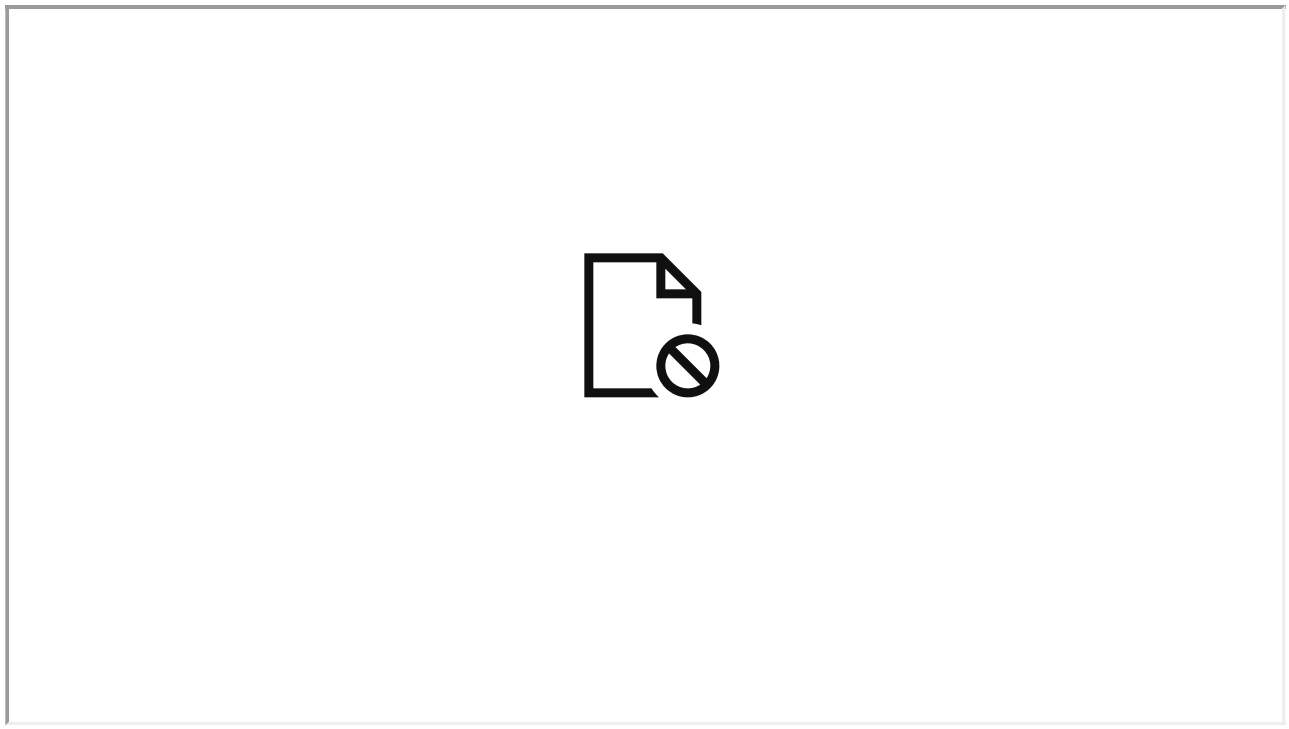
# Raspberry Pi Fitness Tracker

May 18, 2022

By Morgan Cupp (mmc274) and Maanav Shah (mcs356)



*Demonstration Video*

# Objective

Fitness trackers are an invaluable tool for people of all ages and lifestyles. These devices make tracking exercise not only easier, but also more enjoyable. This encourages and empowers people to exercise more, increasing their well-being. Unfortunately, most reliable fitness trackers cost hundreds of dollars. With this project, we set out to prove this does not have to be the case.

# Introduction

In this project, we created a relatively cheap fitness tracker using a Raspberry Pi 4 and GPS module. By connecting to GPS, the device determines and displays the current date and time. The device allows users to log data on activities they do. Specifically, the device tracks distance traveled, elapsed time, current speed, current elevation, and total elevation gain. When finished with an activity, users can save it to the device's memory and view its statistics later.

Furthermore, the device summarizes weekly data across all stored activities and presents the summarized data in convenient ways. This includes listing various totals for each week, along with weekly graphs displaying data for the various activities recorded during that week.


*Figure 1: Photograph of our fitness tracker project.*

# Design and Testing

We began this project by setting up the hardware. This part was rather simple. The hardware components consisted of a Raspberry Pi 4, Adafruit Ultimate GPS Breakout module, Adafruit PiTFT display, and a 5V power bank.

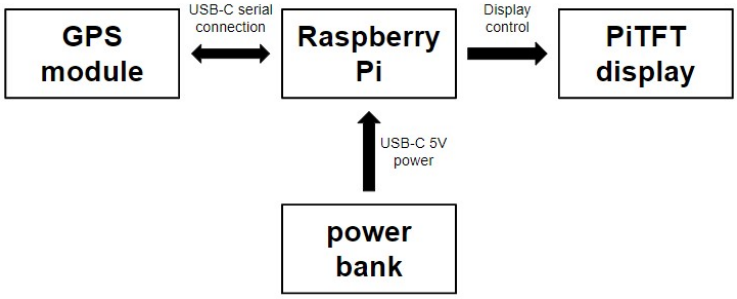



*Figure 2: Hardware schematic for the project. The GPS module communicates with the Raspberry Pi using a USB-C serial connection. The Pi communicates with the PiTFT using numerous pre-configured pins on the Pi. An external power bank powers the entire system.*


*Figure 3: Photograph of the physical system showing the hardware components. The Raspberry Pi 4 is beneath the PiTFT display which is currently showing the Linux command line. The power bank is on the right, and the GPS module is on the upper left.*

Setting up the hardware was relatively easy. In the beginning of the semester, the Raspberry Pi's SD card was configured with the latest Raspbian Linux kernel using provided instructions [1]. The PiTFT was also configured using instructions in the ECE 5725 Lab 1 handout at this time [2]. The only setup left to do for this project was to configure the GPS module. We selected this GPS module due to its position accuracy of three meters, velocity accuracy of 0.1 meters per second, and low power requirement at 25 mA listed on its datasheet [3]. Setting up the GPS module was straightforward as well. We simply had to install two Python libraries using the command line: the Adafruit_Blinka library and Adafruit_GPS library. Instructions for doing these two simple installations were linked to the datasheet webpage [4].

We considered several designs. Our initial goal was to design a wearable fitness tracker that a user could wear around his/her wrist. This would have been possible with the Pi Zero. However, given the size of the Pi Zero, using a touchscreen such as the PiTFT was not practical as it would leave little room for user interactions. We ultimately decided that we valued practical user interaction more than making it portable. Hence, we decided to stick with the Raspberry Pi 4, envisioning our system as something that might be mounted on bike handlebars.

Once these hardware decisions were made, the rest of the project consisted of software development. The high-level functionality of the software is as follows:

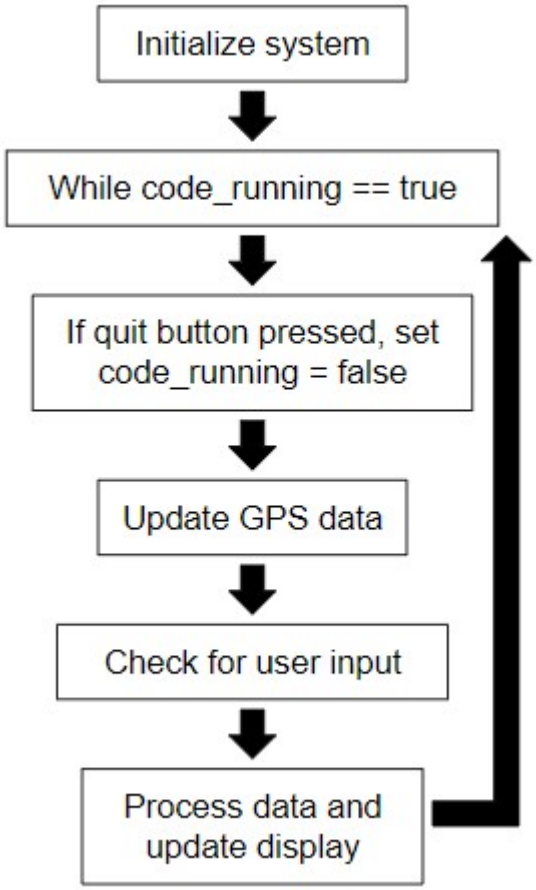



*Figure 4: Software overview of the system. After initialization, a while-loop runs repeatedly, responding to user inputs and updates to the GPS data. A physical quit button is also provided to terminate the program.*

In the first step shown in Figure 4, the system is initialized. This includes creating constants and global variables, configuring GPIO, initializing the Pygame library [5], loading images, and creating touch buttons. After this step, the program runs in an infinite while-loop. This while-loop can be terminated by pressing a physical quit button; however, this button is primarily for testing purposes. The while-loop starts by updating the data measured by the GPS module. The GPS data refreshes at a rate of one Hz, meaning a new data point is output by the GPS module every second. By refreshing at this rate, the serial connection does not get overwhelmed while still providing timely data updates. This data point includes quantities such as the current latitude, longitude, speed, altitude, and more. Furthermore, it contains the current time and date. Next, the system checks for user inputs and responds to them accordingly. Dep nding on the scenario, the system also will process data and update what is displayed on the screen if necessary. The while-loop then starts again.

