit (IMU) is a form of dead reckoning navigation system that uses a combination of accelerometer and gyroscope sensors working in concert to detect displacement relative to a starting point[1]. The system measures both linear accelerations given by its accelerometer and angular velocity changes from its gyroscope. World referenced-frame acceleration data can then be integrated to calculate the velocity and position of the sensors over time, but because the INS can only measure motion relative to a starting location, the initial position must be supplied by some outside system (in the case of this project, using a Global Positioning System (GPS) receiver). Additionally, to compensate for the drift in the inertial navigation system caused by various defects in inertial sensors, the outside references (GPS and barometer) must be polled occasionally to correct for the position error.

Using a Raspberry Pi microcomputer as the base system and an MPU 9150 IMU, an inertial navigation system will be developed. Kalman Filtering with GPS and barometer data will be used to complete a “strapdown solution” - a closed-loop system which can self-correct for error [1].

Introduction

The objective of this project is to combine inertial and GPS sensors via I2C on the Raspberry Pi single board computer for the purpose of maintaining accurate positioning in areas without GPS or in the event of losing a GPS signal. This is accomplished with heavy post processing, including Kalman filtering of the inertial measurements (accelerometer, gyroscope data) combined with drift reduction using magnetometer data, and finally through sensor fusion with GPS data.

Background

While much of the project work involved the physical interfacing of sensors, it is important to understand the theory behind the operation of the sensors involved, namely the concepts of inertial navigation.

Inertial Navigation

Inertial navigation refers to estimating a system’s position based on measured changes to the motion of a system. To accomplish this, a combination of acceleration and angular rate measurement is used to compute linear motion as well as attitude (yaw, pitch, and roll). Accelerometers and gyroscopes are used to accomplish this task.

Dead Reckoning

Dead reckoning is the process of estimating a system's position by using the previous position value (such as a coordinate provided by the GPS system) and then calculating a new position based on the sensed movement of the system[1]. In the scope of this project, the GPS provides an initial position and the system calculates future positions based on the integration of world-reference-frame acceleration derived from accelerometer and gyroscope measurements.

Strapdown Solution

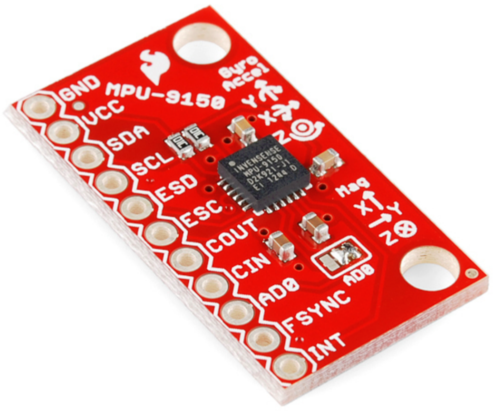
Unlike a gimballed system, which is able to keep inertial sensors at a fixed attitude regardless of system orientation, a strapdown solution has sensors mounted on the system body to measure body-reference-frame changes in inertia[1]. In order to compute acceleration in the fixed or world-reference-frame, any perceived changes in attitude must be integrated to determine system orientation with respect to the starting orientation (in this case, a level orientation with the accelerometer Y-axis facing north). With a known system attitude, it is both possible to compute the direction of gravity's acceleration to remove its influence on system data and additionally to rotate the body-reference-frame acceleration into the world reference frame.

System Design

The complete signal and hardware block diagrams can be seen in figures 1 and 2, respectively. Each system will be discussed in the following section.

InvenSense MPU-9150 Inertial Measurement Unit

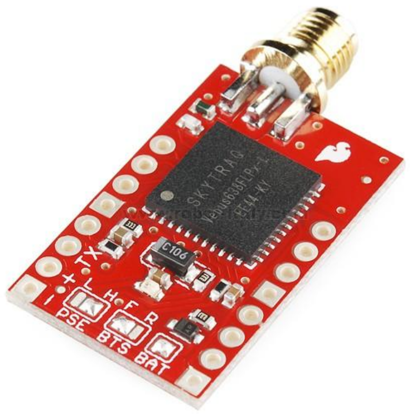
The main sensor used for the project is the Invensense MPU-9150 IMU (Inertial Measurement Unit) with breakout board designed by Sparkfun. This chip sends out nine axes of data: x-acceleration, y-acceleration, z-acceleration, yaw-rate, pitch-rate, roll-rate, and three axes dedicated to magnetometer data. Additionally, the chip has the capability to calculate the quaternion and magnitude of the gravity vector in order to send world-reference-frame data to the user. World reference frame, as opposed to body or system reference frame, is the fixed coordinate frame denoted "North, East, and down" and does not change even as the IMU rotates along any axis. This calculation is performed by the MPU-9150's onboard DMP (Digital Motion Processor) in the form of three 16-bit accelerations (N, E, down) relative to the starting orientation. The magnetometer is used to help the user the system's determine starting orientation so that the north, east and down reference frame of the inertial sensor corresponds to that of the GPS.



InvenSense MPU-9150 Inertial Measurement Unit

SkyTraq Venus634FLPx GPS

A GPS unit, the SkyTraQ Venus634FLPx (breakout board by Sparkfun), is used to give the program its initial position and to correct for inertial sensor error during the course of operation for X and Y axis measurements. The GPS receiver outputs standard National Marine Electronics Association (NMEA) sentences over UART serial communications.



SkyTraq Venus638FLPx GPS Receiver Unit

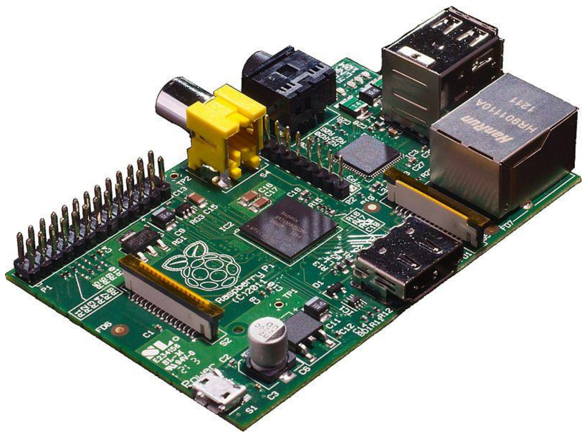
Bosch SensorTec BMP 180 Barometric Pressure Sensor

The GPS unit provides accurate X and Y coordinate positions, however, the application is for navigation in the event GPS is lost. As a result, the BMP 180 (breakout board by Adafruit) was added to correct for deviations in the Z direction even when the GPS is unavailable. The sensor reads surrounding pressure and temperature data which are then used to calculate altitude.

Bosch SensorTec BMP180 Barometric Pressure Sensor

Raspberry Pi Single-Board Computer

The heart of the system is the Raspberry Pi single-board computer. Its Linux operating system is used to run the multi-threaded C++ code which interfaces with sensors and saves processed sensor data to file. The IMU, GPS, and barometer are attached to the I/O pins of the Raspberry Pi via I2C, UART, and I2C, respectively. The program starts three threads which read and timestamp data from the external sensors, at which point a fourth thread processes the data and formats it for saving to file. Once the user stops data acquisition via a GPIO push button, the program writes all saved data to file.



Raspberry Pi Single-Board Computer

Signal Operation

Using the IMU-9150's Digital Motion Processor which performs six-axis sensor fusion to compute a quaternion representation of attitude, the world-reference-frame acceleration is calculated by rotating a linear acceleration vector (raw acceleration with the magnitude of gravity subtracted). This data is then integrated to calculate a displacement from the starting position which the barometer and GPS data is then used to correct in MATLAB.

