Using Heatmap Visualizations to Explore Pseudorandom Number Generators

Christopher Thomas

Honors College Thesis, May 2024

Departmental Honors in Computer Science

Faculty Thesis Advisor: Jane H. DeBlois PhD.

University of Massachusetts Boston

Abstract

Pseudorandom number generators produce a seemingly random sequence of values by making use of a deterministic algorithm to provide the illusion of randomness while being repeatable. Of the many methods of producing pseudorandom values, there is much variance in the degree of 'randomness' produced by different deterministic algorithms. Data visualization methods such as heatmaps provide an outlet for producing a visually intuitive way to give the user a better understanding of different pseudorandom number generation algorithms and their potential shortcomings. Heatmaps that track the distribution of values generated by a specific PRNG algorithms over multiple iterations or those that track the values produced from a batch of unique seeds can be used to provide a visual method of assessing the quirks of a given PRNG algorithm. Through a variety of methods of both producing pseudorandom numbers and visualizing them, there is ample room to explore PRNG's, the interesting visuals they can produce, and what those visuals imply about the algorithm producing the data.

Introduction

The issue of generating random numbers has been prevalent since the dawn of computing. Randomness, or the illusion of randomness, is a required aspect of many computer-based applications such as cryptography, where random values are required to create a level of unpredictability for security purposes. Also, computer-generated randomness is a foundational aspect of creating and testing simulations, as they allow the simulated system to account for the variability of the physical world [13]. The process of computer-based random number generation can be performed via either hardware or software. A Hardware Random-Number Generator, or HRNG, uses some aspect of the natural world that produces a degree of random noise, such as

measuring the degree of decay of a nuclear element over some time period [1]. Anything in the physical world that can be used as a source of entropy can be the root of a Hardware Random-Number Generator. Often, sources of physical entropy that are more easily accessible to the operating system are used, such as the timing of keyboard strokes and mouse movements [2]. Early methods of computer-generated random numbers, such as that seen in the Ferranti Mark 1, used electrical noise as their form of physical entropy to produce a series of 20 random bits [3]. The glaring issue with the electrical noise or physical entropy-based approach to computer-generated randomness comes with the reality that we are often not actually seeking true randomness in the computing space. To be effective for programmers and users alike, randomness must be repeatable and controllable, hence the 'pseudo' aspect of the pseudorandom number generators that will be discussed.

Pseudorandom number generators, or PRNG's, attempt to mimic the 'true randomness' of hardware random number generators using a deterministic algorithm instead of a physical source of entropy. In practical applications or situations where true unpredictability is required, such as cryptography, a random number generator would be comprised of a hardware-based generator to produce the input seed which is then fed into a pseudorandom number generator to produce a sequence of random values from the hardware-generated random seed [11]. Such a method of combining different forms of random number generators attempts to circumvent the potential security risk for some applications that a deterministic method of generating random values could create. The deterministic nature of pseudorandom number generators allows for results to be repeatable, i.e. given the same initial value, the PRNG will produce the same series of pseudorandom numbers [4]. The algorithms behind pseudorandom number generation often vary quite a bit in how they are implemented and how effectively 'random' their results are. One

measure of effectiveness for pseudorandom number generation algorithm is the period, which is the number of states the PRNG goes through before cycling back to the initial seed value and therefore repeating the sequence of values indefinitely. A ‘bad’ or inefficient PRNG algorithm will have a low period, resulting in less of an illusion of randomness as the deterministic sequence of unique values is too small to create the illusion of randomness before repeating. An efficient Pseudorandom number generation algorithm will produce an even distribution of values with an extremely long, or ideally infinite, period of states in the sequence before the sequence repeats itself. In this project however, we are not concerned with producing the most efficient or evenly distributed algorithms, we simply want to use heatmap visualizations to explore different methods of producing pseudorandom numbers.