We will now discuss the components of the system in more detail. When the system starts up, users are on the home screen as shown in Figure 5.

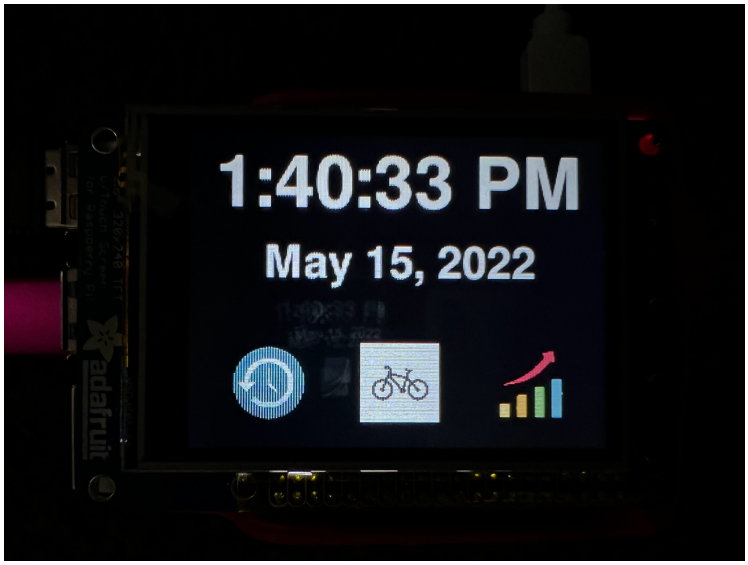



*Figure 5: Photograph of the home screen. It displays the current time, current date, and the three menu buttons to access different features of the system.*

This screen displays the current time, current date, and three buttons which can be used to access the system's various functions. The date and time are obtained from the GPS module. Since the GPS module updates every second, the current time data can be displayed to the nearest second just like most digital watches. Since a Raspberry Pi needs an internet connection to accurately determine the date and time on its own, it was very helpful that the GPS module could provide this information. While all outdoor areas have a GPS connection, most outdoor areas do not have internet. The GPS time data had to be processed in two key ways before it could be used. All GPS modules report time in UTC (Coordinated Universal Time), which is five hours ahead of EST. We first had to adjust the hours to match our time zone, EST. Since the GPS measured time in 24-hour format, we could adjust the hours using a simple subtraction. Next, the time had to be processed to match the 12-hour format used in the United States. The 12-hour format is more human readable, and thus a better choice for the home screen. The 24-hour to 12-hour conversion was achieved by adding a few if conditions to determine if the 'hours' had to be decremented by 12 and decide whether to display 'AM' or 'PM'. The date, on the other hand, was directly taken from the GPS without any pre-processing.

Finally, the three buttons take the user to three different screens. These buttons, like all buttons in this project, are touch buttons on the PiTFT implemented using Pygame. The first button (from the left) takes the user to the 'history' screen, which displays all the activities saved by the user. The second button takes the user to the 'bike' screen, which is where the user can save workout sessions and record common statistics, such as total distance traveled, total time elapsed, total elevation gain, and so on. The third button takes the user to the 'summary' screen. On this screen, the user can see an accumulation of all stats over a week. Additionally, the user can view graphs that show activity trends on a day-to-day basis for a particular week.

It is important to note that accurate timekeeping as well as datekeeping require an active GPS connection. Without a GPS connection, the code would error out at first. To prevent that from happening, the program displays 'No GPS' in place of the time whenever the module does not have a clear and strong GPS signal.

The primary function of our device is to record physical activities. To do so, the 'bike' icon is pressed on the home screen, taking the user to the screen below. Again, this is because we designed this project with biking in mind.


*Figure 6: Screenshot of the bike screen before an activity has been started. The "back" button returns to the home screen, and the "start" button starts recording an activity.*

The code determines which screen it is on using flags. For example, the "on_bike_screen" flag is set to true when the "bike" button is pressed, and the flags for the other screens are set to false. Figure 6 shows the fields displayed to the user while they complete activities: distance, time, speed, elevation gain, and altitude. Two of the values, speed and altitude, are only based on a current value, whereas distance, time, and elevation gain are cumulative. Therefore, prior to pressing "start", speed and altitude are displayed immediately while the cumulative values remain at zero. All of the data fields being displayed are stored as global variables in the code. To print this data screen, a function is called which prints all of the variables and their labels in a particular location.

When on this initial bike screen, users may press "back" to return to the home screen. If users want to start recording an activity, they must press "start". Without GPS, data can not be accurately recorded. Therefore, if there is no GPS connection available, this screen will tell users there is no connection and will not allow them to start recording an activity. If GPS is connected and "start" is pressed, all of the data fields begin showing data.


*Figure 7: Live photograph taken during the recording of an activity. The "save" button stops recording data and saves the activity, while the "pause" button just stops recording data.*

Figure 7 shows what the display looks like while recording an activity. As mentioned, the speed and altitude are live values read from the GPS every second. The altitude is output from the GPS module in meters, our desired unit, so this measured value could be used directly. Speed on the other hand, was provided by the GPS module in units of knots. Speed was thus converted from knots to kilometers per hour by multiplying the knots value by 1.852.

One small issue with the speed and altitude data was that it came in delayed by approximately ten seconds. This was likely due to the low cost of our GPS module, but it would not pose much of an issue with mainstream use cases (such as going for a bike ride, for example).

Computing the other data fields was more complex. We will first describe the elapsed time calculation. When the “start” button is initially pressed, a “reference time” is stored. This is simply the time at which the button was pressed. Every loop iteration, the “current time” is measured, and the elapsed time is calculated as “current time” - “reference time”. We initially believed this was all we had to do, but this logic failed once the pause button was implemented since the time continued incrementing while paused. To solve this problem, we first stopped incrementing the time when the “pause” button was pressed. Then, when the “start” button was pressed to resume, we stored the current value of the elapsed time into a “previous time” variable, and updated “reference time” to be the current time once again. As a result, the formula for elapsed time becomes “current time” - “reference time” + “previous time”, and this worked as desired. One interesting note about this computation is that the measurements were all done in seconds, but the values were displayed in hours, minutes, and seconds. This required various conversions back and forth throughout the code.

Perhaps the most unique computation done was for calculating distance. This was done with the help of the distance function from the GeoPy library which, according to the documentation, computes the “shortest distance on the surface of an ellipsoidal model of the earth” [6]. This is referred to as geodesic distance. The function takes two pairs of latitude and longitude coordinates as an input, and outputs the distance between these pairs of coordinates on the earth’s surface. Our first approach to calculating distance was very simple. Every iteration of the loop, the current values of latitude and longitude were measured and used as one input to the distance function. For the other input, we used the previous latitude and longitude measured to get the distance between the two. Then, this distance was added to an accumulator variable summing all of the distances measured.

This idea was conceptually sound, but it did not work in practice due to the limitations of our GPS module. Because the GPS module is not perfectly accurate, the coordinates measured would slightly change with each measurement. As a result, we were measuring changes in location and ultimately accumulating distance even while standing still. To resolve this issue, we modified our algorithm by adding a threshold. Specifically, we still retrieve the current coordinates with every loop iteration and compute the distance between these coordinates and the previously stored coordinates. However, if the distance is not greater than 50 meters, we simply discard these current coordinates. If the distance is greater than 50 meters, then the accumulated distance is updated, and the current coordinates are stored as the previous coordinates. This concept is shown in Figure 8.

