~~CSC110 course work~~

~~Motor Vehicle Safety Act~~

~~Canadian Environmental Protection Act~~

~~Federal Agenda on Cleaner Vehicles, Engin~~ ~~uels~~

~~On-Road Vehicle and Engine Emissio~~ ~~tions~~

~~Off-Road Compression-Ignition E~~ ~~Regulations~~

~~Heavy-duty Vehicle and En~~ ~~Greenhouse Gas Emission Regulations~~

The Amendment trend: - ~~On-Road Vehicle and Engine Emission Regulations~~

Observing Emission Regulations in Canada through Odd Oxygen

CSC110

Salwa Abdalla, Majda Lojpur, Cornad Stanek, Devan Srinivasan

The Amendment Trend: Observing Emission Regulations in Canada through Odd Oxygen

Salwa Abdalla, Majda Lojpur, Cornad Stanek, Devan Srinivasan

Monday December 14th, 2020

Abstract

The continuous measurement of nitrogen dioxide (NO_2) and ground-level ozone (O_3) was conducted across Canada from 1974 to 2020. This is measured by the federal government through the Environment and Climate Change Canada's National Air Pollution Surveillance Program (NAPS). It is worth noting that Ozone and Nitrogen Dioxide are secondary pollutants, meaning that they are created in the atmosphere due to a chemical imbalance caused by primary pollution (like car exhausts or factory pollution). The reactions that produce these compounds are cyclical, which helps visualize why they are closely related, and can help explain why lowering their general concentration will have a cyclical positive effect on the whole system of pollutants. By visually representing NO_2, O_3 and a close relationship between the two we can analyze which regulations and policies affect the levels of pollutants in the air.

Introduction

Ozone (O_3) and Nitrogen Dioxide (NO_2) are some of the largest contaminants in urban areas, and have clinically proven adverse effects on human health and the environment across Canada (Wood, 2012).

For the past half a century, Canada committed to monitoring the air quality across the country in the hopes to reduce the concentration of ground-level ozone and the levels of smog. The reasoning behind these changes was that at low atmospheric levels, these pollutants are devastating to the air quality and contribute to climate change by warming the earth. Additionally, they increase agricultural crop damage as well as cause respiratory issues for many species including humans (Access Environment, 2016).

Ozone retains heat, so when large amounts are continually released into the atmosphere, the atmosphere traps massive quantities of heat energy which disperses onto Earth and warms it. Nitrogen Dioxide depletes the ozone layer, which amplifies The Greenhouse Effect and furthermore traps more heat in the atmosphere. The concentration trends of these chemicals are remarkably closely related, so the term "Odd Oxygen" is used, and it represents the sum of the two pollutants (usually measured in concentration of parts per billion/million) (Wood, 2012). The chemical reaction is as shown below:

$$NO_2 + light \longrightarrow NO + O \qquad \text{(direct affect on smog)}$$

$$O + O_2 \longrightarrow O_3 \qquad \text{(direct affect on local temperature fluctuations)}$$

$$O_3 + NO \longrightarrow O_2 + NO_2 \qquad \text{(direct relationship to air pollution)}$$

Within the recent years, climate change has begun showing its effects visibly, whether it be in wildfire season destroying more land, the ocean levels rising or even the general temperatures around the world departing from what was considered the norm. In the past decades the Government of Canada has passed multiple bills, agendas and acts to cut down on emissions and air pollution in general.

Using this data and the three chemical relationships above, have the laws and regulations passed achieved their goals? More specifically, how were the trends of Odd Oxygen (and it's constituents) affected by such regulations. This can confirm what climate change initiatives and policies are working, and unveil ones that aren't. This is imperative as such realizations will contribute to a positive change in the planet and its natural wildlife, as well as a prosperous future for humanity.

With several amendments, Canada has introduced regulations and policies such as the Motor Vehicle Safety Act (1971), the Canadian Environmental Protection Act (1999), Passenger Automobile and Light Truck Greenhouse Gas Emission Regulations (2010) and Multi-Sector Air Pollutant Regulations (MSAPR) (2016) and more (Emission Standards, 2017).

So the topic of this research report is the following: **How do the emissions acts passed within the past decades actually affect the concentration of Odd Oxygen in the air?**

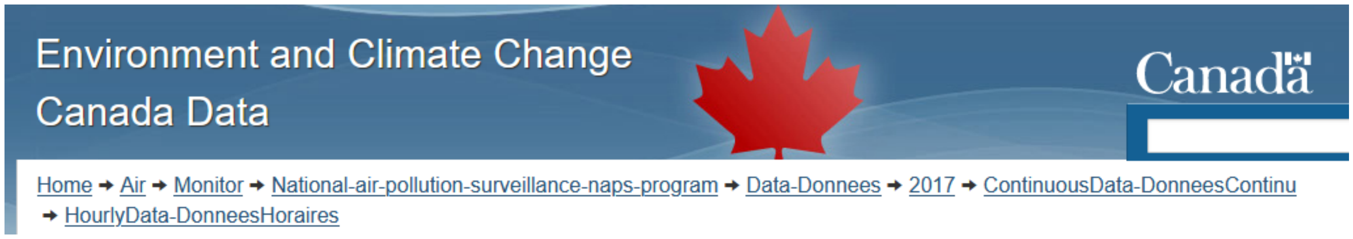
Dataset

Canada has a large database of long-term air-quality monitoring data from collection stations across the country (e.g. St. Johns, Toronto, Vancouver). The name of the dataset is the National Air Pollution Surveillance (NAPS) Program. In general it is the source of "ambient air quality data" with nearly 260 stations dating back to 1969 (Environment and Climate Change Canada Data, 2020).

To measure and compare these concentrations of Odd Oxygen in the air in Canada, we will be accessing a data set from the Government of Canada that is recorded in a unit of ppb (parts per billion). This unit measurement is very helpful since it is relative to the amount of atoms present, meaning we can ignore the density of the various locations or altitudes.

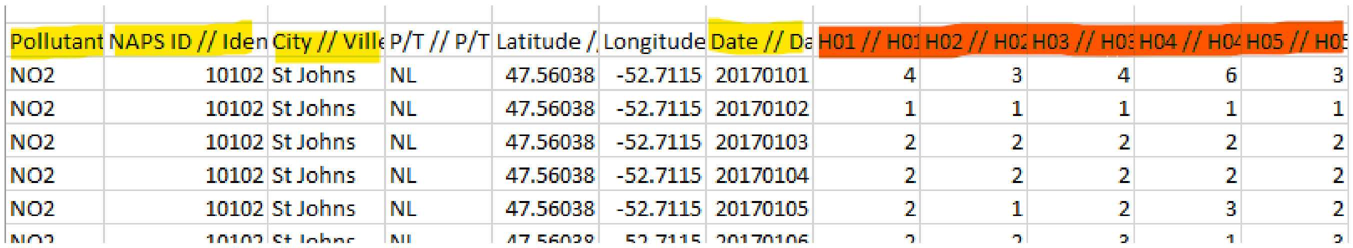
The data source we will use is: http://data.ec.gc.ca/data/air/monitor/national-air-pollution-surveillance-naps-program/. The format of the data set is csv. The data is available at 7-day increments for the years 1974 to 2020. We will focus on the past decades and see if any bills passed with the intent of lowering pollution actually changed the levels of Odd Oxygen in the air. We will specifically be using the NO_2 and O_3 data sets to calculate and graph the changes in Odd Oxygen over the years.

