Documentation

BMI Dashboard, Team Dash, December 3, 2015

Table of Contents

I. Data Acquisition

A. File: Simulator.py

Simulator.py is the module that calls the mock sensor (Sensor.py) and the parser (Parser.py) in order to simulate and parse “live data”. It simulates live data by threading a process to run the main function until the sensor has completely scanned the entire file that it is currently operating in.

The file that Sensor.py is scanning is an eso file containing a whole day’s worth of data, located in the SENSOR/MBNMS_Monday.eso directory.

It also keeps track of global variables such as *numOfAcq*, *linenr*, and *tsc*, which are essential for the sensor and parser modules.

VARIABLES:

numOfAcq - The number of data acquisitions that the program has performed in the SENSOR/MBNMS_Monday.eso file.

linenr - The line number that the sensor module is currently grabbing from the SENSOR/MBNMS_Monday.eso file.
tsc - Stands for Timestamp Counter and represents the most recent timestep that the parser has parsed thus far.

tup - A tuple pair that will contain the current timestamp counter and line number that the sensor module updates.

```
global numOfAcq
global linenr
global tsc
tup = []
```

FUNCTIONS:

openFile(filepath) - takes in a filepath or directory path. It opens the file located in the path as a read-only file and returns it as a single chunk of string.

main() - runs the main simulator module.

ALGORITHM:
The main function in the simulator module does the following algorithm:

If the timestamp counter is less than or equal to 2 then decrement it by 1 to compensate for the offset that occurs during the identification of the first timestamp line. This is because the first timestamp line is not an actual time stamp, it is only the attributes of a timestamp, so we do not want to treat it as an actual timestamp. But it must still be included into the parsing.
The first timestamp line looks like the follow:

"2,6,Day of Simulation[],Month[],Day of Month[],DST Indicator[1=yes 0=no],Hour[],StartMinute[],EndMinute[],DayType"

```
tup = Sensor.mock(tsc, linenr)
if tsc >= 2: tsc -= 1
```

After the mock sensor finishes acquiring data from the SENSOR/MBNMS.eso file, it writes it into another file located in RAW_ESO/MBNMS.eso, which is where the parser looks for data to parse. The call Parser.parse(tsc) is what executes the parsing procedure.

```
Parser.parse(tsc)
```

After the parser parses the data, the simulator module makes a call to the database module using a bash script that runs an executable JAR file of the entire database program. In order to do this, we fork a sub-process that runs a “cd” command into the directory containing the bash script followed by a call to actually execute the bash script. In this case, the bash script is named “bmi.sh” and it is located in the directory “DATABASE”.

```
cmd = “cd DATABASE; ./bmi.sh”
subprocess.call(cmd, shell=True)
```

Increment the numOfAcq variable since new data has been acquired from the sensor's file, parsed into clean files, and inserted into the database. Note that tup[0] contains the updated line number (linenr) and tup[1] contains the updated timestamp counter (tsc). So we can update line number using the value inside tup[0] as well as the timestep counter from tup[1].

```
numOfAcq += 1
linenr = tup[0]
tsc = tup[1]
print("DATA HAS BEEN ACQUIRED " + str(numOfAcq) + " TIMES\n")
```

If the line number is -1, that means the sensor has finished reading and mocking all of the data within the SENSOR/MBNMS_Monday.eso file. In this case, we halt the program because it has finished simulating and parsing an entire day's worth of data. Otherwise, we continue to thread a new process to execute main once again, in a recursive-like manner. The line thread.Timer(0, main).start() is what does the actual threading. Timer() takes in a number x in seconds and a function. the start() call runs the function provided in timer after waiting x seconds. In this case, we make it wait zero seconds so that it runs instantaneously.

```
if not (linenr == -1): threading.Timer(0, main).start() # threads in seconds
else: return
```

This pretty much concludes what the Simulator.py module does.

B. File: Sensor.py

There is only one necessary function in Sensor.py, it is the mock() function. It mocks the simulation of "live" data. As previously explained, it will be called by the Simulator.py module to scan and write increments of data within the SENSOR/MBNMS.eso file. It takes in a timestamp counter (tsc) and a line number (linenr) so it knows where it last left off in the SENSOR/MBNMS_Monday.eso file.

