Project document: Numerical data visualization library

Han Le, Student number: 793919

Bachelor's Programme in Science and Technology, Third year

May 11, 2022

1 General description

In general, this library can be used to visualize numerical data graphically. In a sense, its idea is like matplotlib library of Python; however, currently my library only support line graph.
In this basic line graph, 1 to n straight lines with starting and ending points determined by a set of (x, y) coordinates given by the user.
The given data is read from a CSV file, which is given by the user. The CSV file should have column names (that can be used as category names) and corresponding values.
If the user does not specify the graph name, the default name "Graph" is shown. The legend (the colors used) are included automatically in the graph. The user are able to name the axes and units of the axes.
The coordinate axes are numbered in a way that there is not too much spacing between labels and labels are not drawn on top of each other.
The user has the possibility to decide whether a grid will be drawn behind the graph. The grid will be drawn with a faintly visible dotted line. The size of the grid tiling can be configurable by the user.
The level of difficulty of the final result is estimated to be moderate.

2 User interface

First, the user creates a new object with a freely chosen name. In the figure below, the object is Demo. In this object, create a new Visualization, which creates graph, handles components, and visualizes the graph.

```
object Demo {
  def main(args: Array[String]): Unit = {
    val demoGraph = new Visualization()
    val reader = new CSVReader("./example/silver.csv")
    val data = reader.records

    // User can choose the range of values for x axis and convert them to Double
    val xAxis = Vector.range(1, data("Date").size + 1, 1).map(_.toDouble)

    // User can choose which columns to visualize.
    // For example, here we can choose columns Open, High, Low, Close and the last 30 values of each column to visualize.
    // Then, add the line to the graph.
    val open = new Line("Open price", (xAxis zip (data("Open").map(y => y.toDouble))).takeRight(30))
    demoGraph.addLine(open)

    val close = new Line("Close price", (xAxis zip data("Close").map(y => y.toDouble)).takeRight(30))
    demoGraph.addLine(close)

    val high = new Line("Highest price", (xAxis zip data("High").map(y => y.toDouble)).takeRight(30))
    demoGraph.addLine(high)

    val low = new Line("Lowest price", (xAxis zip data("Low").map(y => y.toDouble)).takeRight(30))
    demoGraph.addLine(low)
```

Figure 1: An example use case.

The user can create a new `CSVReader` to read a csv file. The data read by `CSVReader` is stored in the variable `records`.
The user can choose the range of values for x axis. User can choose which columns to visualize. For example, here we can choose Open, High, Low, Close and the last 30 days of each column to visualize. These columns are open price, close price, highest price, and lowest price of silver in a day. Then, user must add the lines to the graph.
The user can set the name and unit for x axis and y axis by using the methods shown below. If the user does not specify unit, it is not shown.

The graph name can be chosen by the user by the method `nameGraph`. The size of the grid tiling can be configurable by the method `gridSize`. Finally, the graph is shown by method `show()`.

```
demoGraph.nameGraph("Silver price")

demoGraph.nameXAxis("day number")
demoGraph.nameYAxis("price")

demoGraph.yUnit("USD")    // name the unit of y axis
demoGraph.showGrid(true)  // choose to show the grid

demoGraph.gridSize(10, 10)

demoGraph.show()
```

Figure 2: An example use case.

3 Program structure

The relationships between classes are described in a simple UML class diagram below. Note that it only includes the key methods.
`Visualization` is the class that describes the user interface. It is the class that user can create and add components (i.e., trend lines). Used can change axis names by calling the methods `nameXAxis` and `nameYAxis`. If the user does not specify the axis names, titles `X` and `Y` are shown for corresponding axes.
User can choose to show the grid or not by the boolean parameter of method `showGrid`. If the user does not call this method, the grid is not shown. To show the graph, user must call method `show()`.
Class `CSVReader` is only used by the user to read the input data. This data can be then used to choose, e.g., the columns to visualize. Therefore, class `CSVReader` is not connected to other classes in the UML diagram.
Class `Line` is created by the user to make a trend line with `name` (i.e., line name, which can be used for legend) and `data` (i.e., vector of tuples of (x,y) coordinates).

