AERE 361

Computational Techniques for Aerospace Design

Laboratory Manual

Spring 2021

Prof. Matthew E. Nelson
Teaching Assistants
Hang Li

Contents

1 | Lab 10 - API's, Binary Data and graphing

Reference:
C Programming Language, 2nd Edition
GNU Plot Manual: http://www.gnuplot.info/documentation.html FIT Protocol Information: https://developer.garmin.com/fit/protocol/

1.1 Pre-Lab

1.1.1 Background

1.1.2 Objectives

- Gain experience working with non-text data files.
- Gain experience working with a software developer kit (SDK).
- Gain experience with GNUPlot
- Gain experience in reading and writing files
- Think about how we can approach a problem and use software to solve it

1.1.3 Methodology

For Lab 10, students can opt to complete the lab in a couple of ways. In Lab 1, you were introduced to Repl.IT and using the online editor. This allows you to clone your repo to Repl.IT and push changes to your develop branch. Repl.IT also has a nice feature where the graders and TA can also view your Repl.IT so we can help you. We have designed the lab with Repl.IT in mind.

Students could use other sources. The HPC also has GCC and could be used. Other IDE's could be used. However, a word of caution. While it is true you can use these, you are on your own. The support that myself, my TA and graders can give you will be limited. You have been warned.

To Do:
In your lab report, under the methodology section, explain what you used to work on your programming. Don't forget the objectives, which you can copy and past from this write-up. That will conclude the Pre-Lab section of the report.

Deliverable	Points
Program Compiles	10
Program executes	5
Program generates data files	10
Proper Makefile	5
Graphs generated	20
Report	70
Total Score	**120**

Table 1.1: Rubric for Lab 9

1.2 Grading

There are 3 exercises in this lab. Your first step will be looking at the provided `decode.c` file and making the needed changes. I would suggest that you then start creating your Makefile and use that to make sure you are compiling correctly. Once you have the data decoded and stored in the needed files, make sure to read about GNUPlot and and create your GNUPlot script. Add to your Makefile to be able to run your Makefile that compiles, run and generates the graphs you need. Grading is a little different for this lab. We are grading at various checkpoints. The program compiling without warnings or errors is the first one, from there we are checking execution and finally your files. With this lab, we are putting a few points in your report. You will need to copy the generated graphs and include them in your report.

1.2.1 Report

In the report folder you will find a starter "main.tex" file. Like the last lab, this template is very barren as you are expected to fill it out. At this point, you are now expected to read the lab writeup and when asked, fill out the appropriate section.

> *Be Careful:*
> Make sure your report compiles. It should not only compile in Overleaf but needs to compile using our LaTeX builder script in GitHub. We will always grade what the GitHub action generates. This happens during a pull request and can be triggered manually.

1.3 Lab Work

1.3.1 Getting Started

To start with, go to the link found in Canvas for this lab. Click on it, and then clone the repo. You should have a an icon for Repl.IT in your Readme.MD file. You can click on that to clone this to Repl.IT and start working. This is the same process you did for Lab 1.

> *Be Careful:*
> Don't forget that your first step is to create a new branch called "develop". You will do all of your programming work in this branch.

If you decide to not use Repl.IT, then you can use the normal *git* commands to pull the repo, create a branch, and do all of your commits. Any computer running GCC and has the Git CLI tools installed should work. But, as we stated, you are own your own with this.

1.4 Data files

CSV files are commonly used as they are nice and simple text files that are easy to work with and are human readable. This makes confirming information fairly easy as you can also open the file with any text editor. But CSV files have two major drawbacks. First, text files that are readable like a CSV often take up more space both in memory and storage. Second, because of this overhead, CSV files also require more processing, especially when it comes to creating the files.

If you have a system that is processing a lot of data, then writing this data to a text based file can be a bottleneck for that system. For this reason, a lot of systems use binary file formats. This can sometimes be problematic as it means the file is not in a human readable format. It can also sometimes mean that the file format is proprietary or closed source. But, this is not always the case, there are binary formats that are fully documented which means that others can write software to both read and write these file formats. While you still lose the ability to read these files by a human (without software help), you gain in performance improvements.

The Flexible & Interoperable Data Transfer (FIT) protocol is one such file format that is fully documented and is often used with devices that use the ANT and ANT+ wireless protocol. ANT and ANT+ is used in a number of Garmin devices like their sport watches, bike computers and action cameras. We are not going to do anything with ANT or ANT+, but these devices will store their data using the FIT protocol and this is what we will work with in this lab.

What is nice with FIT is that they fully document the protocol and also provide a Software Developers Kit (SDK). SDKs are often used in industry where a company has developed a protocol or format and wish to have others write software that uses their protocol or format. You should think of a SDK as a set of libraries like you learned in Lab 7 and 8. They also often provide example code which can aid the developer to quickly learning how to use their libraries.

For this lab you are going to read a FIT file that is provided using the SDK. The SDK that Garmin provides has examples for C, C++, C# and Java. Of course, we will be using C for this lab. We will provide the FIT file and also a slightly modified version of the example code they have provided.

1.4.1 About the data

The data you have been given is a FIT file recorded from a Garmin Virb Ultra 30 action camera. The Garmin Virb has a GPS, accelerameter, rate gyro, magnetometer, and pressure sensor built in. This data was recorded by your instructor and this is his personal camera. In addition, the data was recorded while your instructor was out flying, so the data is actual data from a flight in his Piper Cherokee 180. To make things interesting, a flight path was used to do some "virtual" sky writing.

> *Note:*
> For this lab, we are using a Software Developer's Kit or SDK. SDKs are common and are there to provide information, examples and tools that developers can use to rapidly develop a software solution. The files included in this lab include these SDK files. We have already made any adjustments to these files to make sure they compile in the Linux environment and work with the data we have provided. You should not modify any of the other files, only the ones we tell you to modify. More on the SDK next.

1.4.2 About the SDK

The SDK provided gives us libraries that will allow us to decode the FIT file that we have provided. As mentioned before, this file is a binary format, so we cannot simply read the file in, we need to understand how the binary file is organized in order to be able to decode it. The SDK provides a number of libraries that can be used to both read and write in the FIT file protocol. Because the FIT protocol can be used with a number of devices, it also includes libraries to help parse only the data that is needed which can also be

determined by the model of the device used. The SDK includes a CSV file that is used with a file generation program. This allows the developer to select which messages should be included and will write this to the fit_example.c and it's header file.

For this lab, this has been done for you. The instructor has generated the correct files to make sure that the correct records are retrieved. However, if you wanted to modify this for a device that, for example, has a heart rate monitor, that can be done. We did not include the full SDK since we had done this step for you and it included extra code for C++, C# and Java. However, you can download the full SDK from https://www.thisisant.com/resources/fit-sdk/.

The SDK provides us with a number of files that are used to read and parse the FIT data file we have provided. The list of files are listed below.

```
decode.c
fit.c
fit.h
fit_config.h
fit_convert.c
fit_convert.h
fit_crc.c
fit_crc.h
fit_example.c
fit_example.h
fit_include.h
fit_product.c
fit_product.h
fit_ram.c
fit_ram.h
```

All files that begin with *fit* are files that are part of the SDK. The *decode.c* file is an example file that the SDK provides us. We have modified this file for this lab. This is the file that you will work with. You do not need to (and you should not) modify any of the other files.

1.5 Decoding a FIT File

Now that we have talked about the FIT protocol and the SDK a little, let's dive in on decoding our FIT file that we have. In this section we will break down the protocol and then examine the libraries that the SDK gives us. We will examine the example code and look at how the example code is reading a file. We will then use their example code to print to the screen some, but not all, of the data in this file.

