Generating Random Graphs for Training Generative Adversarial Networks

Michael Kazman

michael.kazman@carleton.ca

Supervised by: Dr. Majid Komeili

majidkomeili@cunet.carleton.ca

COMP4905 - Honours Project
School of Computer Science
Carleton University

December 17th, 2021

Declaration of Authorship

I hereby certify that this thesis is entirely my own original work except where otherwise indicated. I am aware of the University's regulations concerning plagiarism, including those concerning consequent disciplinary actions. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Abstract

The work being developed is based around the idea of creating a Generative Adversarial Network (GAN) for translating computerized images of 2-D graphs to tactile graphics [7]. Tactile graphics are the standard format used by the Braille Authority of North America for conveying non-textual information to people who are blind or visually impaired [8]. While this may include tactile representations of pictures, maps, graphs, diagrams, and other images, however, in the context of this project it will only represent 2-D graphs. Graphs of varying types will need to be generated, including but not limited to area graphs, bar charts, box plots, bubble plots, contour plots, error bar plots, histograms, KDE plots, line graphs, scatter plots, and violin plots. As having a reliable and robust set of training data is the fundamental step in any data science problem, this project is concerned with generating an endless amount of fully randomized training data in the form of 2-D graphs. These graphs will be plotted using random data generated through a multitude of ways and stylized with various libraries and themes. Once generated, the graphs will be fed into a GAN for translation in a future project alongside a graduate student of the same supervisor.

Acknowledgements

Throughout the writing of this project I have received a great deal of support and assistance.

I would first like to thank my supervisor, Dr. Majid Komeili, who presented me with the project idea and provided invaluable expertise along the way. Your sympathy and understanding throughout the global pandemic have been truly appreciated and speak true to your welcoming character.

In addition, I would like to thank my parents for their constant check-ins and interest throughout the entire process. You have been a staple to my incredible progress and make me proud to call you my parents.

Finally, I could not have completed this project without the support of my closest friend, and developmental rubber duck, Elodie Kane. The stimulating discussions and much needed distractions were tremendous factors to my success, and will not be forgotten.

Thank you.

Table of Contents

List of Figures ix

List of Tables xi

1 Introduction 1

2 Motivation 3

2.1 Problem Domain 4

2.2 Use Cases 4

2.3 Existing Solutions 6

3 Methodology 7

3.1 Midpoint Displacement 7

3.1.1 Generation 7

3.1.2 Use Cases 8

3.2 Regression 10

3.2.1 Generation 10

3.2.2 Use Cases 10

3.3 Distribution Sampling . . . 12
3.3.1 Generation . . . 12
3.3.2 Use Cases . . . 16
3.4 Geometric Brownian Motion . . . 17
3.4.1 Generation . . . 18
3.4.2 Use Cases . . . 19
3.5 Perlin Noise . . . 19
3.5.1 Generation . . . 20
3.5.2 Use Cases . . . 21
3.6 Random Number Generation . . . 23
3.6.1 Generation . . . 23
3.6.2 Use Cases . . . 23

4 Procedure **26**
4.1 Environment Setup . . . 26
4.2 Generation Pipeline . . . 28
4.2.1 Data Separation . . . 28
4.2.2 Data Generation . . . 30
4.2.3 Style Generation . . . 30
4.2.4 Graph Creation . . . 33
4.2.5 Graph Exportation . . . 33
4.3 Regeneration Pipeline . . . 35
4.3.1 Chart Regeneration . . . 35

5 Design Decisions **37**

5.1 Environment 37

5.2 Modularity 38

5.2.1 Dynamic Modules 38

5.2.2 Graph Regeneration 39

5.3 File Structure 39

5.4 Future Support 40

5.5 Exportation 41

5.6 Stylization & Theming 42

6 Future Work **43**

6.1 Data Modifiers & Multipliers 43

6.2 Additional Theming 44

6.3 Mini-Batching 44

6.4 Multimodal Histograms 45

References **46**

APPENDICES **48**

A Graph Generation **49**

A.1 Data Generation 49

A.2 Data Stylization 50

A.3 Graph Creation & Graph Exportation 50

A.4 Graph Regeneration 51

B Graph Regeneration **52**

B.1 Graph Regeneration 52

C Generation Output **53**

C.1 Example Data Object 53

C.2 Example Style Object 54

List of Figures

2.1 A decision tree dictating whether a tactile graph is appropriate [8] 5

2.2 Visualization of graphs in braille using the BrailleNote Touch [14] 6

3.1 Example Midpoint Displacement Landscape Generation [1] 8

3.2 Iterations of Midpoint Displacement Algorithm [1] 8

3.3 Midpoint Displacement Chart Generation 9

3.4 Noise in Positive Linear Regression - Scatter Plot 11

3.5 Linear and Logarithmic Regression - Bubble Plot 11

3.6 Multivariate Gaussian Distribution [16] 15

3.7 2-D Continuous Probability Distributions - Histogram 16

3.8 Gaussian Distributions 17

3.9 Multivariate Gaussian Distribution - Contour 17

3.10 Geometric Brownian Motion - Line Graph [5] 18

3.11 Example Applications of Perlin Noise 19

3.12 Generation Steps of Perlin Noise [3] 20

3.13 2-D Representation of Perlin Noise Influence Values [3] 21

3.14 Improved Perlin Noise easing function 22

3.15 Random Number Graph Generation 24
3.16 Bar Chart Random Correlations 24

4.1 Graph generation system architecture 27
4.2 Example of a bar chart data generation object 31
4.3 Example of a line graph's stylization JSON file 32
4.4 Various plots with random styles and themes generated using the pipeline . 34

List of Tables

4.1 Graph-type to library occurrences (150) for equal and random distributions 29

Chapter 1

Introduction

This report entertains the idea of generating random graphs. More specifically, the graphs will be used as training data for a Generative Adversarial Network tasked with Text-to-Image-to-Text Translation [11]. However, rather than using generic text as output, the GAN will be tasked with converting the images to a tactile writing system, more specifically braille. And while this would prove extremely beneficial for the visually impaired, before any real-world applications can be entertained, the first barrier of entry must be solved. In this case, the first obstacle was obtaining an appropriate dataset.

As having a large and refined data set is one of most important aspects to any data science problem, generating a dataset of random graphs is a fundamental function for the entire text-to-image-to-text translation procedure and should be documented accordingly. The following report will provide an in-depth look into various facets of the graph generation and regeneration pipeline. To start off, the report will cover an overview of the motivation and problem domain to further set the scene as to what challenges are being solved and why. The following chapter will provide be an overview of the methodology, which will discuss topics such as the generation algorithms and their use cases. In addition, examples of theorized solutions and discarded approaches will also be brushed upon. Subsequently, there will be a deep dive into the implementation details such as how the

entire chart generation and regeneration pipelines are structured. Moreover, a detailed explanation of what is needed for future application support such as additional libraries and graph types will also be examined. The final two chapters discuss the important design decisions made along the way and the rationale behind them, as well as the state of project going forward and any unfinished tasks respectively.

That being said, the main objective of this project is to create a system for automatically generating 2-D plots in different styles using a multitude of unique Python libraries. This system will then be used to create training data for the network pipeline, as reliably gathering tens of thousands of randomized graphs is a tedious and difficult process. The main deliverable for the project is a Python project executed through a Jupyter notebook for generating an endless amount of images from the visualization libraries Altair, Bokeh, and plotnine respectively. These images representing the 2-D plots, along with the raw values and parameters used to generate each chart, will then be saved to a specified directory for future testing and model assessment.

Chapter 2

Motivation

The project detailed in this report was created to generate random graphs for the purpose of training Generative Adversarial Networks (GAN) [7]. Furthermore, the graphs created would be used as input data for a GAN that would translate images of graphs to a tactile writing system, more specifically braille. The fundamental basis for solving this translation is based around the idea of a Text-to-Image-to-Text GAN.

More technically, the model would learn the meaning of a sentence and generate an image depicting the sentence. That image would then be put through a second "captioning" network to provide captions to the generated images. The distance between the ground truth captions and generated captions is then explored to further improve the network [11]. This system would use the data generation processes introduced in this paper as a basis for the text inputted into the system, as well as the regeneration pipeline for further image validation. The details of which will not be further brushed upon in this paper, however, it should be noted that the network will be completed by a graduate student under the same supervisor, Dr. Majid Komeili.

2.1 Problem Domain

One of the issues with representing scientific graphs in tactile format is understanding of what figures should be translated. The Braille Authority of North America has a diagram on this topic to dictate whether a figure is appropriate for a tactile graphic. As Figure 2.1 would indicate, the graphs generated in this project fall under the category of images appropriate for a tactile graphic with one condition being most notable. The condition being the node in the decision tree that inquires as to whether a reader needs the information from a map, figure, or graph to participate in discussion, answer questions, or complete a task. This shows a significant need for tacticle graph translation which is not yet widely accessible.

2.2 Use Cases

The ability to translate mathematical and scientific graphs into braille is incredibly useful. As indicated by Dr. Sile O'Modhrain from the University of Michigan, refresh-able braille displays have a wide range of problems [17]. On top of the devices being extremely costly in the range of $3,000 - $5,000 per line, text is only displayed one line at a time. This would be comparable to reading a book on an e-reader line by line, which is most unwelcome amongst the visually impaired [17]. The final issue, which is the underlying motivation for this problem, is that modern braille devices are unable to communicate spatially distributed information [17]. This would include bar charts, histograms, line graphs, and spreadsheets amongst others. Part of the issue comes with graph flexibility, as providing a digital medium to express a multitude of graph types is not an insignificant task.

