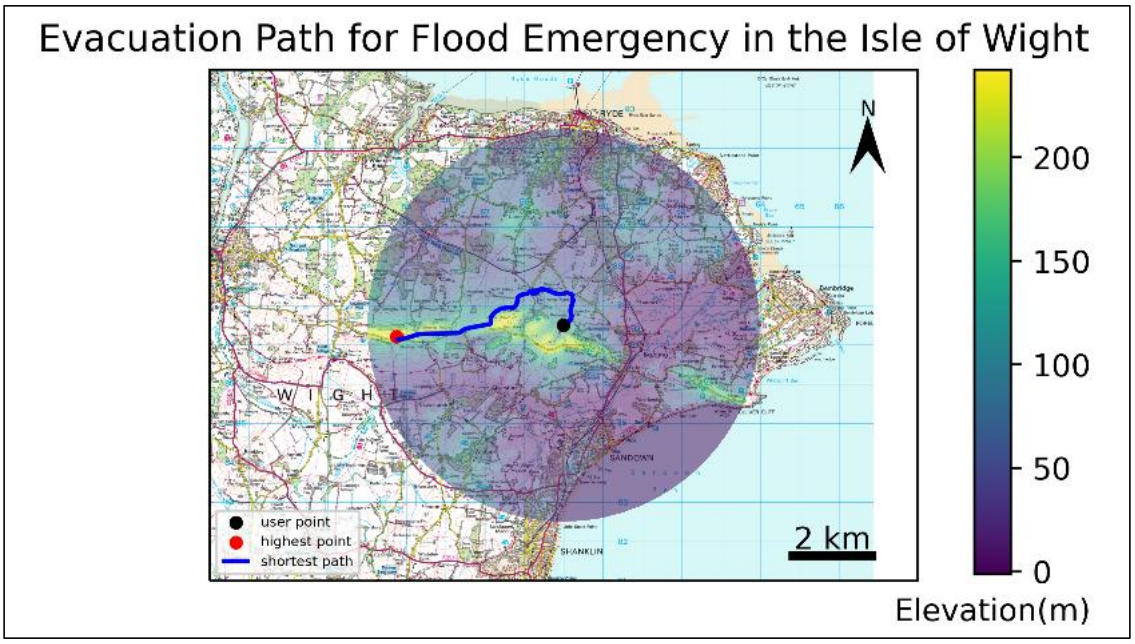
Flood Emergency Planning Report



Adam Morgan Jia Shi Jiayun Kang

Yiru Xu Zhongxian Wang

Introduction

The Flood Emergency Plan is a program for mapping the fastest path to the highest point within a 5 km radius in order to provide the user with the best way to escape a flood emergency. To implement the OOP, the program consists of several classes.

For this project, We develop one python program, which reads the user input point firstly that will be tested and reads the boundary, ITN, roads, elevation, and background map files from the material folder. In this program, we use the point user input to find the highest points (if exist) by elevation ASCII file. Then, searching for the nearest ITN node from the user and the highest points. After that, we identify the shortest path by Naismith's rule to determine the highest point and the quickest path to the point. Finally, we plot the map with the highest point, buffer, shortest route and other essential factors of the map for visualization.

In addition, we consider doing some error handling. Fox example, the commands 'try' and 'except' are used in the class Evacuation App. This function allows the user to enter their eastern and northern coordinates and then tests whether the user is on the Isle of White or its various islands. If the user enters a location on these islands, the (x, y) coordinates are returned as point data and in list format.

In our team, Zhongxian Wang, Jia Shi, Adam Morgan, Jiayun Kang, and Yiru Xu finished the task 1 to task 5 separately, besides, Adam Morgan completed task 6 and was responsible for connecting parts from each one together. To better complete our collaborative programming projects, we created five branches in GitHub for each member so that our code would not conflict with other members and had good online record storage, so everyone could push code to their branch of GitHub flood-emergency-planning-robin regularly. Following the order in which each task was observed, we negotiated the data type of the function's return value and attempted to execute each one at the same time. Thanks to GitHub, we can see the progress and content of other team members easily in different branches, which saves time and avoids waiting for others to finish their parts.