1.5.1 FIT Protocol

The SDK provides some excellent documentation that we can use to better understand this protocol. Below is some excerpts from the SDK documentation, specifically the D00001275 document. This document covers the basics on the FIT protocol. Items that are a direct qoute from the documentation are in the boxes.

> A FIT file contains a series of records that, in turn, contain a header and content. The record content is either a definition message that is used to specify upcoming data, or a data message that contains a series of data-filled fields as shown in Figure 1.1. The FIT protocol defines the type and content of messages, the data format of each message's field, and methods of compressing data (if applicable).

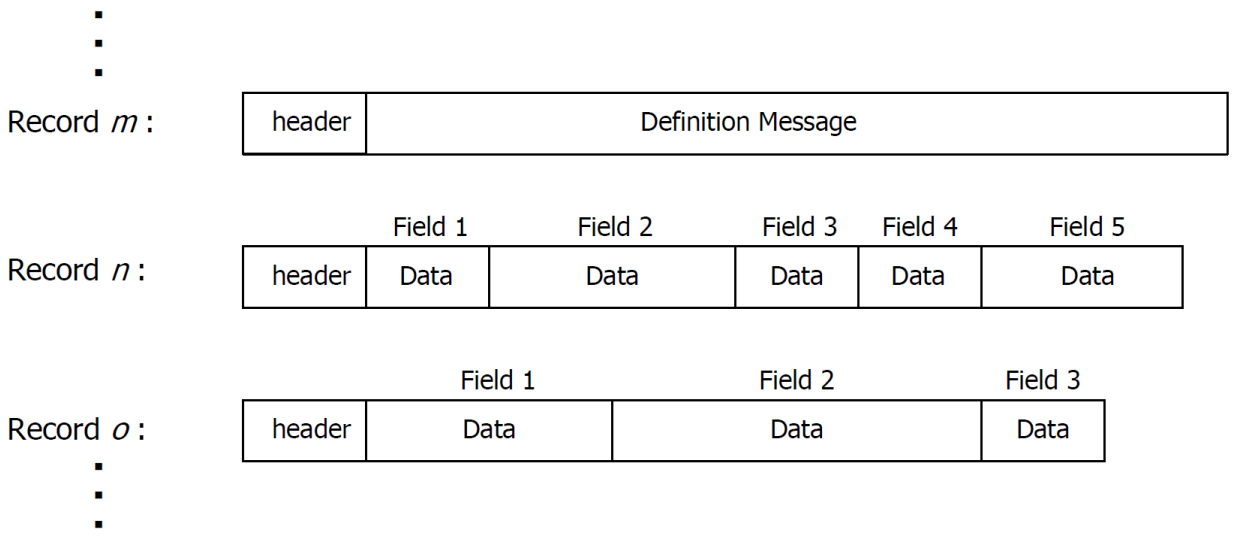

Figure 1.1: Basic FIT File components

A FIT file is basically a collection of records. Each record then contains a header and then the content. As you can see in Figure 1.1, we can have a definition message, that indicates what the next record has or the actual data. The FIT format is meant to be flexible as it can support a wide range of data types, so the fields for that data is not fixed unlike some other protocols. Let's look more closely how the FIT protocol is made up.

The FIT protocol defines the process for which profiles are implemented and files are transferred. Figure 1.2 provides an overview of the FIT process. Typically, ANT+ broadcast data is collected by a display device. The display device would then encode the data into the FIT file format according to its product profile (i.e. product profile 1). The FIT file is then transferred to another device which would then decode the received files according to its own implemented product profile (i.e. product profile 2).

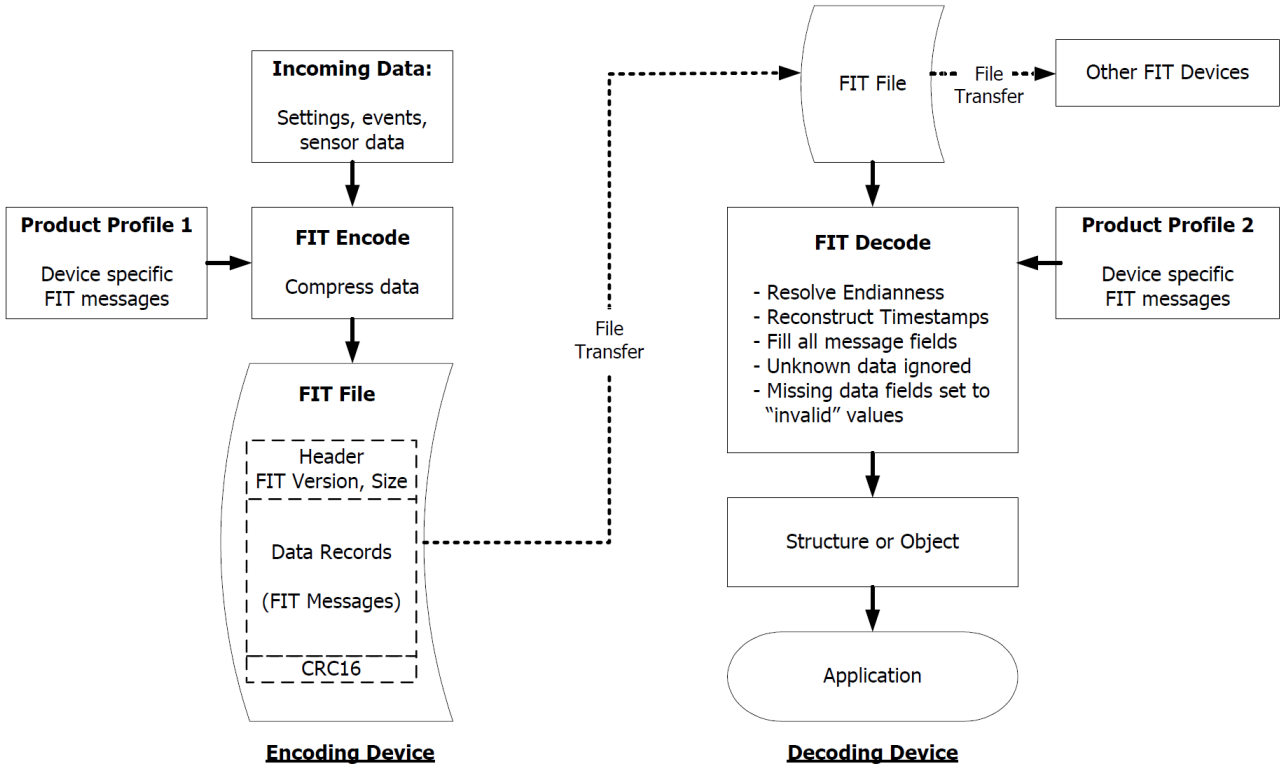

Figure 1.2: Overview of the FIT file protocol

The data collected by the device are written into the message fields and this is defined by the device's product profile. That same product profile is then used to decode it. The SDK that they provide will take care of the timestamp information and the message fields for us. As long as we have configured the product profile, then we should be able to retrieve the fields that we want. We have done some of this for you that is found in the `decode.c` file provided. Let's look more on how this works.