Currently, no accessible solution exists for effectively communicating a wide range of graphs through a tactile format like braille. While existing solutions, such as the BrailleNote Touch [12], have some graphing functionality (Figure 2.2), they still face the issues of cost and flexibility mentioned above. As specified in section 5.2, a major developmental

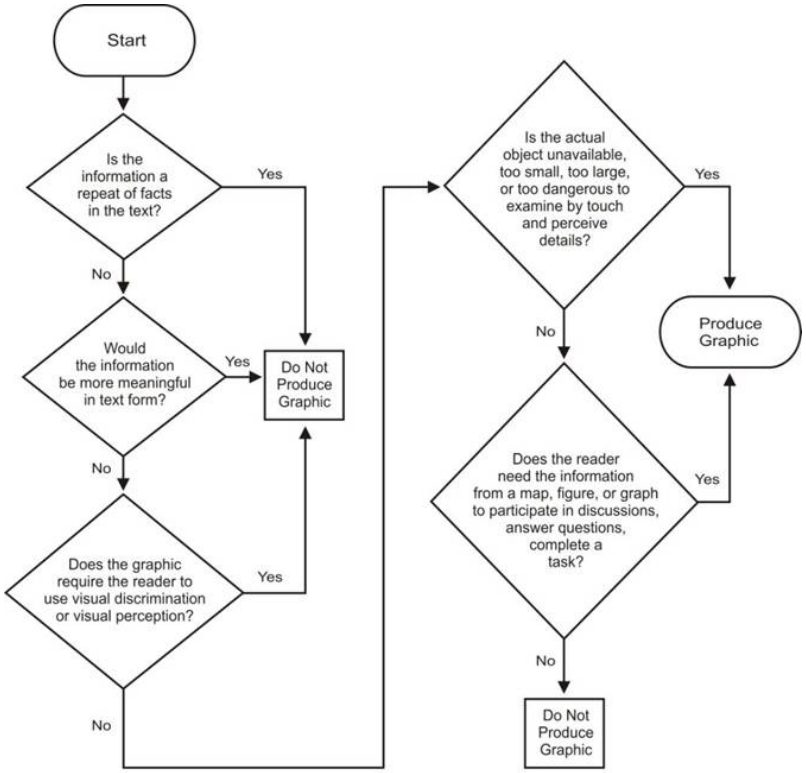


Figure 2.1: A decision tree dictating whether a tactile graph is appropriate [8]

milestone of the project was the modularity and flexibility allowed by the graph generation pipeline. As the approach is entirely digital, the proposed approach combats all aforementioned issues with an additional benefit of accessibility as well.

By converting the text to an intermediary language prior to tactile representation, the language could be processed through a text-to-speech pipeline. This would be critical for assistive technologies like screen readers or Apple's VoiceOver [2] software as information pertaining to the graphs could be relayed through a medium like English. As much of the content in today's world is absorbed through digital mediums like computers, phones, and

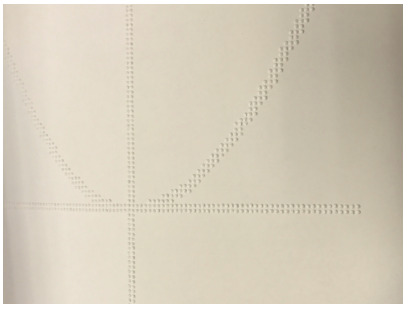


Figure 2.2: Visualization of graphs in braille using the BrailleNote Touch [14]

other electronic devices, having a digital solution for graph translation would be invaluable.

2.3 Existing Solutions

While there are existing solutions for individual graph generation, such as using Gaussian distributions for a histogram, there is no widespread solution for generating random charts of varying types. Many projects use techniques such as Gaussian distribution (section 3.3.1) for histogram generation and a sorted uniform distribution (section 3.3.1) for bar charts, but having a customizable approach to the entire process is hard to come by. This is especially true as most projects tend to pick a single visualization library and stick to it.

Chapter 3

Methodology

This chapter will discuss the various methods and algorithms used in this project in order to generate randomized graphs and their data. Each section discusses a certain generation process, as well as its applied use cases.

3.1 Midpoint Displacement

One of the first algorithms explored was Midpoint Displacement, also known as the Diamond-square Algorithm. As seen in Figure 3.1, this method is typically used for 2-D mountainous landscape generation.

3.1.1 Generation

Midpoint Displacement begins by simply initializing a straight line segment. After computing the segment's midpoint, it then displaces the y coordinate value by a "bounded random value" [1]. This process is then iterated over as many times as desired, reducing the bounded displacement value each time. This has the effect of creating twice as many

Figure 3.1: Example Midpoint Displacement Landscape Generation [1]

segments in every iteration. Successive steps of the Midpoint Displacement algorithm result in segments similar to the image shown in Figure 3.2.

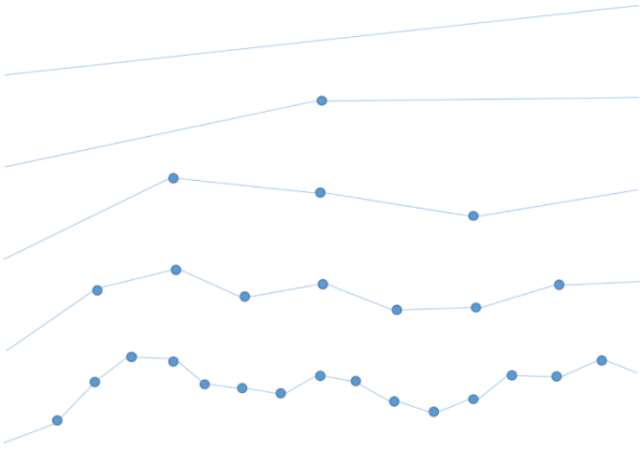
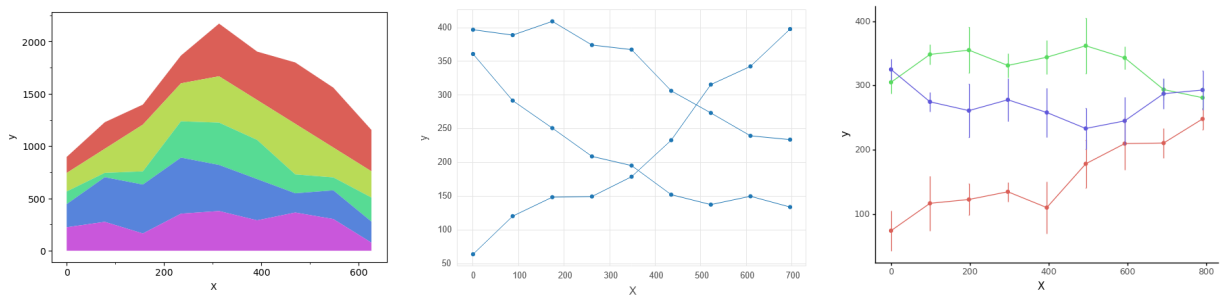


Figure 3.2: Iterations of Midpoint Displacement Algorithm [1]

3.1.2 Use Cases

Midpoint Displacement was applied for area charts, line graphs, and error bar charts. These were chosen because their correlation of data tends to be visually similar in terms



(a) Area Plot (b) Line Graph (c) Error Bar

Figure 3.3: Midpoint Displacement Chart Generation

of mountainous peaks and ranges.

To execute this, each graph was given randomly generated parameters such as the number of layers, number of iterations, vertical displacement and roughness (section 3.6). These parameters were then fed into a Midpoint Displacement algorithm that randomly generated X and y points for each line, or layer.

A key difference between these types of graphs is that the area charts required the y coordinates to be "stacked" on top of each other, such that no lines crossed over each other. This stacking effect is displayed in section Figure 3.3a. Moreover, line graphs required less "jagged peaks" in their display, and so the range of roughness was decreased, as well as the number of iterations. As seen in Figure 3.3b and Figure 3.3c, this resulted in smoother-looking correlations, with less points per line. Error bar graphs were formatted in a similar manner to line graphs, but with randomly-generated lengths for the error bars at each point. It is also worth nothing that line graphs and error bar graphs had a 50% chance of implementing Random Number Generation instead of just Midpoint Displacement, which will be further discussed in section 3.6.

3.2 Regression

Both Linear and Logarithmic Regression were implemented in order to generate a correlation effect in non-categorical graphs such as scatter plots and bubble plots.

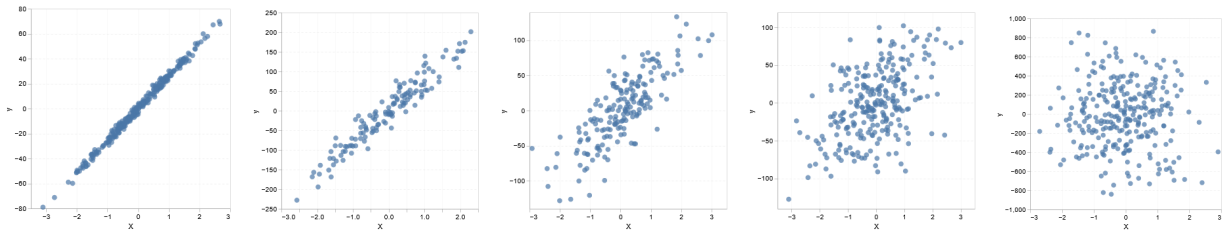
3.2.1 Generation

Linear Regression's data points visually form the shape of a line, which follows the algorithm $y = mx + b$, where m is the given slope and b is some given intercept. The generation of positive Linear Regression was implemented through the use of scikit-learn's `make_regression()` method, which returns a number of X and y values. Its input parameters of the number of samples, number of features, noise level, and tail strength were all randomly generated within a set range (section 3.6). Negative Linear Regression follows the same process, but all y values are negated.

Although the X values are still created with scikit-learn's `make_regression()` method, Logarithmic Regression requires that $x > 0$, and so all invalid data is filtered out and replaced. Furthermore, unlike Linear Regression, the creation of each y value is mapped to the logarithm of its matching X value. This results in an equation similar to $y = log(x)$. In this project, each calculation has a random logarithmic base, which ranges between two values specified in the generation's parameters (section 3.6). If the Logarithmic Regression results in a negative correlation, these y values will then be negated.