Heatmaps are a common method of visualizing multiple dimensions of data, with the color of data points being drawn to correspond with a mapping from the data value to some color of the color map [5]. The two most common types of heatmaps are spatial and grid heatmaps, with spatial heatmaps having data points that correspond to location, such as a map of where users most often click on a webpage [12], and grid heatmaps which represent a matrix of values, where each value in the matrix is represented by a cell in the grid that is mapped to a color based on the color map being used. Often, color maps will consist of a gradient from dark to light or warm to cold, or any other array of colors to represent the range of values in the dataset. This way, data can be interpreted in a more visual and intuitive sense. Data visualization is becoming more important as tasks become more complicated, and more data is involved. Being able to devise methods of interpreting datasets, no matter their complexity, is crucial to almost any field in the computing space. The beauty of data visualizations come with how they can create an intuitive understanding of a dataset from an image that can be quickly interpreted.

Statement of Objective

The purpose of this project is to produce a command-line based application for a client that can be used to produce multiple forms of heatmap art using multiple different implementations of pseudorandom number generators. The nature of the project is mainly exploratory. The goal is not to implement the most efficient or cryptographically secure cutting-edge PRNG algorithms, but to implement a variety of mostly not-so-efficient algorithms to be able to visualize their inefficiencies to produce interesting results using heatmaps as an artistic medium.

Stipulations for the project, as requested by the client, include that the pseudorandom number generation portion should consist of object-oriented C++ code with some class hierarchy that explores a variety of PRNG algorithm implementations. Also, the heatmap visualization portion should support a variety of methods of heat-mapping, with no limitations on the languages or libraries used to support visualization generation. As we are not seeking to come up with any novel methods of generating pseudorandom values, the most pressing problem for this project is to create unique data visualizations that give the user some intuitive understanding of different pseudorandom number generation algorithms and their potentially quirky and inefficient behavior.

The project was completed and delivered to the client as a part of the capstone course within the Computer Science department of UMass Boston. The team for the project consisted of myself and seven other students. The project is publicly available on GitHub at the following address: https://github.com/UMB-Heatmap/heatmap, with detailed use instructions available in the readMe.txt file for any readers who wish to explore further. All figures included in this report were produced by the project and list the exact parameters used in their description in case the

reader wishes to reproduce any of the heatmaps used in this report. The following will be a discussion of the methodology and results of only the portion of the project that I personally contributed. The other algorithm implementations and visualizations included in the delivered project that I did not contribute to the development of will not be discussed in detail.

Methods

Project Structure

The structure of the project consists of an object-oriented C++ backend that produces a text file containing a sequence of N random scalar values in the range [0, 1) as output and a python-based front end which generates the heatmap visualizations by calling the external C++ to generate all necessary random values. The C++ backend that houses the PRNG implementations is interacted with via command-line switches that are used to specify which algorithm and seed value to use, as well as the length of the pseudorandom sequence desired. There are additional switches for debugging, specifying a custom output file name, and supplying additional arguments to certain algorithms that require them. The main driver for the project is a simple python script which checks that the C++ backend is properly compiled and built for the machine running it and handles the command-line arguments to resolve the algorithm, visualization, and seed specified by the user. If the main driver is not able to access a pre-existing version of the compiled C++, or if there is no compiled object available, it will call the Makefile to clear the current version and rebuild the C++ backend. Once the command-line arguments have been validated, the main driver calls the python script corresponding to the visualization specified by the user, passing through the algorithm and seed information to be used by the visualization script. Any additional parameters that are visualization-specific are

acquired by the visualization script and all visualizations take advantage of the matplotlib library functionality for heatmap generation.

The delivered project includes nine total options for pseudorandom number generation algorithms and eight total options for heatmap visualizations, of which I implemented the algorithms XORSHIFT and SPLITMIX as well as the following visualizations: 2D Heatmap, Distribution Heatmap, Seed Evaluation Heatmap, 3D Scatterplot Heatmap, and 3D Walk Heatmap.

XORSHIFT

The xorshift pseudo-random number generator algorithm is a deterministic algorithm that generates a sequence of numbers by repeatedly performing an xor-assign operation on the current state with a bit-shifted version of the current state, hence the name ‘xorshift’ [6]. There are slight variations of the xorshift algorithm depending on the number of bits used to represent the state of the deterministic machine, including 32-bit, 64-bit, and 128-bit. The version implemented for the purpose of this project is the 64-bit xorshift. Xorshift-based random number generators have the property of being exceptionally fast and requiring very little code to implement, as shown in Figure 1 below. The major issue pertaining to the xorshift PRNG algorithm is that it fails to escape the case where the state consists of all zeros. Also, when xorshift is used in practice, it is often initialized using a different algorithm such as splitmix [6], in order to help minimize the likelihood of xorshift producing erroneous results.

```
class XorShift64 : public Algorithm {
public:
    XorShift64(uint64_t seed) : Algorithm(seed) {};
    uint64_t peekNext() {
        uint64_t x = this->state;
        x ^= x << 13;
        x ^= x >> 7;
        x ^= x << 17;
        return x;
    };
};
```