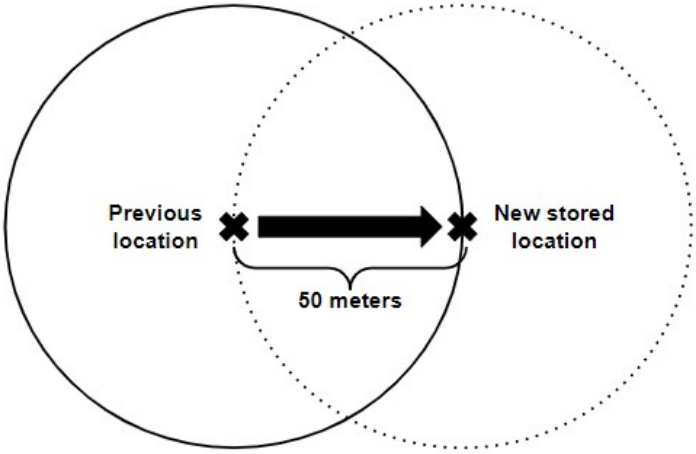



*Figure 8: Depiction of the algorithm to compute distance traveled using a threshold. The X on the left is the previous coordinate pair stored. Suppose the user travels to the position marked by the right X. This exceeds the 50 meter threshold marked by the solid circle, so their location gets updated. The new threshold boundary is marked by the dotted circle.*

This modification greatly improved our distance calculations. It completely eliminated any false distance accumulation, and led to very accurate distance measurements when traveling in paths with up to moderate turns. With that said, it is not perfect. This is because if the user were to walk around within the threshold boundary, none of this distance would be measured. To increase the accuracy using this algorithm, the threshold would need to be made as small as possible. We may have been able to shrink our threshold with additional experimentation, but a more expensive (and hence accurate) GPS module certainly would have helped as well.

Another challenge faced was factoring in elevation gain. An approximation of the problem is illustrated in Figure 9.

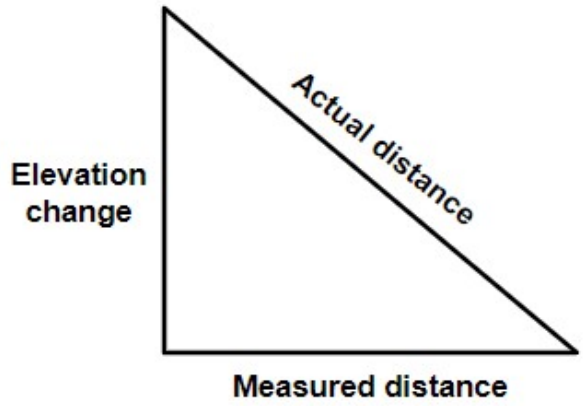



*Figure 9: Illustration of the error in distance caused by elevation changes. The distance function used does not factor in changes in elevation, meaning it ignores the vertical component of distance traveled.*

Because the Geopy distance function approximates the earth as a perfect ellipsoid, it does not capture any changes in elevation information. On a small scale, it essentially looks as if the user is traveling across perfectly flat land. This error could be corrected with the Pythagorean theorem to include the measured elevation change, however, this is complicated by the fact that earth's surface is not flat on

average. We ultimately used the assumption that the earth is a perfect ellipsoid for this project, because the measurements were still quite accurate for our purposes. With more time, we would have investigated more sophisticated ways to modify our algorithm.

The calculation for elevation gain was similar to, but simpler than, the one for distance. Each loop iteration, the current altitude was measured and compared to the previously stored altitude. A threshold was again used to prevent jitter from being interpreted as elevation changes. We found that a threshold of three meters worked well for this. Note that for elevation gain, we only accumulated positive changes in elevation gain. This is because during physical activities, users may be interested in specifically seeing how many meters uphill they have traveled. Although this elevation gain algorithm could not be improved much, we were again limited by the accuracy of the GPS module itself. We found that it definitely does detect uphills and downhills, but their magnitudes often seemed several meters off.

Similar to accumulating elapsed time, we had to account for the paused state while accumulating distance and elevation gain. To do so, we stopped updating distance and elevation while paused. Once resuming the activity, previous coordinates and altitude were set to the current coordinates and altitude, respectively. This essentially changes the user's initial point of reference to wherever they may be when they resume their activity.

We conducted numerous tests to verify the operation of our activity recording. We started by going on walks up to 600 meters long. Once we were confident in our calculations, we conducted numerous tests by driving in a car with our system. During these tests, we used a Garmin GPS running watch as a reference for our distance and elevation gain. We found that the speed measured was almost perfectly accurate, as was the distance traveled when moving in a straight line. The elevation gain was accurate enough to be useful but certainly showed some error. For example, on a hill 23 meters tall, our system measured a change of approximately 18 meters. Perhaps most importantly, our distance calculations remained fairly accurate during a drive with a fair amount of turning. We drove for around eight minutes, with two to three minutes through grid-like city streets. At the end of the test, our distance was off by around 300 meters. While not perfect, we were impressed by the accuracy of our system considering its limitations.

To finish recording an activity, the "save" button is pressed. Doing so writes the data to a file, as will be described shortly. A software diagram summarizing the operation of recording activities is in Figure 10.

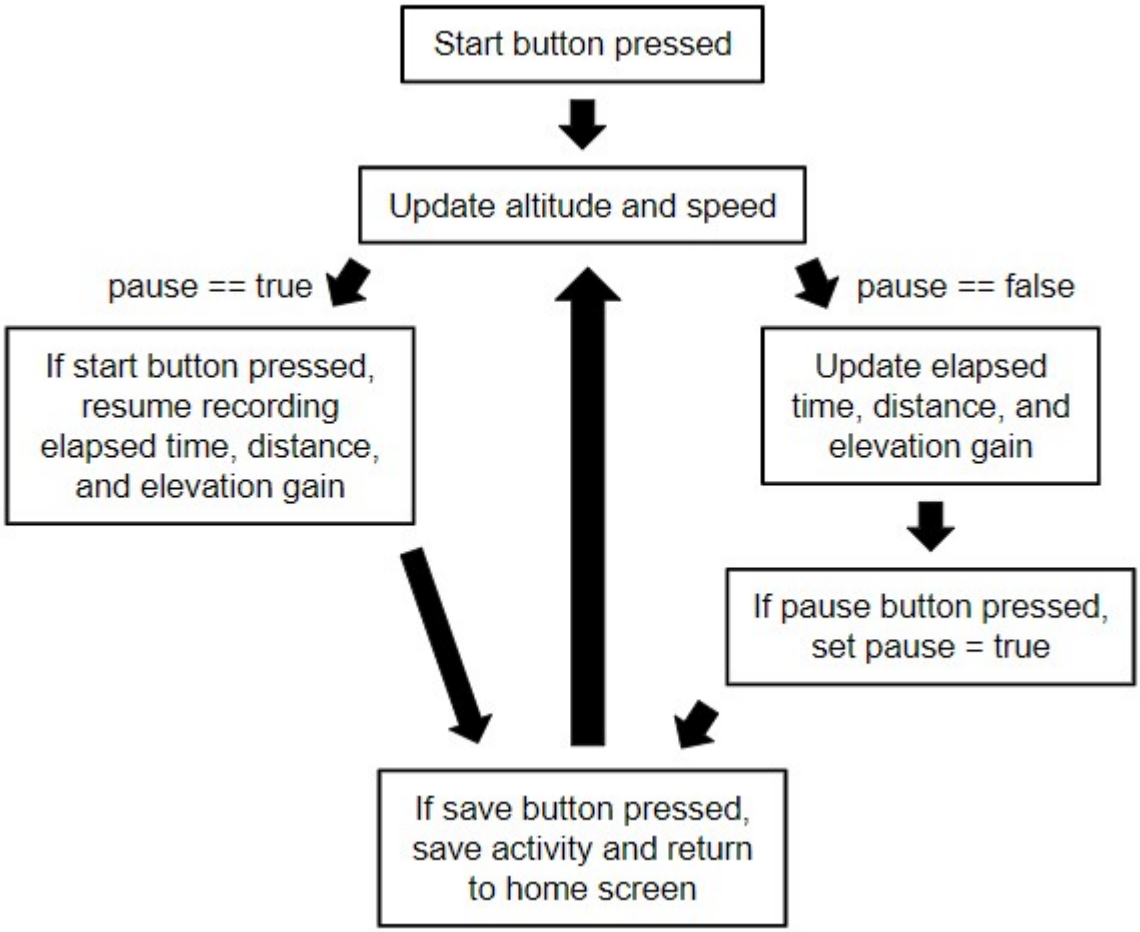



*Figure 10: Software flowchart for recording activities. Live altitude and speed are always displayed. When paused, users can either save the activity or resume recording. When not paused, distance, time, and elevation are accumulated. Users can pause or save the activity at this time.*

Our system is not just designed to record activities; it is also meant to allow users to view their past activities. This leads us to the other two buttons on the home screen. The leftmost button is the “history” button. Pressing this button allows users to view all of the past activities they have recorded. Figure 11 shows a sample activity.