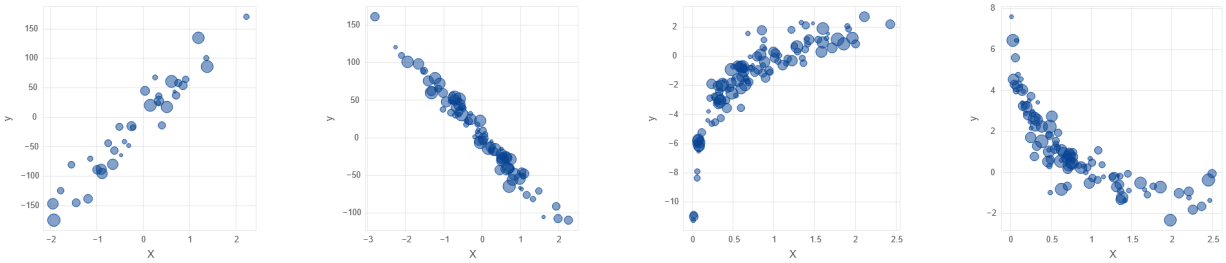
3.2.2 Use Cases

Linear Regression was utilized for generating both scatter plots and bubble plots. Various noise is randomly applied to each generation, resulting in the graphs ranging from seemingly straight lines to no visible correlation. This variation in noise is displayed through the scatter plots shown in Figure 3.4.



(a) $x = 2.09$ (b) $x = 25.18$ (c) $x = 29.72$ (d) $x = 40.17$ (e) $x = 355.77$

Figure 3.4: Noise in Positive Linear Regression - Scatter Plot



(a) Positive Linear (b) Negative Linear (c) Positive Log (d) Negative Log

Figure 3.5: Linear and Logarithmic Regression - Bubble Plot

In addition to Linear Regression, bubble plots have a 50% chance to follow Logarithmic Regression when being generated. This difference in pattern can be clearly seen in Figure 3.5, which displays the two types of regressions as bubble plots. Other than the inclusion of logarithms, an important distinction between bubble and scatter plots is that bubble plots also required a random generation of bubble size, which is discussed later in section 3.6.

3.3 Distribution Sampling

The process of sampling values from a provided statistical distribution is known as distribution sampling. The distributions discussed below are Continuous Probability Distributions, most of which are implemented using Python's NumPy library. These distributions are repeatedly sampled to create a set of datapoints based on an underlying distribution. In this context, the term "Continuous" refers to the "probabilities of the possible values of a continuous random variable", where each random variable has a range of possible values [15]. Furthermore, each type of Continuous Probability Distribution follows a Probability Density Function, which is used to calculate the probabilities of continuous random variables.

3.3.1 Generation

As many types of graphs utilize Continuous Probability Distributions for data generation, this section will discuss a wide variety of these distribution types. Specifically Gaussian Distribution, Log-normal Distribution, Gamma Distribution, Weibull Distribution, Uniform Distribution, and Multivariate Gaussian Distribution will be covered.

Gaussian Distribution

Gaussian Distribution, also termed Normal Distribution, is a symmetrical distribution. The correlation forms a shape that is often described as a "bell curve", which is a result of its centralized mean. With μ as the mean and σ as standard deviation, Gaussian Distribution follows the Probability Density Function shown below:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-1}{2}(\frac{x-\mu}{\sigma})^2}$$

The data generation for Gaussian Distribution was created through NumPy's `random.normal()` method, which takes input parameters for the mean and standard deviation of the distribution. These required values were randomly generated within a predetermined range for each graph (section 3.6).

Log-normal Distribution

Visually similar to Gaussian Distribution (section 3.3.1), Log-normal Distribution is identifiable by having a right-skewed bell curve, which results in a non-symmetrical shape. As the name suggests, a Log-normal Distribution is "a continuous distribution of random variable y whose natural logarithm is normally distributed" [13]. With μ as the mean and σ as standard deviation, Log-normal Distribution satisfies the following Probability Density Function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-1}{2}(\frac{lnx-\mu}{\sigma})^2}$$

Rather than using NumPy's `random.normal()` function, graphs following a Log-normal Distribution utilize `random.lognormal()` instead. The input parameters remain the same as the ones used for Gaussian Distribution (section 3.3.1).

Gamma Distribution

Gamma Distribution is a maximum entropy probability distribution that is skewed rightwards. It fits the Probability Density Function displayed below, with k as the shape parameter, θ as the scale parameter, and Γ as the gamma function, which can be defined as an "extension of the factorial function to real (and complex) numbers" [6]:

$$f(x) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-\frac{x}{\theta}}$$

Data for Gamma Distribution was generated through the use of NumPy's `random.gamma()` method, which received randomized values within a specified range for its required input parameters of shape and scale (section 3.6).

Uniform Distribution

A Uniform Distribution is one in which every result, within a certain maximum and minimum bounds, is equally likely. This results in a fairly constant-looking correlation, with no noticeable peaks or skews. With a as the minimum and b as the maximum such that $a \leq x \leq b$, Uniform Distribution fits the Probability Density Function displayed below:

$$f(x) = \frac{1}{b-a}$$

Generation for Uniform Distribution was performed with NumPy's `random.uniform()` method. The required input parameters of the minimum and maximum were randomly generated within a specified range (section 3.6).

Weibull Distribution

Related to exponential functions, Weibull Distribution takes k as the shape parameter and λ as the scale parameter in order to satisfy the following Probability Density Function:

$$f(x) = \begin{cases} \frac{k}{\lambda}(\frac{x}{\lambda})^{k-1} e^{-(\frac{x}{\lambda})^k}, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

The data generation process for Weibull Distribution involved using NumPy's `random.uniform()` method to initially generate a Uniform Distribution (section 3.3.1). This Uniform Distribution data, represented as d, was then placed into the following equation:

$$data = \lambda(-log(d))^{\frac{1}{k}}$$

As seen, this equation transforms the Uniform Distribution into a Weibull Distribution final data array through the use of NumPy's `log()` function, as well as the randomized input of the shape parameter k, and the scale parameter λ (section 3.6).

Multivariate Gaussian Distribution

Also named Multivariate Normal Distribution, Multivariate Gaussian Distribution place Gaussian Distributions (section 3.3.1) into a multi-dimensional format. This is clearly visualized in Figure 3.6.

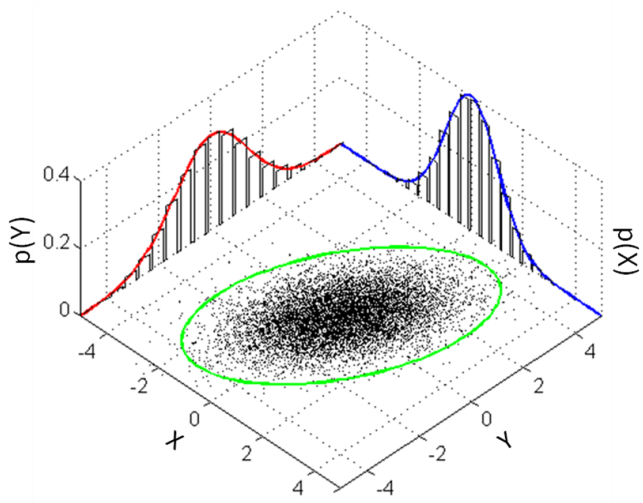
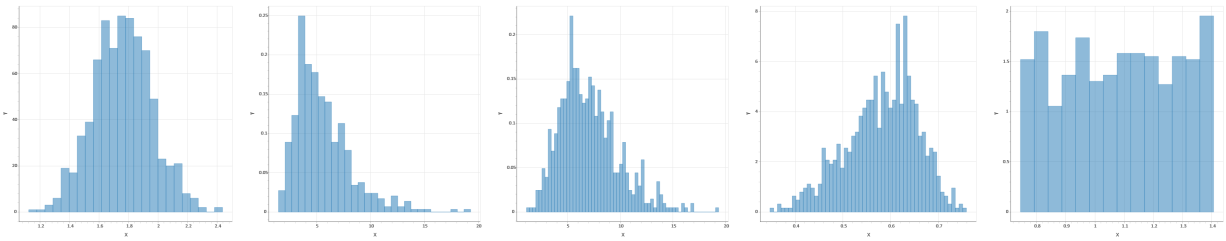


Figure 3.6: Multivariate Gaussian Distribution [16]

With k number of varieties, and μ as the mean, Multivariate Gaussian Distribution's Probability Distribution Function is displayed below:

$$f(x) = \frac{1}{(2\pi)^{\frac{k}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu)'\Sigma^{-1}(x-\mu)}$$



(a) Gaussian (b) Lognormal (c) Gamma (d) Weibull (e) Uniform

Figure 3.7: 2-D Continuous Probability Distributions - Histogram

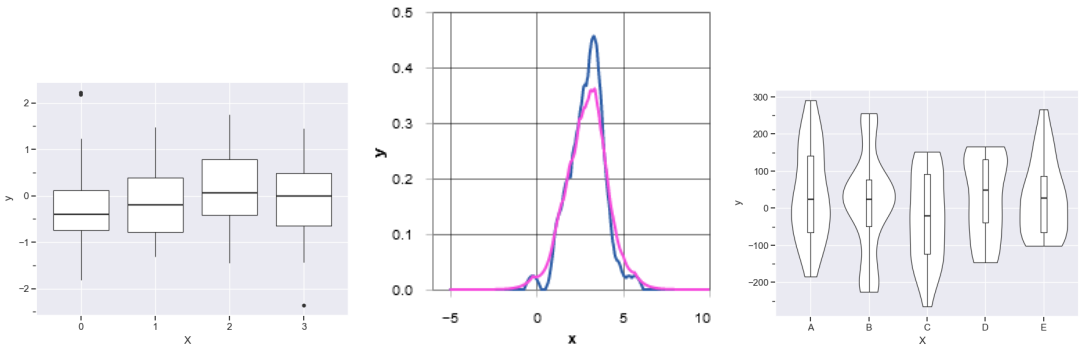
In this project, data for Multivariate Gaussian Distribution was generated through NumPy's `random.multivariate_normal()` function. The first required input of the mean was given through randomly generating a value within specified parameters (section 3.6). The second required input was a 2-D array represented a co-variance matrix, whose data was generated with the use of Uniform Distributions (section 3.3.1).

