GEOM 4009 – Group Project Final Progress Report

Spacetime Continuum Visualization

Gillian Chapman - 101031958

Rajpal Dhaliwal- 100832829

Joshua Goutte-101013760

Jacob Wright - 101037975

Yussuf Yassine-101077732

Derek Mueller

GEOM 4009: Applications in GIS

April 14th, 2021

TABLE OF CONTENTS

Introduction:

Background on the client:

Master Corporal Michael Obersnel is a geomatics technician with the Canadian Forces Intelligence Command (CFINTCOM). His roles include the capture, analysis, processing, presentation, dissemination, and management of spatial data as support to the Canada national security objectives.

Some tasks for that intelligence command include intelligence collection which varies from weather information to the Armed Forces, or providing geospatial information and geomatics. Master Corporal Obersnel's role includes enabling intelligence assessment to provide analysis, strategic warning, and threat assessment.

Client requirements and priorities:

Master Corporal Obersnel provided us with positional data of tigers located in the country of India. Using this as a reference, he wanted us to take positional coordinates and timestamp values to create a 3D spacetime visualization map of geoposition over time. The data contains x,y positional data and a time argument. The priorities for the client were to take in a csv file with those arguments using user input while asking that same user to identify those 3 components on the csv. Next, we had to find an appropriate time span, which led us to vertically exaggerate the data to make proper conclusions. Furthermore, we had to create 3D lines and points using those positional and time arguments. Finally, we could export the data to a KML file so we can visualize the results in Google Earth.

Purpose:

The purpose of the project was to visualize the movement of different objects in space and time. These objects could vary from different tigers (such as our client's sample data), or ship routes in the arctic. This project would be versatile in the sense it could take any csv with positional data and time argument and we could then 3D plot it into a software such as Google Earth to be able to visualize paths that would be more difficult to do in a 2 dimensional space.

Scope:

In order to accomplish this, the project would ask the user to enter a directory where the csv file is located, this is the first step in getting what you want to get exported onto a KML. The user will then have to enter the name of the csv with the .csv extension so the program can read the file. Once the file is read, the user has to input the rows that contain the object id, the x position of the object, the y position of the object and the timestamp.

The program will then assign predefined names to the user identified columns, allowing it to go to the next step which is the simplification algorithm. The data now simplified, we can go about determining an adequate time span to be able to go to our next step which is vertical exaggeration. Vertical exaggeration is important for the data to not be too compact when visualized in the 3 dimensions.

Next step is the function that creates the 3 dimension lines, we use the object id of each and we separate them all into separate lines. Next, distance is calculated between two objects at every shared timestamp as well as some statistics are generated.There is also an option to create a distance matrix. Finally, we have the full product so we can export it to the KML file, to be able to properly visualize on software such as Google Earth.

Workflow:

The following bullet point list is how our code works in order.

1. Ask the user for the directory, file name, object id position in the csv, latitude and longitude position in the csv, and the time position in the csv.

2. Runs the dataset through the first function to rename it.

3. Runs the df through the simplification algorithm to declutter it.

4. Runs the simplified data frame through a timespan function to determine the timespan of the dataset.

5. Run the dataset through the vertical exaggeration function to calculate the vertical exaggeration.

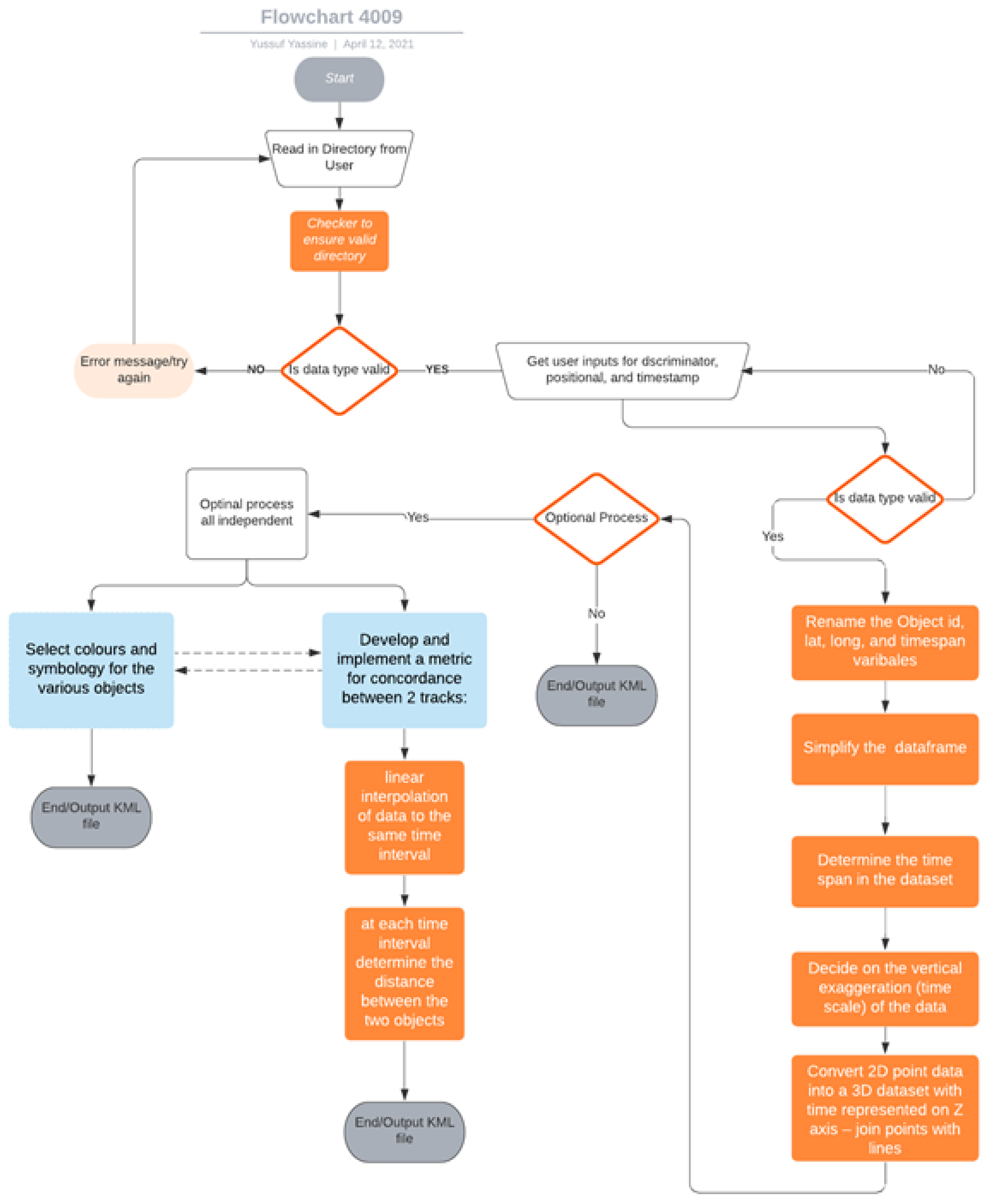
6. Use the convert 2d points to 3d points function.

7. Distance between the objects is calculated using the distance function (optional function)

8. Calculate the distance matrix for specified dates (optional function).

9. Export the final KML file to the user directory.

The flowchart below is a comprehensive overview of how our code works and what steps are taken throughout the script.


Flowchart 4009
Yussuf Yassine | April 12, 2021
Start
Read in Directory from User
Checker to ensure valid directory
Is data type valid
NO
Error message/try again
YES
Get user inputs for discriminator, positional, and timestamp
Is data type valid
No
Yes
Rename the Object id, lat, long, and timespan varibales
Simplify the dataframe
Determine the time span in the dataset
Decide on the vertical exaggeration (time scale) of the data
Convert 2D point data into a 3D dataset with time represented on Z axis – join points with lines
Optional Process
No
End/Output KML file
Yes
Optinal process all independent
Select colours and symbology for the various objects
End/Output KML file
Develop and implement a metric for concordance between 2 tracks:
linear interpolation of data to the same time interval
at each time interval determine the distance between the two objects
End/Output KML file

Documentation:

Dependencies:

Similar to most applications, there are software dependencies that are needed. The appropriate libraries and packages need to be downloaded before proceeding with the application. This script was tested on a Windows 64-bit system and we do not recommend running it on a system that does not have at least 8GB of RAM (4GB could work but might be very slow and affect system performance). The script was tested with Python 3.8.3, and it is the version we recommend running this script with. Other python versions might work, but we cannot confirm the capability of the script with other versions.

Anaconda can be used to download all packages and libraries that were used in this script. In general, to import a package you need to write the following in the anaconda window “conda install package-name”. You can find a quick guide on how to install packages using Anaconda in their documentation. Here is a list of the packages that need to be installed. ‘pip install’ can also work to install packages. The documentation for that is here.

Follow the hyperlinks on each of the libraries for more information.*Python os module.*

1. *Pandas library.*
2. *geoPandas library.*
3. *Datetime module.*
4. *NumPy package.*
5. *Fiona library.*
6. *Time module.*
7. *Shapely package.*
8. *Sklearn package.*

Installation/setup:

Before running the script, all the packages must be installed properly on the system. Without all the packages installed the script will not run and will give output errors. Take the time to read to anaconda documentation to ensure that all the packages get installed correctly. Before running the script, we recommend creating a new folder in an easy-to-access location, and put the file that the script will be working on in that folder. Knowing the location of this folder is important and in most cases, right-clicking on the folder and looking at its properties will give you the location of it.