1.5.2 Decoding and parsing the FIT file

Let's take a look at this file and see what this file is doing.

```
1  #define _CRT_SECURE_NO_WARNINGS
2  //INCLUDES
3  #include "stdio.h"
4  #include "string.h"
5  #include "math.h"
6  #include "fit_convert.h"
7
8  //MAIN FUNCTION
9
10 int main(int argc, char* argv[])
```

None of this should look odd to anyone at this point. We have discussed libraries and we are also using some of the common libraries used in previous labs. The *fit_ convert.h* of course starts to pull in the helper libraries that makes it easier to work with these file. Also notice our *main* function. We are passing in a file name of the file that we will decode. Let's look further on in the code.

```
1      FILE* file;
2      //File pointer for our file used for writting data to
3      FILE* fp = NULL;
4      FILE* fp2 = NULL;
```

We have three file pointers. Our first one is for the file we are reading in. We will also create two files that we will work with for graphing later. The first file will log our GPS data including latitude, longitude, altitude, ground speed, and heading. The second file will log our attitude information which includes our pitch, roll and yaw. We provide the two file pointers you can use. You will need to write the code to initialize the two files we will use to write to. One reason for this is that you want the code to completely write over any old file that might be from a previous run.

```
        // The following is code used by the SDK - DO NOT MODIFY
        #if defined(FIT_CONVERT_MULTI_THREAD)
                FIT_CONVERT_STATE state;
        #endif

    if (argc < 2)
    {
        printf("usage: decode.exe <fit file>");
        return FIT_FALSE;
    }

    printf("Testing file conversion using %s file...\n", argv[1]);

        #if defined(FIT_CONVERT_MULTI_THREAD)
                FitConvert_Init(&state, FIT_TRUE);
        #else
                FitConvert_Init(FIT_TRUE);
        #endif

    if ((file = fopen(argv[1], "rb")) == NULL)
    {
        printf("Error opening file %s.\n", argv[1]);
        return FIT_FALSE;
    }
    printf("Processing...\n");
```

The next part will check the FIT file itself and read it in. Notice that this program is designed to work by passing in a command line argument which you should have noticed when looking at the *main* function. If the file is valid and it can read it in, it will continue to the while loop where we will start working with the data. Let's take a look at that now.

```
    while (!feof(file) && (convert_return == FIT_CONVERT_CONTINUE))
    {
        for (buf_size = 0; (buf_size < sizeof(buf)) && !feof(file); buf_size++)
        {
            buf[buf_size] = (FIT_UINT8)getc(file);
        }

        do
        {
            #if defined(FIT_CONVERT_MULTI_THREAD)
                convert_return = FitConvert_Read(&state, buf, buf_size);
            #else
                convert_return = FitConvert_Read(buf, buf_size);
            #endif
```

If we have not hit the end of file, we continue to read in the buffer from the file and read in each character. Again, the data is in binary form, so we call the *FitConvert_Read* function to read the binary format and convert it to a character string. From here, we can build up our message that we want to look at in more depth.

We need to decide what to do with the data. If you looked at the FIT protocol, you might have noticed that we can a variety of different messages. We are only looking for certain ones. So we will look for those and only act on certain messages. To do this, we can use a switch statement. Let's take a look at the first one provided.

```
            switch (convert_return)
            {
```

```
            case FIT_CONVERT_MESSAGE_AVAILABLE:
            {
                #if defined(FIT_CONVERT_MULTI_THREAD)
                        const FIT_UINT8* mesg = FitConvert_GetMessageData(&state);
                        FIT_UINT16 mesg_num = FitConvert_GetMessageNumber(&state);
                #else
                        const FIT_UINT8* mesg = FitConvert_GetMessageData();
                        FIT_UINT16 mesg_num = FitConvert_GetMessageNumber();
                #endif
                //printf("Mesg Num: %d\n",mesg_num);
                printf("Processing Message # (ID): %d (%d) \n", mesg_index++, mesg_num);
```

Our first thing we will check is if a valid message is available. If we do, we can identify the message number. With that message number, we now need to add a second switch statement that will start checking that information. Based on that, we will either handle the GPS data, the aircraft attitude information or we will add a default statement that will just state that the data is ignored.

```
                switch (mesg_num)
                {
                        case FIT_MESG_NUM_FILE_ID:
                        {
                                const FIT_FILE_ID_MESG* id = (FIT_FILE_ID_MESG*)mesg;
                                printf("File ID: type=%u, number=%u\n", id->type, id->number
                                    );
                                break;
                        }
```

The code above starts our switch statement based on the message number that is being passed in. Again, we will look for certain message numbers. However, we do not need to know the actual numbers. The helper function uses constants and therefore we can simply call those constants to match up with the number coming in. Based on that, we have different switch statements based on the message number. The first case statement only happens at the very beginning of the file. This identifies the file id and number. Now, we want to start working with other message numbers, so we need to create additional case statements. Let's look at the first one we started for you.

```
                        case FIT_MESG_NUM_AVIATION_ATTITUDE:
                        {
                                fp2 = fopen("attitude_log.txt", "a");
                                const FIT_AVIATION_ATTITUDE_MESG* attitude = (
                                    FIT_AVIATION_ATTITUDE_MESG*)mesg;
                                roll_data = ((attitude->roll[0]) / 10430.38) * (180 / 3.14);
                                //Insert your code here to decode the representations

                                break;
                        }
```

The code above is starter code for the aviation attitude information that is stored in the file. This information is derived from the rate gyro and accelerometer sensors that are on-board. Although we can access the raw data, it requires us to convert the raw Analog to Digital Converter (ADC) values as well as calibrate the data as well. While your instructor thought that would be a fun exercise, he figured his students wouldn't enjoy it as much as he would. So, we will just take advantage of data that is already converted from ADC values and has been calibrated.

Looking at the code again, we define a constant that will point to the aviation attitude helper that the SDK provides for us. We will store this in a variable called attitude. From this pointer, we can now access a variety of data that is available. If we look into the *fit_example.h* file, we can actually see what these are. Below is a listing from the header, somewhat edited to fit better in this write-up.

```
    typedef struct
{
   // 1 * s + 0, Timestamp message was output
   FIT_DATE_TIME timestamp;
```

```
    // 1 * ms + 0, System time associated with sample expressed in ms.
    FIT_UINT32 system_time[FIT_AVIATION_ATTITUDE_MESG_SYSTEM_TIME_COUNT];
    // 1 * ms + 0, Fractional part of timestamp, added to timestamp
    FIT_UINT16 timestamp_ms;
    // 10430.38 * radians + 0, Range -PI/2 to +PI/2
    FIT_SINT16 pitch[FIT_AVIATION_ATTITUDE_MESG_PITCH_COUNT];
    // 10430.38 * radians + 0, Range -PI to +PI
    FIT_SINT16 roll[FIT_AVIATION_ATTITUDE_MESG_ROLL_COUNT];
    // 100 * m/s^2 + 0, Range -78.4 to +78.4 (-8 Gs to 8 Gs)
    FIT_SINT16 accel_lateral[FIT_AVIATION_ATTITUDE_MESG_ACCEL_LATERAL_COUNT];
    // 100 * m/s^2 + 0, Range -78.4 to +78.4 (-8 Gs to 8 Gs)
    FIT_SINT16 accel_normal[FIT_AVIATION_ATTITUDE_MESG_ACCEL_NORMAL_COUNT];
    // 1024 * radians/second + 0, Range -8.727 to +8.727 (-500 degs/sec to +500 degs/sec)
    FIT_SINT16 turn_rate[FIT_AVIATION_ATTITUDE_MESG_TURN_RATE_COUNT];
    // 10430.38 * radians + 0, Track Angle/Heading Range 0 - 2pi
    FIT_UINT16 track[FIT_AVIATION_ATTITUDE_MESG_TRACK_COUNT];
    FIT_ATTITUDE_VALIDITY validity[FIT_AVIATION_ATTITUDE_MESG_VALIDITY_COUNT]; //
    FIT_ATTITUDE_STAGE stage[FIT_AVIATION_ATTITUDE_MESG_STAGE_COUNT]; //
    FIT_UINT8 attitude_stage_complete[FIT_AVIATION_ATTITUDE_MESG_ATTITUDE_STAGE_COMPLETE_COUNT];
} FIT_AVIATION_ATTITUDE_MESG;
```

