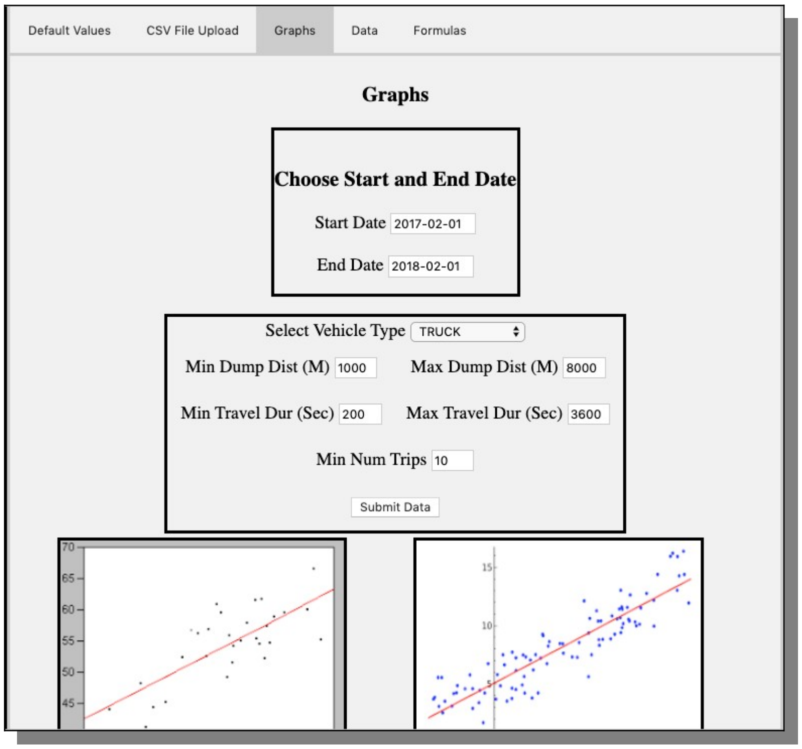
KearlCurves Web Application

Client Side Overview

The first part of the web app is to generate quarterly graph data. These graphs are based on data from the properties that are first displayed and the data from the CSV file that the user selects. Once the user selects a data file, the UI displays a tab labeled “Graphs”. When this tab is selected, there is a button at the bottom of the tab page labeled “Submit Data”. When the user clicks on this button, the quarterly graphs are displayed. The screen looks like:



Graphs

Choose Start and End Date

Start Date 2017-02-01

End Date 2018-02-01

Select Vehicle Type TRUCK

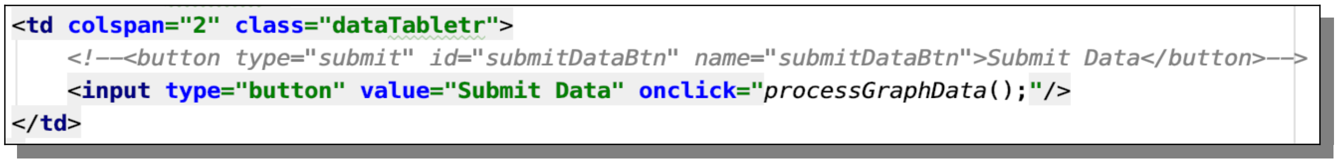
Min Dump Dist (M) 1000 Max Dump Dist (M) 8000

Min Travel Dur (Sec) 200 Max Travel Dur (Sec) 3600

Min Num Trips 10

The purpose of this doc is to explain the mechanism behind the “Submit Data” button action. We will trace the code from this button action through the generation and display of the quarterly graphs.

The action begins at the button element in the HTML:



```
<td colspan="2" class="dataTabletr">
    <!--<button type="submit" id="submitDataBtn" name="submitDataBtn">Submit Data</button>-->
    <input type="button" value="Submit Data" onclick="processGraphData();"/>
</td>
```