It is important to note that python does not work with backslashes, therefore all backslashes need to be replaced with front slashes. For example, if the folder that you wanted to set up as your directory has the following path “home\user\folder”, it would need to change to “home/user/folder” when inputting the directory in the script. This directory will be the location that the final KML file will be saved in and therefore is important to keep track of its location.

User guide:

set_directory: This function takes a directory path as the argument and sets the directory to this location, using os.chdir() (Figure A1).

read_file: This function takes the name of the csv file as an argument, as well as the numerical position of the object ID column, x positional data column, y positional data column, and the timestamp values column. It reads in the csv file from the directory, and renames each of these columns “object_id”, “x_pos_data”, “y_pos_data”, and “timestamp”, respectively. The columns are renamed to allow for consistent naming throughout the program. It allows other functions to easily call on the columns of interest by the column names. The function returns a pandas dataframe with the columns of interest renamed (Figure A2).

firstLast: This function takes the first timestamp and the last timestamp from the renamed column timestamp to determine the timespan difference of the dataset. First, the user has two options to decide on when calculating the timespan difference, depending on how their timespan is present in their csv file. Currently, the function is set to take the timestamp rows that are formatted as DD/MM/YYYY:H-M in the panda dataframe that was created in the read file, and convert them using the class library to a datetime class format DD/MM/YYYY:H-M-S.

The second option that the user has when reading the timestamp data from their dataframe is more generalized, meaning that it can read a variety of different time formats. This second option can be useful for regions that set their timestamps as MM/DD/YYYY. This allows users to have flexibility with the date formats being entered into the program. But from the sample data provided, this code was made using the first option as the pandas dataframe could only be read as the datetime DD/MM/YYYY:H-M. Please refer to the troubleshooting/FAQ section to learn more about how to switch between and select each timespan code.

Next, the firstLast function sorts the timedate dataset, and is assigned to the variable name *time*. The start time and the end time of the dataset are assigned to their own variable, and are captured by using iloc, which enables us to select a particular cell (the start time, and the end time) of the datetime dataset. Finally, the difference of the timespan is calculated by subtracting the endtime variable from the startime variable (Figure A3).

simple: This function works by taking the datasets and splitting it by the different object id’s to create a number of the smaller dataset. These smaller datasets are then stripped to only contain the latitude and longitude data. Next, the Ramer–Douglas–Peucker algorithm simplifies the dataset by removing the redundant points. The algorithm runs through the original datasets and retrieves the missing metadata that was originally present to recombine the simplified points with their metadata. Each smaller simplified dataset is added together to create the final simplified version. This final version gets returned to the main function where more manipulation can take place on the final dataset (Figure A4).

haversine: This function takes two latitude values and two longitude values and calculates the distance between them. This distance formula uses great circle distances, which take into the consideration the curvature of the earth. This distance formula is most appropriate when calculation distances between geolocation. This is a helper function that is referenced in the zScale and distance_bw_2objs functions (Figure A5)

zScale function: This function takes in the dataframe as an argument. A range of elevation values are created from 500 m to 10,200 m. The numpy linspace method is used to create the elevation array with equal intervals. The size of the intervals depend on the size of the dataframe’s time column. The elevation array will be the same size as each data frame column. The interval will be the difference between the maximum and minimum elevation divided by the total number measurements there are in the dataframe. If the user wants a different set of elevation, they change the zmin and zmax values inside the zScale function. These 2 variables are used inside the numpy linspace method. The 3rd parameter that controls the size should remain constant. The function returns a numpy array that contains the elevation that is used as one argument for the PointsLine function (Figure A6).

PointsLine function: The PointsLine function takes an object ID, the name of the object; an x and y positional argument, a location in x and y coordinates where the objects are located; and a vertically exaggerated time argument. This is necessary so that the line is not compacted together. The function starts by ensuring it contains the proper projection, WGS84, which is used for visualization in an exterior software, such as Google Earth. Contains a variable taking in all the unique objects from your dataset. We define two empty lists, which are used to contain the results of an iteration to create an exportable dataset. The iteration will go through each object in the unique objects variable created beforehand, and it will make a variable by creating a separate data frame for every different object, which the function will iterate through. Using the shapely library and the LineString function, the function zips each object's x,y, time components, the inputs of our function. Lists are created from appending the index values into our index_list and the geometry values in our geometry_list. Outside of the iteration, a GeoDataFrame is created taking an index, projection and geometry. Index values and the geometry values are from our iteration in the form of index_list and geometry_list. The final product of the function is a GeoDataFrame which is fully exportable, which is used for visualization later(Figure A7).

KMLExport function: This function takes as input the GeoDataFrame containing the lines created in the PointsLine function. Using the Fiona library, a library that is used to export variables into multiple formats, adds KML as a supported driver. An assigned variable takes the input file(our GeoData Frame), exports it, and gives it a name and assigns it a KML driver. The function's capabilities enables the project to be visualized using KML supported programs such as Google Earth (Figure A8).

Distance_bw_2objs: This function takes a dataframe and two distinct object IDs as the arguments. The data frame is filtered for all observations related to the two object ID arguments. The filtered data frame is then filtered again for timestamps that occur exactly twice. This implies that the timestamp occurs once for each object, and all unique timestamps (i.e. timestamps that occur for one object and not the other) are removed. This ensures that the distances are only being calculated when the objects share the exact same timestamp value. It then splits the two objects into two different data frames. For each dataframe, the latitude and longitude is turned into a numpy array to ease looping. From there, the latitude and longitude arrays for each dataframe are looped through and inputted into a haversine distance formula. This returns the distance in kilometers at each shared timestamp value (Figure A9).

Distance_Matrix: This function takes the dataframe and a single timestamp value as arguments. It filters the dataframe for all objects at that specific timestamp value (this assumes that each object will only have one measurement for each timestamp). Once it finds all the objects at that point in time, it performs a pairwise distance calculation between the latitude and longitude for each object. The distance formula used for this is haversine distance. It then returns a matrix displaying the distance between all objects at that point in time (Figure A10).

Troubleshooting/FAQ:

i) Resolving issues with the directory

The first step in this program is to identify the directory path. A prompt for the directory path will appear in the console. Once the directory path has been provided, the path is then passed as an argument into the “set directory” function (Figure B1).

The set directory function will automatically set the directory to the location provided. It will provide the warning “something is wrong with the directory inputted” if the directory is typed incorrectly or does not exist. If this happens, double check your directory location and re-type the initial directory path. Directories often use / instead of \.

ii) Resolving issues with the filename and column values

In the read_file function, if the file name does not exist or is typed incorrectly, it will throw the error “filename could not be found”. It is up to the user to ensure they have identified the correct columns for the object ID value, timestamp data, x positional data and y positional data. If the column number identified doesn’t exist, this function will not work. If incorrect column numbers have been entered, this function will run, but errors will occur in later functions (e.g. if the wrong column is set to timestamp, the date cannot be parsed correctly in the firstLast timespan function) (Figure B2).

iii) Resolving issues with the parsing of dates using the firstLast function

Depending on the timestamp format that the data is in, there are two options for reading in the timestamp to calculate the dataset timespan. Figure B3 is designed to read the sample data that was provided by our client. In the .csv sample data, the date and time format is presented as: DD-MM-YYYY. Figure 3 takes the timestamp from the .csv file and translates it into datetime format. What happens if this does not work?

Figure B4 shows the alternative time format that can be used, which is found one line below the line presented in figure 3 in the coding package. It is commented out by a ‘#’. This line is more generalized, and is made to have a wider acceptable range of dates and times that might be present in different .csv data, which could be dependent on the region you or the data are from. If the first datetime format in figure 3 does not work, please do the following:

1. Remove the # from figure B4
2. Add a # in front of the *df* in figure B3
3. Re-run the function

iv) Resolving issues with the distance_bw_2objs function

The distance_bw_2objs function can be used optionally to obtain information about the distance between two objects, along with the mean, standard deviation, and coefficient of variation. If the output of the function is not assigned to four different variables, it will store all of the information together as a tuple. The first variable in this figure (dist) is a dataframe, whereas the other three are stand-alone float values.

Ensure the object ID codes are inputted properly and everything is spelled correctly. If the data is a string type within the dataframe, ensure there are “ “ around the argument. If object codes are numerical, ensure they are written without " " surrounding them.

If there are no timestamps in common between the two objects, it will throw the error message "There are no shared time values between the two objects" (Figure B5).

v) Resolving issues with the distance matrix function

The distance matrix function takes the data frame and a unique timestamp value and calculates the distance between all objects at that specific point in time.

If the timestamp does not exist, it will throw the error "There are no objects at this timestamp value". Users must ensure the timestamp they pass as an argument is a properly formatted string. For example, passing a date as "2019/07/12" when the data frame stores the date as "2019-07-12" will result in the date value being unrecognized (Figure B6).

vi) Optional: Adding colours to the kml file

In order to change the line string colours, please open the KML file in a text edit software. Underneath each unique object identifier, and above the absolute altitudeMode line string, please copy and paste the following into your KML file:
<Style><LineStyle><color>ff0000ff</color></LineStyle><PolyStyle><fill>0</fill></PolyStyle></Style>

The highlighted *ff000ff* represents the colour of choice. The user can change that to suit their needs. Please refer to http://www.zonums.com/gmaps/kml_color/ to get different KML colour codes (2012). For simplicity but less colour choice, the user may also change colours in Google Earth (Figure B7).

