Report on:
GPS NAVIGATION IMPLEMENTED IN PYTHON

Submitted to:
Dr. John Scales
Colorado School of Mines

Prepared By:

Nathan Neibauer
Everett Hildenbrandt
Alexandra Nilles
Kento Okamoto
Matt Jones

December 4, 2012

Contents

1 Introduction

1.1 Project Description and Specifications

This report will provide an overview of a GPS navigation project. The purpose of this project is to explore the functionality of a portable GPS unit. Our project was coded in Python, and this report will describe the coding methods used, as well as summarize our results.

The overall project goal is to write a Python program which will interact with a serial GPS device. The device will connect to a computer capable of receiving the data. Our application will then store the data and analyze variation in latitude, longitude, and elevation as functions of other relevant data (such as the number of satellites, time, etc.). In addition to studying variation in the data, our application will also plot elevation against time at a stationary position in order to analyze any fluctuations in the data. Also, if the user moves the GPS device in space, the program will upload a path to Google Maps representing the position data for that journey. Overall, the code in Python is to be clean, literate, and self-explanatory. The end product is to be released under General Public License or the MIT license as open source [1].

1.2 Design Solution

To begin, the Python code was divided into discrete subsystems that perform the required tasks. One subsystem of code manages the serial connection, the second studies variations and performs error analysis, the third creates a class which organizes and performs operations on the data, the fourth saves and opens files, creating files which can be uploaded to Google Maps, and the fifth provides a convenient interface to use the program. In Figure 1, we can see that there are many different functions for the user to choose from, represented by buttons. Data can be saved, deleted, opened, trimmed, or uploaded to Google Maps. Overall, our design meets all requirements efficiently and effectively.

1.3 Overall Design Flow

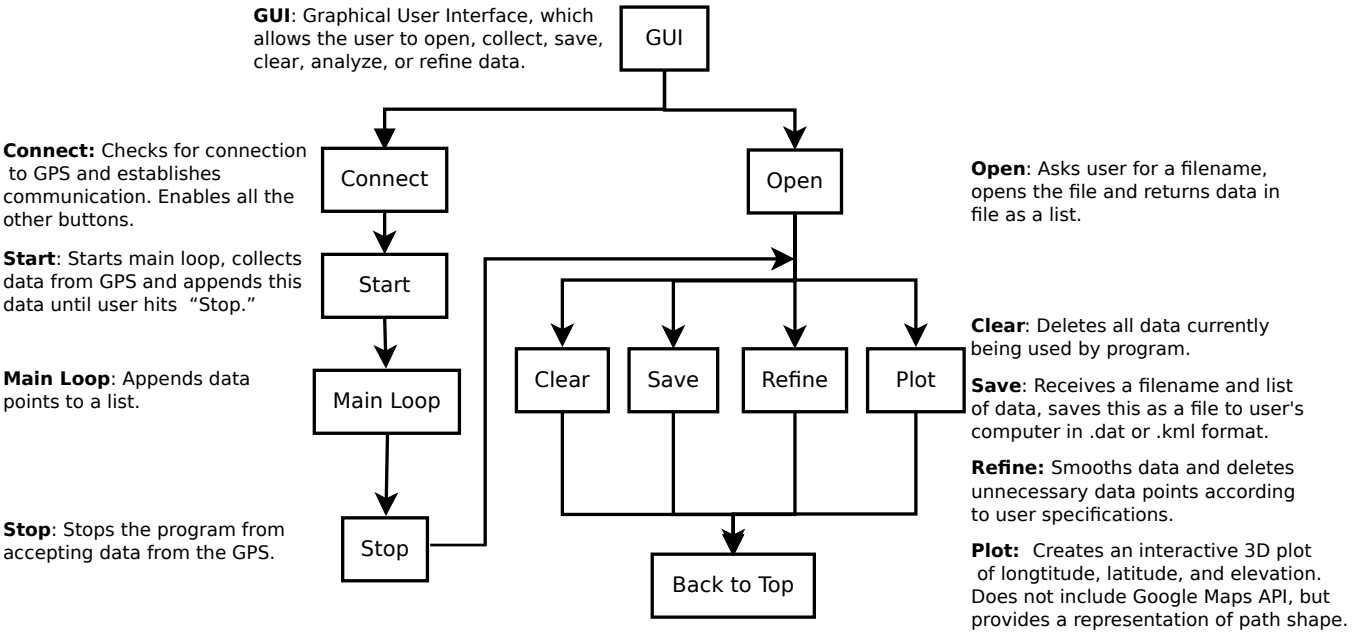

Figure 1: Basic Operations Flowchart

2 GUI

What is the GUI?

Any computer program is broken into two parts: the part seen and used by the user and the behind-the-scenes programming part. In a well-written program, the user will never interact with the code itself, but rather a high-level interface that separates the user from the code. More specifically, this is called a *graphical user interface*, or *GUI*. In our program, the GUI not only acts as the user-code interface, but also acts as the central hub of all behind-the-scenes program operations. Most functions of our program are called and excecuted by the GUI, even though the user may not ever see the interaction.

How does it work?

The GUI works like a switchboard. The user has several options of buttons to click and parameters to specify, and the GUI calls on functions from within the modules of our other subsystems. For example, a REFINE button will call on a *refine* function from another section of code in order to refine the current data. Every button we have has two states of operation, *active* and *disabled*. When a button is *active* it is clickable and usable - when *disabled* it is not. The GUI has its own intuitive control system that enables or disables buttons based on the current state of the entire system (for an example, see Figure 2, which demonstrates that when the GPS is not connected, the user can only connect a GPS or open a data file).

Additionally, the GUI has text boxes for displaying messages such as the current state of the GPS, the current data being extracted, and error messages. This display system acts to keep the entire program encompassed on one display so the user does not need to open up additional windows and dialogs to successfully and efficiently run the code. Specific operating instructions are included in Appendix 1.

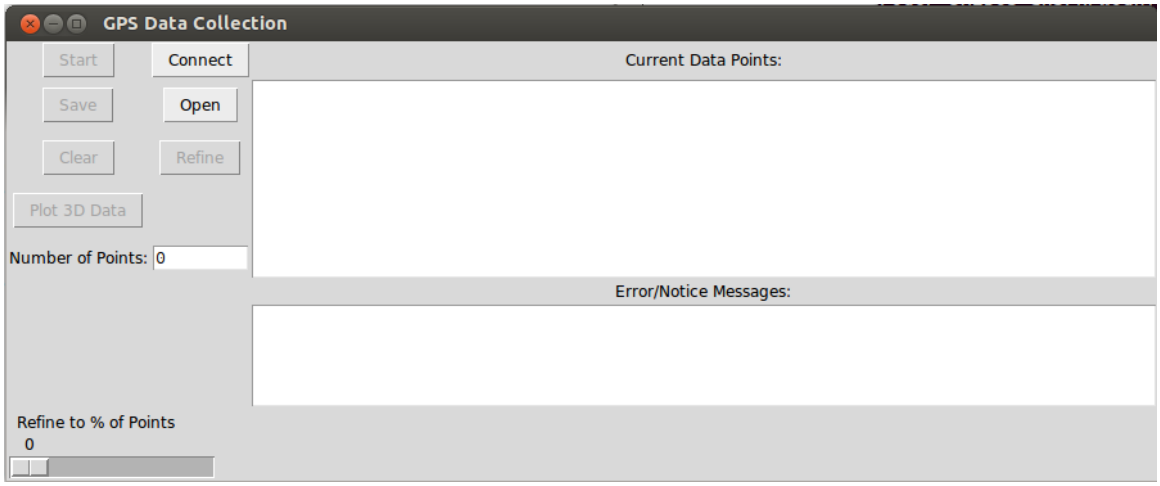

Figure 2: The GUI when the user first opens the program

2.1 Alternatives

Why use a GUI?

