6-2011

A GPS Enabled iPod Touch for Skiers

Camila Dorin
Union College - Schenectady, NY

A GPS Enabled iPod Touch for Skiers

By

Camila Dorin

* * * * * * * * *

Submitted in partial fulfillment

of the requirements for

Honors in the Department of Computer Engineering

UNION COLLEGE

June, 2011

ABSTRACT

DORIN, CAMILA A GPS Enabled iPod Touch for Skiers
Department of Electrical and Computer Engineering, June 2011

ADVISOR: Prof. Shane Cotter, Prof. James Hedrick

The purpose of my senior project was to design and build a system that provides GPS capabilities to the iPod touch. The system is used in conjunction with an iPod application, giving skiers the possibility to have access to the following outputs: graph of speed during run, distance skied, calories burned, and map of the slopes followed. The system consists of a GPS receiver, an Arduino microcontroller, a Wifi Shield, and iPod touch, and an iPod application.

A central point during the development of the system was the transmission of data from one piece of hardware to the other and then finally to the iPod application. The GPS receiver obtains the GPS data and sends it to the Arduino. The Arduino then puts this data in HTML form in order for the WiShield to send it to a web page. Through the WiShield an ad-hoc network between the Arduino and the iPod is established. Through this connection the iPod application is able to obtain the contents from the web page and manipulate it to make necessary calculations, display it, and graph it.

Report Summary

The purpose of this project was to design and build a system that provides GPS capabilities to the iPod touch. The system is used in conjunction with an iPod application, giving skiers the ability to have access to a set of important variables while skiing. In order to provide skiers with accurate and reliable data, there are certain requirements that the system needs to satisfy. First, the system must display an updated version of the distance skied every 30 seconds with at least one decimal point. Second, the application has to show the current speed, updated every 30 seconds and with at least one decimal point. The GPS connection should be reliable and work accurately on the mountain, to show an updated version of the map with the slopes skied every 1 minute. The system should be small enough to carry in a pocket of a backpack or pants. It should be able to communicate with the iPod through a wireless local network. The user's cost of the hardware system should be no more than $100 and the cost of the iPod application should be $4. Finally, the system built should have enough power to stay on for at least 8 hours and the iPod touch should be able to store 8 hours of data.

The hardest and most demanding part of implementing the system was the connection between the iPod touch and the GPS through a Wifi connection since it was not clear if this was going to be possible. This was not very complicated on the Arduino side, but ended up being quite challenging in the iPod application's side. Other challenges were plotting the speed and finding out how to parse the data coming from the web page.

Due to time limitations the map of the runs and the height during jumps could not be implemented in the current prototype. However, using the GPS Logger from Adafruit the user can produce an online map of the slopes skied. All the necessary

documentation has been collected to develop these two outputs in the application's display in the future.

Table of Contents

Table of Figures and Tables

1. Introduction

I enjoy being outside and playing sports and skiing is one of my favorite activities. I have been looking for a way to relate skiing and computer engineering since my junior year. During my year abroad at University College London (UCL) I took a class on Multimedia Computing. This class required a final project which had to contain some kind of multimedia covered in class, such as Photoshop, Dreamweaver, iPhone/Android applications, or 3D modeling, to name a few. I wanted to create an application for the iPhone and the iPod touch which would be useful while skiing, but at that time I did not know what kind of application to create. Instead, I ended doing a website which combined a number of different multimedia.

At the beginning of the 2010-2011 academic year, when looking for possible supervisors for my senior project, I met with Prof. Cotter. He told me about his interests and what other students have been working on under his supervision. One of these projects caught my attention. It consisted of using the iPhone or the iPod touch accelerometer to detect when a person fell, or is about to fall. After hearing about this, the idea for my senior project came to life.

I realized that it would be very useful for any skier to know important features while skiing, such as speed, distance skied, or calories burned. There are devices that provide this information to skiers; however none of these use the iPod as the interface. I am certain that I am not the only person skiing with an iPod (a lot of skiing helmets come with incorporated speakers). Hence, the idea to use an iPod touch to display important variables for skiers seemed perfect.

The aim for this project is therefore to provide skiers with a system that will allow them to keep track of different variables while skiing and at the same

time display these in an intuitive and user-friendly manner using an iPod application. Moreover, the fact that there is nothing similar using the iPod touch made the project unique and challenging.

The remainder of this paper is organized as follows. Section two presents existing technologies that track the skiing experience. It also provides information about iPod application

and accessories for athletes that are currently being used. Section three discusses the requirements that the design needs to satisfy in order to be useful and innovative for skiers. Section four presents five design considerations that the system takes into account. Section five examines possible designs that were considered at the beginning of the project. Section six describes the complete design, both hardware and software, for the system. Section seven discusses the final design of the system and its implementation. Section eight presents the results obtained. Section nine provides the schedule followed for the completion of the required tasks. Section ten outlines the cost for each component and the total cost of the system. Section eleven provides a user manual to use and maintain the system. Finally, section twelve presents the conclusions of the project and some ideas for future development.

2. Background

2.1 Existing Skiing Devices

There are several devices specially designed for skiers to help them keep track of their skiing experiences. Examples of these are the products designed by PhatRat, a company that develops sensors for athletes. The airRat allows skiers to know how much time is spent in the air while jumping. The dropRat tells athletes the maximum height obtained during a jump. Finally, the speedRat records speed with a high accuracy [1]. All these products consist of a transmitter sensor and a receiver. Figure 1 below shows a typical PhatRat product.

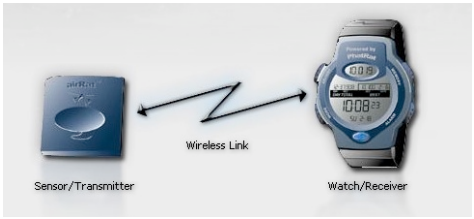

Figure 1: PhatRat Technology [1]

This winter Vail Resorts launched the EpicMix consisting of a chip incorporated on the ski pass that allows skiers to look at a map of slopes skied together with statistics about speed achieved. In addition, it combines social media features such as Facebook or Twitter to maximize the skiing experience, adding a new social dimension to it. Skiers and snowboarders access the data using a pin code through a smart phone or a computer. Figure 2 shows a ski pass from Vail Resorts with the sign that it is compatible with EpicMix [2].

Figure 2: EpicMix Ski Pass [2]

Although the data collected by EpicMix is accessible from an iPod touch or the iPhone, its use is limited to Vail Resorts only. The rest of the products do not use iPod as their interface and as a result, their features and user experience are somewhat limited.

2.2 iPhone and iPod Kits for Athletes

There exist a number of different sports gadgets compatible with the iPod. The most widely used is the Nike+ iPod. The Nike+iPod sensor attaches to a Nike shoe and sends data using a transmitter and an antenna to a receiver attached to the iPod [3]. The information is sent at a radio frequency of 2.4 gigahertz using a proprietary protocol. Together with the data, a unique code is sent to identify from which sensor the information is coming from. The receiver plugs into the iPod using a multi-pin connector and it consists of a processor, a receiver, an antenna, and a number of resistors and capacitors. The transmitter has a range of 18.2 meters [4]. The runner can access a wide array of information while working out, such as distance covered, calories burned, and speed [5]. Figure 3 below presents an iPod with the Nike+iPod application and the sensor that attaches to the shoe.

Figure 3: Nike+iPod Accessory [6]

More recently Nike launched a similar idea, but this time it does not require an extra accessory for the iPod. Instead, the Nike+ GPS uses the accelerometer and the GPS in the iPhone together with an application that allows runners to use any kind of shoe and still enjoy the advantages of the Nike+ iPod combination. Furthermore, if runners use an iPhone a map with the route followed can be seen. The Nike+GPS interface is shown in Figure 4.

Figure 4: Nike+GPS Application[7]

These devices allow runners to have a completely new experience while working out. Due to the innovation of using the iPod or iPhone as their interface and the fact that the application displays data in an intuitive and user-friendly way these products have had a great impact on the market.

With these ideas in mind, I designed a device and an application that will allow skiers to keep a record of important variables while skiing.

3. Design Requirements

3.1 Outputs

The system needs to provide with significant data to skiers in an accurate and timely manner. The following set of outputs will be displayed to the user:

- Maximum and average speed
- Number of kilometers skied
- Map of mountain with paths/slopes skied
- Calories burned
- Height of jumps

All the variables outlined above are of the utmost importance to skiers. Keeping track of the maximum and the average speed as well as the height during jumps will allow skiers to overcome their own previous records. The number of kilometers skied will provide skiers with a greater notion of the distance covered while skiing. The map of the slopes skied will be a graphical way for skiers to know what they skied, what they have left to ski, and it will also work as a map to find your way in case of being lost. Finally, the calories burned will be a very important feature to every skier who in addition to having fun while skiing is also trying to burn calories and lose weight.

3.2 Requirements

Based on products that already exist both for skiing and other sports, the system designed will have to satisfy a series of requirements described below.

Display: The system must be easy to use and intuitive for every skier, no matter the age.

- Calculate calories: Can be an approximation based on time spent skiing and weight of the user.

- Maximum speed: Needs to take speed measurements every 30 seconds and recalculate maximum every time a new measurement is made. It should be accurate to at least one decimal point.
- Average speed: Needs to take speed measurements every 30 seconds and find the new average speed after each new measurement. It should be accurate to at least one decimal point.
- Map of mountain with slopes: This uses GPS and static images of ski resorts. Since each resort will need its own static image, the system will be designed for the top 3 ski resorts for Union students.
- Height of Jumps: This should have at least one decimal point. Need to be able to find the exact time when the height is at its maximum.
- Time: Need to keep track of time skied in order to calculate speed, distance, and calories.

Memory: Needs to store at least eight hours of skiing data per time (equivalent to one day skiing).

Power: Should have power to be on for at least eight hours.

Size: Needs to be smaller than 8x5x2 (in cm) if it is a stand-alone device. This is a reasonable size to be able to store the device in a pocket and carry it while skiing.

Price: Assuming that the skier already owns an iPod touch the cost for the compatible GPS will be

$100 plus the cost of the application which will be $4.

TargetMarket: Any skier. Although it might be more popular for people between twenty and fifty years old.

4. Design Considerations

The following design considerations were taken into account when defining the system.

- Health: Since one of the features of the system is to provide the amount of calories burned while skiing, the system can have a very positive impact on skiers who are trying to lose weight and do exercise. Moreover, since the system will give information about speed and distance skied it might be useful for skiers who are preparing themselves for competitions.
- Safety: The system designed will not incur any safety issues to its users. One might think that it is not safe to use the system on the slopes. However, the aim is for skiers to access the data while taking a break for lunch or coffee or when on the lift and not when skiing on the slopes. In addition, the system might work as a safety tool for people who get lost. The application would need to incorporate some way of contacting the rescue staff and by tracking the GPS the rescue would be able to found the person.
- Social: The system will add more fun to the skiing experience. With the data provided by the system skiers will be able to keep track of their record for speed, height on jumps, and distance, which will allow them to set new goals and make the experience more competitive.
- Marketability: If the system is actually sold in the market, an MFI license will be needed.

  This is not easy to obtain and it is only given to companies that are already established. Hence, marketing the product would have some

limitations.

- Manufacturability: All the components necessary to build the system are easy to obtain.

  The system is made of mainly three components which come already built. These components will be discussed in section 7.1.

5. Design Alternatives

5.1 The Three Possibilities

In order to produce the outputs defined above, three different design options were considered at first.

The first one consisted of building my own system, with an accelerometer, a GPS, and the computer or a small LCD display as the interface. The main disadvantages of this system would have been the lack of innovation, since similar products already exist and the fact that skiers would only be able to access the data after skiing and not on the mountain.

The second design consisted of using the iPod touch as the interface with its integrated accelerometer. In addition, a GPS would be built to provide the iPod with GPS capabilities. The main advantage of this device would be the innovation, since there are no similar products or applications for skiers on the iPod touch. In addition, since there is a high number of skiers that listen to music while skiing, a lot of them already carry an iPod touch while skiing.

The third and last option was to use an Android and its integrated GPS and accelerometer and build an application. The main disadvantages of this alternative are that the project would only contain software design, that the market is limited to owners of an Android phone, and that to use the application the users will be paying the phone's monthly plan.

Table 1 shows a summary of each of the possible designs.