VARIABLES:

sensor_file - The file that contains all of the data for a single day. The program will be looking at the file in increments that are separated by timestamp in

order to simulate "live" data. It is located in the directory "SENSOR\MBNMS.eso".

raw_file - The file that will contain all of the raw data that the parser will parse. It located in the directory "RAW_ESO/MBNMS.eso". Note this file is constantly changing with respect to the most recent timestamp. For example, if the sensor writes timestamp 8 and all of the data that appears in timestamp 8, then only those data(s) will be inside “RAW_ESO/MBNMS.eso”. This is happening every second, so the parser has to parse new/unique data every second. This can be scaled to every two seconds, or every minute, or even every hour.

```
sensor_file = openFile("SENSOR\MBNMS_Monday.eso").splitlines()
raw_file = open("RAW_ESO\MBNMS.eso", "w+")
```

FUNCTIONS:

openFile(filepath) - takes in a filepath or directory path. It opens the file located in the path as a read-only file and returns it as a single chunk of string.

mock(tsc, linenr) - takes in a timestamp counter and a line number. It mocks the sensors that simulate live data. It is the main function for the Sensor.py module.

ALGORITHM:

The main function in the sensor module is the mock() function. This is how it works:

Starts at the last line that it left off at and continue scanning until the entire sensor_file has been mocked. For every line in the sensor_file, the program checks if it's key starts with "2,". This is because the lines with keys starting with “2,” are all of the timestamp lines. They contain the timestamps for all of the data below them until the next timestamp line.

```
print("Starting with line %d" % linenr)
    for data in range(linenr, len(sensor_file)):
        d = sensor_file[data]
        if d.startswith("2,", 0, 2):
```

If it starts with “2,” then it must be a timestamp. But we have to check if it is the first timestamp because the first timestamp in any of the MBNMS files are just the attributes and do not actually contain any real timestamps. This first timestamp file looks like this:

"2,6,Day of Simulation[],Month[],Day of Month[],DST Indicator[1=yes 0=no],Hour[],StartMinute[], EndMinute[],DayType"

If it is in fact the first timestamp file, we increment the timestamp counter (tsc) and let it continue with the simulation. However, if it isn't then we increment the line number (linenr) along with the tsc to record what line we are on so we can continue off from it later.

```
                if tsc == 0:
                    tsc += 1
                else:
                    linenr += 1
                    tsc += 1
                    raw_file.write(d + "\n")
                    break
```

These lines are written to the raw_file as "raw" data for the parser to read.

```
            raw_file.write(d + "\n")
            linenr += 1
```

Check if line number has reached the end of the sensor file. If it did, then set the line number to -1 so that the Simulator.py module knows that the data simulation is over and it can end the program. Otherwise, we want to store the current initialize a tup = [0, 0] variable so we can store the line number and the timestamp counter in it. We do this in the sensor module and return it as a tuple value for the simulator program to receive it in it’s own tup variable. If you scroll up to the Simulator module in part A of Data Acquisition, you can see that there is a line that says:

tup = Sensor.mock(tsc, linenr)

This line runs the mock sensor as well as returns the latest line number and timestamp counter to the simulator to utilize.

```
            if linenr >= len(sensor_file):
                linenr = -1
                break
        print("Ending with line %d\n" % linenr)
        tup = [0, 0]
        tup[0] = linenr
        tup[1] = tsc
        return tup
```

This is what the Sensor.py module does in its attempt to mock live data.

B. File: Parser.py

The Parser.py module parses out the keys, attributes, and data values within the the data dictionary located in RAW_ESO/MBNMS.eso. It also converts the .eso files into .csv files.