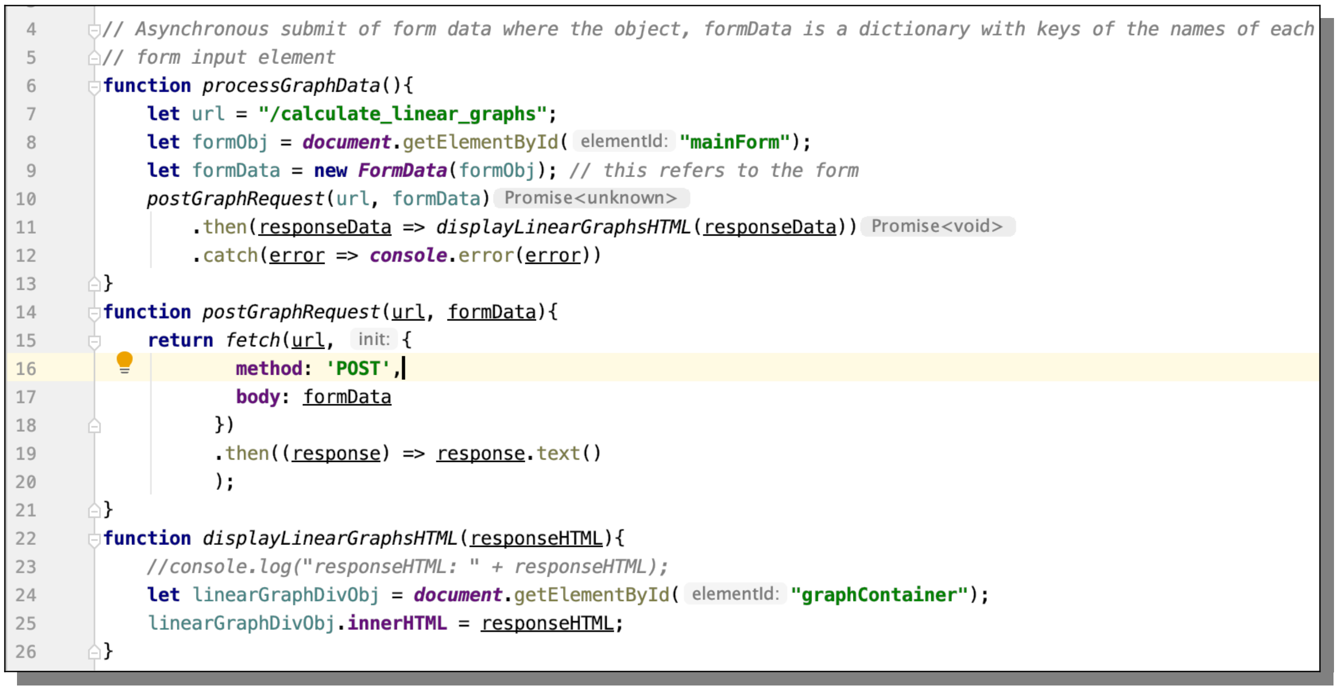
Notice that instead of using <button type=”submit”…., we just create an <input type=”button”…..This is because the action of clicking the button will be an asynchronous (ajax) call. So the web page will not be refreshed, but instead, the graphs will be inserted into the page’s DOM directly.

Client Side Code

The button has an **onclick** listener that calls upon a javascript function, processGraphData(). Overall, the processGraphData() function will do the following tasks:

1. Create an HTTP Request object that contains the action URL, the method type as POST, and the form data, which contains all the user defined input parameters.
2. Post the request to the server with the given URL.
3. Get the response from the server, which will be HTML code that contains the graph references.
4. Insert this returned HTML into the page's DOM.

The code snippet below shows the javascript that accomplishes the 4 tasks:



```
 4  // Asynchronous submit of form data where the object, formData is a dictionary with keys of the names of each
 5  // form input element
 6  function processGraphData(){
 7      let url = "/calculate_linear_graphs";
 8      let formObj = document.getElementById( elementId: "mainForm");
 9      let formData = new FormData(formObj); // this refers to the form
10      postGraphRequest(url, formData) Promise<unknown>
11          .then(responseData => displayLinearGraphsHTML(responseData)) Promise<void>
12          .catch(error => console.error(error))
13  }
14  function postGraphRequest(url, formData){
15      return fetch(url,  init: {
16              method: 'POST',
17              body: formData
18          })
19          .then((response) => response.text()
20          );
21  }
22  function displayLinearGraphsHTML(responseHTML){
23      //console.log("responseHTML: " + responseHTML);
24      let linearGraphDivObj = document.getElementById( elementId: "graphContainer");
25      linearGraphDivObj.innerHTML = responseHTML;
26  }
```

Line 7 specifies the action URL.