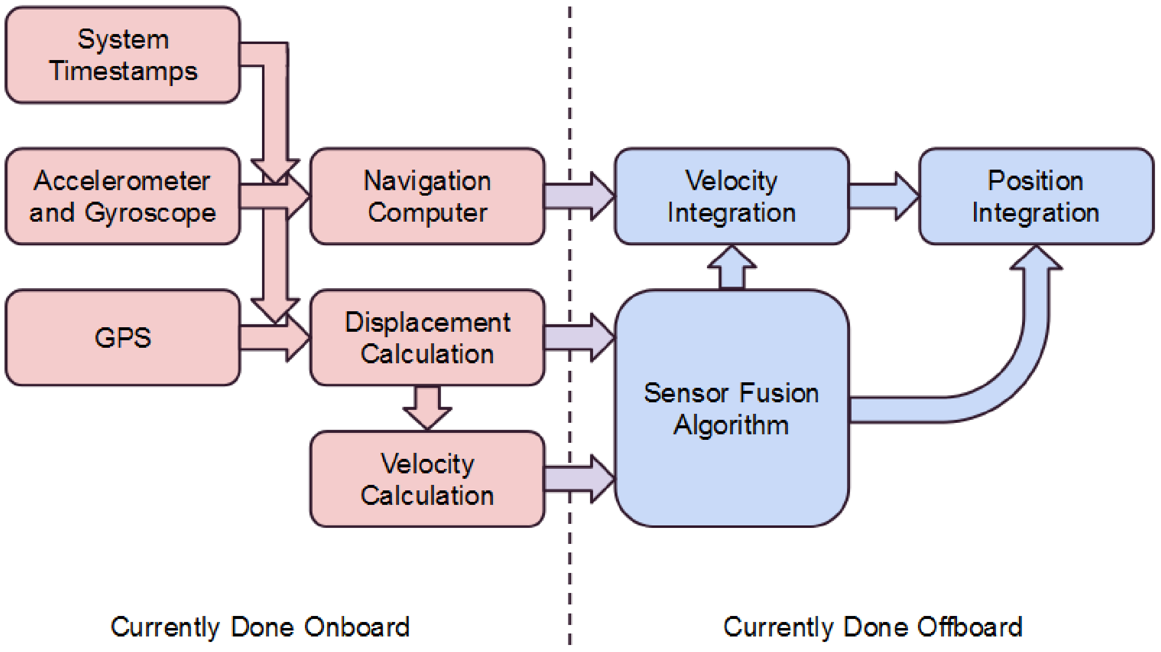
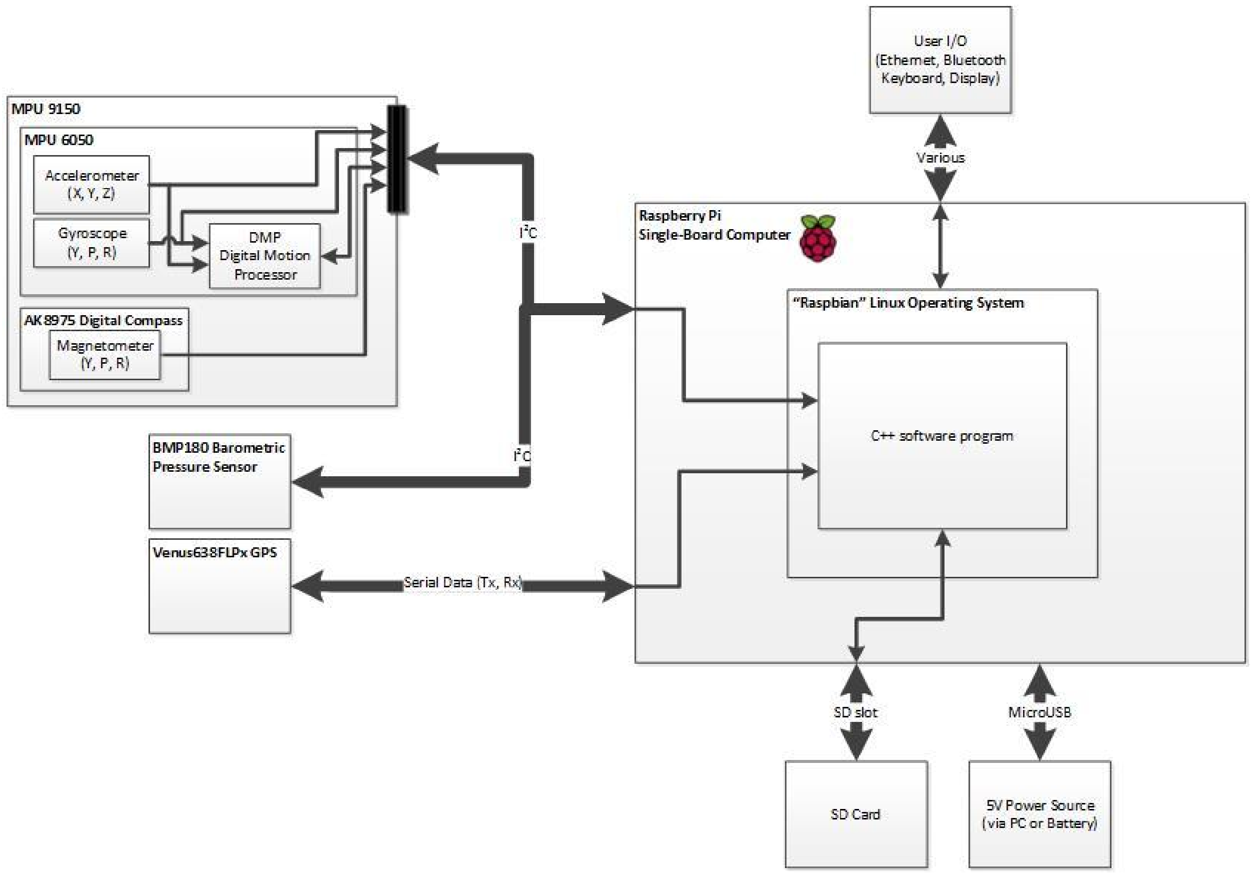


Figure 1. Signal Block Diagram



User I/O (Ethernet, Bluetooth Keyboard, Display)
Various
MPU 9150
MPU 6050
Accelerometer (X, Y, Z)
Gyroscope (Y, P, R)
DMP Digital Motion Processor
AK8975 Digital Compass
Magnetometer (Y, P, R)
I²C
Raspberry Pi Single-Board Computer
"Raspbian" Linux Operating System
C++ software program
BMP180 Barometric Pressure Sensor
I²C
Venus638FLPx GPS
Serial Data (Tx, Rx)
SD slot
MicroUSB
SD Card
5V Power Source (via PC or Battery)

Figure 2. Hardware Block Diagram

Methods and Procedures

The project work can be divided into several sections, including the initial setup of the Rapsberry Pi, the interfacing of inertial sensors, the writing of MATLAB post-processing code, the interfacing of the GPS receiver, the writing of multi-threaded data acquisition code, and the testing methods used to acquire data:

Setting up the Raspberry Pi

The first step in making the data acquisition system was to set up the Raspberry Pi linux-based single board computer. This was achieved in the following steps:

1. Installing the "Raspbian" Linux distribution
2. Setting up SSH control of the system through PuTTY on Windows
3. Setting up samba file sharing between the Raspberry Pi and Windows
4. Setting up internet connection sharing via ethernet (sharing the host computer's WiFi)
5. Setting up WiFi on the Raspberry Pi via USB wireless adapter

6. Setting up cross-compilation on the Pi via NetBeans in Windows (this was later replaced by onboard g++ compilation)
7. Setting up bluetooth keyboard and composite display output for portable operation (later testing was controlled with a GPIO pushbutton and LED for information)
8. Installing program-related drivers (*wiringPi[7]* for GPIO and serial control, *i2cdev* for reading and writing to the I^{2}C bus, and *ncurses* for additional I/O while running program)

Once setup was complete, it became possible to write code on a Windows PC, send it to the Pi via PSCP or samba file sharing, and control the Pi via SSH in PuTTY. Additionally, the Pi could be powered by an external 5V battery for an extended period of time (several hours) and taken on indoor/outdoor tests to collect data to file for later analysis.

Interfacing the Inertial Sensors

The MPU-9150 was connected to the Raspberry Pi via the I^{2}C interface. It is composed of a MEMS accelerometer and gyroscope (which together make up the *MPU6050* IC) and a MEMS magnetometer (making up the *AK8975* IC). Using open-source drivers[5] written for the MPU-6050 six-axis accelerometer+gyroscope IC, it is possible to read either raw acceleration, gyroscope, and magnetometer data from the respective devices or to make use of the onboard digital motion processor (DMP) which performs 6-axis sensor fusion with the accelerometer and gyroscope data. In the following steps, the DMP data can be used to determine world-reference frame data suitable for a strapdown solution:

1. Read quaternion data from the DMP
2. Read raw acceleration data from the DMP
3. Compute the gravity vector using the quaternion data
4. Remove the gravity component from the raw acceleration data
5. Convert body-reference-frame acceleration to world-reference frame acceleration with quaternion data

The DMP also has the ability to return euler angles for yaw, pitch, and roll of the MPU6050 since initialization. In this project, the above procedure with DMP data was in favor of reading raw acceleration and gyroscope data.

Writing MATLAB Post-Processing Code

Data was saved to a text file using the Raspberry Pi in a predetermined format for timing, GPS, INS, and Barometric data. In MATLAB, the following tasks were completed and implemented as functions:

1. Read in saved data
2. Integrate acceleration data to velocity and position
3. Filter Data
4. Present Data

Data was read in using the textscan function. This saves data matching the passed format into a cell array. The code then saves each array location to a corresponding variable and returned to main. Then, the read acceleration and time data is used to integrate data and determine velocity and position using the trapezoidal method. Any filtering would be completed at this time. Methods for filtering included the following:

1. Removing acceleration offsets which occurred after an acceleration.
2. Removing accumulated offsets which occurred as acceleration was sensed.
3. Implementing a “deadzone” to compensate for idle noise and vibration.