There are a plethora of ways to create a user-interface for our application. Because our final goal is to upload this code as an open-source script, the primary concerns are user-friendliness and adaptability. If our code was to be exclusively used by ourselves or the general programming-literate demographic, an efficient way of interacting with the code is via excecution of specific commands within the python shell. This method, however, is extremely unintuitive for those who have never been exposed to the concept of programming. A graphical interface, on the other hand, is intuitive to nearly everyone. Even people with little to no technical background can understand how to move a mouse around and click on buttons, given the proper instructions and knowledge on what the buttons do. Granted, an arbitrary person is not going to search online for a python script able to extract data from the GPS dongle he/she just bought; however, we coded the GUI in a user-friendly manner all the same.

Additionally, our code must be fully adaptable and extensible to other applications. As a future member of the open-source software community, our code is useful to a GPS-loving Python-wiz (Eugene) who would like to add excessive levels of customization and endless functionality. Our GUI is written using concepts of object-oriented programming (outlined below) as opposed to functional programming, so that Eugene can easily implement as many additional buttons as his heart desires [2]. As he implements his own code, our program will gracefully accept his code without completely barfing.

2.2 The 'Guts'

This section is very code-oriented and technical

The GUI is written as its own class (entitled *App*), utilizing its own local variables and object methods that we defined. The GUI class also creates local instances of external classes from within our other subsystems. Besides the standard *init* method, there is a specific method directly associated with each button, as well as methods for clearing text and enabling/disabling buttons. The details are outlined below.

init

This method is automatically called when an instance of the *App* class is called, and it creates all variables necessary to run the GUI. These variables include boolean values that determine the state of each button, local instances of the classes *serObject* and *GPGGA* (see section 4), and initializations of every button and text box we use. This section also defines our visual layout of the GUI itself [3].

setState

This method defines the current state of each button as *active* or *disabled* based on different conditions and using boolean logic. Every time a button is pressed, the *setState* function is automatically called to set the states of each button. The final button states are then determined via the total boolean expression of all conditions. There are 4 conditions that this method checks [4]:

1. **Is the GPS connected?** Via the *serObject* class, this condition checks whether the computer is actively communicating with the GPS or not. If the GPS is not connected, all buttons except CONNECT and OPEN are disabled. This is very intuitive because the user shouldn't be able to extract data from a GPS that isn't connected. If the GPS is connected, then all buttons become enabled (providing the following conditions hold) except CONNECT. If the GPS gets disconnected at any point, the program will recognize this event and set the buttons accordingly.

2. **Are we collecting data?** This condition uses a boolean value to check whether or the main loop (detailed below) is running. If the loop is running, the program is storing data from the GPS and all buttons are disabled except STOP. Naturally, we don't want any operations to be performed on a dynamic data set because it could lead to data contamination. Any manipulation and saving operations must be performed on a static data set.

3. **Do we have any data?** This condition checks to see if any data is currently loaded into memory. If not, then the REFINE, SAVE, and PLOT buttons are disabled. Again, this is very intuitive, as it does not make sense to refine, plot, or save a data set consisting of nothing.

4. **How long is the data?** This condition very simply checks to see if we have more than 50 data points. If we don't have more than 50, then the REFINE button is disabled. We added this because it does not make sense to refine a data set to below 50 points, especially with data sets upwards of 50,000 points.

start

The first of our buttons, the method called by this button will start the data collection process within a while loop. Every iteration of the loop will first check the incoming data point for errors. If an error is found with said point, the GUI will print that error in the text box. If an error is not found, the legitimate data point is appended to the current data set. This process will continue indefinately until the STOP button is pressed. See Figure 3 for an example of what the GUI looks like when the START button is pressed.

stop

This method stops the loop process described just above in the **start** section. It is associated with the STOP button

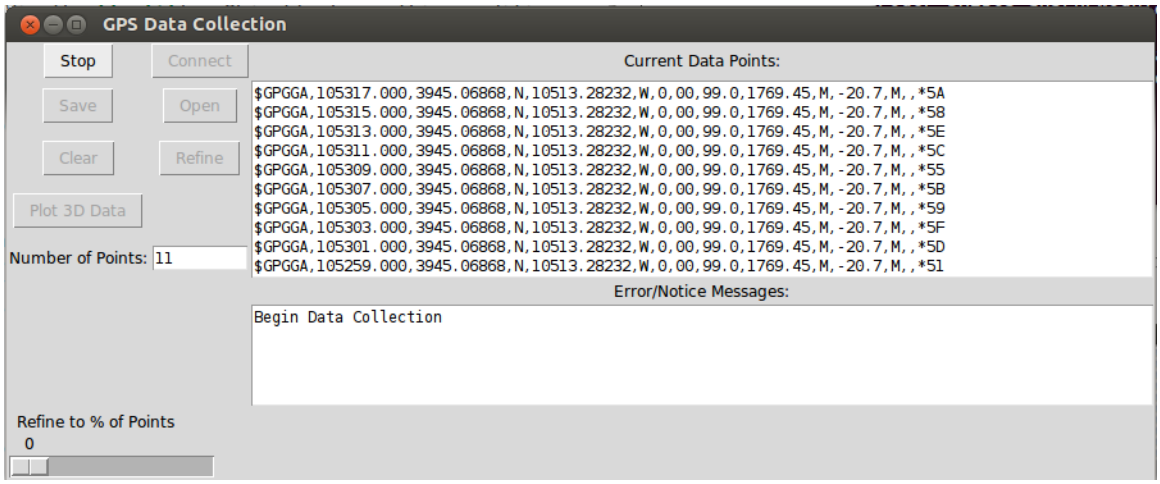

Figure 3: The GUI collecting data and displaying it

openFile

This method also calls on the *tkFileDialog* module, but this time it opens up an 'open file' dialog. Our program only supports opening .dat files. This filename is then passed to a function that opens the file itself (again see section 4). If there is a problem opening the file (as in the data is in the wrong format, etc.), an error message is printed in the text box. If the file successfully opens, the data from within the file is appended the the current data set. This method is associated with the OPEN button.

saveData

This method calls on the *tkFileDialog* module, which opens up a standard 'save file' dialog (see Figure 4 for what this looks like). We have limited the data types to .dat and .kml because those are the only two data types compatible with the program. Once the filename is specified, it is then passed to a function that writes all of the current data to the file (see section 5). If all goes well, a nice message tells the user that the file was saved successfully. If not, the user receives an error message. This method is associated with the SAVE button.