System	Own system with GPS + accelerometer + computer as interface	Ipod with accelerometer + build GPS + application	Android with application
Advantages	-Easier to program -Easier to read/get data	-Nothing similar using iPod -Most skiers have an iPod while skiing -Attractive design	-Can get phone to develop project -GPS included -More open source
Disadvantages	Cannot look at the information while skiing -Not as attractive -Similar products exist	-Need to join MFI developer program to sell on market -Limited resources	-Need to pay phone's monthly plan -Not as attractive -Limited market

Table 1: Design Possibilities

Due to the advantages of the system that uses the iPod touch and the disadvantages that the other alternatives presented I chose the iPod design alternative. At first, the idea was to create a system consisting of a GPS module and a microcontroller and connect it to the iPod using a Bluetooth connection. However, this original idea of connecting the two systems with Bluetooth was modified. This will be explained in detail in the Design section.

5.2 Chosen System Hardware

The system consists of both hardware and software design. Table 2 shows the hardware of the chosen system.

Hardware	GPS	Microcontroller	Wifi	Accelerometer	Ipod
Function	Receive GPS signal	Process GPS data and send it to iPod	Provide Wifi connectivity to microcontroller in order to be able to send data to iPod	Get acceleration of X, Y, and Z axis	Process data Play the role of interface to display all data to user

Table 2: Hardware for the System

The GPS lets the skier know what slopes were skied. In addition, using the iPod's accelerometer the skier is able to keep a record of the maximum and average speed, distance covered, and height during jumps. Using the skier's

weight and time skied, it is possible to calculate the amount of calories burned.

5.3 Chosen System Software

The software for the system is shown in Table 3 below.

Software	Microcontroller's code to get GPS data	Microcontroller's code to transfer data through a local network	Ipod application
Function	Obtain data from GPS receiver	Send data to iPod through Wifi	Process data and interface for user

Table 3: Software for the System

The software of the system consists of the two codes on the microcontroller, one that obtains the GPS data from the GPS receiver and another one that establishes a connection with the iPod and then sends the GPS data to the iPod. The iPod application is the processing tool to make all the calculations necessary to obtain the outputs and the interface for the skier.

6. System Design

6.1 Design Process

The steps that were followed during the design process are the following:

- Designed a device that keeps a record of important information from a skier's runs and that is capable of connecting and interacting with an iPod touch. A GPS module, which communicates with a microcontroller and then transmits the data through a wireless link to the iPod. The code to obtain the GPS data and transmit it to the iPod was written using the Arduino environment.
- Created an iPod touch application that acts as the interface for the user. This application should be able to use the data given by the iPod's accelerometer to determine the skied distance, the height during jumps, and the skier's maximum and average speed. In addition, the application should be able to map the data from the GPS to a static map image in order to display the slopes skied. The iPod application was be programmed in Objective-C using the Xcode editor.
- Manufacture a prototype of the GPS device with the microcontroller and test it on a real iPod touch using the created application. Building a prototype and testing it allowed me to make the necessary adjustments to ensure that it works correctly and that the data obtained makes sense.

6.2 initial System – Using Bluetooth

The Figure below shows the initial block diagram for the project.

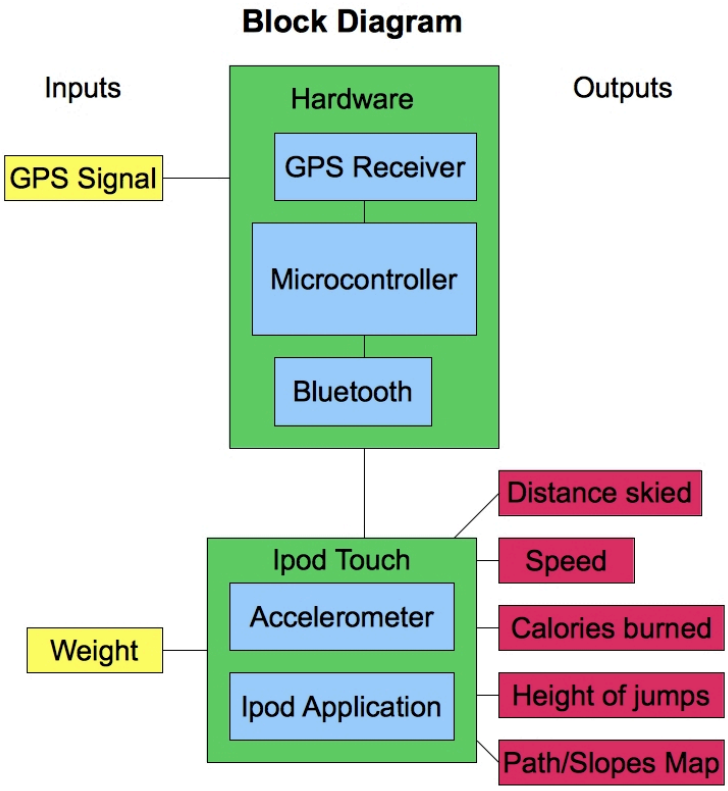

Figure 5: Initial Block Diagram for the System

The inputs to the system are the GPS signal and the weight that the user manually inputs. The inputs are processed by the microcontroller and/or by the iPod. In the case where the microcontroller processes the data it is then transferred to the iPod through Bluetooth. The outputs

of the system are: speed, distance skied, calories burned, and a map of the slopes skied. The user can access these data through an iPod application.

As seen above the original idea was to use a Bluetooth device to send the data from the microcontroller to the iPod touch. However, after doing

research on how to achieve this, it was discovered that although the iPod comes with Bluetooth it has limitations. The Bluetooth on the iPod touch is only to play games, transfer pictures using applications, or listen to music with a Bluetooth headset [8]. It does not allow for file exchange between Bluetooth devices. It is only possible to exchange files and data between Bluetooth devices and the iPod through the Made for Ipod (MFI) Developer program [9]. From information read on different forums in order to gain access to the MFI Developer Program it is necessary to be a company with at least one million dollars of equity. As a result an alternative way for establishing communication between the iPod and the microcontroller was explored.

6.3 Updated Version of the System – Using Wifi

The alternative to connect the Arduino and the iPod consists of creating an ad-hoc Wifi network. Figure 2 shows the updated version of the block diagram.

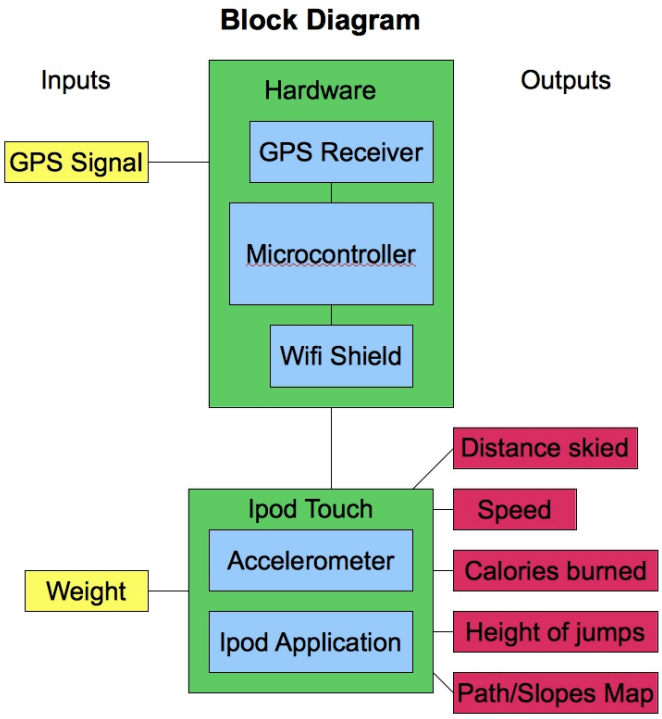

Figure 6: Updated Block Diagram for the System

A shield is a pre-built board that connects directly through Arduino through pin connection and provides some extra capability to the microcontroller, such as different sensors, Wifi connection, Bluetooth connection, or GPS.

6.4 Wifi Communication between Arduino and iPod

There are a number of projects that use an iPod and a Wifi network in order to control cameras, robots, or control trains[10]. There are some applications that create a Wifi network in order to share files, such as the Wifi Photo Transfer which allows transferring pictures from an iPod to the computer using a web browser and an IP address [11]. Also, the iFTPStorage allows sending any type of file from the computer to the iPod using an FTP client and the IP address given by the application [12]. As a result, in theory, any device with Wifi capabilities can interact with the iPod touch through a local network.

The NSURL and the NSURLConnection classes will be used to achieve connectivity between the microcontroller and the iPod touch [13]. Using these classes would also allow us to download the contents from the server to the iPod application.

6.5 Calculation of Outputs

The quantities needed to produce the outputs were computed using the following:

- Current speed: Comes from the GPS, it is transferred to the iPod application through a wifi ad-hoc connection.
- Distance skied: It is calculated from the speed. Assuming the speed remains constant between measurements, the distance is calculated by multiplying the

current speed times the time between measurements. The distance d is, d=v*t, where v is the speed in km/sec and t is the time in seconds. The current value is added to all previous values to obtain total distance skied.

- Plot of speeds: All the speeds are saved in a speed array and then used as the y coordinate in a graph. The x coordinate is the time in minutes.
- Calories burned: When launching the application the user inputs weight. The weight together with the time spent skiing are used to approximate the amount of calories burned. This is updated every two minutes. The relation between the weight and the time spent skiing to find out the calories burned[14] is given by Table 2.

Weight	130 lbs	155 lbs	180 lbs	205 lbs
Calories per Hr	413	493	572	651

Table 4: Weight Related to Calories Burned While Skiing

The outputs described above are the ones that were implemented during this prototype. However, information was collected to implement the following outputs as well.

- Map of slopes: A static map image will be used. The GPS data sent to the iPod will be matched to the static map image. Since a static map image is needed for every ski resort the application will be available for the top 3 resorts for Union students. The MapKit Framework provides an interface to embed maps directly into the application views. This framework also allows methods for annotating the map and performing reverse-geocoding lookups to determine placemark information for a given map coordinate [15]. Consequently, the MapKit Framework will be used to perform the mapping

from the GPS coordinates to the static map image in order to accurately display a map of the slopes skied.

- Average speed: To find the average speed, the speed of the current measurement will be added to all previous measurements and divided by the amount of measurements taken. This value would be updated every 30 seconds.
- Maximum speed: The current speed will be compared to all previous measurements and kept if it is the largest and discarded if it is not. This value will be updated every 30 seconds.
- Height during jumps: The y-coordinate of the acceleration (obtained using the accelerometer) will be integrated twice over a 2 seconds period of time. Only the 20 largest values are kept in memory and displayed to the user.

The accelerometer on the iPod works in the following way: when the iPod is rolled to the sides vertically the X value for the acceleration changes, going from 1 when it is rolled all the way to the left to -1 when it is rolled all the way to the right. The Y-axis is a horizontal line right in the middle of the iPod. When the iPod is pointing upwards is -1 and 1 when it is pointing downwards. Finally the Z-axis, which is -1 if the iPod is facing up, 1 if it is facing down and 0 if it is on any of its sides [16]. Since the X, Y, and Z coordinates in the iPod do not correspond to the X, Y, and Z coordinates of our acceleration while walking, running, or skiing, it will be necessary to find a way to relate and match each of the coordinate pairs. The data from the accelerometer can be noisy and a filter needs to be used to clean it up. For future work, the idea would be to use both the accelerometer and the GPS to find speed and height during jumps and combine them to obtain more reliable information.

6.6 Components

The system uses the following components:

- Arduino Duemilanove: Microcontroller that will receive data from the GPS module.
- EM 406 GPS: Receiver of GPS signal. Will connect to the Arduino and send the data to it.
- Sparkfun GPS Shield: To connect the GPS to the Arduino.
- WiShield: Wifi module, will give Wifi connectivity to the Arduino, allowing it to communicate with the iPod.
- iPod's accelerometer: Will be used to calculate distance skied, average and maximum speed, and height during jumps.
- Ipod: Used as the interface with an application

7. Final Design and Implementation

7.1 Hardware Design

The Arduino and the WiShield come ready to be used. However, the stackable headers that come with the GPS Shield needed to be soldered and this was done as described in the Sparkfun GPS Assembly Guide [17].

The hardware is setup in a way that the three shields are stacked on top of each other. This is a very efficient way of connecting the Wishield and the GPS Shield to the Arduino because it saves space, making the whole system fit in a smaller area. The figure below shows this design.