In order to remove an offset, MATLAB searched for areas where the min and max deviation fell within a certain difference. The area size and difference were determined by variables and could be changed in post processing. The accumulated offset was removed by searching where one offset ended and the next began. Since each acceleration resulted in an offset, any space between offsets were built by a gradually increasing or decreasing offset. This was removed by assuming the growth was linear and removing the slope of acceleration. Lastly, the data was compared to the deviance recorded during initialization and all point falling within these bounds were pulled to 0. Figures 3 and 4 show the unfiltered and filtered data respectively. Finally, the data is plotted to show information such as velocity over time, displacement, or acceleration over time.

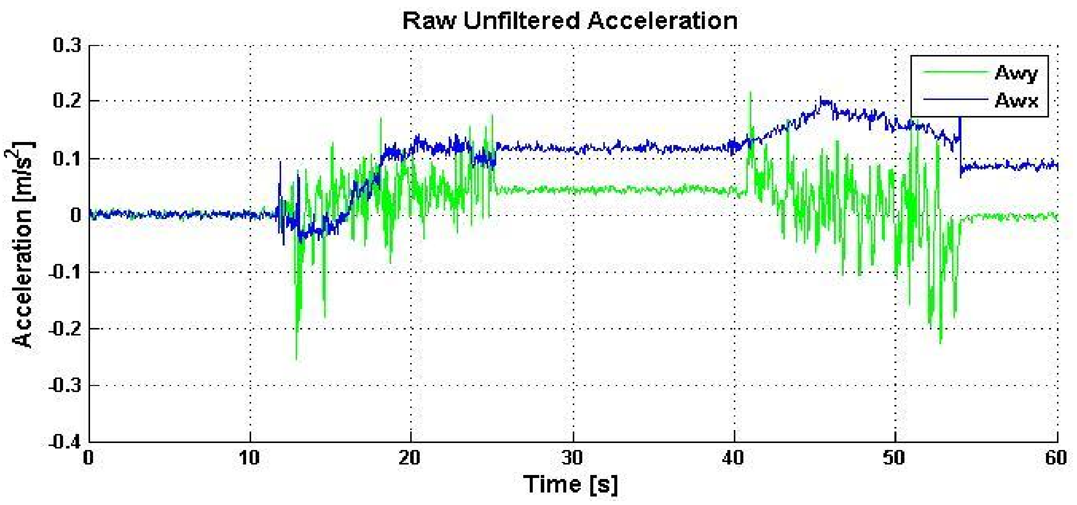


Figure 3. Accelerations in the X and Y axes before filtering

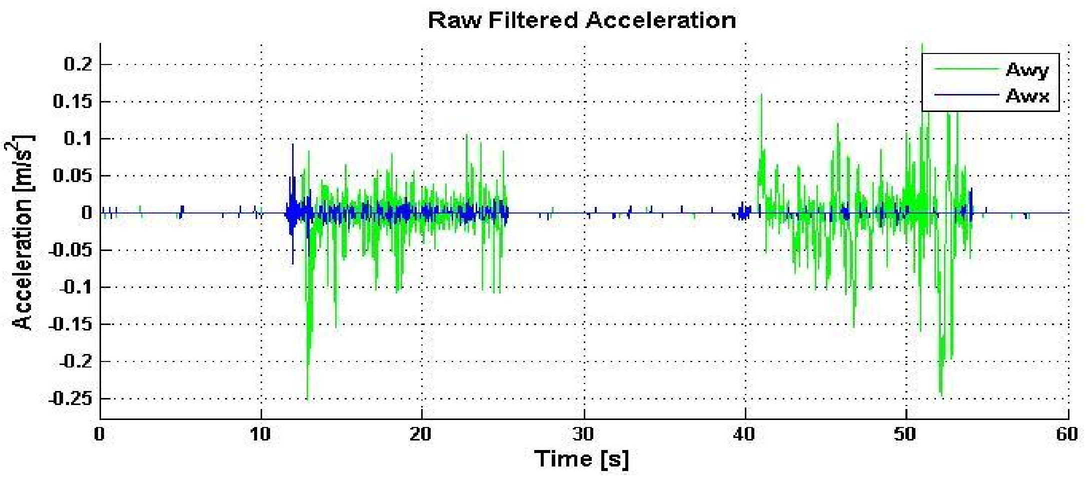


Figure 4. Accelerations in the X and Y axes after filtering

Interfacing the GPS Unit

The Venus634FLPx GPS receiver was connected to the Raspberry Pi via the serial port. In its default configuration, the chip sends standard NMEA messages at a rate of 1 Hz over a 9600 baud serial interface while a GPS lock is maintained.

The receiver has several modes of data presentation, and they are all available simultaneously, and are constantly sent over the serial bus. This data was parsed and the desired format (GPGGA) of GPS data was used for position measurements.

The GPGGA format of GPS data provides the following information:
- Time - in 24hr format
- Latitude
- Longitude
- Fix quality -indicating whether a GPS fix is present and whether differential GPS is available
- The number of satellites
- The horizontal dilution of precision - that indicates the accuracy of horizontal position
- Altitude
- Height of geoid above the WGS84 ellipsoid - specified in meters
- Time since the last DGPS update - if DGPS is available
- DGPS reference station ID - if DGPS is available
- Checksum - used to check for transmission errors

From this signal the latitude and longitude data was parsed for position calculation.

Writing Multithreaded Data Acquisition Code

Once multiple sensors were being read from reliably, the next task was to write multithreaded C++ code in order to reliably take data from multiple sensors without delays or overflows – the DMP of the MPU6050 chip has a 1024-byte FIFO buffer which will overflow if readings are not taken as soon as they are available.

Multithreading was accomplished via the POSIX C library, enabling multiple concurrent threads (the main function, one additional thread for each sensor, and a final thread for post-processing). Mutexes were initially used to control access to variables in shared memory (the data logged by each thread) between the data acquisition threads and the post-processing thread (which would format and save the data for printing to file). It was discovered, however, that mutexes are not required unless both threads are writing to the same variable. Since the processing thread only read from the variable, neither thread needed to use mutex locks, and they were removed. Finally, string streams were implemented for use by the post-processing thread instead of constant file writes (the latter resulted in inconsistent data acquisition times). See Appendix A for the complete main.cpp function. Finally, in addition to including necessary libraries in the working directory, the following command was used to compile the final code:

```
g++ -Wall main.cpp Adafruit_BMP085_U.cpp I2Cdev.cpp MPU6050.cpp -o test1 -lncurses
-pthread -lrt -lwiringPi
```

Interfacing the Barometer

Adding the BMP180 Barometric Pressure Sensor was a late decision when it was discovered that barometric sensors were very useful for altitude measurements. In the absence of GPS, this sensor can help correct the inertial measurement data for altitude change calculation. Just as with the MPU-9150, the sensor was connected to the system via I^2C. The greatest challenge with interfacing this sensor was the particular process for reading pressure and temperature data from its registers and converting it into the desired units of Pascals and degrees Celsius. A close inspection of the BMP180 datasheet to decipher the I^2C registers' little- and big-endian notation as well as expected datatype lengths was necessary in order to modify the Adafruit-supplied C++ drivers[6] (intended to be used on 16-bit microprocessor).

The sensor provides the system with both barometric pressure and temperature measurements, which are used to calculate altitude.

System Operation

The system functioned in the following states:

1. In order to relay that the system has completed startup, the program notifies that user that it is ready to begin with a solid LED, waiting for the user to press the push button.

2. The program reads data from the magnetometer and lights the LED when the system faces North. This state is advanced by pressing the pushbutton.
3. The program enters the initialization process which allows for the DMP to stabilize. After 25 seconds of fast LED flashing during this time, the program continues.
4. The program begins storing accelerometer, gyroscope, and GPS data to memory. The LED pulses at 1 Hz and the state is advanced by pressing the pushbutton.
5. The LED turns off and the string buffer is written out to the SD card, ending the program.

After the program terminates the user can move data from the SD card to a PC. Resetting the Raspberry Pi will cause the program to restart at state 1 and will create a different save file allowing for multiple data collections without loss of data.

Results

The resulting system is a reliable data acquisition system using a Raspberry Pi with integrated inertial and GPS sensors. A summary of final system specifications is seen in Table 1.