3.3.2 Use Cases

Histogram generation utilizes the greatest number of Continuous Probability Distributions. It has an equally random chance of using Gaussian, Lognormal, Gamma, Weibull and Uniform Distributions, all of which can be seen in Figure 3.7.

Gaussian Distribution was also applied for kernel density estimation, box and violin plots, which resulted in graphs such as the ones displayed in Figure 3.8.

The only graph type to use Multivariate Gaussian Distribution was contour plot, as it graphs a multi-dimensional surface, demonstrated in Figure 3.9.



(a) Box Plot (b) KDE Plot (c) Violin Plot

Figure 3.8: Gaussian Distributions

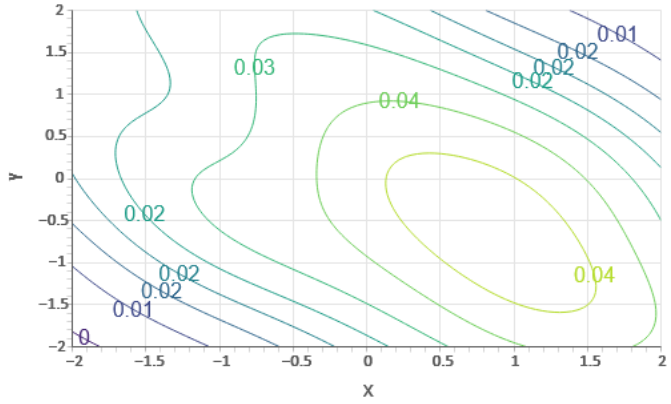


Figure 3.9: Multivariate Gaussian Distribution - Contour

3.4 Geometric Brownian Motion

Geometric Brownian Motion, also named Exponential Brownian Motion, was not implemented in this project for a variety of reasons. Not only was it difficult to implement and more predictable than other generation methods, but it also would only be used for line graphs. Nevertheless, if this project was to be expanded, then the implementation of methods such as Geometric Brownian Motion should be added. It is due to this logic that Geometric Brownian Motion's generation and possible use case shall continue to be discussed.

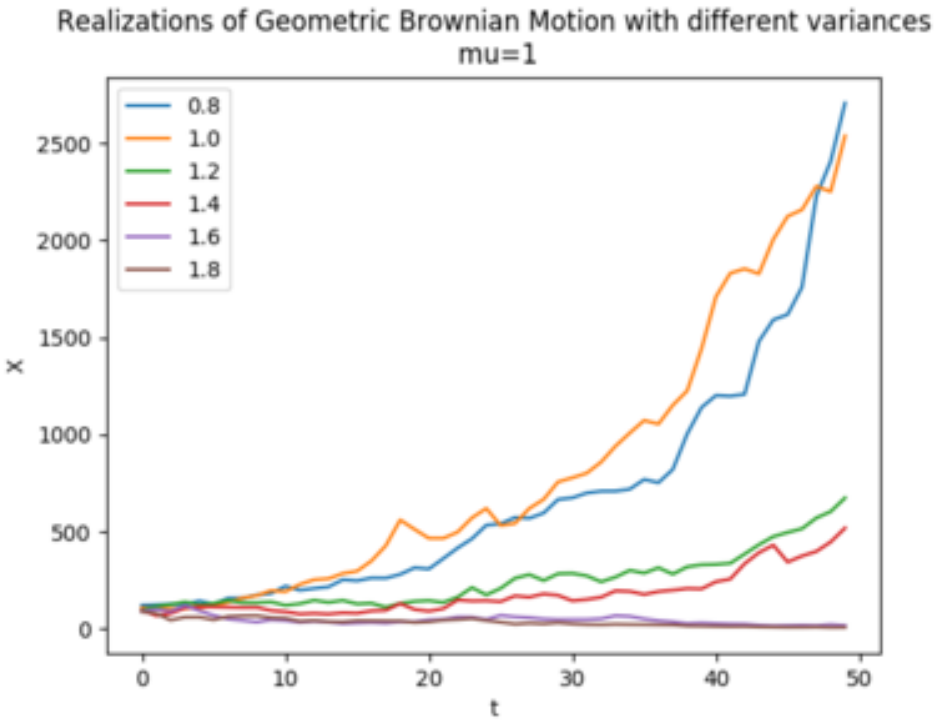


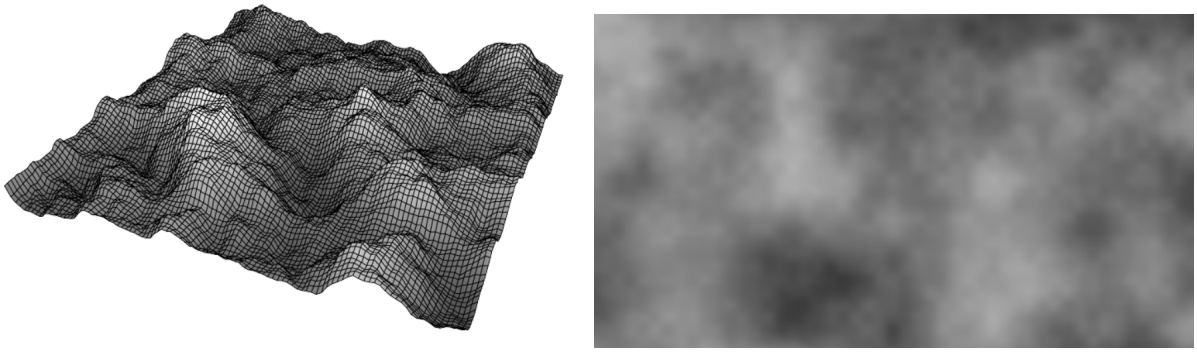
Figure 3.10: Geometric Brownian Motion - Line Graph [5]

3.4.1 Generation

Graphs that follow Geometric Brownian Motion typically start at an origin of zero and grow exponentially, with a slight visual appearance of roughness. Since Geometric Brownian Motion is a stochastic process with linear drift, its data follows a stochastic differential equation. This equation is displayed below, where W_t is the Brownian motion, μ is the drift percentage, and σ is the volatility percentage:

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

To generate data using this equation, W_t would be Normally Distributed (section 3.3.1),



(a) Procedurally Generated Terrain [21] (b) Perlin Noise Mapped to 2-D Texture [20]

Figure 3.11: Example Applications of Perlin Noise

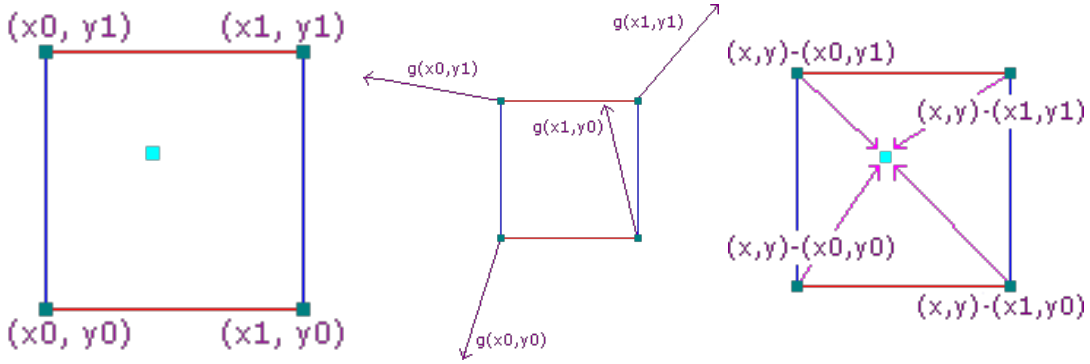
and the rest of the input would be randomly generated within specified ranges (section 3.6).

3.4.2 Use Cases

If Geometric Brownian Motion had been implemented then it would have been used as another correlation type for line graphs. This would have resulted in generations similar to the graph displayed in Figure 3.10.

3.5 Perlin Noise

Perlin Noise is an extremely powerful pseudo-random generation algorithm used for creating random patterns that appear more "organic" or lifelike in nature. Moreover, the algorithm is heavily used for procedurally generating content as the generated patterns contain patches, or gradients, of similar values which makes them appear to be only somewhat random. This makes it extremely effective for generating terrain (Figure 3.11a) as well as creating textures for objects such as cloud, marble, or fire, as demonstrated in Figure 3.11b.



(a) 2-D Unit Cube (b) Gradient Vectors (c) Distance Vectors

Figure 3.12: Generation Steps of Perlin Noise [3]

3.5.1 Generation

While Perlin Noise has a few variations, namely Improved Perlin Noise [19] and Simplex noise, the base function remains largely the same. In this particular case, Improved Perlin Noise was further examined and is the topic of discussion below. The summary below is largely influenced by the article authored by Adrian Biagioli [3].

The algorithm begins with a Perlin Noise function. This function takes in an x, y, and z coordinate as input and outputs a value between 0.0 and 1.0 [9]. To generate this value, the coordinates are first divided into unit cubes to "normalize" the coordinate within the desired dimension space. A 2-D representation of this can be found in Figure 3.12a. The 4 corner coordinates of this cube are then utilized to generate a pseudo-random gradient vector. Comparatively, if 3-D Perlin Noise is being generated, this cube would possess 8 unit coordinates rather than the 4 mentioned above. These gradient vectors, as seen in Figure 3.12b, will be used to calculate distance vectors from the provided point to the 4 corner points (Figure 3.12c).