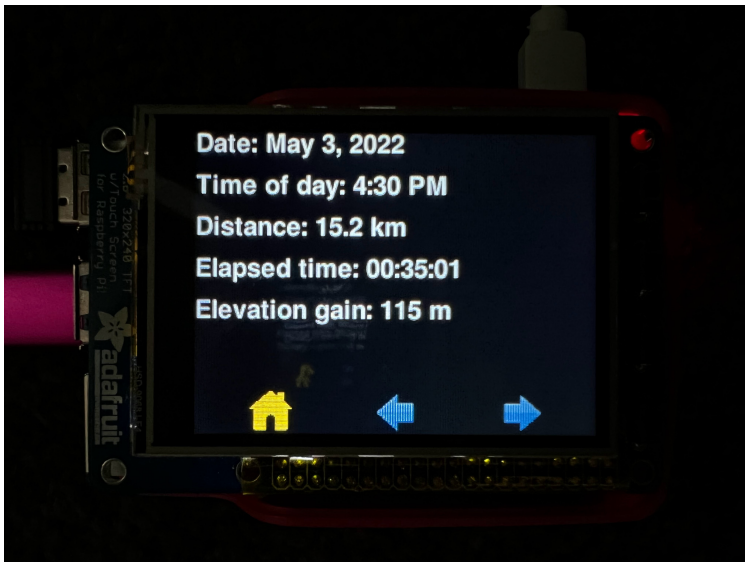


*Figure 11: Sample activity in the history screen. Pressing the home button returns to the home screen. The left and right arrows can be used to view less recent and more recent activities, respectively.*

The majority of this feature's implementation actually occurs when the "save" button is pressed. When this happens, a function is called to write the current activity being recorded to a file. A .txt file is written to a filesystem folder called "activities". The name of the file is the current date followed by the current time in 24-hour format. The current values of the global variables for the date, time of day, distance, elapsed time, and elevation gain are used when writing this file. The file looks exactly like the text in Figure 11; this made displaying the file later easy.

This file structure was crucial when implementing the history functionality. First, we read the names of the files in the "activities" directory. Due to the naming scheme for the activity files, this list of activities automatically has the activities in chronological order. The activity being displayed is selected by maintaining an index into this activities list. In each loop iteration, the index is used to retrieve one activity file name from the list. The contents of this activity file are then displayed. When an arrow button is pressed, the index is incremented or decremented accordingly to display a different activity.

We faced two bugs while implementing this feature. First, our activities were not completely chronological initially, because we were not using double-digit dates. For example, May 2 and May 10 were represented as 5/2 and 5/10 rather than 05/02 and 05/10. The first naming scheme's alphabetical ordering did not result in a chronological ordering, whereas the second naming scheme does. The 24-hour naming scheme was also required instead of a 12-hour naming scheme for similar reasons. The second bug was that the history screen would crash when no activities were stored yet. Thus, we only try to read files from the "activities" directory if there in fact exists files in the directory.

Fitness trackers, however, should not just let you save your sessions and view your past activities. They should also allow you to view and assess your progress in a meaningful way. Our system achieves this by processing the data collected in the 'activities' folder in two key ways. Firstly, the system reads all the data stored in the 'activities' files, and stores and displays weekly totals of distance, elapsed time, and elevation gain. These weekly summaries can be accessed by clicking the rightmost button on the home screen which takes the user to the 'summary' screen. The summary screen is shown in Figure 12.


*Figure 12: Sample summary screen, showing the total distance, total time, and total elevation gain of the week May 09 to May 15, 2022.*

As demonstrated in Figure 12, the ‘summary’ screen displays the starting date of the week, as well as the ending date of the week at the top of the screen. It then displays the ‘Total distance’ traveled within the week, ‘Total time’ spent exercising during the week, and the ‘Total elevation gain’. The distance is displayed in ‘km’, the total time in hours and minutes, and the total elevation gain in ‘m’. These units were chosen because it is highly unlikely that the user would travel less than a ‘km’ throughout a week, or exercise for less than a minute. Thus, it was not necessary to increase the resolution of these particular statistics.

The arrow keys can be used to browse through the weeks. Their implementation on this screen is identical to their implementation on the ‘history’ screen: left arrow for an earlier week, and the right arrow for a later week (unless the weeks need to be ‘wrapped’ around, in which case, they switch). The ‘home’ button, again, takes the user to the ‘home’ screen.

Secondly, the system plots the weekly data to produce distance, time, and elevation gain graphs. The resulting graphs are bar plots that show how the statistics vary over the course of the week, from Monday to Sunday. The x-axis of the graphs shows the day of the week, and the y-axis shows the particular statistic in question. The ‘graphs’ screen can be accessed by clicking the three buttons placed above the navigation buttons. The leftmost button can be clicked to access the distance graph corresponding to the week currently being displayed on the summary screen. The middle button can be clicked to access the time graph, and the rightmost button can be clicked to access the elevation gain graph. Moreover, the arrow keys can be used with the graphs as well. Once on the ‘graphs’ screen, the arrow keys can cycle through the graphs of different weeks. Figure 13 shows examples of all three graphs.

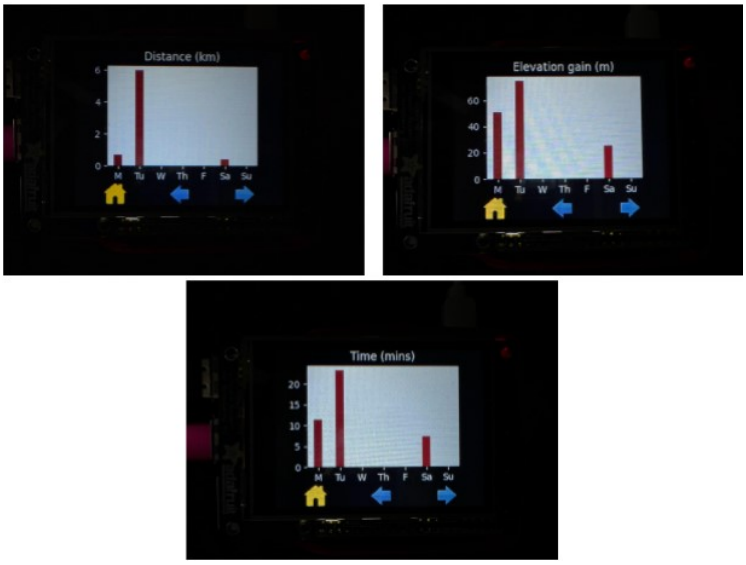



*Figure 13: Example distance, time, and elevation gain graphs corresponding to the week May 09 to May 15, 2022.*

We used helper functions to achieve the processing highlighted above. Figure 14 below shows a flowchart of the additional tasks the system performs every time a user hits the ‘save’ button.


*Figure 14: Software flow diagram depicting what occurs when the save button is pressed. First, a file is made for the current activity. Then, weekly data across all activities is recomputed. This data is saved in two forms: .txt files containing weekly totals, and .jpg files containing graph plot images.*

In addition to the tasks described in the ‘history’ screen section, hitting the ‘save’ button initiates the summary calculation and writing process. Once the activities folder has been updated, the system starts reading from the most recent activity file, and uses the date in its file name to calculate on which day of the week the activity took place. The system does this by using the DateTime module, which contains useful helper methods that enable easy and efficient date-related calculations. Then, the system calculates what date would correspond to the start of that particular week (i.e., a Monday), and what date would correspond to the end of that week (i.e., a Sunday). These dates are stored as the start_date and the end_date. For every subsequent activity file that is read, the system checks whether the activity falls in the same week as the previous activity. If it does, the distance, time, and elevation gain measurements from the current activity are accumulated into their respective accumulators. Once all the activities belonging to a week have been processed, the total distance, total elapsed time, and total elevation gain are written to a ‘summary’ file. This summary file is stored in the ‘summaries’ folder, distinct from the ‘activities’ folder. The start_date is used as the file name to keep things alphabetical and hence chronological as well.

In addition to the summary writing process, the system keeps track of daily measurements as well. While reading the activity files, it adds up all the session statistics corresponding to a particular day. Hence, there are two accumulations occurring simultaneously: one for the weekly totals, and one for each day during that week. The daily accumulations are used to create the graphs. We use Matplotlib to generate all our graphs. Instead of keeping the plots open at all times, we decided to maintain three folders for our plots, one for each measurement. Once the graphs have been generated, we first save the plots in their respective folders, and then immediately close the plots. This was done to ensure that the system never gets overloaded. Matplotlib plots consume a significant amount of RAM when open. During our initial runs, we ran into issues wherein Matplotlib complained about shortage of RAM. Moreover, the system had gotten noticeably slower. Saving the plots as images, and then simply loading the image when the

user requested a particular plot, saved a lot of memory and was not very challenging to implement, especially since we had all the file management code ready. We simply re-used the activity loading and saving code to load and save plots.