Figure 1: 64-bit XORSHIFT algorithm implementation

SPLITMIX

Like xorshift, splitmix is a pseudorandom number generation algorithm that may not be cryptographically secure, but is computationally inexpensive and requires only a few lines of code to implement, as shown in Figure 2 below. The splitmix algorithm is best known for being the default algorithm used by the Java language [7] and works by incrementing the current state by a constant value, then taking advantage of a combination of xor-assign operations with bit shifted versions of the current state (similar to xorshift) with additional multiplication of the current working state by a constant for each step of xor-assign and bit shift operations.

```
class Splitmix : public Algorithm {
public:
    Splitmix(uint64_t seed) : Algorithm(seed) {};
    uint64_t peekNext() {
        uint64_t x = this->state;
        uint64_t result = (x += 0x9E3779B97f4A7C15);
        result = (result ^ (result >> 30)) * 0xBF58476D1CE4E5B9;
        result = (result ^ (result >> 27)) * 0x94D049BB133111EB;
        return result ^ (result >> 31);
    };
};
```

Figure 2: 64-bit SPLITMIX algorithm implementation

Visualization Utilities Library

As discussed previously in the project structure section, the visualizations are each contained in their own python script which is called by the main driver script that the user calls from the command-line. With so much overlap in the implementation of each visualization, it made the most sense to centralize common functionalities that would be needed by multiple visualizations into a single python library. The main purpose of the visualization utilities library is to abstract common functionalities to make visualization-specific code more readable and easier to expand upon. The library includes functionality for the following: getting and validating additional visualization parameters from the user via standard input (including variations for getting an Integer, Boolean, Float, and a valid Color Map), calling the C++ backend with the appropriate switches for each algorithm and processing the resulting output file into a list containing the sequence of PRNG values that can be returned to each visualization, checking for a valid C++ object file and rebuilding if necessary, handling command-line input for the main python driver, and opening completed visualizations with the correct command depending on the

user's operating system. The library also houses the master list of algorithms and visuals, with functions available to validate user input for both the algorithm and visualization options before passing the data back to the main driver to call the specific visualization script. All user input and error-handling is managed within the centralized visualization utilities library for the sake of consistency and code reusability.

2D Heatmap Visualization

The 2D Heatmap Visualization is the most basic implementation of generating heatmap art from pseudorandom number generators. This is the first, and most obvious approach to visualizing PRNG's with heatmaps. The visualization generates an *N* by *M* heatmap of random numbers, where each coordinate cell of the heatmap corresponds to a random number generated by the algorithm specified by the user when calling the main python driver. In theory, the more efficient or 'random' a PRNG algorithm is, the more the 2D Heatmap will appear as noise. Any clustering or consistency between rows or columns in the 2D Heatmap would indicate a degree of predictability in the PRNG algorithm, often not desired for producing the illusion of randomness. As the most basic implementation of a heatmap visualization, this was also used as a template to create other 2-dimensional visualizations and can be broken down into five steps that will be common to most visualization. First, acquire and validate visualization-specific inputs. In this case, 2D Heatmap requires three additional inputs: number of rows *N*, number of columns *M*, and the color mapping option. Color map options are native to matplotlib and the user is given a choice between twenty-one available mappings. Second, generate data for the visualization. This is done through the function nRandomScalars(algo, seed, num_values, algo_args) which is defined in the previously mentioned visualization utilities library and

abstracts the calling of the backend C++ with the correct switches for the chosen algorithm and seed. Third, generate the visualization using matplotlib's imshow function to produce a heatmap with the data generated from the chosen PRNG algorithm. Fourth, save the visualization into the heatmaps output folder with the correct format, which in this case is an SVG. Scalable Vector Graphic files were chosen as they allow for the best scalability without causing the images to become distorted. Fifth, and finally, the saved visualization SVG file is opened using the visualization utilities library function openVisual(path) which checks the host machine's operating system and runs the corresponding operating system-specific command to open the visualization file with the system's default application (typically the browser). An example of the 2D Heatmap Visualization can be seen in Figure 3 below.