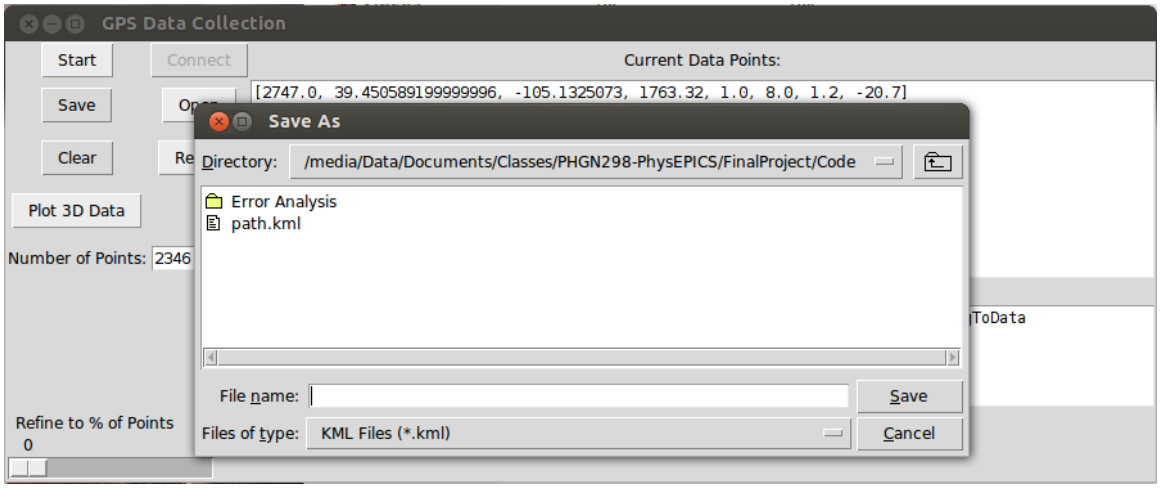

Figure 4: Save Dialog in Linux

refineData

This method passes the value from *setRefine* above to the *minimize* function (see section 4), which basically trims the data down to a reasonable size. If there is an error in this process, the appropriate error message is printed.

plotData

This method takes the current data and generates 3D plot by calling the *plot3D* function (see figure 5 in Section 4.2). It is associated with the PLOT button.

connect

This method runs through a 100 iteration for loop of the function *connect* (see 4). This basically checks all the serial ports until the computer locates the GPS, and then this method will initialize communication with the GPS. It is associated with the CONNECT button.

3 Serial Input

The purpose of this subsystem is to have the GPS device communicate with the user's computer. Without the serial input module, we cannot get the location data the GPS dongle is trying to send us. Without this data, we do not have the information we need to record our path when we run the application.

3.1 Program Requirements

The GPS dongle that was provided for this project was the Delorme Earthmate GPS LT-20 [5]. The device connects to the computer using a USB cable. Because the device uses a USB port, the Python program must first identify the USB ports mounted on the computer. These specifications differ from each computer as well as operating system the computer uses such as Windows, Linux or Mac. For our application, the only supported operating system is Linux.

3.2 Communication Over the Serial Port

Serial Connection

The serial input object, *serObject*, is instantiated at the application's launch. The object connects to the GPS unit when the user, through the Graphical User Interface, clicks the CONNECT button. The source, *pyserial* [6], is called at the beginning of the module to help identify what devices are connected to the computer. Once *pyserial* has been initiated, it determines the path of every port on the computer and checks to see if a GPS unit is connected to one of these ports. Once the GPS is located, it will then begin taking in data from the device as it sends data over the serial port to the computer.

Collecting Data

The *serObject* class also contains support for inputting a data point from the Serial buffer. When called by the main program (while in data collection mode), the object trys to import a data point. If it's unsuccessful, an error is returned for the main GUI to display; if successful, the data point is checked for validity and returned to the main program.

4 Data Storage and Manipulation

Our application handles and manipulates a large dynamic dataset; to make it possible to handle this data in real time, an efficient data-handling class (or classes) is needed. We designed and coded two classes to meet these needs - GPSDataSet and GPGGA. The first, GPSDataSet, is generic and robust enough to be used in other applications by other developers (our project is open-source). The second class we coded, GPGGA, inherits much of the functionality of GPSDataSet while extending some functionality to our specific needs.

4.1 Data Storage Class - GPSDataSet

The GPSDataSet class is used to store a dynamic list of data points as strings. The operations supported include adding elements, adding lists of elements, and removing elements with a selective filter. We programmed this class for generic data storage, but also so that it can be easily extended for others to use.

Instantiating the GPSDataSet Object

The first thing an application using the GPSDataSet class will do is instantiate an object of the class. This will create an empty list for data storage, an integer specifying the length of the list (set to zero initially), and set a few internal variables specific to the class. The object can (optionally) be instantiated with an initial data set that the initializing function will check for validity, then use the *addList* method to add the entire data set to the data storage list.

Adding Elements

Now that a GPSDataSet object has been created, the application using it needs a way to add elements to its data set. To do this, the *addElement* method is called, which will place the new data point at the end of the data list after verifying it is in the correct form. The simplest way to add an element to a list in Python is to append the element to the end - however this is a memory and time inefficient process because of the way Python appends elements to lists. Specifying an initial list length and only allowing that many data points to be stored at any one time is another option, but no one can be certain ahead of time how many data points they'll need. Instead, we designed the data storage as a hybrid of the two. Everytime *addElement* is called, it checks whether the data list is long enough and extends it by a large number of empty elements if not - the default being 200 elements. If the list already has these empty values at the end, it overwrites them with the new data element. Because it only extends the list every 200 times it needs to add a data point, the application using our module can avoid most of the inefficiency generated by using Python's append function for every element. If any errors occur while trying to add a data element to the data set, the error is returned to inform the user.

Add Lists

Lists of data (as opposed to individual points) can be added by calling the *addList* method. This method simply adds each element of the list term-by-term to the data set using the already defined *addElement* method.

Removing Elements

Our GPSDataSet module has two ways to filter the data to a subset of the current points. The simpler method of the two, *trim*, simply accepts parameters specifying which data points to keep and discards the rest. It is included for completeness so that this class can have even more adaptability for use in the open source community, but does not otherwise serve a purpose for us. We do, however, have use for the *filterSet* function, which accepts a filtering parameter and removes all the data that does not match that filtering parameter. For example, in the generic class GPSDataSet, which only stores string data value, it checks the filtering parameter against the first characters of each data element - matches mean that data element is kept. In applications that other developers write using this data structure, it's expected that this method will be modified to fit their specific needs.