Specification	**Final Result**
Accelerometer/Gyroscope Rate	100 Hz from DMP (raw data available at 1 kHz) [3]
GPS Rate	1 Hz (scalable up to 20 Hz) [4]
Barometer Rate	1 Hz (scalable up to 100 Hz) [2]
Time to Acquire GPS Lock	3-4 Minutes from Cold Start 30 Seconds to Reacquire Lock
Program Structure	C++, 5 Threads (Main, 3 Sensor Threads, Post-Processing)
Data Log Storage	Text file on SD card Fill rate: ~539 kB/min
Post-Processing	GPS displacement/velocity calculations performed onboard Sensor fusion and integration performed offboard in MATLAB
CPU Utilization	Under 17% average usage with current data acquisition code RTIMULib[5] 4-state Kalman filter code tested, <20% usage

Table 1. Final system Specifications

The system can be operational within 30 seconds of startup. Gyroscope stabilization is completed in the 30 seconds following power-up. The system acquires a GPS lock within three to four minutes and, provided it has sufficient power and makes use of the added capacitor to remain powered up when power is removed, is able to reacquire a GPS lock in 30 seconds after a loss when GPS is available.

Data is acquired from the external sensors without any delays, FIFO buffer overflows (of the INS module), or delayed timestamps. Inertial sensor data is recorded at 100 Hz, GPS data at 1 Hz, and barometer data at 1 Hz. All of this data is stored to the SD card with timestamps at a rate of approximately 539 KB per minute. The SD card data log includes the following timestamped data: N/E/Down (Y/X/Z) accelerations; GPS latitude, longitude, the change in GPS displacement since last coordinates, and the change in GPS velocity since last coordinates; barometer-measured altitude and temperature.

Data rates for the various sensors as well as system sensitivity can be adjusted if needed. The raw data from the accelerometer, gyroscope and magnetometer can be read in at speeds up to 1kHz, though the Digital Motion Processor is limited to 100 Hz to reduce the amount of noise. The barometer data can be read in at speeds up to 200 Hz and the GPS data can be read at speeds up to 20 Hz, if desired. Additionally, the sensitivity of the accelerometer can be adjusted to ±2g, ±4g, ±8g, or ±16g. The gyroscope angular rate is set to ±250 degrees/second for maximum resolution, but can be increased to ±2000 degrees/second for faster response times.

While maintaining a GPS lock, it is possible to correct inertial data velocity and position to maintain accuracy within several meters of true position even after extended periods of time. However, if GPS is not available, stationary displacement reported is within approximately one meter, but when moving, odometer distance error is often more than 50%, with an even greater absolute displacement error.

Through testing, it was discovered that the MEMS accelerometer sensors were not reliable on their own for purposes of position estimation due to accumulating error. Figure 5 shows the sensor fusion method in which the velocity and displacement integration is corrected to GPS coordinates once per second, and the significant influence of a simulated GPS outage.

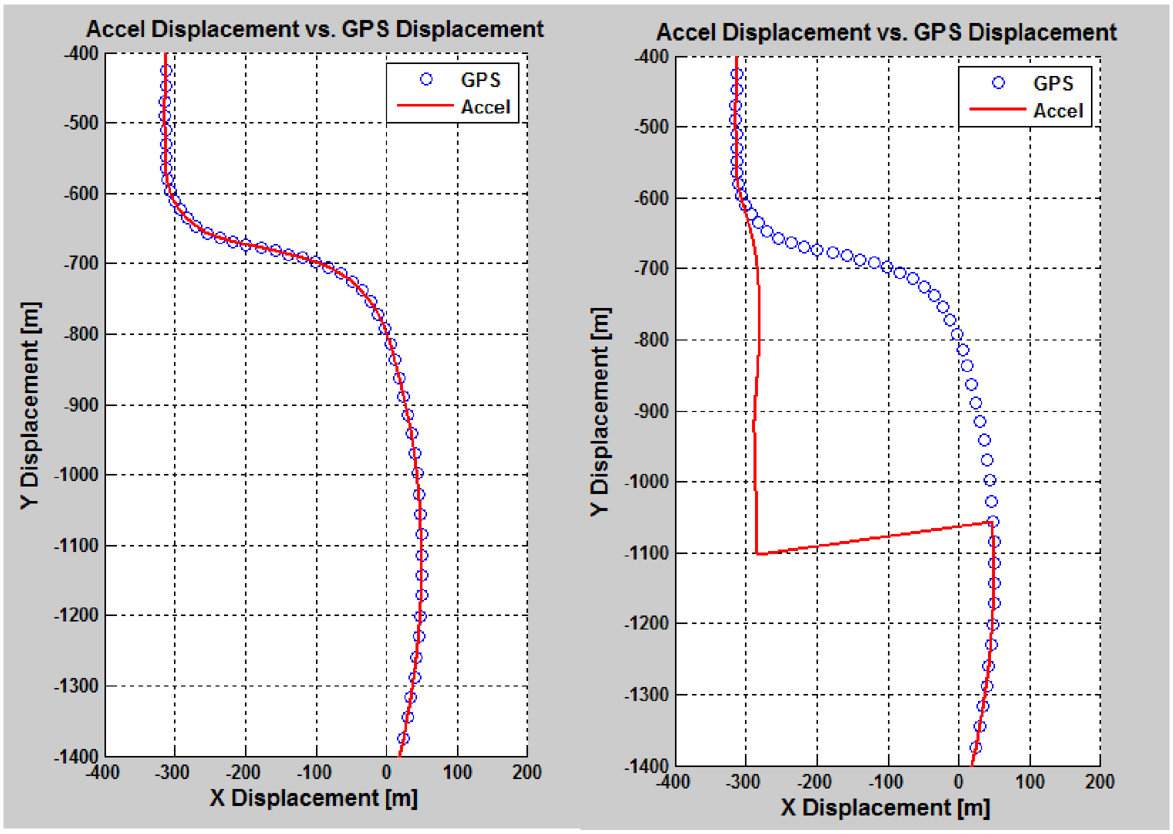


Figure 5. Sensor fusion and GPS outage during highway test

On a smaller scale, the system can appear to move much farther than it actually has due to the accumulating displacement error which results from the double integration from acceleration to displacement. Figure 6, depicts the displacement of the system as it is moved in a square holding pattern with one foot side lengths.

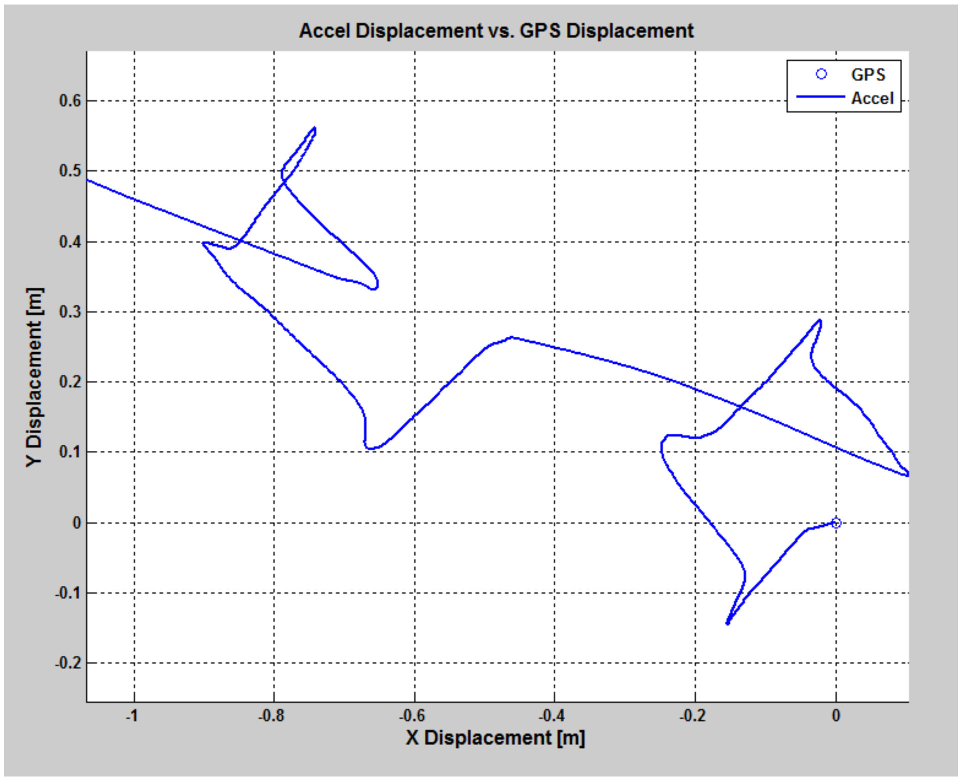


Figure 6. X-Y displacement during one foot square test without rotation

Linear tests show that when a system is displaced in a single dimension, an offset in acceleration data is introduced when the system is stopped. Worse yet, an additional acceleration is sensed in an axis perpendicular to the system's movement. This problem causes a drift that can be seen when the acceleration is integrated to show a displacement.