The majority of the regulations and policies that were put into effect that affect Odd Oxygen were in between the years 1995 and 2016, so for the sake of display purposes we've only used data from in between those years. The path to get to the data is as follows:


Environment and Climate Change Canada Data

Home ➔ Air ➔ Monitor ➔ National-air-pollution-surveillance-naps-program ➔ Data-Donnees ➔ 2017 ➔ ContinuousData-DonneesContinu ➔ HourlyData-DonneesHoraires

- National-air-pollution-surveillance-naps-program
- Data-Donnees
- Year Value (Select any year between 1995 to 2016 inclusive)
- ContinuousData-DonneesContiu
- HourlyData-DonneesHoraires
- Molecule (Select O_3_(year).csv and NO_2_(year).csv)



Pollutant	NAPS ID // Iden	City // Ville	P/T // P/T	Latitude /	Longitude	Date // Da	H01 // H01	H02 // H02	H03 // H03	H04 // H04	H05 // H05
NO2	10102	St Johns	NL	47.56038	-52.7115	20170101	4	3	4	6	3
NO2	10102	St Johns	NL	47.56038	-52.7115	20170102	1	1	1	1	1
NO2	10102	St Johns	NL	47.56038	-52.7115	20170103	2	2	2	2	2
NO2	10102	St Johns	NL	47.56038	-52.7115	20170104	2	2	2	2	2
NO2	10102	St Johns	NL	47.56038	-52.7115	20170105	2	1	2	3	2
NO2	10102	St Johns	NL	47.56038	-52.7115	20170106	2	2	3	1	3

The data columns that are used are the City (and Station Number), Date, and H1 to H24 (which are the hour values for each day). Specifically, The O_3_(year).csv measure the level of O_3 pollutants, the NO_2_(year).csv measures the level of NO_2 pollutants and we combine the two values to get the Odd Oxygen Value.

Computational Overview

To begin, let's examine how our data is parsed which we did utilising the Python CSV Module. The python file titled loading_data.py is where our data transformation and filtering took place. In this file is a class called DataFile. Its purpose is to store a CSV file and can return data to be computed and/or plotted. This class is utilized in our main application framework. The instance attributes include:

- data: stores all the data (A List of Lists that contains Any data type)
- file_path: string to store the csv file path (A string)
- header_row: represents the column headers (A List that contains Any data type)
- stations: represents data rows corresponding to station ids (A Dictionary containing string keys that map to Lists of integers, which are the key-values.)
- pollutant: pollutant in this data file (A string)
- year: year this data was collected (A string)

This class contains an initializer method, titled _ _ init_ _ where the parameters include self, and a file path in the form of a string. Each instance attribute is initialized to an empty version of its data type. The second method is titled format, which was created to ease the difficulty of accessing the data. Here is the code for reference:

```
for i in range(len(self.data)):
    for j in range(len(self.data[i])):
        if j == 1 and len(self.data[i][j]) < 6:  # format ID
            self.data[i][j] = '0' + self.data[i][j]
        if j == 6:  # format DATE
            self.data[i][j] = self.data[i][j].replace('-', '')
            self.data[i][j] = self.data[i][j].replace('/', '')
        if i > 0 and 7 <= j <= 31:  # format measurements
            self.data[i][j] = float(self.data[i][j])
```

It standardized the data appearance for the dates in the data. The method consisted of a nested for loop. The outer for loop's range is from zero, to the length of the instance attribute data. Since range is exclusive, the variable i iterates from 0 all the way to the length of the instance of the class' instance attribute: data minus 1. Since data is a List of Lists, containing Any data type, the second for loop iterates through the length of each list within the data list, using the variable i from the outer loop to access the index of each internal list. Since range is exclusive, in the inner loop the variable j iterates from 0 all the way to the length of the list in data at index i minus 1. Within the inner for loop are three if branches. Since we discovered there was inconsistent formatting within the data sets we were using, parsing it would be difficult which is exactly why we created this method. After examining the inconsistencies within the data set, such as dashes or slashes or spaces, the pattern was found and in the three if branches we identify the cases where there are inconsistencies such as the ones named and then replace them to match the rest of the data set. For example, if the j is equal to six that means at self.data[i][j] there is a dash that we then choose to replace with a space via variable reassignment. Similar actions with the appropriate formatting adjustments are performed for other cases such as when j == 1 and len(self.data[i][j]) < 6 or i > 0 and 7 <= j <= 31. The next method within this class utilizes the Python CSV Module, and is titled load. The only parameter is self, so in other words the instance of the class. Here is the code for reference:

```
found_header = False
self.data = []
with open(self.file_path, 'r') as file:
        csv_reader = csv.reader(file, delimiter=',')
        line_count = 0
        for row in csv_reader:  # csv reader using module
            if not found_header and helper_header(row):  # finding table header
                found_header = True
                self.header_row = row
                self.data.append(row)
                line_count += 1
                continue
            if found_header:
                good_row = False
            if '-999' not in row:
                good_row = True
                self.data.append(row)
                line_count += 1
                st_id = row[1]
            if st_id in self.stations and good_row:  # updating stations dictionary
                self.stations[st_id][1] = line_count - 1
            elif good_row:
                self.stations[st_id] = [line_count - 1, line_count - 1]
    file.close()

    self.pollutant = self.data[1][0]
    self.year = self.data[1][6][0:4]
    self.format()
```

This method's purpose is to read the csv file path stored in the instance attribute file_path to the collection : data. First, a variable found_header is set to the boolean value False. self.data is also set to an empty list. Next, in order to give Python access to a file by opening it, we utilized the open() function as so:

```
with open(self.file_path, 'r') as file:
csv_reader = csv.reader(file, delimiter=',')
line_count = 0
```

We used the word 'file' to refer to the file object. Contained within the open function is also a for loop, which iterates through each row in 'csv_reader' which is a variable we created using the Python CSV Module, accessing the csv.reader method of the csv library as can be seen above. This method utilizes the stream reader which is an object in Python that reads the file line by line. There are two if branches within the for loop. The first one is responsible for finding the table header. If the conditions are met, which are if not found_header and helper_header(row) Then it performs a few things. First of all, helper_head is a helper method within the class which checks if the length of the row is less than two, and if so returns False. So, if the conditions are met for this if statement the variable found_header is set to True and it sets the self.header_row to the current row on that iteration, and appends that same row to the lists of lists which is the instance attribute data. The line_count is also increased by 1. Then we continue on in the for loop, utilising the Python built-in function 'continue'. The next conditional is simply if found_header, which is a boolean. So if found_header is true, there are a series of if statements whose task is to determine the various stations contained within the data set and update the station's dictionary. In this segment of code there are two if branches and one elif, and depending on what conditions are met the self.stations attribute could be modified, more rows appended to self.data, and the line_count increased. Once the for loop completes every iteration, we close the file as we no longer need it. Then the instance attributes self.pollutant and self.year are modified, and the last step involves calling the method format which was previously described and explained above. There are many other helper methods contained within this class, but one more significant one is the return_plot_daily which takes in the instance of the class itself and the station_id which is a string. The method returns the x-coordinates and corresponding y-coordinates for the daily average emissions given a station_id. Note, that a station_id represents one of the many possible stations across Canada which record levels of odd oxygen in the air. This method also utilizes a for loop, and initializes two empty lists to which it appends x and y coordinates.

Now, this brings us to the file generated_graphs.py. In this file, it generates the necessary DataFiles instances (the class discussed above) for the viewing part of our project, and preloads them into the software. First, a few imports are made such as the class DataFile from the file loading_data.py and the Python Library Pygame. We also import two other files that we implemented, which are graph.py and compute.py. In this file the CSV data file is read. The function generate_time_graphs does the following: It uses the DataFile class to return hourly plots of concentration versus time. The specific method used to do this is in the file loading_data.py and is the method called return_plot_hourly which is part of the DataFile class. Then, it uses that information as well as the various variables and classes stored in the file dataclass.py to make a graph, and then returns all of these graphs. In addition, it also makes the O_3 versus NO_2 graph by using a function from the file compute.py which is called gen_points_matching_date. To make the O_x graph we used a function called add_values which is from the compute.py file. Once these graphs are preloaded they are previewed in the main.py file. In terms of how we put graphs on the screen, the current graph for previewing is stored in gui which is a GuiSlider class. Then the main function init_visuals starts up the whole visual setup and in the while loop of generated_graphs.py the current graph's function is always called which draws the graph on the screen using pygame and pixel-coordinate ratios.