There are three values we are interested in, *pitch*, *roll*, and *turn_rate*. Note the comments added. In the header file these are off to the side, but to make it easier to read here I put them above the declaration of the values. These comments are giving vital information on how to handle this data. Let's take a look at *roll*. If we write that in equation form we get what is seen in Equation 1.1.

$$10430.38 * Roll_{radians/sec} + 0 = Roll_{Raw} \tag{1.1}$$

So, to convert this value to a radians per second, we need to solve for the $Roll_{radians/sec}$. We can see this in Equation 1.2.

$$Roll_{radians/sec} = \frac{Roll_{Raw}}{10430.38} \tag{1.2}$$

So, to obtain our roll data, we need to first pull that information and then convert that value to what we want. We are also going to require you to convert it from radians per second to degrees per second, but that should be easy. Let's look at the example line we provided you.

```
1 ||                    roll_data = ((attitude->roll[0]) / 10430.38) * (180 / 3.14);
```

The *attitude->roll[0]* pulls the data and in this case we are just selecting the first element in the array. The -> that you see dereferences the struct that is created by the SDK. Recall that in the header we looked at, the structure includes a FIT_SINT16 roll. The first part is just assigned the variable type, in this case a 16-bit signed integer. The roll[] is our array that we will use. Again, we are not done yet as we need to convert this value to degrees per second. This is done for you as an example. Now, you should do the same to extract the information for the *pitch* and *turn_rate*.

We want to store this information into a file that we can use later to graph with GNUPlot. GNUPlot will accept a variety of data files. For this lab, we want you to store this information in a tab delimited format. Store the information four columns. You should have in the first column the message index (which is just the variable *mesg_index*. We are storing the message index to use for graphing later. After that place your pitch data in the next column (which is just hitting tab again) and then roll and finally yaw. Remember, we provided the file pointer *fp2* and store this information in a file called "attitude_log.txt".

```
1 ||                 case FIT_MESG_NUM_GPS_METADATA:
```

The next case statement we need to work with is to handle the GPS data we have available. Like the previous statement, we will define our constant that will point to our data handler for the GPS Metadata that is provider. You can see that line of code above. We are looking for the following data:

- Latitude
- Longitude
- Altitude (called enhanced_altitude)
- Speed (which we will call ground speed)
- Heading

Just like before, we can look at the header files to find out what data is available to us and also if we need to convert them and if so how. Let's look at what is in the header file now.

```
    typedef struct
{
   FIT_SINT32 position_lat; // 1 * semicircles + 0,
   FIT_SINT32 position_long; // 1 * semicircles + 0,
   FIT_UINT32 enhanced_altitude; // 5 * m + 500,
   FIT_UINT32 enhanced_speed; // 1000 * m/s + 0,
   FIT_UINT16 heading; // 100 * degrees + 0,
   // 100 * m/s + 0, velocity[0] is lon velocity.  Velocity[1] is lat velocity.
   Velocity[2] is altitude velocity.
   FIT_SINT16 velocity[FIT_GPS_METADATA_MESG_VELOCITY_COUNT];
} FIT_GPS_METADATA_MESG;
```

It should be clear which one lines up with what data we are working with. Just like above, you will need to dereference these to get the value. The difference here though is that none of the ones that we will be using are an array. Except for the heading, they are all signed 32-bit integer values.

We do need to convert the values again and comments once again tell us how to do that. The altitude data is in meters and the conversion is fairly easy. Again, remember you need to solve for m. Same thing for enhanced speed and heading. I have included the equations for these in Equations 1.3, 1.4, 1.5. These will give you results with the altitude in feet, the speed in knots and the heading in degrees.

$$Altitude_{feet} = \left(\frac{enhanced_{altitude}}{5} - 500\right) * 3.28 \tag{1.3}$$

$$Speed_{knots} = \frac{enhanced_{speed}}{1000} * 1.944 \tag{1.4}$$

$$Heading = \frac{gps_{heading}}{100} \tag{1.5}$$

If, the latitude and longitude look different, you are right. I talk about this more in depth in lecture, but I will give a quick summary here. Recall that working with integers is often faster and takes up less space. For that reason, the latitude and longitude is stored as integers. Converting these at first may seem strange, but it is fairly easy. Equation 1.6 will help out in converting both latitude and longitude.

$$Latitude = position\_lat * \frac{180}{2^{31}} \tag{1.6}$$

This works for both the Latitude and Longitude. It should go without saying that you will need to have the appropriate variables declared. I would suggest doubles and I have tried to help out by declaring these at the beginning of the code. Of course if you don't like those, well you do you then.

Like the attitude information, we also want to store this information in a tab delimited file. Call this file "gps_log.txt". Again, store the message index like you did in the attitude file. Then store longitude, latitude, altitude, ground speed and finally heading. With that we are all done...well, not quite. We have one small issue we need to take care of.

Unfortunately, not all of the data is completely error free. There are some points in the data where the GPS did not have a proper lock and we see some weird latitude, longitude or altitude data. We need to filter these out. Fortunately, there is an easy fix and again, your instructor spent the time to track this down and come up with an easy solution.

The solution is that any data that is not valid has a very high altitude value. The Piper Cherokee 180 that your instructor flies has a maximum ceiling of 16,000 feet. So any data above that will not be valid. So, before you write your data to your file, you should check for that condition before storing the data in your text file.

With that you are done (told you this wouldn't be too hard). The rest of the file is checking for some other messages like the timestamp which we are not going to bother with. When you run this, you will see some *Unknown* messages pop up. This is due to a few message types that we have not told the program to handle. These are fine and will be ignored and should not be written to your files (if you did everything properly).

1.6 Introduction to gnuplot

Gnuplot is an application that generate plots very much like MatPlotLib used in Python or using plot in Matlab. It offers several customizable features and is very easy to use. In the first part of this lab, we focus on basic plotting with `gnuplot`. To start the interactive use of gnuplot, type `gnuplot` at the terminal. As the first step, we want to set how we want the plots to be generated.

For this lab instructions, we use `.svg` as the default output for our plots. Why `.svg`? SVG is a vector format (like PDF). Vector formats are preferable because you can scale the image without losing quality in the image. If you use a PNG, JPG or even BMP file format, scaling the image results in an image that may not look correct. Scaling one of these formats to a large scale often results in a pixelated image. In both cases, the image will not look as crisp or detailed.

```
gnuplot> set terminal svg enhanced
gnuplot> set output 'plot.svg'
```

The above command sets the desired output format to `.svg` and the name of our generated plot will be `plot.svg`.

1.6.1 Basic Plotting

In our first graph we want to plot a sine and a cosine functions. Therefore, we specify our functions and plot them.

```
gnuplot> a = 1.3
gnuplot> f(x) = a * sin(x)
gnuplot> g(x) = a * cos(x)
gnuplot> plot [0:10] f(x) title 'sin(x)' with lines linestyle 1, \
>                    g(x) notitle w l ls 2
```

The definitions of functions in gnuplot are straight forward. We want to plot more than one function that is why we have to divide the two commands with a comma. The backslash tells gnuplot that we have a line break at this position. We can also abbreviate commands ("w" for with, etc.). The result of the command is shown in Figure 1.3.

We can also gnuplot in non interactive mode by saving the plotting commands to a file. For example, to plot the sine and cosine functions, create a new file named `simpleplots.gp` with the following contents.