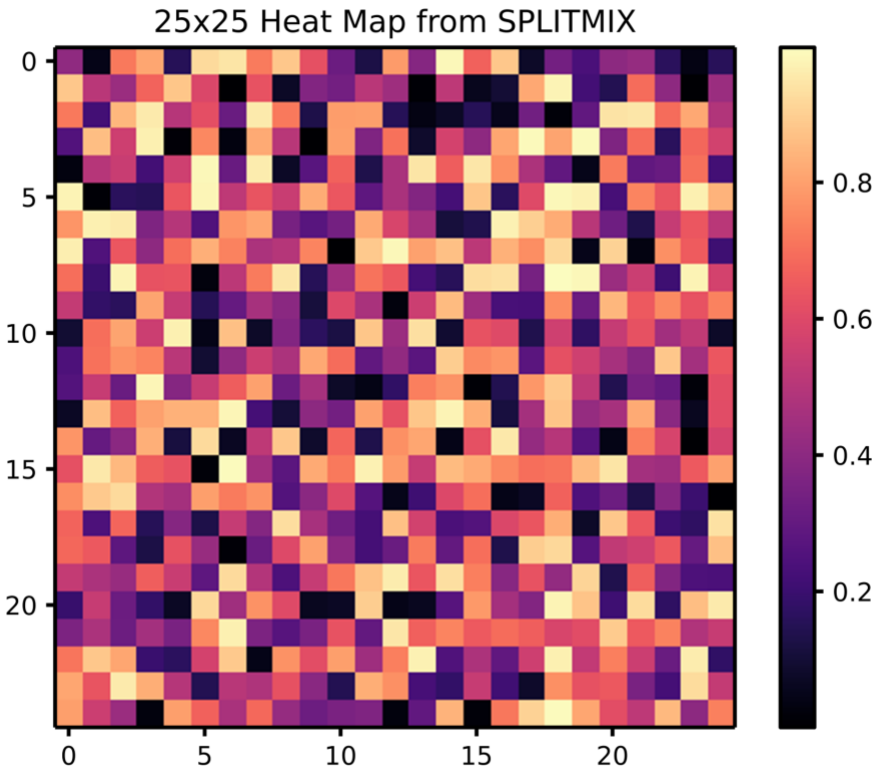

Figure 3: 2D Heatmap Visualization example using command-line-interface arguments [algorithm = splitmix, visual = 2d, seed = 999] and visualization-specific arguments [rows = 25, columns = 25, colormap = magma]

Distribution Heatmap Visualization

The Distribution Heatmap Visualization is a variation of the 2D Heatmap with the intent of generating an animated GIF file that shows the distribution of random values over multiple iterations of selecting a user-specified number of values and incrementing heatmap cell values according to the random values chosen. The visualization works by having N possible candidates, which are represented by the cells of each row or the number of columns, and X iterations of Z random values in the range $[0, N)$, which are represented by the X total rows where each subsequent row from the top to bottom of the heatmap choses Z random values from $[0, N)$ and increments the corresponding cell in the row each time its position is randomly chosen. The cell values in the row are then divided by the total number of random values chosen so far to give the distribution of random values in the range $[0, N)$ for an increasingly large sample size with each row. These distributions are then mapped to the color pallet of the user's choice and displayed as an animated GIF file. The purpose of the visualization is to create an intuitive way to show the inefficiencies of some pseudorandom number generation algorithms in real time as they generate values and therefore populate the rows of the distribution heatmap animation with each iteration of generating values and updating the distribution of pseudorandom values in the range of the size of each row. In this case, as the heatmap is populated with values, the rows should blend towards a consistent color across each row, provided that a sufficient sample of random values is being tested with. Consistent colored rows in the distribution heatmap indicate a uniform distribution of selected random values, which is an indicator of an efficiently 'random' PRNG algorithm. As can be seen in Figure 4 below, the sample size of randomly selected values in each row increases with each subsequent row, starting from the top row of the heatmap, causing the first set of rows to have more clearly

defined differences in the distributions of each cell, which are smoothed out towards the bottom of the heatmap as the PRNG's distribution of selected values becomes more uniform with a larger sample size.