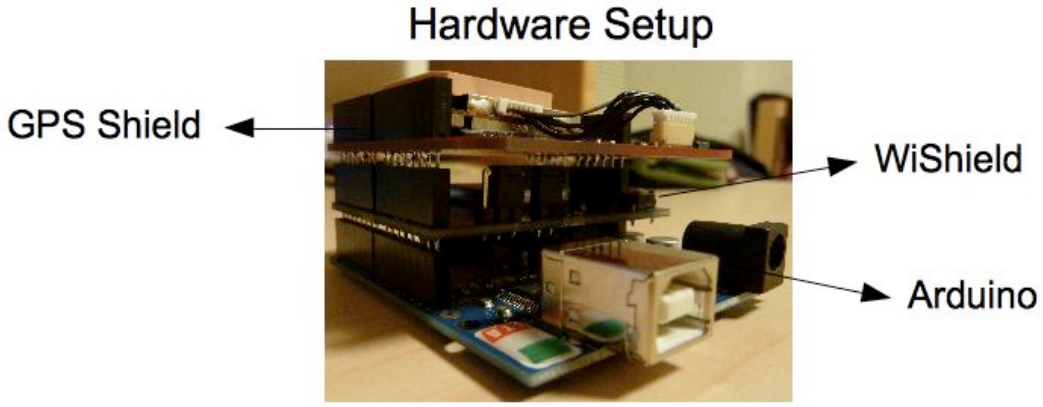

Figure 7: Hardware Setup

7.2 Application Design

Two different design options were considered regarding the graphical user interface for the iPod application. The first one consisted of a view-based application, which contains only one view and it is on this view that all the outputs are displayed. Figure 8 shows the initial design for this application.

Figure 8: Design for View-Based Application

The second option consisted of a multi-view application, where there is a main view from which it is possible to navigate to the secondary views. The figure below shows this design possibility.

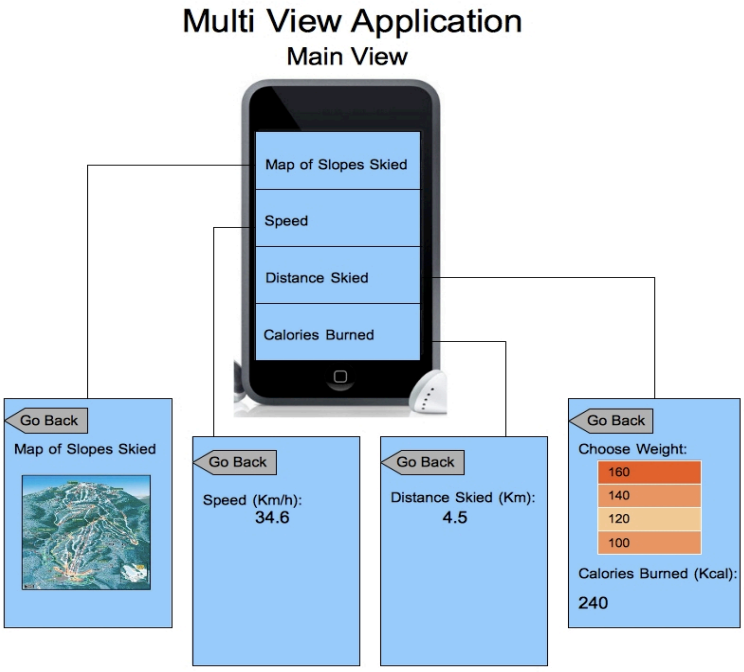

Figure 9: Design for Multi-View Application

7.3 Where to Store Data and How Much Will be Stored?

There are two possible places to store the data from the GPS. The first one is to store it in the Arduino by creating an array for each of the variables. However, Arduino sketches are very limited in size (30000 bytes) and the code with the GPS reading and the creation of the server already take 2/3 of the size. In addition, this data will not be instantly uploaded to the iPod application and instead there would be a waiting time between filling the array and loading the data. The second option is to store the data in the iPod. The advantage would be that there are no memory limitations and as soon as the data is read from the GPS is stored in the iPod.

Supposing a ski run takes around 20 minutes, then if the data is updated every 30 seconds, multiplying 20 times 2 gives us 40 data points. Hence, the iPod will have to store 40 speeds, 40 latitudes, and 40 longitudes, during each run. However, it is important to know that as we chose 30 seconds we could have chosen 2 or 5 seconds too. It was only a number that seemed enough to collect enough data. Nevertheless, the system has the capabilities to refresh the application with new data more frequently.

7.4 Implementation

7.4.1 Transferring GPS data to Arduino

Once the GPS Shield and the Arduino were attached to each other as shown in Figure 7 the Sparkfun GPS Quickstart Guide[18] was followed to transfer the data from the GPS to the microcontroller. The main portions of the code to obtain the GPS data is shown below and was taken from Sparkfun's website.

```
void loop()
{
  while(uart_gps.available())     // While there is data on the RX pin...
  {
```

```
      int c = uart_gps.read();    // load the data into a variable...
      if(gps.encode(c))      // if there is a new valid sentence...
      {
        getgps(gps);         // then grab the data.
      }
  }
}
// The getgps function will get and print the values we want.
void getgps(TinyGPS &gps)
{
        //speed (similar code for all other variables)
        Serial.print("Speed(kmph): "); Serial.println(gps.f_speed_kmph());
}
```

Figure 10: Log GPS Data to Arduino Code Listing

As seen in the piece of code shown above the main loop of the code keeps reading data from the GPS while there is a connection and whenever new data is read, the getgps() method is called in order to display this data. To achieve this, the Serial.print method is used and inside of it the gps.f_speed_kmph() method is called to get the speed data from the GPS (similar methods are used for all the other variables). The results obtained in the serial monitor when the code is run are shown in below in Figure 11.

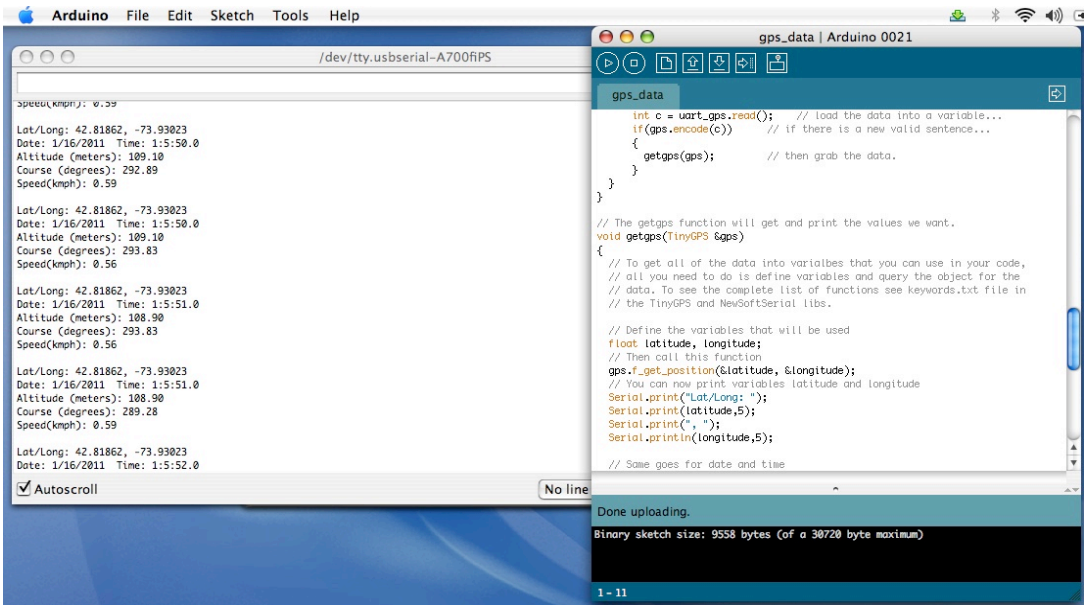

Figure 11: Output of GPS from Arduino's Serial

7.4.2 Establishing an Ad-hoc Wifi Connection between Arduino and iPod

The code used to create an ad-hoc connection on the Arduino was taken from the Asynclabs Wiki [19]. To define adhoc as the wireless mode the following line of code was used:

```
unsigned char wireless_mode = WIRELESS_MODE_ADHOC;
```

In addition, an IP address was specified for the WiShield by using the following:

```
unsigned char local_ip[] = {192,168,1,55};
```

Finally, a name for the adhoc network was specified: `const prog_char ssid[] PROGMEM = {"Skiing"};`

The three steps described above were the most important ones to configure the wireless network. Some of the code to create the webpage and to initialize and run the WiShield is shown below. The complete code that creates the local network, the server and puts the contents coming from the GPS into the server can be found on the Appendix.

```
boolean sendMyPage(char* URL) {
       // Check if the requested URL matches "/"
        if (strcmp(URL, "/") == 0) {
            // Use WiServer's print and println functions to write out the page
content
            WiServer.print("<html><head><title>GPS Data</title><head>");
            WiServer.print("<body>");
            WiServer.print("Hello World");
            WiServer.print("</body></html>");
           return true;
       }
          return false;
    }
void setup() {
      // Initialize WiServer and have it use the sendMyPage function to serve pages
      WiServer.init(sendMyPage);
```

```
void loop(){
     // Run WiServer
      WiServer.server_task();
}
```

Figure 12: Local Network Code Listing

Once the sketch is uploaded to the Arduino and the connection is set up (red light on WiShield is on) the network becomes available for any device with Wifi capabilities to connect to. There were some changes to the iPod the first time it connected to the network. First, in the iPod settings we have to choose correct network. Then, it is necessary to edit the settings to use static IP. The IP address given to the iPod could be any IP on the subnet (in our case any 192.168.1.x) other than the one chosen for the WiShield. The netmask should be set to 255.255.255.0. The last step is to go to safari (or any web browser) and enter http://192.168.1.55 (the IP defined for the WiShield) [20]. The results in this case were similar to the ones obtained for section 7.4.3.

7.4.3 Transferring the GPS data from the Arduino to the iPod

Since Asynclabs provided us with a code to set up a wireless connection by setting up a server with a web page, it seemed reasonable to display the GPS data on this web page in order to transfer it to the iPod. In order to do this, some changes were made to the previous two codes (gps and wiserver) and these were combined into a single sketch. The code is shown on the appendix. The results obtained when the GPS data is transferred to the server and displayed in Safari are shown below.

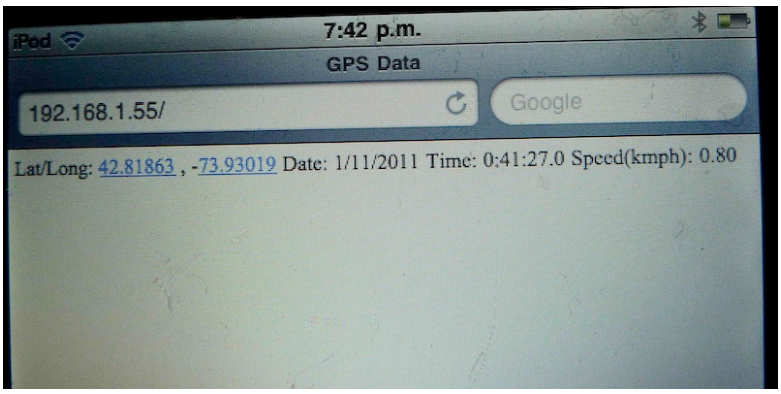

Figure 13: Server Contents Displayed in Safari

We used Wireshark to understand how the ad-hoc connection was working. The information obtained from this experiment is shown below.

GET / HTTP/1.1

Host: 192.168.1.55

User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.2.13) Gecko/20101203 Firefox/3.6.13 (.NET CLR 3.5.30729)

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf

Figure 14: Wireshark Results

7.4.4 Automatically Refresh Server

The GPS receiver is continuously receiving new GPS data. As a result, we needed to find a way to continuously update the server with the new data coming from the GPS. To achieve this the META tag in the HTML description of the web page was used as follows: <html><head><META HTTP-EQUIV=\"Refresh\"CONTENT=\"30; URL=http://192.168.1.55\"></head>

The line of code above defines the period between each reload to be 30 seconds and the reloading is done to the URL defined (in this case the server's IP). The whole

URL is used in order to avoid the cache of data [21].