4.2 Our Data Manipulation Class - GPGGA

For our application, we needed to extend the functionality of the data storage class, GPS-DataSet, to support conversion of the data points into lists of numbers describing each GPS data point, as well as minimizing the points used to represent the path and basic plotting. While this class worked very well for our application, it would be less likely than the more generic GPSDataSet to work for other developers needs - it's more specific to our exact needs.

Class Hierarchy

Because the functionality of the GPSDataSet class (see section 4.1) is still useful to us, we decided to program this class, GPGGA, as a subclass of that class. This means that all of the methods from GPSDataSet will be available to our class, and that we will be able to overload (over-ride) some of the methods that we want to perform a slightly different function. This saves code, and the concept of Object Oriented Programming greatly simplifies the use and reuse by third parties of written code [7].

Data Conversion

The data from that our application collects via the serial port (see Section 3) or through opening a file (see Section 5) will be coming in as string values. Because Python strings cannot be used directly for calculations, our class GPGGA converts these string values into usable numerical values with a method called *stringToData*. The method checks whether the string is a valid data point, and if it is, exctracts the data values from it and saves them in the data set.

Minimizing Points

When the dataset gets large, it will become less efficient to store the entire data set, which could possibly be represented accurately with a smaller subset of the data. For example, if the user travels in a straight line for a period of time, it's less important to keep the data points along the line than to keep the points at the end of the line; for a traveller moving in a circle it would be necessary to keep a larger selection of points. To find out which points are more important, we numerically analyzed the curvature of the path at each individual

point. Finding the curvature of discrete data is no easy feat - fortunately the Python SciPy library has some useful tools which simplify it.

The formula for curvature is [8]:

$$\kappa = \frac{|\vec{r}' \times \vec{r}''|}{|\vec{r}'|^3} \tag{1}$$

where $\vec{r}$ is a vector and κ is the curvature. However, this formula applies only to continuous functions, so we came up with a continuous representation of our data. SciPy has a package called interpolate which allows for cubic spline representations of discrete data, as well as smoothing of the data before creating the representation of it (which helps eliminate noise in the signal). SciPy also offers the functionality of evaluating this continuous function or its derivatives at any points, which we used to calculate the curvature at each data point [9].

To select only the points of highest curvature for keeping, a priority queue is created. This queue stores the position in the data set of the highest curvature point as the first element of the queue, then the next highest as the next element in the queue, all the way down to the lowest curvature element. Then, only the specified number of points (a number specified as a percent in the GUI by the user, see section 2), are taken off the queue and saved as the new data set.

To demonstrate this functionality working, see Figure 5 and Figure 6 below. The first figure, 5, shows a dataset collected over a path through Golden, CO and up Lookout Mountain. After refining (shown by the number of points being less in the second figure, 6), the plot still looks very similar, yet takes up less space in memory because it only stored 23% of the original data points. The path is distorted somewhat, but by only trimming out 50% of the points, we still get better memory efficiency yet a good representation of the path.

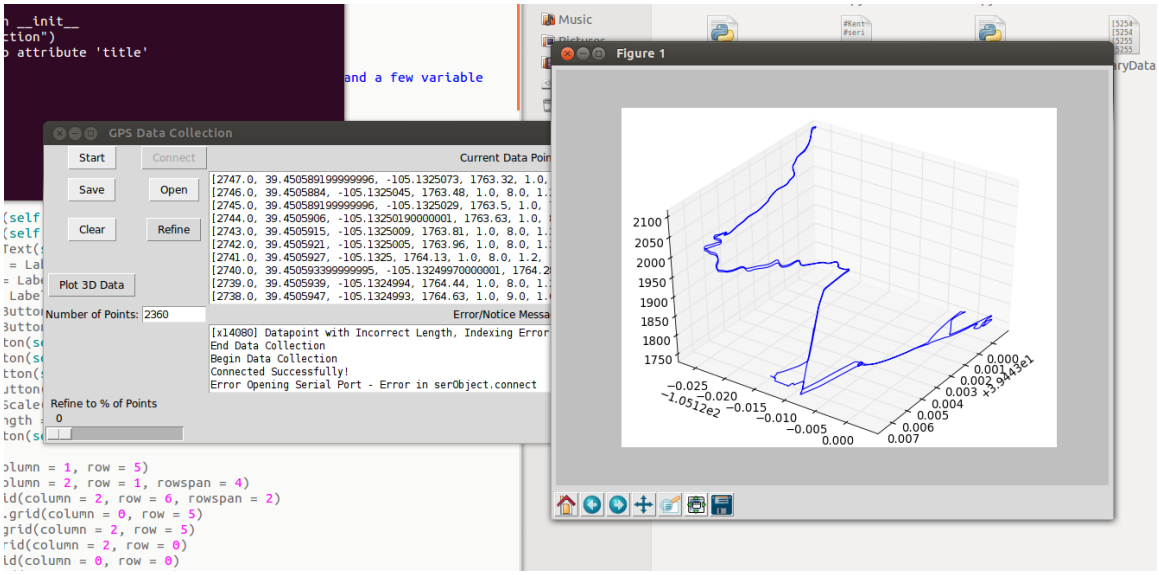

Figure 5: Original Data Collected Around Golden, CO

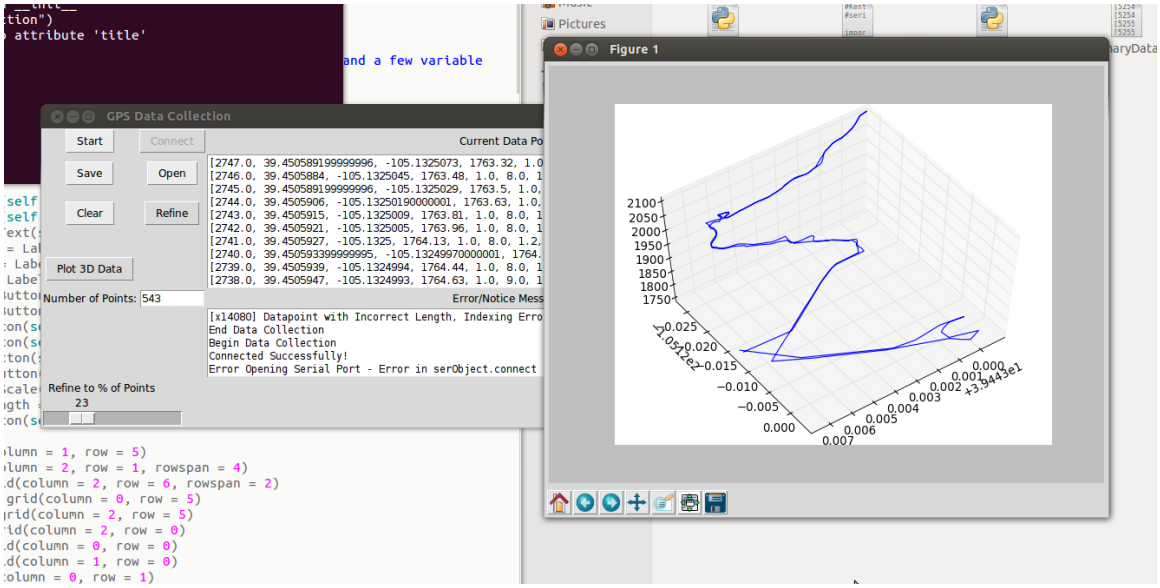

Figure 6: Representation of Data After Refinement

Plotting

To ensure that the data being collected seems to fit the path the user is travelling, we have included a "Plot 3D Data" button in the GUI. When this button is pressed, it calls the *plot3d* method in the GPGGA class, which plots a line-connnected representation of the data in 3D. An example of this method being called is actually shown above in the **Refine** section, see Figure 5 above.