Next, the dot product between each gradient vector and it's corresponding distance

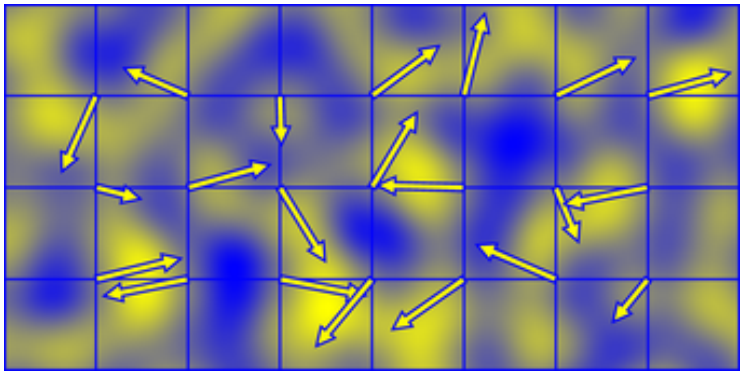


Figure 3.13: 2-D Representation of Perlin Noise Influence Values [3]

vector are computed, creating an influence value. This value is positive when it is in the direction of the gradient, negative when opposite facing, and has a value of 0 if both vectors are perpendicular (Figure 3.13). All influence values in the grid must then be interpolated to create a value much like a weighted average. One final step, known as easing, must be completed due to the linear interpolation, as the averaged values would otherwise look unnatural.

A fade function, or ease curve, is used on the averaged coordinate values to create a smoother transition between gradients Figure 3.14. By using this ease curve, the coordinates are changed more gradually when approaching integral coordinates, thus creating patterns that seem to flow together. The easing function used in the Improved Perlin Noise algorithm can be seen below, where t represents a given unit cube coordinate.

$$6t^5 - 15t^4 + 10t^3$$

3.5.2 Use Cases

In the context of graph generation, Perlin Noise could be used for generating area graphs as well as contour plots. The former was attempted to be implemented, however, in

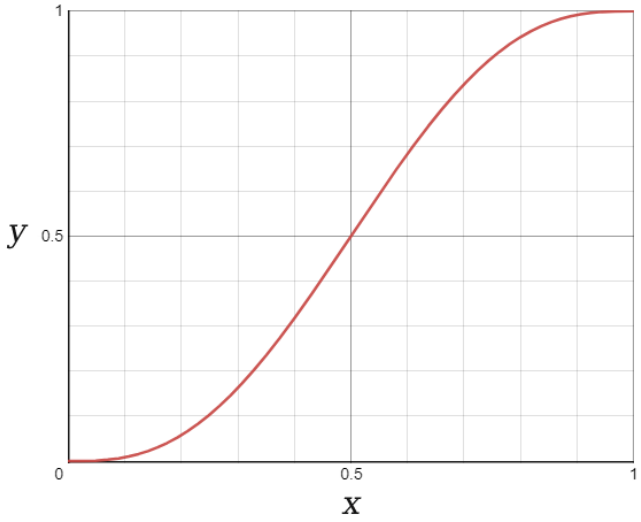


Figure 3.14: Improved Perlin Noise easing function

testing, the data did not provide examples visually similar to a standard contour plot. However, it is still believed that Perlin Noise could be used to generate random contour plots, as the following articles by Benjamin Hauer [10] and Amit Patel [18] seemingly do so, although further research must be conducted before reaching any final conclusions. As it is commonly used in terrain generation, for both 2-D and 3-D environments, it is no surprise that Perlin Noise is feasible for generating adequate area plots. Due to the aforementioned roadblocks, as well as stringent time constraints, Perlin Noise was not further researched for either contour or area graph generation.

Additional implementations such as CONREC [4], a subroutine for contouring a specified surface as a triangular mesh, and Simplex noise [9], an enhanced version of the original Perlin Noise algorithm, were also considered and subsequently discarded. The aforementioned CONREC algorithm was extremely hard to randomize and did not provide representative contour plots which is ultimately why it was rejected.

On the other hand, the Simplex noise algorithm does alleviate some of the problems with Perlin Noise, such as computational complexity and an inequal direction distribution, however, it was abandoned for the same reasons faced by Perlin Noise. While it is believed that CONREC and simplex noise could still be used for contour generation, similar to Perlin Noise, further research must be completed before reaching a conclusion.

3.6 Random Number Generation

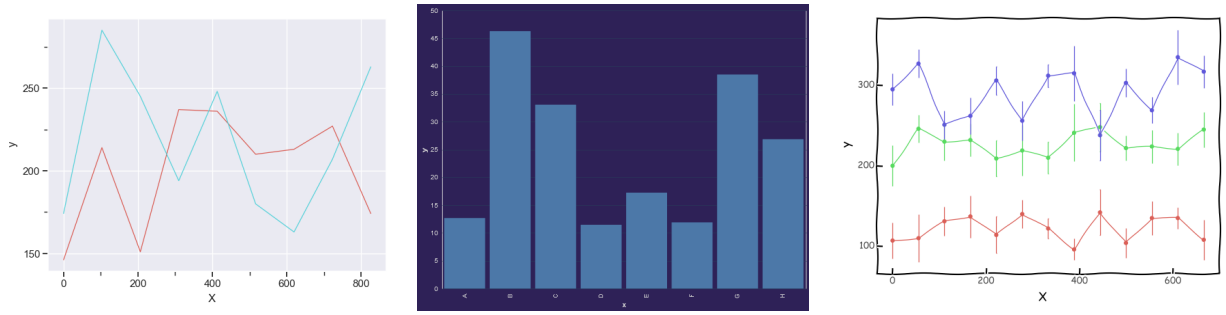
Random Number Generation allows for integers and floats to be randomly generated. This is useful for data generation, stylization, theming, and randomizing input parameters.

3.6.1 Generation

Random Number Generation produces a random value, or a range of data points with seemingly no pattern or correlation in a specified range. Depending on the type of variable desired, either NumPy's `random.randint()` or NumPy's `random.uniform()` method was applied for generation, which utilizes Uniform Distributions (section 3.3.1) to return random values. Since its input parameters require a smallest possible integer and a largest possible integer, these parameters were also randomly generated within a previously specified range.

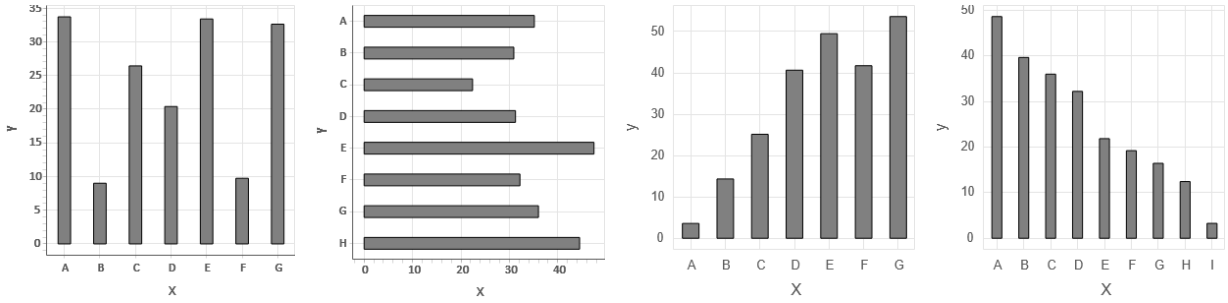
3.6.2 Use Cases

As Random Number Generation was heavily relied upon during this project, there were a large variety of factors and use cases it was responsible for. The first of which was simply to generate data with no correlation, which was utilized in line graphs, error bar graphs and bar charts. This data generation can be seen in Figure 3.15.



(a) Line Graph (b) Bar Chart (c) Error Bar

Figure 3.15: Random Number Graph Generation



(a) Random Vertical (b) Random Horizontal (c) Positive Vertical (d) Negative Vertical

Figure 3.16: Bar Chart Random Correlations

It is also worth noting that correlation for bar charts were generated by ordering the random data points. This resulted in the addition of positive and negative correlations, each with a random amount of noise and a random chance of displaying horizontally rather than vertically. These correlations are displayed in Figure 3.16.

In addition to data points, Random Number Generation was used to generate graph-specific input, such as graph orientation and number of layers, lines, boxes, or violins. Another example of this is graphs that required specific sizes per data point, such as bubble size or error bar length. Furthermore, any random input parameters needed for generation functions were created using Random Number Generation, with the ranges predetermined

by the user.

Lastly, Random Number Generation was used for theming and stylization purposes in order to further graph type variation. Themes were randomly picked per library, and examples of random stylization include line thickness, colour (RGB values ranging between 0 and 255), as well as marker sizes. More information on theming and stylization can be found in subsection 4.2.3.

Chapter 4

Procedure

The entire process from start to finish will be documented in the following chapter, alongside subsections for each step. To begin, a high-level overview of the generation pipeline will be discussed, followed by a more technical breakdown of what each of the three major stages entail.

To start, Figure 4.1 demonstrates the system architecture, which is broken down into the environment setup, generation pipeline, and regeneration pipeline steps. Since the first and third components play a lesser role throughout the project, they will only be discussed briefly, while the second component will be the primary focus of this chapter.

4.1 Environment Setup

The environment setup stage is tasked with setting up the development environment as well as all required dependencies and packages. An essential part of this step is that the target system must contain both Anaconda as well as some sort of interactive Python command shell such as iPython, Jupyter, or Google Colab. Jupyter was used for the development