Accumulating and plotting the data went smoothly overall. A bulk of the issues we faced with the plots stemmed from formatting. The plots were simply too big for the small PiTFT screen. Resizing the plots did not do much, since the axes labels became unreadable. In addition, the white space around the plot did not blend well with the black background. The plots looked awkward and unnatural. To solve these issues, we had to search for the specific method calls and attribute settings that could allow us to alter these parameters. We changed the whitespace around the plot area to 'black'-space, which allowed the plot to blend in with the background. We also had to change the font color from black to white. Additionally, we found a parameter that allowed us to change the axes labels font size. We could then make the axes labels bigger without making the plots too big for the screen. These changes, along with a few more minor changes, made the plots look better on the small PiTFT screen.

# Results

Overall, everything in our project performed as planned and we met the goals outlined in our project proposal. Our ultimate goal was to create an affordable fitness tracker that could reasonably calculate key metrics provided by more expensive, mainstream fitness trackers. We also wanted to implement a system that could effectively store and summarize user data. We certainly accomplished this goal. All of the metrics mentioned can be calculated by our system with a reasonable amount of accuracy. The data for these activities can be viewed in an intuitive and useful way, and the weekly summaries and graphs provide a meaningful analysis of this data. Furthermore, our system operates smoothly and without any bugs.

Perhaps most importantly, our system would be practical to use in serious applications. The user interface is easy to use and "just works". With more compact packaging, we feel that this device could be put to real use.

# Conclusions

This project is ultimately a proof of concept that a useful fitness tracker does not need to cost hundreds of dollars. It also is a testament to how capable a system as simple as the Raspberry Pi can be. With additional time and refinement, it seems plausible that this system could serve as a genuine daily fitness tracker. Perhaps the code could be made open-source, allowing anyone who purchases the parts to create this device for themselves.

We did not necessarily discover anything that definitely did not work while doing this project. However, we did discover some design considerations that must be made while designing a system like this. First, the screen must be large enough. We considered using a smaller screen and immediately could see how this would make designing a usable interface more challenging. Second, the system must remain portable. Our system was fine for our purposes, but in real use we would certainly need to get a smaller battery pack, fasten the GPS more securely, and enclose everything. Striking a balance between this usability and portability is key, and veering too far in either direction will certainly result in a system that is not practical.

# Future Work

With additional time, we primarily would have worked on improving our algorithm for calculating distance. Perhaps we could devise a way to factor elevation changes into the calculation. We also could have tried shrinking the distance threshold further by doing more experiments, allowing sharper turns to be measured more accurately. One possible stretch goal would be to allow users to upload their activity's route to a computer as is done with some popular fitness trackers. By doing so, map data could be used to accurately factor elevation changes into the distance calculation.

We also could spend time making the system more compact, as described above. We could have also continued implementing more features such as customizable data fields and layouts, more summary statistics, a compass, and more. Another interesting possibility if this project were to be open source is that anyone wanting a new feature on this device could add it if they knew how to do so.

---

# Budget

- Raspberry Pi 4 $35 (given to us in lab)
- Adafruit Ultimate GPS module $30
- PiTFT [8] $20 (given to us in lab)

## Total: $85

---

# References


[1] J. Skovira. "Setting up an SD card on a Mac." ECE 5725 Canvas Website. (accessed Mar. 9, 2022)

[2] J. Skovira. "ECE 5725: Lab 1." ECE 5725 Canvas Website. (accessed Mar. 9, 2022)

[3] "Adafruit Ultimate GPS Breakout - 66 channel w/ 10 Hz updates - PA1616S." Adafruit. https://www.adafruit.com/product/746#technical-details (accessed Apr. 15, 2022)

[4] "CircuitPython & Python Setup." Adafruit. https://learn.adafruit.com/adafruit-ultimate-gps/circuitpython-parsing (accessed Apr. 15, 2022)

[5] "Pygame Documentation." Pygame. https://www.pygame.org/docs/ (accessed May 3, 2022)

[6] "Welcome to GeoPy's documentation!" GeoPy. https://geopy.readthedocs.io/en/stable/#module-geopy.distance (accessed May 3, 2022)

[7] "Raspberry Pi 4 Model B." Raspberry Pi. https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf (accessed May 3, 2022)

[8] "MI0283QT-11 Specification Datasheet by Adafruit Industries LLC." Digi-Key Electronics. https://www.digikey.com/htmldatasheets/production/1640715/0/0/1/mi0283qt-11-specification.html (accessed May 3, 2022)


