Deliverable Iteration #2 – Implementation/Testing Document

For

APP #1 (Group 1-11)

Prepared by Group 11

Alvira Konovalov (40074264)
Hamzah Muhammad (40156621)
William Chittavong (40048632)
Dmytro Chychkov (40034351)
Samuel Chuang (40133237)

March 27, 2022

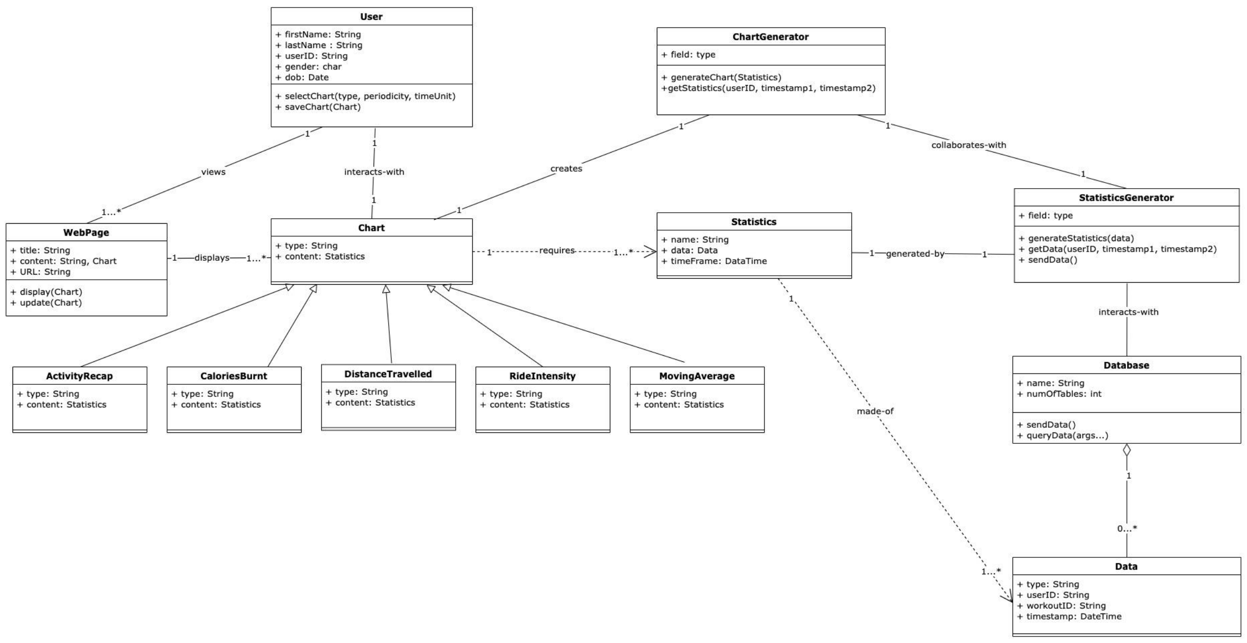
Table of Contents

Revision History

Name	Date	Reason For Change	Version
Alvira Konovalov	Mar 27	- Implemented initial version of Use Case 1 - Developed initial web application/dynamic website - Created and filled-in deliverable #2 document - Helped with UC5	1.1
Hamzah Muhammad	Mar 27	- Modified the content of Use Case 2 from macros to calories to better work with real data given to us in next iteration - Implemented the initial version of Use Case 2	1.1
Samutl Chuang	Mar 27	- Designed mockup design for web application - Slight modification in Use case 3 for it to display the total (sum) distance of the week, month and year. - Implemented the initial version of Use Case 3	1.1
Dmytro Chychkov	Mar 27	- Implemented the initial version of Use Case 4 - Added meter-feet conversion functionality to UC4 - Added Ride Intensity coloring scheme to ease the visualization of UC4	1.1
William Chittavong	Mar 27	- Implemented the initial version of Use Case 5 - Added form for the starting and end dates as well as the window size to be used in calculation.	1.1

Project Features

1. Updated Class Model



2. Modifications

2.1 Use Case 1 - Activity Recap

- Use case 1 offers weekly, monthly, yearly, and all cycling recaps. The daily activity recap was dropped.
- Activity recap is shown in two distance units: kilometres or miles. Hours, minutes and seconds were dropped for now. Time unit will potentially be implemented in the next iteration.

2.2 Use Case 2 - Calories Burnt

- Previous iteration of this use case was a chart displaying macros consumed (% of protein, carbohydrates, fat) based on meal plans entered into the application.
- Upon communication with the statistics team and due to the fact that there was no collection of data on meal plans, this use case will now generate a chart displaying calories burnt per day for every day of a user-selected week based on data calculated from the statistics team.

- Timeframe filter to select which week to showcase as well as timeframe filter to switch between weekly basis and monthly basis has yet to be implemented in this iteration.

2.3 Use Case 3 - Distance Travelled

- Display TOTAL distance travelled instead of just distance travelled.
- Users can select to display between weekly, monthly, or yearly total. (Unimplemented yet)
- Currently, the Distance Travelled chart is able to generate the weekly total by default. It is connected to a mock mySQL database to be able to pull data out and do the necessary calculations.
- Connected system to Google Charts to generate charts.

2.4 Use Case 4 - Elevation / Ride Intensity

- Added feet to meters conversion toggle for graph visualization
- Replaced *time frame* feature for graph visualization with *workout select* to enhance user experience
- Added a color scheme to more easily identify Descending, Ascending and Flat surfaces traversed during the workout
- Added a prototype screenshot button (see User Case 4 full description)

2.5 Use Case 5 - Moving Average

- Currently the Moving Average Chart graph generates the Moving Averages itself. Will have to change to accept JSON object that stores the already calculated Moving Averages. Is not connected to a Statistics Database which was originally supposed to pass the Moving Averages to our chart generator.
- Added form to accept user input of the starting and end dates in the data set. Just as mentioned in Deliverable 1. New addition of input field for changing window size of the moving average. Each submission changes the graph. Still needs form validation in case wrong values are entered.
- The Web Interface does indeed receive the Moving Average Graph generated as specified in Deliverable 1.
- No error messages are displayed if there is a failure to access the database of workouts or Statistics Generator. Once integration is done between the Statistics team and back end teams then error messages will be implemented.

2.6 Github

- Working Github repository is found at: https://github.com/ElviraKonovalov/charts

Major Libraries and Components

Two software stack options are being tested:

Option 1

- *Plotly:* an Open Source Python graphing library
- *Flask*: a Python web framework
- *Pandas:* an Open Source data analysis and manipulation tool
- *HTML, CSS* and *JavaScript*

Option 2

- *Google Charts:* an Open Source chart generator library
- *HTML5/SVG*, *Javascript*

Implementation and Testing

Integration Testing

Our team is currently experimenting with two software stacks. Testing is undergoing to evaluate which stack performs best. The final stack will be determined once integrated with real data received from the Statistics group. At the moment, testing is done on mock data where the data is retrieved from *CSV* files or a *MySQL* database.

Given the circumstances (database, statistics and web interface not available) we had to adjust by adopting the Top-down integration testing strategy. We started with the high-levels of a system (server, mockup web-interface and statistics database) to establish the necessary infrastructure for visualization and testing, which allowed us to dynamically integrate additional modules using the established control flow of architecture structure.

Use cases 1, 4, and 5 are developed using the web framework *Flask* and *Plotly*, a Python Open Source graphing library. Use cases 2 and 3 are implemented using *JavaScript* and *PHP*.

Functional Testing

1. Web Application

The languages *HTML*, *CSS, PHP,* and *JavaScript* were used to construct the basic structure of the Web Interface of Fitness APP #1. The modern CSS framework *W3.CSS* was used to support responsiveness across different devices. Figures below show the initial design of the Web Interface of the Fitness Tracker APP.

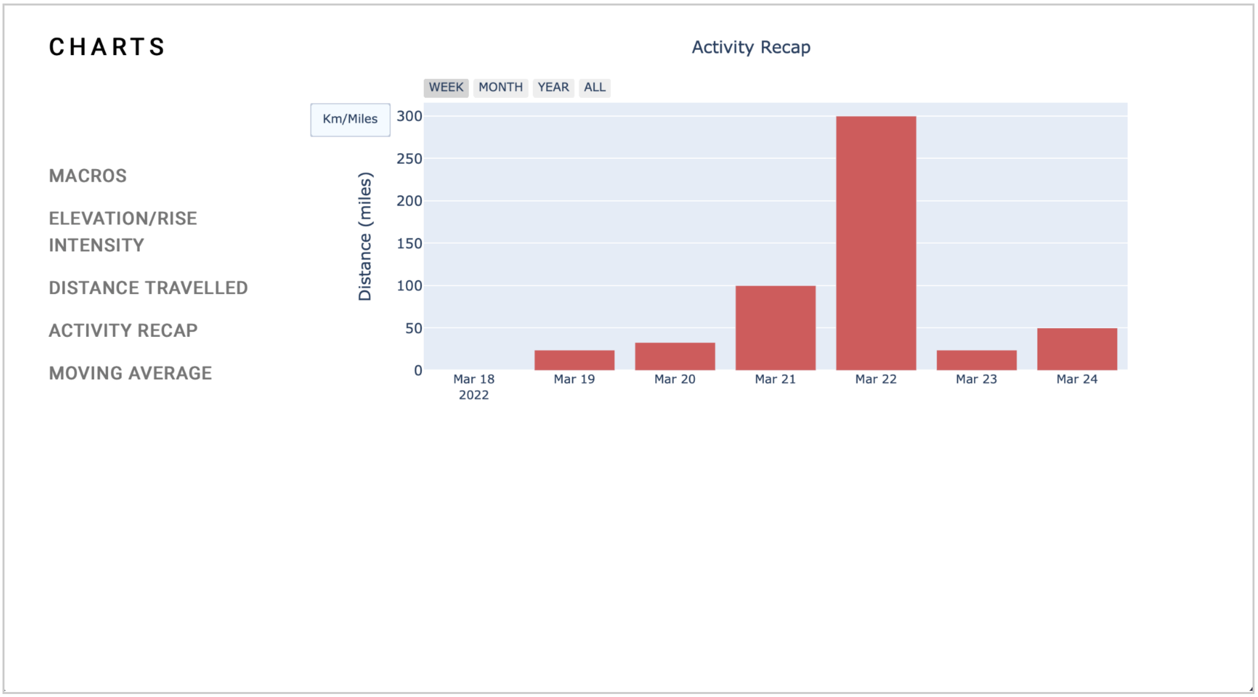


Figure 1: Initial Web Interface design displaying the Activity Recap chart

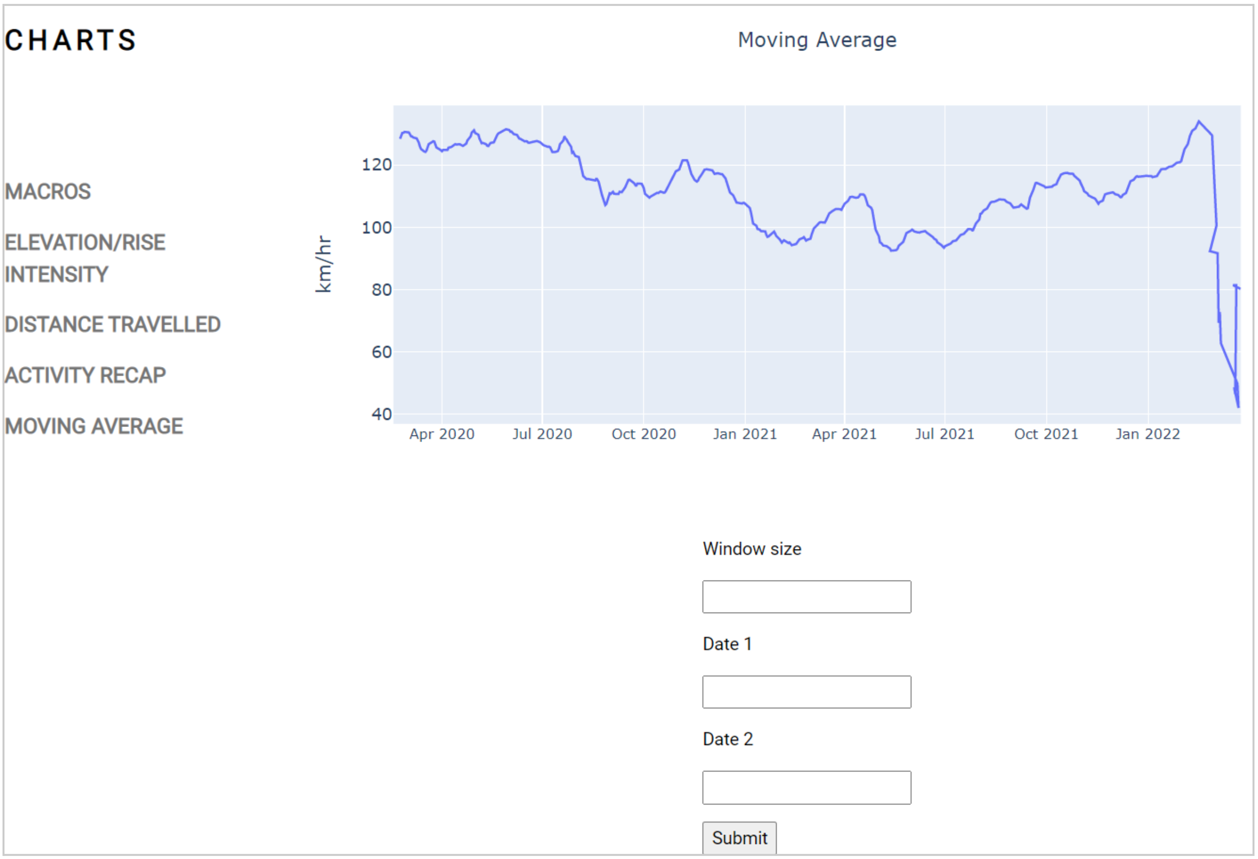


Figure 2: Initial Web Interface design displaying the Moving Average chart

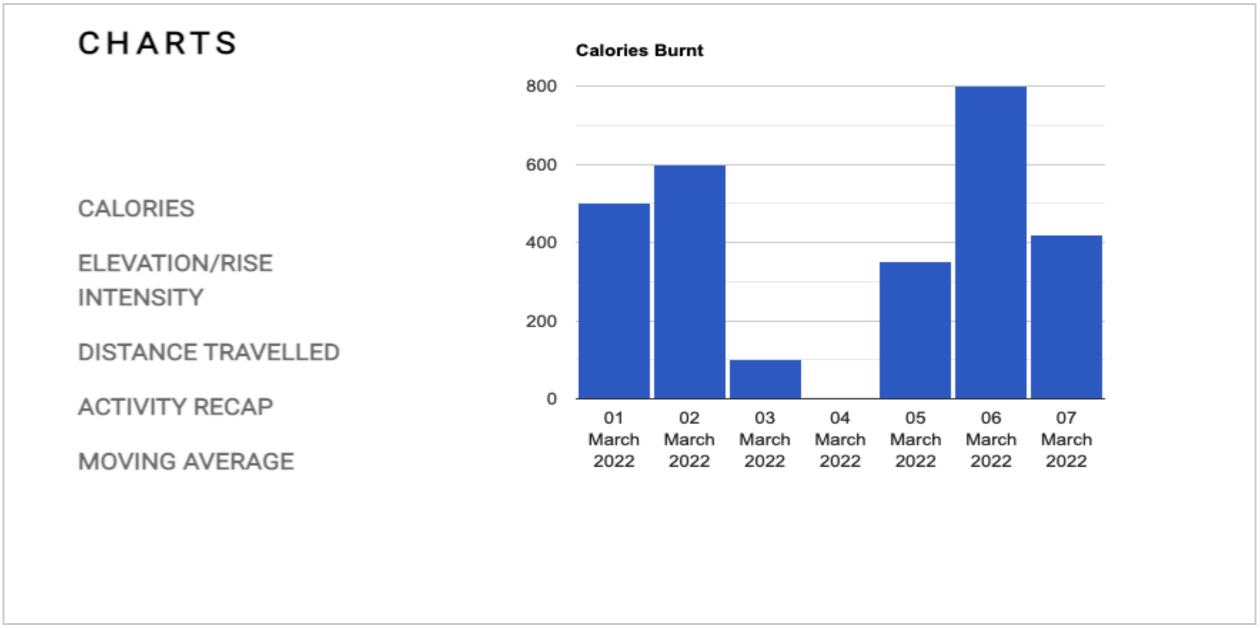


Figure 3: Initial Web Interface design displaying the Calories Burnt chart

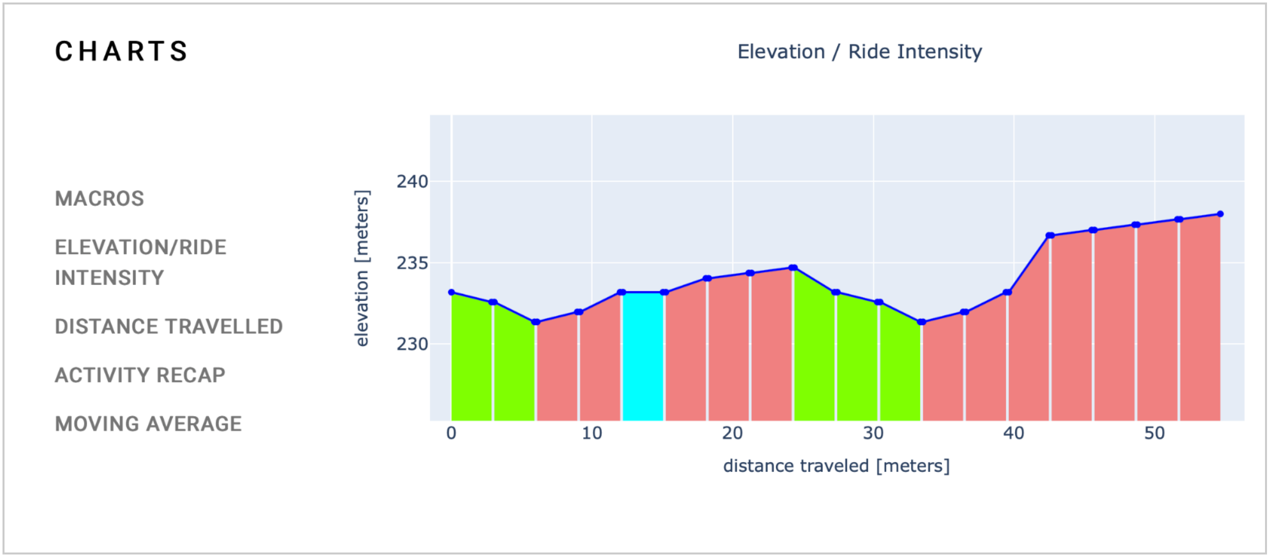


Figure 4: Initial Web Interface design displaying the Elevation/Ride Intensity chart