7.4.5 From Server to iPod Application: HTTP Request

After doing research on what types of web formatting iPod applications can handle, it was found that XML is the most commonly used. However, objective-C's NSURLConnection class is able to send HTTP requests to URLs in HTML format. Different sample codes found online were used to: create an NSURL connection between the server and the application, produce an HTTP request for the contents of the server, take the contents and store them in a string in the iPod application, and take this string and display it as a UILabel on the application's interface [22]. In addition, I received help from the Apple Developer Forum [23]. The main body of the NSURL connection code is shown in the next page.

```
NSURLRequest *request =[NSURLRequest requestWithURL:[NSURL
URLWithString:@"http://192.168.1.55"]];
[[NSURLConnection alloc] initWithRequest:request delegate:self];
- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse
*)response
{
      [receivedData setLength:0];
}
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
        [receivedData appendData:data];
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
{
        // Show error
}
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
        // Once this method is invoked, "receivedData" contains the complete result
}
```

Figure 15: NSURL Connection Code Listing

There were many complicated steps during this process. First, since I was not very familiar with Objective-C, it took me some time to understand that all the code had to be inside methods (single lines of code are not allowed). Second, it was necessary to convert the data received from the HTTP request (NSMutableData object) into a NSString object so that I could use it as the text property of a UILabel. This was done using the following lines of code:

```
NSString *labelText = [[NSString alloc] initWithData: receivedData encoding:
NSUTF8StringEncoding];

theLabel.setText: labelText;

[labelText release];
```

Figure 16: Conversion from NSMutableData to NSString Code Listing

Third, since the NSString object containing the data to be displayed was in the appDelegate class and the graphical objects (what is shown in the application's view) were in the viewController class, it was not straightforward to reference one to the other. As a result, a reference pointer was made to the view controller's UILabel object using:

```
UILabel *labelPointer = [viewController theLabel];

labelPointer.text = labelText;
```

Figure 17:Pointer to UILabel Code Listing

Finally, after some trials the code worked and the contents of a simple web page were displayed in a UILabel object in the application's view. The web page opened in Safari is displayed

below together with the result of downloading its contents to the iPod application.

Figure 18: Server Contents in Safari and its Contents Downloaded into the Application

7.4.6 Parsing HTML Data to Strings and Displaying Speed

Without any HTML tags the contents of the web page created by the server can be displayed in a UILabel. However, the META tag required to refresh the web page requires HTML tags and hence the whole web page must be defined inside HTML tags. As a result, when the NSURLConnection is established between the server and the iPod the requested contents contained the whole HTML document, including tags. For this reason, parsing the HTML data is required to display only the contents needed, such as speed, latitude, and longitude.

At first, a third party parser called ElementParser was used [24]. However,

after researching on how to parse data just by manipulating strings and arrays, I was able to obtain the values for speed, latitude, and longitude using the different methods provided in the NSString and NSArray classes. The code used is shown below.

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
        //labelText contains the whole string with web page contents
        NSString *labelText = [[NSString alloc] initWithData: receivedData encoding:
        NSUTF8StringEncoding];

        //chunks contains the contents of labelText that are separated by “,”
        NSArray *chunks = [labelText componentsSeparatedByString: @","];
        //create a pointer to the speed UILabel
        labelPointer = [viewController speed];

        //fill pointer’s text field with the chunks[1]
        labelPointer.text = [chunks objectAtIndex: 1];
}
```

Figure 19: Data Parsing Code Listing

The code above shows how the speed data was parsed, the complete code that contains the parsing of latitude and longitude as well is shown in the Appendix. The figure below shows the speed field in the application’s interface.

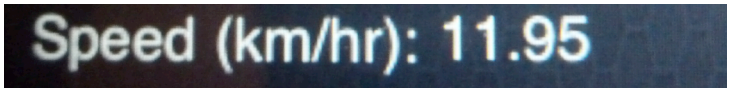

Figure 20: Speed Field in Application

7.4.7 Automatically Update Application: NSURL Connection

Using the META tag provided us with a way for refreshing the contents of the web page with new GPS data every 30 seconds. However, a way for refreshing the iPod application’s data was also needed. Hence, the idea of using the NSTimer class was explored. It was clear that the type of NSTimer that was needed was the scheduledTimerWithTimeInterval [25], since we wanted to create a new HTTP request every 30 seconds. The timer is defined as follows:

```
+ (NSTimer *)scheduledTimerWithTimeInterval:(NSTimeInterval)seconds target:(id)target
selector:(SEL)aSelector userInfo:(id)userInfo repeats:(BOOL)repeats
```

Figure 21: NSTimer Definition

Where scheduleTimerWithTimeInterval specifies how often the timer is fired, the target is the object to which to send the message specified by the aSelector when the timer fires. The aSelector is the message to send to target when the timer fires. The userInfo is the user info for the timer, and the repeats, which specifies if the timer will reschedule itself or not [26].

It was hard to understand where the NSTimer should be declared and what method should it called. The options were either on the AppDelegate class (where all the NSURLConnection declarations take place) or the ViewController class (where everything related with the display takes place). In addition, if the NSTimer was added to the AppDelegate class, there were two methods that could be used as selectors. The createRequest method, where the connection between the server and the iPod takes place or the connectionDidFinishLoading method, where the contents of the server are converted into an NSString. After trying multiple codes and researching how to use NSTimers, I finally understood how it worked and where it should go. The timer had to be declared in the applicationDidFinishLaunching method in the AppDelegate class and the selector should be the createRequest method. The code is as follows:

```
[NSTimer scheduledTimerWithTimeInterval: 30.0 target: self
selector:@selector(createRequest)userInfo: nil repeats:YES];
```

Figure 22: NSTimer to Create new HTTP Request Code Listing

However, this caused a null pointer exception. After some time trying to find out why the null pointer exception error was coming up, I realized that in the connectionDidFinishLoading (which is the last method called in the NSURL connection) I was releasing all the data objects, as follows: `[labelText release];`

Releasing objects in Objective-C causes the pointer to those objects to disappear. Hence, the second time that the NSTimer was fired and the createRequest method was called, the pointers to those data objects did not exist any more. Finally, by removing all the releases the code worked and I was able to see in the console that the arrays were updated (as the GPS was moved). Using the NSTimer automatically refreshed the application's display every 30 seconds too.

7.4.8 Plotting Speed Data

The speed is updated every 30 seconds and the values are saved in an NSMutableArray. Using this array as the y-axis and time in minutes as the x-axis a plot for the speeds during a ski run was generated. The framework used to produce the plot is CorePlot, which is the most popular open framework to create graphs in objective-C and almost the only one in current development.

It was not very easy to set up the framework and add the code to produce the plot, but after reading the instructions and guidelines provided by the CorePlot project [27], I was able to create a graph with fake data. The next step was to be able to pass the array of speeds as the y-axis. This was complicated due to the fact that the speeds array is created in the AppDelegate class, while all the plot code needs to be in the ViewController class and particularly in the viewDidLoad method. With some help from the Apple Developer Forum [28], I was able to create an array in the ViewController class that pointed to the AppDelegate speeds array. The code below shows how this pointer to the ViewController is created.

```
 [speeds addObject:[NSString stringWithFormat:@"%.2f",speed_float]];

//update pointer to viewController array with new data for speeds
[self.viewController updateWithDataPointer: speeds];
```

Figure 23: Pointer to Speed Array Code Listi

The figure below shows the plot generated using CorePlot.

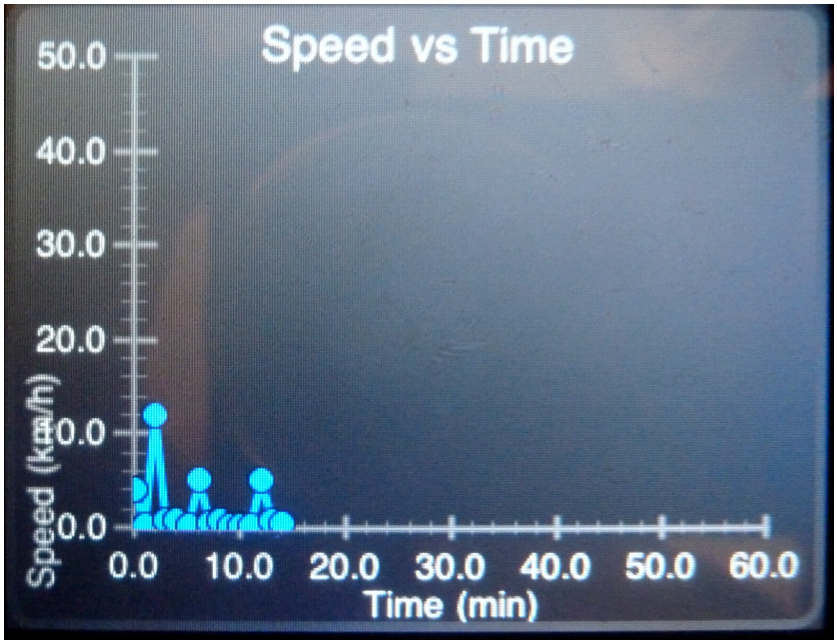

Figure 24: Plot Being Generated as the Array of Speeds is Filled

7.4.9 Calculating Distance Skied

To calculate distance skied the relationship between speed, distance and time was used. Since new data for the speed is obtained every 30 seconds, we can assume that during those 30 seconds the speed remains approximately constant. Hence, the distance skied is the speed multiplied by the time (30 seconds). The speed is obtained from the GPS receiver in km/h and as a result it was necessary to convert this to km/s by dividing it by 3600. The code below shows how the distance for the current speed is calculated. In addition, the distance obtained is added to the distances array.

```
NSUInteger i;
CGFloat dist;
NSString *dist_string;

for(i = 0; i < speeds.count; i++) {
dist = speed_float*30/3600;
dist_string = [NSString stringWithFormat:@"%.2f", dist];
[distances addObject: dist_string];
}
```

Figure 25: Calculation of Distance Skied Code Listing

The elements in the distances array have to be added together to obtain the total distance skied. Once the sum of the distances is calculated it is converted into a string so that it can be passed as a pointer to the distance UILabel in the ViewController class. Figure 26 below shows the distance field in the application.

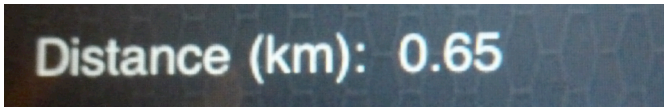

Figure 26: Distance in Application's Interface

7.4.10 Calculating Calories Burned

As mentioned in section 6.5 the calories burned are approximated using the following table.

Weight	130 lbs	155 lbs	180 lbs	205 lbs
Calories per Hr	413	493	572	651

Table 5: Weight Related to Calories Burned While Skiing

At the beginning of the application the user is required to input his/her weight. Then, this weight is used to find out the calories burned. The code below shows the two NSTimers used, one that calls the initializeCal method only once and the other one that calls the updateCalories method every 2 minutes. Both of them are included in the ViewController class in the viewDidload method.

```
//update calories burned every 2 minutes
[NSTimer scheduledTimerWithTimeInterval:5.0 target:self
selector:@selector(initializeCal) userInfo:nil repeats:NO];
[NSTimer scheduledTimerWithTimeInterval:120.0 target:self
selector:@selector(updateCalories) userInfo:nil repeats:YES];
```

Figure 27: NSTimers to Update Calories Code Listing

The initializeCal method is displayed below. Its only function is to set the initial calories amount to 0 and to display 0.00 in the application's screen.

```
-(void)initializeCal
{
cal = 0.00;
self.calories.text = [NSString stringWithFormat:@"%.2f", cal];
}
```

Figure 28: initializeCal Method Code Listing

Next, the updateCalories method was written. This finds out in which range the user's weight is and assigns a new calorie amount to the calories UILabel, adding it to the previous value.

```
-(IBAction)updateCalories
{
NSInteger weight = [userWeight.text integerValue];
CGFloat time = 2.00;

if (weight <= 130) {
cal = cal + (295*time/60);
self.calories.text = [NSString stringWithFormat:@"%.2f", cal];
}
else if (weight > 130 && weight <= 155)
{
cal = cal + (352*time/60);
self.calories.text = [NSString stringWithFormat:@"%.2f", cal];
}
else if (weight > 155 && weight <= 180)
{
cal = cal + (409*time/60);
self.calories.text = [NSString stringWithFormat:@"%.2f", cal];
}
else
{
cal = cal + (465*time/60);
self.calories.text = [NSString stringWithFormat:@"%.2f", cal];
NSLog(@"value of cal: %.2f", cal);
}
}
```

Figure 29: updateCalories Method Code Listing

In addition, since the user has to input his/her weight using the keyboard in the iPod, it is necessary to create a method that will make the keyboard disappear from the screen once the user touches the display. The code is shown below.