Discussion

Challenges:

There was initially an issue with the Ramer–Douglas–Peucker simplification algorithm. This algorithm simplifies redundant points (e.g. if two measurements are very similar in time and position) to downsize large datasets. Originally, the simplification function was taking the entire dataframe at once, and simplifying the whole dataset together. The issue was, if two different objects were near each other they would be simplified together. The way around this was to separate each object out of the original dataframe, simplify it, and then re-combine all of the objects together in a new simplified dataframe.

Another issue that was overcome was the separation of the linestrings for each unique object. Initially, all of the linestrings were being outputted as a single line file which did not make sense when being interpreted in Google Earth. We overcame this by first identifying all unique object values, and creating a for loop that subsetted the dataframe by each of these unique values. The loop then created a linestring for each unique object, stored each linestring in an empty list, and then used the list of linestrings as the geometry when converting the linestrings into a geopandas dataframe.

Limitations:

When reading in the timestamp values, the values were parsed to the format of the sample data provided to our group (e.g. d/m/Y H:M). However, not every date that is read in will match that format, and parsing it like this will throw an error. A way around that was to infer the data format when creating a pandas datetime object (e.g. pd.to_datetime(df['timestamp'], infer_datetime_format=True). This allows users to have flexibility with the date formats being entered into the program. This is because the date format and how to parse it is interpreted on the fly. However, given that some countries write their date as d/m/y and others as m/d/y, the day and month are sometimes read in backwards. The issue has been noted in the user manual and explains how the user can work around this.

There were issues setting the maximum and minimum values for the Z-scale function. According to online documentation, time values needed to be converted to a float data type. For there, the maximum and minimum timestamp values are determined, and then used in a calculation to perform appropriate scaling. However, the float values derived from this method did not make sense since the derived float numbers would not represent elevation very well. Also the range of the float times was too small and it was difficult to see elevation differences. Within the Z-scale function, the minimum was set to 500 m and the maximum was set to 10200 m manually within the function. Usually can manually toggle this depending on how exaggerated they would like the z-scale. Further details are provided in the user manual.

Our group tried to implement colours for each individual linestring created, but could not get it implemented. When researching how to implement colour online, a large portion of information discussed colouring lines within a matplotlib plot. This did not make sense for exporting the kml file with colours. It was later that our group discovered that once the kml file was produced, that we should approach it by opening the file and replacing the colour codes between the <color> tags. There was discussion on how to implement this in an automated way, with the main conclusion that there should be a list with roughly 100 colour values (this would mean that the user dataset could not have more than 100 objects). The KML files could be opened, and then using a text search the value between the colour tags could be replaced by a value in the colour list. This would be done by looping through the colour list. However, due to time constraints this did not end up being implemented.

There is a function to determine the distance between two objects, where time values are shared. This function works well in the sample dataset provided, as several timestamps between different objects are identical. However, without the linear interpolation of time function completed, this function will not work well in all datasets. For example, if there were two objects and one had all the measurements on odd numbered days, and one object with measurements on even number days, the distance calculation would never work from them. This is because the times do not line up, and there is no interpolation to the same time values for these two objects.

Future work:

Creating a function that determines where two objects overlap, and then linearly interpolating the time during the period of overlap is an important next step. This would allow for distance to be determined between two objects during a period of time where two objects overlap.

Having a graphical user interface (GUI) to improve user experience. Right now the whole program is text based, and the user has to manually type in the column number containing key data instead of being able to click and select the columns of interest using a GUI. Having this would make it more interactive, and reduce errors related to typing in the wrong column number for where the values of interest are located.

Creating a function to automatically change colour values for the linestrings in the KML file would also be good to add in the future. It would make the outputted KML file more visually appealing overall. As mentioned in the limitations section, the idea for how to implement was brainstormed, but not brought to completion.

Converting from military grid reference system coordinates to WGS84 coordinates would also be a helpful function to add. Right now, all coordinates inputted into the file must be in the WGS84 format, though the client mentioned that military grid system coordinates are sometimes used and it would be helpful to convert.

Client interactions:

We communicated with the client twice throughout the lifespan of this project. The first meeting happened in February where we got to meet with our client, Master Corporal Michael Obersnel. After brief introductions and a demonstration of some of the work MCpl Obersnel does with Canadian Forces Intelligence Command, the group got to hear his vision of what he wanted to see in the final product.

Our second meeting happened in the middle of March. During this meeting, we got to show MCpl Obersnel our pseudocode and we discussed the progress of our project. We also got to ask him some questions and got clarifications on some of the wish list functions.

Although we only met with our client twice, the conversations that were held during these meetings were beneficial, and they further helped us shape our final product to the project’s goal.

Conclusion:

Our client MCpl Michael Obersnel is a geomatics technician with the Canadian Forces Intelligence Command. The objective of this project was to create a spacetime continuum visualization map. The spacetime map would represent 3D space of geographic position over time. MCpl Obersnel provided our group with sample data of tiger positional data taken in India. Our group used this sample data to formulate our code to export a spacetime map as a KML.

As a team, we completed the code's high priorities which were: created a function that specified object identifiers, positional data, and the timestamp. We also implemented a simplified algorithm, determined the time span, picked a vertical exaggeration, converted the data into a 3D dataset, and exported the lingstrings to a KML file. The team was also able to meet some lower priority wish list requests such as finding the distance between two specified unique identifiers and output the mean, standard deviation and covariance. We also created a distance matrix, where the distance between every object at a particular point of time was calculated.

Although we ran into some bumps on the way, we were still able to complete a code for MCpl Obsernel. With that said, we noted that with more time some of the wishlist that MCpl Obsernel mentioned to us such as; having an interactive GUI which would allow for better user interaction and for having flexible coordinate systems, that our code could use some future work. Furthermore, we hope that this code can further be developed and used to create a wide range of spacetime continuum visualization maps.

Acknowledgements:

As a group, we want to give a big thanks to Master Corporal Obersnel for giving us the opportunity to work on this project. We would also like to thank Derek, for the guidance that you have provided for us throughout this entire project. Also a big shout out to the team for having continuous open communication through discord. Everyone always showed up to meetings, brought ideas and fully participated. Thank you to everyone for a great term.

References:

Creating a GeoDataFrame with coordinates. 2021. *GeoPandas.* Retrieved from: https://geopandas.org/gallery/create_geopandas_from_pandas.html#sphx-glr-gallery-create-geopandas-from-pandas-py

Datetime - Basic Date and Types. 2020. *Python.* Retrieved from: https://docs.python.org/2/library/datetime.html

How to calculate distance in Python and PANDAS using Scipy spatial and distance functions. (2019, December 27). Retrieved April 14, 2021, from https://kanoki.org/2019/12/27/how-to-calculate-distance-in-python-and-pandas-using-scipy-spatial-and-distance-functions/

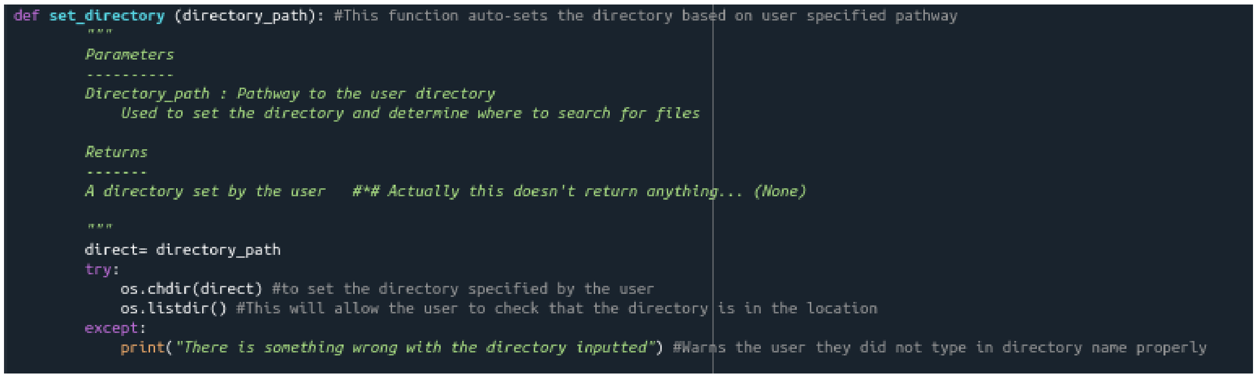
KML Colour. 2012. *Zonum Solutions.* Retrieved from: http://www.zonums.com/gmaps/kml_color/

Tiger Data. 2019. *ZoaTrack.* Retrieved from:' ZoaTrack.org

Digital Appendix:

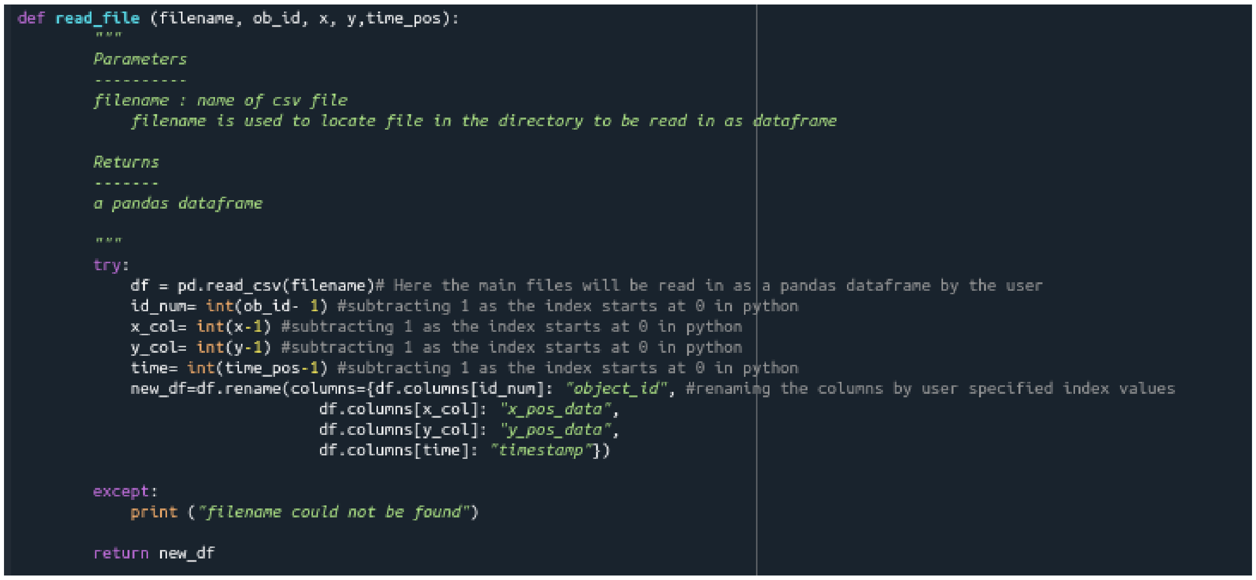
Appendix for the project can be found here: Digital Appendix.

Appendix A: User Guide Functions



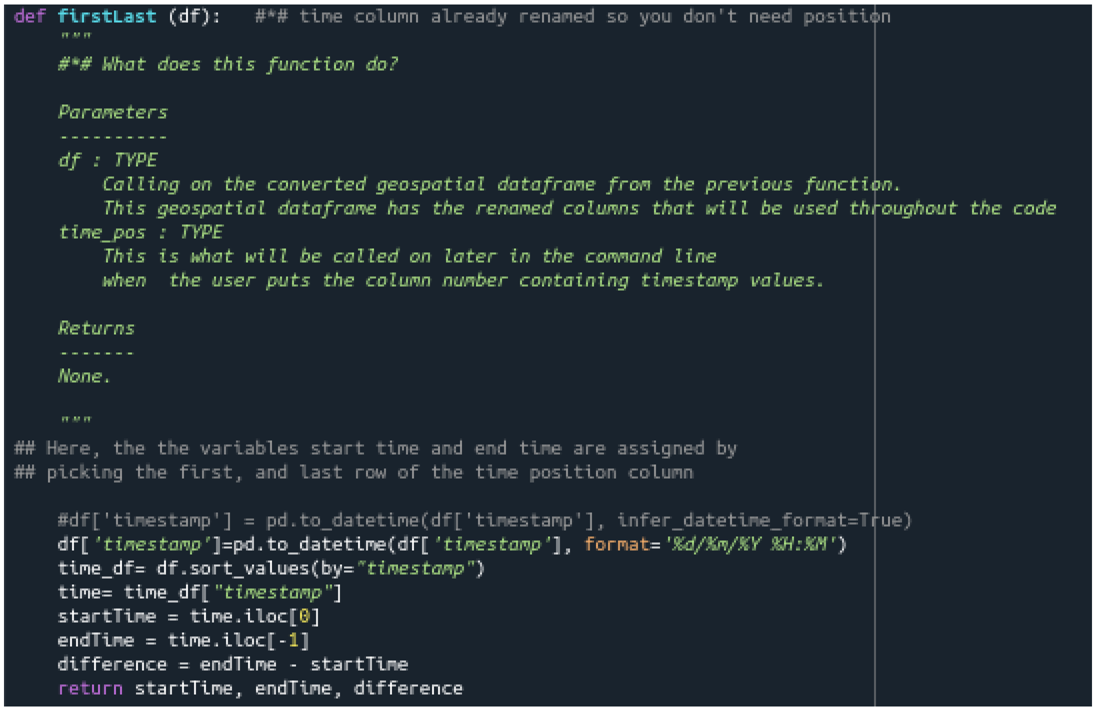
```python
def set_directory (directory_path): #This function auto-sets the directory based on user specified pathway
        """
        Parameters
        ----------
        Directory_path : Pathway to the user directory
            Used to set the directory and determine where to search for files

        Returns
        -------
        A directory set by the user   #*# Actually this doesn't return anything... (None)

        """
        direct= directory_path
        try:
            os.chdir(direct) #to set the directory specified by the user
            os.listdir() #This will allow the user to check that the directory is in the location
        except:
            print("There is something wrong with the directory inputted") #Warns the user they did not type in directory name properly
```

Figure A1: The set_directory function



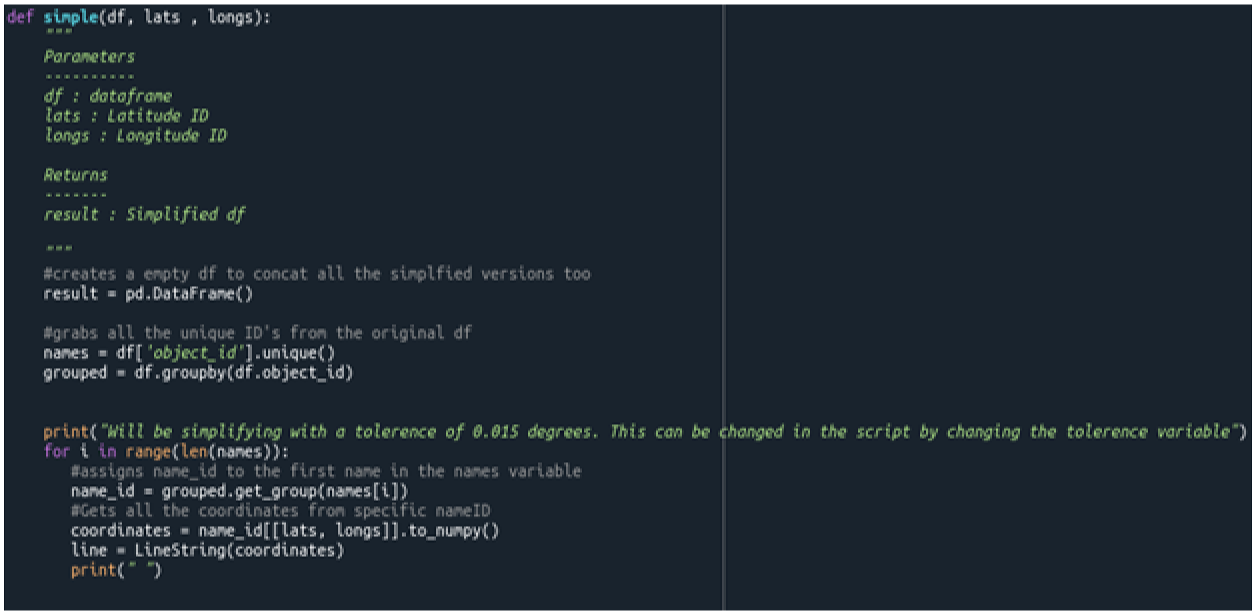
```python
def read_file (filename, ob_id, x, y,time_pos):
        """
        Parameters
        ----------
        filename : name of csv file
            filename is used to locate file in the directory to be read in as dataframe

        Returns
        -------
        a pandas dataframe

        """
        try:
            df = pd.read_csv(filename)# Here the main files will be read in as a pandas dataframe by the user
            id_num= int(ob_id- 1) #subtracting 1 as the index starts at 0 in python
            x_col= int(x-1) #subtracting 1 as the index starts at 0 in python
            y_col= int(y-1) #subtracting 1 as the index starts at 0 in python
            time= int(time_pos-1) #subtracting 1 as the index starts at 0 in python
            new_df=df.rename(columns={df.columns[id_num]: "object_id", #renaming the columns by user specified index values
                                  df.columns[x_col]: "x_pos_data",
                                  df.columns[y_col]: "y_pos_data",
                                  df.columns[time]: "timestamp"})

        except:
            print ("filename could not be found")

        return new_df
```