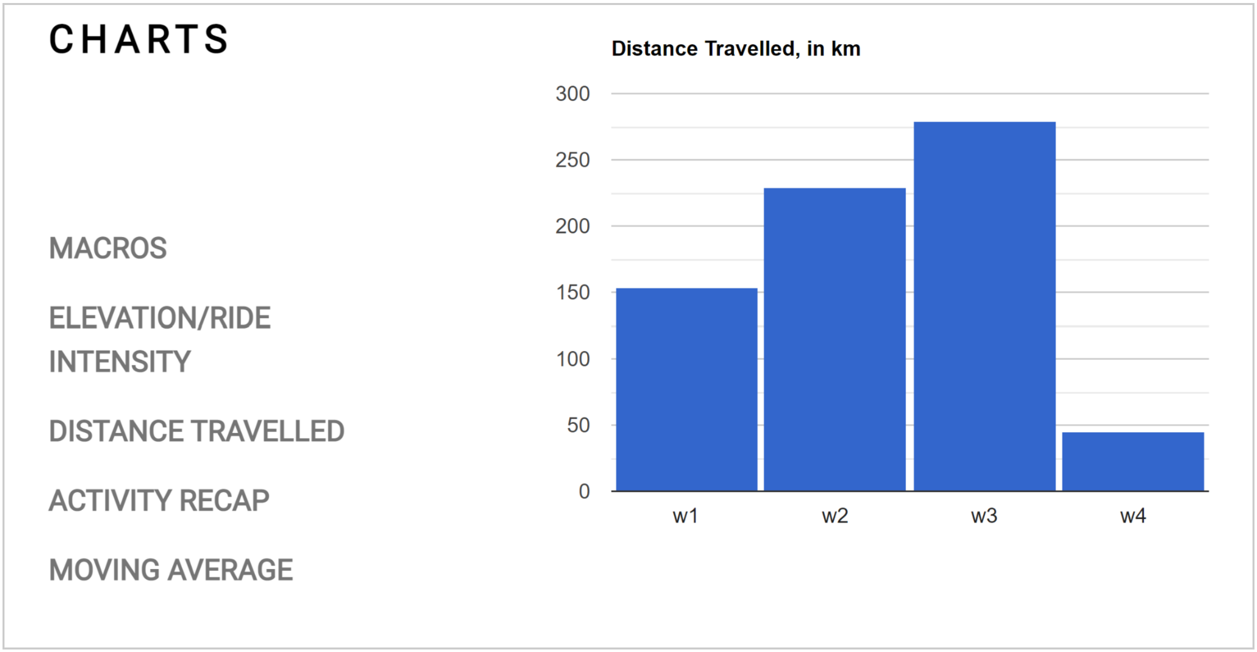
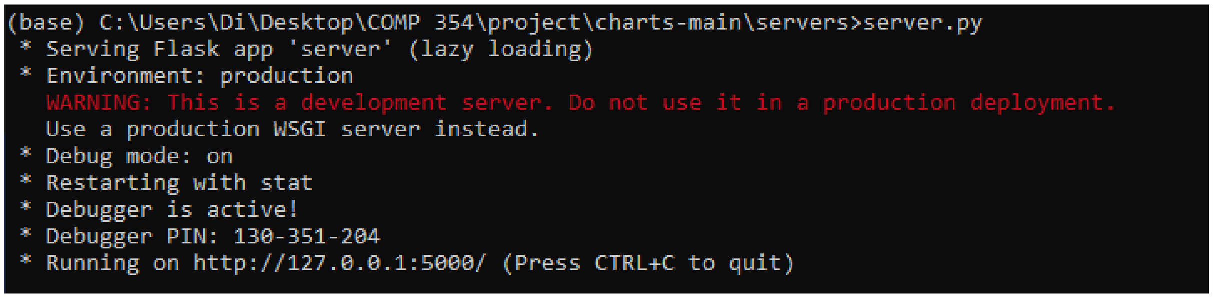


Figure 5: Initial Web Interface design displaying the Distance Travelled chart

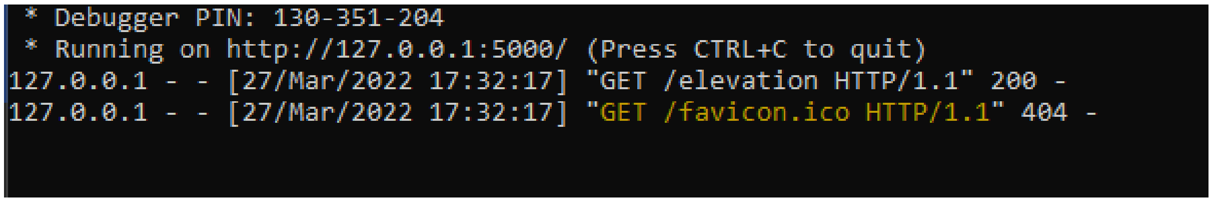
2. Server Debugger and Event handler

Server start:



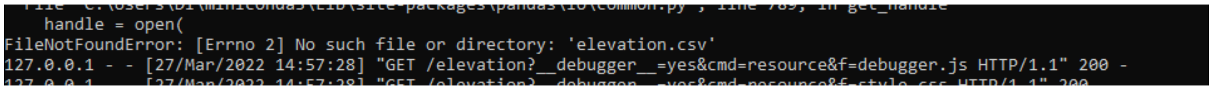
```
(base) C:\Users\Di\Desktop\COMP 354\project\charts-main\servers>server.py
 * Serving Flask app 'server' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 130-351-204
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Functionality/library unreachable/not available (favicon icon generator in this case):



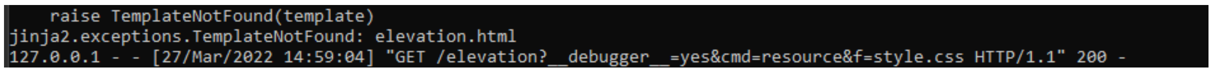
```
 * Debugger PIN: 130-351-204
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [27/Mar/2022 17:32:17] "GET /elevation HTTP/1.1" 200 -
127.0.0.1 - - [27/Mar/2022 17:32:17] "GET /favicon.ico HTTP/1.1" 404 -
```

Missing csv/database unreachable:



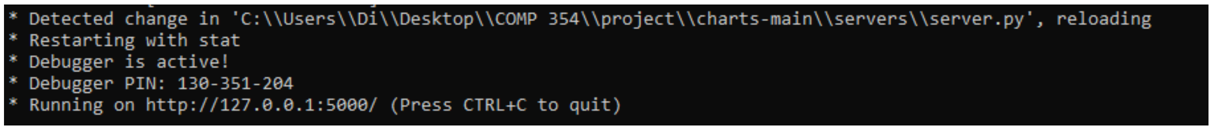
```
    handle = open(
FileNotFoundError: [Errno 2] No such file or directory: 'elevation.csv'
127.0.0.1 - - [27/Mar/2022 14:57:28] "GET /elevation?__debugger__=yes&cmd=resource&f=debugger.js HTTP/1.1" 200 -
```

Missing template:



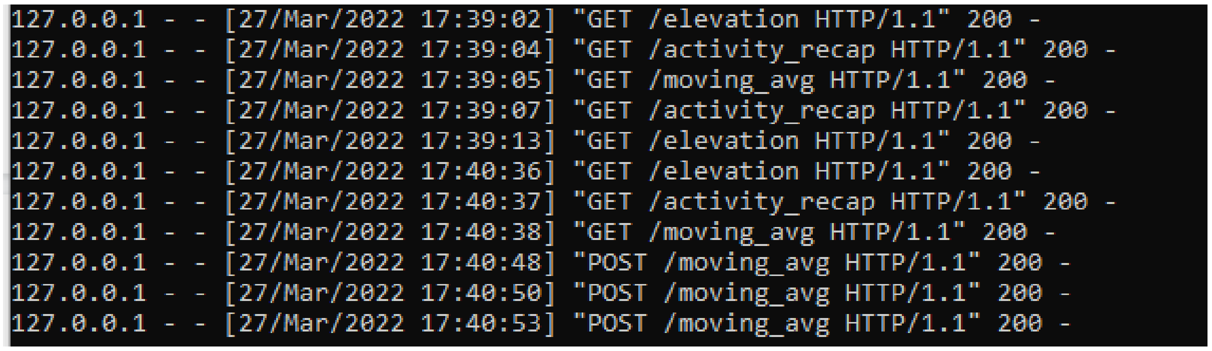
```
    raise TemplateNotFound(template)
jinja2.exceptions.TemplateNotFound: elevation.html
127.0.0.1 - - [27/Mar/2022 14:59:04] "GET /elevation?__debugger__=yes&cmd=resource&f=style.css HTTP/1.1" 200 -
```

Server side update:



```
* Detected change in 'C:\\Users\\Di\\Desktop\\COMP 354\\project\\charts-main\\servers\\server.py', reloading
* Restarting with stat
* Debugger is active!
* Debugger PIN: 130-351-204
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

GET and POST calls tracing:

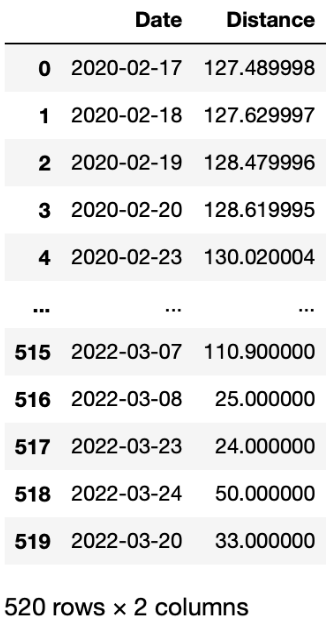


```
127.0.0.1 - - [27/Mar/2022 17:39:02] "GET /elevation HTTP/1.1" 200 -
127.0.0.1 - - [27/Mar/2022 17:39:04] "GET /activity_recap HTTP/1.1" 200 -
127.0.0.1 - - [27/Mar/2022 17:39:05] "GET /moving_avg HTTP/1.1" 200 -
127.0.0.1 - - [27/Mar/2022 17:39:07] "GET /activity_recap HTTP/1.1" 200 -
127.0.0.1 - - [27/Mar/2022 17:39:13] "GET /elevation HTTP/1.1" 200 -
127.0.0.1 - - [27/Mar/2022 17:40:36] "GET /elevation HTTP/1.1" 200 -
127.0.0.1 - - [27/Mar/2022 17:40:37] "GET /activity_recap HTTP/1.1" 200 -
127.0.0.1 - - [27/Mar/2022 17:40:38] "GET /moving_avg HTTP/1.1" 200 -
127.0.0.1 - - [27/Mar/2022 17:40:48] "POST /moving_avg HTTP/1.1" 200 -
127.0.0.1 - - [27/Mar/2022 17:40:50] "POST /moving_avg HTTP/1.1" 200 -
127.0.0.1 - - [27/Mar/2022 17:40:53] "POST /moving_avg HTTP/1.1" 200 -
```

Unit Testing

1. Use Case 1 - Activity Recap

The graphing library *Plotly* was used to generate an interactive chart that displays the activity recap/cycling history. *Flask* is a *Python* web framework that was to embed the chart into the HTML web application. The framework also takes care of updating the chart as new data arrives. The library *Pandas* was used to read the data from the *CSV* file and manipulate it as needed. Mock data stored in a *CSV* file was created and used to test the chart. Figure 6 shows a sample of the mock data.



	Date	Distance
0	2020-02-17	127.489998
1	2020-02-18	127.629997
2	2020-02-19	128.479996
3	2020-02-20	128.619995
4	2020-02-23	130.020004
...	...	...
515	2022-03-07	110.900000
516	2022-03-08	25.000000
517	2022-03-23	24.000000
518	2022-03-24	50.000000
519	2022-03-20	33.000000

520 rows × 2 columns

Figure 6: Sample of mock data used for chart testing

The cycling recap/history can be displayed in terms of weekly cycling recap, monthly cycling recap, yearly cycling recap or a recap of all the cycling done since the first cycling log-in in the App.

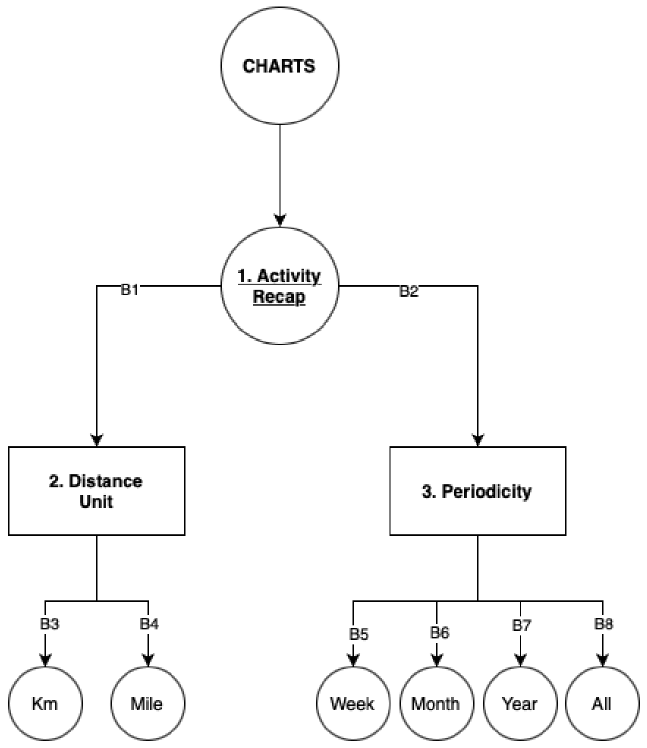


Figure 7: Possible display options

White Box Testing

Regardless of the periodicity and distance unit picked by the user, all statements {1, 2, 3} are triggered. Coverage 3/3 statements = 100% coverage.
Similarly for the branches, each option (test case) goes through B1 and B2 and picks one branch from {B3, B4} and one branch from {B5, B6, B7, B8}. Coverage 4/8 branches = 50% coverage.

Display Option 1: Kilometer or Miles

A 'Km/Miles' toggle button that converts default distance in miles to kilometers (B4 to B2). B4 is the default branch.
The formula below is used to convert the distance from miles to kilometers.

```
y_miles = df.Distance
y_km = df.Distance * 1.60934 # Miles to Km formula
```

When the user selects kilometers as distance units, the `'args'` arguments are replaced in the chart. While, if the user selects miles as distance units, the `'args2'` arguments are replaced.

```
updatemenus = [{
                'buttons': [{'method': 'update',
                             'label': 'Km/Miles',
                             'args': [
                                      # 1. updates y axis to km
                                      {'y': [y_km],
                                       'visible': True},
                                       # 2. Updates y axis label
                                      {'yaxis_title_text':'Distance (km)'},
                                      # 3. which traces are affected
                                      [0, 1],

                                      ],
                             'args2': [
                                       # 1. updates y axis to miles
                                       {'y': [y_miles],
                                       'visible':True},
                                       # 2. updates y axis label
                                      {'yaxis_title_text':'Distance (miles)'},
                                       # 3. which traces are affected
                                       [0, 1]
                                      ]
                              },
                            ],
                'type':'buttons',
                'direction': 'down',
                'showactive': True,}]
fig.update_layout(updatemenus=updatemenus)
```

Test case #1: Figure 8 shows the cycling done throughout the week of March 20, 2022 - March 26, 2022. Default distance is in miles.

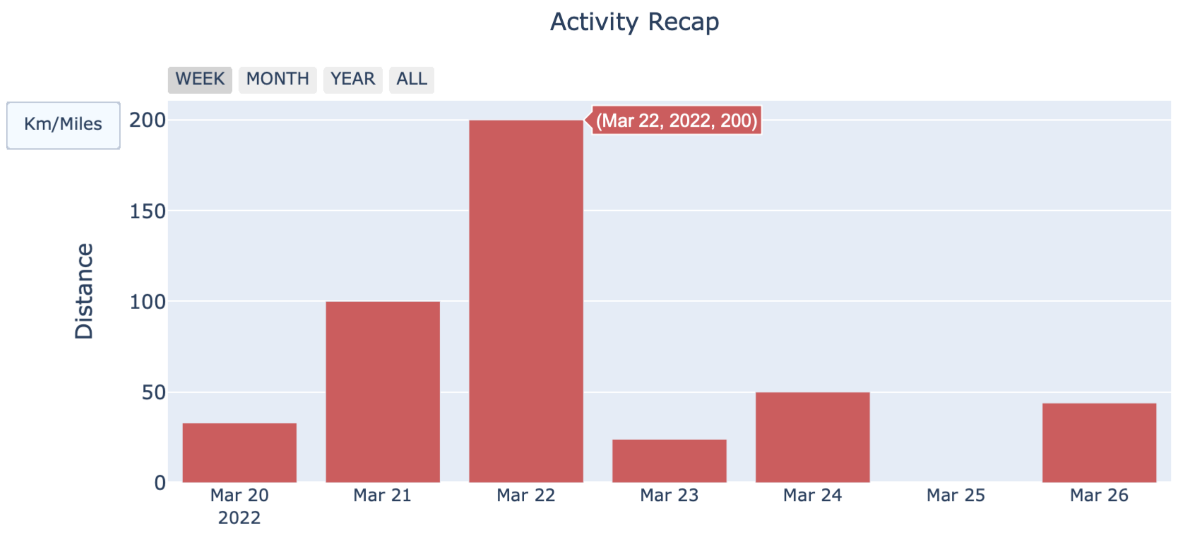


Figure 8: Cycling done throughout the week in miles

Test case #2: The user has the option to display the distance in kilometers rather than in miles.