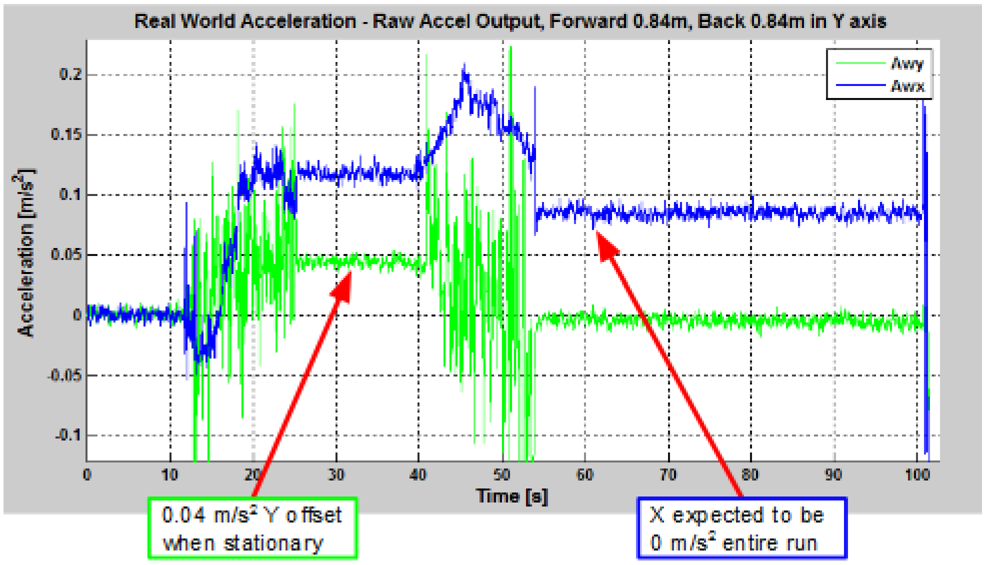


Figure 7. Linear acceleration test to showing raw acceleration data errors

Using Kalman filtering, it is possible to combine sensor data to help mitigate any errors within a single sensor. For example, the gyroscope and digital compass sensors can be combined to create a more reliable system attitude measurement. Combining barometer and z-axis acceleration data, it is possible to get a more reliable estimate of altitude changes. Finally, combining GPS and X and Y axis accelerometer data, it is possible to get reliable displacement estimate within a couple of meters of actual, real-world displacement. However, when GPS data is not available, such as indoors of a large building, in a heavily forested area, inside a large city, or underground, the system has to rely entirely on the MEMS acceleration data for X and Y displacement, and this has been proven to not be reliable for purposes of navigation.

In its current state, the sensor fusion as well as position estimation is done offboard using a MATLAB, however system testing indicates that implementing this process on the system as it is today should not cause any trouble.

Conclusion

The project was successful in building a reliable Data Acquisition system for Inertial Navigation with GPS integration. The system is able to get data from sensors, perform some computation and store the results. After building and extensively testing a data acquisition platform, it was determined that the current MEMS accelerometer is insufficient to provide accurate indoor positioning via dead reckoning or double integration - even constrained to a single axis, the accelerometer it was found to report positioning errors well over 100%. Conclusions therefore include that for accurate indoor navigation, our platform must rely on either additional positioning sensors with which a Kalman filter can correct the world-reference-frame acceleration data, such as a tachometer for a mobile robot application, or it must rely on an application-specific algorithm, such as a pedometer algorithm for pedestrian applications. Future project work should revolve around the creation of such an algorithm or investigation into corrective indoor position sensors.

Acknowledgements

The authors would like to thank Dr. Yufeng Lu and Dr. In Soo Ahn for their invaluable support and expertise throughout the duration of the project.

References

[1] D. Titterton and J. Weston, Strapdown Inertial Navigation Technology, 2nd ed. Reston, VA: AIAA, 2005.

[2] Bosch SensorTec. BMP180 Datasheet, 5 Apr. 2013. Web. *http://www.adafruit.com/datasheets/BST-BMP180-DS000-09.pdf*

[3] InvenSense. MPU-9150 Datasheet, 18 Sept. 2013. Web. *http://www.invensense.com/mems/gyro/documents/PS-MPU-9150A-00v4_3.pdf*

[4] SkyTraq Technology, Inc. Venus638FLPx GPS Receiver Data Sheet, 24 Feb. 2010. Web. *https://dlnmh9ip6v2uc.cloudfront.net/datasheets/Sensors/GPS/Venus/638/doc/Venus638FLPx_DS_v07.pdf*

[5] A. Weiss and J. Rowberg. 9 Degrees of Freedom - MPU-9150 Breakout (2013), GitHub repository. https://github.com/sparkfun/MPU-9150_Breakout

[6] K. Townsend. Adafruit Unified BMP085/BMP180 Driver (2013), GitHub repository. https://github.com/adafruit/Adafruit_BMP085_Unified

[7] G. Henderson. Gordon's Projects - WiringPi (2011), Web. https://projects.drogon.net/raspberry-pi/wiringpi/

