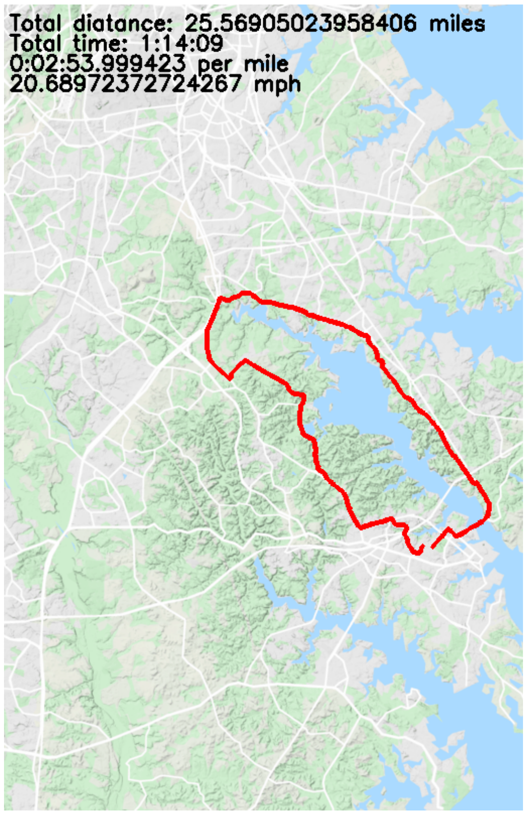
Lab 7: GPS Tracking

Using a fitness tracking device like a Garmin, FitBit, or even just your phone, it's easy to upload your activities and look where you went, how far, and how fast. Popular websites like Strava are happy to collect this data and display it to you to share with your friends or (sometimes with unfortunate consequences) even the whole world.



```
Total distance: 25.56905023958406 miles
Total time: 1:14:09
0:02:53.999423 per mile
20.68972372724267 mph
```

In this lab, you will write programs to read and process a track of GPS positions (like one that might be created by a fitness tracker), figure out things like total distance and average speed, and finally (and optionally) make a picture with all of this drawn on a map.

You will get plenty of practice writing **functions**, reading **files**, and using **lists**.

Step 0: Directory Setup

1. Create a new folder for this lab (lab7), and `cd` into that folder on the command line.
2. Download the zip file gpsdata.zip, which contains two example runs and two example rides around the Annapolis area. Unzip it on Windows, or from the command line by running:

   ```
   unzip gpsdata.zip
   ```

   If it gives you an error, just run their suggested unzip install command first!

Step 1: Distance on a globe (40pts)

The data we will be looking at is organized by GPS coordinates, which are pairs of numbers representing the latitude and longitude of a location, in degrees (not radians!).

Your first task will be to create a function `gps_dist` that takes in four numbers for the starting longitude and latitude and the ending longitude and latitude, and uses the Haversine formula to compute the distance between them, in miles.

Here is the Haversine distance formula:

$$d = 2r*\arcsin\left(\sqrt{\textrm{hav}(\varphi_2-\varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\textrm{hav}(\lambda_2-\lambda_1)}\right)$$

where

- \(d\) is the distance between the two points on the globe, in miles
- \(r\) is the radius of the earth, in miles. Use the followng approximation in your code:

```
EARTH_RADIUS = 3958.7613
```

- \(\varphi_1,\lambda_1\) are the latitude and longitude of the starting point, **in radians**
- \(\varphi_2,\lambda_2\) are the latitude and longitude of the ending point, **in radians**
- hav is the haversine of an angle \(\theta\) in radians, defined by \[\textrm{hav}(\theta) = \frac{1-\cos(\theta)}{2}\]

You should create two helper functions in your program that look like this:

```
def hav(theta):
    """ Fill in the definition of hav() """

def gps_dist(lat1, lon1, lat2, lon2):
    """ Fill in with calls to the hav() function and using the formula above """
```

To compute things like square roots and sines, you use the math library in Python. At the top of your program, `import math`, and then use functions such as `math.radians()` and `math.sin()`. Check out the documentation here to find the functions you need, or use the built-in `help()` from Python's interpreter on the command line.

Don't forget that you need to convert from degrees to radians first! (hint: your input lat/lon coordinates are in degrees)

Your job: Distance from Annapolis

In a file called **dist.py**, write a program which uses your two functions to compute the distance from Annapolis to any other point on Earth. Use the location of Tecumseh as the starting point for Annapolis:

```
TECUMSEH_LAT = 38.982439
TECUMSEH_LON = -76.484226
```

Here is how your program should work. (This example calculates the distance to Mission Beach in San Diego.)

```
$ python3 dist.py
Ending latitude: 32.77
Ending longitude: -117.25
Distance: 2302.678022262103 miles
```

Step 2: Compute the total distance of a GPS track (85pts)

Copy your **dist.py** to a new program **calc.py**. You should keep the same two functions you wrote, but delete the rest of it about computing the distance from the Tecumseh statue.