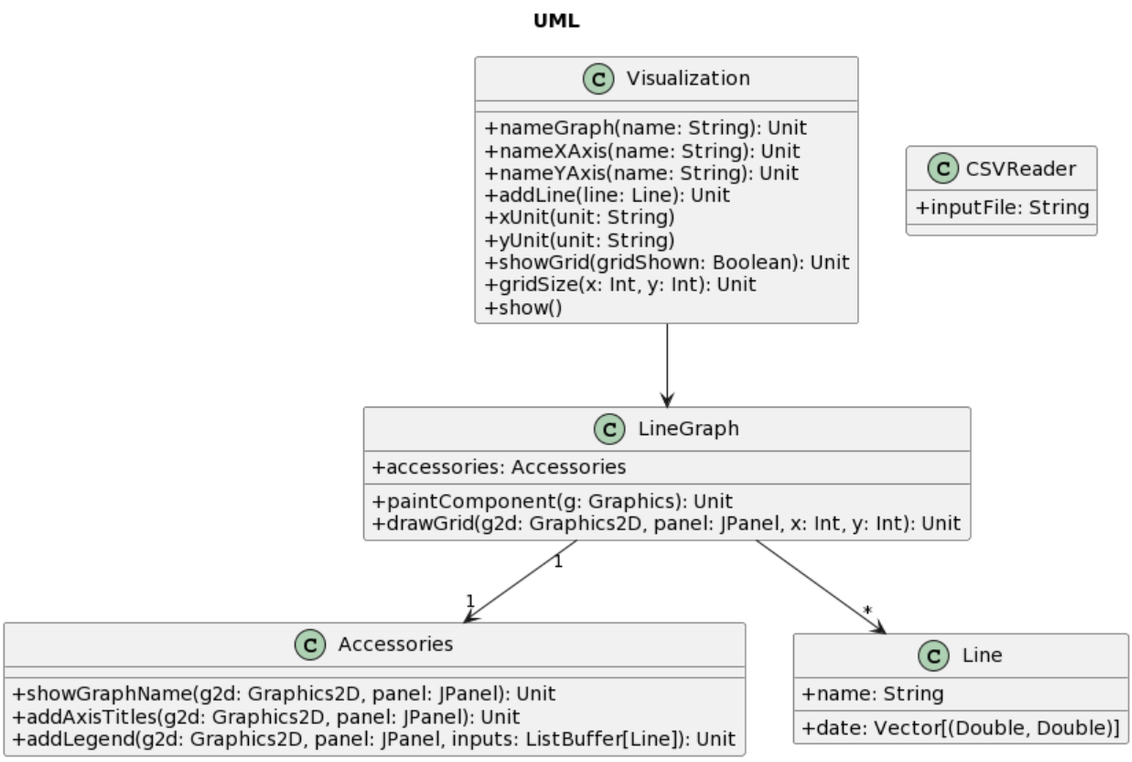

Figure 3: A simple UML diagram of the project. Note that it only includes the key methods.

4 Algorithms

This part explains how the main problems are solved. The scaling from number (i.e., from the input data) to pixel is important. Basically, minimum and maximum bounds of all inputs are computed.

Also, scaling ratios for x and y in pixel are computed. Then, go through all (x,y) coordinates to scale each point to pixels. From these points, we can draw the lines (by connecting two adjacent points) and plot the points.
There are 4 default line colors assigned by the library. If there are more than 4 input lines, the library generates a random line color.
Default number of ticks on both x axis and y axis is 10. These also takes care the grid size, because it looks nice if the locations of grid lines match the locations of ticks. This is a design choice.
If there are less than 10 data points in each line, the ticks and labels on axes are adjusted based on the amount of data points.

5 Data structures

Data read by the `CSVReader` is stored in a mutable `LinkedHashMap`. It is chosen because this it is insertion-ordered and it can be used to record the input column names as keys and data of columns as values.
Class `Line` store data in a vector of tuples of (x,y) coordinates. Vector supports constant-time element access.

6 Files

The library deals with text csv file. Comma is used to separate values. The csv file should have the the column names in the first row, which can be used to name the trend lines. As an example, check the csv file of silver prices on different dates in the folder *example* of the project.
Each line of the file is a data record. For example, in the silver price example, each line is the silver prices of a single date. The valid file format should not have missing data or redundant data.

7 Testing

For graphical things, testing was mainly carried out through GUI, which was in the initial plan.
In the implementation process, testing was also done quite a lot by printing out the values. For example, when creating the lines with a few data points (such as 5), the data of the lines were printed out to be checked. However, there could have been a unit test for this part.
The CSVReader was tested by unit test (check class `CSVReaderTest`) to check if the data was read correctly.
Also, the CSV reader should throw exception if it loads a file that does not exist, or if the input file format is invalid. Invalid file format has, for example, missing value or redundant value in a column. As an example, there is an invalid file format in the folder *example* of the project.
As the results, all the tests in `CSVReaderTest` are passed.
Regarding data scaling, it was manually tested by changing a few values of silver prices to negative. The generated graph seemed to have the expected scaling. I should have written a test for data scaling (i.e., scaling the data from numbers to pixels) because this step is quite complicated.