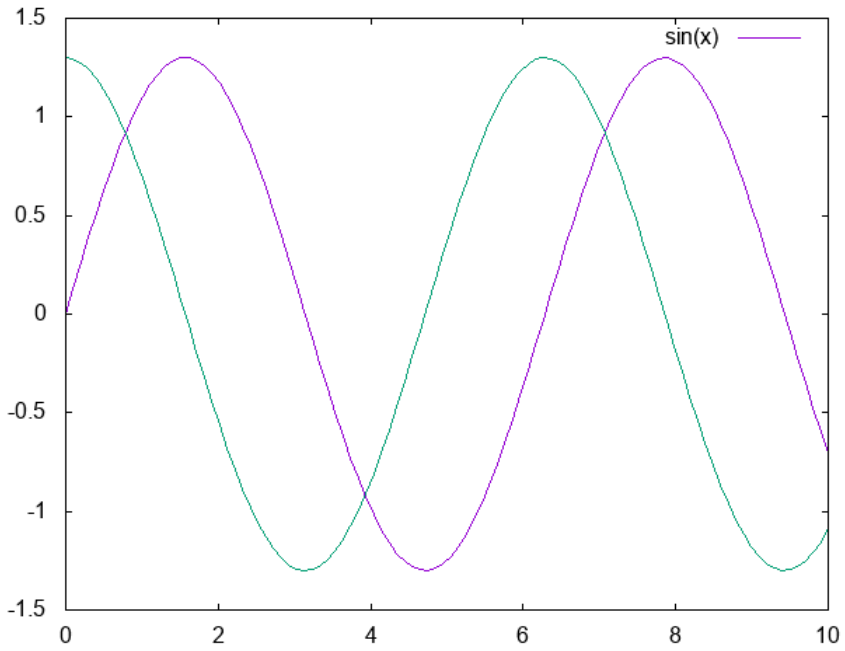

Figure 1.3: Plot for sine and cosine functions

```
a = 0.9
b = 1.3
f(x) = a * sin(x)
g(x) = b * cos(x)
plot [0:10] f(x) title 'sin(x)' with lines linestyle 1, \
g(x) title 'cos(x)' w l ls 2
```

Run `gnuplot` with the above file by calling the below command from the terminal.

```
gnuplot simpleplots.gp
```

1.6.2 Plotting Data

Plotting data, e.g., results of calculations or measurements, using `gnuplot` works the same way as plotting functions. We need a data file and commands to manipulate the data. Let's start with the basic plotting of simple data. A data file can be a plain text file containing the data points as columns. Create a file with the data below and save it as `dataplot.dat`

```
# dataplot.dat
#   X   Y   Z
    1.  1.  1.
    2.  4.  8.
    3.  9.  27.
    4.  16. 64.
    5.  25. 125.
```

A line starting with '#' is a comment and is ignored by gnuplot. A command to plot the data in the files is

```
gnuplot> plot 'dataplot.dat' using 1:2 with linespoints, \
>             '' u 1:3 w lp
```

The above command will generate a plot as shown in Figure 1.4.

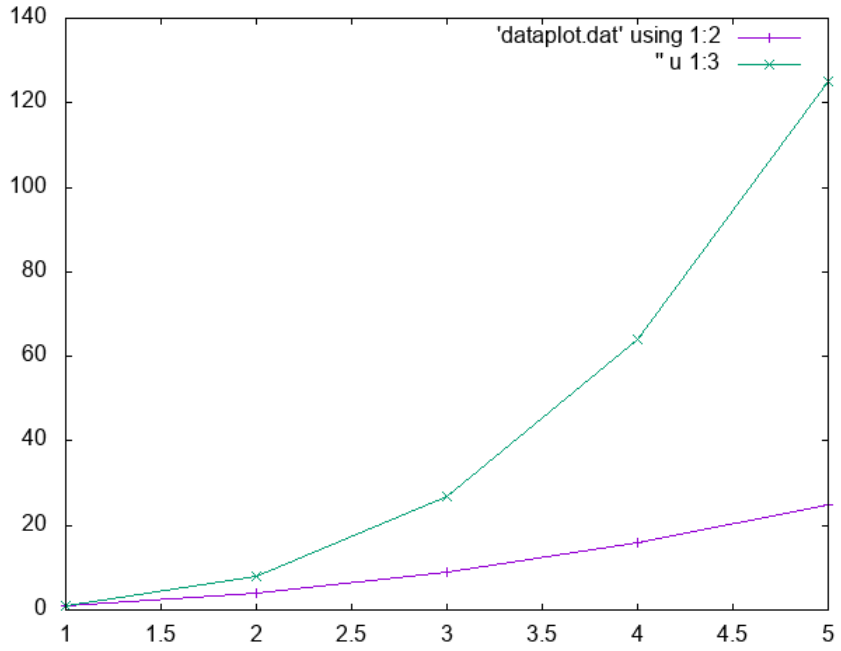

Figure 1.4: Plot for data in `dataplot.dat`

For advanced usage of `gnuplot` and available options, please refer to https://www.usm.uni-muenchen.de/CAST/talks/gnuplot.pdf. Typing `help` at the gnuplot prompt is also very useful.

1.6.3 `gnuplot` Hints

- gnuplot can print text-based pictures to your terminal - useful for monitoring a running program or development
- The slides linked above have an example of using color maps
- By exporting snapshots of the data being plotted, then combining the images, you can make animation of your data!

1.6.4 Graphing the data

Now that we have our data, we want to do some additional analysis with the data. There is roughly around 28,000 data points in the GPS log file and 3,800 in the attitude log file. While we could open the data and look at it in a text editor, it doesn't mean any thing to us yet. For that reason, a very common way to look at data is by graphing it. With graphing we can look at trends and very easily see outliers in the data.

As we talked about earlier in this write-up, GNUPlot can be used to plot data. The lab write-up takes you through a basic example and I would encourage you to try the example there. Go ahead and read that part since you probably skipped over that. It's fine, I'll wait. All good? Great, let's move on.

The first thing we want you to do is to graph the attitude information that we collected and that you stored in your "attitude_log.txt" file. Remember that you have a message index so we can use this for our x-axis. Now we can plot our data but we will do things a little bit differently from the example. Let's walk through this.

```
set terminal svg enhanced
```

```
    set output 'attitude.svg'
    set title "Roll/Pitch/Yaw Plot"
    set xlabel "Message Index"
    set ylabel "Degrees/Second"
    plot 'attitude_log.txt' u 1:2 with lines lw 1 lc rgb 'blue' ti \
    'Pitch','attitude_log.txt' u 1:3 with lines lw 1 lc rgb 'red' \
    ti 'Roll','attitude_log.txt' u 1:4 with lines lw 1 lc rgb 'green' ti 'Yaw'
reset
```

First thing is that we are not going to use PNG for our file format. Why? Well, it has to do with the difference between a vector image and a raster image. A vector image has several advantages including smaller sizes and the ability to be scaled to almost any screen or printing size. This is ideal for printing and even screen representation as it doesn't matter on the size it ends up on, it looks good. The drawback though is that more processing power is needed to reconstruct the image. Also, LaTeXhas no problem handling a vector image so we will take advantage of that as well.

All graphs should have a title and this one is no different. I prefer to keep titles short and to the point. So, since this graph is our Pitch, Roll and Yaw plot, it has been names as such. Our axes should also be labeled as well. Here we label our message index which is our x-axis and the values in degrees per second.

Finally we will plot our data. I have picked a blue line for our pitch, red for our roll and green for yaw data. Note the 1:2, this tells GNUPlot which column to use from the data file. Assuming you used the same order of columns I told you to do ealier, then should work just fine.

Now you try. Since I have basically provided you with the script needed you can use what is above. Your graph should look identical to the one shown in Figure 1.5. How can you view this? If you remote desktop into the linux machine, you can open the file from the file browser. It will not look very good due to the transparency. If you want a better view, the linux servers also have InkScape installed that will also view SVG files. If you are porting the files back to your Windows machine, then you can also download and install InkScape or any browser can also view these files (Edge, Chrome, Firefox).