Now you will read in GPS locations from a text file, one per line, and compute the total distance between all the points in a GPS track.

Type `head run1.txt` to see what the first few lines in the file look like:

```
38.923082 -76.560321 2017-04-02 12:00:35
38.923078 -76.560335 2017-04-02 12:00:36
38.923025 -76.560567 2017-04-02 12:00:44
38.922975 -76.560822 2017-04-02 12:00:51
38.922922 -76.561058 2017-04-02 12:00:58
38.922904 -76.561303 2017-04-02 12:01:05
38.922893 -76.561539 2017-04-02 12:01:12
38.922852 -76.561764 2017-04-02 12:01:18
```

```
38.922827 -76.561997 2017-04-02 12:01:24
38.922805 -76.562227 2017-04-02 12:01:30
```

Each line contains a latitude value (single **float** number), longitude value, and then a date and time.

You need to write a program which reads in the name of a txt file, then uses Python functions such as **open()** and **split()** to process each line in the file, extracting the numeric values for the latitude and longitude on each line.

Your program should then use the `gps_dist` function you wrote from part 1 to compute and add up the distances between each pair of consecutive points in the file. That is, you will add up the distance from point0 to point1, plus the distance from point1 to point2, plus the distance from point2 to point3, plus the distance from point3 to point4, etc.

Hints: There are many ways to do this! One way is to start by creating a list of all the latitude numbers and a list of all the longitude numbers, and then write a second loop which computes all the distances and adds them up. The tricky thing is that your second loop will need to run *one fewer time* than the first loop. For example, if the file contains 10 points, then there are only 9 *pairs* of points to get the distances. Think about it, try it, experiment, and debug to get it right!

Your job: Total distance

Here is an example running your **calc.py** program:

```
$ python3 calc.py
Filename: run1.txt
Total distance: 10.04414461844762 miles
```

If you're having trouble getting this correct distance, try the shorter test file and debug from that first:

```
$ python3 calc.py
Filename: run1-test.txt
Total distance: 0.10459822278285695 miles
```

Step 3: Incorporate the time (100 pts)

Now you will extend your **calc.py** program to account for the time on each line, and use that to get the total speed.

We will use the deteutil library to convert a date/time from a string into numeric values. At the top of your `calc.py` program, add this line:

```
from dateutil import parser
```

Now you can use the function `parser.parse()` to convert a string representing a date and time into a "datetime object" that breaks out the different aspects like year, date, hour, minute, and seconds. For example, calling `dt = parser.parse('2020-02-05 09:30:00')` creates a datetime object in the `dt` variable for you to then later use for date/time math!

For instance, you can *subtract* one datetime object from another to get the difference in how much time elapsed! Here is an example that creates two datetime objects, takes the difference, and prints out the total number of hours in the difference. (Notice that there are 3600 seconds in one hour!)

```
from dateutil import parser
t1 = parser.parse('2020-02-05 09:30:00')
t2 = parser.parse('2020-02-15 15:00:00')
diff = t2 - t1
hours = diff.total_seconds() / 3600
```

```
print(diff)           # 10 days, 5:30:00
print(hours, 'hours') # 245.5 hours
```

F

a

Y

M

might want for running), and the miles per hour (distanceInMiles / totalTimeInSeconds * 3600)).

Here is an example run showing how your program should work:

```
$ python3 calc.py
Filename: ride1.txt
Total distance: 25.56905023958406 miles
Total time: 1:14:09
Average rate: 0:02:53.999423 per mile
Average speed: 20.68972372724267 mph
```

What to turn in

Visit the submit website and upload `dist.py`, `calc.py`, and (if you do the last part below) `map.py` to Lab07 for grading.

Step 4 (Optional, but fun): Use your own gpx file

The files that are actually used by your Garmin device or Strava are usually stored in a format called **gpx**, which has the same latitude/longitude and time information of the .txt files you've been using, so far, but in a more structured format and with some extra information thrown in.

Included in the zip file you downloaded is a Python program **gpx2txt.py** which reads in a gpx file on the command-line and converts it to .txt like we've been using for this lab. So, if you go into favorite fitness tracker and download data from your own run or ride as a .gpx file, you can use this to convert it to .txt and then run your **calc.py** program on your own stuff.