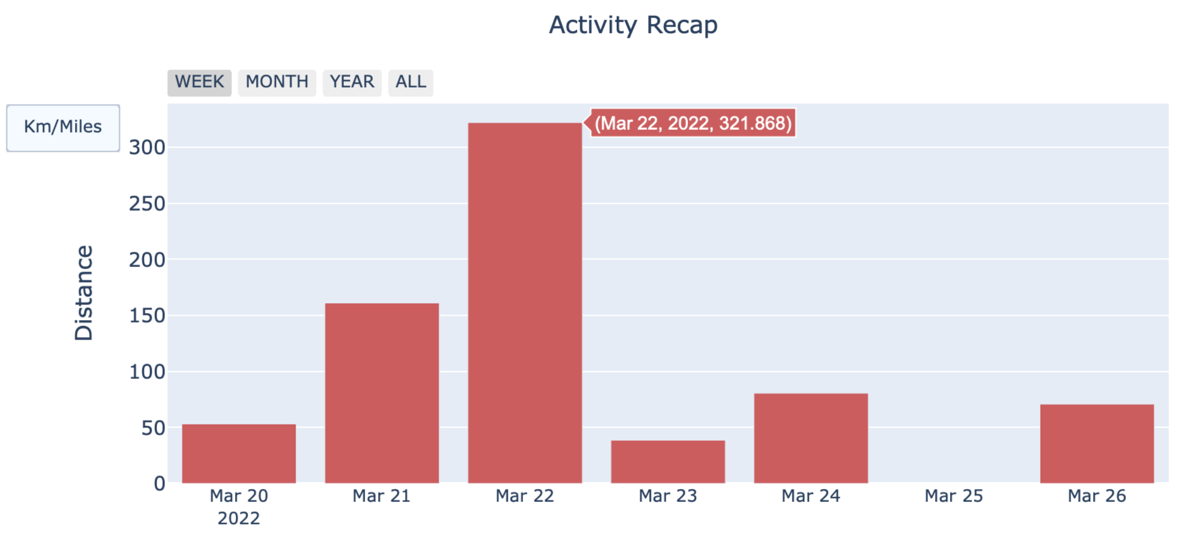


Figure 9: Cycling done throughout the week in km

Display Option 2: Weekly, monthly, yearly or all

A range slider allows App users to display a weekly, monthly, yearly or all-time recap of cycling done. Button 'WEEK' displays a recap from the 7 most recently logged cycling workouts (Dictionary with `label="WEEK"` activates B5). Button 'MONTH' displays a recap from the most recent month of recorded cycling (Dictionary with `label="MONTH"` activates B6). Button 'YEAR' shows the history of the cycling workouts completed throughout the current year (Dictionary with `label="YEAR"` activates B7). Lastly, button 'ALL' shows all cycling workouts done since the first use of the App (`label="ALL"` activates B8).

```
# Add range slider
fig.update_layout(
    yaxis=dict(
        title='Distance',
        titlefont_size=16,
        tickfont_size=14,
    ),
    xaxis=dict(
        rangeselector=dict(
            buttons=list([
                dict(count=7,
                     label="WEEK",
                     step="day",
                     stepmode="backward"),
                dict(count=1,
                     label="MONTH",
                     step="month",
                     stepmode="backward"),
                dict(count=1,
                     label="YEAR",
                     step="year",
                     stepmode="backward"),
                dict(label="ALL",
                     step="all")
            ])
        ),
        rangeslider=dict(
            visible=False
        ),
        type="date"
    )
)
```

Test case #1: Figure 8 and Figure 9 above show a weekly recap of the cycling done on the App by the user.
Test case #2: Figure 10 below displays the cycling distance the user has covered throughout March 2022.

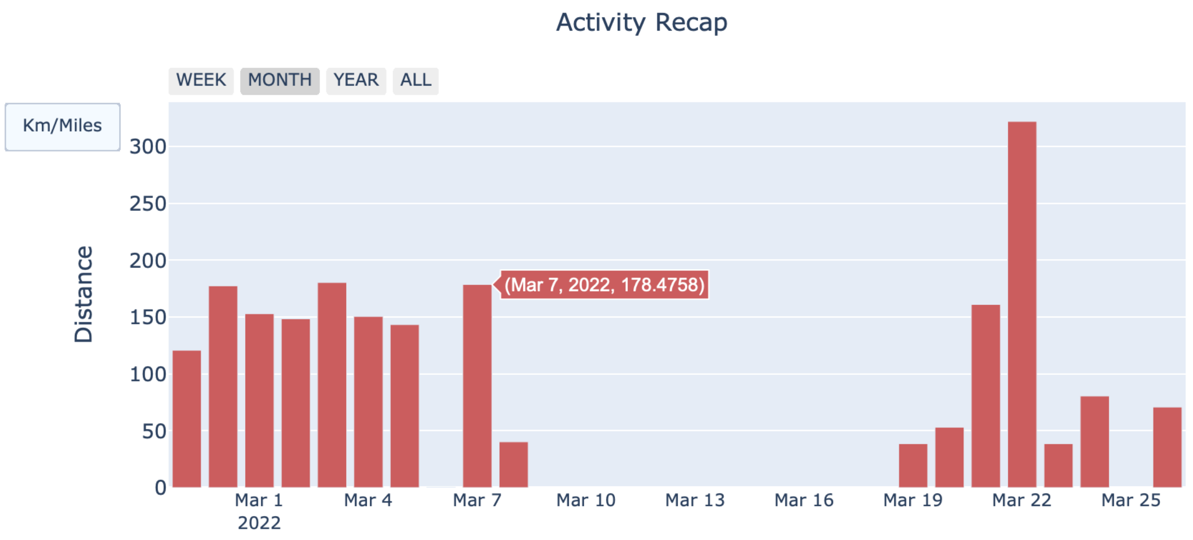


Figure 10: Cycling done throughout the week in km

The two other cases 'YEAR' and 'ALL' follow the same logic.

2. Use Case 2 - Calories Burnt

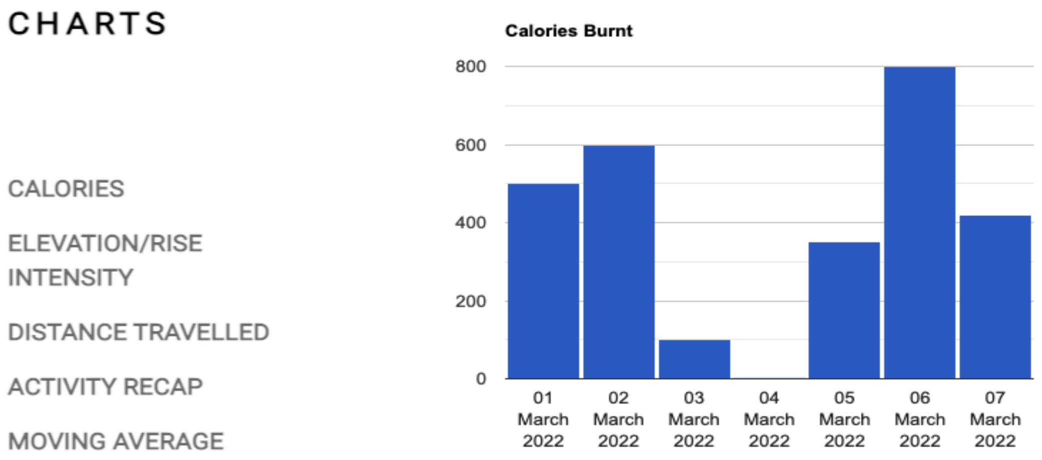
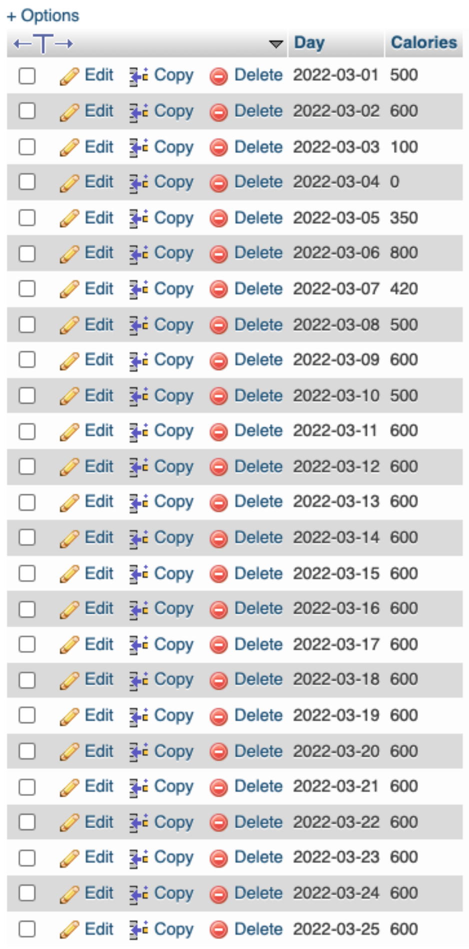


Figure 11: User Interface with generated chart for calories burnt from 03/01/2022-03/07/2022

Implementation:

- For the front-end, the use of HTML, CSS and JavaScript was implemented. To be able to generate charts for this use case in a way that is easily customizable, I made use of Google Charts which is a pure JavaScript library and provided the framework to display the data retrieved from the back-end. To retrieve the statistics displayed, I used PHP language to connect, access and fetch data from a MySQL database.
- This use case is not fully functional on the client-side yet since while it is fetching mock statistics from the SQL database, as of now it is merely displaying all the data collected without any working time frame filter. Temporarily, I have only implemented the logic for the statistics to be categorized by a weekly basis and plan on fully implementing a categorization for the statistics to be displayed not only on a weekly basis, but a monthly basis as well or give the user the option to select their own start-end date (up to 31 day interval) to view how many calories they burned for those days.
- To summarize, as of now, the chart for this use case is supposed to fetch the calories burnt every day as calculated from the statistics team and display it as a bar chart either for the first week of march, second, third or fourth.

Below is a sample of the mock data consisting of the days of March and the corresponding calories burned:



+ Options

	Day	Calories
Edit Copy Delete	2022-03-01	500
Edit Copy Delete	2022-03-02	600
Edit Copy Delete	2022-03-03	100
Edit Copy Delete	2022-03-04	0
Edit Copy Delete	2022-03-05	350
Edit Copy Delete	2022-03-06	800
Edit Copy Delete	2022-03-07	420
Edit Copy Delete	2022-03-08	500
Edit Copy Delete	2022-03-09	600
Edit Copy Delete	2022-03-10	500
Edit Copy Delete	2022-03-11	600
Edit Copy Delete	2022-03-12	600
Edit Copy Delete	2022-03-13	600
Edit Copy Delete	2022-03-14	600
Edit Copy Delete	2022-03-15	600
Edit Copy Delete	2022-03-16	600
Edit Copy Delete	2022-03-17	600
Edit Copy Delete	2022-03-18	600
Edit Copy Delete	2022-03-19	600
Edit Copy Delete	2022-03-20	600
Edit Copy Delete	2022-03-21	600
Edit Copy Delete	2022-03-22	600
Edit Copy Delete	2022-03-23	600
Edit Copy Delete	2022-03-24	600
Edit Copy Delete	2022-03-25	600

Figure 12: temporary mock data from MySQL database

White Box Testing

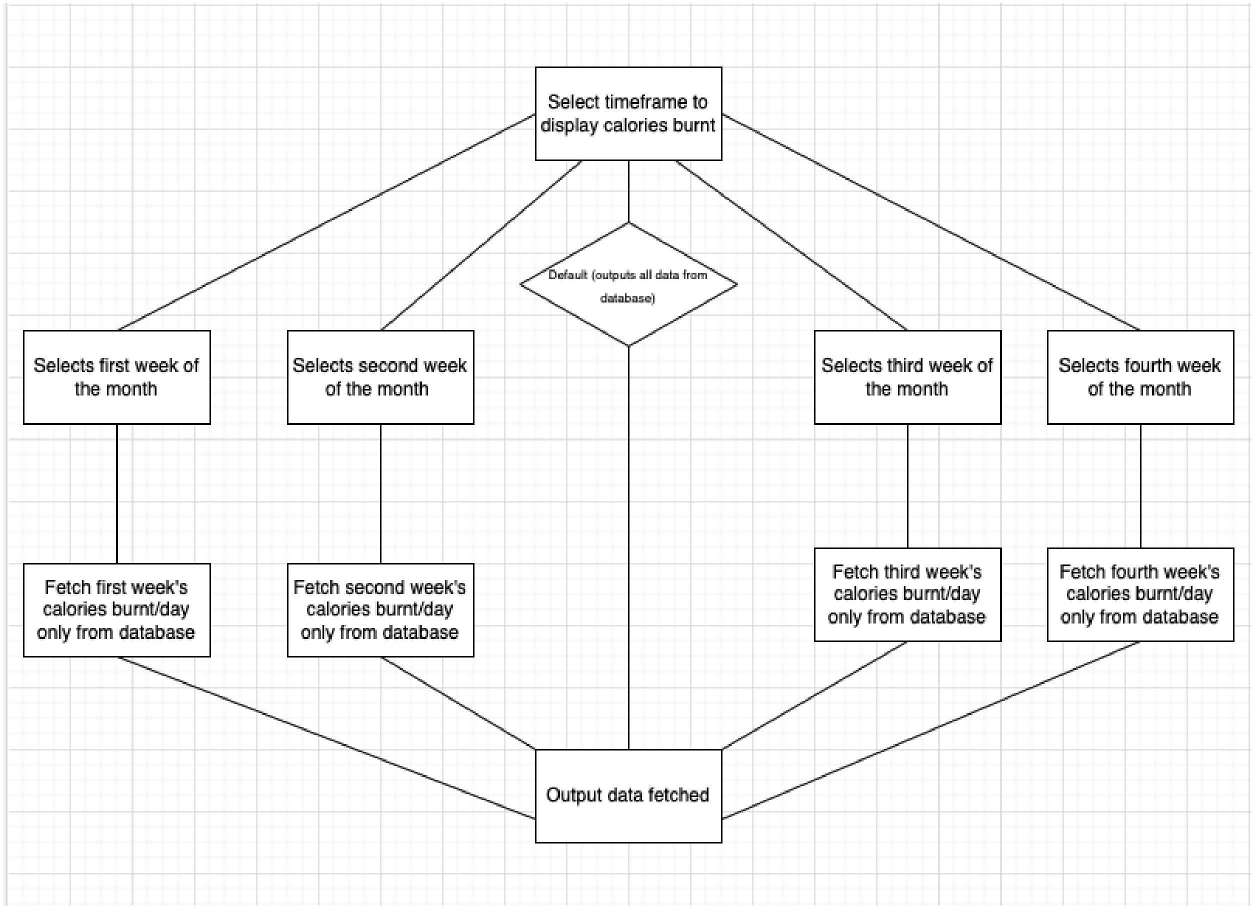
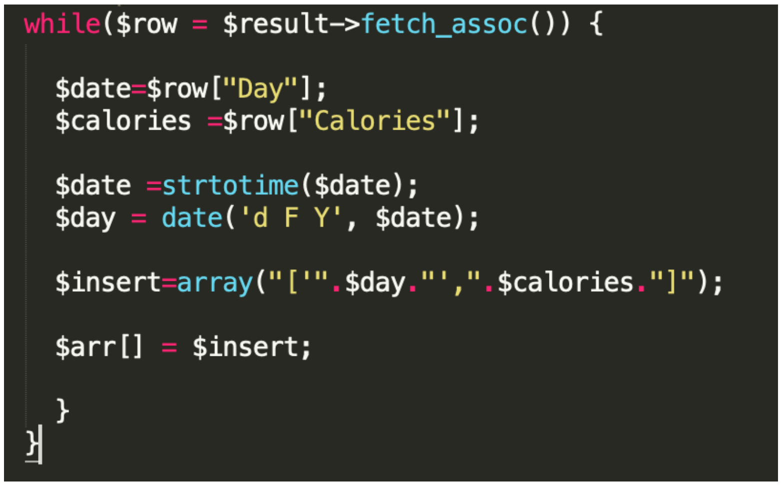

Select timeframe to display calories burnt
Default (outputs all data from database)
Selects first week of the month
Selects second week of the month
Selects third week of the month
Selects fourth week of the month
Fetch first week's calories burnt/day only from database
Fetch second week's calories burnt/day only from database
Fetch third week's calories burnt/day only from database
Fetch fourth week's calories burnt/day only from database
Output data fetched

Figure 13: control flow for generation of calories burnt chart

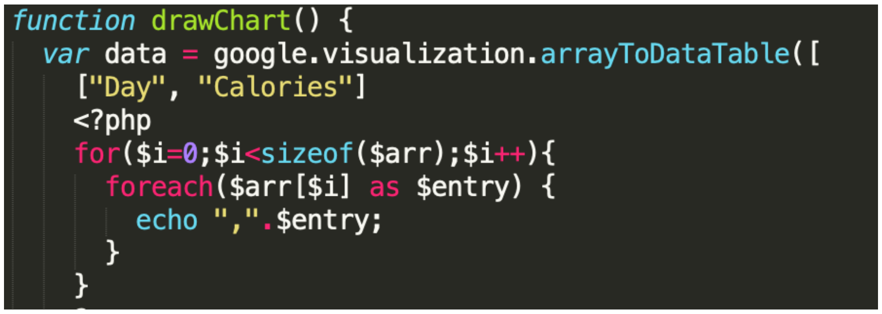
Default test case:

//All rows of data, regardless of date, is fetched from the database of calories burned
// It is then inserted into an array which will later be used to generate the chart on the web interface



```
while($row = $result->fetch_assoc()) {

    $date=$row["Day"];
    $calories =$row["Calories"];

    $date =strtotime($date);
    $day = date('d F Y', $date);

    $insert=array("['".$day."',".$calories."]");

    $arr[] = $insert;

    }
}
```

//Data from array is now used to generate the chart



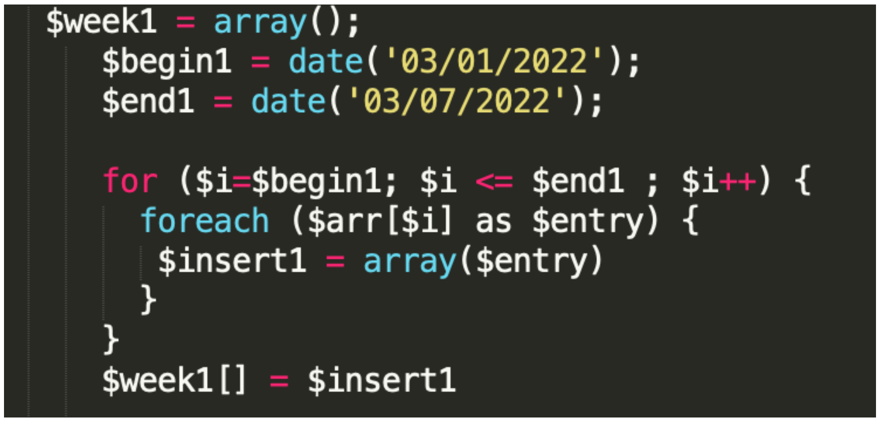
```
function drawChart() {
  var data = google.visualization.arrayToDataTable([
    ["Day", "Calories"]
    <?php
    for($i=0;$i<sizeof($arr);$i++){
      foreach($arr[$i] as $entry) {
        echo ",".$entry;
      }
    }
```