Ok, so this was pretty straight forward. But what if you want to graph two plots of data but with different ranges of values. Let's say we want to plot your altitude vs the ground speed which we can get from our GPS data log. Our altitude varies from 989 feet to around 5,000 feet. But our ground speed only varies from 0 to around 120 knots. If plot them both, it would be tough to view changes on the graphs. Instead, we want to be able to graph them and have one Y axis from one range and the other Y axis with another range. How? GNUPlot (like any good plotting program) supports having two y-axis and each having their own range (or you can still autorange it). To do this, we use can do something like shown below.

```
    set ytics 1000 nomirror tc lt 1
```

This will setup our first y-axis (which shows up on the left hand side) with tics at 1000 intervals. We then setup our second y-axis but this time it is called as `y2tics` and we can use a similar setup as the first. However, we probably do not want intervals at 1000 for our ground speed data. Instead, 5 is probably a more reasonable number. Now, all we do plot our data. Pick blue for our altitude and green for our ground speed. You should get a graph that looks like the one in Figure 1.6. Don't forget to put a title and label your axes. To label the second y-axis, you use `y2label`.

We have data that includes coordinates in a 3D space. We have a longitude, latitude and altitude information. Can we just plot these in a 3D graph? Yes, yes we can. Figure 1.7 is such a graph that shows this data in 3D space. For now we are just saving these as static images, but if we used GNUPlot itself with the GUI we could also rotate and move around in this 3D space. So how do we do this? It is actually very simple, just plot all three data points using `splot` instead of `plot`. GNUPlot will understand that you want a 3D plot and plot them in the x, y and z-axis. Of course you also need to label your graph and each axis. For that we use `zlabel` to label our z-axis. It looks like your instructor may have written something, hmmmmm.

Latitude and Longitude is a coordinate system, so we can treat it like a coordinate system and just simply graph the data. Of course we are working with data that is geographical, so it would be nice to see how this

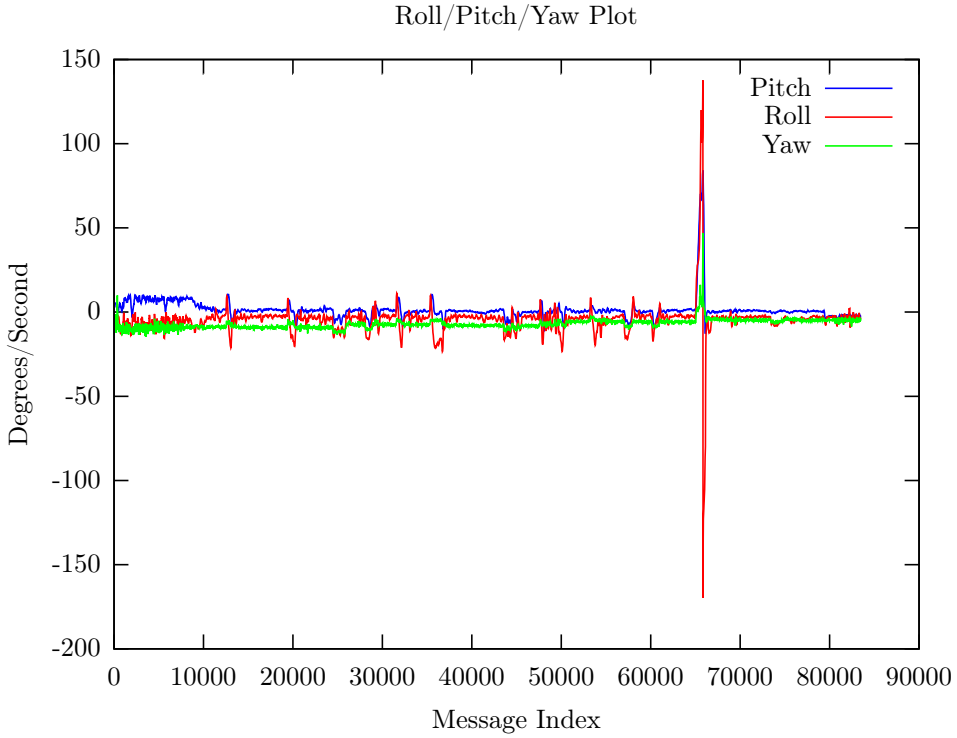

Figure 1.5: Attitude plot for flight path data from the FIT file.

compares geographically. We have included a file called `usa.txt`. This file contains all of the state border information for the United States. We can use this file to plot the various states and our data at the same time like what is shown in Figure 1.8

To plot this, you first plot the `usa.txt` file with lines. You will want to plot column two first, then column one. For this, draw the state borders as blue. Then plot your latitude and longitude data and use the same plot. You can use the "," to start a new plot. Do this as a red line. The tricky part with this graph however is defining the range. By default, the entire US will be plotted out with our data being the small red blotch in the middle of Iowa. Go ahead and try it if you want. What you need to do is specify a x-range and y-range with `xrange` and `yrange` respectively. Set the values so that you can see the state of Iowa. Remember, these will be in latitude and longitude coordinates with your x-axis as your longitude data and the y-axis as latitude. I would suggest that you set your longitude to between -97 and -90 decimal degrees and your latitude to between 40.5 and 43.6 decimal degrees.

Something is clearly there, but it is still a little hard to read. Again, we can use `xrange` and `yrange` to zoom in further. I would zoom in to around -94 to -93 decimal degrees on the longitude and 41.9 to 42.3 decimal degrees on the latitude. You should get what is shown in Figure 1.9.

Ok, the last thing we are going to do is combining some data again. This time we will look at our flight path but also plot our heading information. How? With Arrows! GNUPlot can draw arrows to represent data that is pointing or is useful for vector information. To do this though we need to do some calculations to get our heading information to a format that GNUPlot can work with. We can do this with Equations 1.7 and 1.8.

$$x_f(phi) = h * cos\left(\frac{phi}{180} * \pi\right) \tag{1.7}$$

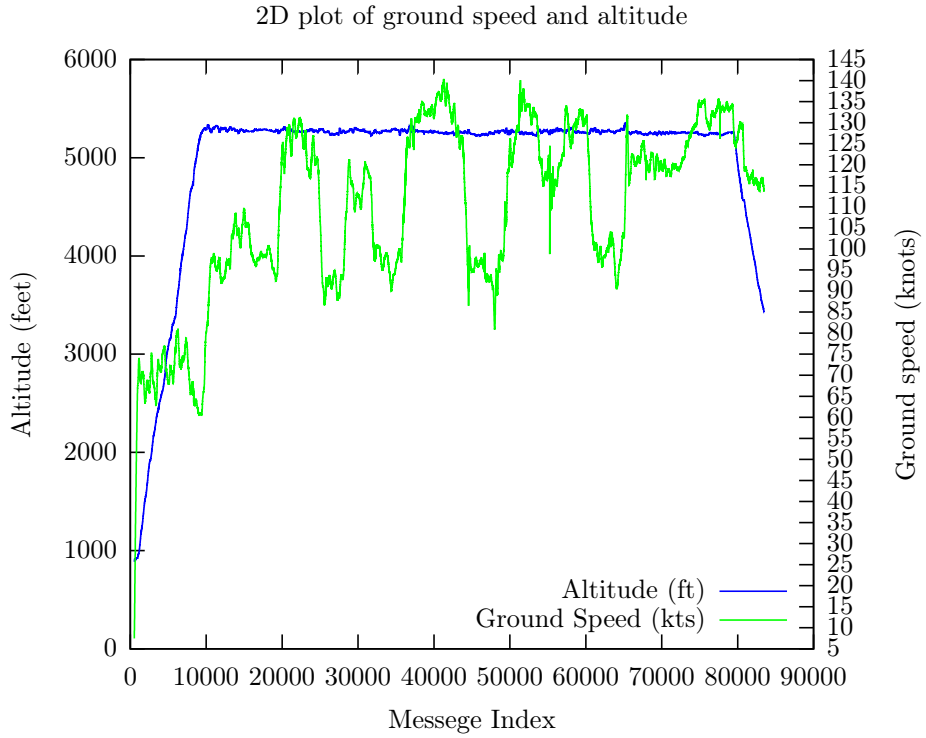

Figure 1.6: Groundspeed and altitude plot for flight path data from the FIT file.