5 File Operations

The client has specified that our program needs to upload the collected GPS data to Google Maps, displaying the path taken by the user [1]. In order to accomplish this task, our program must have several capabilities: it must be able to open and save data files, and it needs to be able to convert GPS data into a form recognizable by Google Maps. This subsystem contains one function that can open a data file (extension .dat) which may was previously saved to the user's computer. A second function is the ability to save a data file in a file type of the user's choice (.dat or .kml). In order to upload a path to Google Maps, the user must save the file in Keyhole Markup Language (KML) format. KML is a file format designed specifically for Google Maps and Google Earth. When a KML file is opened in Google Maps or Earth, the browser displays whatever graphic is specified by the KML file [10], which in this case is a path, constructed from a list of data points.

5.1 Alternative Designs

An alternative method for plotting paths, instead of using KML files, is the Static Maps Application Programming Interface (API) [11]. A major limitation of the Static Maps API is that it can only plot a path with approximately 200 (latitude, longtitude) points. Each data point is converted to a string of ASCII characters, which is appended to the URL. The data size limit is thus due to the set character limit of URLs [12]. For complicated paths involving lots of turns, or paths taken over a long period of time, the Static Maps method would not provide a useful output, thus KML was chosen for our design.

5.2 Module Functionality

Opening and Saving Files

When the user chooses to open a file through the Graphical User Interface (see Section 2), the GUI prompts the user for a filename. Our program then opens the file, reads the data one line at a time, and returns a list of the raw data. When a file is saved, the user selects a data set to save and chooses a filename. If the user chooses to save the file in KML format, the program converts the data into the correct format. See Figure 4 for an example of what the saving and opening dialogs look like.

Creating KML Files

The GPS device transmits data in the NMEA format [13]:

$GPGGA,001138.000,3944.8184,N,10513.6361,W,1,06,1.6,1826.5,M,-20.7,M,,0000*5F

This data contains the latitude, longitude, and elevation data which we need to create a path in KML. The method "kmlForm" in the GPGGA class (see Section 4) converts a list of NMEA data points into a list of position data points. The latitude, longitude, and elevation numbers are separated by commas, and each data point is separated by a line break.

By writing a template KML file [14] containing no position data, we are able to create new KML files by simply inserting the list of data points into the template. The KML file defines the color and width of the line, and sets the altitude mode for the path, which can be

measured from sea level or from the surface of the Earth at the given latitude and longitude [10]. Our GPS data is measured from sea level (in meters) [13]. Our latitude and longitude data is returned with the values for degrees and minutes before the decimal place, whereas KML requires only the degrees to be before the decimal. Therefore, our program divides the latitude and longitude data by 100 and then adds our data into the premade KML template. This template is then saved under a name given by the user with the .kml file extension.

Displaying a Path

In order to view the path, the user must upload this KML file to either Google Maps or Google Earth. To view the path in Google Maps (inside an internet browser), the user must be signed into a Google account. From the Google Maps homepage, the user should click on "My Places," then "Create Map," then "Import." Google Maps will then open a dialog box where the user can select the previously created KML file. When uploaded, this file will overlay the path taken by the GPS device onto a normal Google Maps interface, so the user can zoom in and out and move around the map [15].

To view the path in Google Earth, the user must first download the Google Earth program to his or her machine [16]. Once Google Earth is up and running, the user simply needs to click on the "File" option in the upper right hand corner of the Google Earth window, then click "Open," and select the KML file that the program generated [17]. Google Earth will then display the path with the added benefit of a three-dimensional view, which allows the user to view altitude data. Altitude data is not supported in Google Maps.

5.3 Results

The KML conversion program has been extensively tested. For example, several thousand data points were collected during a drive around Golden, CO, and the data was converted into a KML file. The resulting file was then uploaded to Google Maps, and it can be seen in Figure 7:

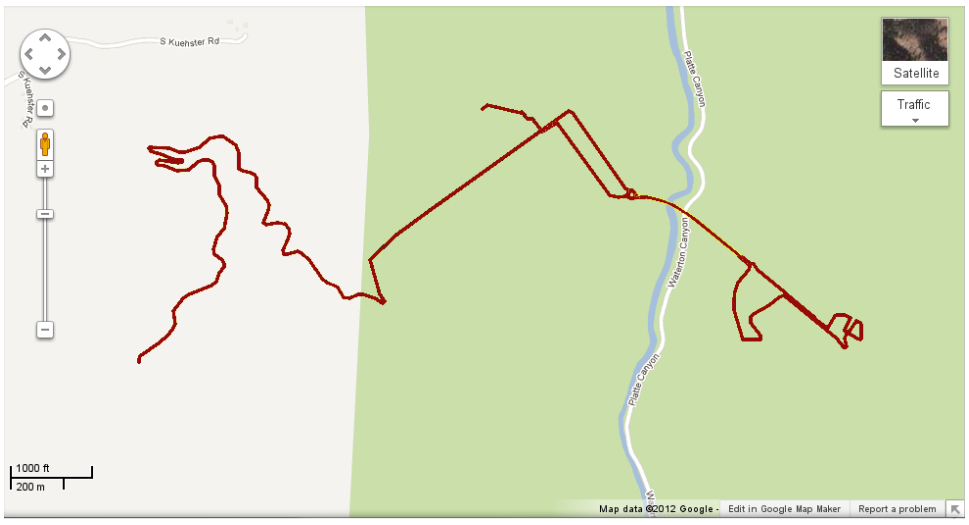

Figure 7: Path from GPS Data

This path is located a considerable distance south of Golden, but this is due to systematic error inherent in the GPS device (see Section 6). We confirmed that the shape of the path generated was correct by comparing the path our program collected with a path generated by a GPS App on an Android smartphone. The smartphone path can be seen in Figure 8. Since the shapes of the paths match, we can verify that the relative positions of the GPS data points were correct. Comparing the two paths shows an advantage of our design: the path generated on the smartphone has several "holes" in the data where the phone lost reception, whereas the path generated by our GPS device is continuous due to constant communication with satellites.