No user input given = coverage 2/10 statements = 20%
= coverage ⅕ branches = 20%

User selects a week test case:

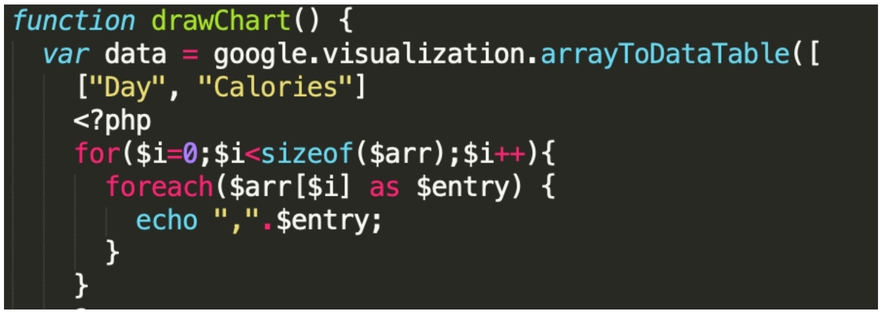
//In the case of user selecting first week of march, only rows from those dates is fetched

//It is then inserted into an array which will later be used to generate the chart on the web interface



```
  $week1 = array();
    $begin1 = date('03/01/2022');
    $end1 = date('03/07/2022');

    for ($i=$begin1; $i <= $end1 ; $i++) {
      foreach ($arr[$i] as $entry) {
        $insert1 = array($entry)
      }
    }
    $week1[] = $insert1
```

//Data from array is now used to generate the chart



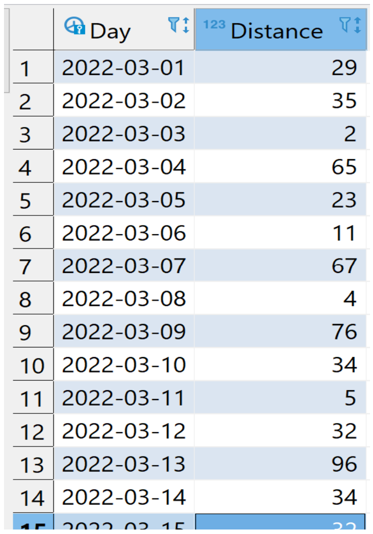
```
function drawChart() {
  var data = google.visualization.arrayToDataTable([
    ["Day", "Calories"]
    <?php
    for($i=0;$i<sizeof($arr);$i++){
      foreach($arr[$i] as $entry) {
        echo ",".$entry;
      }
    }
```

User input given (selects first week) = coverage 3/10 statements = 30%
= coverage ⅕ branches = 20%

The same coverage applies for week2, week3 and week4 as it does for week1, i.e. 30% statement coverage and 20% branch coverage.

3. Use Case 3 - Total Distance Travelled

Implementation: *Google Charts* was used as the chart generator of the total distance travelled. It is commonly used by embedding a simple Javascript on the web page. The freedom of customizing the chart is presented in the documentation of it. The integration of this chart allows us to pass an array of stats as a variable to then display it as a chart. The array of stats is taken from a mySQL database and calculated/set up afterwards. Charts are rendered through using HTML5/SVG technology.



	Day	Distance
1	2022-03-01	29
2	2022-03-02	35
3	2022-03-03	2
4	2022-03-04	65
5	2022-03-05	23
6	2022-03-06	11
7	2022-03-07	67
8	2022-03-08	4
9	2022-03-09	76
10	2022-03-10	34
11	2022-03-11	5
12	2022-03-12	32
13	2022-03-13	96
14	2022-03-14	34

Figure 14: Mock database table structure

The total distance travelled can be displayed weekly, monthly, or yearly depending on the user's preference based on all the stats collected since the beginning.

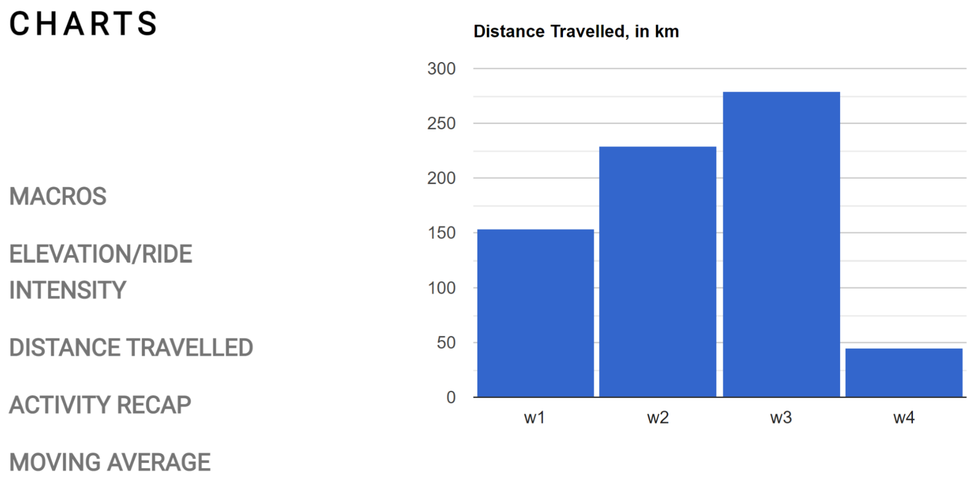


Figure 15: Chart display weekly total

As of now, users are only able to view a weekly total of their distance travelled. During the next iteration, I will work on the feature to let the user view the monthly, and yearly total.

White Box Testing

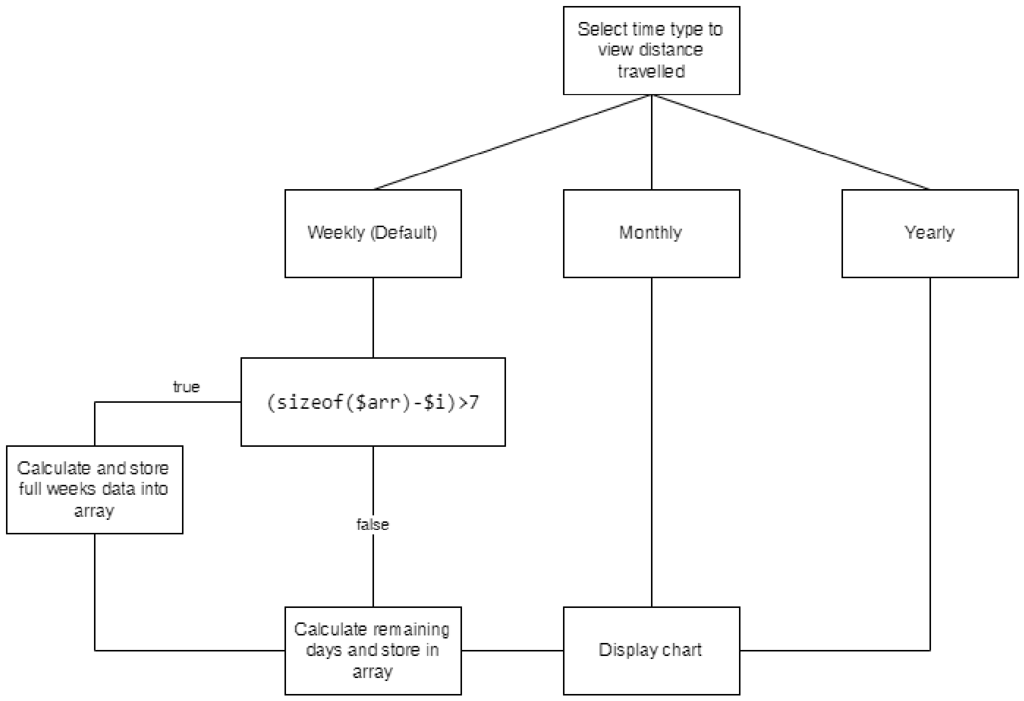
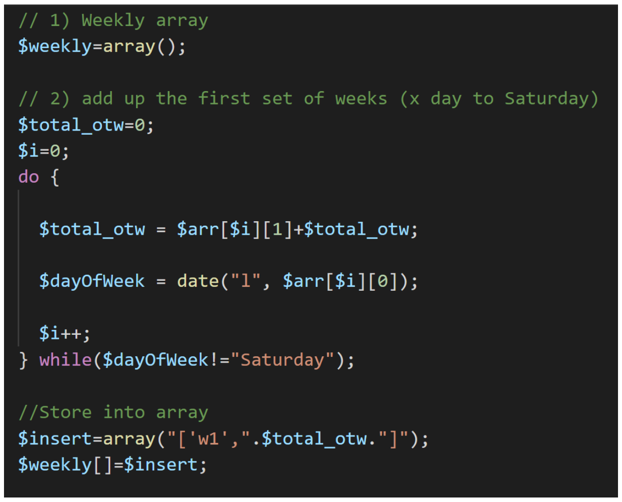

Select time type to view distance travelled
Weekly (Default)
Monthly
Yearly
true
(sizeof($arr)-$i)>7
Calculate and store full weeks data into array
false
Calculate remaining days and store in array
Display chart

Figure 16: control flow for generation of distance travelled chart

Test Case 1: User chooses to view Monthly total
Branch coverage = 20%

This gets run by default considering that the database stores data. If there's no data to show, the Chart generator generates a chart that displays 0.

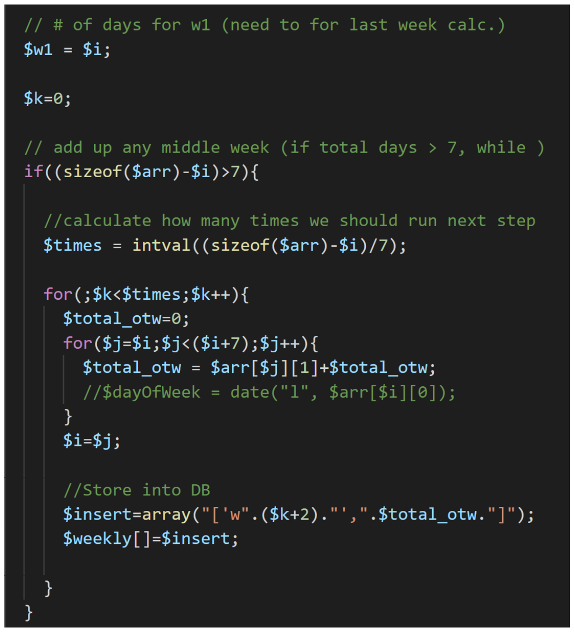


```
// 1) Weekly array
$weekly=array();

// 2) add up the first set of weeks (x day to Saturday)
$total_otw=0;
$i=0;
do {

  $total_otw = $arr[$i][1]+$total_otw;

  $dayOfWeek = date("l", $arr[$i][0]);

  $i++;
} while($dayOfWeek!="Saturday");

//Store into array
$insert=array("['w1',".$total_otw."]");
$weekly[]=$insert;
```

Figure 17: Calculate first week's stat

After the step above, the system checks if the user has more than 7 days remaining to add to the database.

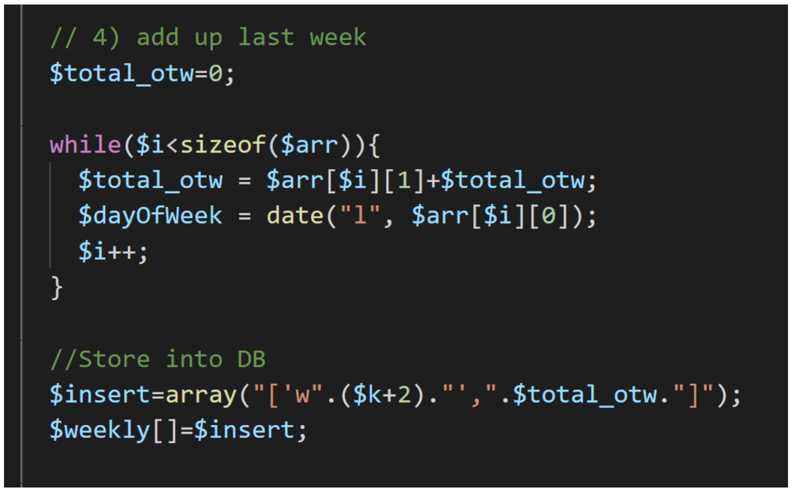
Test case #2: User has more than 7 days of stats registered after the first week
Branch coverage = 40%



```php
// # of days for w1 (need to for last week calc.)
$w1 = $i;

$k=0;

// add up any middle week (if total days > 7, while )
if((sizeof($arr)-$i)>7){

  //calculate how many times we should run next step
  $times = intval((sizeof($arr)-$i)/7);

  for(;$k<$times;$k++){
    $total_otw=0;
    for($j=$i;$j<($i+7);$j++){
      $total_otw = $arr[$j][1]+$total_otw;
      //$dayOfWeek = date("l", $arr[$i][0]);
    }
    $i=$j;

    //Store into DB
    $insert=array("['w".($k+2)."',".$total_otw."]");
    $weekly[]=$insert;

  }
}
```

Figure 18: Calculate complete weeks if there are more than 7 remaining days

This calculates the remaining days



```php
// 4) add up last week
$total_otw=0;

while($i<sizeof($arr)){
  $total_otw = $arr[$i][1]+$total_otw;
  $dayOfWeek = date("l", $arr[$i][0]);
  $i++;
}

//Store into DB
$insert=array("['w".($k+2)."',".$total_otw."]");
$weekly[]=$insert;
```

Figure 19: Calculate remaining days

Test case #3: User has less than 7 days of stats registered after the first week
Branch coverage = 40%

4. Use Case 4 - Elevation / Ride Intensity

4.1 Changelog:

4.1.1 Added the color scheme to more easily identify Descending, Ascending and Flat surfaces.

🟩 - going downhill
🟥 - going uphill
🟦 - no elevation change

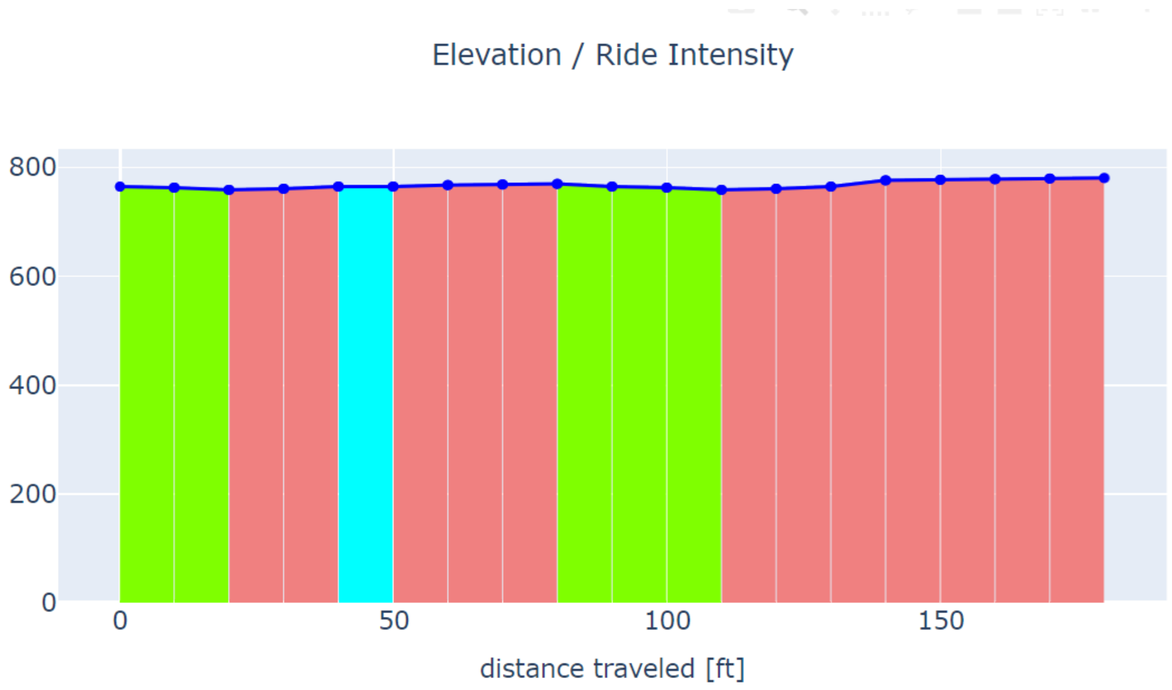


Figure 20: Elevation / Ride Intensity with color scheme

4.1.2 Replaced *time frame select* feature with *workout select*. Reason: relevance. (fully implemented on the server side, but missing user controls in front end)

4.1.3 Added the ft to meters conversion feature.
(fully implemented on the server side, but missing user controls in front end)

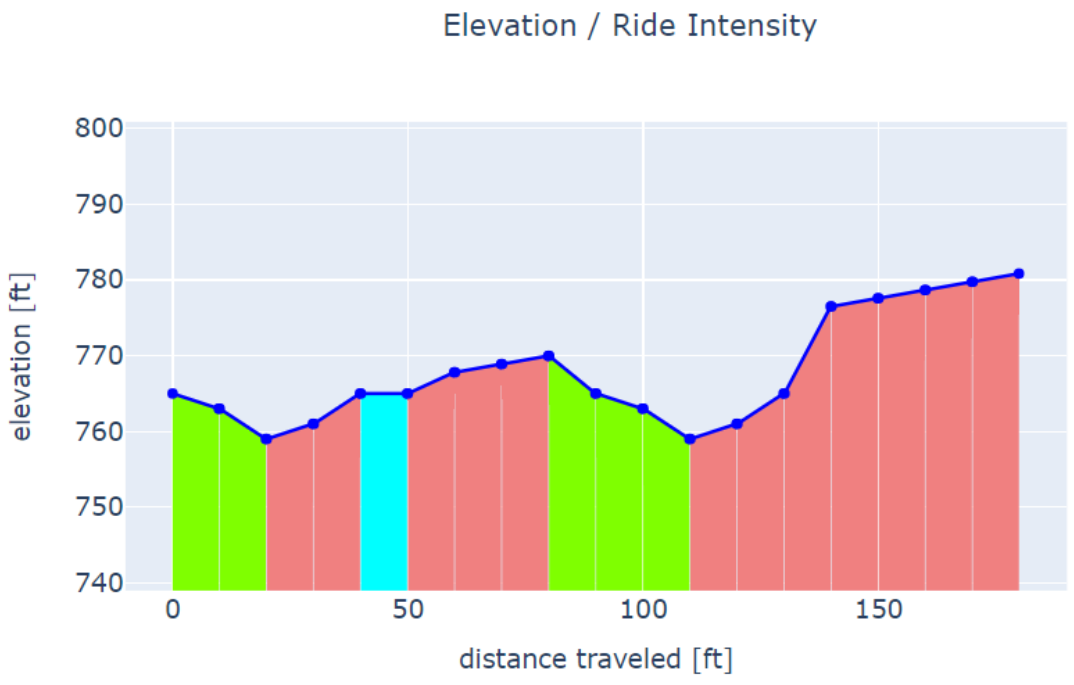


Figure 21: Elevation / Ride Intensity [ft]