The user defined input is specified inside an HTML form tag whose id is "mainForm". So Line 8 gets the form as an object, and line 9 retrieves all the data in the form.

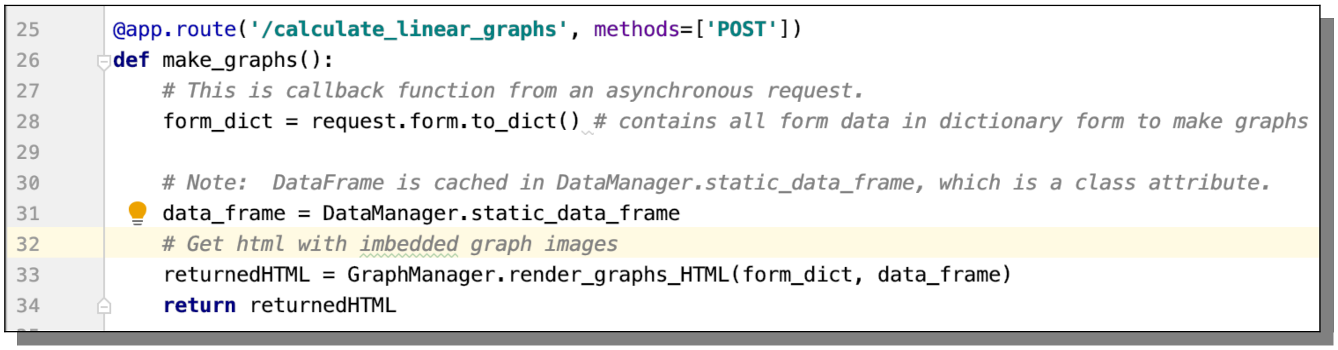
Line 10 calls postGraphRequest() which builds the HTTP Request, posts the request, and gets back the response from the server (variable **response**). The response will be the HTML that the server returns.

Line 11 takes the server's response and calls displayLinearGraphsHTML(). This function simply finds the div that will contain the graphs and sets the response HTML to that div.

So that is an overview of how the submit button action causes the graphs to be displayed in the UI.

Server Side Overview

Notice that in line 7 of the javascript, we specified the action url to be: "/calculate_linear_graphs". When we issued the HTTP Request with this URL, the server will look in our main application, app.py for a Flask route with this name:



```
25  @app.route('/calculate_linear_graphs', methods=['POST'])
26  def make_graphs():
27      # This is callback function from an asynchronous request.
28      form_dict = request.form.to_dict() # contains all form data in dictionary form to make graphs
29
30      # Note:  DataFrame is cached in DataManager.static_data_frame, which is a class attribute.
31      data_frame = DataManager.static_data_frame
32      # Get html with imbedded graph images
33      returnedHTML = GraphManager.render_graphs_HTML(form_dict, data_frame)
34      return returnedHTML
```

In line 28 we retrieve all the form data from the request. Note that the object, **request**, is a global Flask variable that is available in the context of the main app, namely app.py. A **request** object has an attribute called **form**. So we take the **form** and convert it to a Python dictionary.