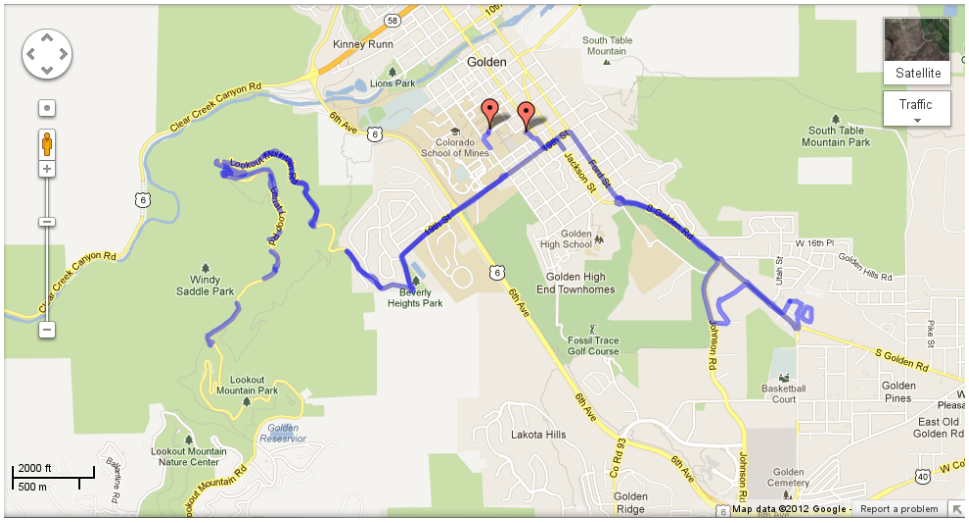

Figure 8: Path from Android phone GPS

6 Error Analysis

The error analysis program is designed to evaluate the devices performance. The client will not use this subsystem extensively. The purpose of this subsystem is to allow the team to estimate various systematic errors in the data that the device returns. Some aspects of the code, like its plotting functions, will see some use by the client; however, the error analysis program acts independently of the other functions in the project code.

6.1 Description of Program

There are often many sources for error in measurements. Systematic error is error inherent to the process by which data is collected. Chance error is the resulting error from uncontrollable conditions. In the case of the GPS device, chance error might be the result of the atmospheric conditions at the time data was collected. Chance errors can be reduced by taking a larger sample of data [18]. In order to determine what systematic errors exist, the team places the device at a predetermined site with coordinates provided by the USGS and records position and time information. A large data set is compiled, minimizing chance error. Analysis of the large data set may produce a consistent error when compared with the USGS coordinates for the site. The consistent error would be systematic error. In general, the error analysis program is designed to accept a large set of data. The program also expects the data to appear in a particular format, for example:

[UTC of Position (0), Latitude (1), N or S (2), Longitude (3), E or W (4), GPS Quality (5), Number of Satellites (6), Horizontal Dilution of Position (7), Antenna Altitude (8), Meters (9), Geoidal Separation (10), Meters (11), Age in Seconds (12), Diff. Reference Station (13), Checksum (14)]

Where UTC of Position represents time; N, S, E, and W represent cardinal directions, and so forth. The program performs a simple function to organize the data so that all of the latitude, longitude, elevation, time, and number-of-satellites information appear in independent lists. For example, list one might consist of only latitude coordinates, while list two might contain only longitude coordinates. In the python programming language, organizing the data this way allows the use of more powerful functions associated with lists and arrays. The error analysis program uses the latitude, longitude, and elevation coordinates to evaluate systematic errors by comparing them with the USGS coordinates for the predetermined site. The time and number-of-satellites information are useful in organizing the data based on when satellites are in range. The error analysis program computes the average of the latitude, longitude, and elevation coordinates and compares the averages with the USGS coordinates for the predetermined site by computing a relative error. The program also uses the large data set to compute the standard deviation of the mean (SDOM). The error analysis program breaks the large data set down into smaller subsets, computes the SDOM, and then plots the SDOM against the time information.

6.2 Alternative Designs

In designing the error analysis program, two error calculations are considered: accuracy and precision. In studying the accuracy of the measurements, the program simply compares an experimental data value to what is called the true value. In the case of the GPS device, the team calls the USGS predetermined site coordinates the true value. The team then compares the average data values from the GPS device to the USGS predetermined site coordinates. By making this comparison, the team can determine how well the data converges to the correct value. Alternatively, each value could be compared to the true or accepted value in the same fashion. However, comparing individual data points to a known point exacerbates situational or chance error [18]. Utilizing averages from a large data set also minimizes outlier error [19].

In discussing the calculation of precision, SDOM is the primary method. SDOM is a measure of precision in the data [20]. With a measure of the precision in the data set, the team can estimate how precise the data is. Consider a position that is not changing with time. It is expected that the position coordinate that the GPS device returns is the same coordinate each time; however, this is not the case. The GPS device returns coordinates that vary with time. By computing the SDOM of a data set and realizing that the data set is not to change with time, a measure of the precision is derived. The relative error calculation (comparing the average value of the set to the true value reported by the USGS) presents a measure of how accurate the data set is. The SDOM allows the team to determine how noisy the data is, while the relative error tells the team how far from the true value the data is. Another way to describe the error would be to test if it is periodic. By using a Fourier transform to plot the error on a frequency domain, new and interesting patterns might present themselves [21]. Several assumptions go into using the Fourier transform. One assumption is that the error in the data is oscillatory. Assuming the data is oscillatory is logical; however, this pattern likely presents itself regardless of the transform. If a Fourier transform is not used, some kind of periodicity in time appears if there is one. The Fourier transform requires some continuity in the functions it transforms [21]. Another assumption is that the error can be reasonably fit to a continuous function. Assuming the error does not fit a continuous function does not affect how the final plot appears when using the SDOM and relative error. Overall, using the standard error analysis techniques (SDOM and relative error) is safer. Using SDOM and relative error also possess the benefit of timeliness. These calculations are computed quickly whereas a Fourier transform onto a frequency domain might be more costly when the program is asked to analyze the data in a short period of time.

6.3 Testing

A typical data set is expected to contain as many as ten-thousand coordinate positions. In practice, the error analysis program returns cusps in data that is not naturally continuous. The discontinuous nature of the data that the GPS device returns generates cusps in the plots that the error analysis program generates. In testing with real data, the functionality of the program was expanded to perform plots of elevation, latitude, and longitude against the number of satellites connected. This allowed the team to estimate how the number of

satellites altered precision. The test data set contained 816 data points. Figure 9 shows how the elevation changes with time.