In order to generate the animated GIF file of the heatmap being populated row-by-row as each iteration selects more random numbers and updates the distribution across each row, a PNG file is generated for each row. The GIF file is then saved using the Image module of the python's Pillow library with each row's PNG file as a single frame of the final animation. Finally, the visualization script uses the abstracted function from the visualization utilities library to open the GIF file with the default application on the user's machine. An example can be seen in Figure 4 below.

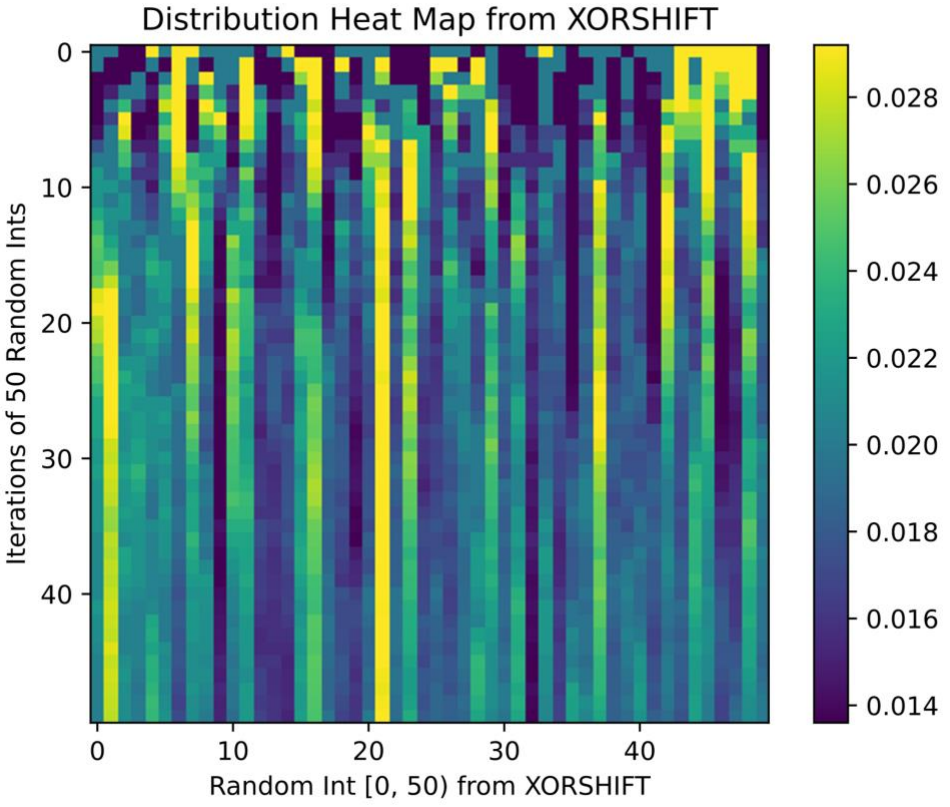

Figure 4: Distribution Heatmap Visualization example (final frame of animation) using command-line-interface arguments [algorithm = xorshift, visual = distribution, seed = 9876] and visualization-specific arguments [N = 50, X = 50, Z = 50, colormap = viridis]

Seed Evaluation Heatmap Visualization

The Seed Evaluation Heatmap Visualization is another variation of the 2D Heatmap with the intent of allowing the user to visualize the period and cycles of different seed values of a specified algorithm. In this visualization, each row corresponds to an initial seed given to the chosen pseudorandom number generator. The user is prompted to input the desired number of seeds to view, which is also the number of rows, the number of values to show from each seed, which is also the number of columns, the color map, and the minimum and maximum seed values to show. The script then generates the data by calling the C++ backend via the abstracted function defined in the visualization utilities library to get the values for each seed. If the range of seeds exceeds the number of rows chosen, the program will use an evenly spread-out range of seeds between the minimum and maximum inputted by the user. The heatmap is then generated, mapping the values of each cell to the user-specified color map, and saving the finished product as an SVG file. Finally, the visualization script opens the SVG file with the default application on the user's machine. An example can be seen in Figure 5.