---

# Code Appendix

Link to code repository. (https://github.com/mcs356/ECE5725-Final-Project)

```
# By Morgan Cupp (mmc274) and Maanav Shah (mcs356)
# ECE 5725 Final Project: Fitness Tracker
# Completed: May 9, 2022

import time
import RPi.GPIO as GPIO
import pygame
from pygame.locals import *
import os
import adafruit_gps
import serial
from geopy import distance
import datetime
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

# configure GPIO
GPIO.setmode(GPIO.BCM) # set broadcom
GPIO.setup(27, GPIO.IN, pull_up_down=GPIO.PUD_UP) # quit button

# piTFT setup
os.putenv('SDL_VIDEODRIVER', 'fbcon') # display on piTFT
os.putenv('SDL_FBDEV', '/dev/fb0') # use this when HDMI is not plugged in
os.putenv('SDL_MOUSEDRV', 'TSLIB') # track mouse clicks on piTFT
os.putenv('SDL_MOUSEDEV', '/dev/input/touchscreen')

pygame.init() # initialize pygame

# set up GPS module
uart = serial.Serial('/dev/ttyUSB0', baudrate=9600, timeout=10) # set up serial to GPS mo
gps = adafruit_gps.GPS(uart, debug=False) # create GPS module instance
gps.send_command(b'PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0') # enable GGA and RMC i
gps.send_command(b'PMTK220,1000') # update GPS once per second

# pygame configuration
pygame.mouse.set_visible(False) # false when running on piTFT
screen = pygame.display.set_mode((320,240)) # display size
large_font = pygame.font.Font(None, 80) # large font size
medium_font = pygame.font.Font(None, 50) # medium font size
small_font = pygame.font.Font(None, 30) # small font size
home_button_size = 60, 60
button_size = 40,40

# RGB color representations
WHITE = 255, 255, 255
BLACK = 0,0,0
RED = 255, 0, 0
GREEN = 0, 255, 0

# global variables
last_time = time.monotonic() # last time gps was updated
labels = {'distance': (53,15), 'time': (159,15), 'speed': (265,15), # labels for data fie
          'elevation gain': (100,100), 'altitude': (221,100)}
altitude = 0 # current altitude
elevation_gain = 0 # total elevation gain during an activity
first_val_alt = True # true when no altitude values have been measured yet
prev_altitude = 0 # previous altitude measured
```

```
speed = 0 # current speed
dist = 0 # accumulator used for distance travelled
prev_latitude = 0 # previous latitude measured
prev_longitude = 0 # previous longitude measured
first_val = True # true when no distance values have been measured yet
ref_time = 0 # reference time being used to measure activity duration
elapsed_time = '00:00:00' # elapsed time of the user's current activity
prev_time = 0 # previous time measured
screen_num = 0 # index of the screen being displayed (when using arrow buttons)
activities = [] # list of names of all activity files
summaries = [] # list of names of all activity summmary files
graphs = [] # list of names of all activity summmary files
graph_path = '' # path to the name of a particular graph file

# flags
code_running = True # run code while this is true
on_home_screen = True # true when when user is on the home screen
on_bike_screen = False # true when user is on bike screen
on_history_screen = False # true when user is on history screen
on_summary_screen = False # true when user is on summary screen
on_graph_screen = False # true when user is on graph screen
start = False # true when an activity has been started
pause = False # true when an activity is paused

screen.fill(BLACK) # erase workspace

# load images
bike = pygame.image.load('/home/pi/final_project/images/bike.jpg')
bike = pygame.transform.scale(bike, home_button_size)
history = pygame.image.load('/home/pi/final_project/images/history.png')
history = pygame.transform.scale(history, home_button_size)
left_arrow = pygame.image.load('/home/pi/final_project/images/left_arrow.png')
left_arrow = pygame.transform.scale(left_arrow, button_size)
right_arrow = pygame.image.load('/home/pi/final_project/images/right_arrow.png')
right_arrow = pygame.transform.scale(right_arrow, button_size)
home = pygame.image.load('/home/pi/final_project/images/home.png')
home = pygame.transform.scale(home, button_size)
summary = pygame.image.load('/home/pi/final_project/images/summary.png')
summary = pygame.transform.scale(summary, home_button_size)
dist_graph = pygame.image.load('/home/pi/final_project/images/distance.png')
dist_graph = pygame.transform.scale(dist_graph, button_size)
time_graph = pygame.image.load('/home/pi/final_project/images/time.png')
time_graph = pygame.transform.scale(time_graph, button_size)
elev_graph = pygame.image.load('/home/pi/final_project/images/elevation.png')
elev_graph = pygame.transform.scale(elev_graph, button_size)

# create home screen buttons
home_screen_buttons = {history: (60,190), bike: (160,190), summary: (260,190)}
home_screen_button_rects = {}

for image, position in home_screen_buttons.items():
    rect = image.get_rect(center=position)
    home_screen_button_rects[image] = rect

# create bike screen buttons
bike_screen_buttons = {'save': (107,200), 'start': (214,200), 'pause': (214,200), 'back':
bike_screen_button_rects = {}

for text, position in bike_screen_buttons.items():
    text_surface = small_font.render(text, True, WHITE)
```

```
119     rect = text_surface.get_rect(center=position)
120     bike_screen_button_rects[text] = rect
121 
122 # create arrow buttons
123 arrow_buttons = {home: (60,220), left_arrow: (160,220), right_arrow: (260,220)}
124 arrow_button_rects = {}
125 
126 for image, position in arrow_buttons.items():
127     rect = image.get_rect(center=position)
128     arrow_button_rects[image] = rect
129 
130 # create graph selection buttons
131 graph_selection_buttons = {dist_graph: (60,170), time_graph: (160,170), elev_graph: (260,
132 graph_selection_button_rects = {}
133 
134 for image, position in graph_selection_buttons.items():
135     rect = image.get_rect(center=position)
136     graph_selection_button_rects[image] = rect
137 
138 # display the time of day on the home screen
139 def display_time_of_day():
140     global time_of_day
141 
142     if not gps.has_fix: # if GPS not connected, can't tell the time
143         time_of_day = 'NO GPS'
144     else:
145         time_of_day = '{:02}:{:02}:{:02}'.format( # get UTC time from gps
146                 gps.timestamp_utc.tm_hour,
147                 gps.timestamp_utc.tm_min,
148                 gps.timestamp_utc.tm_sec,
149             )
150         hour = (int(time_of_day[:2])-4)%24 # convert UTC time to EST
151         if (hour < 12): # convert 24-hour EST time to AM/PM
152             am_pm = 'AM'
153         elif (hour == 12):
154             am_pm = 'PM'
155         else:
156             am_pm = 'PM'
157             hour = hour-12
158         time_of_day = str(hour) + time_of_day[2:] + ' ' + am_pm
159     text_surface = large_font.render(time_of_day, True, WHITE)
160     rect = text_surface.get_rect(center=(160,40))
161     screen.blit(text_surface, rect)
162 
163 # display the date on the home screen
164 def display_date():
165     global date
166 
167     if not gps.has_fix: # if GPS not connected, can't tell the date either
168         date = ''
169     else:
170         date = '{:02}/{:02}/{}'.format(         # get date from gps
171                 gps.timestamp_utc.tm_mon,
172                 gps.timestamp_utc.tm_mday,
173                 gps.timestamp_utc.tm_year)
174 
175     text_surface = medium_font.render(get_verbose_date(date), True, WHITE)
176     rect = text_surface.get_rect(center=(160,100))
177     screen.blit(text_surface, rect)
178 
```

```
# convert date from form '04/30/2022' to form 'April 30, 2022'
def get_verbose_date(numeric_date):
    if numeric_date == '': # corner case when there is no GPS signal
        return ''
    parts = numeric_date.split('/')
    index = int(parts[0])-1
    months = ['Jan.', 'Feb.', 'Mar.', 'Apr.', 'May', 'Jun.', 'Jul.', 'Aug.', 'Sep.', 'Oct
    month = months[index]
    day = parts[1]
    year = parts[2]
    return month + ' ' + day + ', ' + year

# display data labels on bike screen
def display_data_labels():
    small_font.set_underline(True) # underline the data labels
    for text, pos in labels.items():
        text_surface = small_font.render(text, True, WHITE)
        rect = text_surface.get_rect(center=pos)
        screen.blit(text_surface, rect)
    small_font.set_underline(False)

# display data values on bike screen
def display_data():
    # display time
    text_surface = small_font.render(elapsed_time, True, WHITE)
    x,y = labels['time']
    y = y + 30
    rect = text_surface.get_rect(center=(x,y))
    screen.blit(text_surface, rect)

    # display distance
    text_surface = small_font.render(str(round(dist,2)) + ' km', True, WHITE)
    x,y = labels['distance']
    y = y + 30
    rect = text_surface.get_rect(center=(x,y))
    screen.blit(text_surface, rect)

    # display altitude
    text_surface = small_font.render((str(int(altitude))) + ' m', True, WHITE)
    x,y = labels['altitude']
    y = y + 30
    rect = text_surface.get_rect(center=(x,y))
    screen.blit(text_surface, rect)

    # display speed
    text_surface = small_font.render(str(speed) + ' km/h', True, WHITE)
    x,y = labels['speed']
    y = y + 30
    rect = text_surface.get_rect(center=(x,y))
    screen.blit(text_surface, rect)

    # display elevation gain
    text_surface = small_font.render(str(int(elevation_gain)) + ' m', True, WHITE)
    x,y = labels['elevation gain']
    y = y + 30
    rect = text_surface.get_rect(center=(x,y))
    screen.blit(text_surface, rect)

# tell user to move to better location when the GPS is not connecting
def tell_user_to_move():
```

```
    lines = {'GPS not connected.': (160,80), 'Move to location with clear sky.': (160,100
    for text, pos in lines.items():
        text_surface = small_font.render(text, True, WHITE)
        rect = text_surface.get_rect(center=pos)
        screen.blit(text_surface, rect)

# update altitude displayed to the user
def update_altitude():
    global altitude
    if gps.altitude_m is not None:
        altitude = gps.altitude_m

# update the elevation gain displayed to the user
def update_elevation_gain():
    global elevation_gain, first_val_alt, prev_altitude

    current_altitude = gps.altitude_m
    if first_val_alt: # if very first measurement, set current = previous
        prev_altitude = current_altitude
        first_val_alt = False
    altitude_diff = current_altitude - prev_altitude
    print("Altitude difference: " + str(altitude_diff))
    if abs(altitude_diff) > 3: # only record altitude changes of greater than 3 m to redu
        prev_altitude = current_altitude
        if altitude_diff > 0: # only positive elevation changes contribute to gain
            elevation_gain += altitude_diff
            print("Total elevation gain: " + str(elevation_gain))

# update speed displayed to the user
def update_speed():
    global speed

    if gps.speed_knots is not None:
        if gps.speed_knots > 1:
            speed = round(gps.speed_knots * 1.150779,1) # convert knots to km/h
        else:
            speed = 0

# update distance displayed to the user
def update_distance():
    global dist, first_val, prev_latitude, prev_longitude

    lat = gps.latitude
    lon = gps.longitude
    coord1 = (lat,lon)
    if first_val: # if very first measurement, set current = previous
        prev_latitude = lat