You will need an XML library for Python, so run the following command to install it:

```
sudo apt install python3-lxml
```

And then here's how the converter program works:

```
$ python3 gpx2txt.py ride1.gpx ride1.txt
```

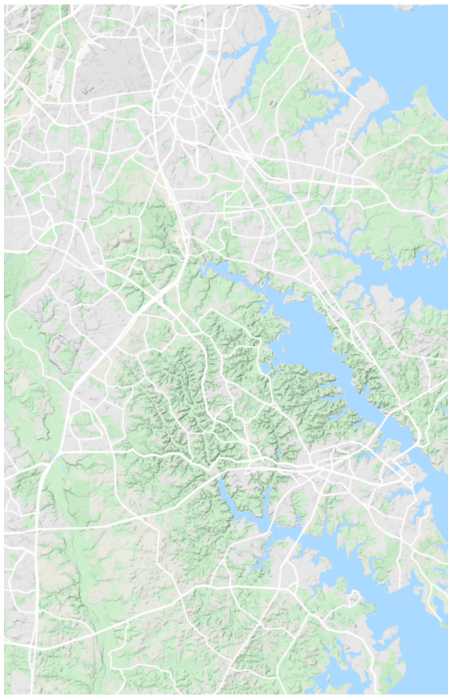
will read from the first file (in gpx format) and write to the second file (in txt format).

If you're curious, you can look at how the `gpx2txt.py` program works - it's only about 20 lines of code, but uses some things we haven't learned about in class yet.

Step 5 (Optional, if you have time): Draw it on the map

Copy your **calc.py** to **map.py** for this part.

The last step is to draw the map of your route on top of the blank map of the Annapolis area that's included in the zip file as `annapolis.png`:



For this program, you will do all the same calculations like before, but in addition, will draw a series of line segments on the map between each pair of points, to make an image like what you see at the beginning of this lab.

To know where to draw the points, you should add in these constants to the top of your **map.py** program, which represent the boundaries of the `annapolis.png` image in terms of latitude and longitude:

```
MAP_MINLAT = 38.874463110537214
MAP_MAXLAT = 39.207782959371684
MAP_MINLON = -76.73744201660156
MAP_MAXLON = -76.46415710449219
```

Now, your **map.py** program should start by getting the name of a .txt file to read in, just like before, but should also open the `annapolis.png` file as a cv2 image at the beginning. (You might want to look back at Lab 2 for a reminder of how that works.)

Then in addition to calculating the distances between each consecutive pair of points, your program should also add a line onto the image for the path between each pair of points.

To add each line, you call the `cv2.line` function line this, which draws a red line on the image called `img` with thickness 4 from `pt1` to `pt2`:

```
cv2.line(img, pt1, pt2, (0, 0, 255), 4)
```

Now the tricky part is: how to you convert the original points from the file, in terms of GPS coordinates, to the points for the image, which are in terms of pixel locations? I recommend you **write a function** to help out, which takes in a longitude and latitude value in GPS coordinates, and returns a CV2 point location in the image. Here are some tips to help you think about this function:

- The point you return for CV2 needs to be a tuple of `(x_coord, y_coord)`, where both numbers are integers up to the size of the image.
- You can write `img.shape` to get the dimensions of the map image as a tuple `(y_dim, x_dim, 3)`. Yes, the dimensions in the opposite order from what CV2 needs!
- The point (0,0) in terms of pixels is at the top-left of the image.
- In terms of GPS coordinates, the x-dimension is longitude and the y-dimension is latitude. So the top-left point has the smallest longitude (`MAP_MINLON`) but the largest latitude (`MAP_MAXLAT`).
- You need to take any point in the middle of the image, with a given latitude and longitude, and convert to pixel coordinates. For each dimension (longitude/x and latitude/y), you will want to use the MIN and MAX values of the map image to find out where your coordinate fits as a percentage, then multiply this by x/y dimenstions of the image to get a pixel coordinate.

This function won't be very long, but it is complicated because you need to think carefully about how the math should work. Nothing fancy is required from the math library, just plus, minus, division, and multiplication - the tough part is putting them together in the correct way. So work carefully and try out many small examples (like, converting a single coordinate) before you try to put everything together.

Finally, once you have the path drawn correctly, you should also add the text onto the image too. The code to do that looks like:

```
cv2.putText(img, "display this text", pt, cv2.FONT_HERSHEY_SIMPLEX, .7, (0,0,0), 2)
```

where `pt` is a tuple (x,y) of pixel coordinates for the bottom-left of where the text is displayed. (In the above example, .7 is a scaling factor for the size of the text, `(0,0,0)` is for the color black, and the final `2` is the thickness of the text.)

Your job: A program to create and display the map

Once you are finished, you should be able to run your program and give it a filename like `ride1.txt` in order to pop up an image like you see at the top of this page. Look back at Lab 2 for a reminder of how to create a window to display a CV2 image and wait for the user to type a key before exiting the program.