Figure A2: The read_file function

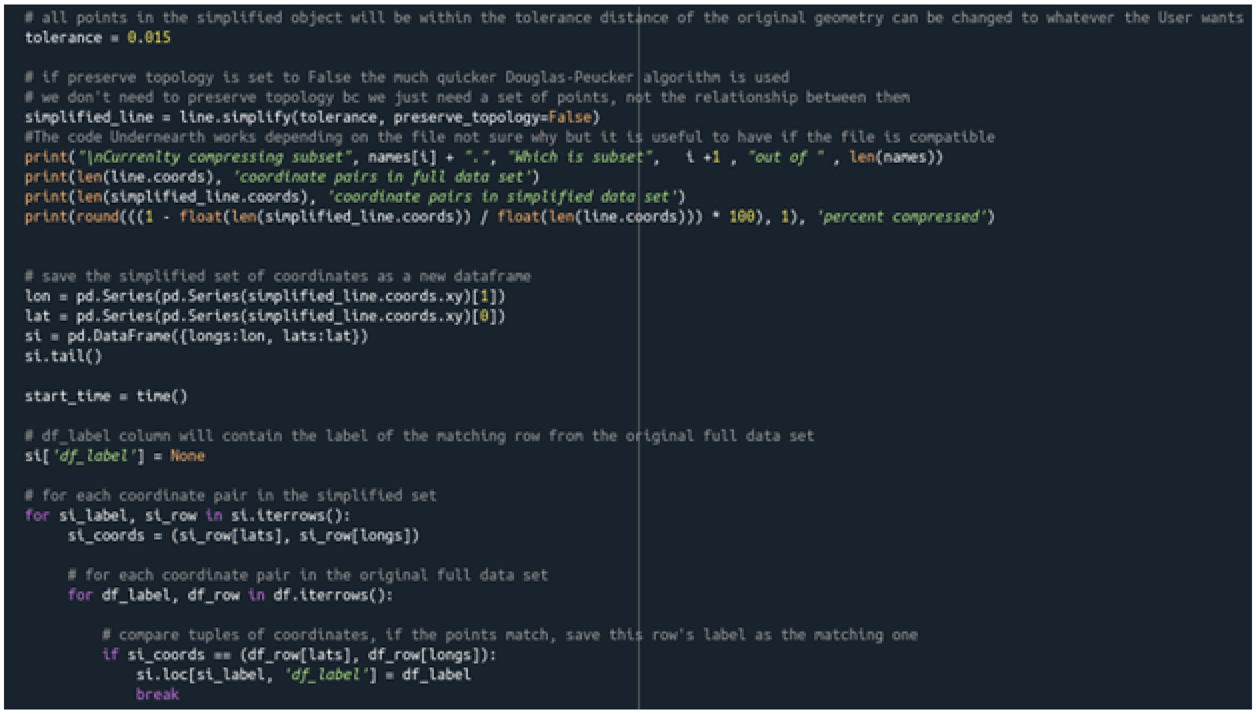


```python
def firstLast (df):   #*# time column already renamed so you don't need position
    """
    #*# What does this function do?

    Parameters
    ----------
    df : TYPE
        Calling on the converted geospatial dataframe from the previous function.
        This geospatial dataframe has the renamed columns that will be used throughout the code
    time_pos : TYPE
        This is what will be called on later in the command line
        when  the user puts the column number containing timestamp values.

    Returns
    -------
    None.

    """
## Here, the the variables start time and end time are assigned by
## picking the first, and last row of the time position column

    #df['timestamp'] = pd.to_datetime(df['timestamp'], infer_datetime_format=True)
    df['timestamp']=pd.to_datetime(df['timestamp'], format='%d/%m/%Y %H:%M')
    time_df= df.sort_values(by="timestamp")
    time= time_df["timestamp"]
    startTime = time.iloc[0]
    endTime = time.iloc[-1]
    difference = endTime - startTime
    return startTime, endTime, difference
```

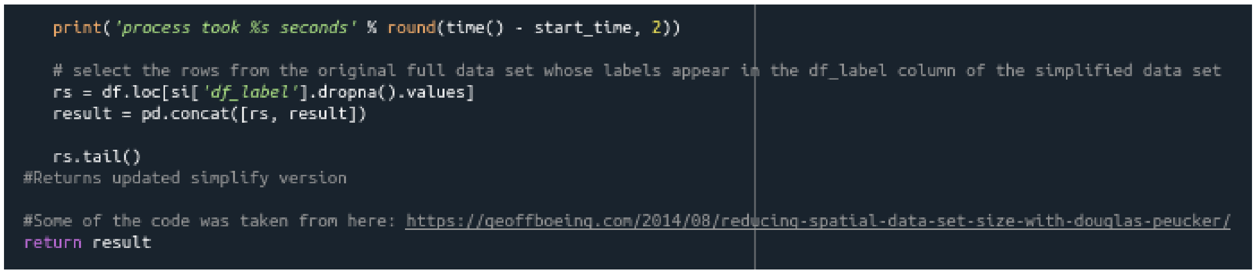
Figure A3: The firstLast Function



```python
def simple(df, lats , longs):
    """
    Parameters
    ----------
    df : dataframe
    lats : Latitude ID
    longs : Longitude ID

    Returns
    -------
    result : Simplified df
    """
    #creates a empty df to concat all the simplfied versions too
    result = pd.DataFrame()

    #grabs all the unique ID's from the original df
    names = df['object_id'].unique()
    grouped = df.groupby(df.object_id)


    print("Will be simplifying with a tolerence of 0.015 degrees. This can be changed in the script by changing the tolerence variable")
    for i in range(len(names)):
       #assigns name_id to the first name in the names variable
       name_id = grouped.get_group(names[i])
       #Gets all the coordinates from specific nameID
       coordinates = name_id[[lats, longs]].to_numpy()
       line = LineString(coordinates)
       print(" ")
```

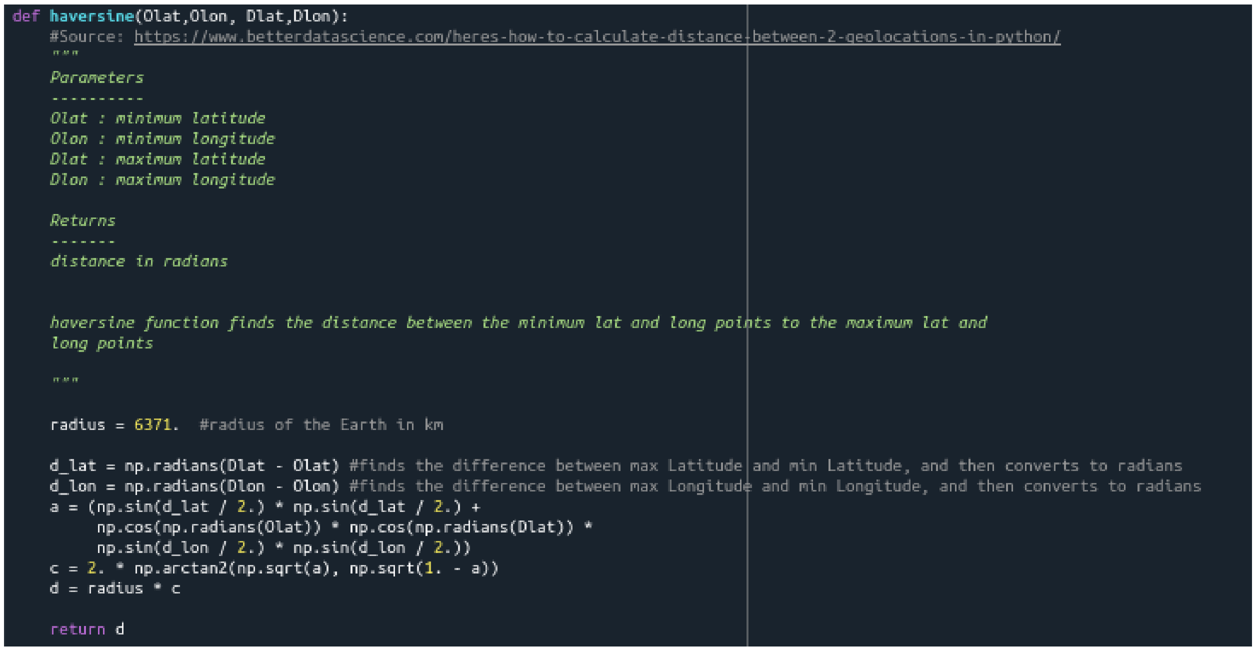


```python
 # all points in the simplified object will be within the tolerance distance of the original geometry can be changed to whatever the User wants
 tolerance = 0.015

 # if preserve topology is set to False the much quicker Douglas-Peucker algorithm is used
 # we don't need to preserve topology bc we just need a set of points, not the relationship between then
 simplified_line = line.simplify(tolerance, preserve_topology=False)
 #The code Undernearth works depending on the file not sure why but it is useful to have if the file is compatible
 print("|nCurrenlty compressing subset", names[i] + ".", "Which is subset",   i +1 , "out of " , len(names))
 print(len(line.coords), 'coordinate pairs in full data set')
 print(len(simplified_line.coords), 'coordinate pairs in simplified data set')
 print(round(((1 - float(len(simplified_line.coords)) / float(len(line.coords))) * 100), 1), 'percent compressed')


 # save the simplified set of coordinates as a new dataframe
 lon = pd.Series(pd.Series(simplified_line.coords.xy)[1])
 lat = pd.Series(pd.Series(simplified_line.coords.xy)[0])
 si = pd.DataFrame({longs:lon, lats:lat})
 si.tail()

 start_time = time()

 # df_label column will contain the label of the matching row from the original full data set
 si['df_label'] = None

 # for each coordinate pair in the simplified set
 for si_label, si_row in si.iterrows():
     si_coords = (si_row[lats], si_row[longs])

     # for each coordinate pair in the original full data set
     for df_label, df_row in df.iterrows():

         # compare tuples of coordinates, if the points match, save this row's label as the matching one
         if si_coords == (df_row[lats], df_row[longs]):
             si.loc[si_label, 'df_label'] = df_label
             break
```