The whole program took about 20 days to finish, how to achieve Naismith's rule is kind of trouble for us. What's more, we have to define a lot of things, which we are struggling with for a long time, but we finally solve all problems we met after reviewing the materials from our lessons and academic internet. Although everyone has different tasks in our group, someone who has completed the front part would work together with others in the next task for the code and report. So we work well together as a group.

Project Description

Python 3.8 in the environment of geospatial and PyCharm Community Edition 2022.2 were used for programming in this coursework. In this program, some packages like geopandas, rasterio, NumPy, Fiona, JSON, Sharply, and Rtree were available. Therefore, it can be run with packages needed above in geospatial environment and Python 3.8 or a more advanced Python version by navigating the terminal to the directory where the script is located using the cd

command and typing python 3.8 main.py in the terminal to execute the script. Apart from this, simply executing the program with a Python IDE(Pycharm) by clicking the button “run this current file” is also accepted. The Material folder is needed to be stored in the same directory to support the “main.py” execution.

Software

Task 1: User Input

In order to finish user input, the task 1 can be divided into two parts, the first part is defining the coordinate of x and y, the second part is determining the extent of a polygon and excluding out-of-range points.

This part of the procedure is implemented in the following steps:

1. User should input their British National Grid coordinate as x and y (easting and northing), which will be converted to float.
2. Then the program will test the input points. If the test point is out of range, it would show ‘This coordinate is outside this box’. If the test point is in range, the program would proceed to the next step.

```
def user_input():
print('Input your British National Grid coordinate (easting and northing)')

x = input('What is your easting coordinate?')  # This returns a string
x = float(x)  # This converts it to a float
y = input('What is your northing coordinate?')  # This returns a string
y = float(y)  # This converts it to a float

current_location = ''
if x < 430000 or x > 465000 or y < 80000 or y > 95000:
current_location = 'This coordinate is outside this box'
print(current_location)
else:
return (x,y)
```

Task 2: Highest Point Identification

To achieve the highest point identification, the task can be divided into three parts, the first one is buffering a point with 5 kilometres, the second, masking the elevation file by this buffer polygon, and lastly, finding the highest point within this scope.

This part of the procedure is implemented in the following steps:

1. The point input by the user in task 1 is used to buffering a 5000m radius circle, which returns a polygon and stores it in GeoDataFrame format with CRS information.
2. In order to clip the raster file by rasterio, we get the coordinates of the geometry in the format that the mask() function of the rasterio package needs by JSON.

3. Then, clip the raster file, and search for the maximum value among all the raster pixels, return the location by where() and xy() function of Numpy and Rasterio to receive the highest points projected (x,y) coordinates.

```
    # Masking/clipping Raster: Copyright 2018, Henrikki Tenkanen. Last updated on Jan 17, 2
018.
    # https://automating-gis-processes.github.io/CSC18/lessons/L6/clipping-raster.html
    def clip_point_buffer(self, point, search_dist):
    # 5km buffer created for inputted point
    point_buffer = point.buffer(search_dist)
    point_buffer_gdf = gpd.GeoDataFrame({'geometry': point_buffer}, index=[0])
    point_buffer_gdf.crs = 'EPSG:27700'
    point_buffer_json = [json.loads(point_buffer_gdf.to_json())['features'][0]['geometry']]
    # Extract dtm data from buffer
    out_img, out_transform = mask(self.dtm, point_buffer_json, True)
    return out_img

    def highest_point_calculator(self, search_raster):
    # reference: https://numpy.org/doc/stable/reference/generated/numpy.where.html
    highest_pixel = np.where(search_raster[0] == np.max(search_raster[0]))
    # highest point pixel location
    highest_coords_crs = self.dtm_data_reader.xy(highest_pixel[0], highest_pixel[1])
    # reference: https://rasterio.readthedocs.io/en/latest/api/rasterio.transform.html,
    # transform from image pixel to geographic/projected (x, y) coordinates
    highest_point.append(Point(highest_coords_crs [0][0], highest_coords_crs [1][0]))
    print(highest_point[i])
    return highest_point
```

Task 3: Nearest Integrated Transport Network

Task 3: Nearest Integrated Transport Network

The Solent Integrated Transport Network JSON file is loaded into python. The dictionary containing the road node coordinates are then extracted from the JSON file. The JSON file is then converted to a list and a spatial index is added to every single node.

```
    def __initialize_node_data(self, dtm):
    # Create list of roadnode name and easting and northing coordinate
    for node, value in self.itn_road_nodes.items():
    name = node
    for coordinates in value.values():
    point = [name, [coordinates[0], coordinates[1]]]
    self.point_nodes.append(point)
    # Add a spatial index to the roadnodes
    for n, point in enumerate(self.point_nodes):
    self.idx.insert(n, point[1], point[0])
```