The file compute.py is responsible for all of the computations that are performed on the collected data. I will go through the most important functions, which were the regressions. Here is the code for simple_linear_regression for reference:

```
x_bar, y_bar = calculate_average_collection(points)
x_bracket = [x - x_bar for x in points[0]]
y_bracket = [y - y_bar for y in points[1]]
b_top = sum(x_bracket[i] * y_bracket[i] for i in range(len(x_bracket)))
b_bottom = sum(x_b ** 2 for x_b in x_bracket)
b = b_top / b_bottom
a = y_bar - b * x_bar
return (a, b)
```

First there is the function called simple_linear_regression, whose parameter is a variable called points. This variable's data type is a tuple of two lists that contain floats. The first list represents x values and the second one represents y values. The coordinates pairs are formed by matching up x and y values that are identical in index. In this function we also utilize a helper function titled calculate_average_collection which returns the two averages of a collection of numbers passed in as a tuple. The first thing that happens in the function simple_linear_regression is that the variable: points is passed through the helper function calculate_average_collection and from the returned tuple we create two variables, one for each float in the tuple, via parallel assignment. The variable x_bar refers to the average of the x-values in the tuple points and the y_bar refers to the average of the y-values in the tuple points. Next, for every item within the list at index 0 of the tuple points (so the list representing x-values) we take x_bar and subtract it from each value. The same thing is done for the list in points representing the y-values. Next, with the use of a list comprehension, corresponding x-y pairs (identified with having an identical index) are multiplied and then once the list is complete we take the sum of it and this is stored in the variable b_top. Then, the variable b_bottom represents the sum of every value in the list x_bracked squared. Once all these variables are accounted for, this allows us to return the linear regression, as the pair of floats that the function returns (a, b) are as follows:

```
a = y\_bar - b * x_bar
b = b_top / b_bottom
```

Note that the line y = mx + b is an approximation of this data. This concludes the linear regression.

The next function is polynomial_regression which calculates the coefficients of a polynomial function with the degree that best fits the input data. Here is the function body for reference:

```
    matrix_r = Matrix(deg + 1, deg + 1)
    matrix_l = [[]]

    # fills the coefficient matrix
    for row in range(0, (deg + 1)):
        for column in range(0, deg + 1):
            matrix_l[row].append(sum([pow(x, row + column) for x in x_val]))
        if row != deg:
            matrix_l.append([])

    constant_vec = []
    x_temp = 0

    # fills the column vector of constants
    for row in range(0, (deg + 1)):
        for i in range(0, len(x_val)):
            x_temp += pow(x_val[i], row) * y_val[i]
        constant_vec.append(x_temp)
        x_temp = 0

    matrix_r.set_matrix(matrix_l)
    return matrix_r.solve(constant_vec)
```

This function takes in three parameters:

- deg which is an integer and represents the degree of the polynomial function we are calculating the coefficients for.
- x_val which is a list of floats representing the x-values of the inputted data
- y_val which is a list of floats representing the y-values of the inputted data

This function utilizes a class that was implemented in a separate file called matrix.py, which I will explain in parallel with the function polynomial_regression. Just for some context, the class Matrix is responsible for the 2D list implementation of a matrix. Its instance attributes are:

- rows: number of rows (An integer)
- columns: number of columns (An integer)
- Matrix: the 2D list that holds the entries (A List of Lists)

Back to the function polynomial_regression. Within this function we initialize two variables. The first is an object matrix_r which is an instance of the Matrix class. Then, there is a list nested within a list that is stored in the variable matrix_l. Then comes a nested for loop. The outer for loop is responsible for filling the coefficient matrix, and the loop variable: row has values from 0 to the deg (since range is exclusive). Then, the inner for loop whose loop variable column also takes on values from 0 to deg (since range is exclusive). Within this for loop, depending on the value of the outer for loop variable: row, the following is appended to the list of lists matrix_l:

```
    matrix_l[row].append(sum([pow(x, row + column) for x in x_val]))
```

Then after that, there is an if branch, and if the loop variable of the outer loop 'row' is not equal to deg, then another empty list is appended to matrix_1. After this nested for loop executes, we create two more variables which are constant_vec (an empty list) and x_temp which is equal to 0. The second nested for loop is responsible for filling the column vector constants and executes as follows:

```
for row in range(0, (deg + 1)):
   for i in range(0, len(x_val)):
       x_temp += pow(x_val[i], row) * y_val[i]
```

This brings us to where we utilize the Matrix class I mentioned above. Since matrix_r was initialized as an object of the Matrix class, this means we can utilize all of the methods available in the Matrix class. We first access the method: set_matrix and pass our list matrix_l through it. Now, the set_matrix method is responsible for changing the matrix instance attribute, which was previously initialized as an empty list. Instead, now that we passed our list of lists matrix_l through it, it reassigns the instance attribute matrix to matrix_l. It is done in the following fashion:

```
    self.matrix = matrix
```

Once that has been done, then we access the: solve method within the Matrix class, and pass the list constant_vec through it which we created in our second nested for loop. The solve method within the Matrix class is responsible for solving a square matrix for a given list of constants by row reduction. It outputs the list of variables corresponding to the order of the columns. The first nested for loop puts the matrix in row echelon form. The outer loop variable is column which goes from the value 0 to the instance attribute columns - 1. The inner for loop variable is row, which goes from the value of column + 1 to the instance attribute rows - 1. This nested for loop creates and modifies a variable called multiple which is used later on. It also modifies the variable matrix, which was a copy of self.matrix, as to not mutate the original instance attribute. Lastly, it also modifies the parameter constants which is a list of lists. The nested for loop that proceeds this one puts the matrix in reduced row echelon form, and a for loop after that makes the pivots 1. After making the pivots 1, in which the parameters constants is modified, the method returns constants which is now a list of floats, also a solved square matrix for the given list of constants. It is important to note that this method relies on the assumption that the matrix object has a unique solution.

The next regression that we calculated in this file is exponential regression, done in the function exponential_regression. Here is the function body code for reference:

```
log_values = []
for value in y_val:
    log_values.append(log(value, 10))
coefficients = polynomial_regression(1, x_val, log_values)
return [pow(10, coefficients[1]), pow(10, coefficients[0])]
```

It calculates the coefficients of an exponential function that best fits the given input. The output is a list in the form [a, b], where $y = b * a^x$. This function takes in two lists, one is represented by the variable x_val which is the list of x-values, and the other list, y_val, represents a list of the y-values that we found from our data set. A list is initialized under the name log_values, and then we proceed with a for loop. The for loop iterates through each y-value in the list y_val. Then, each y-value is logged to the base of 10 and this resulting value is appended to the list we initialized at the start under the name log_values. After this is completed, the list of x-values, x_val, and the list log_values, representing the modified y-values, are passed through the function polynomial_regression for the first degree, which calculates the coefficients of the desired degree polynomial. The output is ordered in increasing degree. Note this function was discussed in further detail above. The returned coefficients are stored in a variable titled coefficients. Once all this has been completed, we can return the coefficients of an exponential function that best fits the given input, as we do the following:

```
coefficients = polynomial_regression(1, x_val, log_values)
return [pow(10, coefficients[1]), pow(10, coefficients[0])]
```

This concludes exponential regression and the functions worth noting within the compute.py file.