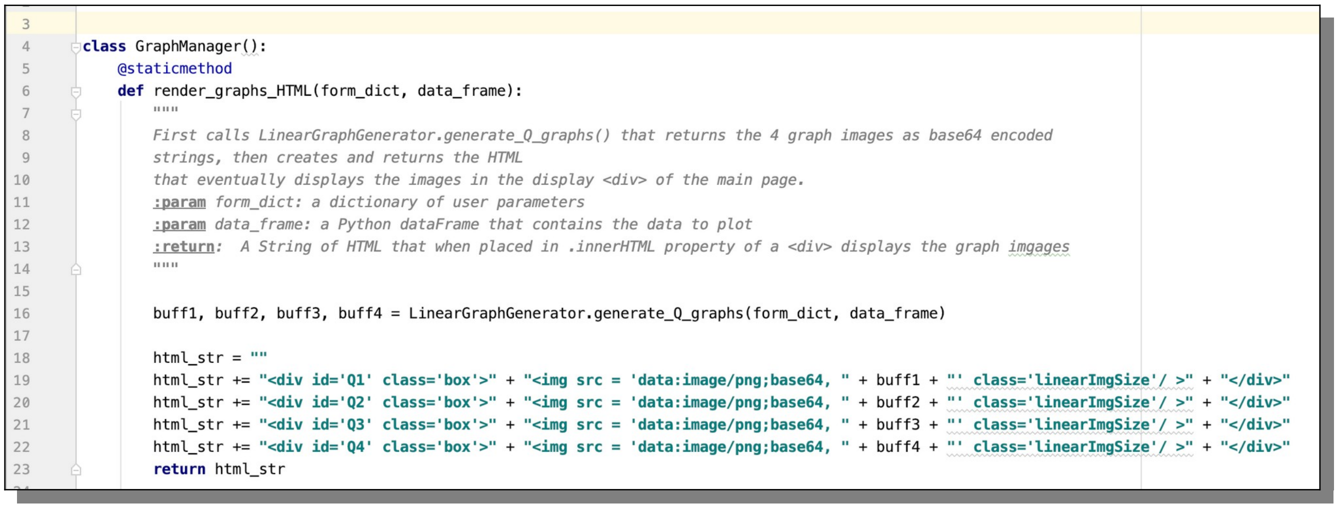
During the process of choosing the CSV data file (See data_manager.py) we converted the read CSV file into a Python DataFrame and stored it in a class attribute of the class DataManager. Remember that a class attribute acts as a static data member of the class, so once assigned, the data member called static_data_frame will contain the DataFrame object. So line 31 retrieves that DataFrame object.

Line 33 calls upon a static method of the GraphManager that generates the HTML that contains references to the graph images.

Line 34 returns the HTML for the graphs, and that is an overview of how the server side code responds to the HTTP Request and generates and returns the HTML for the graphs.

Server Side Code

We will now trace the server side code details that generate the graphs and return the graphs. The class GraphManager has one method: render_graphs_HTML(). This method takes two parameters that are used to bring in the data needed to make the graphs:



```python
class GraphManager():
    @staticmethod
    def render_graphs_HTML(form_dict, data_frame):
        """
        First calls LinearGraphGenerator.generate_Q_graphs() that returns the 4 graph images as base64 encoded
        strings, then creates and returns the HTML
        that eventually displays the images in the display <div> of the main page.
        :param form_dict: a dictionary of user parameters
        :param data_frame: a Python dataFrame that contains the data to plot
        :return:  A String of HTML that when placed in .innerHTML property of a <div> displays the graph imgages
        """

        buff1, buff2, buff3, buff4 = LinearGraphGenerator.generate_Q_graphs(form_dict, data_frame)

        html_str = ""
        html_str += "<div id='Q1' class='box'>" + "<img src = 'data:image/png;base64, " + buff1 + "' class='linearImgSize'/ >" + "</div>"
        html_str += "<div id='Q2' class='box'>" + "<img src = 'data:image/png;base64, " + buff2 + "' class='linearImgSize'/ >" + "</div>"
        html_str += "<div id='Q3' class='box'>" + "<img src = 'data:image/png;base64, " + buff3 + "' class='linearImgSize'/ >" + "</div>"
        html_str += "<div id='Q4' class='box'>" + "<img src = 'data:image/png;base64, " + buff4 + "' class='linearImgSize'/ >" + "</div>"
        return html_str
```