Rtree was used to speed up the process. Nearest neighbours was used to find the road node nearest to coordinate entered seen in the code below:

```
    def nearest_road_node_to_coordinate(self, point):
    """ Returns the name of the road node nearest to a (x,y) coordinate

    :param point: shapely.geometry.Point data
    :return: The name of the nearest road node in string format
    """

    coord = [point.coords.xy[0][0], point.coords.xy[1][0]]
```

```
55.
56.     nearest_node = list(self.idx.nearest(coordinates=coord, num_results=1, objects=False))
57.     # Retrieve roadnodes names for both node nearest to the high point and node nearest to
the user
58.     nearest_node_name = self.point_nodes[nearest_node[0]][0]
59.     return nearest_node_name
```

The node nearest to the highest point and the user coordinate is then found by entering their respective (x, y) coordinates into the nearest_road_node_to_coordinate function.

```
60.     # return road node nearest to evacuee coordinate
61.     start_node =
self.integrated_transport_network.nearest_road_node_to_coordinate(self.evacuee.point)
62.
63.     # return node nearest to highest point in evacuation zone
64.     end_node = self.integrated_transport_network.nearest_road_node_to_coordinate(high_point)
65.     print("Start node: " + start_node)
66.     print("End node: " + end_node)
```

Task 4: Shortest Path

Task 4 was completed by using Networkx MultiDiGraph, this allowed for multiple edges to be drawn between nodes and for the edges to be directed. This was important as depending on the direction the evacuee walks, they may gain or lose elevation. To calculate the change in elevation between start and end rode nodes the z-coordinate must be found for each road node. The z-coordinate for every single road node was extracted by finding where the road nodes point coordinate is on the raster and then extracting the value from the raster which is the meters height.

```
67.     def elevation_at_point(self, point):
68.     """Returns the elevation in metres for the point provided"""
69.     # Based on info from
70.     # https://gis.stackexchange.com/questions/317391/python-extract-raster-values-at-point-
locations
71.     # https://geopandas.org/en/stable/gallery/geopandas_rasterio_sample.html
72.     coord_list = [point.coords[0]]
73.     try:
74.     v = [x[0] for x in self.dtm_data_reader.sample(coord_list)]
75.     except Exception as e:
76.     print(e)
77.     v = None
78.
79.     return v[0]
```

The change in elevation between nodes are in the equations below:

1. Elevation change between start node and end node in meters = end node elevation (m) – start node elevation (m)
2. Elevation change between end node and start node in meters = start node elevation (m) – end node elevation (m)

Naismith's rule was implemented taking into consideration that a relatively fit person walks 5 kilometres and walking up a 600-meter hill adds on additional hour of walking time. In this application only the time taken to ascend is factored in, not descend so any negative value from

change in elevation is set to zero. The time taken to walk between nodes is then calculated by the calculation below:

- Ascent speed meters per second (ms) = 600 (meters) / 3600 (seconds)
- Walking speed meters per second (ms) = 5000 (meters) / 3600 (seconds)
- Walking time in seconds = (ascent (m) * ascent speed (ms)) + (distance (m) * walking speed (ms))

The code implemented to calculate time taken to walk a distance using Naismith's rule this below:

```
80.    def __calculate_walking_time(self, distance_m, ascent_m):
81.    walking_time_seconds = (ascent_m * self.ascent_speed_ms) + (distance_m *
self.walking_speed_ms)
82.    return walking_time_seconds
```