8 Known bugs and missing features

If the input values are discrete, the labels on axes are shown as decimal numbers, which looks quite unusual. For example, in the silver price demo, the x axis data is discrete (e.g., day number), but the the decimal x values are shown.

Figure 4: The line graph generated from the demo with silver prices.

Honestly, I am not sure if there is any other bugs. Testing special cases would be helpful to discover possible bugs.

9 Three best sides and three weaknesses

Regarding good sides, first, if the user resize the graphical frame by using mouse, the graph components are still visible and look okay. Second, the user does not have to specify the names and units of axes, graph name, and grid size because there are default values that take care of these.
Regarding weakness, first, the library currently only supports non-discrete numerical values. This point is discussed in more details in part 8. Second, although the users can manipulate the input data, they have to do some complicated commands to create the trend lines and have to add the lines to graph by themselves. Third, the library should be re-structured a bit to allow potential expandability in the future. For example, adding other graph types (basic histogram and pie diagram) should be relatively simple.

10 Deviations from the plan, realized process and schedule

Initially, the library was planned to support also simple histogram and pie chart. However, due to my schedule, I did not have enough time to implement these two types of graph.
The final class structure is actually simpler than the initial plan because of the same reason. I have learned that it actually can take a lot of time to organize the code structures.
In the actual implementation, the general process is below:
1. Scale the data to pixels
2. Draw the lines and create a demo
3. Read the input file
4. Create tests for CSVReader
5. Draw the X and Y axes, ticks and units of the axes
6. Make the grid
7. Make the legend

11 Final evaluation

The good and bad aspects are discussed in part 10. Moreover, more testing could be done, as mentioned in part 8.
The drawing of ticks should be fixed to show discrete values. It should be easier and simpler for the users to manipulate the data and create the lines.

The library should be re-structured a bit to allow potential expandability in the future. For example, class `Accessories` should have been used to take care of axes, grids, etc. For further development, this class can contain, e.g., grids, axes and ticks (these are currenly in class `LineGraph`), so that `Accessories` can be used for other types of graphs such as histogram.

12 References

1. Guide to Scaladoc. https://www.baeldung.com/scala/scaladoc
2. Scaladoc Style Guide. https://docs.scala-lang.org/style/scaladoc.html
3. Java Swing Tutorials. https://docs.oracle.com/javase/tutorial/2d/index.html
4. Abstract Window Toolkit (AWT). https://docs.oracle.com/javase/8/docs/technotes/guides/awt/index.html

13 Appendices

Full source code: https://version.aalto.fi/gitlab/leh11/numerical-visualization-library

Demo of a use case:

```
// Demo of a specific use case. This dataset is downloaded from Kaggle.

object Demo {
  def main(args: Array[String]): Unit = {
    val demoGraph = new Visualization()
    val reader = new CSVReader("./example/silver.csv")
    val data = reader.records

    // User can choose the range of values for x axis and convert them to Double
    val xAxis = Vector.range(1, data("Date").size + 1, 1).map(_.toDouble)

    // User can choose which columns to visualize.
    // For example, here we can choose columns Open, High, Low, Close and the last 30 values of each column to visualize.
    // Then, add the line to the graph.
    val open = new Line("Open price", (xAxis zip (data("Open").map(y => y.toDouble))).takeRight(30))
    demoGraph.addLine(open)

    val close = new Line("Close price", (xAxis zip data("Close").map(y => y.toDouble)).takeRight(30))
    demoGraph.addLine(close)

    val high = new Line("Highest price", (xAxis zip data("High").map(y => y.toDouble)).takeRight(30))
    demoGraph.addLine(high)

    val low = new Line("Lowest price", (xAxis zip data("Low").map(y => y.toDouble)).takeRight(30))
    demoGraph.addLine(low)

    demoGraph.nameGraph("Silver price")

    demoGraph.nameXAxis("day number")
    demoGraph.nameYAxis("price")

    demoGraph.yUnit("USD")    // name the unit of y axis
    demoGraph.showGrid(true)  // choose to show the grid

    demoGraph.gridSize(10, 10)

    demoGraph.show()

  }
}
```

Figure 5: An example use case. Object Demo is included in the project directory.