$$y_f(phi) = h * sin\left(\frac{phi}{180} * \pi\right) \tag{1.8}$$

To start with, `h` is simply the size or length of our vector arrow. I would set this to 0.001 which works well. Then, you will need to write each equation as given in Equation 1.7 and 1.8 in GNUPlot. Since h will be defined as a constant, GNUPlot will take each data point and plug it in to phi. Below is an example of this code.

```
xf(phi) = h*cos(phi/180.0*pi)
```

To use this in your plot, first plot your flight path as we have done before, then add a second plot but now we will use the `with vectors` instead of the `with lines`. This needs four points of data to work. The first two is the same longitude and latitude points. This is where the arrow begins. Next, we will draw the end of the arrow which is calculated from `xf` and `yf`. You use those like the code below.

```
(yf($6)):(xf($6))
```

The `$6` tells GNUPlot to pull data from column six in your data which should be your heading information. GNUPlot will then do the calculations and plot both the line and arrows. You should have a graph just like Figure 1.10.

1.6.5 A note on LaTeX

You have included other image files in your reports before, but SVG files get handled a little differently. We actually use a different command for the SVG image format. In order to do this you will use the command `includesvg` instead of `includgraphics`. Below is an example of how we included these images in this writeup.

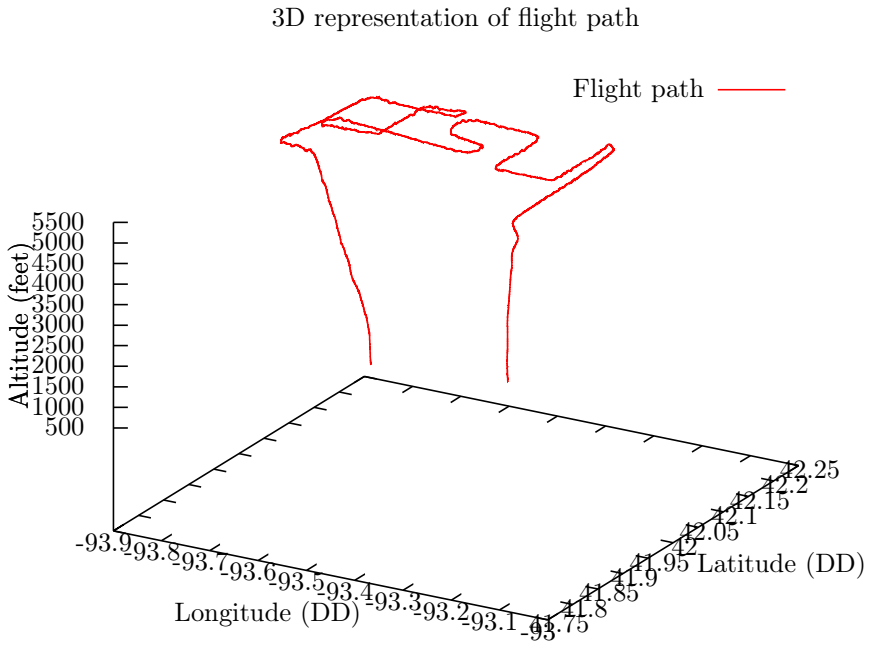

Figure 1.7: 3D plot for flight path data from the FIT file.

```
\begin{figure}[ht]
    \centering
    \includesvg[scale=0.8]{figs/chap12b/attitude.svg}
    \caption{Attitude plot for flight path data from the FIT file.}
    \label{fig:attitude_plot_FIT}
\end{figure}
```

1.7 Exercises

For this lab you will create a program called `convert_fit.out` that will read in the FIT data file we provided you. This program will generate two data files called `gps_log.txt` and `attitude_log.txt`. To compile your program, you will create a Makefile that will be able to generate your executable, run the program with the provided data, and generate the graphs with GNUPlot. Next, you will use GNUPlot to generate six graphs as outlined in this lab manual. Store each graph as a SVG file. Finally, you will write a report to answer some questions and include all of the graphs you generated. You do this in LaTeX like all other reports. Below is a breakdown for each part of the exercises.

Exercise 1. Parsing Data — (25 points): Create a program that will parse the data provided and store the data into two data files called `gps_log.txt` and `attitude_log.txt`. Your program must do the following:

- Decode Pitch, Roll and Turn Rate (Yaw) information,
- Decode Latitude, Longitude, ground speed, altitude and heading,
- Toss out any bad data,
- Store the data in the appropriate file,

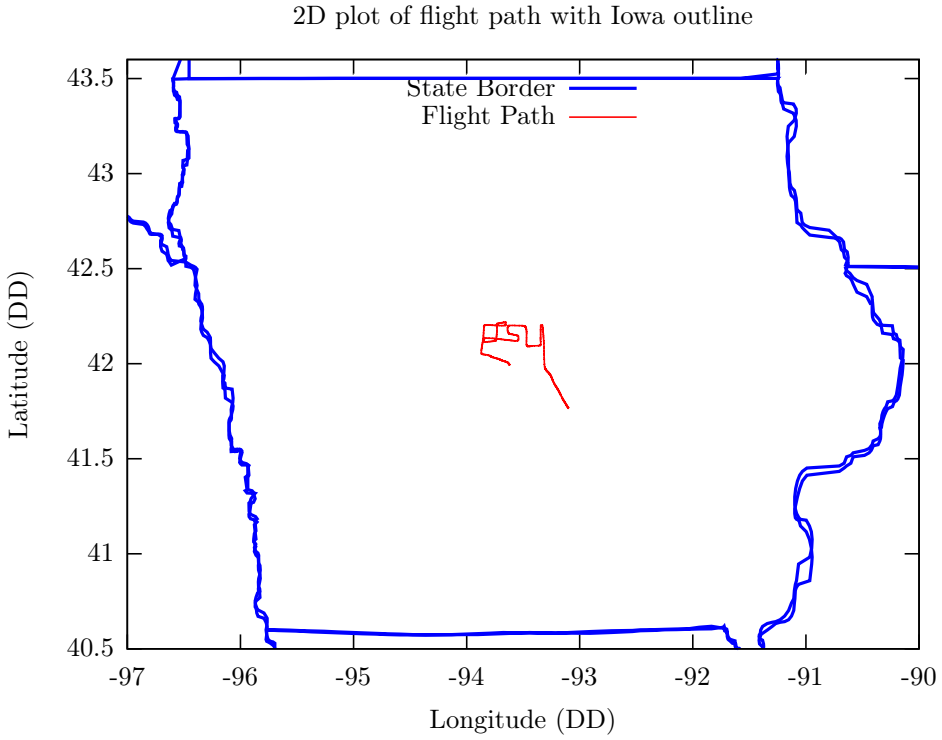

Figure 1.8: Mapped flight path data with state borders from the FIT file.

- Use a tab delimiter to store your data in your text files.

The point breakdown will be as follows:

10 point Program compiles, no warnings or errors

5 points Program runs with no errors from the API

10 points Program generates the two data files outlined above (5 points each)

Exercise 2. Makefile — (5 points): Write a Makefile for your program and graphing. It must be named `Makefile` and should:

2 points Compile with all of the SDK Libraries

2 point Run the executable `convert_fit.out` and processes the data provided

1 point Has a graphing option that you access by running `make graph` that produces your graphs.