The code used to add an edge to the MultiDiGraph between nodes taking into how the direction impacts elevation change and therefore time taken to walk in seconds.

```
83.    # Calculating time taken walking start node to end node factoring in elevation and
distance
84.    elevation_diff_start_end_m = end_elevation_m - start_elevation_m
85.    elevation_diff_start_end_m = 0 if elevation_diff_start_end_m <= 0 else
elevation_diff_start_end_m
86.    seconds_walking_start_end = self.__calculate_walking_time(distance_m=length_m,
87.    ascent_m=elevation_diff_start_end_m)
88.
89.    # Calculating time taken walking end node to start node factoring in elevation and
distance
90.    elevation_diff_end_start_m = start_elevation_m - end_elevation_m
91.    elevation_diff_end_start_m = 0 if elevation_diff_end_start_m <= 0 else
elevation_diff_end_start_m
92.    seconds_walking_end_start = self.__calculate_walking_time(distance_m=length_m,
93.    ascent_m=elevation_diff_end_start_m)
94.
95.    # save link in network graph - twice: once for each direction
96.    self.network_graph.add_edge(start_node, end_node, length=length_m,
97.    walk_time_seconds=seconds_walking_start_end)
98.
99.    self.network_graph.add_edge(end_node, start_node, length=length_m,
100.   walk_time_seconds=seconds_walking_end_start)
```

The quickest route is then found between the road node nearest to the user and the road node nearest the highest point within 5km from the users' location. This was done using Dijkstra's shortest path using the time taken to walk in seconds as the weight. This means that the path that is the fastest to walk based upon Naismith's rule is the one that is selected.

```
101.   def quickest_evacuation_route_calculator(self, start_node, end_node):
102.   evacuation_route_nodes = nx.dijkstra_path(self.network_graph, source=start_node,
103.   target=end_node, weight="walk_time_seconds")
104.
105.   evacuation_route_links_gdf = self.__get_links_for_nodes(evacuation_route_nodes)
106.   return evacuation_route_links_gdf
```

The road links have their coordinates returned as a Linestring in a GeoDataFrame to make plotting easier. Higher up in the code the coordinates of each individual road link were converted into Shapely Linestring data format and stored in a GeoDataFrame.

```
107.    # Convert coords list to Shapely LineString
108.    # https://shapely.readthedocs.io/en/stable/reference/shapely.LineString.html
109.    line_vertices = LineString(coords)
110.
111.    # store link details as new row at end of geodataframe
112.    # https://www.statology.org/pandas-add-row-to-dataframe/
113.    new_row = [name, start_node, end_node, length_m, line_vertices]
114.    self.links_gdf.loc[len(self.links_gdf)] = new_row
```

The line strings of all the road links that comprise of the shortest path taken are returned as separate GeoDataFrame using the code below.

```
115.    def __get_links_for_nodes(self, node_list):
116.    """Retrieve links geodataframe for nodes provided"""
117.    # Query the geodataframe to retrieve the links and their linestring geometries
118.    # ie any road link whose start and end nodes are both in the list of evacuation nodes
119.    try:
120.    links_gdf = self.links_gdf.loc[(self.links_gdf['start_node'].isin(node_list)) & (
121.    self.links_gdf['end_node'].isin(node_list)),
122.    ['link_name', 'start_node', 'end_node', 'geometry']].drop_duplicates()
123.    except:
124.    links_gdf = None
125.
126.    return links_gdf
```

Task 5: Extend the Region

To extend the region the point the program checks if the point coordinate is contained inside one of the four polygons that make up the Isle of Wight. By doing this it prevents the user from placing their coordinate in the ocean or a completely different area that is not the Isle of White.

The user is asked for their easting and northing coordinate it converts their input from string to float. The coordinates are then stored both as a shapely.Point data type and a list.

Isle_of_wight.shp was opened using Fiona to convert the ESRI shapefile to GeoJSON. The coordinates of the multipolygon which is comprised of four separate polygons was then extracted from the GeoJSON file.

Each polygon of the multipolygon was converted into shapely.Polygon data type. The Point data containing the users inputted coordinates was tested if they were contained within one of polygons which comprised the total land mass of the Isle of Wight. This allows the software to test that the user is located on any of the Isle of Wight 4 islands and not in the ocean or wrong location.

If the user coordinates were outside of every polygon the software would alert the user, the inputted coordinates are invalid and asks them to re-enter the correct coordinates.

```
127.    def contains_location(self, point):
128.    """ Checks if the evacuees' coordinate is inside the flood extent area
129.
130.    :param point: point: The (x,y) coordinates of the evacuees location
shapely.geometry.Point data format
```

```
131.    :return: True or False
132.    """
133.
134.    # For each polygon comprising Isle of Wight test if the input of the user location is
within each polygon
135.    for i in self.area_boundary:
136.    polygon = Polygon(i[0])
137.    result = polygon.contains(point)
138.    if result is True:
139.    return True
140.
141.    print(f"This coordinate is not inside flood evacuation area: {point}")
142.    return False
```

Task 6: Map Plotting

To plot the map we created a class called “MapPlotter”, inside which we created a function that plot the required elements.