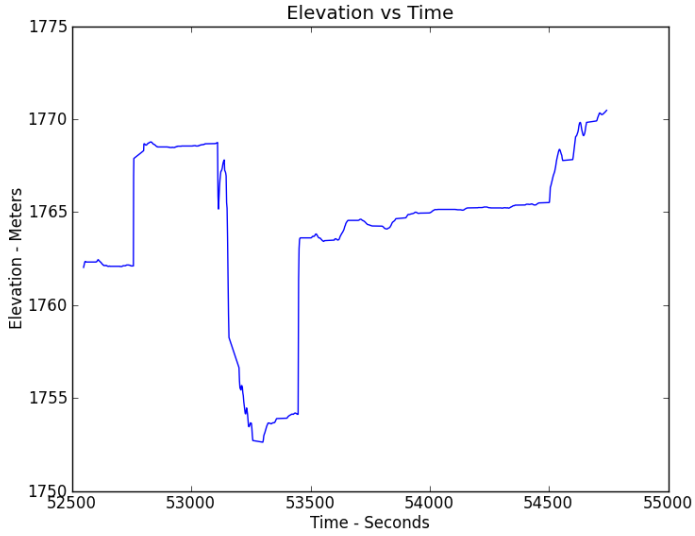

Figure 9: Elevation vs. Time

Figure 10 plots the latitude coordinate against the time coordinate. The longitude graph is almost identical in scale and behavior.

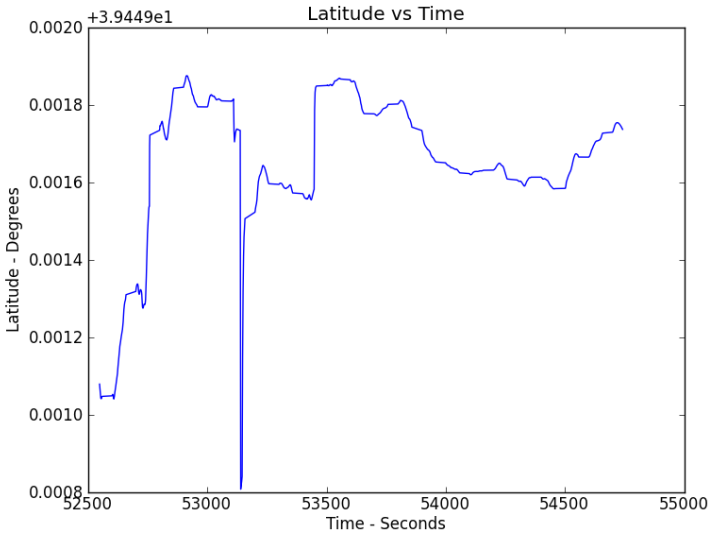

Figure 10: Latitude vs. Time

An interesting correlation is that as the number of satellites increased, the variation in the data decreased, and as the number of satellites decreased, the precision in the data decreased. There was also a correlation between how often the elevation changed and how many satellites the GPS device was connected to. With increasing numbers of satellites, the elevation jumped around more. This is likely because the increasing number of satellites over-defined the elevation coordinate. It is expected that with a number of satellites on the order of ten, the elevation coordinate would become more constant.

Figures 11 and 12 are plots of the standard deviation of the mean of longitude and elevation against the time coordinate. The latitude and longitude standard deviation of the means against time experienced extremely small precision, whereas the elevation coordinate experienced rather large precision comparatively. We can conclude that the latitude and longitude coordinates were more precise than the elevation coordinate.

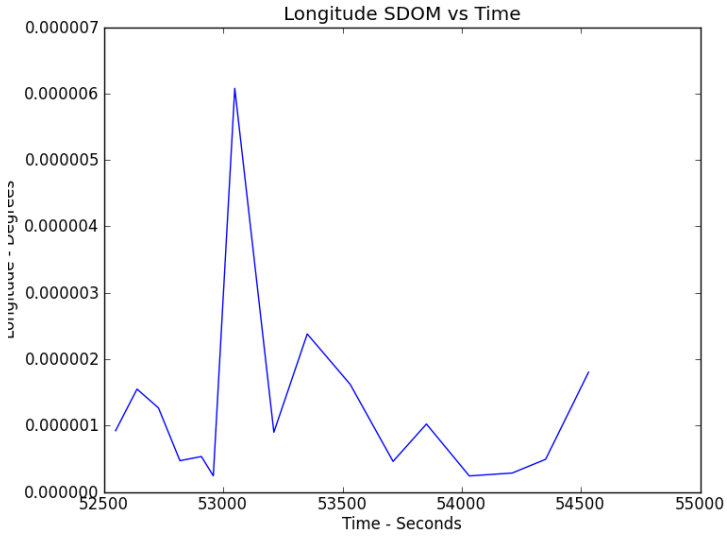

Figure 11: Standard Deviation of the Mean of Longitude vs. Time

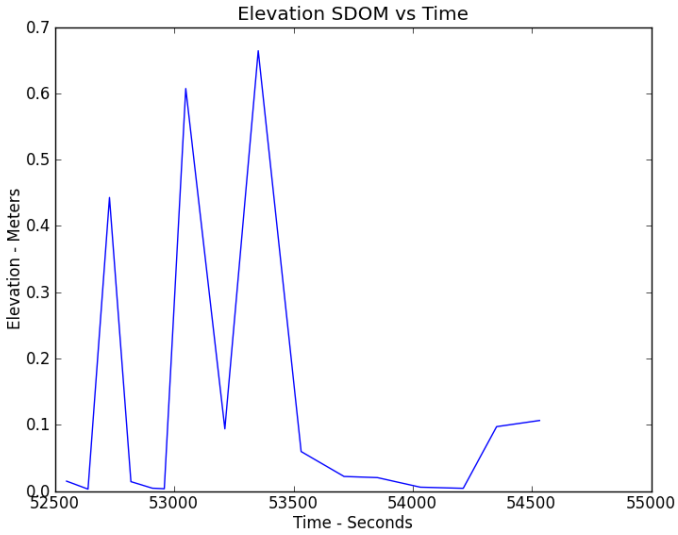

Figure 12: Standard Deviation of the Mean of Elevation vs. Time

6.4 Instructions

After the team has acquired the large data set for testing, the program performs its analysis by calculating SDOM and relative error. The error analysis program accepts a large data set as an argument, as well as the USGS predetermined site coordinates to compare the data set with. The error analysis program also accepts a step size to determine how to divide the large data set into smaller subsets. The team inputs a properly formatted numpy array containing the data sets to be analyzed. The program returns the plots and values that are relevant. It should be noted that the error analysis program is not for the use of the client. The error analysis program is designed to allow the team to evaluate the GPS device and to determine if the device is to be discarded and replaced.

7 Cost Analysis

The only software used in this subsystem is Python and the Google Maps API, both of which are free for non-commercial use. Thus, the only costs included are for labor, charged at $15.00 an hour, and overhead. See Table 1 for documentation and summation of all costs.

Table 1: Cost Analysis: Google Maps Subsystem

Item	Quantity	Unit	Unit Cost	Total Cost
Python	1	Software	$0.00	$0.00
Google API	1	Software	$0.00	$0.00
Labor	100	Hour	$15.00	$1500.00
Overhead	50 % of Labor			$750.00
Total				$2250.00

8 Conclusion

As we have demonstrated, our program successfully completes all required specifications. Data is imported from a GPS device, the accuracy of the position data is analyzed, and the data is smoothed and uploaded in path form to Google Maps. Our Python program creates a Graphical User Interface which makes the program user-friendly. Our code is efficient, utilizing classes to organize our program. We also used open source Python modules in order to streamline our code.