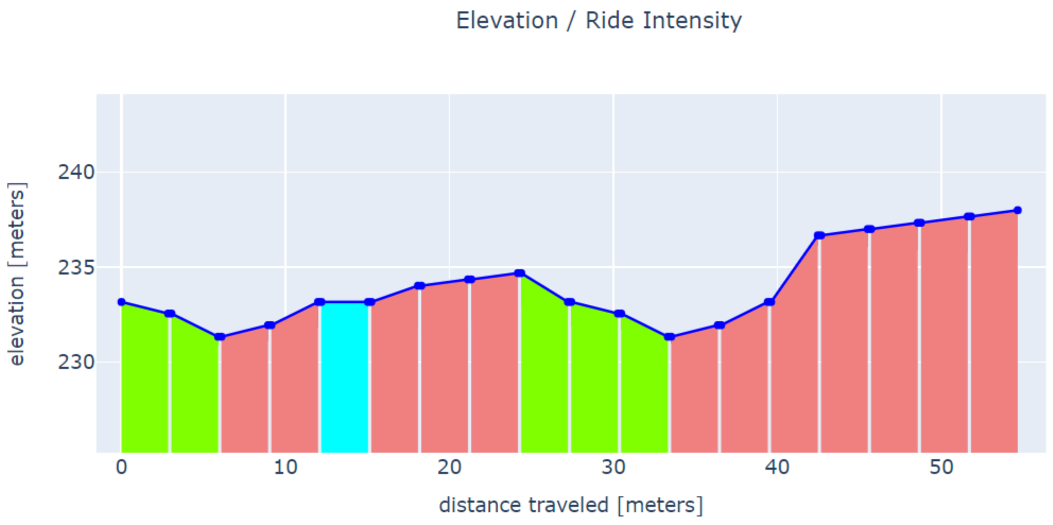


Figure 22: Elevation / Ride Intensity [meters]

4.1.4 A prototype snap functionality was implemented using plotly native controls. This feature is still in development and will be modified at release date.

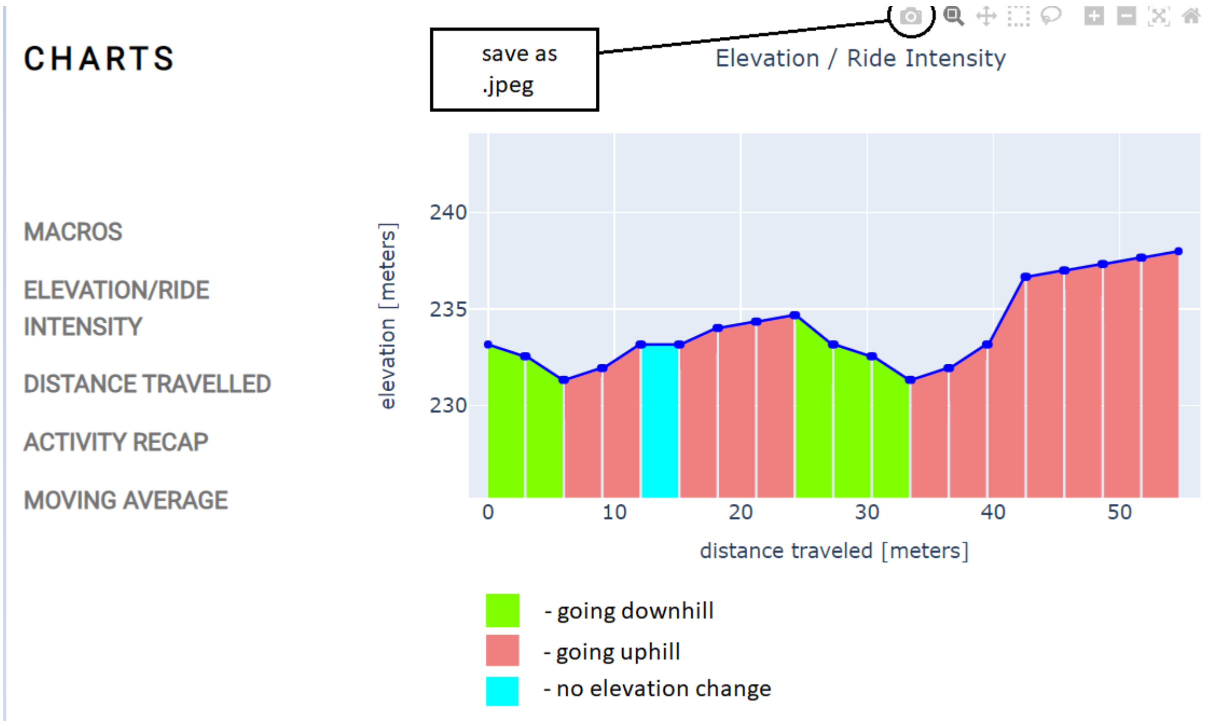
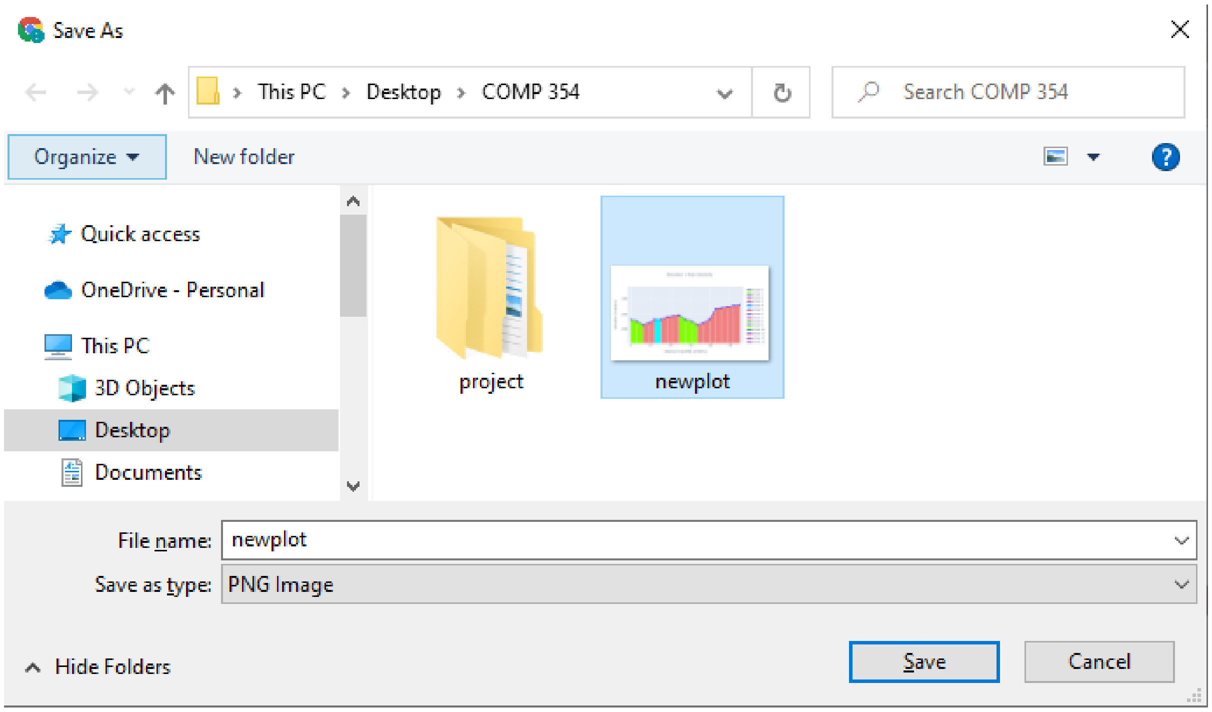


Figure 23: Elevation / Ride Intensity save as jpeg feature

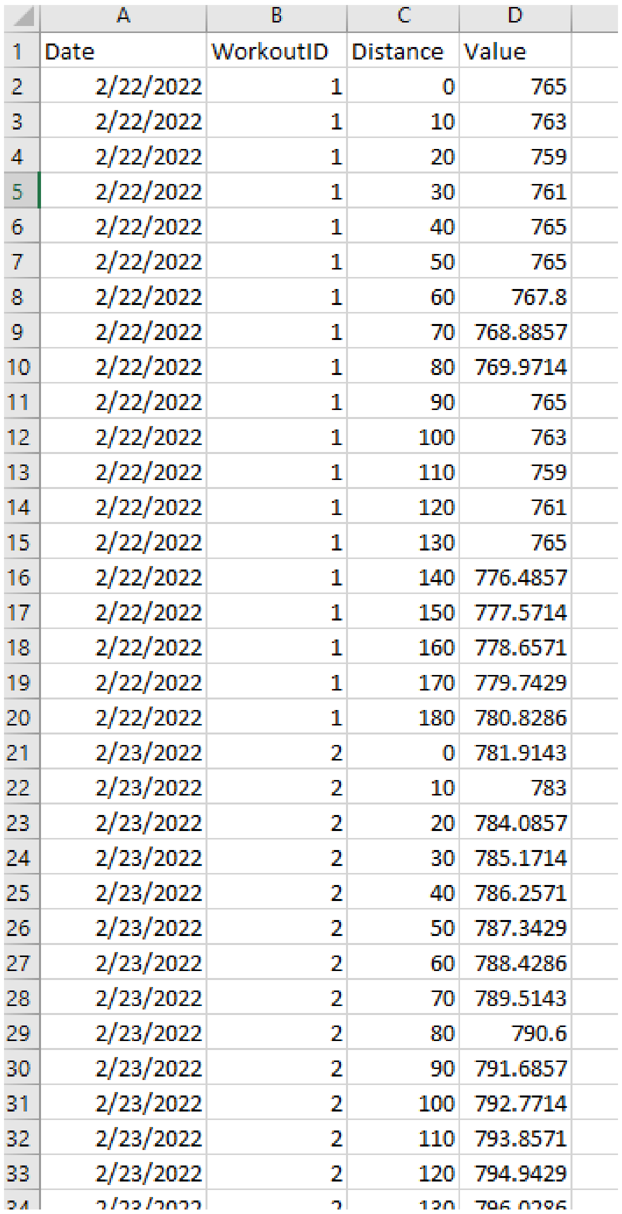


File name: newplot

Save as type: PNG Image

Figure 24: Elevation / Ride Intensity save as jpeg feature

4.2 Mock-up database structure:

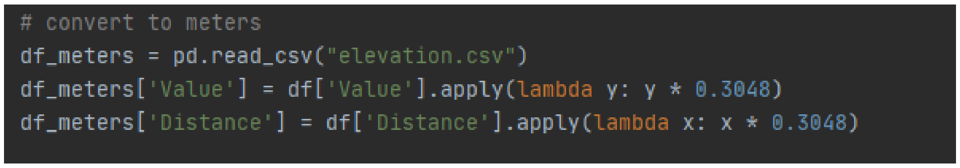


	A	B	C	D
1	Date	WorkoutID	Distance	Value
2	2/22/2022	1	0	765
3	2/22/2022	1	10	763
4	2/22/2022	1	20	759
5	2/22/2022	1	30	761
6	2/22/2022	1	40	765
7	2/22/2022	1	50	765
8	2/22/2022	1	60	767.8
9	2/22/2022	1	70	768.8857
10	2/22/2022	1	80	769.9714
11	2/22/2022	1	90	765
12	2/22/2022	1	100	763
13	2/22/2022	1	110	759
14	2/22/2022	1	120	761
15	2/22/2022	1	130	765
16	2/22/2022	1	140	776.4857
17	2/22/2022	1	150	777.5714
18	2/22/2022	1	160	778.6571
19	2/22/2022	1	170	779.7429
20	2/22/2022	1	180	780.8286
21	2/23/2022	2	0	781.9143
22	2/23/2022	2	10	783
23	2/23/2022	2	20	784.0857
24	2/23/2022	2	30	785.1714
25	2/23/2022	2	40	786.2571
26	2/23/2022	2	50	787.3429
27	2/23/2022	2	60	788.4286
28	2/23/2022	2	70	789.5143
29	2/23/2022	2	80	790.6
30	2/23/2022	2	90	791.6857
31	2/23/2022	2	100	792.7714
32	2/23/2022	2	110	793.8571
33	2/23/2022	2	120	794.9429
34	2/23/2022	2	130	796.0286

Figure 25: Mockup database for Elevation / Ride Intensity as .csv file

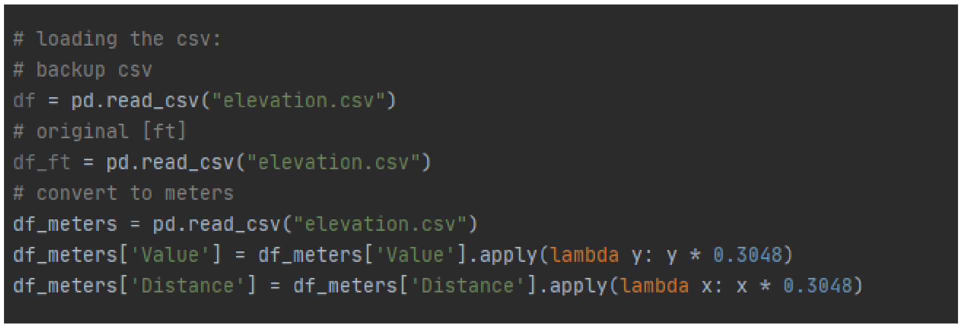
4.3 Whitebox testing:

- **Converting ft to meters:**



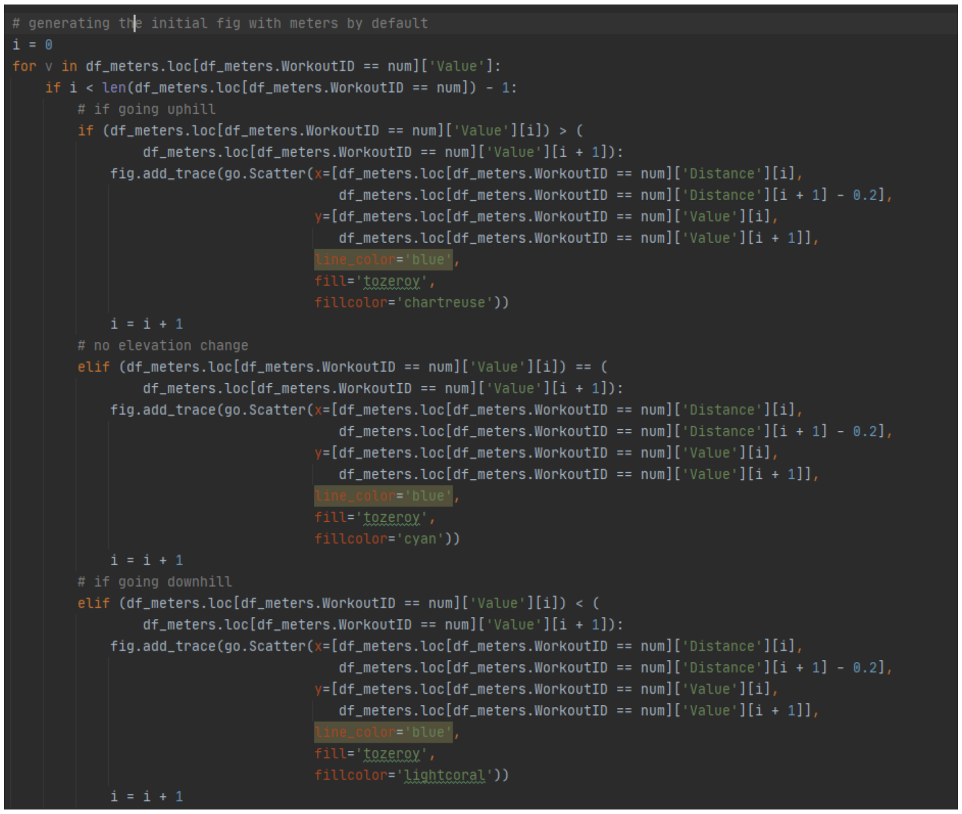
```python
# convert to meters
df_meters = pd.read_csv("elevation.csv")
df_meters['Value'] = df['Value'].apply(lambda y: y * 0.3048)
df_meters['Distance'] = df['Distance'].apply(lambda x: x * 0.3048)
```

- **Data pre-processing:**



```python
# loading the csv:
# backup csv
df = pd.read_csv("elevation.csv")
# original [ft]
df_ft = pd.read_csv("elevation.csv")
# convert to meters
df_meters = pd.read_csv("elevation.csv")
df_meters['Value'] = df_meters['Value'].apply(lambda y: y * 0.3048)
df_meters['Distance'] = df_meters['Distance'].apply(lambda x: x * 0.3048)
```