Lastly, we will touch upon the graphing software which is located in the file main.py. One of the imports made was the file user_input, which contains the method u_input.Whenever someone clicks or does any action with the mouse or mousepad it then triggers the method u_input to initiate the mouse's events. This event is processed and depending on what was clicked, the appropriate method is then called (storing a graph, viewing the right graph) so we used pygame to do the following: Using pixel to coordinate ratios relative to the screen size (everything adjusts to the screen size, meaning nothing is hard coded) we plot our graphs completely by scratch. To do so, we plot points then connect them with line segments so that the whole graph can be stretched or compressed and everything is accommodated for using the software's immense relative math calculations (relative to the screen and plots). Additionally you can pan throughout the whole graph and view all of using the two sliders which again restructure the whole graph every time. These methods are also acheived using the scroll wheel / mouse pad which uses x and y coordinates of your mouse to pan throughout the graph and zoom in on your cursor. All of this was using pyagme's event inputs and manually coding and implementing relevant math calculations. The biggest use of the pygame library was to manually made buttons and labels for user input. We used the Python Library plotly to trace scatter plots and generate multiple traces on one figure. This was so we can easily view the data in an intuitive manner, as you can plot any graph and whenever you use the pygame interface to restrict the section of the graph you view (i.e zoom in, compress, stretch) only the portion on the window is graphed. Of course if one was to completely zoom out, plotly plots the whole graph. This same flexibility and choice applies to the graphs of regressions too. Regressions are completely handled relative to the section of the graph the user decides to view. Regressions are only available when plotted using the "perform regression" button on plotly. The key feature is that the user can decide which regression they want to view. While we mentioned abstractly what we used the pygame and plotly libraries for, we will now give a few concrete examples of how we put these libraries to use.

First of all, in the file graph.py, in the class Graph, there is a method called draw_graph that draws the graph on the screen.We use pygame to update the display while keeping track of the signals. We update the display as so:

pygame.display.flip(). There are also two helper methods within that method called helper_draw_graph_scale and helper_draw_graph_items where the pygame library is used. In helper_draw_graph_scale we used FONT.render to generate what the font should look like and .blit (block image transfer) actually transfers the font onto the screen. In helper_draw_graph_items we used pygame.draw.lines to draw the points, pygame.draw.line to draw the axes, and pygame.transform.rotate to position the y-axis. We also used FONT.render and .blit again to display certain labels, such as the x and y axis. The second heavy use of pygame can be found in the main.py file under the init_visuals function. This function's purpose is to draw all the buttons and sliders when the main loop is first called. In other words it initializes the whole application. We used pygame.draw.rect to draw the buttons, FONT_BUTTON.render for the plotly buttons, as well as to store graph button labels. These were then transferred on the screen via window.blit. We also included sliders which were drawn using pygame.draw.rect and the circles for the sliders were drawn via pygame.draw.circle. In terms of the plotly library, in the graph.py file for the function plotly_with_reg it uses plotly to plot the currently viewed graph and the three regressions on top of it. Here is a sample of plotly being used:

```
fig.add_trace(go.Scatter(x=new_x_portion, y=self.y_portion,
mode='lines+markers', name=pollutant))
```

And lastly to conclude, we used class abstraction in the files user_input.py and dataclass.py.

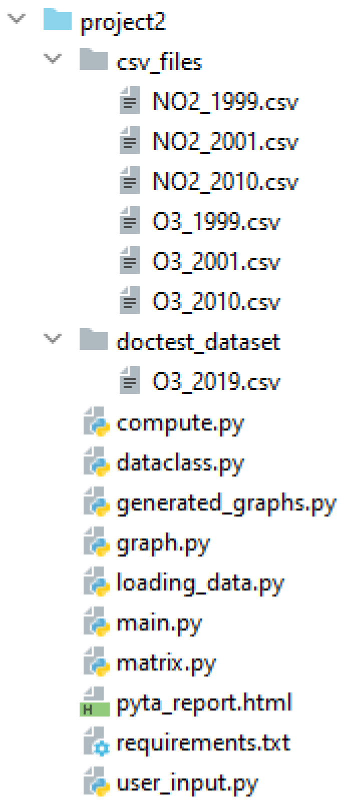
Instructions

1. The data set that is required too large to upload to markus. These should be saved under the subfolder "csv_files". The raw source for the data can be located here for "O3_(year)" and "NO2_(year)" for all three links listed below.:
   - http://data.ec.gc.ca/data/air/monitor/national-air-pollution-surveillance-naps-program/Data-Donnees/1999/ContinuousData-DonneesContinu/HourlyData-DonneesHoraires/?lang=en
   - http://data.ec.gc.ca/data/air/monitor/national-air-pollution-surveillance-naps-program/Data-Donnees/2001/ContinuousData-DonneesContinu/HourlyData-DonneesHoraires/?lang=en
   - http://data.ec.gc.ca/data/air/monitor/national-air-pollution-surveillance-naps-program/Data-Donnees/2010/ContinuousData-DonneesContinu/HourlyData-DonneesHoraires/?lang=en
2. The data used to run in our doctest is also located in a file "doctest_dataset". (This is available on markus). It should be saved under a folder of the same name. The raw source for the data can be located here for "O3_2019":
   - http://data.ec.gc.ca/data/air/monitor/national-air-pollution-surveillance-naps-program/Data-Donnees/2019/ContinuousData-DonneesContinu/HourlyData-DonneesHoraires/?lang=en

3. Install all of the libraries that are in the ”requirements.txt” that are the following:

   - pytest, python-ta, plotly, pygame==2.0.0.dev10
   - math, csv, typing, datetime, random

4. The workspace should look as below:



```
project2
    csv_files
        NO2_1999.csv
        NO2_2001.csv
        NO2_2010.csv
        O3_1999.csv
        O3_2001.csv
        O3_2010.csv
    doctest_dataset
        O3_2019.csv
    compute.py
    dataclass.py
    generated_graphs.py
    graph.py
    loading_data.py
    main.py
    matrix.py
    pyta_report.html
    requirements.txt
    user_input.py
```

5. READ ME:

When in the application, there are interactive features we offer. This is comprised of 7 buttons you can press and 3 sliders, as well as scrolling capabilities. Below is a list of the following capabilities and a visual of what they do.

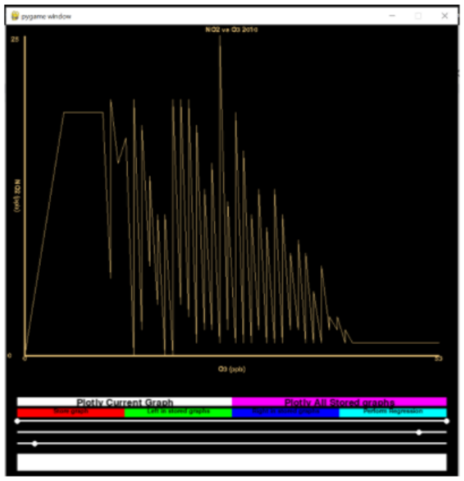
Start the Application:

1. Run the main.py file
   a. It will prompt you for a window height and width. These must be larger than 350 pixels for an effective experience. We recommend 800x800.
   b. After this just wait a few seconds and the application will open in a new window

In the Application:

Note: All images are small so the document takes less space and is easier to view. Please zoom in as needed to properly view the photos.

1. You're screen should look like this:



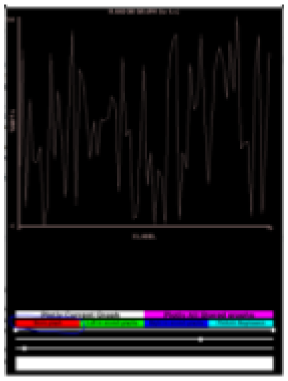
The purpose for our research was to explore the changes in concentration of pollutants relative to important times in history, as well as the trend relations between NO2 and O3.

When in the application, there are interactive features we offer. This is comprised of 7 buttons you can press and 3 sliders, as well as scrolling capabilities. Below is a list of the following capabilities and a visual of what they do.