Appendix A: Project Code

Main.cpp:

```
/*
* File:   main.cpp
* Author: Anton Volkov, Aleksey Lykov, William Tarpley
* Multithreaded version with hardware (3 separate threads: GPS, INS, PROC)
*
* Created on November 5, 2013, 2:33 PM
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>
#include <math.h>

#include <iostream>
#include <iomanip>
#include <time.h>
#include <fstream>
#include <sstream>
#include <ncurses.h>

//Serial Libraries:
#include <wiringPi.h>
#include <wiringSerial.h>

//BMP180 Library:
#include "include/Adafruit_BMP085_U.h"
```

```
//MPU-9150 Libraries:
#include "include/I2Cdev.h"
#include "include/MPU6050.h"
#include "include/MPU6050_6Axis_MotionApps20.h"
#include "include/helper_3dmath.h"

//For delays in no-HW Windows testing:
// #include <windows.h>

//POSIX
#include <pthread.h>
#define NUMTHREADS 4 //ins, gps, bar, proc
// #define NUMTHREADS 3

using namespace std;

pthread_mutex_t insfile_mutex; //ins.txt file control mutex
pthread_mutex_t gpsfile_mutex; //gps.txt file control mutex
pthread_mutex_t barfile_mutex; //bar.txt file control mutex
pthread_mutex_t procfile_mutex; //proc.txt file control mutex
pthread_mutex_t procrawfile_mutex; //procraw.txt file control mutex

pthread_t insThread;
pthread_t gpsThread;
pthread_t barThread;
pthread_t procThread;

///////////////////////////////////////////////////////////////////////////

//output files (text documents accessed by all functions one at a time)
// FILE * procFile;
// FILE * procRawFile;
ofstream procFile;
ofstream procRawFile;

stringstream ssBuffer1, ssBuffer2; //temporary storage of data to be saved to SD card
int resMem = 50; //Megabytes
///////////////////////////////////////////////////////////////////////////

//Thread functions:
void *thread_INS(void*);
void *thread_GPS(void*);
void *thread_BAR(void*);
void *thread_PROC(void*);
//Mag start procedure functions:
void magInit();
void getMag(int16_t* mx, int16_t* my, int16_t* mz);
void readMagData(int16_t* mx, int16_t* my, int16_t* mz);
int selfTest(int16_t* mx, int16_t* my, int16_t* mz);
int checkNorth(int16_t* mx, int16_t* my, int16_t* mz);
//Post processing functions:
double gpsDisplacement(double *lat1, double *long1, double *lat2, double *long2);

//Global variables (passing b/w threads):
volatile double INSrecT, BARrecT, GPSrecT; //Timestamps
volatile float INSrecX, INSrecY, INSrecZ; //Accelerometer (X, Y, Z)
volatile double GPSrecN, GPSrecE; //GPS (Northing, Easting)
volatile float BARrecA, BARrecC; //Barometer (Altitude, Celsius)
volatile double ACLrawT, MAGrawT; //Timestamps
volatile float ACLrawX, ACLrawY, ACLrawZ; //Accelerometer Raw Data (X, Y, Z)
volatile float GYRrawX, GYRrawY, GYRrawZ; //Gyroscope Raw Data (X, Y, Z)
volatile float MAGrawX, MAGrawY, MAGrawZ; //Magnetometer Raw Data (X, Y, Z)

volatile int gStop;
struct tm * timeinfo;
struct timespec gettime_now;
struct timespec startTime;
time_t rawtime;
//

int main() { //create files, start INS, GPS, and post processing threads, listen for user input to end
program
{
    //Set up GPIO
    wiringPiSetupGpio(); //set up using Broadcom GPIO pin numbers

    pinMode(11, OUTPUT); //set LED pin to output
    pinMode(7, INPUT); //set PB pin to input
    pullUpDnControl(7, PUD_UP);
    digitalWrite(11, 1); //turn on LED on to signal start of program to user
    sleep(2);
    digitalWrite(11, 0); //
    usleep(200000);
    while(digitalRead(7)){ // flash to inform user that program has started
        digitalWrite(11, 1);
        usleep(100000);
        digitalWrite(11, 0);
        usleep(100000);
    }
    while(!digitalRead(7)){ usleep(10000); } //wait for user to let go of button
    usleep(250000); //wait .25 seconds for de-bouncing

    //File creation
    // Get current time with which to timestamp filenames:
    time ( &rawtime );
    timeinfo = localtime ( &rawtime );

    char str[60];

    // Create PROC file:
    pthread_mutex_lock(&procfile_mutex);
    strftime(str, 60, "/home/pi/SP14/Data/PROC_%F_%H-%M-%S.txt", timeinfo);
    // procFile = fopen(str,"w");
    procFile.open(str);

    if(procFile == 0){
        cout << "PROC file creation failed!" << endl;
        return 0;
    } else {
        cout << "File " << str << " opened successfully!" << endl;
    }
    pthread_mutex_unlock(&procfile_mutex);

    // Create PROCraw file:
    pthread_mutex_lock(&procrawfile_mutex);
    strftime(str, 60, "/home/pi/SP14/DataRaw/PROCraw_%F_%H-%M-%S.txt", timeinfo);
    // procRawFile = fopen(str,"w");
    procRawFile.open(str);
```

```
        if(procRawFile == 0){
            cout << "PROCraw file creation failed!" << endl;
            return 0;
        } else {
            cout << "File " << str << " opened successfully!" << endl;
        }
        pthread_mutex_unlock(&procrawfile_mutex);

        // Reserve ssBuffer1/ssBuffer2 memory:
        try{
            ssBuffer1.str().reserve(resMem*1024*1024);
            cout << "String Stream Buffer 1: " << resMem << "MB Reserved successfully." << endl;
        }catch (std::bad_alloc e){
            cout << "Failed to reserve " << resMem << "MB of space." << endl;
            return(0);
        }

        try{
            ssBuffer2.str().reserve(resMem*1024*1024);
            cout << "String Stream Buffer 2: " << resMem << "MB Reserved successfully." << endl;
        }catch (std::bad_alloc e){
            cout << "Failed to reserve " << resMem << "MB of space." << endl;
            return(0);
        }
    }
    { //Initialization and Main Loop

        //Starting conditions:
        gStop = 1; //threads will wait while gStop > 0

        //Set up magnetometer to notify user of northern direction
        magInit(); //makes magnetometer separate I2C device
        int16_t mx, my, mz;
        if(!selfTest(&mx, &my, &mz))
            cout << "Magnetometer self test failed! Readings (X, Y, Z): "
                 << mx << ", " << my << ", " << mz << endl;

        //Wait for user to push button (ideally when facing North)
        cout << "Orient Device North, Press Button to Start (Press again to stop)..." << endl;
        while(digitalRead(7)){
            getMag(&mx, &my, &mz);
            if(checkNorth(&mx, &my, &mz)){
                digitalWrite(11, 1); //turn on LED on
                cout << "N" << flush;
            } else {
                digitalWrite(11, 0); //turn on LED off at start
                cout << "." << flush;
            }
            usleep(150000);
        } //wait for user to press button

        while(!digitalRead(7)){ usleep(10000); } //wait for user to let go of button
        usleep(250000); //wait .25 seconds for de-bouncing

        digitalWrite(11, 0); //turn LED off
        cout << endl << endl;

        //Set Global Start Time:
        clock_gettime(CLOCK_REALTIME, &gettime_now);
```

```
startTime = gettime_now;

gStop = -1; //threads will run while gStop < 0
//Start threads
int rc;
rc = pthread_create(&insThread, 0, thread_INS, 0);
if (rc){
    cout<< "ERROR; return code from pthread_create(ins) is " << rc << endl;
    exit(-1);
}
rc = pthread_create(&gpsThread, 0, thread_GPS, 0);
if (rc){
    cout<< "ERROR; return code from pthread_create(gps) is " << rc << endl;
    exit(-1);
}
rc = pthread_create(&barThread, 0, thread_BAR, 0);
if (rc){
    cout<< "ERROR; return code from pthread_create(bar) is " << rc << endl;
    exit(-1);
}

//Wait 25 seconds to start PROCthread (don't record data until gyro is stable)
for(int i = 0; i < 250 && digitalRead(7); i++){ // flash to inform user that program has started
    digitalWrite(11, 1);
    usleep(49000);
    digitalWrite(11, 0);
    usleep(49000);
}

//Reset Global Start Time:
clock_gettime(CLOCK_REALTIME, &gettime_now);
startTime = gettime_now;

rc = pthread_create(&procThread, 0, thread_PROC, 0);
if (rc){
    cout<< "ERROR; return code from pthread_create(proc) is " << rc << endl;
    exit(-1);
}

//initiate curses (ideally to clear the screen)
initscr();
timeout(0);
intrflush(stdscr, FALSE);
cout << endl << "Running" << flush;
char key = ' ';
int t = 0;
while(key != 'e' && digitalRead(7)){ //check for button press
    //threads are running

    //grabbing key to stop operation (former INS code):
    key = getch();

    usleep(499000);
    if(t%2 == 0){
        cout << " (" << (t/2)/60 << " m, " << (t/2)%60 << " s) " << flush;
        refresh();
        digitalWrite(11, 0); //turn LED off
    }else if(t%10000 == 0 && t > 1){ //~50 MB after 5200 seconds (5k seconds =~ 1.44 hrs)
        procFile << ssBuffer1.str(); //write stringStream to file
        ssBuffer1.str(std::string());
```

```
            procFile << ssBuffer2.str(); //write stringStream to file
            ssBuffer2.str(std::string());
        }else{
            clear();
            refresh();
            digitalWrite(11, 1); //turn LED on
        }
        t++;
    }
    endwin();
    digitalWrite(11, 0); //turn LED off