- **Dynamically calculating elevation data. A new trace is added to the figure along with the associating color scheme after the if-elif check determines the slope between two consecutive points.**



```python
# generating the initial fig with meters by default
i = 0
for v in df_meters.loc[df_meters.WorkoutID == num]['Value']:
    if i < len(df_meters.loc[df_meters.WorkoutID == num]) - 1:
        # if going uphill
        if (df_meters.loc[df_meters.WorkoutID == num]['Value'][i]) > (
                df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]):
            fig.add_trace(go.Scatter(x=[df_meters.loc[df_meters.WorkoutID == num]['Distance'][i],
                                        df_meters.loc[df_meters.WorkoutID == num]['Distance'][i + 1] - 0.2],
                                     y=[df_meters.loc[df_meters.WorkoutID == num]['Value'][i],
                                        df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]],
                                     line_color='blue',
                                     fill='tozeroy',
                                     fillcolor='chartreuse'))
            i = i + 1
        # no elevation change
        elif (df_meters.loc[df_meters.WorkoutID == num]['Value'][i]) == (
                df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]):
            fig.add_trace(go.Scatter(x=[df_meters.loc[df_meters.WorkoutID == num]['Distance'][i],
                                        df_meters.loc[df_meters.WorkoutID == num]['Distance'][i + 1] - 0.2],
                                     y=[df_meters.loc[df_meters.WorkoutID == num]['Value'][i],
                                        df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]],
                                     line_color='blue',
                                     fill='tozeroy',
                                     fillcolor='cyan'))
            i = i + 1
        # if going downhill
        elif (df_meters.loc[df_meters.WorkoutID == num]['Value'][i]) < (
                df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]):
            fig.add_trace(go.Scatter(x=[df_meters.loc[df_meters.WorkoutID == num]['Distance'][i],
                                        df_meters.loc[df_meters.WorkoutID == num]['Distance'][i + 1] - 0.2],
                                     y=[df_meters.loc[df_meters.WorkoutID == num]['Value'][i],
                                        df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]],
                                     line_color='blue',
                                     fill='tozeroy',
                                     fillcolor='lightcoral'))
            i = i + 1
```

- **Normalizing the graph to provide a better visualization and to enhance viewer experience by adding spacing on all sides of the graph and restricting the viewing window.**

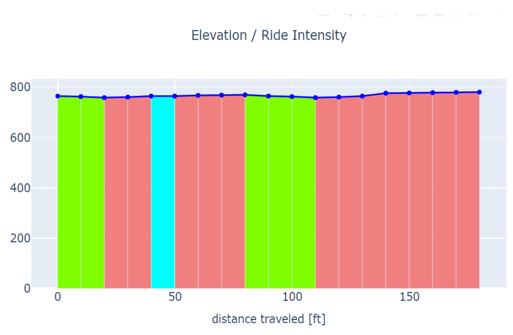
x-axis:

```
x_max = df.loc[df.WorkoutID == num]['Distance'].max() + 10
fig.update_layout(xaxis=dict(range=[-10, x_max]))
```

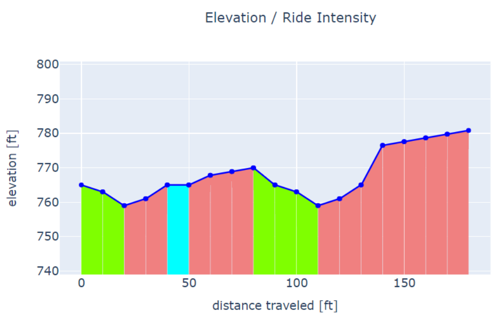
y-axis:

```
min = df.loc[df.WorkoutID == num]['Value'].min() - 20
max = df.loc[df.WorkoutID == num]['Value'].max() + 20
fig.update_layout(yaxis=dict(range=[min, max]))
```

Before:



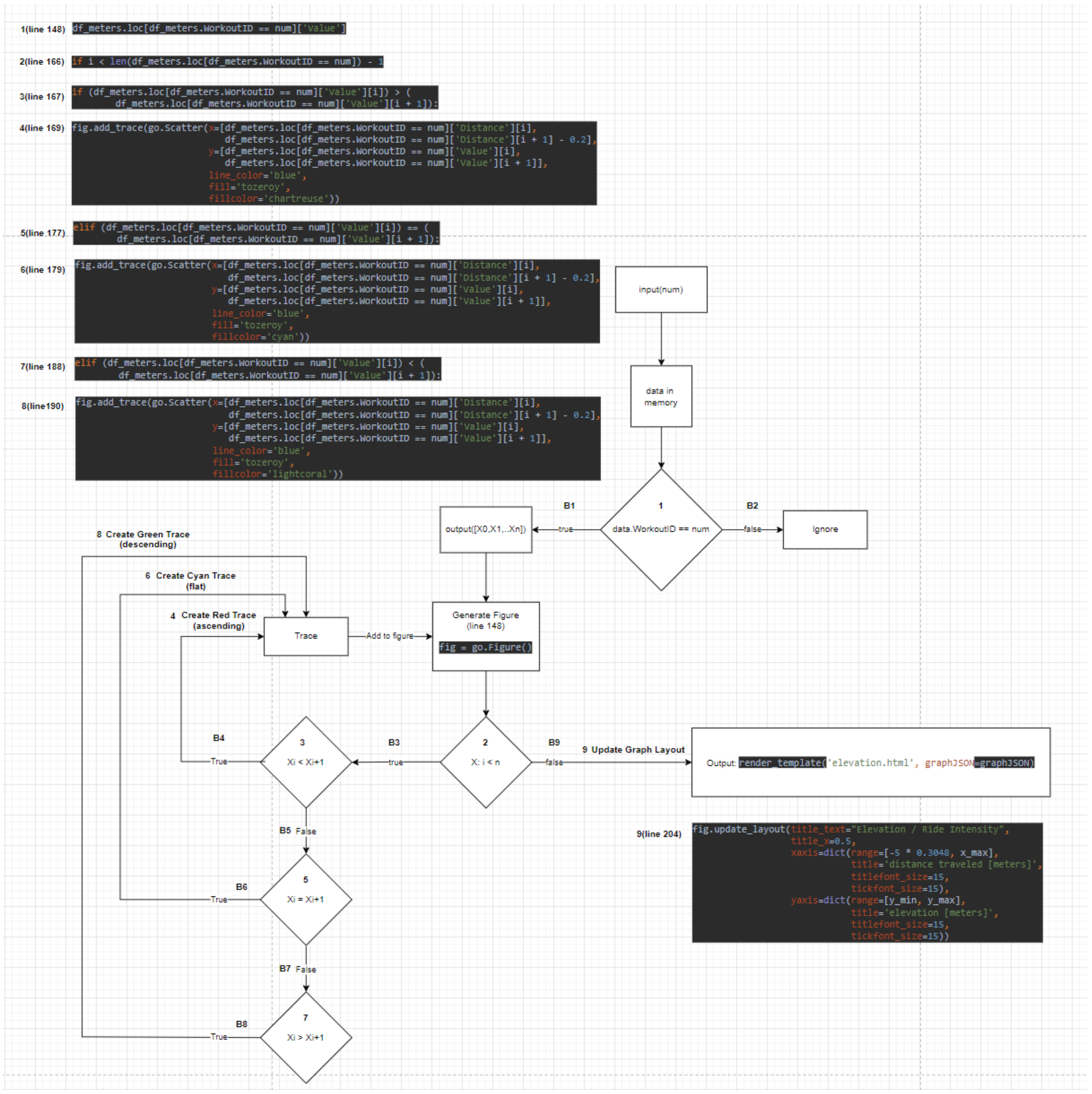
After:



4.4 Statement and Branch testing of Graph Generator for Elevation

This program excerpt describes the process of graph generation once the user provides the *num* input which represents the workoutID of their workout. By design, the input can only be selected from the existing workout IDs fetched from the database which eliminates the risk of invalid user input.

num will be used to fetch the corresponding lines from the database and to generate a trace (a line) between every two points. The color of the line "underglow" will be dictated based on the slope of the line formed by every two consecutive points(statements [3,5,7] and their corresponding branches) (see Use Case 4 for visual examples)



1(line 148)
```
df_meters.loc[df_meters.WorkoutID == num]['Value']
```

2(line 166)
```
if i < len(df_meters.loc[df_meters.WorkoutID == num]) - 1
```

3(line 167)
```
if (df_meters.loc[df_meters.WorkoutID == num]['Value'][i]) > (
        df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]):
```

4(line 169)
```
fig.add_trace(go.Scatter(x=[df_meters.loc[df_meters.WorkoutID == num]['Distance'][i],
                            df_meters.loc[df_meters.WorkoutID == num]['Distance'][i + 1] - 0.2],
                         y=[df_meters.loc[df_meters.WorkoutID == num]['Value'][i],
                            df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]],
                         line_color='blue',
                         fill='tozeroy',
                         fillcolor='chartreuse'))
```

5(line 177)
```
elif (df_meters.loc[df_meters.WorkoutID == num]['Value'][i]) == (
        df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]):
```

6(line 179)
```
fig.add_trace(go.Scatter(x=[df_meters.loc[df_meters.WorkoutID == num]['Distance'][i],
                            df_meters.loc[df_meters.WorkoutID == num]['Distance'][i + 1] - 0.2],
                         y=[df_meters.loc[df_meters.WorkoutID == num]['Value'][i],
                            df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]],
                         line_color='blue',
                         fill='tozeroy',
                         fillcolor='cyan'))
```

7(line 188)
```
elif (df_meters.loc[df_meters.WorkoutID == num]['Value'][i]) < (
        df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]):
```

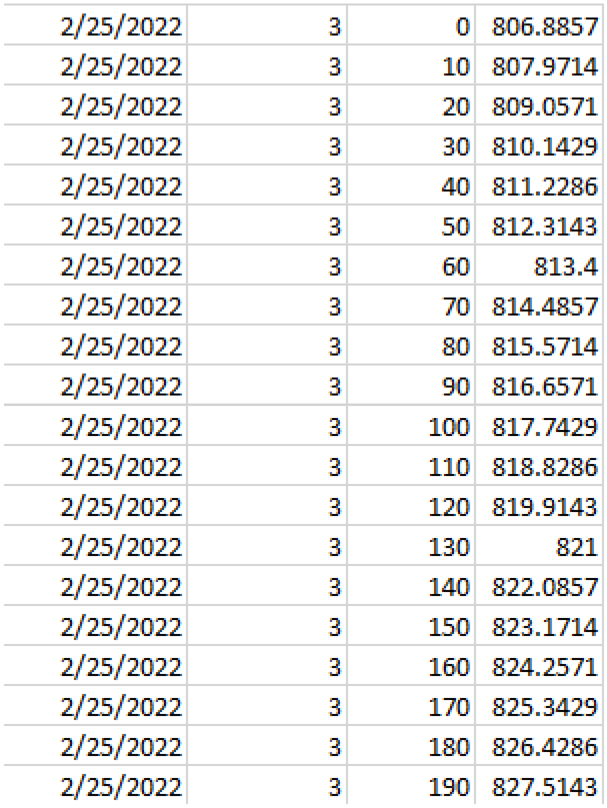
8(line190)
```
fig.add_trace(go.Scatter(x=[df_meters.loc[df_meters.WorkoutID == num]['Distance'][i],
                            df_meters.loc[df_meters.WorkoutID == num]['Distance'][i + 1] - 0.2],
                         y=[df_meters.loc[df_meters.WorkoutID == num]['Value'][i],
                            df_meters.loc[df_meters.WorkoutID == num]['Value'][i + 1]],
                         line_color='blue',
                         fill='tozeroy',
                         fillcolor='lightcoral'))
```

Output:
```
render_template('elevation.html', graphJSON=graphJSON)
```

9(line 204)
```
fig.update_layout(title_text="Elevation / Ride Intensity",
                  title_x=0.5,
                  xaxis=dict(range=[-5 * 0.3048, x_max],
                             title='distance traveled [meters]',
                             titlefont_size=15,
                             tickfont_size=15),
                  yaxis=dict(range=[y_min, y_max],
                             title='elevation [meters]',
                             titlefont_size=15,
                             tickfont_size=15))
```

Test case #1: num = 3

Based on the mockup database for Workout #3 below:



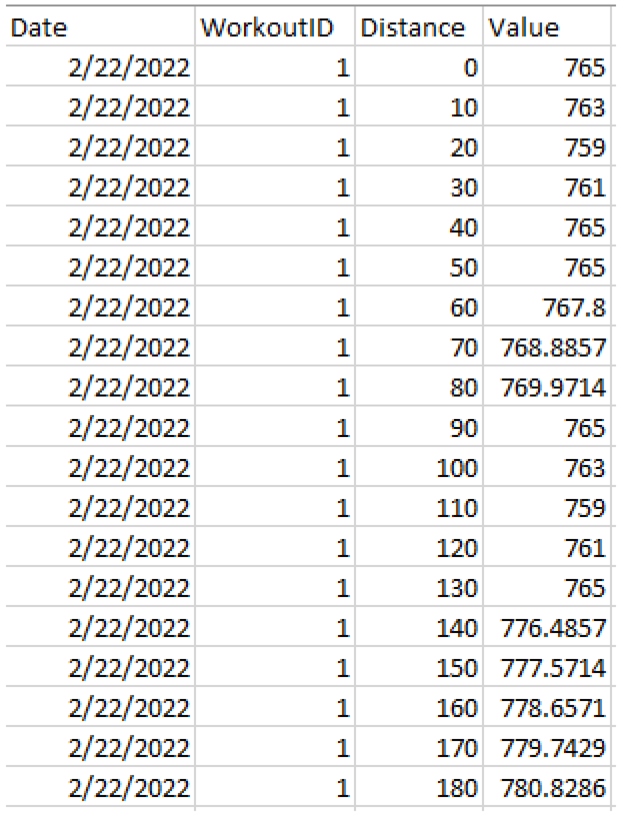
2/25/2022	3	0	806.8857
2/25/2022	3	10	807.9714
2/25/2022	3	20	809.0571
2/25/2022	3	30	810.1429
2/25/2022	3	40	811.2286
2/25/2022	3	50	812.3143
2/25/2022	3	60	813.4
2/25/2022	3	70	814.4857
2/25/2022	3	80	815.5714
2/25/2022	3	90	816.6571
2/25/2022	3	100	817.7429
2/25/2022	3	110	818.8286
2/25/2022	3	120	819.9143
2/25/2022	3	130	821
2/25/2022	3	140	822.0857
2/25/2022	3	150	823.1714
2/25/2022	3	160	824.2571
2/25/2022	3	170	825.3429
2/25/2022	3	180	826.4286
2/25/2022	3	190	827.5143

Workout #3 only contains increasing values in the last column, which means that the statement #3 will always evaluate to True, therefore statements 5, 6, 7 and 8 will never be triggered since branches B5, B6, B8 and B10 can never be visited.

Triggered statements: [1,2,3,4,9] = 5/9 = 55.55% coverage

Visited branches: [B1,B2,B3,B4,B7,B9] = 6/10 = 60% coverage

Test case #2: num = 1
Based on the mockup database for Workout #1 below:



Date	WorkoutID	Distance	Value
2/22/2022	1	0	765
2/22/2022	1	10	763
2/22/2022	1	20	759
2/22/2022	1	30	761
2/22/2022	1	40	765
2/22/2022	1	50	765
2/22/2022	1	60	767.8
2/22/2022	1	70	768.8857
2/22/2022	1	80	769.9714
2/22/2022	1	90	765
2/22/2022	1	100	763
2/22/2022	1	110	759
2/22/2022	1	120	761
2/22/2022	1	130	765
2/22/2022	1	140	776.4857
2/22/2022	1	150	777.5714
2/22/2022	1	160	778.6571
2/22/2022	1	170	779.7429
2/22/2022	1	180	780.8286

By analyzing all consecutive pairs we can see that it will cover all 3 scenarios (ascending, descending and flat line segments) therefore num = 1 will cover all the possible cases, trigger every statement and visit every branch.

Triggered statements: [1,2,3,4,5,6,7,8,9] = 9/9 = 100% coverage
Visited branches: [B1,B2,B3,B4,B5,B6,B7,B8,B9] = 9/10 = 90% coverage

<u>Note</u> that by design branch B10 can never be visited, but it's still present in the diagram because that's how the interpreter would treat the code, therefore the testing with
num == 1 provides 100% coverage.

5. Use Case 5 - Moving Average

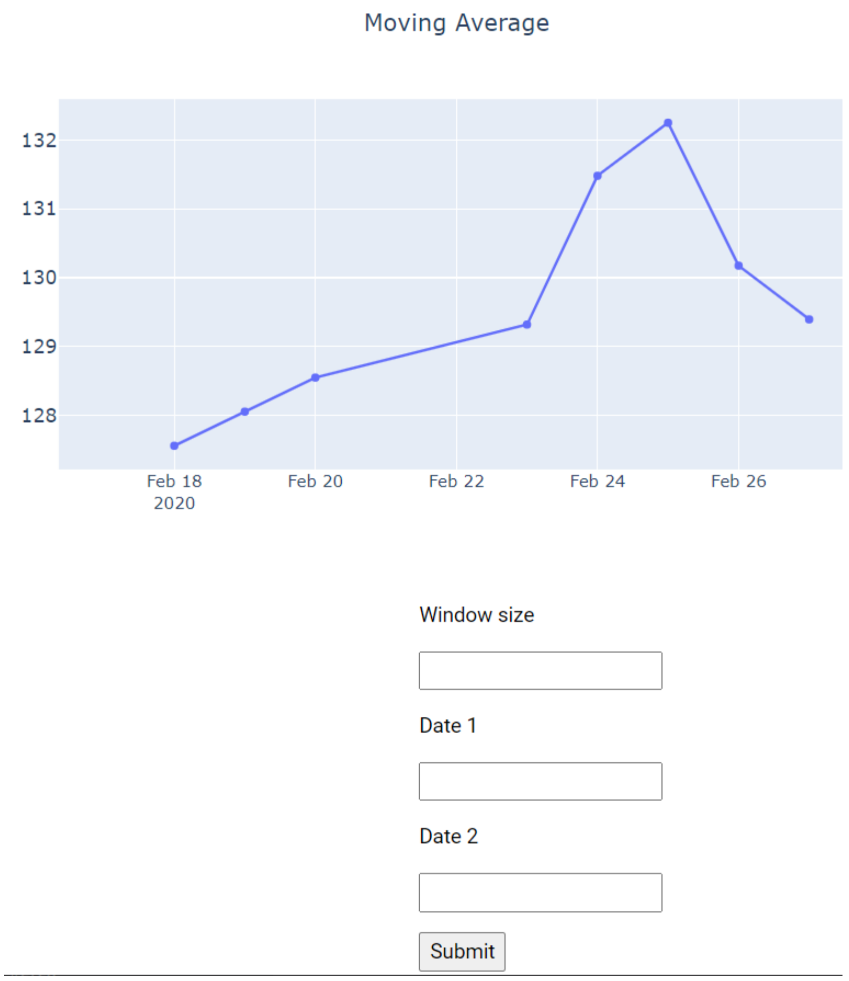
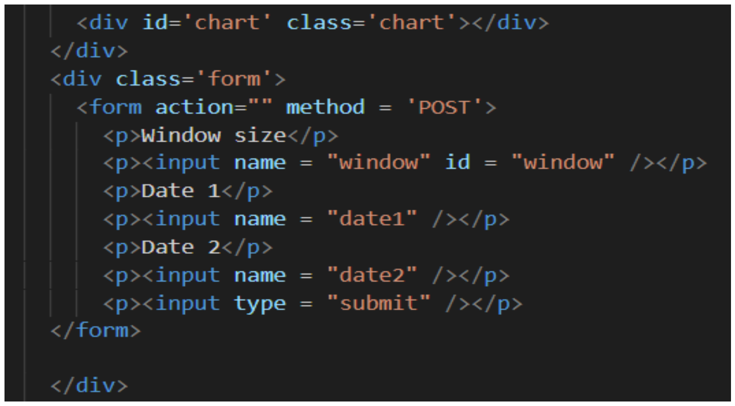


Figure 26: Implemented Moving Average chart with form to enter the period of calculation and window size of the moving average calculated. Changes the graph after each submission.