Buttons

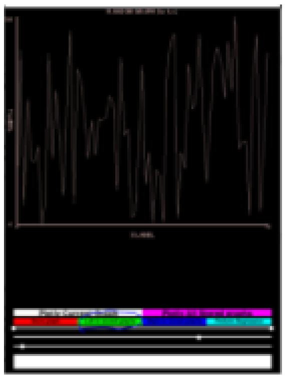
1. Store Graph

When viewing a new graph, click this to store it in your saved graphs inventory.



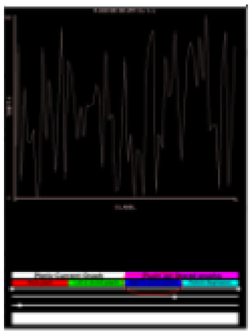
2. Left Stored Graph

Click this to view the graph left of the current one in the inventory.



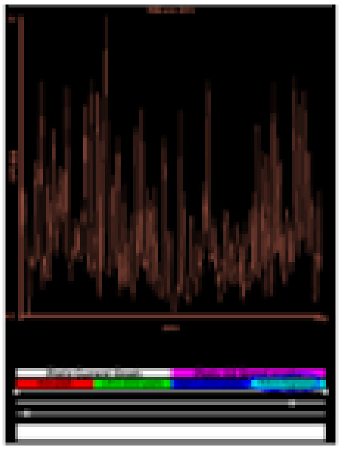
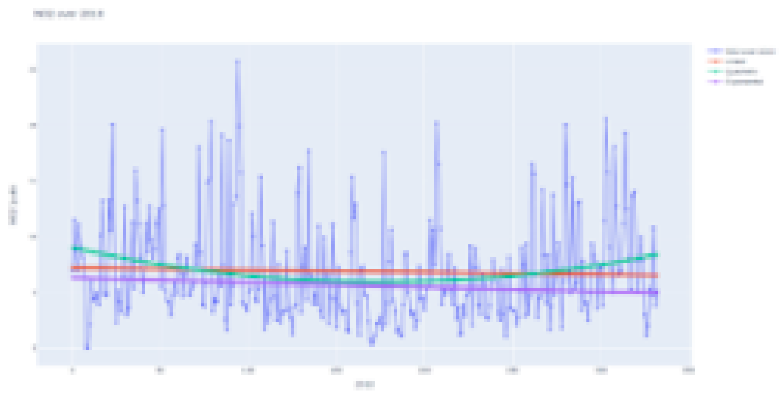
3. Right Stored Graph

Click this to view the graph right of the current one in the inventory.



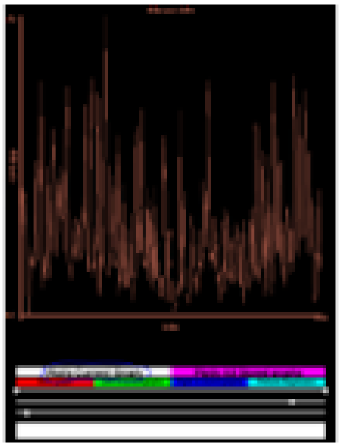
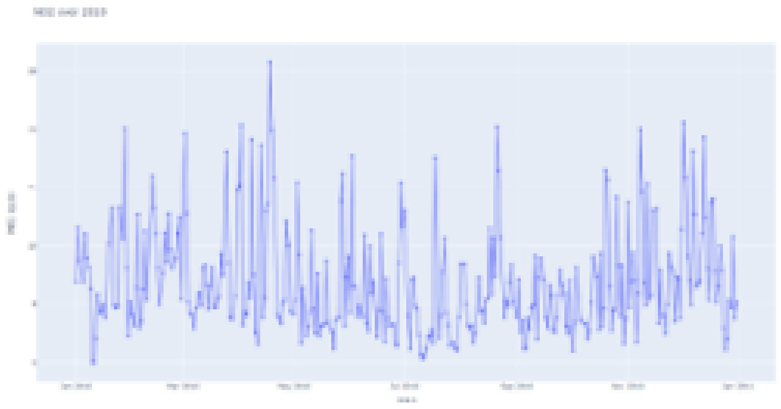
4. Perform Regression

Click this button to generate linear, quadratic, and exponential regressions on the current graph viewing and view them on plotly.

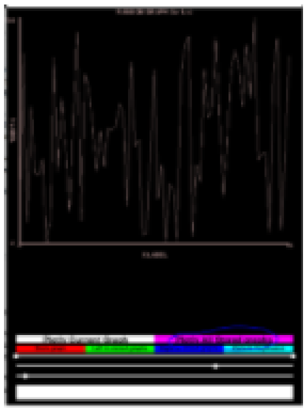
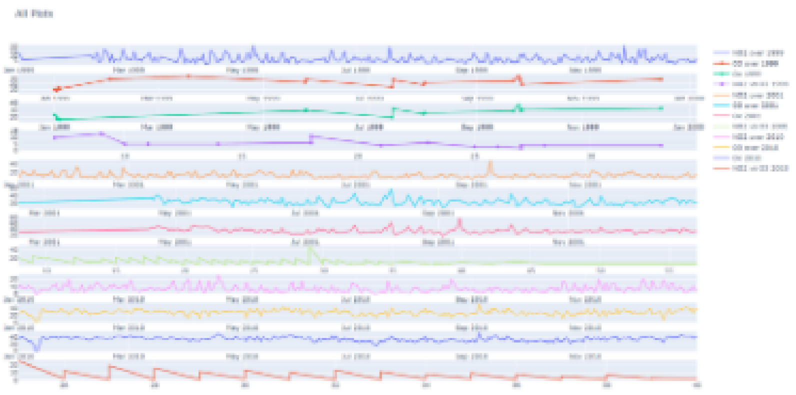
5. Plotly Current Graph

Click this button to plot the current graph viewing on plotly.

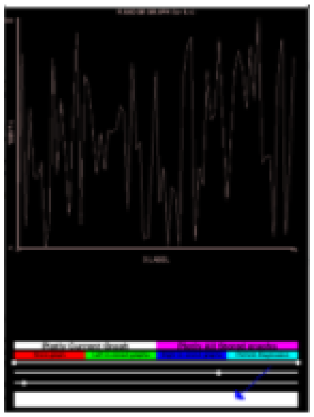
6. Plotly All Stored Graphs

Click this button to plot all the current graph viewings stored in inventory.

7. Add Random Graph

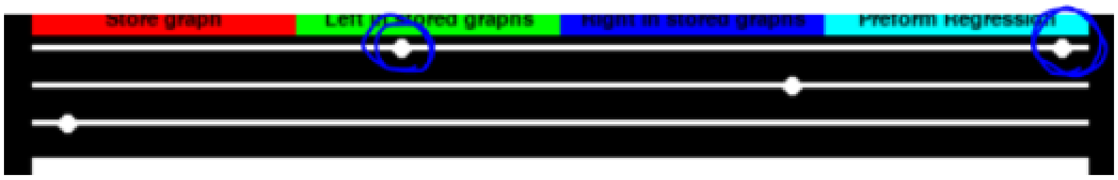
This is a fun button (hence it has no label) that generates arbitrary random graphs for you to view and practice using the software with.



Sliders

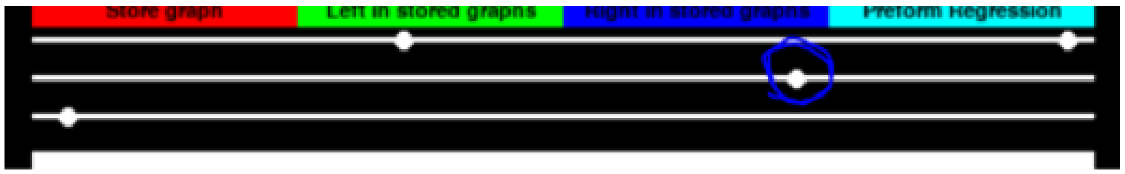
1. Slider #1 (Top)

These two sliders can be used to pan and zoom the graph on either side, allowing you to focus on any section of the graph. Additionally, for ease of the user, if your computer's mouse/trackpad sensitivity isn't too high you can also use these sliders by scrolling on you're device.



