COMP125 2014: Assignment 3 - Processing GPS Tracks

Due: **Week 13: 5pm, Monday 10th November 2014**

Updates

- Initial version.

This assignment builds on Assignment 2 where you wrote classes to read GPS data from a CSV file and provide summary information. It generalises the implementation a little and adds some new capabilities to the classes we've implemented.

If you did not manage to pass all tests in Assignment 2 you will get a second chance to do so this time around. If you pass the corresponding tests this time around, you will gain the marks that you missed first time around. This assignment is intended to be an extension of the last on, and if you didn't finish that work you should concentrate on that before looking at the extension work. The only change this time around is that we will use `ArrayList` to store the data rather than an array (see below).

A starter pack is available at
`https://github.com/stevecassidy/comp125-2014-gpstracks/archive/master.zip`. This is an updated version of the pack for the second assignment. It contains new tests and some extra test data. You may want to copy these over to your existing project or copy your existing code into this project.

All queries to Steve.Cassidy@mq.edu.au or via the Discussion Forum on iLearn.

1 Requirements

1.1 Use of `ArrayList`

The first change from Assignment 2 is to make use of `ArrayList` to store waypoints in the `Track` class and tracks in the `TrackLog` class. This is an internal change to the implementation of these classes and doesn't change the external interface – except that we can now remove the limit of 1000 waypoints per track and 20 tracks per track log.

We have provided an updated version of the tests from Assignment 2 that have updated distance and elevation gain measures for the test tracks. Apart from these changes, the tests are the same. Passing the tests will gain you the marks from Assignment 2 if you didn't pass them before. There are also a few additional marks for making the change to `ArrayList`.

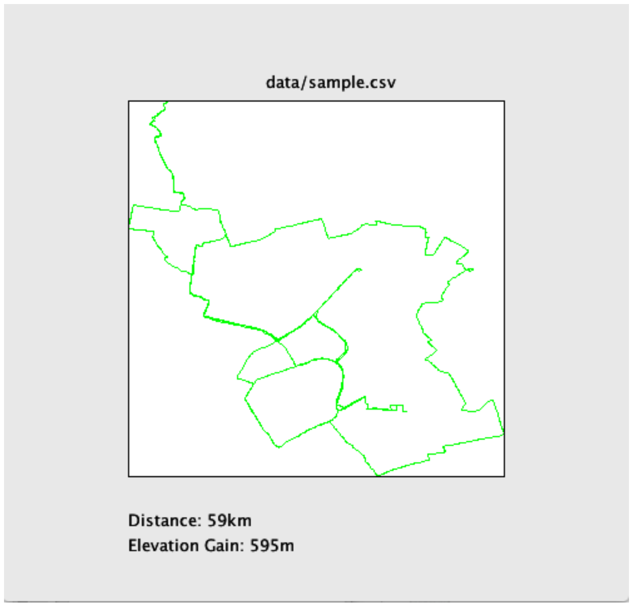
1.2 Default Constructors for `Track` and `TrackLog`

To implement the following methods it is useful to be able to make an empty track that we can add waypoints to from our code using the `add` method. Similarly we want to be able to make an empty track log and add tracks. Your code should implement these default constructors (that take no arguments). We have provided tests for these in the updated JUnit tests.

1.3 Drawing Tracks

A natural operation on a track is to render it on the screen - even better if we can overlay it on a map. In the final task for this assignment you will write code to draw a track on the screen using some utility code that we have provided.

Graphics in Java is a little more complex than in Processing because we are dealing with a general purpose GUI toolkit rather than a specialised drawing library. We don't want to have to worry about all of the detail and so we have provided a simple framework that will allow you to just write drawing commands to generate the display of your track. We have written a class `TrackFrame` that generates a blank canvas for your drawing and calls the `draw` method of a track to render it on the screen. This means that you only need to write the `draw` method on the `Track` class to get this to work. You don't need to modify the `TrackFrame` class at all.



Distance: 59km
Elevation Gain: 595m

Figure 1: An example of the output from the `draw` method on the track `sample.csv`.