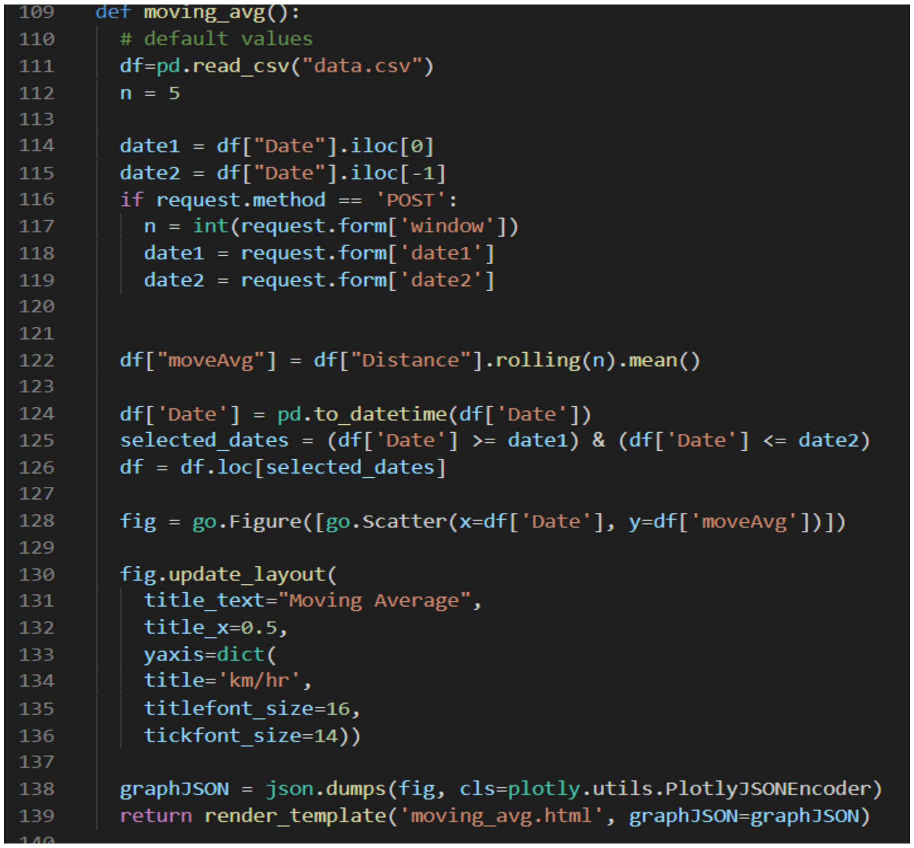
5.1 Implementation and Code

Pandas was used to read the csv files to get the distance data for moving average calculation. Plotly was used to generate the charts for the movingAverage.Html page. Flask was used as a Python web framework so that the charts could be hosted on the movingAverage.Html page. The same mock data used in case 1 was used for case 5.



```html
    <div id='chart' class='chart'></div>
  </div>
  <div class='form'>
    <form action="" method = 'POST'>
      <p>Window size</p>
      <p><input name = "window" id = "window" /></p>
      <p>Date 1</p>
      <p><input name = "date1" /></p>
      <p>Date 2</p>
      <p><input name = "date2" /></p>
      <p><input type = "submit" /></p>
  </form>

  </div>
```

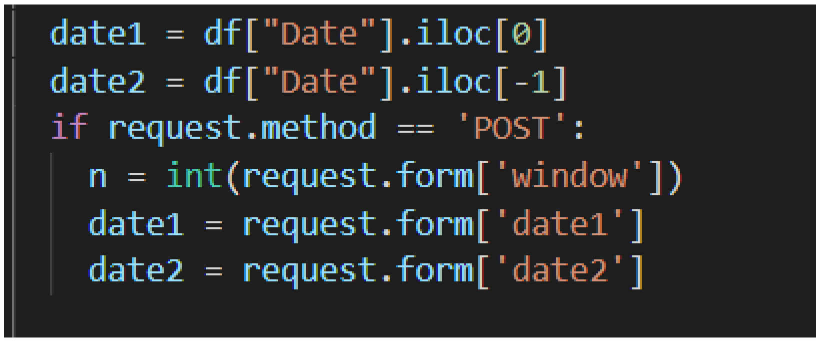
Figure 27: The form implemented on the moving_average.html



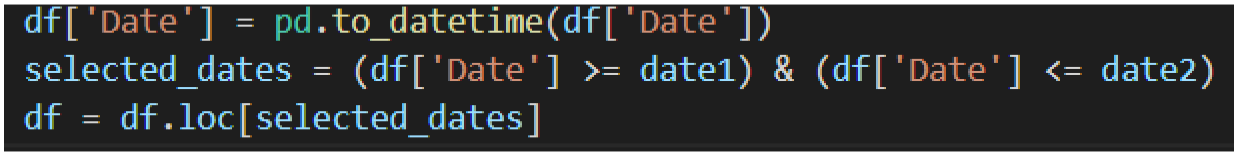
```python
def moving_avg():
    # default values
    df=pd.read_csv("data.csv")
    n = 5

    date1 = df["Date"].iloc[0]
    date2 = df["Date"].iloc[-1]
    if request.method == 'POST':
        n = int(request.form['window'])
        date1 = request.form['date1']
        date2 = request.form['date2']


    df["moveAvg"] = df["Distance"].rolling(n).mean()

    df['Date'] = pd.to_datetime(df['Date'])
    selected_dates = (df['Date'] >= date1) & (df['Date'] <= date2)
    df = df.loc[selected_dates]

    fig = go.Figure([go.Scatter(x=df['Date'], y=df['moveAvg'])])

    fig.update_layout(
        title_text="Moving Average",
        title_x=0.5,
        yaxis=dict(
        title='km/hr',
        titlefont_size=16,
        tickfont_size=14))

    graphJSON = json.dumps(fig, cls=plotly.utils.PlotlyJSONEncoder)
    return render_template('moving_avg.html', graphJSON=graphJSON)
```

Figure 28: The moving_avg() function used in white box testing.

5.2 Date Processing and Graphing



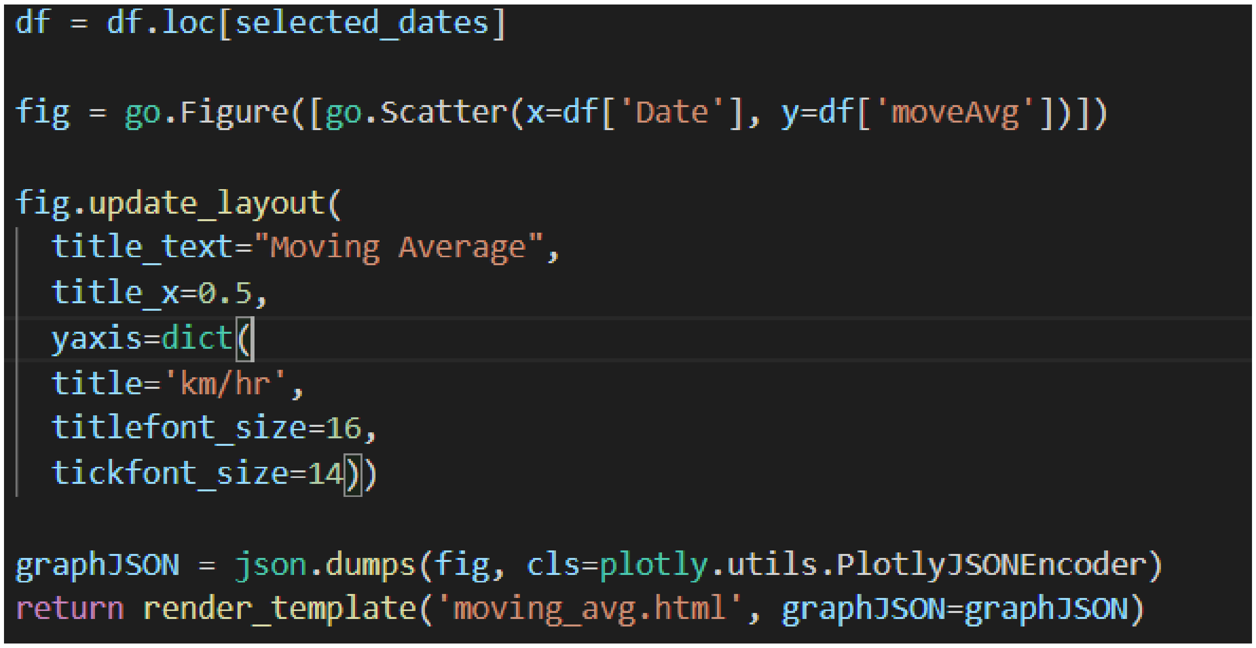
```
date1 = df["Date"].iloc[0]
date2 = df["Date"].iloc[-1]
if request.method == 'POST':
  n = int(request.form['window'])
  date1 = request.form['date1']
  date2 = request.form['date2']
```



```
df['Date'] = pd.to_datetime(df['Date'])
selected_dates = (df['Date'] >= date1) & (df['Date'] <= date2)
df = df.loc[selected_dates]
```

Figure 28.1: Date selection and processing

The form gives to the movingAverage() the starting and end dates. The selected_dates variable can store the subset of the data frame that ranges between and including those dates.



```
df = df.loc[selected_dates]

fig = go.Figure([go.Scatter(x=df['Date'], y=df['moveAvg'])])

fig.update_layout(
  title_text="Moving Average",
  title_x=0.5,
  yaxis=dict(
  title='km/hr',
  titlefont_size=16,
  tickfont_size=14))

graphJSON = json.dumps(fig, cls=plotly.utils.PlotlyJSONEncoder)
return render_template('moving_avg.html', graphJSON=graphJSON)
```

Figure 28.2: Graphing of Moving Average

After the moving average and dates have been determined, the function stores the graph generated in graphJSON and returns it to the moving_avg.html page to be displayed.

5.3 Flow Chart for Moving Average Chart Test Cases

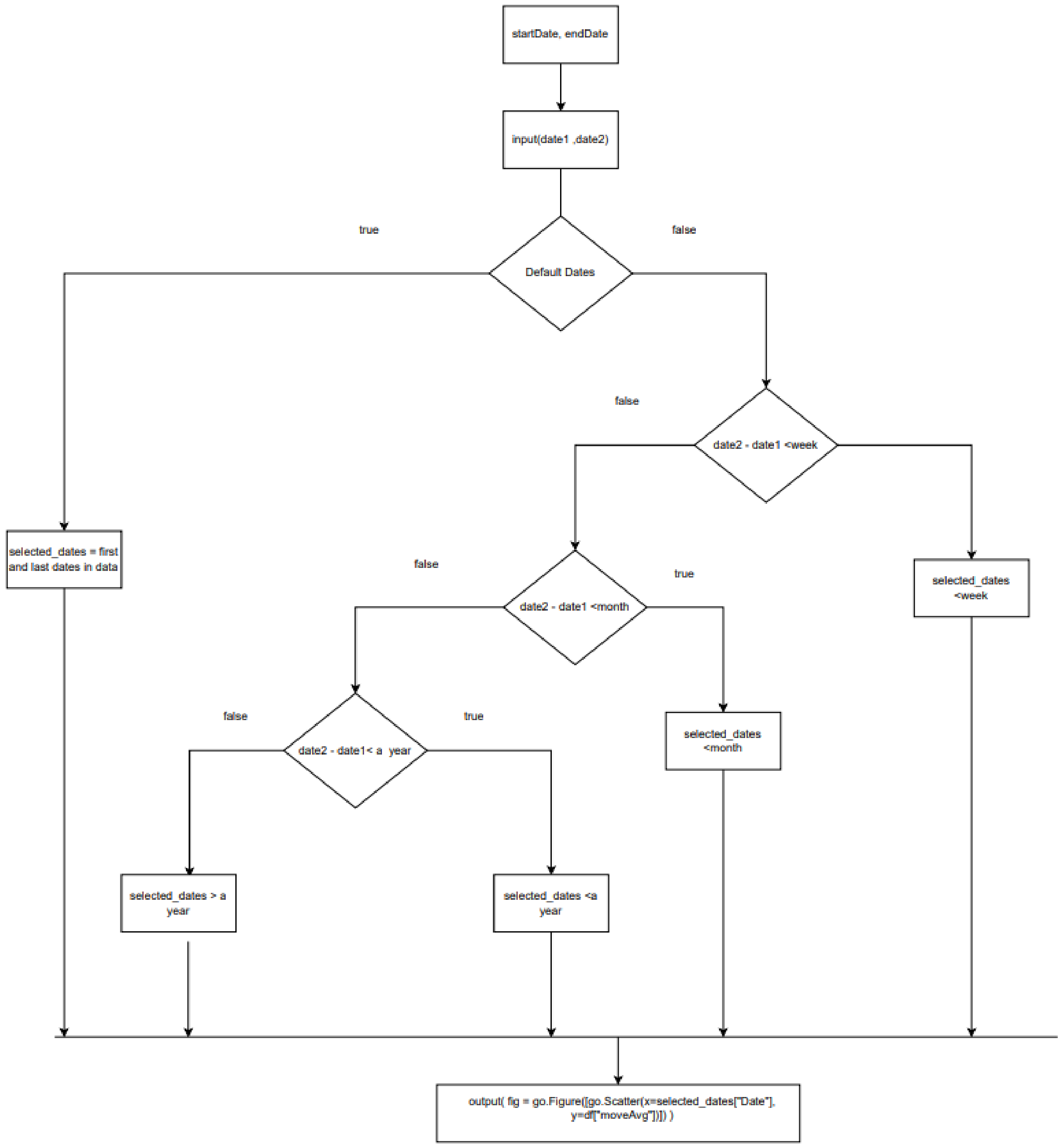
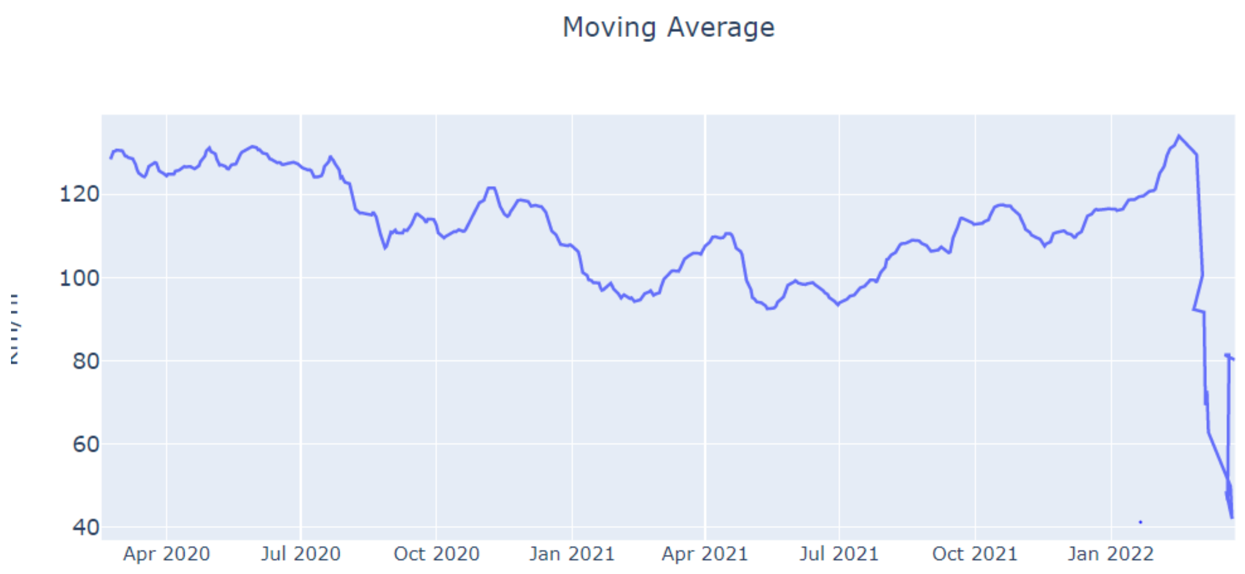
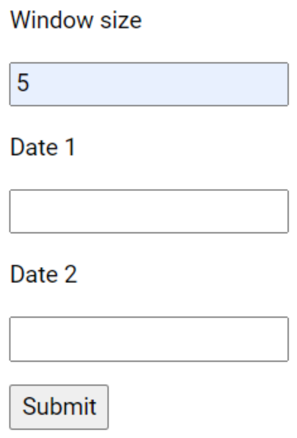


Figure 29: Flow chart of the moving_avg() function.