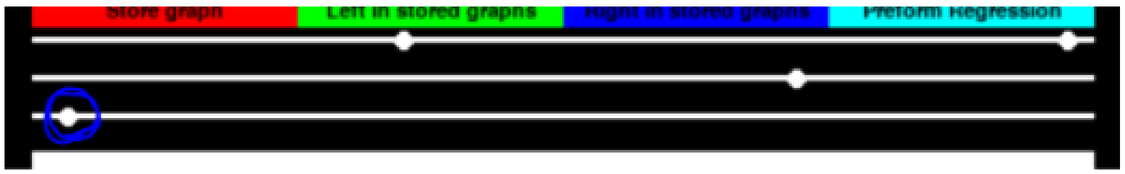
2. Slider #2 (Middle)

This slider vertically stretches the graph



3. Slider #3 (Bottom)

This slider horizontally stretches the graph



When using the pan slider or the scroll wheel, the user can restrict their view of the graph to a certain portion. When this graph is then plotted in any of the three plot buttons our software will helpfully plot the graph only restricted to the portion they decided on by panning / zooming.

Updates

To preface this section, we would like to address a comment given on our project proposal. The teaching assistant who gave feedback on our project warned us against using averages to compute more averages, as well as other data outputs, due to a lack of accuracy. We reviewed the URLs they provided in support of their claim, but found various scientific papers that revealed that countries such as China, India, and Sweden are employing similar methods as we are for Canada in terms of calculating the levels of odd oxygen in the air (Venkitasamy, 2015; Han, 2011; Hagenbjörk et al., 2017). So, we judged it was best to keep the original path we chose in using the averages from our data set in our calculations.

Originally, the team thought that the Python NumPy Library would be necessary to implement our project, but we discovered quite quickly that the Python Math Library had all the features we needed in terms of mathematical calculations. Due to unforeseen difficulty with reading and parsing our data, which was inconsistent in format, we also employed the Python CSV Module to parse the data to the format we desired. The original plan for parsing the data was to have a list of tuples, the first item representing the x coordinate and the second item representing the y coordinate. In the end, the format decided upon was a tuple of two lists, the first representing x coordinates and the second representing y coordinates. The pair-values of the x and y coordinates were still preserved though, as they corresponded to each other by index in their respective lists. Another library we used that we did not originally include in our project proposal was the Python Plotly Library. In terms of the user interface, we ended up adding many more features than previously anticipated to ameliorate the user interaction and capitalize on the usefulness of our data demonstrations. To do so, the Python Pygame Library was heavily used, as forecast by the project proposal, and employed to design a graphing software. This graphing software is discussed in further detail in the computational overview. With this software, users can look at any graph that has been previously generated, zoom in or out on any graph portion, select which portions of the graph to exclusively plot, graph multiple graphs at the same time, save graphs, and go through various graphs.

The last alteration made was in direct response to comments that our data representation and calculations did not properly answer our research question, but rather only tracked how odd oxygen changed over time. To fix this issue, we made sure that in our interface we plotted lines for every element involved, not just odd oxygen, and added extra features to the software such as being able to zoom in and out, as well as plotting regressions to best analyze the impacts of the emission acts passed against pollution. This feature helped our final project directly answer our research question.

Discussion

Overview

As previously mentioned, one of the main problems fueling climate change is the pollutants that are emitted from various sources that use fossil fuels to run. Among some of the most damaging pollutants are ozone (O_3) and nitrogen dioxide (NO_2) since they cause adverse effects on human health (Who, 2000). For example, ground level ozone can increase the intensity of solar radiation, and nitrogen dioxide is a leading cause of smog.

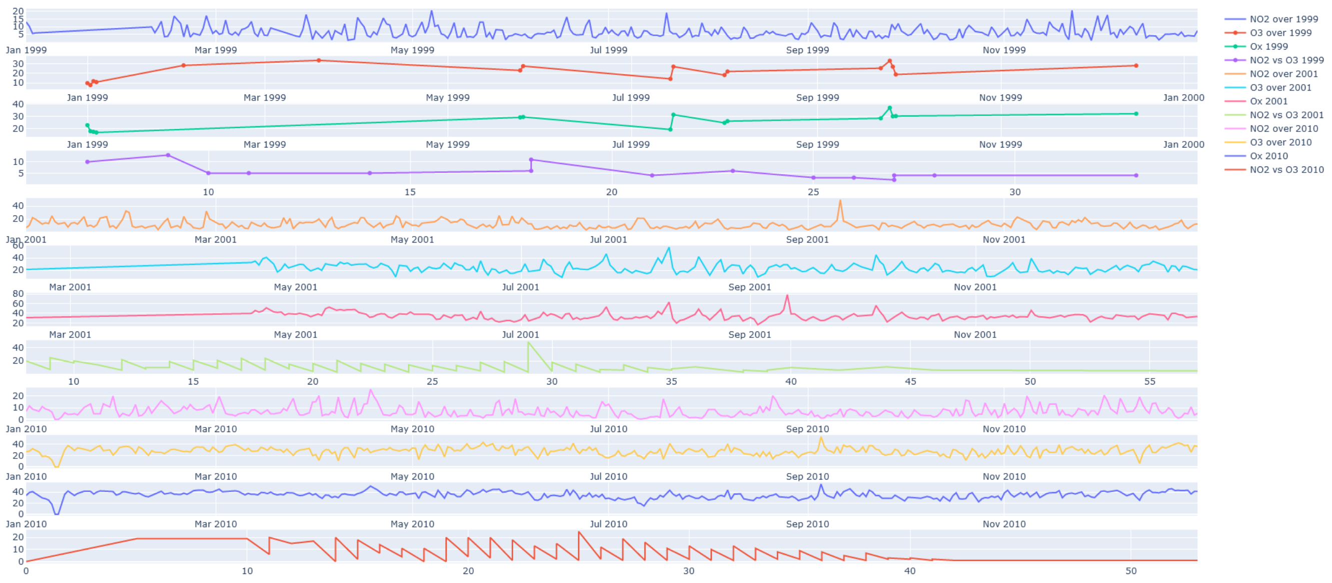
Within the recent years climate change has been called a “large and sprawling problem” (Who, 2000), but when there are so many ways to begin addressing it all efforts usually become watered down. Across the federal level, the Government of Canada and various policy makers have passed multiple regulations to advocate for “clean-energy policies.” Canada is among the top 20 countries for emission rates of ozone and nitrogen dioxide. Despite all the regulations passed and the United Nations Framework Convention on Climate Change, there has been no analysis done on the long term changes of these two secondary pollutants in Canada. Such analyses have been completed in China, UK, US and Sweden. There are multiple reasons for attempting to answer our research question.As described earlier the relationship of “Odd Oxygen” between NO_2 and O_3 is very important since in a regular situation any reduction of either pollutant results in the increase of the counterpart. It is therefore necessary to obtain an understanding of the relationship to attempt to assess if the regulations that target one of the emissions actually cause an increase in

Odd Oxygen - the opposite of what the policies should cause.

