A Basic Pedometer

TODO: Intro

Simplifications

- Ruby instead of native for mobile (Objective-C or Java)
    - A pedometer is a common app built for mobile devices that have hardware/software built in to measure acceleration and gravity.
    - If the mobile device is an iPhone or Android, the pedometer would commonly be written natively for the platform in Objective-C or Java, respectively.
    - Java is verbose, and Objective-C is both verbose and difficult on the eyes for a developer not familiar with it.
    - Additionaly, native mobile APIs are quickly evolving, and code that may be accurate now may not be in several years.
    - Our basic pedometer is written in Ruby for two reasons: to keep the complexities of the language out of the way and allow us to focus on architecture, and to ensure that code for the specifics of native mobile platforms as they are today is not confused with code for data processing and presentation.
- Batch processing instead of real-time
    - A pedometer would rarely be written as a batch processing problem analyzed by a web application, but it has been done this way for the purposes of simplification.
    - The concepts behind our basic pedometer can be extended and directly applied as mobile applications analyzing data in real-time.
    - Data has been collected from an iPhone in two formats, and is being abalzed by our web application in Ruby.
- Error detection can be enhanced
    - Currently we're not counting steps that are too close together.
    - One enhancement would be to discount any steps if there are too many false steps.
    - TODO: More error detection suggestions.
- Many ways to analyze data to count steps
    - There are many methods present to analyze movement data and count steps. Some are more accurate than others in specific instances, for example, day-to-day tracking vs. step counting during a jog.
    - This is just one of many ways.

The Toolchain

- Sinatra web app, using Highcharts to display data.
- This was chosen to be built as a web app because a web app naturally separates the data processing from the presentation.
- Sinatra gives us the ability to demosntrate a fully-functional web app accepting input and presenting output very easily, without worrying about the piping.
- While this project is not intended to show the separation of concerns present in a well-built web app, using Sinatra allows us to naturally segment those concerns and isolate the data processing from the presentation.
- Using Sinatra and Highcharts requires very little additional code and presents our data nicely, so, well, why not have some fancy charts to really satisfy our data craving?

The Platform

TODO: Describe basic flow.

1. Upload data in one of two formats:
    - Combined: Data in what we'll call the **combined** format is user acceleration combined with gravitational acceleration. Data in this format is passed in as x, y, z coordinates, each of which shows combined acceleration in that direction at a point in time.
    - Separated: Data in what we'll call the **separated** format is user acceleration separated from gravitational acceleration. Data in this format is passed in as x, y, z coordinates showing user acceleration is each of the directions, followed by x, y, z coordinates showing gravitational acceleration.
2. Input the following metadata:
    - Sampling rate
    - Actual step count
    - Trial number
    - Method (walking, running, etc.)
    - Gender
    - Height
    - Stride
3. Our program parses the data and outputs:
    - Number of steps taken
    - Distance traveled
    - Time duration

- Charts representing the data in two different stages of parsing

The meat and potatoes of our program is in step 3, where we parse the input data.

Parsing Input Data

The sample data we'll be using here is data collected by an iPhone. Let's look at what our input data will look like in each format.

Combined Format

The first, more rudimentary data format we'll accept is in the combined format. Data in the combined format is simply total acceleration in the x, y, z directions, over time.

"$x1, y1, z1; ...xn, yn, zn;$"

Let's look at what this data looks like when plotted. Below is a small portion of data, sampled 100 times per second, of a person walking with an iPhone in a bag on their shoulder.

TODO: Add plot.

Separated Format

The second data format we'll accept is user acceleration in the x,y,z directions separated from gravitational acceleration in the x,y,z directions, over time:

"$x1_u, y1_u, z1_u|x1_g, y1_g, z1_g; ...xn_u, yn_u, zn_u|xn_g, yn_g, zn_g;$"

Let's look at what this data looks like when plotted. Below is a small portion of data, sampled 100 times per second, of a person walking with an iPhone in a bag on their shoulder.