VARIABLES:

data - Each line of the file located at file path.

dname - The entire directory path of the file, excluding the actual file name.

bname - The basename of the directory path, which is the file name.

newbasepath - The new basename that we will be putting the clean files into.

clean_file - The file name of the clean file.

ts_filepath -The file path for the timestamp file.

ts_file - The time stamp file. (Contains all of the timestamp values and only those values).

timestamps - Reopens the time stamp file so that it can be read as a reference to indicate the latest time stamp.

attributes - The attributes for each data value.

key - The key that for each line in the data dictionary

FUNCTIONS:

openFile(filepath) - takes in a filepath or directory path. It opens the file located in the path as a read-only file and returns it as a single chunk of string.

cleanPVFile(filepath) - takes in a filepath or directory path. It opens the file by calling openFile() to get a string of data. It then calls splitlines() to split the data into sub strings separated by newlines. It also distinguishes the key attributes from their respective data values and rewrites them into a directory CLEAN_PV as .csv files. This function is deprecated because we decided to handle the PV data in a different manner from the BMS data. This function is no longer being called or used by the program but was kept in case we needed to operate on both PV and BMS datas in the same manner.

parseESOKeys(filepath, tsc) - takes in a filepath and a timestamp counter. Parses the eso keys and values in the given file. This runs the main parsing algorithm.

createDataDict(filepath) - takes in a filepath. It opens the file located in the filepath and runs a loop that rewrites the attributes and values of each line in the file until it reaches a line that says “End of Data Dictionary”. This line is present in all of the .eso files that would have been given to the parser to parse. It indicates where the data dictionary ends and where the actual data starts in the file. Without this line, the parser function will not work correctly.

parse(tsc) - takes in a timestamp counter and runs a loop to scan an entire directory for .eso files. It attempts to find any .eso file that needs to be parsed and cleaned within RAW_ESO. This function makes calls to parseESOKeys() after locating “unclean” .eso files.

ALGORITHM:

This is the main parsing mechanism for the Parser.py module. It is the parseESOKeys() function.

It iterates through all of the attributes within the data dictionary and create files for them if they have not already been created (d1). It creates a filename for specific keys by appending the key to an underscore and further appending the basename of the filepath, which has had it's filetype truncated so that we can replace it with ".csv" (Note that it was originally a ".eso"). It then re-iterates through the entire file to append each of the current timestamp data values into their respective keys. This is done in the (d2) loop, which also writes a line containing the key attribute values appended to the timestamp values.

```
        for d1 in data:
          if d1 == "End of Data Dictionary":
             break
        attributes = d1.split(",")
        key = attributes[0]
        t = key + "_" + bname[:len(bname)-4] + ".csv"
        temp_file = open(dname + t, "a")


        for d2 in data:
            if d2.startswith(key + ",", 0, len(key)+1) and not d2.startswith("2,", 0,
 2):
                temp_file.write(timestamps[tsc] + "," + d2 + "\n")
```

This sums up what the parseESOKeys() function does and pretty much how the entire parser module operates when attempting to parse .eso files located in the RAW_ESO directory.

II. Data Storage

File: Driver.java

This program handles database insertion and printing.

Function: insertToDB_BMS_R1

Variables:

line1,2,3: used to pull lines of data from file.
results1,2,3: used to store the parsed string from the line entries.
value1,2,3: used to keep track of the values to be inserted into database.
date: keeps track of the date of an entry.
timeStamp: keeps track of the timeStamp of an entry.

insertDB: prepared statement used by the function to insert entries into the database.
myStmt: statement that is used for inserting into the database using insertDB.

Algorithm:

We start off by using our connection to the database and the prepared statement we have created to set up for the insertion of entries.

```
String insertDB = "INSERT INTO BMSR1"
  +"(TimeStamp, Date, Temperature, RelativeHumidity, CO_2, SensibleHeat) VALUES"
  + "(?,?,?,?,?,?)";

PreparedStatement myStmt = myConn.prepareStatement(insertDB);
```

Next since we don't care about the first line in the csv file that has been partially parsed for us, we grab that first line from each of the files we are reading before we enter the whole loop where all the work happens.

```
line1 = in1.nextLine();
line2 = in2.nextLine();
line3 = in3.nextLine();
```