```
//resign keyboard
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event

{
    [userWeight resignFirstResponder];
    [super touchesBegan:touches withEvent:event ];
}
```

Figure 30: Resign Keyboard Code Listing

The figure below shows both the field where the user inputs his/her weight and the calories burned field.

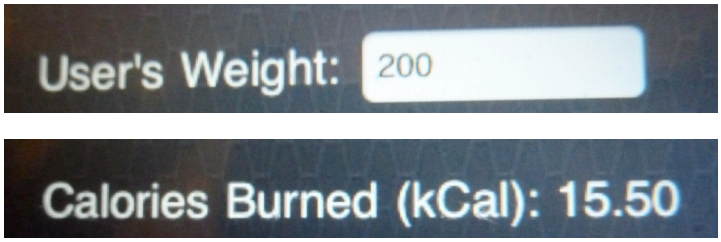

Figure 31: Input Field for User to Provide Weight and Calories Burned Field

7.4.11 Online Plot of Slopes Skied

The GPS logger from Adafruit is a shield compatible with the EM-406 GPS [29]. This shield has the capability to store GPS data using an SD card. Replacing the GPS Shield from Sparkfun with this shield allows the system built to produce a map of the slopes skied online. All the GPS coordinates are logged into the SD card as a text file. The file can then be downloaded into a computer. Using a website called gpsvisualizer.com skiers are able to upload the text file and then generate a map of the slopes skied (provided by Google Maps). The contents of the text file look like this:

$GPRMC,212307.641,A,4249.0640,N,07355.6953,W,1.83,63.52,010311,,*29

$GPRMC,212308.641,A,4249.0649,N,07355.6914,W,0.19,133.31,010311,,*1F

$GPRMC,212309.641,A,4249.0618,N,07355.6973,W,0.51,226.01,010311,,*13

It should be noted that a similar text file with GPS data could have been produced within the iPod application without having to use the GPS Logger Shield from Adafruit. To do this the GPS coordinates that are already being saved in arrays would need to be written into a text file generated in the iPod application. Then, this file could be uploaded to gpsvisualizer.com and an online map could be produced. This approach has not been tried due to time limitations.

8. Performance Estimates and Results

When the project was started the iPod touch capabilities for interacting with other devices were unknown. However, I was able to communicate the Arduino and the iPod through a wifi network and transfer data from the microcontroller to the iPod. I was not able to test the system while skiing. Nevertheless, I tested it while walking and running and everything is updated as expected.

8.1 GPS Results

The first result obtained were the GPS coordinates coming from the GPS receiver. To make sure that the GPS readings were correct the latitude and the longitude given by one of the readings were typed in Google Maps. The results are shown in Figure 32.

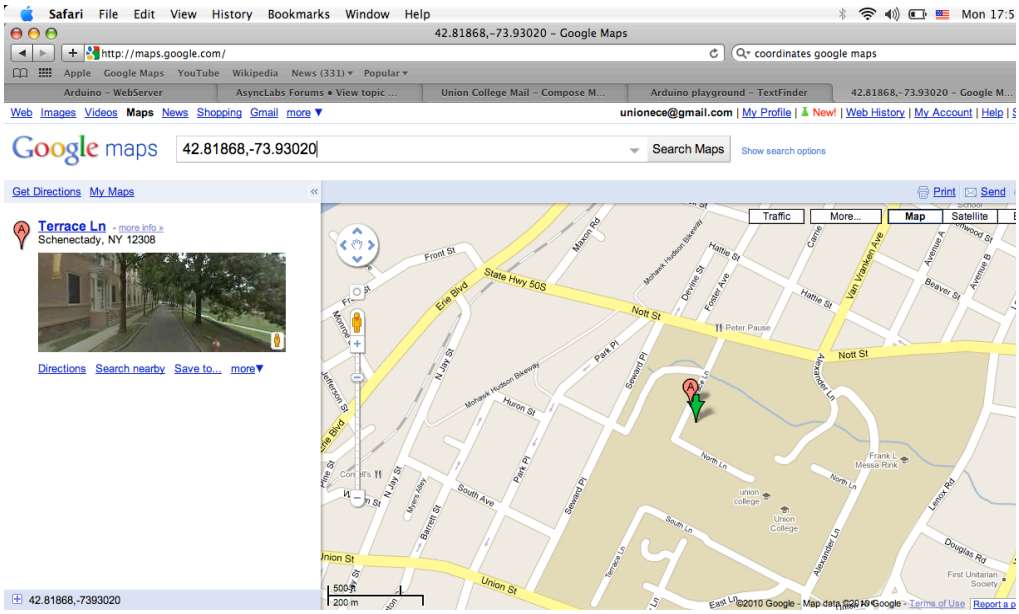

Figure 32: Verification of Latitude and Longitude Given by GPS

As seen above the reading is correct since it took place in my dorm room in Wold House and this is exactly the location displayed in Google Maps.

The accuracy of the results for the speed is limited to the accuracy of the GPS. The GPS is not very good at finding the speed when it is static. Skiers spend most of

the time in motion, so this is not a big problem. In order to completely avoid this problem, a good approach would be to have the option of allowing the user to choose when to start "recording" and when to stop. In future versions, the idea would be to try a hybrid approach using both the GPS speed and the speed from the accelerometer. Since the value for the distance is calculated from the speed its accuracy is directly dependent on the accuracy of the speed. The calories burned value is a complete approximation and at this point I do not have any better information on how to make this value more precise, however it provides a reliable result.

The NSTimer provided a method for creating a new HTTP request every 30 seconds and this allowed us to fill arrays for speed, latitude, and longitude. The figure below shows how these arrays are being filled as time passes by.

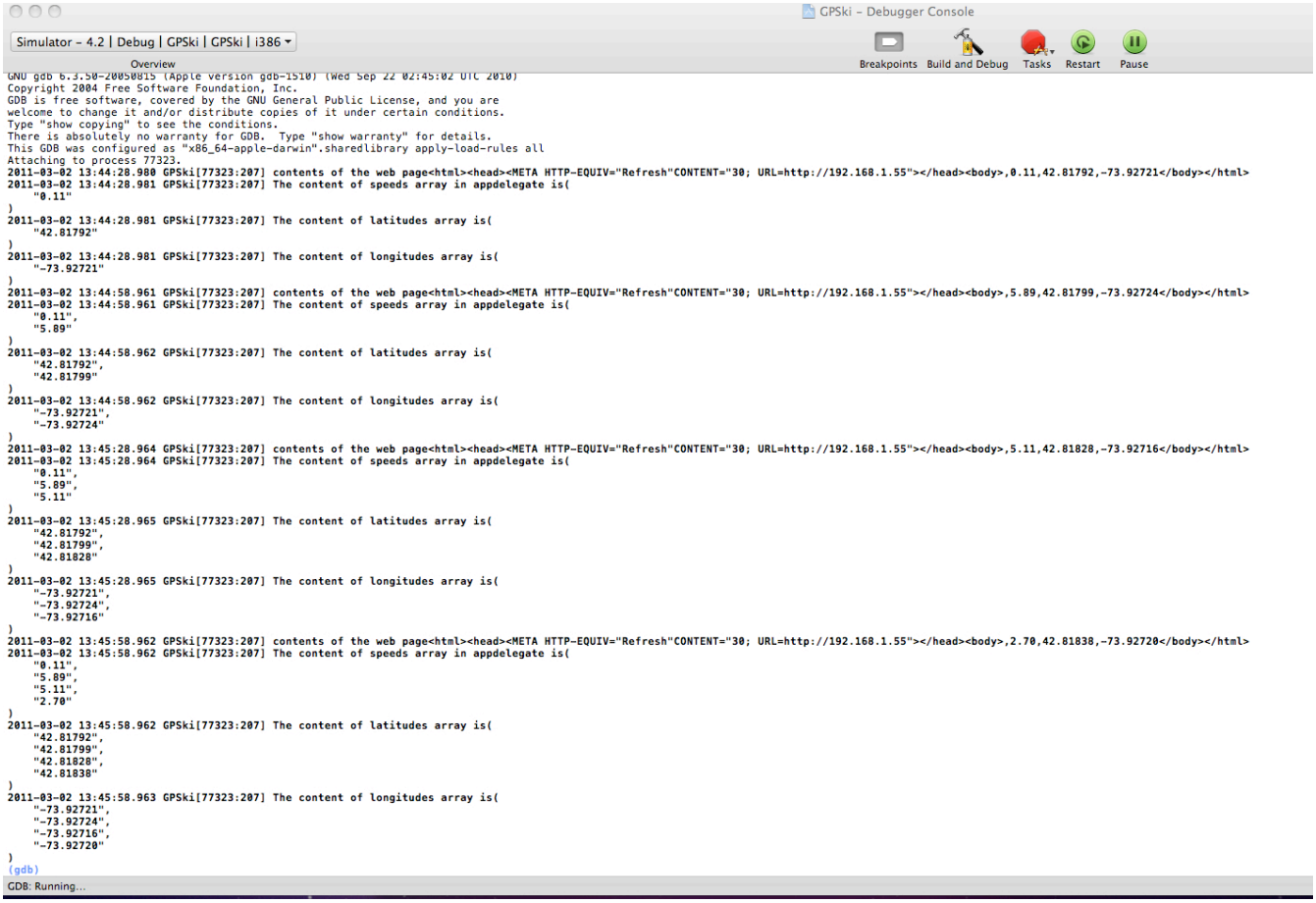

Figure 33: Arrays for GPS Data Being Filled In as Time Passes

Since the system has not been tested while skiing, the figure below shows the map obtained from logging GPS data while walking around campus.

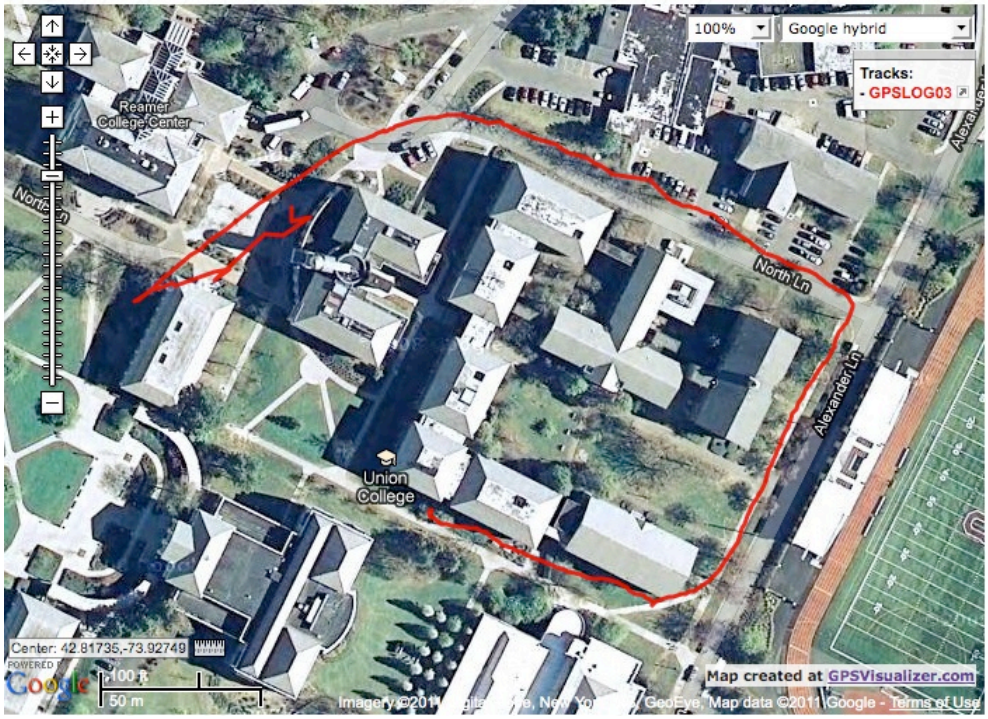

Figure 34: Data Logged using GPS Logger and gpsvisualizer.com

As seen above, the results obtained are correct and correspond to the path that I produced while walking.

8.2 Application Results

The final application's interface containing all the initialized outputs is shown below.

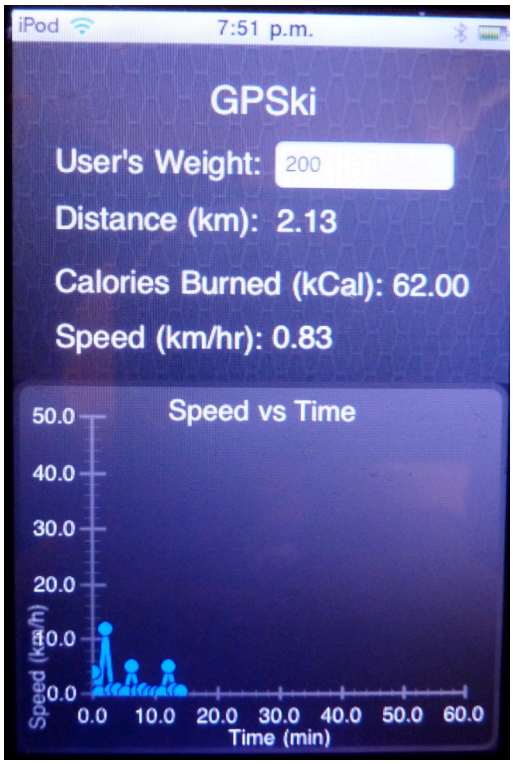

Figure 35: iPod Application's Interface

As seen above the current iPod application has eight UILabels that display the titles and the outputs. It contains a UITextField that allows for user input (user's weight) and a graph object that displays the plot of the speeds. All the outputs are automatically refreshed at different intervals.

8.3 Other Results

The system designed is only a prototype and it does not meet the requirements specified for size and battery life. However, a smaller and lighter version of the system is expected to be produced in future versions. This could be achieved by replacing the Arduino Duemilanove by the Arduino Pro Mini [30], which has the

same capabilities but it is much smaller. In addition, the WiShield could be replaced by the YellowJacket [31], this has the same functionality as the WiShield, but it is designed for the Arduino Mini. While testing the system we found out that the 9V battery only provides enough power for approximately an hour. The original power requirement was at least 8 hours, new methods for powering the system have to be explored and a lithium-ion battery could be an option.

Due to time constraints all the outputs defined were not implemented. Nevertheless, all the background information in order to develop those outputs exists and it is only a matter of spending time in implementing them.

9. Project Schedule

9.1 Fall 2010 Schedule

During the fall term of 2010 the main research for the project was conducted. Essential information was collected on how the accelerometer works, the possibilities of connecting hardware to the iPod, and the GPS alternatives. The deliverables for ECE 498 have also been prepared during the fall of 2010. The detailed schedule that has been followed is shown in Table 6 below.

Fall 2010	Tasks to Complete	
Week 1	Talked to professors to decide what I would like to work on	Looked for supervisors
Week 2	Decide what project to work on	
Week 3	Initial research to finish defining project	
Week 4	IEF Proposal	Join University Developer Program
Week 5	Research on accelerometer	Write accelerometer application
Week 6	Research on Bluetooth	
Week 7	Research on GPS	Prepare design proposal presentation
Week 8	Research on ad hoc network alternative	Give design proposal presentation
Week 9	Write report	
Week 10	Write report	Website

Table 6: Project schedule for fall 2010

9.2 Winter 2011 Schedule

The plan for the winter term was to write the code needed to transfer the GPS data from the GPS receiver to the Arduino, write the code to establish the Wifi connection in order to transfer the GPS data from the Arduino to the iPod, and then finally write the iPod application. The last three weeks were left for debugging and testing. In addition the final report was written throughout the term in order to have it ready for the deadline. Details of the project schedule for winter 2011 can be seen in Table 7.

Winter 2011	Tasks to Complete	
Week 1	Write code to get GPS data	
Week 2	Write code to transfer GPS data to Arduino	Write Report
Week 3	Write code to create local network and server	Write Report
Week 4	Write code to transfer GPS data from Arduino to iPod without going to Safari HTML meta tag to automatically refresh web page	Write Report
Week 5	NSTimer to refresh connection in iPod	Write report
Week 6	Plot speed	Write report
Week 7	Plot speed and write code to get total distance	Write report
Week 8	Calories burned and debugging problem with Wishield	Write Report. Presentation and Poster
Week 9	Offline mapping and testing	Write Report. Presentation and Poster
Week 10	Finish Report	

Table 7: Project Schedule for winter 2011

10. System Cost

10.1 Initial Cost of the System

When the IEF proposal was prepared the main costs of the project were the following: a GPS antenna, a GPS receiver, an Arduino microcontroller, and a Bluetooth module. The GPS receiver provides a signal to keep track of the skier's position. The GPS antenna enhances the signal received, making it easier to process it. The Arduino microcontroller connects to the GPS receiver and will be used to process the signal received. Two microcontrollers were ordered, the Arduino Duemilanove to develop the prototype and the Arduino Pro Mini to actually use it on the iPod accessory. Finally, the Bluetooth module transfers the data from the Arduino microcontroller to the iPod over the wireless connection between the two devices. The table below shows the initial cost for the project.

Item	Cost
12 Channel Micro-miniature GR-10 GPS Receiver	$19.95
Antenna GPS Chip-Scale	$1.5
Arduino Pro Mini 328 - 5V/16MHz	$18.95
Bluetooth SMD Module - Roving Networks	$29.95
Total	$70.35

Table 8: Project's Initial Budget

10.2 Updated Cost of the System

After receiving the parts ordered we realized that the GPS was not mounted on a board and ready to connect to the Arduino. As a result, we ordered a different board and shield. In addition, since we realized that it was not going to be possible to connect the Arduino and the iPod through a Bluetooth link, we had to order a Wifi shield. One last change was made and instead of ordering the Arduino Pro, the Arduino Duemilanove was ordered. An updated version of the project's budget is

shown in Table 9.

Item	Cost
GPS Shield Retail - Sparkfun	$79.95
WiShield - Asynclabs	$55
Arduino Duemilanove	$25
GPS Logger - Adafruit	$19.95
Total	$176.9

Table 9: Project's Updated Budget

10.3 Budget for Future Prototype

Now that an initial prototype for the system has been designed and built the idea is to reduce the system's size and weight so that it is more portable for skiers. This could be achieved by replacing the Arduino Duemilanove by the Arduino Mini, the GPS Shield by just a GPS receiver, and the WiShield by the YellowJacket. Table 10 below shows the cost for this system.

Item	Cost
12 Channel Micro-miniature GR-10 GPS Receiver	$19.95
Antenna GPS Chip-Scale	$1.5
Arduino Pro Mini 328 - 5V/16MHz	$18.95
YellowJacket	$55
Total	$95.4.

Table 10: Budget for Future Prototype

As seen above, the cost for manufacturing the new prototype would go down by $81.5. Moreover, if the system was to actually be produced in large quantities to sell it in the market, the production cost would be even less. If we were to buy more than 100 Arduino Minis the cost would go down to $15.16. The cost for the GPS for more than 100 units would be $15.96 and the antenna would cost $1.20. The cost for the YellowJacket when buying in large quantities is not known since Asynclabs is