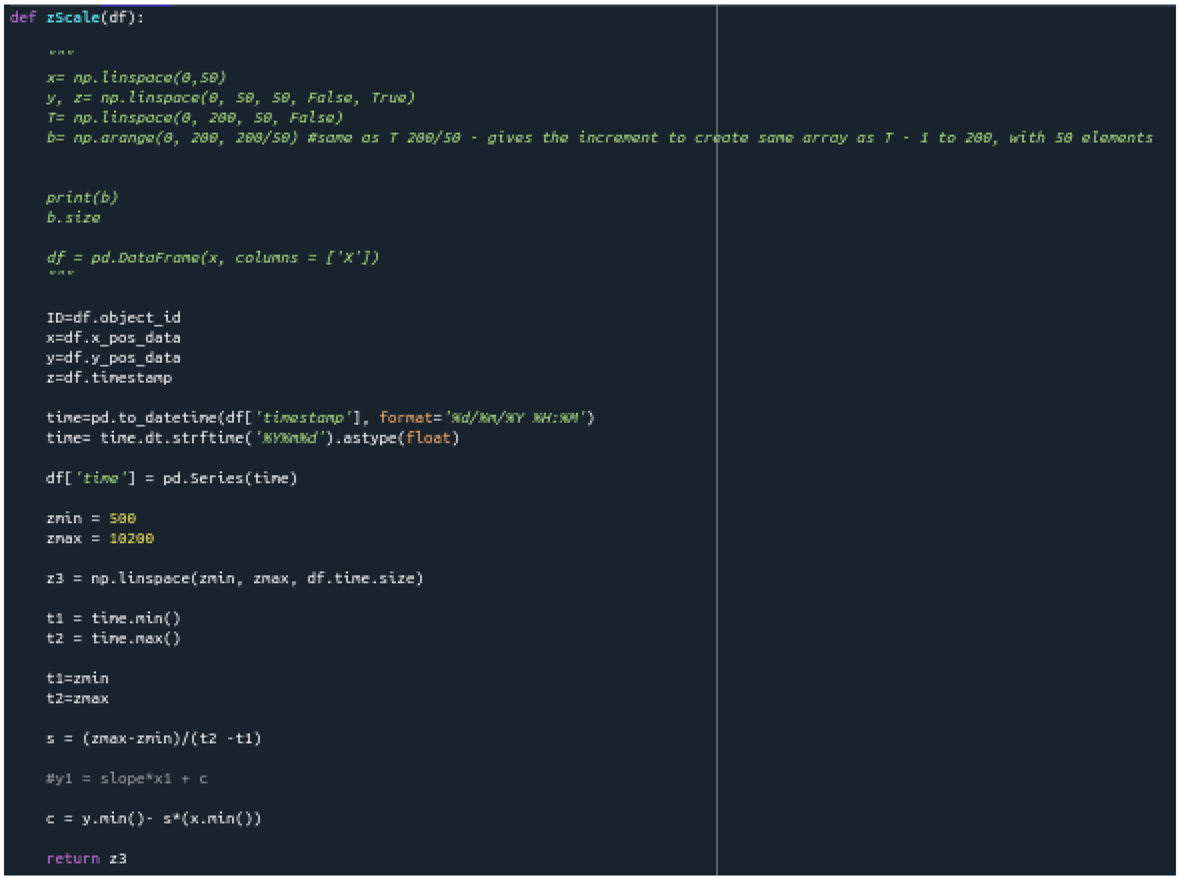
```python
    print('process took %s seconds' % round(time() - start_time, 2))

    # select the rows from the original full data set whose labels appear in the df_label column of the simplified data set
    rs = df.loc[si['df_label'].dropna().values]
    result = pd.concat([rs, result])

    rs.tail()
 #Returns updated simplify version

 #Some of the code was taken from here: https://geoffboeing.com/2014/08/reducing-spatial-data-set-size-with-douglas-peucker/
 return result
```

Figure A4: Simple function



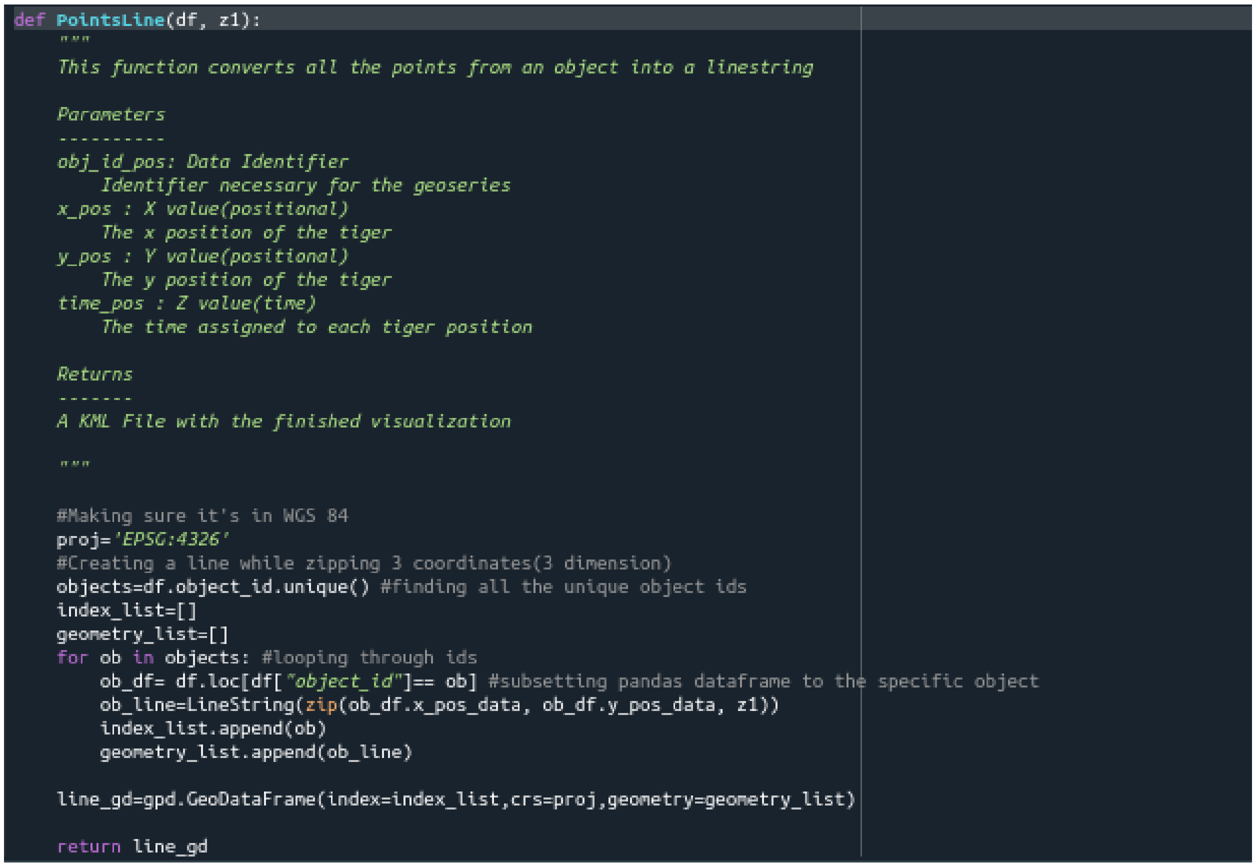
```python
def haversine(Olat,Olon, Dlat,Dlon):
    #Source: https://www.betterdatascience.com/heres-how-to-calculate-distance-between-2-geolocations-in-python/
    """
    Parameters
    ----------
    Olat : minimum latitude
    Olon : minimum longitude
    Dlat : maximum latitude
    Dlon : maximum longitude

    Returns
    -------
    distance in radians


    haversine function finds the distance between the minimum lat and long points to the maximum lat and
    long points

    """

    radius = 6371.  #radius of the Earth in km

    d_lat = np.radians(Dlat - Olat) #finds the difference between max Latitude and min Latitude, and then converts to radians
    d_lon = np.radians(Dlon - Olon) #finds the difference between max Longitude and min Longitude, and then converts to radians
    a = (np.sin(d_lat / 2.) * np.sin(d_lat / 2.) +
         np.cos(np.radians(Olat)) * np.cos(np.radians(Dlat)) *
         np.sin(d_lon / 2.) * np.sin(d_lon / 2.))
    c = 2. * np.arctan2(np.sqrt(a), np.sqrt(1. - a))
    d = radius * c

    return d
```