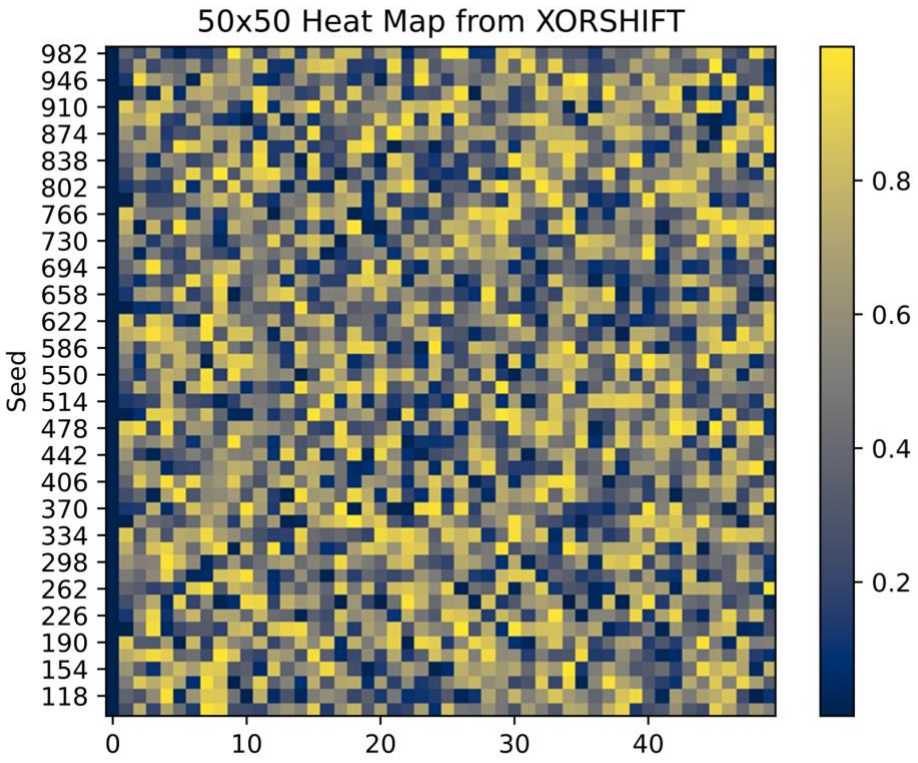

Figure 5: Seed Evaluation Heatmap Visualization example using command-line-interface arguments [algorithm = xorshift, visual = seed_eval, seed = 123] and visualization-specific arguments [rows = 50, columns = 50, min_seed = 100, max_seed = 1000, colormap = cividis]

3D Scatterplot Heatmap Visualization

The 3D Scatterplot Heatmap Visualization takes advantage of matplotlib's 3-dimensional scatterplot functionality to generate an interactive 3D scatterplot of randomly generated points. The visualization opens matplotlib's native plot viewer to allow the user to zoom, move, and explore the random points in 3-dimensional space. Using matplotlib's slider functionality to add more user control to the interactive plot, the visualization script includes sliders so that the user can control both the number of points being displayed as well as the size of each point while interactively viewing the scatterplot through matplotlib. There is also an included option to

generate a GIF animation of the random points being generated, where each frame of the animations adds more random points until reaching the user-specified maximum. The 3D scatterplot visualization also offers five options for how to map the colors from the color map onto the 3D points in the scatterplot. The color mode options include random, diagonal gradient, x gradient, y gradient, and z gradient, with each specifying which value, or combination of values, to use for the color mapping of each random point in the plot. An image of an example 3D Scatterplot Heatmap Visualization can be seen in Figure 6.