Firstly, we created the class whose attributes are “background” and “elevation”. When creating an object of the class, we need to pass two parameters -- the paths of the background file and the elevation file, which will be used as parameters of rasterio.open() function to open the file.

```
143.    class MapPlotter:
144.    def __init__(self, background_tif_file, elevation_asc):
145.    self.background = rasterio.open(background_tif_file)
146.    self.elevation = rasterio.open(elevation_asc)
```

Inside the class, we created the plot_map() function which takes five parameters.

```
147.    def plot_map(self, user_point_gpd, highest_point_gpd, buffer_img, buffer_out_transform,
shortest_path_gpd):
```

User_point_gpd: A Point() class in the format of GeoDataFrame that is generated from the coordinates that user input.

Highest_point_gpd: A Point() class in the format of GeoDataFrame generated from the coordinates of the highest point.

Buffer_img: Data contained in the raster after applying the 5km-buffer mask.

Buffer_out_transform:Information for mapping pixel coordinates.

Shortest_path_gpd: A LineString() class in the format of GeoDataFrame generated from the coordinates of points of the shortest path.

Below is the codes inside the plot_map() function that plot each part of the map:

1. Plotting the background

1) Reading the background map with the function of read() in rasterio and adjusting the colormap and displayed extent of the background map.

```
148.    background_array = self.background.read(1)
149.    elevation_array = self.elevation.read(1)
150.    palette = np.array([value for key, value in self.background.colormap(1).items()])
151.    background_image = palette[background_array]
152.    bounds = self.background.bounds
153.    extent = [bounds.left, bounds.right, bounds.bottom, bounds.top]
```

```
154.    display_extent = [user_point_gpd['geometry'].x - 9000, user_point_gpd['geometry'].x +
9000,
155.                      user_point_gpd['geometry'].y - 6500, user_point_gpd['geometry'].y +
6500]
```

2) Creating a figure where an GeoAxes() class could be added to plot the other elements.

```
156.    fig = plt.figure(figsize=(5.5, 3), dpi=500)
157.    ax = fig.add_subplot(1, 1, 1, projection=crs.OSGB())
```

3) Plotting the background map with the function of imshow() in matplotlib.plot

```
158.    ax.imshow(background_image, origin='upper', extent=extent, zorder=0)
159.    ax.set_extent(display_extent, crs=crs.OSGB())
```

2. Plotting the user point, the highest point within the buffer area, the buffer elevation and the shortest path with the function of plot() in GeoPandas and plot.show() in rasterio.

```
160.    user_point_gpd.plot(ax=ax, marker='o', color='black', markersize=12, zorder=2,
label='user point')
161.    highest_point_gpd.plot(ax=ax, marker='o', color='red', markersize=12, zorder=3,
label='highest point')
162.    rasterio.plot.show(buffer_img, transform=buffer_out_transform, ax=ax, extent=extent,
alpha=0.5, zorder=1)
163.    shortest_path_gpd.plot(ax=ax, edgecolor='blue', linewidth=1.5, zorder=4,
label='shortest path')
```

3. Plotting the color-bar

1) Creating a new Axes() class with the function of add_axes() in matplotlib.plot

2) Reading the elevation file with the function of read() in rasterio and transforming it to an image with the function of imshow() in matplotlib.plot.

3) Using the function of colorbar() in matplotlib.plot to add the colorbar

```
164.    elevation_array = self.elevation.read(1)
165.    cax = fig.add_axes([0.85, 0.12, 0.03, 0.76])
166.    im = ax.imshow(elevation_array, cmap='viridis')
167.    fig.colorbar(im, cax=cax, orientation='vertical', alpha=0.5)
168.    cax.set_xlabel('Elevation(m)')
```

4. Plotting the legend

```
169.    ax.legend(loc='lower left', prop={'size': 4.5})
```

5. Plotting the north arrow

Referred from the codes online, a new function add_north() was added to plot the north arrow.