    /*** EXIT THREADS ***/
    gStop = 1; //threads will stop when !(gStop < 0)
    while(gStop < NUMTHREADS){}
    /*** WAIT FOR INSFILE MUTEX LOCK ***/
    pthread_mutex_lock(&insfile_mutex);
    /*** WAIT FOR GPSFILE MUTEX LOCK ***/
    pthread_mutex_lock(&gpsfile_mutex);
    /*** WAIT FOR BARFILE MUTEX LOCK ***/
    pthread_mutex_lock(&barfile_mutex);
    /*** WAIT FOR PROCFILE MUTEX LOCK ***/
    pthread_mutex_lock(&procfile_mutex);
    /*** WAIT FOR PROCRAWFILE MUTEX LOCK ***/
    pthread_mutex_lock(&procrawfile_mutex);
    /*** CLOSE ALL FILES ***/
    procFile << ssBuffer1.str(); //write stringStream to file
    ssBuffer1.str(std::string());

    procFile << ssBuffer2.str(); //write stringStream to file
    ssBuffer2.str(std::string());

    procFile.close();
    procRawFile.close();
    /*** RELEASE/DELETE ALL MUTEXES ***/
    pthread_mutex_destroy(&insfile_mutex);
    pthread_mutex_destroy(&gpsfile_mutex);
    pthread_mutex_destroy(&barfile_mutex);
    pthread_mutex_destroy(&procfile_mutex);
    pthread_mutex_destroy(&procrawfile_mutex);
}
return(0);
}

// INS THREAD ///////////////////////////////////////////////////////////////////////////
void *thread_INS(void*){
//Variable instantiation
// MPU control/status vars
MPU6050 mpu;
char sensitivity = 1; //sensitivity: 0 = +/- 2g, 1 = +/- 4g, 2 = +/- 8g, 3 = +/- 16g
//uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
uint8_t devStatus;   // return status after each device operation (0 = success, !0 = error)
uint16_t packetSize; // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount; // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// For Raw Data:
int16_t ax, ay, az;
int16_t gx, gy, gz;
//int16_t mx, my, mz;
```

```
// Orientation/Motion vars - used by INS_thread
Quaternion q;        // [w, x, y, z]       quaternion container
VectorInt16 aa;      // [x, y, z]          accel sensor measurements
VectorInt16 aaReal;  // [x, y, z]          gravity-free accel sensor measurements
VectorInt16 aaWorld; // [x, y, z]         world-frame accel sensor measurements
VectorFloat gravity; // [x, y, z]         gravity vector
//float euler[3];      // [psi, theta, phi]  Euler angle container
float ypr[3];        // [yaw, pitch, roll] yaw/pitch/roll container and gravity vector

{ //Initialization
    // Initialize the DMP (talks with accel/gyro):
    // initialize device
    //cout << "Initializing I2C devices..." << endl;
    //insFile << "Initializing I2C devices..." << endl;
    mpu.initialize(sensitivity);
    // verify connection
    //cout << "Testing device connections..." << endl;
    //insFile << "Testing device connections..." << endl;
    if(mpu.testConnection()){
        //cout << "MPU6050 connection successful" << endl;
        //insFile << "MPU6050 connection successful" << endl;
    }else{
        //cout << "MPU6050 connection failed" << endl;
        //insFile << "MPU6050 connection failed" << endl;
        //insFile.close();
        //pthread_exit(0);
    }
    // load and configure the DMP
    //cout << "Initializing DMP..." << endl;
    //insFile << "Initializing DMP..." << endl;
    devStatus = mpu.dmpInitialize();
    // make sure it worked (returns 0 if so)
    if (devStatus == 0) {
        // turn on the DMP, now that it's ready
        //cout << "\nEnabling DMP..." << endl;
        //insFile << "\r\nEnabling DMP..." << endl;
        mpu.setDMPEnabled(true);
        //mpuIntStatus = mpu.getIntStatus();
        // set our DMP Ready flag so the main loop() function knows it's okay to use it
        //cout << "DMP ready!" << endl;
        //insFile << "DMP ready!" << endl;
        // get expected DMP packet size for later comparison
        packetSize = mpu.dmpGetFIFOPacketSize();
    } else {
        // ERROR!
        // 1 = initial memory load failed
        // 2 = DMP configuration updates failed
        // (if it's going to break, usually the code will be 1)
        //cout << "DMP Initialization failed (code " << devStatus << ")" << endl;
        //insFile << "DMP Initialization failed (code " << devStatus << ")" << endl;
        //insFile.close();
        //pthread_exit(0);
    }
    ////////////////////////////////////////////////////////////////////
}
while (gStop > 0){ usleep(1000); } //wait to start

double curTime = 0.0; //seconds since start
double curTimeR = 0.0;
```

```
while(gStop < 0){ //Main loop
    ////////////////////////////////// DATA ACQUISITION:
    // get current FIFO count
    fifoCount = mpu.getFIFOCount();
    clock_gettime(CLOCK_REALTIME, &gettime_now);
    curTime =
        (gettime_now.tv_sec - startTime.tv_sec) +
        ((double)(gettime_now.tv_nsec - startTime.tv_nsec)/(double)1E9);
    if (fifoCount == 1024) {
        // reset so we can continue cleanly
        mpu.resetFIFO();
        cout << "FIFO overflow!" << flush;
        // otherwise, check for DMP data ready interrupt (this should happen frequently)
        fifoCount = mpu.getFIFOCount();
    } if (fifoCount >= 42) {
        //if(fifoCount > 42) //used as a sort of warning message. Overflow occurs at 1024 bytes.
            //    cout << "!!! fifoCount = " << fifoCount << endl;
            // read a packet from FIFO
        mpu.getFIFOBytes(fifoBuffer, packetSize);
        // Process:
        mpu.dmpGetQuaternion(&q, fifoBuffer);
        mpu.dmpGetAccel(&aa, fifoBuffer);
        mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
        mpu.dmpGetGravity(&gravity, &q);
        //adjust for gravity based on sensitivity (��2g, others)
        mpu.dmpGetLinearAccel(&aaReal, &aa, &gravity, sensitivity);
        mpu.dmpGetLinearAccelInWorld(&aaWorld, &aaReal, &q);
        mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
        clock_gettime(CLOCK_REALTIME, &gettime_now);
        curTimeR =
            (gettime_now.tv_sec - startTime.tv_sec) +
            ((double)(gettime_now.tv_nsec - startTime.tv_nsec)/(double)1E9);
        ///////////////////////////////// DATA OUTPUT/RECORDING:
        /*** WAIT FOR INSFILE MUTEX LOCK ***/
        // pthread_mutex_lock(&insfile_mutex); // Determined to be unnecessary

        //save data to memory for PROC.txt:
        INSrecT = curTime; //timestamp
        INSrecX = aaWorld.x;
        INSrecY = aaWorld.y;
        INSrecZ = aaWorld.z;

        //save data to memory for PROCraw.txt:
        ACLrawT = curTimeR; //timestamp
        // ACLrawX = ax;
        // ACLrawY = ay;
        // ACLrawZ = az;
        ACLrawX = aa.x;
        ACLrawY = aa.y;
        ACLrawZ = aa.z;
        GYRrawX = gx;
        GYRrawY = gy;
        GYRrawZ = gz;

        /*** RELEASE INSFILE MUTEX LOCK ***/
        // pthread_mutex_unlock(&insfile_mutex);
        //release mutex lock before next line, allowing thread_PROC to read INSdata.txt
    }
    usleep(1000); //~3ms is the maximum without causing FIFO overflow as of 10/22/13
    //as of 3/31/14, 3ms causes overflow (with multithreading). Adjusted down to 1ms.
```

```
}
gStop++;
pthread_exit(0);
}

// GPS THREAD ///////////////////////////////////////////////////////////////////
void *thread_GPS(void*){
//Variable instantiation
//Serial and GPS variables
char gpsData; //number of GPS bytes available
char gpsMessage[100]; //serial string read in from GPS unit
char gpgga[] = "$GPGGA";

unsigned char arPos; //array position char
double easting;
double northing;

{ //GPS data acquisition
    /*Initialize serial*/
    /*Note: has to be done in order! Cannot access GPS without initializing serial port!*/

    bool ok = false;

    // GPS connection handle
    char serialPort[] = "/dev/ttyAMA0";
    char handle = serialOpen(serialPort, 9600);
    // Serial port opened.

    while (gStop > 0){ usleep(1000); } //wait to start

    double curTime = 0.0;
    while(gStop < 0){ //infinite loop, each iteration fetches/writes a new GPS coordinate to file
        ok = false;
        do{
            gpsData = serialDataAvail(handle);
            if(gpsData > 0){
                ok = true;
                arPos = 0;
                do{
                    gpsMessage[arPos] = serialGetchar(handle);
                    arPos++;
                }while(gpsMessage[arPos-1] != 10 && arPos < 100);
                //ASCII 10 is the line feed (LF) character
                for(int i = 0; i < 6; i++){
                    if(gpsMessage[i] != gpgga[i])
                        ok = false;
                }
            }
        }while(!ok && gStop < 0);
        string gpggaMessage(gpsMessage);

        easting = strtold(gpggaMessage.substr(30,3).c_str(), NULL) +
            (strtold(gpggaMessage.substr(33,7).c_str(), NULL)/60.0000D);
        northing = strtold(gpggaMessage.substr(18,2).c_str(), NULL) +
            (strtold(gpggaMessage.substr(20,7).c_str(), NULL)/60.0000D);

        if(gpggaMessage[41] != 'E')
            easting *= -1;
        if(gpggaMessage[28] != 'N')
            northing *= -1;