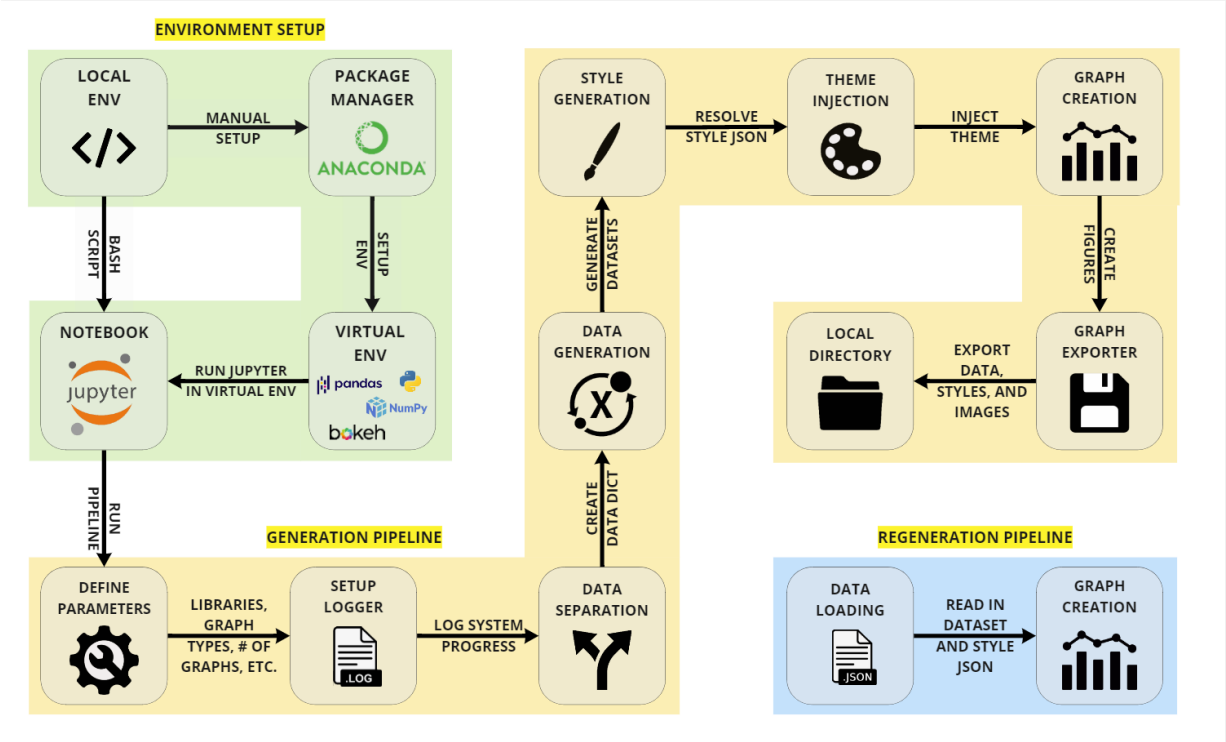


Figure 4.1: Graph generation system architecture

of this project, which is the interactive Python environment that will be described going forward.

Once all the required dependencies are installed, the first step is to utilize Anaconda, the de facto standard for Python package distribution. It will be used to create a virtual environment in the target location. This environment will house all of the required packages in an environment separate from the system and any other projects. This is largely to avoid compatibility issues between sharing libraries, as well as promote reproducible development environments. The setup of this virtual environment will also include the download and installation of required software such as Python, NumPy, pandas, Bokeh, Altair, and plotnine amongst others. The file containing all required dependencies as well as their specified versions can be found at `setup/environment.yml`. Upon successful in-

stallation, the final step is to launch the Jupyter environment, also known as an interactive Python notebook. This notebook will be used to execute the code in a linear fashion while maintaining variables between code blocks and allow for inline documentation through markdown text styling.

A custom Bash script was made to automate the entire environment setup using as a little as a standard computer terminal. The automation takes care of everything from the initial environment creation up until running the Jupyter notebook, which will need to be executed by a user. It is worth mentioning that the script (`setup/setup.sh`) is compatible with both MacOS and Unix based devices, however, it is not recommended for use on Windows-based machines.

4.2 Generation Pipeline

The generation pipeline was the core development that was undertaken, and acts as the backbone for the entire project. It includes everything from generating a list of how many graphs per library are required to the exporting of individual graphs.

As seen in Figure 4.4, the generation process contains a multitude of steps, however, only a few of the critical steps will be brushed on in the coming subsections. Namely, the data separation, data generation, style generation, graph creation, and graph exportation stages will be covered, as they play the most crucial roles and are less self explanatory than others.

4.2.1 Data Separation

Data separation begins with the pipeline's hyperparameters. These include the number of graphs to generate, as well as the graph types and libraries to generate from. An additional exclusion parameter is specified to ignore any graph-library combinations. One such

example would be Altair-based contour plots, as they are not supported by the library's rendering engine.

The data separation step then utilizes these hyperparameters to generate an occurrence dictionary, which represents the number of graphs that should be generated for each graph-library pair. There are optional flags for forcing a randomized distribution between graph types as well as libraries, however, this approach is not recommended as certain combinations may not be selected, thus generating a dataset of lower variance and potentially higher bias.

Graph Type	**Library (Equal)**			**Library (Random)**		
	Bokeh	Altair	plotnine	Bokeh	Altair	plotnine
Area Graph	5	5	4	1	4	9
Bar Chart	5	5	4	0	4	1
Box Plot	5	4	4	6	0	3
Bubble Plot	5	5	4	0	0	0
Contour Plot	5	5	4	0	0	0
Error Bar	5	4	4	13	1	5
Histogram	5	5	4	1	4	1
KDE Plot	5	5	4	0	0	5
Line Graph	5	4	4	52	7	33
Scatter Plot	5	5	4	0	0	0
Violin Plot	5	4	4	0	0	0

Table 4.1: Graph-type to library occurrences (150) for equal and random distributions

4.2.2 Data Generation

The data generation process is arguably the most important process of the entire project. Acting as a middleware service between the data separation and creation steps, it takes the aforementioned occurrence dictionary as input and generates a new dictionary containing the generated graph objects, binning each based on graph type. Each bin stores an array of graph objects with their respective IDs, specified creation library, and generated data.

The data produced in this step typically include a set of X and y data points, however, certain graph-types may require information such as the type of distribution, number of layers, bubble sizes, z values, and correlation type (Figure 4.2). These values are specified based on graph type and the generation for which can be found in the `generators` directory. It should be noted that each Python file (`.py`) in the `generators` is named based on graph type and contains a required `generate_data()` function responsible for the data generation. Methodologies behind the various generation techniques are explained thoroughly in chapter 3. It should also be noted that data of any type and quantity can be provided to the generated object, as it will be used later in the creation and exportation steps.

After sequentially generating the data for each individual graph, this master dictionary is then supplied to the "stylizer" for both style generation and theme injection.

4.2.3 Style Generation

The primary role of the stylization module is to generate styles for a given library-graph pair. Taking in both the graph type and library type, the styles are generated on the fly for each individual graph in the form of a key-value dictionary. An additional input parameter is specified containing the number of repeated styles, if any, that need to be generated. This is beneficial for graphs that require multiple styles based on the size and shape of the

```
{
    "X": [
        "A",
        "B",
        ...
    ],
    "y": [
        6.253763991874621,
        13.491181938914046,
        12.86460662847583,
        ...
    ],
    "is_vertical": false,
    "correlation": "positive"
}
```

Figure 4.2: Example of a bar chart data generation object

dataset. For instance, area plots typically contain a multitude of layers and may require a different colour for each layer.

These style dictionaries are generated by reading in the `JSON` document from the `styles` directory and parsing the corresponding style document. Each file is named based on graph type (i.e. `scatter.json`) and contains a `default` property as well an object property for each respective visualization library (i.e. `bokeh`, `altair`, etc.). These properties contain objects representing the styles that should be given to each or all of the functions responsible for plotting the graphs. Specifically, the values contain a structure similar to Figure 4.3, where the key denotes the property name and the value represents an object containing type and value fields. The type field mentioned above specifies how the value should be resolved and the value field is used as input for that resolution. These property objects are then used by the style resolver, whose details will be covered in the following paragraph.

These properties are then passed to a style resolver, which resolves styles from the input object to usable values. For instance, this may turn a colour field into a usable hex color

```
{
    "default": {
        "line_color": {
            "type": "color",
            "repeat": true
        }
    },
    "bokeh": {
        "line_thickness": {
            "type": "range",
            "value": [0.4, 0.8],
            "float": true
        }
    },
    ...
}
```

Figure 4.3: Example of a line graph's stylization JSON file

code, randomly generate a value between a specified range, or select a single item from a list of options. In addition to the above, the style resolver supports generating random booleans, reading in `JSON` file paths, and returning static values. These resolutions are defined in a dictionary that maps style types to their respective functions. Any additional parameters the style object possesses will be passed down to the corresponding resolve function. This has many uses cases such as indicating whether a field should be repeated, or whether a value should return a float.

After resolving each style to the correct value, a random theme is selected based on library type. Both the chosen theme's name and newly generated styles are attached to the graph object currently being iterated over. This completed data object is then supplied to the graph creation step for image production.

4.2.4 Graph Creation

The graph creation takes in the previously generated data and styles and outputs a graph using the corresponding visualization library. The architecture for this is not unlike the graph generation process, where each Python file (`.py`) in the `creators` directory contains a function for every graphing library (i.e. `create_bokeh_graph()`, `create_altair_graph()`, etc.). Prior to graph creation, any initialization functions, known as pre-creation hooks, are executed. Following, the theme is set for the visualization libraries in a process known as theme injection.

This process involves retrieving the theme name from the generated data and reading in the corresponding theme file which is then injected into the generated graph. Should the visualization library require this to occur after graph generation, a post-creation hook is provided. The injection is the process of injecting usable theme data into the graph for stylization purposes.

More technically, each file in the `libraries` directory possesses a name corresponding to the desired visualization library, and contains a single Python class. These `Library` classes contain the functions for setup, pre-creation, post-creation, theme setting, and graph exportation, with the latter being discussed in subsection 4.2.5.