The following code from the `main` method in `TrackFrame` shows how to use the class:

```java
Track  track = new Track("data/sample.csv");

TrackFrame frame = new TrackFrame(track);

frame.setSize(500,500);

// Make the window show on the screen.
frame.setVisible(true);
```

This code initialises the `TrackFrame` with a `Track` instance. When the frame is redrawn, it will call the `draw` method on this track which is responsible for updating the display. The `draw` method is passed a `Graphics` object that can be used to call the drawing methods and the width and height of the window. Drawing with the `Graphics` class is similar to using Processing; the coordinate system is the same with (0,0) at the top left of the window. You can find a list of the drawing methods here: `http://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html`

An example `draw` method is:

```java
public void draw(Graphics g, int width, int height) {

    // some variables to help with coordinate calculations
    final int border = 100;
    int iwidth = width - 2*border;
    int iheight = height - 2*border;

    // draw a white rectangle
    g.setColor(Color.white);
    g.fillRect(border, border, iwidth, iheight);

    // draw a black border
    g.setColor(Color.black);
    g.drawRect(border, border, iwidth, iheight);

    // draw some random lines and circles
    g.setColor(Color.red);
    for(int i=1; i<5; i++) {
       g.drawLine(border, border, width/2, border + iheight/i);
       g.fillOval( width/2, border + iheight/i, 10, 10);
    }

    // draw some text
    g.drawString("Hello␣World", (width/2)-40, border-10);

}
```

Your task is to write code to render a good looking version of the track. An example of the output of my version is shown in Fig. 1, but you can display it any way you want.

You will be assessed in this task on the quality of your code and the completeness of your display. If you need to add extra helper methods to the `Track` class to complete this task you are free to do so.

1.4 Finding Climbs in a Track

Cyclists are interested in how fast they can climb hills and often compare times and average speeds over their favourite climb. As a step towards being able to display data on climbs, we will extend the `Track` class to be able to automatically find all climbs over a certain distance along the track.

A climb is a series of waypoints that continuously increases in height. If we look at a plot of the elevation of the `bobbin-head.csv` track (a ride to Bobbin Head) we can see that there are three major climbs and a few smaller ones.

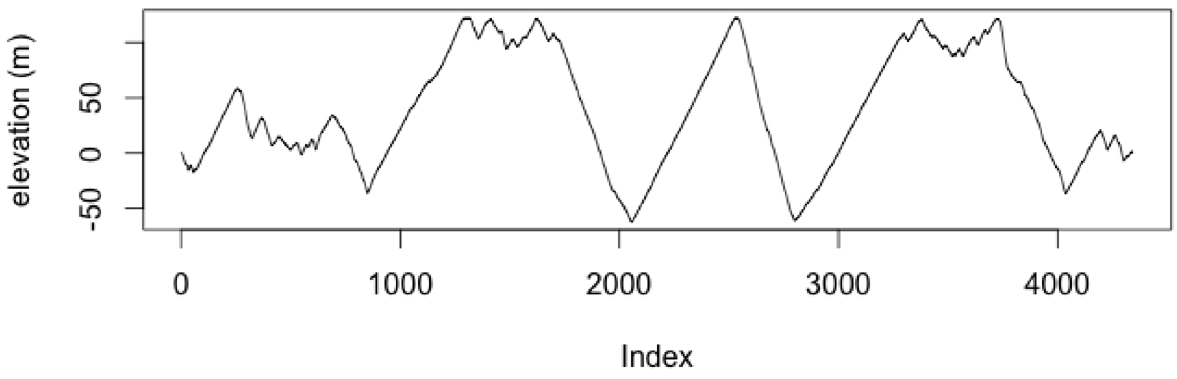


Figure 2: The elevation plot of `bobbin-head.csv` showing three major climbs.

The goal is to write a method `findClimbs` for the `Track` class that returns a list of `Track` instances each of which represents a single climb. Each climb is represented as a new `Track` containing the waypoints from the original track. The method has the signature:

```
    /**
     * Find all of the climbs over a certain length in this track
     *
     * @param minlength minimum length in km for any climb
     * @return an ArrayList of Track objects each representing one climb
     */
    public ArrayList<Track> findClimbs(double minlength) {
        ...
    }
```

This task is made a little more difficult by there not being a single correct solution. It may be relatively easy to come up with an algorithm for finding climbs, but how will you test whether you are getting reasonable answers? Include some comments on this in your documentation and add any suitable tests to the provided JUnit test file `TrackClimbsTest.java`. The provided test just checks that there are 3 climbs over 3km in the `bobbin-head.csv` file.