        clock_gettime(CLOCK_REALTIME, &gettime_now);
        curTime =
            (gettime_now.tv_sec - startTime.tv_sec) +
            ((double)(gettime_now.tv_nsec - startTime.tv_nsec)/(double)1E9);
        /*** WAIT FOR GPSFILE MUTEX LOCK ***/
        // pthread_mutex_lock(&gpsfile_mutex);

        GPSrecT = curTime; //timestamp
        GPSrecN = northing;
        GPSrecE = easting;
        /*** RELEASE GPSFILE MUTEX LOCK ***/
        // pthread_mutex_unlock(&gpsfile_mutex);
        usleep(999000); //~1s delay
    }
    serialClose(handle);
    /*END SERIAL CODE*/
}
gStop++;
pthread_exit(0);
}

// BAROMETER THREAD ///////////////////////////////////////////////////////////////////////////
void *thread_BAR(void*){
Adafruit_BMP085_Unified bmp180;
int errorCode = bmp180.begin();
if(errorCode < 0){
    //barFile << "Unable to connect to the BMP180! Reason: " << errorCode << endl;
    pthread_exit(0);
}

float pressure = 0;
float temp = 0;
float seaLevel = 1021; //This should be a local value. From website in 3/25/14 documentation (hpa)
float altitude = 0;

{ //Barometer data acquisition

    while (gStop > 0){ usleep(1000); } //wait to start

    double curTime = 0.0;
    while(gStop < 0){ //infinite loop, writes a new pressure/magnetometer reading to mem
        //get pressure
        bmp180.getPressure(&pressure);
        bmp180.getTemperature(&temp);
        altitude = bmp180.pressureToAltitude(seaLevel, pressure/100.0, temp);
        //pressure in hPa required (1 hPa = 100 Pa)

        clock_gettime(CLOCK_REALTIME, &gettime_now);
        curTime =
            (gettime_now.tv_sec - startTime.tv_sec) +
            ((double)(gettime_now.tv_nsec - startTime.tv_nsec)/(double)1E9);

        /*** WAIT FOR BARFILE MUTEX LOCK ***/
        // pthread_mutex_lock(&barfile_mutex);
        //update BAR file

        BARrecT = curTime; //timestamp
        BARrecA = altitude;
        BARrecC = temp;
        /*** RELEASE BARFILE MUTEX LOCK ***/
        // pthread_mutex_unlock(&barfile_mutex);
        usleep(998000); //~1s delay
    }
}
gStop++;
pthread_exit(0);
}

// PROC THREAD ///////////////////////////////////////////////////////////////////////////
void *thread_PROC(void*){
//Variable instantiation
//Kalman filter variables - used by PROC_thread
pthread_mutex_lock(&procfile_mutex);
pthread_mutex_lock(&procrawfile_mutex);
{ //Post-processing loop
    //wait for mutex lock to read any files

    while (gStop > 0){ usleep(1000); } //wait to start
    //post-processing variables
    double INSprevT = -1.0;
    double GPSprevT = -1.0;
    double BARprevT = -1.0;

    double INSdeltaT, GPSdeltaT, BARdeltaT;
    double aWNorm[3]; // This is "N, E, Up" - "+Y, -X, +Z" from accelerometer
    double prevLat, prevLong, curLat, curLong; //Lat == Northing, Long = Easting
    double GPSdispX = 0, GPSdispY = 0, GPSvelX = 0, GPSvelY = 0;

    while(gStop < 0){
        // pthread_mutex_lock(&insfile_mutex);
        INSdeltaT = INSrecT - INSprevT;
        if(INSdeltaT > 0){
            if(INSprevT != -1.0){
                //Process raw acceleration data
                aWNorm[0] = ((double)INSrecX/(double)0x7FFF)*4*9.81;
                aWNorm[1] = ((double)INSrecY/(double)0x7FFF)*4*9.81;
                aWNorm[2] = ((double)INSrecZ/(double)0x7FFF)*4*9.81;
                //Save raw acceleration data to file
                ssBuffer1 << "                    INS: @" << setprecision(12) << INSrecT << " s, " <<
                    "N: " << aWNorm[0] << " E: " << aWNorm[1] << " Up: " << aWNorm[2] << endl;
                ssBuffer2 << "@" << setprecision(8) << ACLrawT << " s,"
                    << " aX: " << ACLrawX << " aY: " << ACLrawY << " aZ: " << ACLrawZ
                    << " gX: " << GYRrawX << " gY: " << GYRrawY << " gZ: " << GYRrawZ << endl;
            }
            INSprevT = INSrecT;
        }

        // pthread_mutex_unlock(&insfile_mutex);

        // pthread_mutex_lock(&gpsfile_mutex);
        GPSdeltaT = GPSrecT - GPSprevT;
        if(GPSdeltaT > 0){
            //Process raw GPS data
            curLat = GPSrecN;
            curLong = GPSrecE;
        if(GPSprevT == -1.0){ //if first non-zero data point
            prevLat = GPSrecN;
```

```
                prevLong = GPSrecE;
            } else {
                //Calculate X and Y displacement (meters):
                GPSdispX = gpsDisplacement(&prevLat, &prevLong, &prevLat, &curLong); //E-W displacement
                if(curLong < prevLong)
                    GPSdispX *= -1;
                GPSdispY = gpsDisplacement(&prevLat, &prevLong, &curLat, &prevLong); //N-S displacement
                if(curLat < prevLat)
                    GPSdispY *= -1;
                GPSvelX = GPSdispX / GPSdeltaT;
                GPSvelY = GPSdispY / GPSdeltaT;
                //update previous lat/long values:
                prevLat = curLat;
                prevLong = curLong;
                //Save raw GPS data to file
                ssBuffer1 << "******************GPS: @" << setprecision(12) << GPSrecT << " s, " <<
                    "N: " << curLat << " E: " << curLong <<
                    " dispX: " << GPSdispX << " dispY: " << GPSdispY <<
                    " velX: " << GPSvelX << " velY: " << GPSvelY << endl;
            }
            GPSprevT = GPSrecT;
        }
        // pthread_mutex_unlock(&gpsfile_mutex);
        // pthread_mutex_lock(&barfile_mutex);
        BARdeltaT = BARrecT - BARprevT;
            if(BARdeltaT > 0){
                if(BARprevT != -1.0){
                //Process raw altitude data
                //Save raw altitude data to file
                ssBuffer1 << "**               **BAR: @" << setprecision(12) << BARrecT << " s, " <<
                    "A: " << setprecision(7) << BARrecA << " T: " << BARrecC << endl;
            }
            BARprevT = BARrecT;
        }
        // pthread_mutex_unlock(&barfile_mutex);

        usleep(100); //delay to lower CPU utilization
        }
    }
    pthread_mutex_unlock(&procfile_mutex);
    pthread_mutex_unlock(&procrawfile_mutex);
    gStop++;
    pthread_exit(0);
    }

    void magInit(){
        MPU6050 mpu;
        mpu.initialize();
        I2Cdev::writeByte(0x68, 0x37, 0x02); //set i2c bypass enable pin to true to access magnetometer
    }

    void getMag(int16_t* mx, int16_t* my, int16_t* mz) {
        // Set single measurement mode
        I2Cdev::writeByte(0x0C, 0x0A, 0x01); //write 0x01 to CNTL register
        // Check Data Ready or not by any of the following method.
        uint8_t DRDY = 0;
        while(!DRDY){
            I2Cdev::readByte(0x0C, 0x02, &DRDY);
        }
        readMagData(mx, my, mz);
```

```
}

void readMagData(int16_t* mx, int16_t* my, int16_t* mz){
    uint8_t buffer[6];
    I2Cdev::readBytes(0x0C, 0x03, 6, buffer);
    *mx = (((int16_t)buffer[1]) << 8) | buffer[0];
    *my = (((int16_t)buffer[3]) << 8) | buffer[2];
    *mz = (((int16_t)buffer[5]) << 8) | buffer[4];
}

int selfTest(int16_t* mx, int16_t* my, int16_t* mz) {
    // (1) Set Power-down mode
    I2Cdev::writeByte(0x0C, 0x0A, 0x00); //write 0x00 to CNTL register
    // (2) Write ���1��� to SELF bit of ASTC register
    I2Cdev::writeByte(0x0C, 0x0C, 0x40);
    // (3) Set Self-test Mode
    I2Cdev::writeByte(0x0C, 0x0A, 0x08);
    // (4) Check Data Ready or not by any of the following method.
    uint8_t DRDY = 0;
    while(!DRDY){
        I2Cdev::readByte(0x0C, 0x02, &DRDY);
    }
    //  When Data Ready, proceed to the next step.
    // (5) Read measurement data (HXL to HZH)
    readMagData(mx, my, mz);
    // (6) Write ���0��� to SELF bit of ASTC register
    I2Cdev::writeByte(0x0C, 0x0C, 0x00);

    //check self-test pass criteria (return 0 if fail):
    if( *mx < -100  || *mx >  100 ||
        *my < -100  || *my >  100 ||
        *mz < -1000 || *mz > -300)
        return 0;
    else
        return 1;
    }

    int checkNorth(int16_t* mx, int16_t* my, int16_t* mz) {
    float magnitude, normX, normY, normZ;
    magnitude = sqrt(pow(*mx, 2) + pow(*my, 2) + pow(*mz, 2));
    normX = *mx / magnitude;
    normY = *my / magnitude;
    normZ = *mz / magnitude;
    if( normX < 0.70 && normX > 0.55 &&
        normY < 0.85 && normY > 0.70 &&
        normZ < 0.10 && normZ > -0.10)
        return 1;
    else
        return 0;
}

double gpsDisplacement(double *lat1, double *long1, double *lat2, double *long2){

    double PI = 4.0*atan(1.0);

    double dlat1=*lat1*(PI/180);
    double dlong1=*long1*(PI/180);
    double dlat2=*lat2*(PI/180);
    double dlong2=*long2*(PI/180);
    double dLong=dlong1-dlong2;
    double dLat=dlat1-dlat2;

    double aHarv= pow(sin(dLat/2.0),2.0)+cos(dlat1)*cos(dlat2)*pow(sin(dLong/2),2);
    double cHarv=2*atan2(sqrt(aHarv),sqrt(1.0-aHarv));

    const double earth=6371009; //mean Earth radius (meters)
    double distance=earth*cHarv;

    return distance;
}
```