4.2.5 Graph Exportation

Graph exportation is the process of exporting the data, styles, and image of a particular graph to local storage. The data and styles are stored as separate `JSON` files in the `output/data` and `output/styles` directories respectively. The images, however, are stored in Portable Network Graphics (`.PNG`) format at the `output/images` directory. All three of the graph's output files are saved using the same `type_library_id` naming convention, where type and library represent the graph type and visualization library respectively. The graph ID on the other hand, represents the current number in the sequence of graphs with

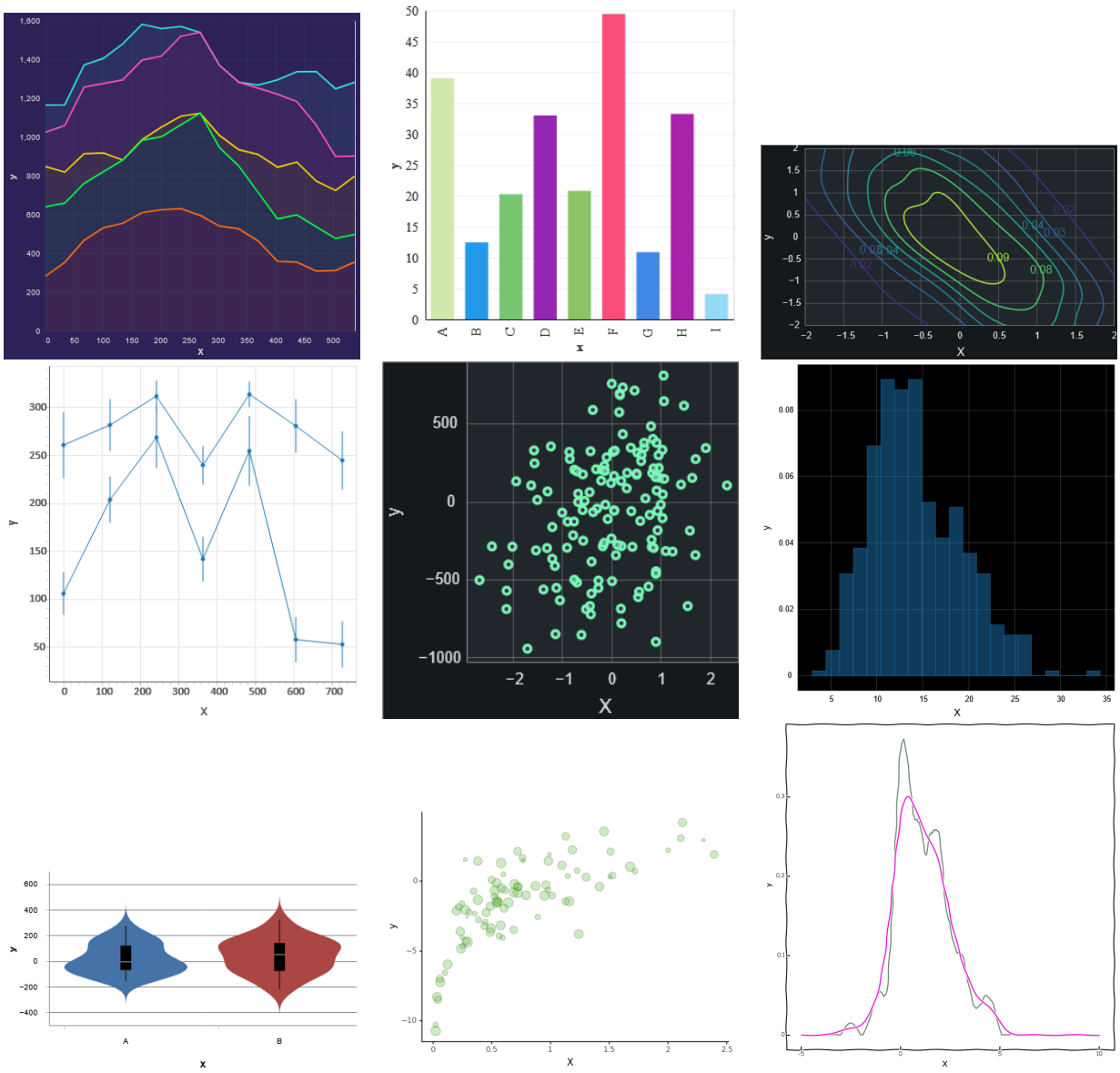


Figure 4.4: Various plots with random styles and themes generated using the pipeline

identical graph-types and libraries. For instance, the naming convention may be seen as `area_bokeh_0` followed by `area_bokeh_1` and `area_altair_0`.

Since the graph content can be exported, the data and style objects need to be serialized into a JSON-compatible format. This serialization process involves turning values such as NumPy arrays or pandas DataFrames into standard Python lists. The provided values are serialized based on type, and recursion is used to serialized nested values such as 2-D or 3-D arrays. Once serialized, the data, styles, and images are ready to be exported. Graph exportation is a crucial aspect of the project as it allows for graph regeneration, which is essential for determining model correctness and will be covered in section 4.3.

4.3 Regeneration Pipeline

The regeneration pipeline is tasked with regenerating graphs from two input sources, namely the data and styles objects, and is a critical aspect of the project. As the pipeline above will be involved in creating input data for a Generative Adversarial Network (GAN), it is necessary to compare the images from both the original pipeline, as well as the GAN, in order to assess network accuracy.

4.3.1 Chart Regeneration

The chart regeneration process was designed to be run as a standalone process and as such, begins by running any required library setup. Following, the `data` and `styles` folders in the `ingestion` directory are intersected to find all valid graphs. Once a valid graph list is made, the same code used for graph creation can be run on the read-in data and style objects. However, the data must be de-serialized as it is being read-in and not exported like mentioned in subsection 4.2.5. This involves reversing the previous serialization procedure by converting values from their basic Python classes to more appropriate instances, such as a primitive list object to a NumPy array.

Once de-serialization is complete, the data is ready to be used for graph creation. This involves passing the content to the graph creation process one at a time, just like the original creation process. During this graph regeneration, the images are outputted to the `ingestion/images` directory in PNG format, using the same filename as the inputted data and style files.

Chapter 5

Design Decisions

5.1 Environment

The project environment is run using a Jupyter Notebook and Anaconda environment. The decision to use an Anaconda virtual environment was an obvious choice for many reasons. To begin, it allows for extreme portability as the pipeline can be re-created in almost any environment regardless of operating system, device hardware, or system architecture. Secondly, it isolates the installed dependencies to a separate environment to avoid version conflicts and any issues that may arise due to shared dependencies. Lastly, Anaconda allows specific versions of libraries to be installed, providing the maximum level of reproducibility available.

Jupyter Notebook on the other hand is an interactive Python environment that allows markdown text, persistent variables, and a multitude of other features all within a single document. It is a necessity for data science and machine learning projects due to its simplicity and ease of use. This easy-to-use simple approach was a main reason why it was selected for this project. Executing the code in separate cells with supporting documentation helps provide a sense of modularity and linearity to the pipeline, allowing

for the graph generation process to be easy visualized.

5.2 Modularity

The concept of modularity was heavily considered when developing this project. In order to support a multitude of graph types and libraries, modularity was essential. This principle is enforced throughout various facets of the code base, which can be seen by the architecture design and file structure. For instance, the data generation for each graph type is entirely independent of one another and resides in their own appropriately named file (i.e. `area.py`, `bar.py`, etc.). This pattern is seen throughout the generation, stylization, creation, and theming steps, as well as within the visualization library classes.

5.2.1 Dynamic Modules

As mentioned in subsection 4.2.1, all aspects of pipeline utilize the graph types and libraries specified in the hyperparameters. This virtually eliminates any hard-coded content, thus creating a dynamic robust system. For instance, the graph generation process uses graph types to dynamically load in the corresponding file and run the data generation code. This is made possible by providing all the generation functions with the same name. Similar to these functions, the same naming convention applies for stylization, as the style objects are loaded in based on file name. In addition, this concept further applies to creation, theming, and the visualization libraries, all of which will be covered below.

The same modularity that is seen in the data generation process above is also applied to style generation, as the styles for each graph are independent and named accordingly (`box.json`, `violin.json`, etc.). Graph creation is in a similar vein, however, each file includes a creation function for the available visualization libraries. These functions are executed dynamically, based on the initial library specified. In terms of libraries, each library is encapsulated in a custom Python class that inherits the base Library class. This is done in

order to isolate each library, while still enforcing required functionality such as pre-creation and post-creation hooks, theme setting, and graph exportation. These classes are loaded in dynamically based on file name (i.e. `libraries/bokeh.py`, `libraries/altair.py`, etc.) which, like the above, is based on the initial library parameter. The concept of theming follows a similar principal, as each theme is defined in the `themes` directory in a sub-folder named after the corresponding visualization library (i.e. `themes/bokeh`, `themes/altair`, etc.). All themes present in the code base are located separately, based on library name, and can be found in either JSON (`.json`) or Python (`.py`) format accordingly.

5.2.2 Graph Regeneration

Modularity played an incredible role in the aspect of graph regeneration. As the processes were heavily decoupled, the creation step could be entirely reused by simply reading in a set of data and style files. Since all of the values, styles, and themes are utilized within the graph creation module, the entire process becomes extremely easy to maintain and leverage, as explained in subsection 4.3.1.

5.3 File Structure

Diving deeper into the file structure, all setup files and scripts can be found in the `setup` directory along with any logs in the `logs` directory. On the other hand, the exportation process outputs data, styles, and images all in their own folders within the `output` directory. The data and styles used for ingestion, as well as the regenerated images are stored elsewhere in the `ingestion` directory.

If any functionality needs to be re-used throughout the application, a `utils` directory exists with files corresponding to each component (i.e. `generators.py`, `styles.py`, `creators.py`, etc.). This creates a very simplistic and scalable file structure that is well-suited for future maintenance.

5.4 Future Support

The aspect of supporting additional libraries and graph types was a major concern when developing this project. The goal was to create a modular system with very little required maintenance all while minimizing the amount of work needed for future implementation. This is believed to be achieved as the current implementation system is extremely straightforward and flexible.

The process of supporting additional graph types and libraries is relatively simple, and the corresponding steps will be documented below. In order to implement an additional graph type, the following conditions must be met:

- The created generation file must meet the following conditions
  - Contains a `generate_data()` function
  - Named `graph_type.py` and stored in the `generators` directory
- The created stylization file must meet the following conditions
  - Be in JSON format
  - Contains a default property
  - Contains a property named after each visualization library
  - Named `graph_type.json` and stored in the `styles` directory
- The created creation file must meet the following conditions
  - Contains a `create_library_graph()` function for each graphing library
  - Named `graph_type.py` and stored in the `creators` directory

A different series of steps must be followed for supporting an additional library, which can be found below:

- The `libraries` hyperparameter must contain the library in the generation notebook
- The created library class must meet the following conditions
  - A Python file under the library's name must be made in the `libraries` directory
  - Inheritance of the base `Library` class from `libraries/library.py`
  - Given the name `Library`
  - Have all required methods implemented
    - A method is classified as required if it raises an `NotImplementedError`
- All creation files must implement multiple library creation functions
  - Functions must be named `create_library_graph()`
    - Where `library` is replaced by the specified library name
- Any themes must be stored in the `themes/library` directory

5.5 Exportation

A crucial design decision was how to store the generated data. By storing the data in a structure like a pandas DataFrame, all rows would be required to have the same length. This would result in countless null values as certain properties such as `is_vertical` or `correlation` would not apply to a majority of the graphs. Not to mention, storing the associated styles alongside the data would have been a significant roadblock.