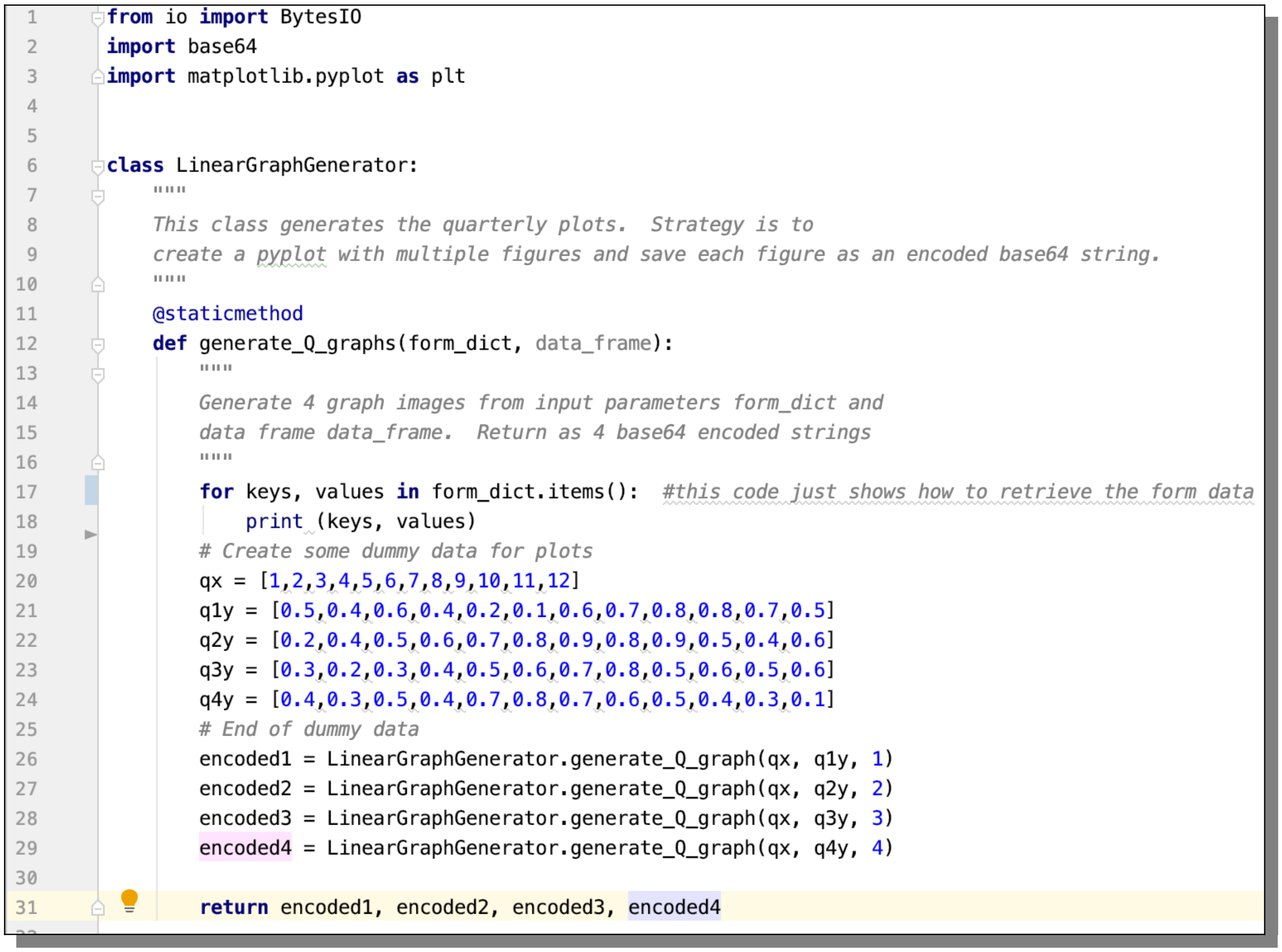
The parameter, **form_dict** is a Python dictionary that contains all the user defined parameters. The parameter, **data_frame** is a Python DataFrame that holds the data from the original CSV data file.

The class method does 3 things:

1. Calls on a method in the LinearGraphGenerator class that returns 4 objects, where each object is a base64 string that is a serialization of a graph quarterly image.
2. Builds the HTML that will contain reference to the graph images as base64 data.
3. Returns the HTML back to the client.

Let’s drill down to the LinearGraphGenerator class to see how the graphs are generated and put into base64 format.