Then we enter the while loop and keep going as long as the file has another line entry that we can read in. Inside the loop we grab the second line from each of the files since this is when the data starts to appear. We then split those lines we have scanned from each of the file by "," since its a csv file and store them in a string array.

```
while(in1.hasNextLine()){
    line1 = in1.nextLine();
    line2 = in2.nextLine();
    line3 = in3.nextLine();
    String[] results1 = line1.split(",");
    String[] results2 = line2.split(",");
    String[] results3 = line3.split(",");

                    ...
}
```

After that's done we need to store the data that has been parsed in the appropriate variables to then be able to set the values in the prepared statement for insertion into the database.

```
while(in1.hasNextLine()){
...
results1[5] = time_adjustment(results1[5]);
date = results1[2] + "-" + results1[3] + "-" + String.valueOf(year) ;
timeStamp = results1[5] + ":" + results1[6];
value1 = results1[10];
value2 = results2[10];
value3 = results3[10];
…
}
```

Finally after all the data has been sorted we set the values for the appropriate sections of the prepared statement for the entry to be inserted into the database. Then execute the update and the entry should now appear in the database. This will happen for every single line in the csv file excluding the first line.

```
while(in1.hasNextLine()){
    ...
    myStmt.setString(1,timeStamp);
    myStmt.setString(2,date);
    myStmt.setString(3,value1);
    myStmt.setString(4,value2);
    myStmt.setString(5, "0.0");
    myStmt.setString(6,value3);
    myStmt.executeUpdate();
}
```

Function: insertToDB_BMS_R2
Variables:

line1,2,3: used to pull lines of data from file.
results1,2,3: used to store the parsed string from the line entries.
value1,2,3: used to keep track of the values to be inserted into database.
date: keeps track of the date of an entry.
timeStamp: keeps track of the timeStamp of an entry.
insertDB: prepared statement used by the function to insert entries into the database.
myStmt: statement that is used for inserting into the database using insertDB.

Algorithm:

We start off by using our connection to the database and the prepared statement we have created to set up for the insertion of entries.

```
String insertDB = "INSERT INTO BMSR2"
  +"(TimeStamp, Date, Temperature, RelativeHumidity, CO_2, SensibleHeat) VALUES"
  + "(?,?,?,?,?,?)";

PreparedStatement myStmt = myConn.prepareStatement(insertDB);
```

Next since we don't care about the first line in the csv file that has been partially parsed for us, we grab that first line from each of the files we are reading before we enter the whole loop where all the work happens.

```
line1 = in1.nextLine();
line2 = in2.nextLine();
line3 = in3.nextLine();
```

Then we enter the while loop and keep going as long as the file has another line entry that we can read in. Inside the loop we grab the second line from each of the files since this is when the data starts to appear. We then split those lines we have scanned from each of the file by "," since its a csv file and store them in a string array.

```
while(in1.hasNextLine()){
     line1 = in1.nextLine();
     line2 = in2.nextLine();
     line3 = in3.nextLine();
     String[] results1 = line1.split(",");
     String[] results2 = line2.split(",");
     String[] results3 = line3.split(",");

                                  ...

 }
```

After that's done we need to store the data that has been parsed in the appropriate variables to then be able to set the values in the prepared statement for insertion into the database.

```
while(in1.hasNextLine()){
...
results1[5] = time_adjustment(results1[5]);
date = results1[2] + "-" + results1[3] + "-" + String.valueOf(year) ;
timeStamp = results1[5] + ":" + results1[6];
value1 = results1[10];
value2 = results2[10];
value3 = results3[10];
...
}
```

Finally after all the data has been sorted we set the values for the appropriate sections of the prepared statement for the entry to be inserted into the database. Then execute the update and the entry should now appear in the database. This will happen for every single line in the csv file excluding the first line.

```
while(in1.hasNextLine()){
    ...
    myStmt.setString(1,timeStamp);
    myStmt.setString(2,date);
    myStmt.setString(3,value1);
    myStmt.setString(4,value2);
    myStmt.setString(5, "0.0");
    myStmt.setString(6,value3);
    myStmt.executeUpdate();
}
```

Function: printfromDB_PV

Variables:

pvStmt: Statement that will be used for querying the database

pvResults: The results from executing the query

Algorithm:

We start off by writing the column name at the beginning of the file separated by commas.

```
out.write("TimeStamp"
+ "," + "Date"
+ "," + "Pac"
+ "," + "Temperature"
+ "\n");
```

Next, we loop through the database table from the beginning to end and print out the results at each loop.

```
while(pvResults.next()){
out.write(pvResults.getString("TimeStamp")
+ "," + pvResults.getString("Date")
+ "," + pvResults.getString("Pac")
+ "," + pvResults.getString("Temperature")
+ "\n");                                                    }
```

Function: printfromDB_BMS_Combined

Variables:

pvStmt: Statement that will be used for querying the database
pvResults: The results from executing the query
out: BufferedWriter that will write out to the file we specify

Algorithm:
We start off by determining the room number with a switch statement.

```
switch (roomNum){
case 1:
out = new BufferedWriter (new FileWriter("BMSOne.csv"));
  break;
case 2:
out = new BufferedWriter (new FileWriter("BMSTwo.csv"));
  break;
}
```

Next, we write the column name at the beginning of the file separated by commas.

```
out.write("TimeStamp"
+ "," + "VarDate"
+ "," + "Temperature"
+ "," + "RelativeHumidity"
+ "," + "COtwo"
+ "," + "SensibleHeat"
+ "\n");
```

Next, we loop through the database table from the beginning to end and print out the results at each loop.

```
while(pvResults.next()){
```

```
out.write(pvResults.getString("TimeStamp")
+ "," + pvResults.getString("Date")
+ "," + pvResults.getString("Temperature")
+ "," + pvResults.getString("Relative-Humidity")
+ "," + pvResults.getString("CO_2")
+ "," + pvResults.getString("Sensible-Heat")
+ "\n");
}
```

Function: printfromDB_BMS
Variables:

bmsStmt: Statement that will be used for querying the database
column: String array that stores name of column from database
columnName: String array that stores the name of the desired column
bmsResults: The results from executing the query
out: BufferedWriter that will write out to the file we specify

Algorithm:

We start off creating a for loop to loop through each variable in the database.

```
for(int i = 0; i < column.length; i++ )
```

Then, we create the out file name based on the room number and columnName.

```
BufferedWriter out = null;
switch (roomNum){
case 1:
out = new BufferedWriter (new FileWriter("BMSOne" + columnName[i] + ".csv"));
  break;
case 2:
out = new BufferedWriter (new FileWriter("BMSTwo" + columnName[i] + ".csv"));
  break; }
```

Next, we write the column name at the beginning of the file separated by commas.

```
out.write("TimeStamp"
+ "," + "Date"
+ "," + columnName[i]
+ "\n");
```

Next, we loop through the database table from the beginning to end and print out the results at each loop.

```
while(bmsResults.next()){
out.write(bmsResults.getString("TimeStamp")
+ "," + bmsResults.getString("Date")
+ "," + bmsResults.getString(column[i])
+ "\n");
}
```

Function: time_adjustment
This helper function will adjust the time from normal time to military time.
Variables:
hour1: variable that will be adjusted
time: temporary variable used to adjust the time to military time

Function: time_parse
This helper function will remove any whitespace that we don't want by looping through the string and replacing whitespace with no space.
Variables:
space: array used to determine where the spaces are in the given string
result: string that has no spaces after we replace spaces with nothing.

Function: data_truncation
This helper function will truncate our data to ensure the successful operation of reading the file we are printing out.
Variables:
extract: initial data values before truncating.
extract1: temporary variable to store truncated data.
extract2: return variable of final truncated data.

Function: delete
This helper function will delete all data from the database.
Variables:
delete: variable to specify delete statement for executing mySQL statement for BMSR1.
delete1: variable to specify delete statement for executing mySQL statement for BMSR2.

delete3: variable to specify delete statement for executing mySQL statement for pv on 05/01.
delete4: variable to specify delete statement for executing mySQL statement for pv on 05/02.
delete4: variable to specify delete statement for executing mySQL statement for pv on 05/03.
delete5: variable to specify delete statement for executing mySQL statement for pv on 05/04.
delete6: variable to specify delete statement for executing mySQL statement for pv on 05/05.

III. Data Display

The following documentation is written by Venkata Karthik Thota for the display module.