currently out of stock, but it can be approximated to $41. As a result, the new cost for the system would be $73.32. The production cost can be reduced even more if instead of using the Wifi chip from Asysnclabs we build our own.

11. User's Manual

The system is designed to be very easy to use. The skier should carry the GPS device in a pocket or in a backpack while skiing. The first time the system is connected to the skier's iPod there are a few setting steps that have to be followed. These are described below:

1. Turn on GPS device and wait for blinking red light.

2. Turn on iPod and go to settings and select the Wifi tab.

3. Choose "Skiing" network and touch the arrow for more information on this network.

4. From the 3 options given for the IP Address select Static.

5. Touch the IP Address and type: 192.168.1.55

6. Touch the Subnet Mask and type: 255.255.255.0

7. All the other fields should be left as they are.

Now the iPod is ready to receive data from the GPS device. Look for the GPSki application on the iPod and open it. The screen shows a range of data: current speed, calories burned, distance skied, and a plot of the speeds.

In addition, by removing the sd card from the socket in the GPS Logger Shield it is possible to download the text files to a computer and create a map of the slopes skied. In order to do this, it is necessary to copy and paste the files from the sd card to the computer and open a web browser and type: www.gpsvisualizer.com Once on this site, the text files from the sd card can be uploaded and a map will be generated.

The only maintenance that the system needs is the change of the 9V battery when this doesn't have any more charge. In addition, it is necessary to charge the iPod when it runs out of battery. In the case of a new version for the GPSki application being released, the latest update can be downloaded from the App Store.

12. Conclusions

The purpose of my senior project was to design and build a system that is compatible with the iPod touch and that will keep track of important variables while skiing. Users are able to access the outputs through an iPod application. The original design contained the following outputs: average and maximum speed, distance skied, calories burned, map of the slopes skied, and height during jumps. Due to time limitations, the last two outputs have not been implemented and only the current speed is displayed.

The system built consists of a GPS receiver, a WiShield, and an Arduino microcontroller. During the fall term all the parts for the system were ordered and received. During the first term of the project enough data was collected to completely program the system during this term.

A central point during the development of the system was to find ways to transfer data from one piece of hardware to the next one. First, the GPS data is logged using an Arduino sketch that asks for the GPS data coming from the GPS receiver. In order to transfer the GPS data from the Arduino to the iPod an ad-hoc local network is created using the WiShield. Moreover, the WiShield creates a server containing the data coming from the GPS, the contents from the server are defined using html and are updated every 30 seconds. Then, the iPod can download the contents from the server by creating an NSURLConnection, where an HTTP request is sent. Using an NSTimer an HTTP request is created every 30 seconds in order to download the new and updated contents from the server. The data downloaded is then stored in the iPod using arrays. Arrays for the speed, the latitude, and the longitude were created and this are being filled with new elements every 30 seconds. Converting speed to distance a new array for distance is declared as well. Finally, using the weight of the

user (user inputs weight at beginning of application) and the time spent skiing an approximation for the calories burned is produced.

The main problems encountered during the implementation of the system were the following. First, data had to travel from the GPS receiver to the Arduino, then uploaded to a server, and from there to the iPod application. Hence, a lot of the steps in the project involved finding ways of transferring data from one place to the other. Although this was not expected it was a very good way of learning how transmission of data between devices works. The main difficulty in this problem was to be able to create an HTTP request in the iPod application in order download the contents from server. Second, the data contained in the server had to be updated every 30 seconds with new GPS data. Refreshing the server and creating a new HTTP request in the iPod application was not difficult. However, due to my lack of knowledge in Objective-C it took me some time to understand how to set the NSTimer in the iPod application to fire a new HTTP request every 30 seconds. Third, being able to create a plot of the speeds was challenging because it was necessary to add the CorePlot framework to the project. This required following a number of steps to change the settings of the GPSki project and be able to use the framework. Once this was achieved, I had to be able to pass the speed array as the y coordinate for the plot. This was not trivial because the speed array was defined in the application delegate class, while the plot object was defined in the view controller class. Finally, this was achieved using a pointer. Once the plot was working correctly, I felt much more confident with objective-C and was able to write the code for obtaining distance and calories burned fairly quickly.

Having to deal with the transfer of data from one place to the next one changed the direction and focus of the project. Overall, the results obtained met

almost all the design requirements. The current prototype effectively displays speed, distance skied, calories burned, and a plot of the speeds, and in addition, the skier can access an online map of the slopes skied. There are two missing features and these are the height during jumps and the display of a map of the runs in the application's view. The first one would be implemented using the accelerometer and the second one would need the GPS coordinates. Since the latitude and longitude are already being stored in arrays in the application and plenty of documentation has been collected about the accelerometer and the map framework the necessary information is available to develop those missing features in the future. In addition to implementing these two new outputs, in future versions of the system I would like to use both the GPS and the accelerometer to obtain a better estimate of the speed. I would also like to implement a multi view application to replace the current view based application. Finally, I would like to use smaller and lighter components. The Arduino Duemilanove could be replaced with the Arduino Pro Mini and the WiShield could be replaced by the YellowJacket.

I am glad that the project took a new direction. It allowed me to not only learn to develop an iPod application but also learn about Arduino and the WiShield, servers and HTML, HTTP requests and parsing of data.

13. References

[1] PhatRat Company, http://www.phatrat.com/

[2] EpicMix – FAQ, http://www.snow.com/epicmix/faq/home.aspx

[3] Nike Running – Nike+iPod, http://nikerunning.nike.com/nikeos/p/nikeplus/en_US/

[4] How Stuff Works – Nike+iPod, http://health.howstuffworks.com/wellness/diet-fitness/information/nike-ipod.htm

[5] Apple – Nike+iPod, http://www.apple.com/ipod/nike/

[6] Nike+iPod Picture - http://www.newlaunches.com/archives/nike_system_coming_to_iphone_plus_gets_wifi.php

[7] Nike+GPS Application - http://sportsvibe.com/nike-gps-application-for-iphone-by-nike/

[8] Ifans Website – iPod Touch Bluetooth Capabilities, http://www.ifans.com/forums/showthread.php?t=266532

[9] Stackoverflow Website – Measuring velocity with iPod/iPhone, http://stackoverflow.com/questions/1994018/measuring-velocity-via-iphone-sdk

[10] Apple Developer Forums – Wifi network, http://devforums.apple.com/message/322904#322904

[11] WiFi Photo Share Application - http://wifiphoto.bjano.hu/

[12] iFTPStorage Application - htpt://steppinstonez.com/tp

[13] Apple Developer – NSURL class, http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSURL_Class/Reference/Reference.html

[14] Nutri Strategy - Calories burned while skiing, http://www.nutristrategy.com/activitylist3.htm

[15] Apple Developer – MapKit Framework, http://developer.apple.com/library/ios/#documentation/MapKit/Reference/MapKit_Framework_Reference/_index.html%23//apple_ref/doc/uid/TP40008210

[16] The Unofficial Apple Weblog – Accelerometer, http://www.tuaw.com/2007/09/10/iphone-coding-using-the-accelerometer/

[17] Sparkfun GPS Assembly Guide, http://www.sparkfun.com/tutorials/184

[18] Sparkfun GPS Quickstart Guide, http://www.sparkfun.com/tutorials/173

[19] Asynclabs Wiki, http://asynclabs.com/wiki/index.php?title=AsyncLabsWiki

[20] Asynclabs Forum - Using WiServer, http://asynclabs.com/forums/viewtopic.php?f=15&t=62

[21] HTML Tutorial – Use of meta tag,

http://www.htmlcodetutorial.com/document/index_tagsupp_4.html

[22] NSURL Connection:
http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/URLLoadingSystem/Tasks/UsingNSURLConnection.html%23//apple_ref/doc/uid/20001836-BAJEAIEE

[23] Apple Developer Forum - Thread on establishing NSURL connection,
https://devforums.apple.com/message/368654#368654

[24] Element Parser, http://touchtank.wordpress.com/element-parser/

[25] Apple Reference - Using NSTimer class,
http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Timers/Articles/usingTimers.html%23//apple_ref/doc/uid/20000807-CJBJCBDE

[26] Apple Reference – scheduledTimerWithTimeInterval,
http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSTimer_Class/Reference/NSTimer.html%23//apple_ref/occ/clm/NSTimer/scheduledTimerWithTimeInterval:target:selector:userInfo:repeats

[27] Core Plot Project, http://code.google.com/p/core-plot/

[28] Apple Developer – Pointer to array in view controller,
https://devforums.apple.com/message/379221#379221

[29] Adafruit - GPS Logger, http://www.ladyada.net/make/gpsshield/

[30] Arduino Pro Mini - http://www.arduino.cc/en/Main/ArduinoBoardProMini

[31] YellowJacket -
http://asynclabs.com/store?page=shop.product_details&product_id=24&vmcchk=1

Appendix

Arduino Code that Uses WiServer to Serve a Web Page with GPS Data:

```
#include <WiServer.h>

    #include <NewSoftSerial.h>

    #include <TinyGPS.h>

    #define RXPIN 2

    #define TXPIN 3

    #define WIRELESS_MODE_INFRA   1

    #define WIRELESS_MODE_ADHOC   2

      int ledPin =  7;    // LED connected to digital pin 7


    // Wireless configuration parameters ----------------------------------------

    unsigned char local_ip[] = {192,168,1,55};   // IP address of WiShield

    unsigned char gateway_ip[] = {192,168,1,1};   // router or gateway IP address

    unsigned char subnet_mask[] = {255,255,255,0};   // subnet mask for the local
network

    const prog_char ssid[] PROGMEM = {"Skiing"};      // max 32 bytes

    unsigned char security_type = 0;   // 0 - open; 1 - WEP; 2 - WPA; 3 - WPA2

    // WPA/WPA2 passphrase

    const prog_char security_passphrase[] PROGMEM = {"1234567"};   // max 64
characters


    // WEP 128-bit keys

    // sample HEX keys

    prog_uchar wep_keys[] PROGMEM = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
0x09, 0x0a, 0x0b, 0x0c, 0x0d,   // Key 0

                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,   // Key 1

                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,   // Key 2

                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00   // Key 3

                };


    // setup the wireless mode
```

```
    // infrastructure - connect to AP
    // adhoc - connect to another WiFi device
    //unsigned char wireless_mode = WIRELESS_MODE_INFRA;
    unsigned char wireless_mode = WIRELESS_MODE_ADHOC;
    unsigned char ssid_len;
    unsigned char security_passphrase_len;
    // End of wireless configuration parameters ----------------------------------------
--



    //Beginning of GPS configuration --------------------------------------------------
--
    //Set this value equal to the baud rate of your GPS
    #define GPSBAUD 4800
    // Create an instance of the TinyGPS object
    TinyGPS gps;
    // Initialize the NewSoftSerial library to the pins you defined above
    NewSoftSerial uart_gps(RXPIN, TXPIN);

    // To get all of the data into varialbes that you can use in your code,
    // all you need to do is define variables and query the object for the
    // data. To see the complete list of functions see keywords.txt file in
    // the TinyGPS and NewSoftSerial libs.
     void getGPS(TinyGPS &gps);


     boolean sendMyPage(char* URL) {
       // Check if the requested URL matches "/"
        if (strcmp(URL, "/") == 0) {
            // Use WiServer's print and println functions to write out the page
content
          //print speed and auto refresh
            float latitude, longitude;
            gps.f_get_position(&latitude, &longitude);


            WiServer.print("<html>"); //contents chunks[0]
```

```
            WiServer.print("<head><META HTTP-EQUIV=\"Refresh\"CONTENT=\"30;
URL=http://192.168.1.55\"></head>");

            WiServer.print("<body>");

           WiServer.print(",");  //contents chunks[1]

            WiServer.print(gps.f_speed_kmph());

           WiServer.print(","); //contents chunks[2]

            WiServer.print(latitude);

            WiServer.print(","); //contents chunks[3]

            WiServer.print(longitude);

            WiServer.print(","); //contents chunks[4]

            WiServer.print("</body>");

            WiServer.print("</html>");

           return true;

       }

           return false;

    }

     void setup() {

      // Initialize WiServer and have it use the sendMyPage function to serve pages

      WiServer.init(sendMyPage);

     // Enable Serial output and ask WiServer to generate log messages (optional)

      Serial.begin(57600);

      WiServer.enableVerboseMode(true);

      uart_gps.begin(GPSBAUD);

    }


    void getgps(TinyGPS &gps){

    }

    void loop(){

      // Run WiServer

      WiServer.server_task();

      // This is the main loop of the code. All it does is check for data on

      // the RX pin of the ardiuno, makes sure the data is valid NMEA sentences,

      // then jumps to the getgps() function.

      while(uart_gps.available())     // While there is data on the RX pin...
```

```
      {
        int c = uart_gps.read();    // load the data into a variable...
        if(gps.encode(c))      // if there is a new valid sentence...
        {
          getgps(gps);         // then grab the data.
        }
      }
    }
```

GPSki Objective-C Project:

GPSkiAppDelegate.h

```
//  GPSkiAppDelegate.h
//  GPSki
//
//  Created by Camila Dorin on 1/31/11.
//  Copyright 2011 __MyCompanyName__. All rights reserved.

#import <UIKit/UIKit.h>

@class GPSkiViewController;

@interface GPSkiAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    GPSkiViewController *viewController;

     NSMutableData *receivedData;
     UILabel *labelPointer;
     UILabel *distPointer;

     //create speeds, distance, latitudes, and longitudes arrays
     NSMutableArray *speeds;
     NSMutableArray *latitudes;
     NSMutableArray *longitudes;
     NSMutableArray *distances;

}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet GPSkiViewController *viewController;

@property (nonatomic, retain) IBOutlet NSMutableArray *speeds;
@property (nonatomic, retain) IBOutlet NSMutableArray *latitudes;
@property (nonatomic, retain) IBOutlet NSMutableArray *longitudes;
@property (nonatomic, retain) IBOutlet NSMutableArray *distances;


-(id)init;
-(void) createRequest;
-(void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse
*)response;
-(void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data;
-(void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error;
-(void)connectionDidFinishLoading:(NSURLConnection *)connection;
```

```
@end
```

GPSkiAppDelegate.m

```
//  GPSkiAppDelegate.m
//  GPSki
//
//  Created by Camila Dorin on 1/31/11.
//  Copyright 2011 __MyCompanyName__. All rights reserved.

#import "GPSkiAppDelegate.h"
#import "GPSkiViewController.h"
#import <UIKit/UIKit.h>

@implementation GPSkiAppDelegate

@synthesize window;
@synthesize viewController;
@synthesize speeds;
@synthesize latitudes;
@synthesize longitudes;
@synthesize distances;

#pragma mark -
#pragma mark Application lifecycle

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    // Add the view controller's view to the window and display.
    [self.window addSubview:viewController.view];
    [self.window makeKeyAndVisible];
     [self createRequest];
     [NSTimer scheduledTimerWithTimeInterval: 30.0 target: self
selector:@selector(createRequest)userInfo: nil repeats:YES];
    return YES;
}


- (void)applicationWillResignActive:(UIApplication *)application {
    /*
     Sent when the application is about to move from active to inactive state. This
can occur for certain types of temporary interruptions (such as an incoming phone call
or SMS message) or when the user quits the application and it begins the transition to
the background state.
     Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL
ES frame rates. Games should use this method to pause the game.
     */
}


- (void)applicationDidEnterBackground:(UIApplication *)application {
    /*
     Use this method to release shared resources, save user data, invalidate timers,
and store enough application state information to restore your application to its
current state in case it is terminated later.
     If your application supports background execution, called instead of
applicationWillTerminate: when the user quits.
     */
}
```

```
- (void)applicationWillEnterForeground:(UIApplication *)application {
    /*
     Called as part of  transition from the background to the inactive state: here you
can undo many of the changes made on entering the background.
     */
}


- (void)applicationDidBecomeActive:(UIApplication *)application {
    /*
     Restart any tasks that were paused (or not yet started) while the application was
inactive. If the application was previously in the background, optionally refresh the
user interface.
     */
}


- (void)applicationWillTerminate:(UIApplication *)application {
    /*
     Called when the application is about to terminate.
     See also applicationDidEnterBackground:.
     */
}

#pragma mark -
#pragma mark Memory management

- (void)applicationDidReceiveMemoryWarning:(UIApplication *)application {
    /*
     Free up as much memory as possible by purging cached data objects that can be
recreated (or reloaded from disk) later.
     */
}

- (void)dealloc {
    [viewController release];
    [window release];
    [super dealloc];
}

-(id)init
{
     self = [super init];

     if(self)
     {
          receivedData = [[NSMutableData alloc] init];
          labelPointer = [[UILabel alloc] init];
          distPointer = [[UILabel alloc] init];
          speeds = [[NSMutableArray alloc] init];
          latitudes = [[NSMutableArray alloc] init];
          longitudes = [[NSMutableArray alloc] init];
          distances = [[NSMutableArray alloc] init];


     }
     return self;
}
-(void) createRequest
{
     NSURLRequest *request =[NSURLRequest requestWithURL:[NSURL
URLWithString:@"http://192.168.1.55"]];
     [[NSURLConnection alloc] initWithRequest:request delegate:self];
```

```
}

- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse
*)response
{
    [receivedData setLength:0];
}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    [receivedData appendData:data];
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
{
    // Show error
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    // Once this method is invoked, "receivedData" contains the complete result
     NSString *labelText = [[NSString alloc] initWithData: receivedData encoding:
NSUTF8StringEncoding];
     NSLog(@"contents of the web page%@", labelText);

     //parse web page content
     NSArray *chunks = [labelText componentsSeparatedByString: @","];

     CGFloat speed_float = [[chunks objectAtIndex: 1] floatValue];
     //create pointer to speed UILabel in view controller class
     labelPointer = [viewController speed];
     labelPointer.text = [NSString stringWithFormat:@"%.2f",speed_float];

     //populate speeds, latitudes, and longitudes arrays
     [speeds addObject:[NSString stringWithFormat:@"%.2f",speed_float]];
     //NSLog(@"The content of speeds in appdelegate is%@",speeds);

     //update pointer to viewController array with new data for speeds
     [self.viewController updateWithDataPointer: speeds];

     //calculate distance
     NSUInteger i;
     //NSString *sp_string;
     //CGFloat sp;
     CGFloat dist;
     NSString *dist_string;
     for(i = 0; i < speeds.count; i++) {
          //NSLog(@"application is in distance loop");
          dist = speed_float*30/3600;
          dist_string = [NSString stringWithFormat:@"%.2f", dist];
          [distances addObject: dist_string];
          //NSLog(@"The content of distances is%@",distances);
     }

     //add all distances
     NSUInteger j;
     CGFloat dist_sum = 0;
     for (j = 0; j < distances.count; j++) {
          //NSLog(@"application is in sum distances loop");
          dist_sum += [[distances objectAtIndex: j] floatValue];
     }

     //create pointer to distance UILabel in view controller class
     distPointer = [viewController distance];
```

```
    distPointer.text = [NSString stringWithFormat:@"%.2f", dist_sum];

    //fill latitude and longitude array
    CGFloat latitude_float = [[chunks objectAtIndex: 2] floatValue];
    [latitudes addObject:[NSString stringWithFormat:@"%.5f",latitude_float]];
    //NSLog(@"The content of latitudes is%@",latitudes);

    CGFloat longitude_float = [[chunks objectAtIndex: 3] floatValue];
    [longitudes addObject:[NSString stringWithFormat:@"%.5f",longitude_float]];

}

@end
```

GPSkiViewController.h

```
//  GPSkiViewController.h
//  GPSki
//
//  Created by Camila Dorin on 1/31/11.
//  Copyright 2011 __MyCompanyName__. All rights reserved.