        prev_longitude = lon
        first_val = False
    coord2 = (prev_latitude,prev_longitude)
    del_dist = distance.distance(coord1,coord2).km
    print("Distance difference: " + str(del_dist))
    if del_dist >= 0.05: # only update when 50 m change in position is detected to reduce
        dist += del_dist
        print("Total distance: " + str(dist))
        prev_latitude = lat # current values become previous values in the next iteration
        prev_longitude = lon

# update the elapsed time displayed to the user
def update_elapsed_time():
```

```
299     global elapsed_time
300
301     time_diff = int(time.monotonic()-ref_time) + prev_time
302     hours = int(time_diff/3600)
303     minutes = int((time_diff-(hours*3600))/60)
304     seconds = time_diff%60
305     elapsed_time = '{:02}:{:02}:{:02}'.format(hours, minutes, seconds)
306
307 # saves data for the current activity in a file
308 def write_activity_to_file():
309     global date, time_of_day, dist, elapsed_time, elevation_gain
310
311     # convert time to 24-hour format to alphabetize file names
312     if 'AM' in time_of_day:
313         time_24hr = time_of_day[:-3]
314     else:
315         i = time_of_day.index(':')
316         time_24hr = str(int(time_of_day[:i])+12) + time_of_day[i:-3]
317
318     filename = date.replace('/','-') + '_' + time_24hr
319     with open('/home/pi/final_project/activities/' + filename, 'w') as file:
320         file.write('Date: ' + get_verbose_date(date) + '\n')
321         i = time_of_day.rindex(':')
322         file.write('Time of day: ' + time_of_day[:i] + time_of_day[i+3:] + '\n')
323         file.write('Distance: ' + str(round(dist,1)) + ' km \n')
324         file.write('Elapsed time: ' + elapsed_time + '\n')
325         file.write('Elevation gain: ' + str(int(elevation_gain)) + ' m \n')
326
327 # resets the user interface to its initial state
328 def reset_system():
329     global altitude, elevation_gain, prev_altitude, speed, dist, prev_latitude
330     global prev_longitude, ref_time, prev_time, first_val_alt, first_val, elapsed_time
331     global start, pause, on_home_screen, on_bike_screen, on_history_screen, on_summary_sc
332
333     altitude = elevation_gain = prev_altitude = speed = dist = 0
334     prev_latitude = prev_longitude = ref_time = prev_time = 0
335     first_val_alt = first_val = True
336     elapsed_time = '00:00:00'
337     start = pause = False
338     on_home_screen = True
339     on_bike_screen = on_history_screen = on_summary_screen = on_graph_screen = False
340
341 # plot and save bar graph
342 def plot_bars(x_axis, y_axis, title, filename):
343     matplotlib.rcParams.update({'font.size': 10, 'text.color' : 'white',
344                                 'xtick.color': 'white', 'ytick.color': 'white'})
345     fig = plt.figure(figsize = (3,2), facecolor='black')
346     plt.bar(x_axis, y_axis, color = 'maroon', width=0.4)
347     plt.title(title)
348     plt.savefig(filename)
349     plt.close()
350
351 # write weekly totals for a particular week to a summary file
352 def write_summary_to_file(dist_sum, time_sum, elev_sum, start_date, end_date):
353     hours = int((time_sum/60))
354     mins = time_sum%60
355     with open('/home/pi/final_project/summaries/' + start_date.replace('/','-'), 'w') as
356         file.write(get_verbose_date(start_date) + ' to ' + get_verbose_date(end_date) + '
357         file.write('Total distance: ' + str(dist_sum) + ' km \n')
358         file.write('Total time: ' + str(hours) + ' hours ' + str(mins) + ' mins \n')
```

```
        file.write('Total elevation gain: ' + str(elev_sum) + ' m \n')

# compute the start date of the week that an activity is in
def get_start_of_week(activity_filename):
    activity_datetime = activity_filename.split('-')
    activity_datetime = datetime.datetime(int(activity_datetime[2][:4]),int(activity_date
    day_of_week = activity_datetime.weekday()
    start_of_week = activity_datetime - datetime.timedelta(day_of_week) # start date of c
    return start_of_week

# create graphs and weekly totals for distance, time, and elevation gain data
def summarize_data():
    global activities

    x_axis = ['M', 'Tu', 'W', 'Th', 'F', 'Sa', 'Su']
    dist_vals = [0]*7 # store distance per weekday
    time_vals = [0]*7 # time per weekday
    elev_vals = [0]*7 # elevation gain per weekday

    for i in range(len(activities)):

        # get date of activity as a datetime object
        activity_datetime = activities[i].split('-')
        activity_datetime = datetime.datetime(int(activity_datetime[2][:4]),int(activity_
        day_of_week = activity_datetime.weekday()
        start_of_week = activity_datetime - datetime.timedelta(day_of_week) # start date
        end_of_week = activity_datetime + datetime.timedelta(6-day_of_week) # end date of

        with open('/home/pi/final_project/activities/' + activities[i], 'r') as file:
            lines = file.readlines()
        for line in lines:
            if 'Distance' in line:
                dist_vals[day_of_week] += float(line.split(' ')[1])
            if 'Elapsed time' in line:
                time_str = line.split(' ')[2]
                time_str = time_str.split(':')
                hours = int(time_str[0])
                mins = int(time_str[1])
                secs = int((time_str[2])[:-1]) # remove newline character from this strin
                time_in_mins = (secs + mins*60 + hours*3600)/60
                time_vals[day_of_week] += time_in_mins
            if 'Elevation gain' in line:
                elev_vals[day_of_week] += float(line.split(' ')[2])

        # if there are no more activity files or the next activity belongs to a different
        if i == (len(activities)-1) or start_of_week != get_start_of_week(activities[i+1]
            # get start and end dates of week
            start_date = start_of_week.strftime('%m/%d/%Y')
            end_date = end_of_week.strftime('%m/%d/%Y')

            # plot and store bar graphs for that week
            plot_bars(x_axis, dist_vals, 'Distance (km)', '/home/pi/final_project/graphs/
            plot_bars(x_axis, time_vals, 'Time (mins)', '/home/pi/final_project/graphs/ti
            plot_bars(x_axis, elev_vals, 'Elevation gain (m)', '/home/pi/final_project/gr

            # write a summary file for that week
            dist_sum = sum(dist_vals)
            time_sum = int(sum(time_vals))
            elev_sum = int(sum(elev_vals))
            write_summary_to_file(dist_sum, time_sum, elev_sum, start_date, end_date)
```

```
419
420             # reset arrays for the following week
421             dist_vals = [0]*7 # store distance per weekday
422             time_vals = [0]*7 # time per weekday
423             elev_vals = [0]*7 # elevation gain per weekday
424
425 while code_running:
426     time.sleep(0.2)
427
428     if (not GPIO.input(27)): # if quit button is pressed, end program
429         code_running = False
430
431     screen.fill(BLACK) # erase workspace
432
433     # get new data from GPS every second
434     current_time = time.monotonic()
435     if current_time - last_time >= 1.0:
436         last_time = current_time
437         gps.update()
438
439     # user is on the home screen
440     if on_home_screen:
441         display_time_of_day()
442         display_date()
443
444         # draw home screen buttons
445         for image, position in home_screen_buttons.items():
446             rect = image.get_rect(center=position)
447             screen.blit(image, rect)
448
449         # check if buttons were pressed
450         for event in pygame.event.get():
451             if (event.type is MOUSEBUTTONUP):
452                 pos = pygame.mouse.get_pos()
453                 for (image, rect) in home_screen_button_rects.items():
454                     if (rect.collidepoint(pos)):
455                         if (image == bike): # enter bike screen
456                             on_home_screen = on_history_screen = on_summary_screen = on_g
457                             on_bike_screen = True
458                             screen.fill(BLACK) # erase workspace
459                         elif (image == history): # enter history screen
460                             on_home_screen = on_bike_screen = on_summary_screen = on_grap
461                             on_history_screen = True
462                             activities = os.listdir('/home/pi/final_project/activities/')
463                             activities.sort() # list from least to most recent
464                             screen_num = len(activities)-1 # always start by showing most
465                             screen.fill(BLACK) # erase workspace
466                         elif (image == summary): # enter summary screen
467                             on_home_screen = on_bike_screen = on_history_screen = on_grap
468                             on_summary_screen = True
469                             summaries = os.listdir('/home/pi/final_project/summaries/') #
470                             summaries.sort() # list from least to most recent
471                             screen_num = len(summaries)-1 # always start by showing most
472                             screen.fill(BLACK) # erase workspace
473
474     # user is on the bike screen
475     if on_bike_screen:
476
477         if not gps.has_fix: # if GPS not connected, tell user to move
478             tell_user_to_move()
```

```
479
480             # draw back button
481             for text, position in bike_screen_buttons.items():
482                 if text == 'back':
483                     text_surface = small_font.render(text, True, WHITE)
484                     rect = text_surface.get_rect(center=position)
485                     screen.blit(text_surface, rect)
486
487             # check if back button was pressed
488             for event in pygame.event.get():
489                 if (event.type is MOUSEBUTTONUP):
490                     pos = pygame.mouse.get_pos()
491                     for (text, rect) in bike_screen_button_rects.items():
492                         if (text != 'start') and (text != 'pause') and (text != 'save') a
493                             if (text == 'back'): # press back button to return to home sc
494                                 on_home_screen = True
495                                 on_bike_screen = on_history_screen = on_summary_screen =
496                                 screen.fill(BLACK) # erase workspace
497         else:
498             display_data_labels()
499             display_data()
500             update_altitude()
501             update_speed()
502
503             # if waiting for user to start the activity
504             if not start:
505
506                 # draw back and start buttons
507                 for text, position in bike_screen_buttons.items():
508                     if text != 'pause' and text != 'save':
509                         text_surface = small_font.render(text, True, WHITE)
510                         rect = text_surface.get_rect(center=position)
511                         screen.blit(text_surface, rect)
512
513                 # check if buttons were pressed
514                 for event in pygame.event.get():
515                     if (event.type is MOUSEBUTTONUP):