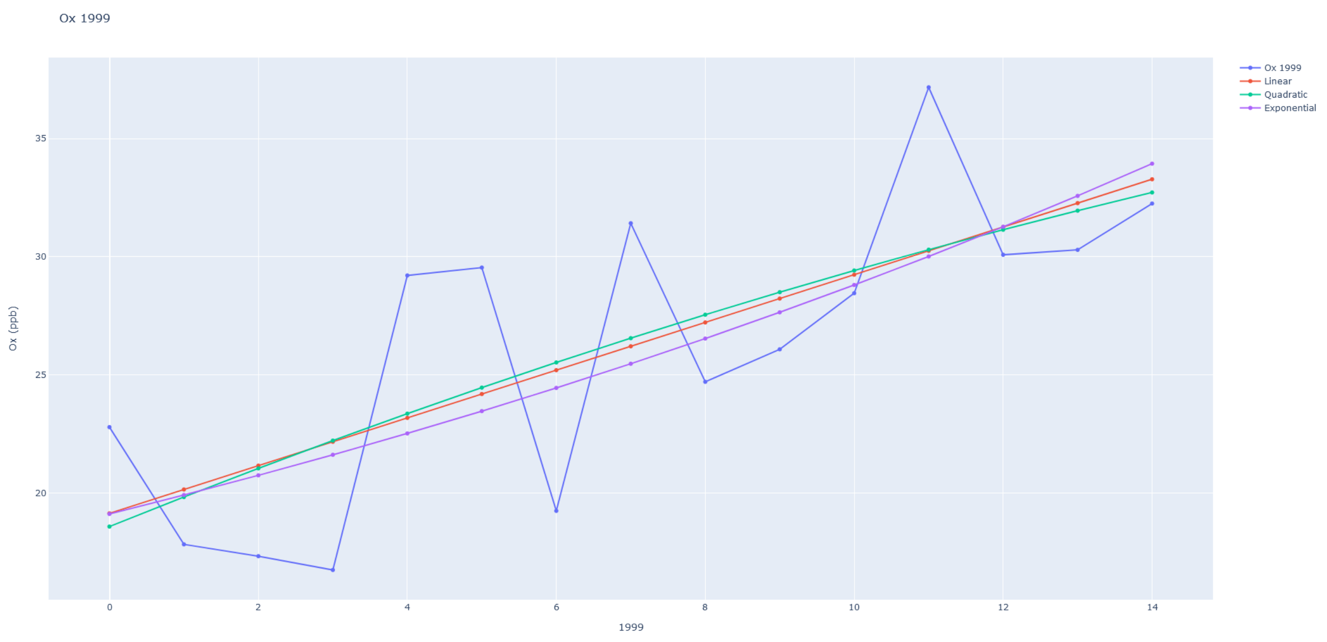
Computational Exploration Goals

The research question we were attempting to answer was how do the emissions acts passed within the past decades actually affect the concentration of Odd Oxygen in the air. The dataset that we used were from the years 1999, 2001 and 2010. Our computational exploration was to create a graphing system that uses the observed hourly variations of NO_2 and O_3 and the calculated hourly variation of Odd Oxygen. Through our plotting system we give users the ability to select a rough estimate of which time period they wish to view in more detail (using the first slider at the bottom) and then allow them to open the data using plotly.

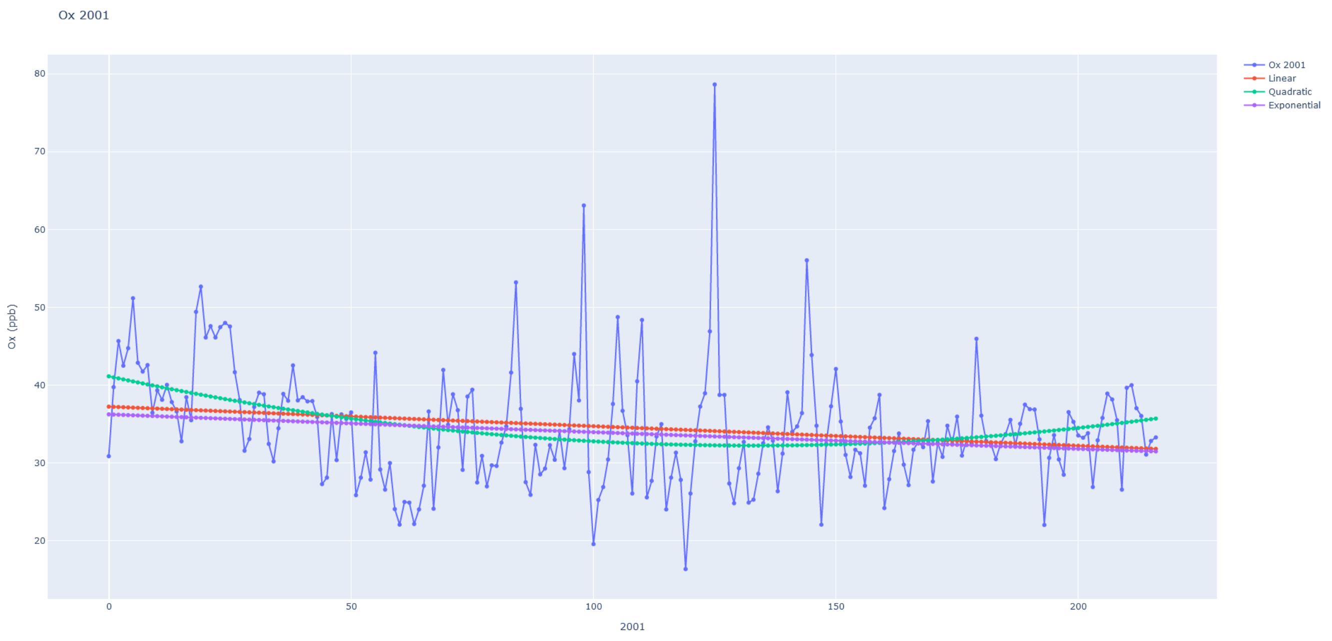
Before we address the effects of the emission acts we recognized the diurnal cycle of NO_2 and O_3 during the day and night. The largest value peaks of NO_2 usually occurred in the morning with O_3 peaking on average 5 hours later at noon. This further supports research done in other countries worldwide that a pattern in temporal variability of air pollutants regardless of latitude and longitude (Sanchez et al., 2007).



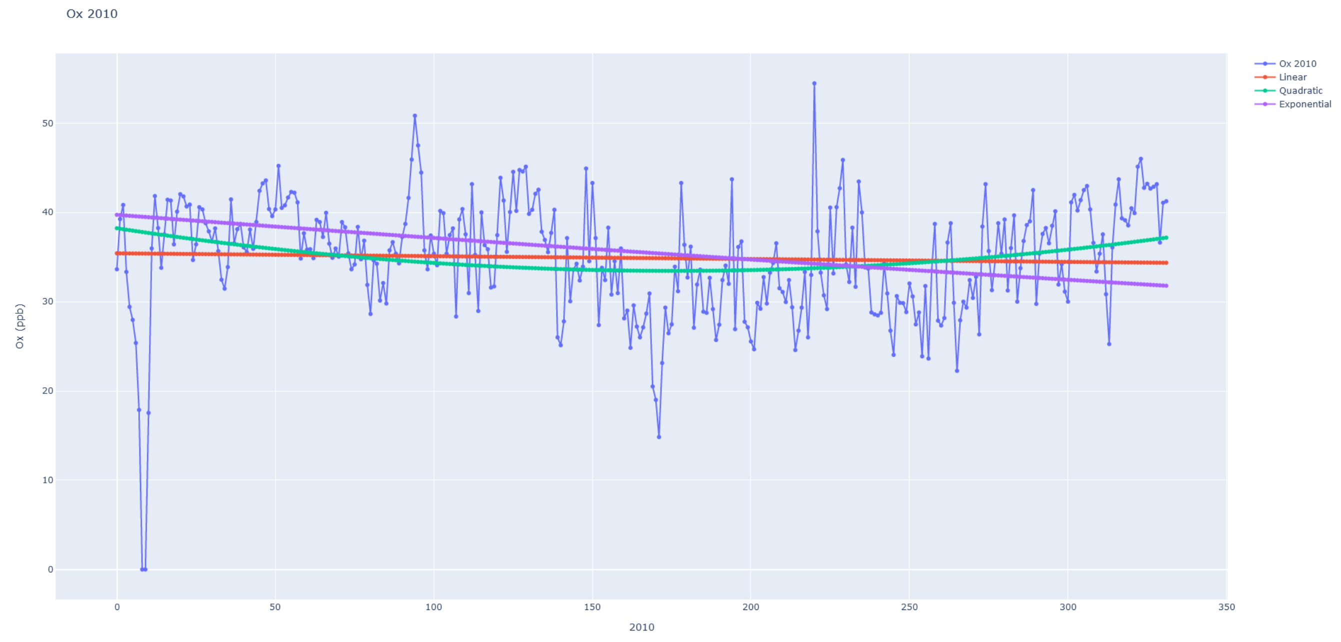
In the graphs shown, it is visible that by looking at the data of these three years that while the overall concentration of O_3 has decreased, the NO_2 levels increased in fluctuations. The level of [OX] was shown to be influenced by the NO_2 independent and dependent contributions and the O_3 dependent contributions. Zooming in on the graphs showed that the production of O_3 was higher on weekends and the levels of production of NO_2 production were higher on weekdays.