The data display module reads the data files (csv files) from the data storage module and displays the different attributes such as temperature and power of the two systems PV(Solar Panels) and BMS. If at any point there is confusion between some of the D3 functions, then please refer to http://d3js.org/ and the read the documentation.

The front end display development stack consists of:
- HTML/CSS
- D3.js
- CSV data files

HTML/CSS - The HTML files, one custom css files, and materialize css library make up what the user see. Only the rendered, interactive display is visible to the user.

File: index.html

The index.html files connects all the other views that display the data. The page rendered by index.html is the first page that is viewed when user visits the dashboard.

Initialize the index.html files with html tags.

```
<!DOCTYPE html>
        .
        .
        .
</html>
```

Declare the title load the css stylesheet dependencies and Javascript libraries

```
<title>BMI DASHBOARD</title>
        <head>
                <!-- Load custom stylesheet -->
                <!-- Load Materialize and Jquery -->
        </head>
```

Inside body tags, declare the navigation bar. The navigation bar is implemented materialize css library. Refer to materialize documentation (http://materializecss.com) for further reference.

```
<nav>
        .
        .
        .
</nav>
```

Inside the body tag, create two tables for displaying the date options for PV (Solar Panels) and BMS. The sections can be created using the HTML tag *div*. Inside the div, load the html files that correspond to the date listed in the table.

```
<!-- Section with dates for PV(Solar Panels) -->
        <div class="row">
                <div class="col s12 m12">
                        <div class="collection">
```

```
                                <h4>Solar Panels - PV</h4>
                        <!— Load the pages —>
                        </div>
                </div>
        </div>

    <!-- Section with dates for BMS -->
        <div class="row">
                <div class="col s12 m12">
                        <div class="collection">
                                <h4>BMS</h4>
                        <!— Load the pages —>
                        </div>
                </div>
        </div>
```

Finally, outside the html tag, declare a script tag that loads the materialize css javascript library.

```
  <!-- Laod materialize js -->
    <script type='text/javascript' src='materialize/js/materialize.min.js'></script>
```

Files: pv1.html, pv2.html, pv3.html, pv4.html, pv5.html

The pv*.html files are layout of the graph. These files load the style sheets, javascript dependencies such as d3.js and the pvGraph*.js files. There are five different pvGraph.js files each designated to one of the five pv*.html files.

Initialize the pv*.html files with html tags

```
    <!DOCTYPE html>
            .
            .
            .
    </html>
```

Declare the title load the css stylesheet dependencies and Javascript libraries

```
<title>BMI DASHBOARD</title>
        <head>
                <!-- Load custom stylesheet -->
                <!-- Load Materialize and Jqurey -->
        </head>
```

Inside body tags, declare/load the D3.js library using the script tags. The D3.js library can be loaded via CDN server or downloaded locally. For faster load time, load D3.js locally. D3 can be downloaded at http://d3js.org/

```
<!-- Load d3.js -->
<script type='text/javascript' src='d3/d3.min.js'></script>
```

Underneath the script tags, declare the navigation bar. The navigation bar is implemented using materialize css library. Refer to materialize documentation (http://materializecss.com/) for further reference.

```
<nav>
        .
        .
        .
</nav>
```

Underneath the nav tag, load the javascript file pvGraph*.js associated with the date that was picked in the main page.

```
<!-- Load 'dashboard' -->
      <script type='text/javascript' src='pvGraph1.js'></script>
```

Lastly, the three functions called via click by the user are crucial for the graph to update base on what attribute was clicked. The three functions temperature(), pac(), and vac() will be further explained below.

```
<a class="col s4 waves-effect waves-light btn" onclick="temperature()">Temperature</a>
<a class="col s4 waves-effect waves-light btn" onclick="pac()">Pac</a>
```

```
<a class="col s4 waves-effect waves-light btn" onclick="vac()">Vac</a>
```

Files: pvGrahph1.js, pvGraph2.js, pvGraph3.js, pvGraph4.js, pvGraph5.js

The pvGraph*.js files are responsible for generating the line chart show in pv*.html web pages.