The main method of the class LinearGraphGenerator is:



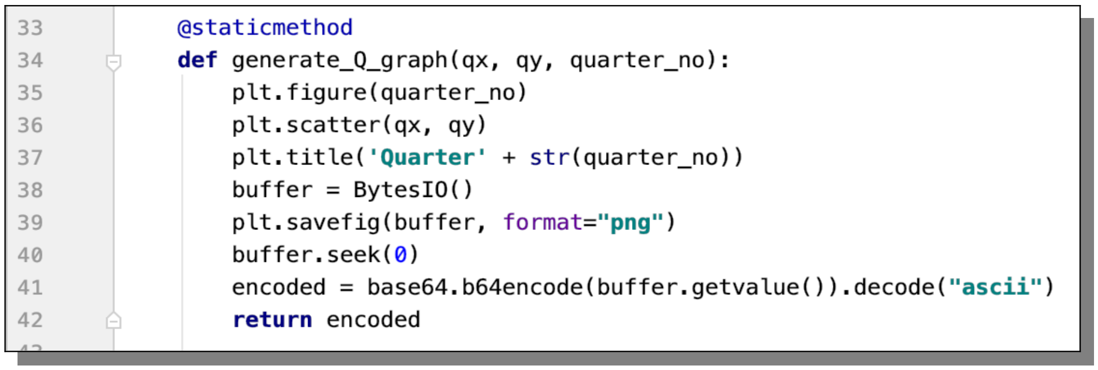
```
1   from io import BytesIO
2   import base64
3   import matplotlib.pyplot as plt
4
5
6   class LinearGraphGenerator:
7       """
8       This class generates the quarterly plots.  Strategy is to
9       create a pyplot with multiple figures and save each figure as an encoded base64 string.
10      """
11      @staticmethod
12      def generate_Q_graphs(form_dict, data_frame):
13          """
14          Generate 4 graph images from input parameters form_dict and
15          data frame data_frame.  Return as 4 base64 encoded strings
16          """
17          for keys, values in form_dict.items():  #this code just shows how to retrieve the form data
18              print (keys, values)
19          # Create some dummy data for plots
20          qx = [1,2,3,4,5,6,7,8,9,10,11,12]
21          q1y = [0.5,0.4,0.6,0.4,0.2,0.1,0.6,0.7,0.8,0.8,0.7,0.5]
22          q2y = [0.2,0.4,0.5,0.6,0.7,0.8,0.9,0.8,0.9,0.5,0.4,0.6]
23          q3y = [0.3,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.5,0.6,0.5,0.6]
24          q4y = [0.4,0.3,0.5,0.4,0.7,0.8,0.7,0.6,0.5,0.4,0.3,0.1]
25          # End of dummy data
26          encoded1 = LinearGraphGenerator.generate_Q_graph(qx, q1y, 1)
27          encoded2 = LinearGraphGenerator.generate_Q_graph(qx, q2y, 2)
28          encoded3 = LinearGraphGenerator.generate_Q_graph(qx, q3y, 3)
29          encoded4 = LinearGraphGenerator.generate_Q_graph(qx, q4y, 4)
30
31          return encoded1, encoded2, encoded3, encoded4
```

Lines 17-18 just show how to retrieve the user input data from the data dictionary.

Lines 20-24 just create some dummy data. You would replace these lines with the extraction of the x and y values for each quarter.

Lines 26-29 call upon this class's method, **generate_Q_graph()** which takes as parameters the x and y arrays of data as well as the quarter number. This method is called 4 times, once for each quarter's data.

The return value of **generate_Q_graph()** is a base64 encoded string of the graph image.



```
33      @staticmethod
34      def generate_Q_graph(qx, qy, quarter_no):
35          plt.figure(quarter_no)
36          plt.scatter(qx, qy)
37          plt.title('Quarter' + str(quarter_no))
38          buffer = BytesIO()
39          plt.savefig(buffer, format="png")
40          buffer.seek(0)
41          encoded = base64.b64encode(buffer.getvalue()).decode("ascii")
42          return encoded
```

The method above creates one encoded base64 string for each quarter graph. The strategy for creating 4 encoded graphs is to create a new figure for each quarter. This is done in line 35.

Lines 36-37 create the scatter plot and assign a title to the plot. We now must serialize the plt image so that it can be transmitted through an HTTP Response back to the client.

Line 38 creates a buffer that is a BytesIO object. A BytesIO object is a 7 bit transformation of a graph image.

Line 39 saves the png image in the BytesIO buffer and line 40 sets the buffer pointer to the beginning for a later read.

Line 41 then converts the contents of the memory buffer to a base64 encoded string which is a format that is compatible with the HTTP Response.

Line 42 returns the base64 encoded form of the graph image.