Moving forward, improvements could be made to the user interface, as well as to the accuracy of the data from the GPS. At this point, the GUI displays the raw NMEA codes. This could be improved by breaking up the data, and having individual displays for data such as position, number of satellites, etc. Additionally, the GPS device has inherent systematic error which we do not correct for. This results in the correctly shaped path being displayed in the wrong location in Google Maps.

We will release our code under an open source license, which will allow us to continually update our code, as well as allow other individuals to access, use, and improve our code. A possible extension for us and for others would be to utilize NMEA codes from the GPS Dongle other than the GPGGA position data.

9 Attributions

GUI and Program Integration - Nathan Neibauer
Data Storage/Manipulation and Program Integration - Everett Hildenbrandt
File Operations and Google Maps/Earth API Usage - Alexandra Nilles
Serial Input and GPS Communication - Kento Okamoto
Error Analysis and GPS Dongle Evaluation - Matt Jones

Appendices

A System Requirements

- Compatible with Linux OS only
- Modules *runGPSModule.py*, *GPSDataStructure.py*,*GPGGA.py*, *serialCOM*, and *fileOps.py* all within the same directory
- Python 2.7.3
- Module *time*
- Module *scipy* 0.9.0
- Module *numpy* 1.6.1
- Module *Tkinter* Revision: 81008
- Module *matplotlib* 1.1.1rc
- Module *tkFileDialog*

B Instructions for use

B.1 Opening the program

Open the Linux terminal and navigate to the directory with all the modules in it, as listed in appendix A. Once in this directory, type "python" into the terminal to initiate python. Now type "import runGPSModule" to run the program. If everything was done correctly, a GUI should pop up, as in Figure 2 in Section 2.

B.2 Connecting to the GPS:

Take the GPS dongle and plug it into any USB port. Some kind of indicator light on the GPS should start flashing to show that the GPS is receiving power. Now press CONNECT on the screen to begin communication with the GPS. A message on the screen will tell you whether or not the connection was successful.

B.3 Using the program:

Once the GPS is connected, there will now be an option to start collecting data. Note that data collection will only work if you are outside. Also, once the GPS is outside and able to connect to satellites, an indicator light should change from red to green on the dongle itself.

Collecting data

After START is pressed, data will be continuously collected and shown on the screen until STOP is pressed. There is an indicator showing how many data points have been collected.

Refining data

After you press stop, there are now several options on manipulating the data. You may specify a percentage of points with the slider across the bottom and then press REFINE in order to trim the data down while keeping the path the same. This function lowers the overall size of the dataset while keeping as accurate a representation of the path as possible.

Saving data

At any point in time while not collecting data, providing that you have at least collected *some* data, you may save it as a .dat or .kml file. Note that in order to re-open the file within the program, it *must* be saved as a .dat file; however, to upload the data to Google Earth, the data must be saved as a .kml file.

Plotting data

At any point in time while not collecting data, you may also plot the data to see what the path looks like by clicking the PLOT 3D button. This will pull up a nice 3D plot showing the path in longitude, latitude, and elevation coordinates.

Clearing data

At any point in time, you may press the CLEAR button to clear the current data set and start anew.

Opening data

This button is available before a GPS device is connected to allow for data manipulation of old data sets. It will pull up a dialog that lets you open a set of data (providing it is in the .dat form) and it will add it to the current data. Note that if used blindly, this can create discontinuities in the data set.

References

[1] J. Scales, "Client letter - GPS navigation." Colorado School of Mines, August 2012. jscales@mines.edu.

[2] M. Cornelius, "Tkinter file dialog." Online, October 2012. http://tkinter.unpythonic.net/wiki/tkFileDialog. Date Accessed: November 2, 2012.

[3] Internet, "The tkinter text widget." Online, September 2012. http://effbot.org/tkinterbook/text.htm. Date Accessed: November 2, 2012.

[4] F. Lundh, "An introduction to tkinter." Online, March 1999. http://www.pythonware.com/library/tkinter/introduction/. Date Accessed: November 2, 2012.

[5] DeLORME, "Earthmate gps lt-20." Hardware. http://www.delorme.com/library/hardware/earthmategpslt-20/Files/LT20_Long_Description.pdf.

[6] "Programming with pyUSB." Source Forge, 2012. http://pyusb.sourceforge.net/docs/1.0/tutorial.html. Date Accessed: November 1, 2012.

[7] M. Foord, "Introduction to OOP with python: Object oriented programming." Online, August 2011. http://www.upscale.utoronto.ca/PVB/Harrison/Vernier/Vernier.html. Date Accessed: November 20, 2010.

[8] "Calculus III notes: Curvature." Online, 2012. http://tutorial.math.lamar.edu/Classes/CalcIII/Curvature.aspx. Date Accessed: November 20, 2010.

[9] S. Community, "Numpy and scipy documentation: Interpolation." Online, 2009. http://docs.scipy.org/doc/scipy/reference/interpolate.html. Date Accessed: November 20, 2010.

[10] "KML tutorial," https://developers.google.com/kml/documentation/kml_tut. Date Accessed: October 15, 2012.

[11] "Static maps: API developers guide," https://developers.google.com/maps/documentation/staticmaps/. Date Accessed: October 20, 2012.

[12] "Maximum URL length," http://support.microsoft.com/kb/208427. Date Accessed: November 12, 2012.

[13] "Common NMEA sentence types." SatSleuth GPS Tracking Systems. http://www.satsleuth.com/GPS_NMEA_sentences.htm. Date Accessed: October 10, 2012.

[14] D. Taylor, "How do I create a KML map data file?," April 2009. http://www.askdavetaylor.com/how_to_create_make_kml_map_data_file.html. Date Accessed: November 20, 2012.

[15] B. Cornell, "Import your KML, KMZ, and geoRSS files." Google Maps LatLong Blog, November 2007. http://google-latlong.blogspot.com/2007/11/import-your-kml-kmz-and-georss-files.html. Date Accessed: November 20, 2012.

[16] "Download google earth." Google Software Downloads. http://www.google.com/earth/download/ge/agree.html. Date Accessed: November 25, 2012.

[17] "Import your KML data." Science Education Research Center, Carleton College, July 2012. http://serc.carleton.edu/eyesinthesky2/week10/intro_importing_data.html#import_kml_data. Date Accessed: November 20, 2012.

[18] M. Lincoln, *Think and Explain with Statistics*. Addison-Wesley, 1986.

[19] J. Taylor, *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements*. University Science Books, 1999. Date Accessed: October 2012.

[20] "Standard deviation." Online. http://en.wikipedia.org/wiki/Standard_deviation. Date Accessed: October 29, 2012.

[21] B. Boashash, *Time-Frequency Signal Analysis and Processing: A Comprehensive Reference*. Oxford: Elsevier Science.