516                         pos = pygame.mouse.get_pos()
517                         for (text, rect) in bike_screen_button_rects.items():
518                             if (text != 'pause') and text != 'save' and (rect.collidepoin
519                                 if (text == 'back'): # press back button to return to hom
520                                     on_home_screen = True
521                                     on_bike_screen = on_history_screen = on_summary_scree
522                                     screen.fill(BLACK) # erase workspace
523                                 elif (text == 'start'): # start recording activity
524                                     start = True
525                                     ref_time = time.monotonic() # store current time to b
526
527             else: # user has started the activity
528                 if not pause: # activity is not paused
529                     update_elapsed_time()
530                     update_distance()
531                     update_elevation_gain()
532
533                     # draw save and pause buttons buttons
534                     for text, position in bike_screen_buttons.items():
535                         if text != 'start' and text != 'back':
536                             text_surface = small_font.render(text, True, WHITE)
537                             rect = text_surface.get_rect(center=position)
538                             screen.blit(text_surface, rect)
```

```
539
540                     # check if buttons were pressed
541                     for event in pygame.event.get():
542                         if (event.type is MOUSEBUTTONUP):
543                             pos = pygame.mouse.get_pos()
544                             for (text, rect) in bike_screen_button_rects.items():
545                                 if (text != 'start') and (text != 'back') and (rect.colli
546                                     if (text == 'save'): # save activity, reset system, a
547                                         write_activity_to_file()
548                                         reset_system()
549                                         activities = os.listdir('/home/pi/final_project/a
550                                         activities.sort() # show activities from least to
551                                         summarize_data()
552                                         screen.fill(BLACK) # erase workspace
553                                     elif (text == 'pause'): # pause recording of activity
554                                         pause = True
555
556
557                 else: # activity is paused
558                     # draw save and start buttons
559                     for text, position in bike_screen_buttons.items():
560                         if text != 'pause' and text != 'back':
561                             text_surface = small_font.render(text, True, WHITE)
562                             rect = text_surface.get_rect(center=position)
563                             screen.blit(text_surface, rect)
564
565                     # check if buttons were pressed
566                     for event in pygame.event.get():
567                         if (event.type is MOUSEBUTTONUP):
568                             pos = pygame.mouse.get_pos()
569                             for (text, rect) in bike_screen_button_rects.items():
570                                 if (text != 'pause') and (text != 'back') and (rect.colli
571                                     if (text == 'save'): # save activity, reset system, a
572                                         write_activity_to_file()
573                                         reset_system()
574                                         activities = os.listdir('/home/pi/final_project/a
575                                         activities.sort() # show activities from least to
576                                         summarize_data()
577                                         screen.fill(BLACK) # erase workspace
578                                     elif (text == 'start'): # start recording again
579                                         pause = False
580                                         prev_time = int(elapsed_time[:2])*3600 + int(elap
581                                         ref_time = time.monotonic()
582                                         first_val = True
583                                         first_val_alt = True
584
585     if on_history_screen: # users can go here to view past activities
586
587         # draw arrow buttons
588         for image, position in arrow_buttons.items():
589             rect = image.get_rect(center=position)
590             screen.blit(image, rect)
591
592         # check if buttons were pressed
593         for event in pygame.event.get():
594             if (event.type is MOUSEBUTTONUP):
595                 pos = pygame.mouse.get_pos()
596                 for (image, rect) in arrow_button_rects.items():
597                     if (rect.collidepoint(pos)):
598                         if (image == left_arrow) and (len(activities) > 0): # view less r
```

```
                            screen_num = (screen_num-1)%len(activities)
                        elif (image == right_arrow) and (len(activities) > 0): # view mor
                            screen_num = (screen_num+1)%len(activities)
                        elif (image == home): # return to home screen
                                on_home_screen = True
                                on_bike_screen = on_history_screen = on_summary_screen =
                                screen.fill(BLACK) # erase workspace

        if (len(activities) > 0): # if there is at least one activity, display past activ
            with open('/home/pi/final_project/activities/' + activities[screen_num], 'r')
                lines = file.readlines()
            y = 0
            for line in lines:
                text_surface = small_font.render(line[:-1], True, WHITE)
                rect = text_surface.get_rect(topleft=(0,y))
                screen.blit(text_surface, rect)
                y += 33

    if on_summary_screen: # users can go here to view summarized data from past activitie

        if (len(summaries) > 0): # if there is at least one summary, display past summari
            with open('/home/pi/final_project/summaries/' + summaries[screen_num], 'r') a
                lines = file.readlines()
            y = 0
            for line in lines:
                if y == 0: # underline date range of the week
                    small_font.set_underline(True)
                    text_surface = small_font.render(line[:-1], True, WHITE)
                    rect = text_surface.get_rect(center=(160,15))
                    small_font.set_underline(False)
                else:
                    text_surface = small_font.render(line[:-1], True, WHITE)
                    rect = text_surface.get_rect(topleft=(0,y))
                screen.blit(text_surface, rect)
                y += 38

        # draw arrow buttons
        for image, position in arrow_buttons.items():
            rect = image.get_rect(center=position)
            screen.blit(image, rect)

        # draw buttons to select graphs
        for image, position in graph_selection_buttons.items():
            rect = image.get_rect(center=position)
            screen.blit(image, rect)

        # check if buttons were pressed
        for event in pygame.event.get():
            if (event.type is MOUSEBUTTONUP):
                pos = pygame.mouse.get_pos()
                for (image, rect) in arrow_button_rects.items():
                    if (rect.collidepoint(pos)):
                        if (image == left_arrow) and (len(summaries) > 0): # view less re
                            screen_num = (screen_num-1)%len(summaries)
                        elif (image == right_arrow) and (len(summaries) > 0): # view more
                            screen_num = (screen_num+1)%len(summaries)
                        elif (image == home): # return to home screen
                            on_home_screen = True
                            on_bike_screen = on_history_screen = on_summary_screen = on_g
                            screen.fill(BLACK) # erase workspace
```

```
659
660                 for (image, rect) in graph_selection_button_rects.items():
661                     if (rect.collidepoint(pos)):
662                         on_graph_screen = True
663                         on_bike_screen = on_history_screen = on_summary_screen = on_home_
664                         if (image == dist_graph): # display distance graph
665                             graph_path = '/home/pi/final_project/graphs/dist_graphs/'
666                         elif (image == time_graph): # display time graph
667                             graph_path = '/home/pi/final_project/graphs/time_graphs/'
668                         elif (image == elev_graph): # display elevation gain graph
669                             graph_path = '/home/pi/final_project/graphs/elev_graphs/'
670                         graphs = os.listdir(graph_path)
671                         graphs.sort()
672                         screen.fill(BLACK) # erase workspace
673
674     if on_graph_screen: # screen that displays graphical summaries
675
676         # draw arrow buttons
677         for image, position in arrow_buttons.items():
678             rect = image.get_rect(center=position)
679             screen.blit(image, rect)
680
681         # check if buttons were pressed
682         for event in pygame.event.get():
683             if (event.type is MOUSEBUTTONUP):
684                 pos = pygame.mouse.get_pos()
685                 for (image, rect) in arrow_button_rects.items():
686                     if (rect.collidepoint(pos)):
687                         if (image == left_arrow): # view less recent graph
688                             screen_num = (screen_num-1)%len(summaries) # note than len(su
689                         elif (image == right_arrow): # view more recent graph
690                             screen_num = (screen_num+1)%len(summaries)
691                         elif (image == home): # return to summary screen
692                             on_summary_screen = True
693                             on_bike_screen = on_history_screen = on_home_screen = on_grap
694                             screen.fill(BLACK) # erase workspace
695
696         graph = pygame.image.load(graph_path + graphs[screen_num])
697         screen.blit(graph, graph.get_rect(center=(160,100)))
698
699     pygame.display.flip() # display everything that has been blitted
700 GPIO.cleanup() # always clean up GPIO when program terminates
```

# Work Distribution

*Photo of Maanav (left) and Morgan (right).*

Both team members were actively involved in all parts of this project. Almost all of the code was written doing pair programming, and tests were also conducted together. Both members were also involved in writing all sections of this report. Morgan focused a bit more on discussing the distance calculation algorithm and history screen. Maanav focused a bit more on discussing the implementation of the summary and graph screens.