The solution chosen for this problem was to serialize the data into separate JSON files and store them on a per-graph basis. This allows graph files to be easily imported and exported, while still maintaining maximum flexibility. This customization of what data and styles are stored has allowed graph regeneration to become an incredibly simple process.

5.6 Stylization & Theming

Each visualization library formats graphs differently. As they all take in distinct parameters and values, there was no uniform approach amongst any visualization libraries. Tackling this problem required careful consideration, as the styles needed to be applied individually while still having a shared pool for common elements like colour, and thickness. The idea was to create a separate JSON file based on each of the graph types. By containing a separate object for each visualization library, as well as a default object, this semi-modular approach was able to seamlessly incorporate flexibility and re-usability. Theming was designed with the same ideologies in mind, for instance, having separate JSON or Python files for each theme promotes modularity and flexibility. Lastly, the decision to remove arbitrary headings, legends, toolbars, and any additional white space was done to avoid any bias that might be introduced.

Chapter 6

Future Work

In terms of future work, the chart generation pipeline will be utilized to generate the training data of another system. On top of that, the chart regeneration will be used for determining the correctness of the aforementioned network. However, aside from future use-cases, there are still a few tasks that were unable to be completed, largely due to time constraints.

6.1 Data Modifiers & Multipliers

Virtually all of the data generation produces values that are within a set range. When generated in vast quantities, the values will lack variance and appear to be repetitive. This can result in poor performance for succeeding systems such as a Generative Adversarial Network. The way to combat this is quite simple in nature, but was unable to be implemented as the process of manually testing all graph-types combinations is quite time consuming. Furthermore, the possible maintenance required to initially fix all existing graph creation code should not be undermined.

Following the data generation step, data modifiers and multipliers could be used to

alter the data in one of two ways. The first is an offset which adds a constant value to all applicable data points. For example, this could involve adding a singular value to all X values in the dataset. The second method involves transforming the data points by a modifier. This would entail multiplying the data points by a constant value. Additionally, the values could be passed into exponent or logarithmic functions, however, many edge cases must be covered in order to preserve the underlying distribution of the generated data.

The values for the offset and multiplication modifiers are intended to be random, as to provide greater variability in the graph generation. On a more technical level, this code would reside in the data generation step (subsection 4.2.2) and could be as simple as adding a function to take in the data after generation is complete.

6.2 Additional Theming

Another aspect that was unable to be completed was the implementation of additional Bokeh themes. Both Altair and plotnine contain a sequence of 10 unique themes, while Bokeh only possesses 6. The main concern with having less stylization options is having reduced uniqueness and variability when this application is used on a larger scale. Provided with more time, this task could be easily completed by simply implementing additional themes.

6.3 Mini-Batching

The main premise of mini-batching is to separate the dataset into smaller batches for further calculation. The benefit of applying mini-batching in this project would be avoiding the hardware limitations of the system executing the pipeline. As loading in upwards of 30,000 images may result in out of memory errors, having smaller batches for the garbage

collection system to dispose of images that have already been generated would be extremely beneficial. The reason this was not implemented is simply due to time constraints and is definitely something worth further examining.

6.4 Multimodal Histograms

The final aspect in terms of future work is to allow for the generation of multimodal histograms. This could be done by splicing together two or more separate distributions. A random number would be used to determine how many values should be discarded when connecting the tail of one distribution to the head of another. Similar to how area and line plots contain multiple layers, adding multimodal histograms would provide a higher degree of variability.

References

[1] Juan Gallostra Acín. Landscape generation using midpoint displacement, Dec 2016.

[2] Voiceover, Nov 2020.

[3] Adrian Biagioli. Understanding perlin noise, Aug 2014.

[4] Paul Bourke. A contouring subroutine. *Byte*, 12:143–150, 06 1987.

[5] Geometric brownian motion, Oct 2021.

[6] Stephanie Glen. Gamma distribution: Definition, pdf, finding in excel, Jan 2014.

[7] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[8] Guidelines and standards for tactile graphics, 2010.

[9] Stefan Gustavson. Simplex noise demystified, 01 2005.

[10] Benjamin Hauer. Finding perlin contours, Feb 2019.

[11] Kai Hu, Wentong Liao, Michael Ying Yang, and Bodo Rosenhahn. Text to image generation with semantic-spatial aware gan, 2021.

[12] Deborah Kendrick. The braillenote touch plus: Competitive technology with a nostalgic feel, Mar 2020.

[13] Robert Kissell and Jim Poserina. Chapter 4 - advanced math and statistics. In Robert Kissell and Jim Poserina, editors, *Optimal Sports Math, Statistics, and Fantasy*, pages 103–135. Academic Press, 2017.

[14] Tracy Lee. Tips on teaching graphing using the braille note touch, Sep 2018.

[15] Continuous and discrete probability distributions.

[16] Multivariate normal distribution, Mar 2013.

[17] Kelly O'Sullivan, Stephen Alvey, and Robert Coelius. *An affordable, refreshable Braille tablet that relies on microfluidics*. University of Michigan Engineering, Dec 2015.

[18] Amit Patel, Jul 2015.

[19] Ken Perlin. Improving noise. *Proceedings of the 29th annual conference on Computer graphics and interactive techniques - SIGGRAPH '02*, 21:681, 07 2002.

[20] Perlin noise rescaled and added into itself to create fractal noise, Apr 2007.

[21] Jean-Colas Prunier. Perlin noise terrain mesh, Feb 2017.

Appendices

Appendix A

Graph Generation

A.1 Data Generation

```
generated_graphs = deepcopy(graphs)
graph_type_id, current_graph_type = 0, None
for (graph_type, library), num_occurences in occurences.items():
    if (current_graph_type != graph_type):
        graph_type_id = 0
        current_graph_type = graph_type

    graphs_list = generated_graphs[graph_type].setdefault('graphs', [])
    for _ in range(num_occurences):
        data = generate_data(graph_type)
        graphs_list.append({
            'id': graph_type_id,
            'library': library,
            'data': data,
        })
        graph_type_id += 1
```

A.2 Data Stylization

```
for (graph_type, graph_object) in generated_graphs.items():
    for graph_content in graph_object['graphs']:
        library, num_repeats = graph_content['library'], graph_content['data'].
            get('num_repeats', 1)
        graph_content['styles'] = generate_styles(graph_type, library,
            num_repeats)
        graph_content['data'].pop('num_repeats', None)
```

A.3 Graph Creation & Graph Exportation

```
for (graph_type, graph_object) in generated_graphs.items():
    for graph_content in graph_object['graphs']:
        library = graph_content['library']
        graph = create_graph(graph_type, library, graph_content)

        id = graph_content['id']
        file_name = '{graph_type}_{library}_{id}'.format(graph_type=graph_type,
            library=library, id=id)

        export_graph_data(graph_content['data'], 'output/{path}/{file_name}.{
            file_type}'
            .format(file_name=file_name, path='data', file_type='json'))
        export_graph_styles(graph_content['styles'], 'output/{path}/{file_name
            }.{file_type}'
            .format(file_name=file_name, path='styles', file_type='json'))
        export_graph_image(graph, library, 'output/{path}/{file_name}.{file_type
            }'
            .format(file_name=file_name, path='images', file_type='png'))
```

A.4 Graph Regeneration

```
for graph_filepath in valid_input_graphs:
  graph_filename = graph_filepath.split('.', maxsplit=1)[0]
  graph_type, library, id = graph_filename.split('_', maxsplit=3)
  graph_content = {}

  for folder in input_folders:
    with open('{dir}/{folder}/{file_name}'.format(dir=INGESTION_DIR, folder=
        folder, file_name=graph_filepath)) as f:
      graph_content[folder] = convert_from_serializable(json.load(f))

  graph = create_graph(graph_type, library, graph_content)
  export_graph_image(graph, library, '{dir}/{path}/{file_name}.{file_type}'
    .format(dir=INGESTION_DIR, file_name=graph_filename, path='images',
        file_type='png'))
```

Appendix B

Graph Regeneration

B.1 Graph Regeneration

```
for graph_filepath in valid_input_graphs:
  graph_filename = graph_filepath.split('.', maxsplit=1)[0]
  graph_type, library, id = graph_filename.split('_', maxsplit=3)
  graph_content = {}

  for folder in input_folders:
    with open('{dir}/{folder}/{file_name}'.format(dir=INGESTION_DIR, folder=
        folder, file_name=graph_filepath)) as f:
      graph_content[folder] = convert_from_serializable(json.load(f))

  graph = create_graph(graph_type, library, graph_content)
  export_graph_image(graph, library, '{dir}/{path}/{file_name}.{file_type}'
    .format(dir=INGESTION_DIR, file_name=graph_filename, path='images',
        file_type='png'))
```

Appendix C

Generation Output

C.1 Example Data Object

The following appendix entry showcases a data object generated for a bar graph.


```
{
   "X": [
      "A",
      "B",
      "C",
      "D",
      "E",
      "F",
      "G"
   ],
   "y": [
      40.30861110084393,
      39.52086715791276,
      23.612520709207068,
```

```
        16.56223510442039,
        14.040791444134694,
        11.77581894319021,
        0.13129366751803193
    ],
    "is_vertical": false,
    "correlation": "negative"
}
```


C.2 Example Style Object

The following appendix entry showcases a style object generated for an Altair error bar graph.


```
{
    "width": 400,
    "height": 400,
    "color": [
        "#213242",
        "#e911b4"
    ],
    "line_thickness": 1.4979890080751266,
    "marker_type": "dot",
    "theme": "excel_theme.json"
}
```