5.4 White Box Testing for Moving Average Chart




Window size

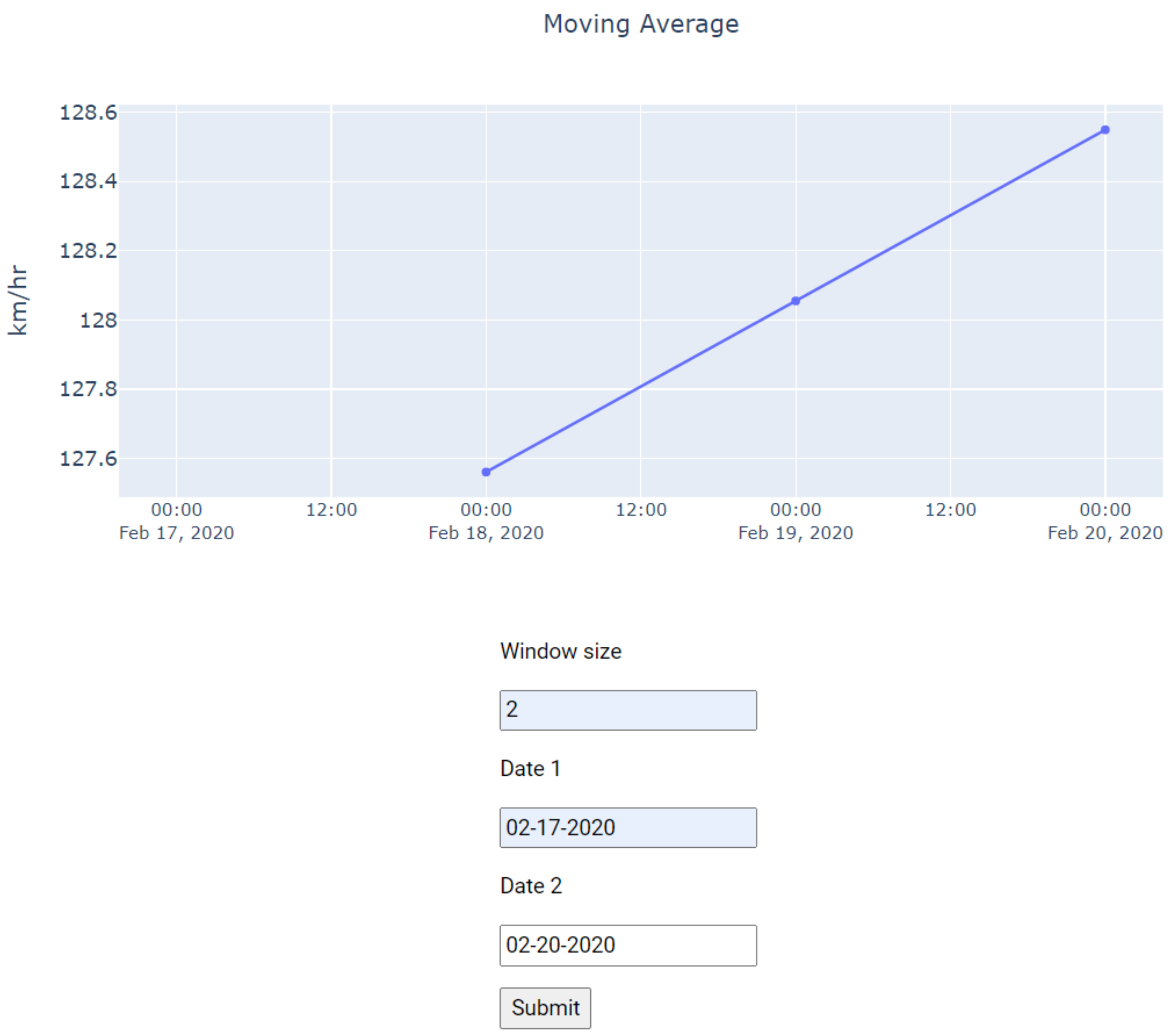
5

Date 1

Date 2

Test Case #1. No submission of dates or default dates are used. The default being the first and last dates in the data set.

Default dates = {109,110,111,112,113,114,115,120,121,122,123,124,125,126,127,128,129,130,131,132,133,134, 135,136,137,138,139} = 27/31 = 87% coverage



Window size

2

Date 1

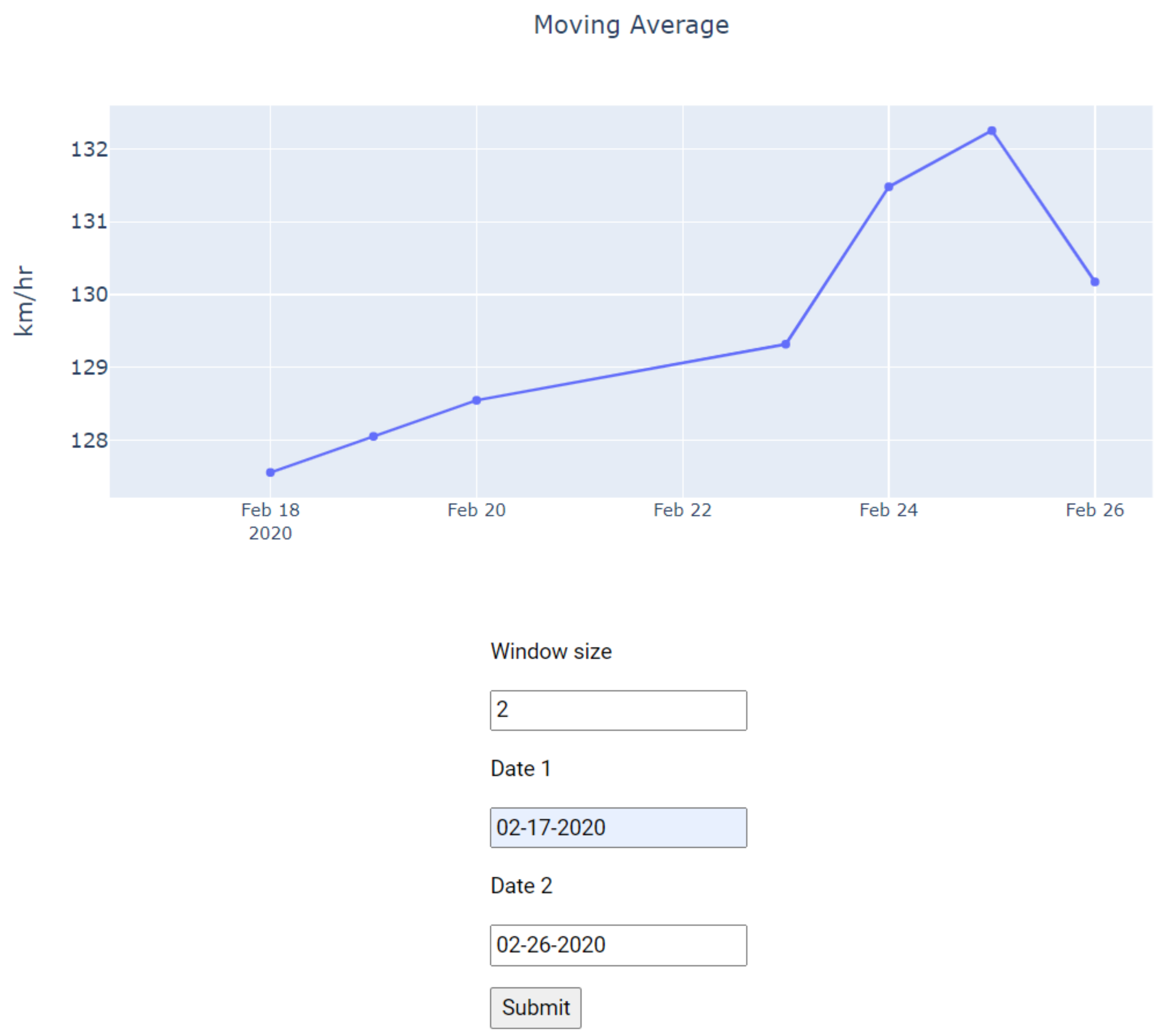
02-17-2020

Date 2

02-20-2020

Test Case #2. Less than a week is selected for moving average.

A couple of days = {109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,139} = 31/31 = 100% coverage



Window size

2

Date 1

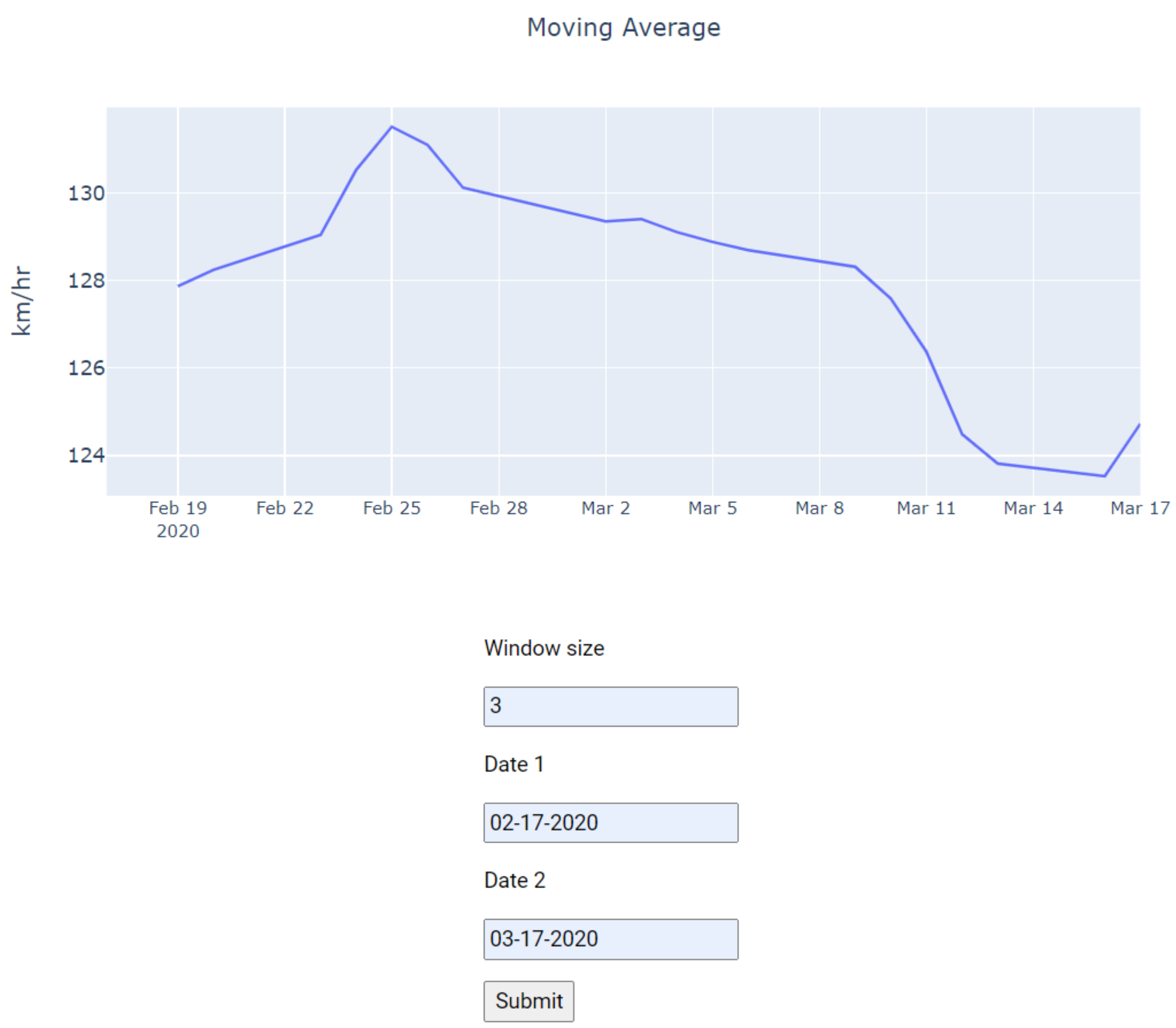
02-17-2020

Date 2

02-26-2020

Test Case #3. At least a week selected but not over a month.

A couple of days = {109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130, 131,132,133,134,135,136,137,138,139} = 31/31 = 100% coverage



Window size

3

Date 1

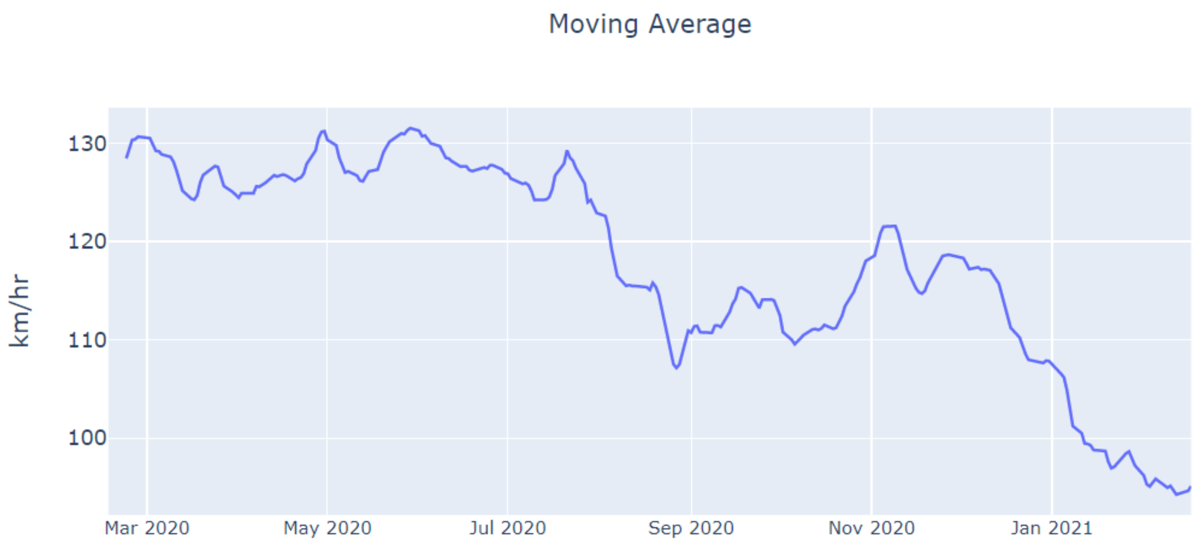
02-17-2020

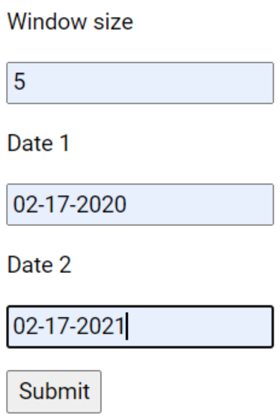
Date 2

03-17-2020

Test Case #4. At least a month selected but not over a year.

A couple of days = {109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130, 131,132,133,134,135,136,137,138,139} = 31/31 = 100% coverage




Window size

5

Date 1

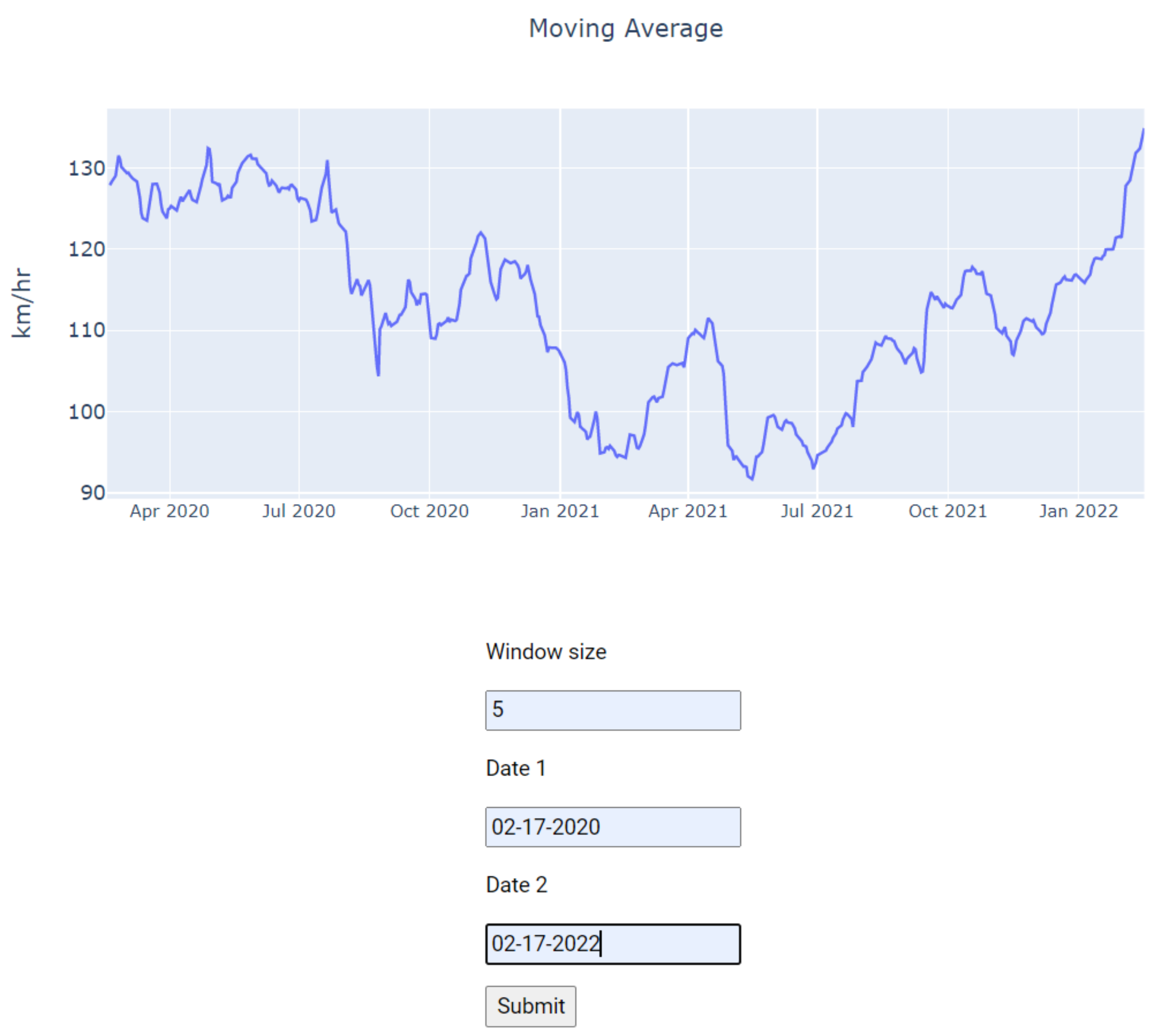
02-17-2020

Date 2

02-17-2021

Test Case #5. At least a year selected but not over many years.

A couple of days = {109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130, 131,132,133,134,135,136,137,138,139} = 31/31 = 100% coverage



Window size

5

Date 1

02-17-2020

Date 2

02-17-2022

Test Case #6. Moving average calculated over more than 1 year.

A couple of days = {109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130, 131,132,133,134,135,136,137,138,139} = 31/31 = 100% coverage

Citations

1. *Python - moving average for plotly*. (n.d.). Stack Overflow. Retrieved March 27, 2022, from https://stackoverflow.com/questions/59442383/moving-average-for-plotly
2. Jones, A. (2021, November 21). *Web Visualization with Plotly and Flask.* Medium. https://towardsdatascience.com/web-visualization-with-plotly-and-flask-3660abf9c946
3. *Plotly Graphics Libraries*. Plotly Python Graphing Library. (n.d.). Retrieved March 27, 2022, from https://plotly.com/python/