#import <UIKit/UIKit.h>

@interface GPSkiViewController : UIViewController {

    //define graph view subview
    IBOutlet CPGraphHostingView *graphView;

    //declare UILabels
    UILabel *appTitle;
    UILabel *speed;
    UILabel *speedTitle;
    UILabel *distance;
    UILabel *distanceTitle;
    UILabel   *weightTitle;
    UITextField *userWeight;
    UILabel *caloriesTitle;
    UILabel *calories;

    NSMutableArray *dataForPlot;

    NSMutableArray *speeds;

    CGFloat cal;

    //create graph object
    CPXYGraph *graph;
}

@property (nonatomic,retain) IBOutlet CPGraphHostingView *graphView;
@property (nonatomic,retain) IBOutlet UILabel *appTitle;
@property (nonatomic,retain) IBOutlet UILabel *speed;
@property (nonatomic,retain) IBOutlet UILabel *speedTitle;
@property (nonatomic,retain) IBOutlet UILabel *distanceTitle;
@property (nonatomic,retain) IBOutlet UILabel *distance;
@property (nonatomic,retain) IBOutlet UILabel *weightTitle;
@property (nonatomic,retain) IBOutlet UITextField *userWeight;
@property (nonatomic,retain) IBOutlet UILabel *caloriesTitle;
@property (nonatomic,retain) IBOutlet UILabel *calories;
@property (readwrite, nonatomic,assign) IBOutlet CGFloat cal;

@property(readwrite, retain, nonatomic) NSMutableArray *dataForPlot;
```

```
//allow for speeds to be used in the graph
@property(assign, nonatomic) NSMutableArray *speeds;

//-(void)initializeCal;
-(void)updateCalories;
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
-(void)updateWithDataPointer:(NSMutableArray *)dp;
-(void)addDataToPlot;


@end
```

GPSkiViewController.m

```
//  GPSkiViewController.m
//  GPSki
//
//  Created by Camila Dorin on 1/31/11.
//  Copyright 2011 __MyCompanyName__. All rights reserved.

#import "GPSkiViewController.h"

@implementation GPSkiViewController

@synthesize graphView;
@synthesize appTitle;
@synthesize speed;
@synthesize speedTitle;
@synthesize dataForPlot;
@synthesize distanceTitle;
@synthesize distance;
@synthesize speeds;
@synthesize userWeight;
@synthesize weightTitle;
@synthesize calories;
@synthesize caloriesTitle;
@synthesize cal;

-(void)initializeCal
{
     cal = 0.00;
     self.calories.text = [NSString stringWithFormat:@"%.2f", cal];
}

-(IBAction)updateCalories
{
     NSInteger weight = [userWeight.text integerValue];
     CGFloat time = 2.00;

     if (weight <= 130) {
          cal = cal + (295*time/60);
          self.calories.text = [NSString stringWithFormat:@"%.2f", cal];
     }
     else if (weight > 130 && weight <= 155)
     {
          cal = cal + (352*time/60);
          self.calories.text = [NSString stringWithFormat:@"%.2f", cal];
     }
     else if (weight > 155 && weight <= 180)
     {
          cal = cal + (409*time/60);
          self.calories.text = [NSString stringWithFormat:@"%.2f", cal];
     }
```

```
     else
     {
          cal = cal + (465*time/60);
          self.calories.text = [NSString stringWithFormat:@"%.2f", cal];
          NSLog(@"value of cal: %.2f", cal);
     }
}

//resign keyboard
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event

{
    [userWeight resignFirstResponder];

    [super touchesBegan:touches withEvent:event ];
}

//update speed data pointer
-(void)updateWithDataPointer:(NSMutableArray *)sp
{
     self.speeds = sp;
}

-
(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrient
ation
{
     return YES;
}

#pragma mark -
#pragma mark Initialization and teardown

-(void)dealloc
{
     [dataForPlot release];
    [super dealloc];
}

-(void)viewDidLoad
{
    [super viewDidLoad];

     [self.view addSubview:graphView];

     // Create graph from theme
     CGRect graphFrame = CGRectMake(0,400, 662, 440);
    graph = [[CPXYGraph alloc] initWithFrame:graphFrame];
     CPTheme *theme = [CPTheme themeNamed:kCPDarkGradientTheme];
    [graph applyTheme:theme];

     CPGraphHostingView *hostingView = (CPGraphHostingView *)self.graphView;
    hostingView.hostedGraph = graph;

     //paddings
    graph.paddingLeft = 0.0f;
    graph.paddingRight = 0.0f;
    graph.paddingTop = 0.0f;
    graph.paddingBottom = 0.0f;

     //define plot area
    graph.plotAreaFrame.paddingLeft = 50.0;
     graph.plotAreaFrame.paddingTop = 20.0;
     graph.plotAreaFrame.paddingRight = 20.0;
```

```
     graph.plotAreaFrame.paddingBottom = 40.0;

    // Setup plot space
    CPXYPlotSpace *plotSpace = (CPXYPlotSpace *)graph.defaultPlotSpace;
    plotSpace.allowsUserInteraction = NO;
    plotSpace.xRange = [CPPlotRange plotRangeWithLocation:CPDecimalFromFloat(0)
length:CPDecimalFromFloat(60)];
    plotSpace.yRange = [CPPlotRange plotRangeWithLocation:CPDecimalFromFloat(0)
length:CPDecimalFromFloat(50)];

     // Graph title
    graph.title = @"Speed vs Time";
    CPTextStyle *textStyle = [CPTextStyle textStyle];
    textStyle.color = [CPColor whiteColor];
    textStyle.fontSize = 18.0f;
    graph.titleTextStyle = textStyle;
    graph.titleDisplacement = CGPointMake(0.0f, -20.0f);
    graph.titlePlotAreaFrameAnchor = CPRectAnchorTop;

     textStyle.fontSize = 8.0f;
    // Setup axes and labels
     CPXYAxisSet *axisSet = (CPXYAxisSet *)graph.axisSet;
    CPXYAxis *x = axisSet.xAxis;
    x.majorIntervalLength = CPDecimalFromString(@"10");
    x.orthogonalCoordinateDecimal = CPDecimalFromString(@"0");
     x.title = @"Time (min)";
     x.titleOffset = 20.0f;
     x.titleLocation = CPDecimalFromFloat(30.0f);

    CPXYAxis *y = axisSet.yAxis;
     y.majorIntervalLength = [[NSDecimalNumber decimalNumberWithString:@"10"]
decimalValue];
    y.minorTicksPerInterval = 4;
     y.minorTickLength = 10.0f;
     y.majorTickLength = 15.0f;
    y.orthogonalCoordinateDecimal = CPDecimalFromString(@"0");
     y.title = @"Speed (km/h)";
     y.titleOffset = 25.0f;
    y.titleLocation = CPDecimalFromFloat(5.0f);

     // Create a blue plot area
     CPScatterPlot *boundLinePlot = [[[CPScatterPlot alloc] init] autorelease];
    boundLinePlot.identifier = @"Speed Plot";
     boundLinePlot.dataLineStyle.miterLimit = 1.0f;
     boundLinePlot.dataLineStyle.lineWidth = 3.0f;
     boundLinePlot.dataLineStyle.lineColor = [CPColor blueColor];
    boundLinePlot.dataSource = self;
     [graph addPlot:boundLinePlot];

     // Do a blue gradient
     CPColor *areaColor1 = [CPColor colorWithComponentRed:0.3 green:0.3 blue:1.0
alpha:0.8];
    CPGradient *areaGradient1 = [CPGradient gradientWithBeginningColor:areaColor1
endingColor:[CPColor clearColor]];
    areaGradient1.angle = -90.0f;
    CPFill *areaGradientFill = [CPFill fillWithGradient:areaGradient1];
    boundLinePlot.areaFill = areaGradientFill;
    boundLinePlot.areaBaseValue = [[NSDecimalNumber zero] decimalValue];

     // Add plot symbols
     CPLineStyle *symbolLineStyle = [CPLineStyle lineStyle];
     symbolLineStyle.lineColor = [CPColor blackColor];
     CPPlotSymbol *plotSymbol = [CPPlotSymbol ellipsePlotSymbol];
     plotSymbol.fill = [CPFill fillWithColor:[CPColor blueColor]];
```

```
     plotSymbol.lineStyle = symbolLineStyle;
    plotSymbol.size = CGSizeMake(10.0, 10.0);
    boundLinePlot.plotSymbol = plotSymbol;


    //add data to plot every minute and a half
     [NSTimer scheduledTimerWithTimeInterval:90.0 target:self
selector:@selector(addDataToPlot) userInfo:nil repeats:YES];

     //update calories burned every 2 minutes
     [NSTimer scheduledTimerWithTimeInterval:5.0 target:self
selector:@selector(initializeCal) userInfo:nil repeats:NO];
     [NSTimer scheduledTimerWithTimeInterval:120.0 target:self
selector:@selector(updateCalories) userInfo:nil repeats:YES];

#ifdef PERFORMANCE_TEST
    [NSTimer scheduledTimerWithTimeInterval:2.0 target:self
selector:@selector(changePlotRange) userInfo:nil repeats:YES];
#endif
}

-(void)addDataToPlot
{
     // Add data
     NSMutableArray *contentArray = [NSMutableArray arrayWithCapacity:100];
     NSUInteger i;
     //NSUInteger j = 0;
     id x;
     id y;
     for ( i = 0; i < speeds.count; i++ ) {
          x = [NSNumber numberWithFloat:i];
          if (i*2 < speeds.count) {
               y = [speeds objectAtIndex:i*2];
          }
          else {
               y = [speeds objectAtIndex:i];
          }
          [contentArray addObject:[NSMutableDictionary dictionaryWithObjectsAndKeys:x,
@"x", y, @"y", nil]];
     }
     self.dataForPlot = contentArray;
     [graph reloadData];
}

-(void)changePlotRange
{
    // Setup plot space
    CPXYPlotSpace *plotSpace = (CPXYPlotSpace *)graph.defaultPlotSpace;
    plotSpace.xRange = [CPPlotRange plotRangeWithLocation:CPDecimalFromFloat(0.0)
length:CPDecimalFromFloat(3.0 + 2.0*rand()/RAND_MAX)];
    plotSpace.yRange = [CPPlotRange plotRangeWithLocation:CPDecimalFromFloat(0.0)
length:CPDecimalFromFloat(3.0 + 2.0*rand()/RAND_MAX)];
}

#pragma mark -
#pragma mark Plot Data Source Methods

-(NSUInteger)numberOfRecordsForPlot:(CPPlot *)plot {
    return [dataForPlot count];
}

-(NSNumber *)numberForPlot:(CPPlot *)plot field:(NSUInteger)fieldEnum
recordIndex:(NSUInteger)index
{
```

```
    NSNumber *num = [[dataForPlot objectAtIndex:index] valueForKey:(fieldEnum ==
CPScatterPlotFieldX ? @"x" : @"y")];

    return num;
}

@end
```