Figure A5: Haversine distance function



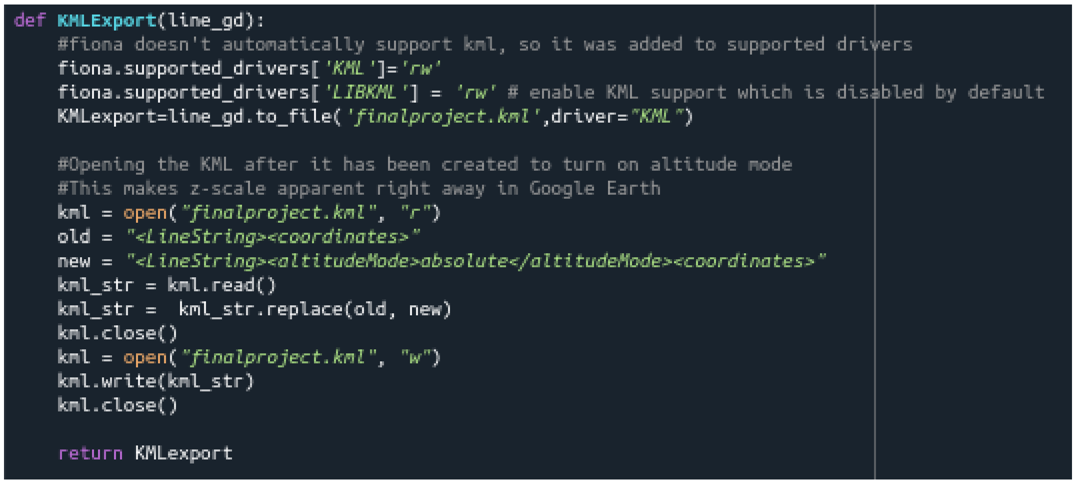
```python
def zScale(df):

    """
    x= np.linspace(0,50)
    y, z= np.linspace(0, 50, 50, False, True)
    T= np.linspace(0, 200, 50, False)
    b= np.arange(0, 200, 200/50) #same as T 200/50 - gives the increment to create same array as T - 1 to 200, with 50 elements


    print(b)
    b.size

    df = pd.DataFrame(x, columns = ['X'])
    """

    ID=df.object_id
    x=df.x_pos_data
    y=df.y_pos_data
    z=df.timestamp

    time=pd.to_datetime(df['timestamp'], format='%d/%m/%Y %H:%M')
    time= time.dt.strftime('%Y%m%d').astype(float)

    df['time'] = pd.Series(time)

    zmin = 500
    zmax = 10200

    z3 = np.linspace(zmin, zmax, df.time.size)

    t1 = time.min()
    t2 = time.max()

    t1=zmin
    t2=zmax

    s = (zmax-zmin)/(t2 -t1)

    #y1 = slope*x1 + c

    c = y.min()- s*(x.min())

    return z3
```

Figure A6: Zscale Function



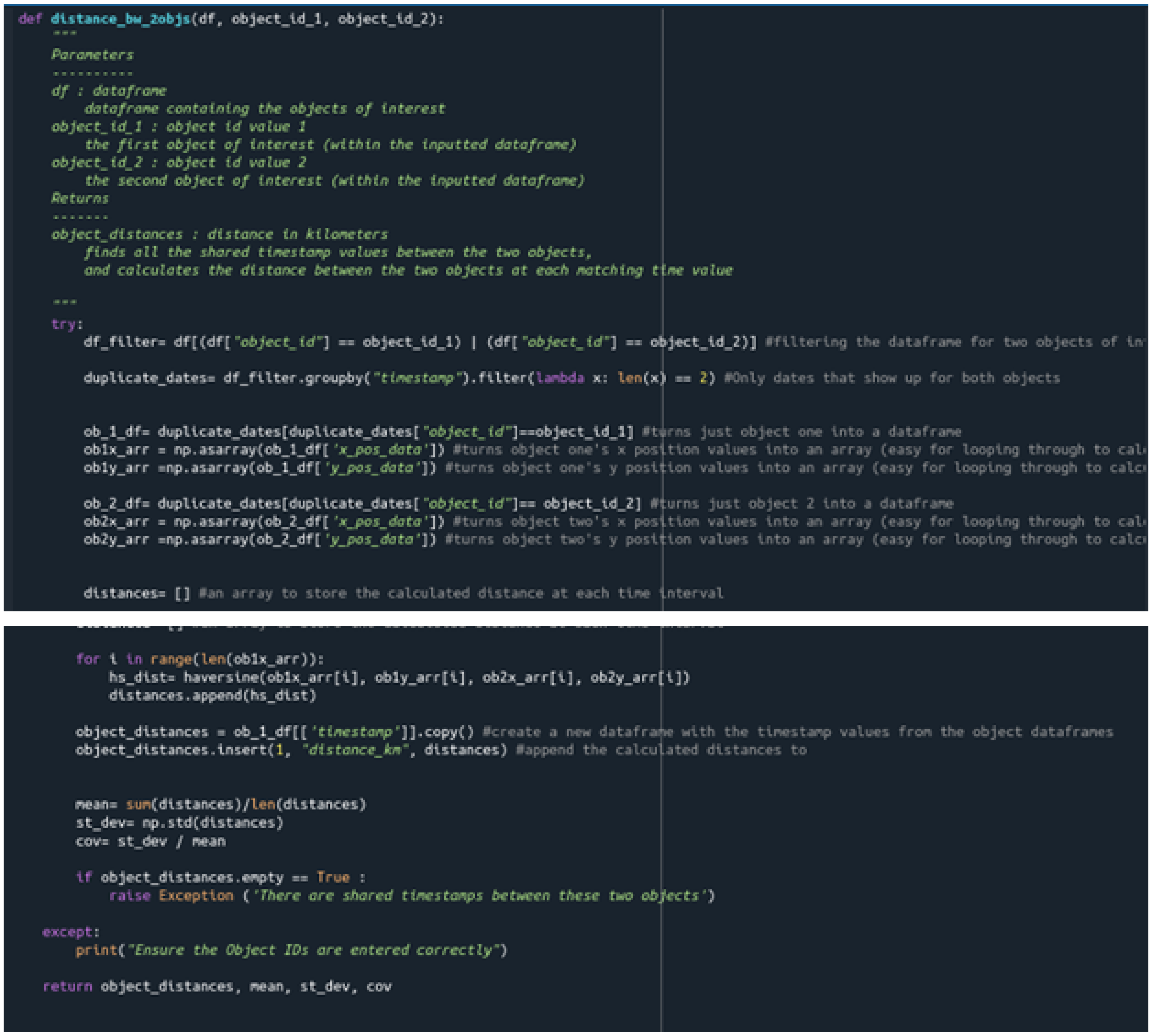
```python
def PointsLine(df, z1):
    """
    This function converts all the points from an object into a linestring

    Parameters
    ----------
    obj_id_pos: Data Identifier
        Identifier necessary for the geoseries
    x_pos : X value(positional)
        The x position of the tiger
    y_pos : Y value(positional)
        The y position of the tiger
    time_pos : Z value(time)
        The time assigned to each tiger position

    Returns
    -------
    A KML File with the finished visualization

    """

    #Making sure it's in WGS 84
    proj='EPSG:4326'
    #Creating a line while zipping 3 coordinates(3 dimension)
    objects=df.object_id.unique() #finding all the unique object ids
    index_list=[]
    geometry_list=[]
    for ob in objects: #looping through ids
        ob_df= df.loc[df["object_id"]== ob] #subsetting pandas dataframe to the specific object
        ob_line=LineString(zip(ob_df.x_pos_data, ob_df.y_pos_data, z1))
        index_list.append(ob)
        geometry_list.append(ob_line)

    line_gd=gpd.GeoDataFrame(index=index_list,crs=proj,geometry=geometry_list)

    return line_gd
```

Figure A7: PointsLine Function



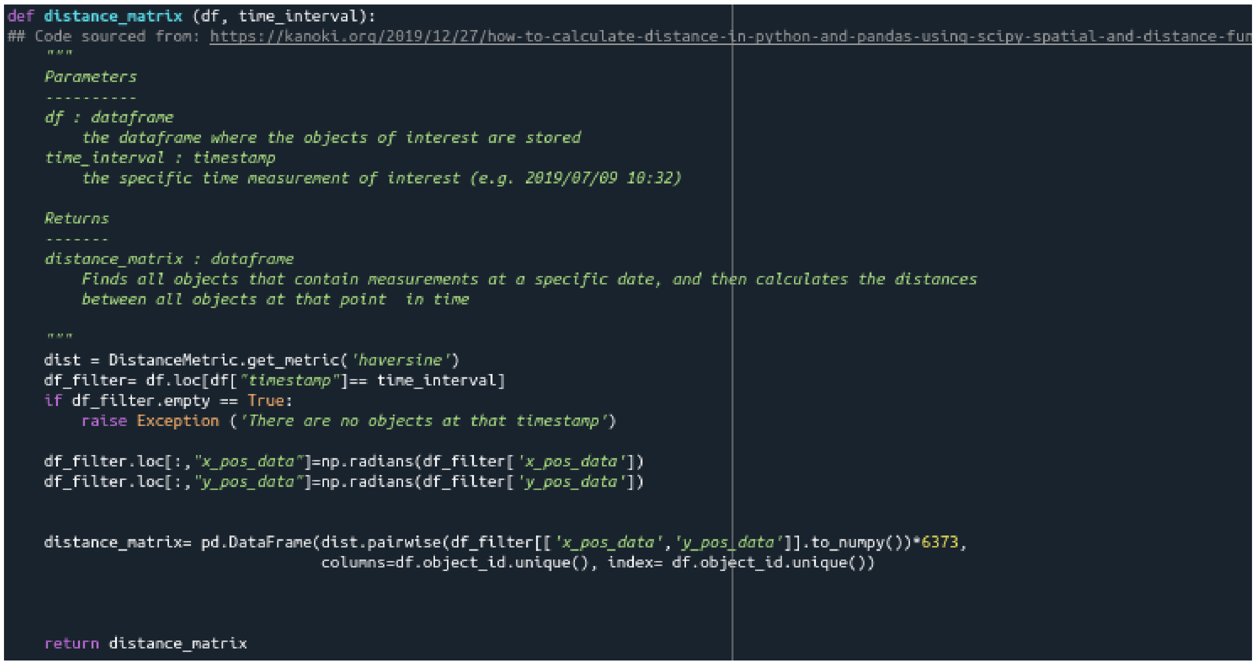
```python
def KMLExport(line_gd):
    #fiona doesn't automatically support kml, so it was added to supported drivers
    fiona.supported_drivers['KML']='rw'
    fiona.supported_drivers['LIBKML'] = 'rw' # enable KML support which is disabled by default
    KMLexport=line_gd.to_file('finalproject.kml',driver="KML")

    #Opening the KML after it has been created to turn on altitude mode
    #This makes z-scale apparent right away in Google Earth
    kml = open("finalproject.kml", "r")
    old = "<LineString><coordinates>"
    new = "<LineString><altitudeMode>absolute</altitudeMode><coordinates>"
    kml_str = kml.read()
    kml_str =  kml_str.replace(old, new)
    kml.close()
    kml = open("finalproject.kml", "w")
    kml.write(kml_str)
    kml.close()

    return KMLexport
```