1.5 Track Distance Measure

Once we have a list of climbs from one track, we can search through other tracks to see if they have been ridden again. One way to do this is to be able to measure the *distance* between two tracks such that tracks that largely overlap will be close to each other while those that don't will be further away.

Measuring distance between two tracks is not as straightforward as measuring distance between two points. There is no simple measure that we can use to compare one sequence of points with another – in fact there are many ways to measure such a distance. This isn't a distance in meters or kilometres, rather it's an abstract measure of how different these tracks are from each other.

One suitable measure is called the Hausdorff distance which measures the distance between all points on the two tracks and for each point in one track finds the shortest distance to any point on the other track. Using these it then finds the longest of these shortest distances. This gives a distance measure such that every point on one path is at most this far from a point on the other line. Details of the algorithm can be found here: `http://cgm.cs.mcgill.ca/~godfried/teaching/cg-projects/98/normand/main.html`.

```
/**
 * Calculate a distance metric between this track and another
 * track.  The metric should be zero for identical tracks,
 * small for tracks that are very similar (eg. that overlap a lot)
 * and larger for more distinct tracks.
 *
 * @param other the other track to compare with
 * @return a similarity measure
 */
public double distance(Track other) {}
```

Again you will need to decide for yourself how you will evaluate your solution to this task. You can implement the Hausdorff distance metric but there are possible extensions to this that you might consider. For example, should the distance metric be symmetrical (a to b equals b to a) - by default Hausdorff is not. Write up your ideas in your documentation and provide any tests that you want to use for your own code. The provided tests in `TrackDistanceTest.java` check that the distance to the same track is zero, that the distance to a subset of a track is positive and that the distance to an unrelated track is larger.

1.6 Challenge

Using the code you've now written you could write an application that reads one track, finds the climbs in that track and then searches through all tracks in a `TrackLog` to find occurrences of the same climbs. The interface to this application is up to you. Your solution should include some documentation on how to run your application and the assumptions you have made in designing it.

2 Assessment

2.1 Documentation/Comments

All code that you submit should have appropriate JavaDoc comments. In this case these should not just describe the inputs and outputs of the method but also the algorithms and techniques that you've used to solve the problem. This documentation forms part of your submission for the assignment. It is where you can tell us how you solved the problems we've set. So, for example, your JavaDoc for `findClimbs` can include a brief statement of the algorithm you used to find climbs and the cases you have tested it with.

2.2 Unit Tests

We have provided some updated unit tests for the base classes in the application. We also provide some basic tests for the `findClimbs` and `distance` methods - however since we can't know the details of your solution, we can't write exact tests. You are encouraged to write additional tests for your code and submit them along with your solution.

If there are aspects of your solution to the `draw` method that can be tested with unit tests rather than a visual check, then you are encouraged to write tests for these also.

2.3 Submitting Your Work

You should submit an exported version of your Eclipse project. Select Export from the file menu, then choose General > Archive File. This should create a zip file containing your source code (it should have the same structure as the zip file you are given to start the project).

Upload the zip file to iLearn in the **Assignment 3 Submission** activity.

2.4 Marking

You have the opportunity of gaining any marks that you missed from Assignment 2 by passing the corresponding tests this time around. The following marks will add to whatever marks you gained in A2:

- (10 marks) Your implementation of the Waypoint class passes our JUnit tests.
- (25 marks) Your implementation of the Track class passess our JUnit tests.
- (25 marks) Your implementation of the TrackLog class passes our JUnit tests.

For this assignment, you will be marked on passing the extended tests but also on the completeness and quality of your implementations of the methods described above.

The assignment is worth 12% of the overall mark for the course but will be marked out of 100 and scaled. The marks are allocated as follows:

- (10 marks) Your code passes all the unit tests on Waypoint, Track and Tracklog using an ArrayList implementation (this is the A3 mark, in addition to any extra A2 marks you get for this).
- (25 marks) For your implementation of `draw`.
- (25 marks) For your implementation of `findClimbs`.
- (20 marks) For your implementation of `distance`.
- (20 marks) For a solution to the challenge problem.

For each implementation requirement above, you will get marks for:

- the completeness of your implementation - you solved the problem
- the quality of your code – readable, clear
- your provided tests
- the documentation (in JavaDoc) of the algorithm you used or any special features of how you solved the problem.