Firstly, define the margin to display the graph within the SVG

```
var margin = {top: 20, right: 20, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 300 - margin.top - margin.bottom;
```

Use the built in d3.js function called d3.time.format() to format the timestep (timestamp) value from the CSV file (dataC.csv)

```
d3.time.format("%H:%M").parse;
```

A graph consists of a x-axis and y-axis. So first define, the x-scale(range of x values) and the y-scale(range of y values). The x-scale is a time scale and the y-scale is linear scale. Then use the scales to define both x and y axes. See D3 documentation for more info on time scales and different types of time scale.s

```
var x = d3.time.scale().range([0, width]);
var y = d3.scale.linear().range([height, 0]);

var xAxis = d3.svg.axis().scale(x).orient("bottom");
var yAxis = d3.svg.axis().scale(y).orient("left");
```

Define the line being plotted using d3.svg.line().interpolate() function

```
var line = d3.svg.line()
```

```
    .interpolate("basis")
    .x(function(d) { return x(d.TimeStep); })
    .y(function(d) { return y(d.Temperature); });
```

Define the SVG (scalar vector graphics) where the graph will be drawn onto SVG has two important attributes and those are height and width.

```
var svg = d3.select("body").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
```

Load the data file using the d3 in-built function called d3.csv() to load the csv files. And then read and parse the data into type integer.

```
d3.csv("dataC.csv", function(error, data) {

  dataset = data.filter(function (d){
    return d.VarDate === "2015-05-01";
  });

    // read and parse the values as int
   dataset.forEach(function(d) {
      d.TimeStep = +parseTime(d.TimeStep);
      d.Temperature = +d.Temperature;
      d.Vac = +d.Vac;
      d.Pac = +d.Pac;
      d.VarDate = d.VarDate;
   });
...
});
```

Once the data has been successfully read and parsed, use the values to determine the domain values for both x-axis and y-axis

```
 x.domain(d3.extent(dataset, function(d) { return d.TimeStep; }));
 y.domain(d3.extent(dataset, function(d) { return d.Temperature; }));
```

Finally, we put together all the components that defined. Append the x-axis and y-axis with the label and draw the line corresponding to the data values

```
svg.append("g")
     .attr("class", "y axis")
     .call(yAxis)
   .append("text")
     .attr("transform", "rotate(-90)")
     .attr("class","tmp-text")
     .attr("y", 6)
     .attr("dy", ".71em")
     .style("text-anchor", "end")
     .text("Temperature");

 svg.append("path")
     .datum(dataset)
     .attr("class", "line")
     .attr("d", line);
```

The last function updateChart(ds) takes in a parameter. This parameter is passed into the conditional statement that determines what attribute and its domain to plot the line that needs to be drawn. The values for ds are: “Temperature”, “Pac” and “Vac.

The three functions temperature(), pac(), and vac() are associated with the buttons in the views (pv*.html files). These functions on button click invoke the updateChart function

```
function temperature() {
  updateChart("Temperature");
}
function pac() {
  updateChart("Pac");
}
function vac() {
  updateChart("Vac");
}
```

Files: bms1.html

The bms HTML files are structured very similar to the pv*.html files. These files load the style sheets, javascript dependencies such as d3.js and the

graphBMS1.js files. However, there are a total of eight functions that are invoked on button click. These four functions are explained further below.

```
          <!-- Load 'dashboard' -->
           <script type='text/javascript' src='graphBMS.js'></script>
           <script type='text/javascript' src='materialize/js/materialize.min.js'></script>
               <!-- Room 1 -->
              <a class="col s4 waves-effect waves-light btn center"
onclick="temperatureB1()">Temperature</a>
              <a class="col s4 waves-effect waves-light btn center" onclick="relhumB1()">Relative
Humidity</a>
              <a class="col s4 waves-effect waves-light btn center" onclick="cotwoB1()">CO 2</a>
              <a class="col s4 waves-effect waves-light btn center" onclick="senheatB1()">Sensible
Heat</a>

        </br>
        </br>
               <!-- Room 2 -->
              <a class="col s4 waves-effect waves-light btn"
onclick="temperatureB2()">Temperature</a>
              <a class="col s4 waves-effect waves-light btn" onclick="relhumB2()">Relative
Humidity</a>
              <a class="col s4 waves-effect waves-light btn" onclick="cotwoB2()">CO 2</a>
              <a class="col s4 waves-effect waves-light btn" onclick="senheatB2()">Sensible
Heat</a>
```

Files: graphBMS1.js

The graphBM1.js file draws the graph for the BMS Room 1 and BMS Room 2 data. The structure of setting the graph is similar to the pvGraph*.js files. However, there are few changes.