TODO: Add 2 plots, one for x,y,z user and one for x,y,z gravity.

Making Sense of Data

Looking at our plots, we can start to see a pattern, but we don't have enough, yet, to count steps.

We need to do 3 things to our input data:

1. Parse our text data and extract numerical data.

2. Isolate movement in the direction of gravity to get a single data series resembling a sine wave.
3. Filter our data series to smooth out our sine wave.

These 3 tasks are related, and it makes sense to combine them into one class called a **Parser**.

The Parser Class

TODO: Code from parser.rb

Let's start with the initialize method. Our parser class takes string data as input and stores it in the @data instance variable. It then calls three methods in sequence: parse_raw_data, dot_product_parsed_data, and filter_dot_product_data.

Each method accomplishes one of our three steps above. Let's look at each method individually.

Step 1: Parsing text to extract numerical data (parse_raw_data)

The goal of parse_raw_data is to convert string data to a format we can more easily work with, and store it in @parsed_data. The first line splits the string by semicolon into as many arrays as samples taken, and then splits each individual array by the pipe, storing the result in accl.

We determine the input format by the first element of accl.

- accl in the combined format: $[[”x1, y1, z1”], ...[”xn, yn, zn”]]$
- accl in the separated format: $[[”x1_u, y1_u, z1_u”, ”x1_g, y1_g, z1_g”], ...[”xn_u, yn_u, zn_u”, ”xn_g, yn_g, zn_g”]]$

Starting with step 1 above, our **Parser** class looks like:

TODO: Code block with parser1.rb

Our parser class takes string data as input, that we want to convert to a format we can more easily work with. We'll store this converted data in an instance variable called @parsed_data.

In the initalizer, we ensure the input is a string by explicitly converting it, and then we call parse_raw_data. The goal of this method is to take a single string in the format above, and convert it to a hash to store in @parsed_data.

TODO: Pull out comments from parser1.rb and insert here to describe the method.

Step 2: Isolating movement in the direction of gravity

First, a very small amount of liner algebra 101.

TODO: Short explanation of why the dot product is used to help us isolate movement in the direction of gravity.

Taking the dot product in our Parser class is straightforward. We add a @dot_product_data instance variable, and a method, dot_product_parsed_data, to set that variable. The dot_product_parsed_data method iterates through our @parsed_data hash and calculates the dot product with collect, and sets the result to @dot_product_data.

TODO: Code block from parser2.rb

Step 3: Filtering our data series

Again, back to the mathematics for some signal processing 101.

TODO: Basics of filtering, Chebyshev filter specifically

Following the pattern from steps 1 and 2, we add another instance variable, @filtered_data, to store the filtered data series, and a method, filter_dot_product_data, that we call from the initializer.

TODO: Code block from parser3.rb

The filter_dot_product_data method initalizes the data series by setting the first two elements to 0, and then iterates through the remaining element indeces in @dot_product_data, applying the Chebyshev filter.

Our Parser now takes string data in the separated format, converts it into a more useable format, isolates movement inthe direction of gravity through the dot product operation, and filters the resulting data series to smooth it out.

Our parser class is useable as is. We can take a simpled data series, below, and pass it through out parser:

TODO: Add lines from parser3_test.rb to show functioning parser.

However, our parser only takes data in the separated format. What happens if we only have data in the combined format, and we need to separate it ourselves?

Enhancing our parser to accept combined data

TODO: Explain final changes with parser.rb

TODO: ROUGH OUTLINE OF REMAINING CHAPTER STARTS BELOW

Counting steps

- Now that we have our data in a workable format, we're ready to count steps.
- Discussion around how that isn't the role of the parser, so introduce analyzer class.
- Take out user and device (and therefore distance and time) from Aanalyzer and explain step counting
- Show some working examples through command line of both classes in action.

Adding features to our program

- Introduce User class and discuss
- Introduce Device class and discuss
- Command line examles of all 4 working together

Adding some friendly

- Sinatra layout
- /trials and a basic table with calculations, pulling from the public directory
- ...