Address of the reference and explanation of the parameters are as follows.

```
170.    # Reference: https://blog.csdn.net/qq_44907989/article/details/125584822
171.    def add_north(axe, labelsize=7, loc_x=0.93, loc_y=0.95, width=0.04, height=0.09,
pad=0.14):
172.    """
173.    Draw a scale with 'N' text annotation
174.    :param axe: The coordinate area to be drawn:param labelsize: Display the size of the
'N' text
175.    :param loc_x: Horizontal proportion of the axe
176.    :param loc_y: Vertical proportion of the axe
177.    :param width: The width of the north arrow as a proportion of the axe
178.    :param height: The height of the north arrow as a proportion of the axe
```

```
    :param pad: Gap of text in the proportion of axe
    :return: None
    """
    minx, maxx = axe.get_xlim()
    miny, maxy = axe.get_ylim()
    ylen = maxy - miny
    xlen = maxx - minx
    left = [minx + xlen * (loc_x - width * .5), miny + ylen * (loc_y - pad)]
    right = [minx + xlen * (loc_x + width * .5), miny + ylen * (loc_y - pad)]
    top = [minx + xlen * loc_x, miny + ylen * (loc_y - pad + height)]
    center = [minx + xlen * loc_x, left[1] + (top[1] - left[1]) * .4]
    triangle = mpatches.Polygon([left, top, right, center], color='k')
    axe.text(s='N', x=minx + xlen * loc_x, y=miny + ylen * (loc_y - pad + height),
    fontsize=labelsize,
    horizontalalignment='center', verticalalignment='bottom')
    axe.add_patch(triangle)
```

6. Plotting the scale-bar

Referred from the codes online, a new function add_scalebar() was added to plot the scale bar.

Address of the reference and explanation of the parameters are as follows.

```
    # plot the scalebar
    # Reference: https://stackoverflow.com/questions/32333870/
    # how-can-i-show-a-km-ruler-on-a-cartopy-matplotlib-plot/63494503#63494503
    # plot the scalebar
    # Reference: https://stackoverflow.com/questions/32333870/
    # how-can-i-show-a-km-ruler-on-a-cartopy-matplotlib-plot/63494503#63494503
    def add_scalebar(axe, location=(0.88, 0.05), linewidth=3):
        """
        :param: axe is the axes to draw the scalebar on.
        :param: location is center of the scalebar in axis coordinates.
        :param: linewidth is the thickness of the scalebar.
        """
        # Get the extent of the plotted area according to the coordinate reference system
        x0, x1, y0, y1 = axe.get_extent(crs.OSGB())
        # Define the scalebar location according to the coordinate reference system
        sbx = x0 + (x1 - x0) * location[0]
        sby = y0 + (y1 - y0) * location[1]
        # Calculate a scale bar length
        length = (x1 - x0) / 5000  # in km
        ndim = int(np.floor(np.log10(length)))  # number of digits in number
        length = round(length, -ndim)  # round to 1sf
        # Returns numbers starting with the list
        def scale_number(x):
            if str(x)[0] in ['1', '2', '5']:
               return int(x)
            else:
               return scale_number(x - 10 ** ndim)
        length = scale_number(length)
        # calculate x coordinate for the end of the scalebar
        bar_xs = [sbx - length * 500, sbx + length * 500]
        # Plot the scalebar
        axe.plot(bar_xs, [sby, sby], transform=crs.OSGB(), color='k', linewidth=linewidth)
        # Plot the scalebar label
        axe.text(sbx, sby, str(length) + ' km', transform=crs.OSGB(),
                 horizontalalignment='center', verticalalignment='bottom')
    add_scalebar(ax)
```

```
231.    add_north(ax)
```

7. Adding the title with the function of set_title() in matplotlib.plot

```
232.    ax.set_title('Evacuation Path for Flood Emergency in the Isle of Wight')
233.    plt.show()
```

Task 7: Creativity

1.The data was loaded in during an initialization phase, this allows for subsequent running of the program to be significantly faster.

2.The application runs indefinitely accepting new coordinates of the evacuees' location and drawing the fastest route and plotting a map until the user types 'quit' to exit.

3.A config file is provided.

4.Readme file to let the user how to set up and run the application and includes other important information

5.Docstrings so the user can see what functions require, what they do and what they return

6.A yml file containing the python libraries their versions to create the correct virtual environment to run the application without problems.