Figure A8: KML Export Function



```
def distance_bw_2objs(df, object_id_1, object_id_2):
    """
    Parameters
    ----------
    df : dataframe
        dataframe containing the objects of interest
    object_id_1 : object id value 1
        the first object of interest (within the inputted dataframe)
    object_id_2 : object id value 2
        the second object of interest (within the inputted dataframe)
    Returns
    -------
    object_distances : distance in kilometers
        finds all the shared timestamp values between the two objects,
        and calculates the distance between the two objects at each matching time value

    """
    try:
        df_filter= df[(df["object_id"] == object_id_1) | (df["object_id"] == object_id_2)] #filtering the dataframe for two objects of in

        duplicate_dates= df_filter.groupby("timestamp").filter(lambda x: len(x) == 2) #Only dates that show up for both objects


        ob_1_df= duplicate_dates[duplicate_dates["object_id"]==object_id_1] #turns just object one into a dataframe
        ob1x_arr = np.asarray(ob_1_df['x_pos_data']) #turns object one's x position values into an array (easy for looping through to cal
        ob1y_arr =np.asarray(ob_1_df['y_pos_data']) #turns object one's y position values into an array (easy for looping through to calc

        ob_2_df= duplicate_dates[duplicate_dates["object_id"]== object_id_2] #turns just object 2 into a dataframe
        ob2x_arr = np.asarray(ob_2_df['x_pos_data']) #turns object two's x position values into an array (easy for looping through to cal
        ob2y_arr =np.asarray(ob_2_df['y_pos_data']) #turns object two's y position values into an array (easy for looping through to calc


        distances= [] #an array to store the calculated distance at each time interval

        for i in range(len(ob1x_arr)):
            hs_dist= haversine(ob1x_arr[i], ob1y_arr[i], ob2x_arr[i], ob2y_arr[i])
            distances.append(hs_dist)

        object_distances = ob_1_df[['timestamp']].copy() #create a new dataframe with the timestamp values from the object dataframes
        object_distances.insert(1, "distance_km", distances) #append the calculated distances to


        mean= sum(distances)/len(distances)
        st_dev= np.std(distances)
        cov= st_dev / mean

        if object_distances.empty == True :
            raise Exception ('There are shared timestamps between these two objects')

    except:
        print("Ensure the Object IDs are entered correctly")

    return object_distances, mean, st_dev, cov
```

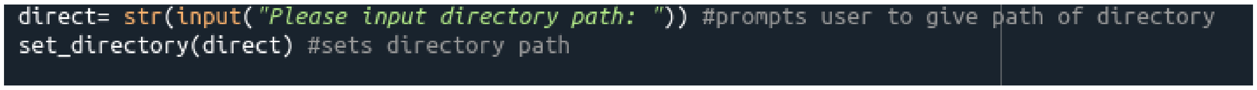
Figure A9: distance_bw_2objs function



```python
def distance_matrix (df, time_interval):
## Code sourced from: https://kanoki.org/2019/12/27/how-to-calculate-distance-in-python-and-pandas-using-scipy-spatial-and-distance-fun
    """
    Parameters
    ----------
    df : dataframe
        the dataframe where the objects of interest are stored
    time_interval : timestamp
        the specific time measurement of interest (e.g. 2019/07/09 10:32)

    Returns
    -------
    distance_matrix : dataframe
        Finds all objects that contain measurements at a specific date, and then calculates the distances
        between all objects at that point  in time

    """
    dist = DistanceMetric.get_metric('haversine')
    df_filter= df.loc[df["timestamp"]== time_interval]
    if df_filter.empty == True:
        raise Exception ('There are no objects at that timestamp')

    df_filter.loc[:,"x_pos_data"]=np.radians(df_filter['x_pos_data'])
    df_filter.loc[:,"y_pos_data"]=np.radians(df_filter['y_pos_data'])


    distance_matrix= pd.DataFrame(dist.pairwise(df_filter[['x_pos_data','y_pos_data']].to_numpy())*6373,
                              columns=df.object_id.unique(), index= df.object_id.unique())



    return distance_matrix
```

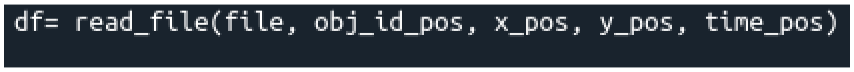
Figure A10: distance_matrix function

Appendix B: Troubleshooting



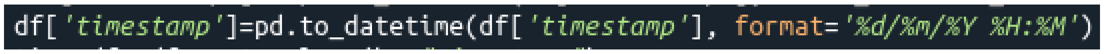
```
direct= str(input("Please input directory path: ")) #prompts user to give path of directory
set_directory(direct) #sets directory path
```

Figure B1: Setting the Directory



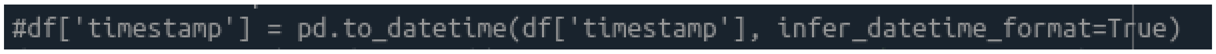
```
df= read_file(file, obj_id_pos, x_pos, y_pos, time_pos)
```

Figure B2: Reading in the user identified file and the identified columns of interest



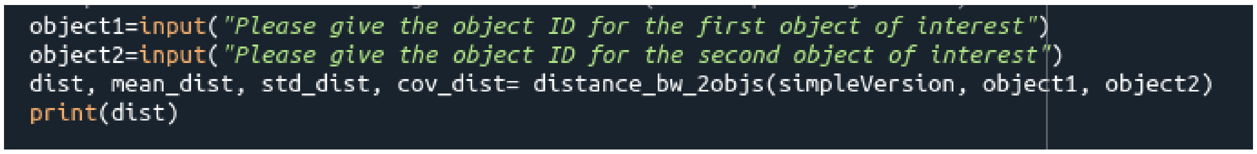
```
df['timestamp']=pd.to_datetime(df['timestamp'], format='%d/%m/%Y %H:%M')
```

Figure B3: Setting the timestamp to a datetime format (DD/MM/YYYY H-M)



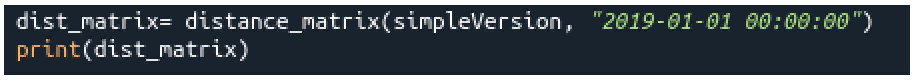
```
#df['timestamp'] = pd.to_datetime(df['timestamp'], infer_datetime_format=True)
```

Figure B4: The alternative code that can be used if the datetime format in figure 3 does not work



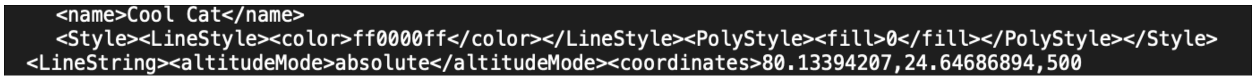
```
object1=input("Please give the object ID for the first object of interest")
object2=input("Please give the object ID for the second object of interest")
dist, mean_dist, std_dist, cov_dist= distance_bw_2objs(simpleVersion, object1, object2)
print(dist)
```

Figure B5: distance_bw_2objs function called in main function



```
dist_matrix= distance_matrix(simpleVersion, "2019-01-01 00:00:00")
print(dist_matrix)
```

Figure B6: Inputting a timestamp value in the distance_matrix function



```
    <name>Cool Cat</name>
    <Style><LineStyle><color>ff0000ff</color></LineStyle><PolyStyle><fill>0</fill></PolyStyle></Style>
<LineString><altitudeMode>absolute</altitudeMode><coordinates>80.13394207,24.64686894,500
```

Figure B7. An edited KML file from the client’s sample data displaying the colour line string