Your Makefile must have at least three targets. The three targets are `make all`, `make run`, `make graph`. The first one will compile, the second one will run the program, and the third will use GNUPlot to plot the data.

To compile your program, you will need to make sure you compile the `fit.c`, `fit_convert.c`, `fit_crc.c`, `fit_product.c`, and `fit_ram.c` and then include each of those shared libraries as you compile your `decode.c` and again, name your final program as `convert_fit`. Your clean option should not only clean your executable and `*.o` files, but should also clean up your generated data files and graphs as well. Finally, add a `graph:` option that calls gnuplot and the `graph_data.plt` you will create in the next exercise.

Exercise 3.

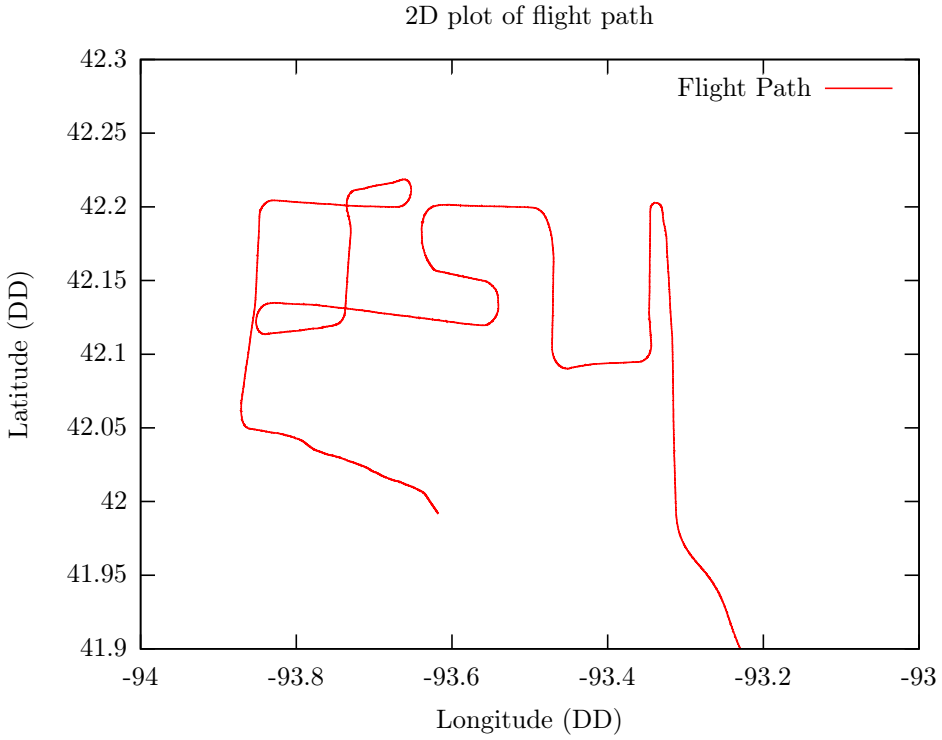

Figure 1.9: Zoomed in mapped flight path data from the FIT file.

Graph the data — (20 points):
Create a GNUPlot script file called `graph_data.plt` that will generate the graphs discussed in the lab write-up. Your script file should produce six SVG images of each graph

3 point Graph Pitch, Roll, and Yaw with correct axis and labels

4 points Graph Altitude and Ground Speed on the same graph with two distinct y-axes

3 point Do a 3D plot of the Longitude, Latitude and Altitude

3 point Do a 2D plot of the Longitude, Latitude and the state of Iowa

3 point Do a 2D plot of the Longitude, Latitude zoomed in to see what was written

4 point Do a 2D plot of the Longitude, Latitude with heading arrows

These should all be in one single script. You can use the `reset` command to reset the plot area as you go from graph to graph. Store each graph as a SVG image file.

1.8 Report

Your report should have your code outputs from the programs you ran above. It should also include copies of the code you wrote. Finally, make sure you explained how you solved the problems for each Exercise in the **Analysis** section. Remember that your report must be in a file called `main.tex` in a directory called `report` and must build without errors.

In your report include each graph that was generated and add one paragraph that describes what that graph is showing. Then, in a new section, answer the following questions:

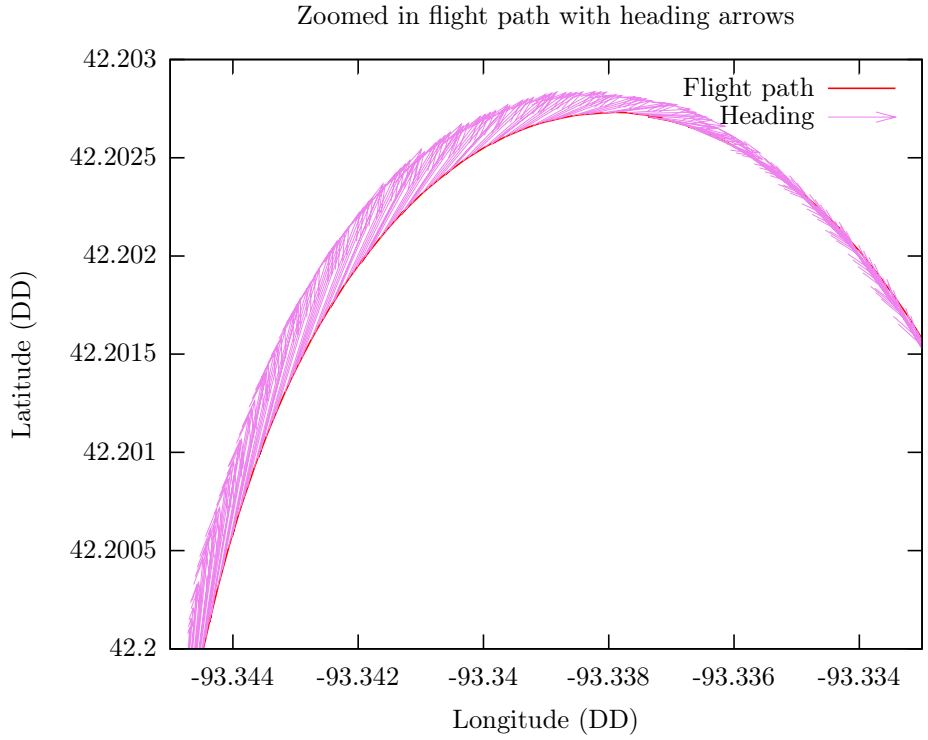

Figure 1.10: Flight path with heading arrows for flight path data from the FIT file.

- At one point in creating this data, the instructor performed what is called a "steep turn". Where did this occur? What other data collaborates that this maneuver was done. *Hint: Steep turns need the aircraft to ??? sharply.*
- In the Altitude vs ground speed graph, the ground speed varies a lot. Note that ground speed is NOT the same as airspeed as it is only from the GPS. However, why would the ground speed vary. *Hint: there are two reasons*
- In the heading with arrows graph, the heading seems to lag from the flight path. Why?

Warning: As stated in the syllabus, programs and reports that do not compile will be awarded zero points! If you have issues with your lab work, utilize your resources (lecture notes, textbooks, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

1.9 Wrapping Up

Congratulations! You have finished lab 9. When you are finished with your lab, check that you have everything done below, and enjoy the rest of your day.

☐ **Read the lab write-up**

☐ **Clone this repo into your Repl.IT, create your develop branch**

☐ **Ensure all 3 exercises are done**

☐ **Ensure the source file compiles with the sample main function without errors**

- ☐ **Push your changes to the develop branch**
- ☐ **Complete your lab writeup**
- ☐ **Open a new Pull Request (and leave it open for the grader to find)**