First of all, different values are being parsed and tested.

```
      // parse the values as type int
        dataset.forEach(function(d) {
           d.TimeStamp = +parseTime(d.TimeStamp);
           d.Temperature = +d.Temperature;
           d.RelateiveHumidity  = +d.RelateiveHumidity ;
           d.COtwo = +d.COtwo;
           d.SensibleHeat = d.SensibleHeat;
           d.VarDate = d.VarDate;
        });
```

The updateChart(ds) function has a slightly different if-else condition that checks which attribute to portray on the display.

```
    if (ds === "Temperature") {
      x.domain(d3.extent(dataset, function(d) { return d.TimeStamp; }));
      y.domain(d3.extent(dataset, function(d) { return d.Temperature; }));

      line = d3.svg.line()
          .interpolate("basis")
          .x(function(d) { return x(d.TimeStamp); })
          .y(function(d) { return y(d.Temperature); });
    }
    else if (ds === "COtwo") {
      x.domain(d3.extent(dataset, function(d) { return d.TimeStamp; }));
      y.domain(d3.extent(dataset, function(d) { return d.COtwo; }));

      line = d3.svg.line()
          .interpolate("basis")
          .x(function(d) { return x(d.TimeStamp); })
          .y(function(d) { return y(d.COtwo); });
    }
    else if (ds === "SensibleHeat") {
      x.domain(d3.extent(dataset, function(d) { return d.TimeStamp; }));
      y.domain(d3.extent(dataset, function(d) { return d.SensibleHeat; }));

      line = d3.svg.line()
          .interpolate("basis")
          .x(function(d) { return x(d.TimeStamp); })
          .y(function(d) { return y(d.COtwo); });
    }
    else {
      x.domain(d3.extent(dataset, function(d) { return d.TimeStamp; }));
      y.domain(d3.extent(dataset, function(d) { return d.RelateiveHumidity ; }));

      line = d3.svg.line()
          .interpolate("basis")
          .x(function(d) { return x(d.TimeStamp); })
          .y(function(d) { return y(d.RelateiveHumidity ); });
    }
```

And finally, there are eight functions. Four functions relate to room one and other four relate to room two. These functions invoke updateChart(ds) function on click of a button. The first parameter that are being passed are “Temperature”, “COtwo”, “RelativeHumidity”, and “SensibleHeat”. The second parameter that is

being passed is the file names (BMSOne.csv and BMSTwo.csv). The phrase in the variable “B1” and “B2” distinguish between room 1 and room 2.

```
function temperatureB1() {
  updateChart("Temperature" , “BMSOne.csv”);
}

function cotwoB1() {
  updateChart("COtwo", “BMSOne.csv”);
}

function relhumB1 () {
  updateChart("RelateiveHumidity", “BMSOne.csv”);
}

function senheatB1() {
  updateChart("SensibleHeat", “BMSOne.csv”);
}



function temperatureB2() {
  updateChart("Temperature", “BMSTwo.csv”);
}

function cotwoB2() {
  updateChart("COtwo", “BMSTwo.csv”);
}

function relhumB2 () {
  updateChart("RelateiveHumidity", “BMSTwo.csv”);
}


function senheatB2() {
  updateChart("SensibleHeat", “BMSTwo.csv”);
}
```

VI. Naming Conventions

For the project, we used camelcase as our primary naming convention.

VII. Sources

D3.js - http://d3js.org/

Materialize - http://materializecss.com/

Line Chart Example - http://bl.ocks.org/mbostock/3883245

Python - https://docs.python.org/3.5/

Oracle JDB - http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html