Above is the graph that shows the Odd Oxygen values in 1999, as well as three different regressions (linear, quadratic and polynomial to the nth degree). In March 1999, the Government of Canada passed the Canadian Environmental Protection Act. Our computational exploration showed that while the value of NO_2 did have a slight decrease of average daily values, the values of O_3 did experience an increase (data collected from NAPS was considerably smaller for the year 1999). Therefore the Odd Oxygen value did not decrease in response to the implementation of the Canadian Environmental Protection Act but instead increased. This shows that the regulation that was put in place had effects on air pollution in a detrimental way.



In February 2001 the Government passed the Federal Agenda on Cleaner Vehicles, Engines and Fuels and in June it passed Memorandum of Understanding (Canadian Vehicle Manufacturers Association). Unlike the previous regulation implemented in 1999, after the month of June, both NO_2 and O_3 experienced a decrease of emission, therefore overall lowering Odd Oxygen. As you can see above, there is an overall decrease which comes to show that this specific regulation that targeted vehicles, engines and fuels has a larger impact since it is one of the biggest source of emissions and secondary pollutants in the world (WHO, 2000).



In October 2010, the government passed the Passenger Automobile and Light Truck Greenhouse Gas Emission Regulations. As shown by the computational exploration we computed, this decreased the values of O_3, but the values of NO_2 slightly increased in response.

The analyses on the concentrations of NO_2 and O_3 measured across Canada showed that the diurnal cycles during the day and night for each pollutant created a rather steady value of Odd Oxygen which is more accurate to use when measuring the "level" of air pollutants. Another feature we implemented was the safety monitoring, which showed that at no point did any city in Canada go over the Ontario 1-hour Ambient Air Quality Criterion (OAAQC, 2018).

Limitations and Obstacles

There were a few limitations we encountered with the datasets we found. Despite being collected by the Government of Canada the way the data was stored was very inconsistent. An example is that some csv files had the date stored as YYYYMMDD while others had YYYY-MM-DD. Although this was a small change, unless we manually found this error when loading in data it would result in issues in the future.

Another issue with the data was the missing collection points. It was noted in the datasets that for values of -999 meant it was not properly recorded/collected. It would not make sense to include these values when calculating Odd Oxygen (the sum of the concentrations of NO_2 and O_3), therefore made us create another data cleaning function to remove the dates that did not have values in both pollutant datasets before calculating Odd Oxygen.

Although this was not exactly a limitation of Matrix (through Numpy) we realized that we only needed to use square matrices when calculating regressions of polynomials to an nth degree (where n is any natural number). Therefore to make our code more efficient we wrote our own Matrix ADT, and only implemented square matrices functions.

Another obstacle we faced was when creating our own graphing system. When usually programming this it would be completely fine to use static values however this proved wrong when we attempted to run it on our laptops and desktops since the view would not be ideal for the screen size. To overcome this we have our display possible for any value over 350 pixels allowing the user to dictate how large they wish the data to be displayed.

Further Exploration

Some next steps for further exploration would be to begin breaking down the regulations that are put in place that cause effects that lower overall air pollutant levels (Odd Oxygen) versus individual pollutant levels (NO_2 and O_3). Once you break these regulations down the goal would be to simulate how it could affect the individual pollutants. One feature we included is the button at the bottom of the screen that can produce random graphs that “simulate” the diurnal trends.

For further exploration, focusing on showing the standard deviation and relative standard deviation graphs for linear, quadratic and polynomial regressions would be a good visual way to share more on the relationship between NO_2 and O_3 for Odd Oxygen.

Goals

For the past half a century, while Canada has committed to monitoring the air quality and has proven to follow through on its promise to reduce the concentration of ground-level ozone and the levels of smog, the policies that get implemented don’t always achieve their goal. While none of the three regulations we focused on had changes that were very significant the relationship between NO_2 and O_3 should be taken into account by policymakers. If such analysis were applied by the Government of Canada we would be able to see more efficient implementation of policies that lower overall levels of pollution and begin to take steps to stop climate change.

References

Clapp, L. (2001). Analysis of the relationship between ambient levels of O3, NO2 and NO as a function of NOx in the UK. *Atmospheric Environment, 35*(36), 6391-6405. doi:10.1016/s1352-2310(01)00378-8

D. (2020). Canada. Retrieved December 13, 2020, from https://dieselnet.com/standards/ca/

Dhorde, A., & Wakhare, A. (2009). Detection and understanding of climatic response to urbanization over major cities of India. *IOP Conference Series: Earth and Environmental Science, 6*(9), 092013. doi:10.1088/1755-1307/6/9/092013

G. (2018, November 21). National Air Pollution Surveillance (NAPS) Program. Retrieved December 13, 2020, from http://data.ec.gc.ca/data/air/monitor/national-air-pollution-surveillance-naps-program/?lang=en

Government of Ontario, Ministry of the Environment. (2012). Access Environment. Retrieved December 13, 2020, from http://www.airqualityontario.com/science/pollutants/ozone.php

Government of Canada. (2020). JavaScript is required to view this site. Retrieved December 14, 2020, from https://www.ontario.ca/document/air-quality-ontario-2017-report/canadian-ambient-air-quality-standards

Government of Ontario. (2020). JavaScript is required to view this site. Retrieved December 14, 2020, from https://www.ontario.ca/page/ontarios-ambient-air-quality-criteria

Hagenbjörk, A., Malmqvist, E., Mattisson, K., Sommar, N. J., & Modig, L. (2017). The spatial variation of O3, NO, NO2 and NO x and the relation between them in two Swedish cities. *Environmental Monitoring and Assessment, 189*(4). doi:10.1007/s10661-017-5872-z

Han, S., Bian, H., Feng, Y., Liu, A., Li, X., Zeng, F., & Zhang, X. (2011). Analysis of the Relationship between O3, NO and NO2 in Tianjin, China. *Aerosol and Air Quality Research, 11*(2), 128-139. doi:10.4209/aaqr.2010.07.0055

Sanchez, M.L., Torre, B.D., Garcia, M.A. and Péreza, I. (2007). Ground-level Ozone and Ozone Vertical Profile Measurements Close to the Footfills of the Guadarrama Mountain Range (Spain). Atmos. Environ. 41: 1302–1314.

WHO (2000). Guidelines for Air Quality, World Health Organization, Geneva. p. 190.

Wood, J. (2012). *Canadian environmental indicators - air quality*. Vancouver, BC: Fraser Institute.