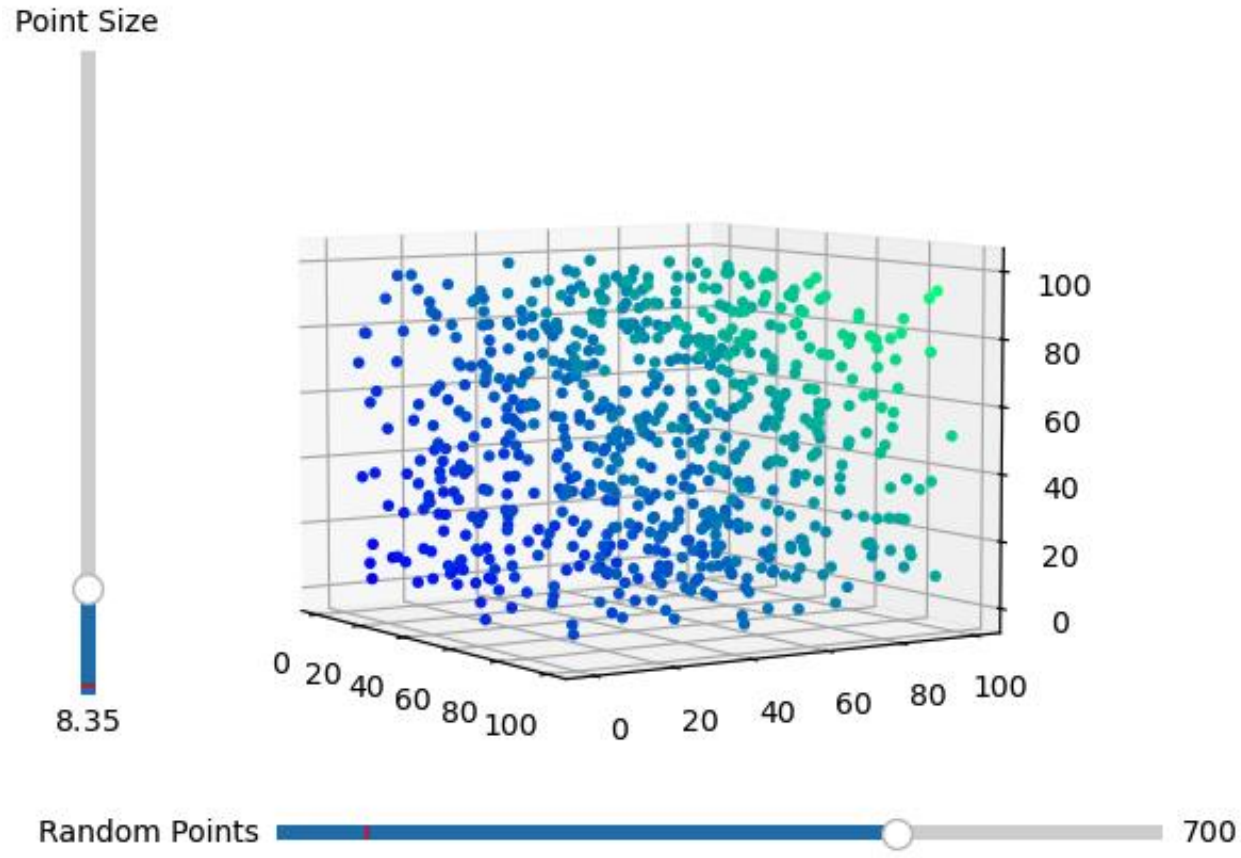

Figure 6: 3D Scatterplot Heatmap Visualization example using command-line-interface arguments [algorithm = xorshift, visual = 3d_scatter, seed = 91239] and visualization-specific arguments [num_points = 1000, colormap = winter, colorMode = diagonal gradient]

3D Walk Heatmap Visualization

The 3D Walk Heatmap Visualization also takes advantage of matplotlib's 3-dimensional scatterplot functionality to generate an interactive 3D scatterplot. In the case of this visualization, it prompts the user to specify the total number of random steps, the maximum size of each step, and the color map. The visualization begins by choosing a point in 3D space to be the starting point, and for each step, increment the x, y, and z components of the current point each by a randomly generated value between zero and the user-specified maximum step size. A line is then drawn between the current point and the next point, and the loop continues until all points have been generated. The color of the point is then mapped to the chosen color map according to the progress of the random walk, being the current step number divided by the total number of steps. This makes the random walk much more visually intuitive as the eye can follow the path of points both by the line connecting them and by the gradient of color from the starting point to the final point in the random walk path. Like the 3D Scatterplot Visualization, 3D Walk uses matplotlib's built-in interactive plot viewer to display the results to the user, with additional sliders that allow the user to control in real-time how many points of the random walk should be displayed and how large each of the points should be. An image of an example 3D Walk Heatmap Visualization can be seen in Figure 7.

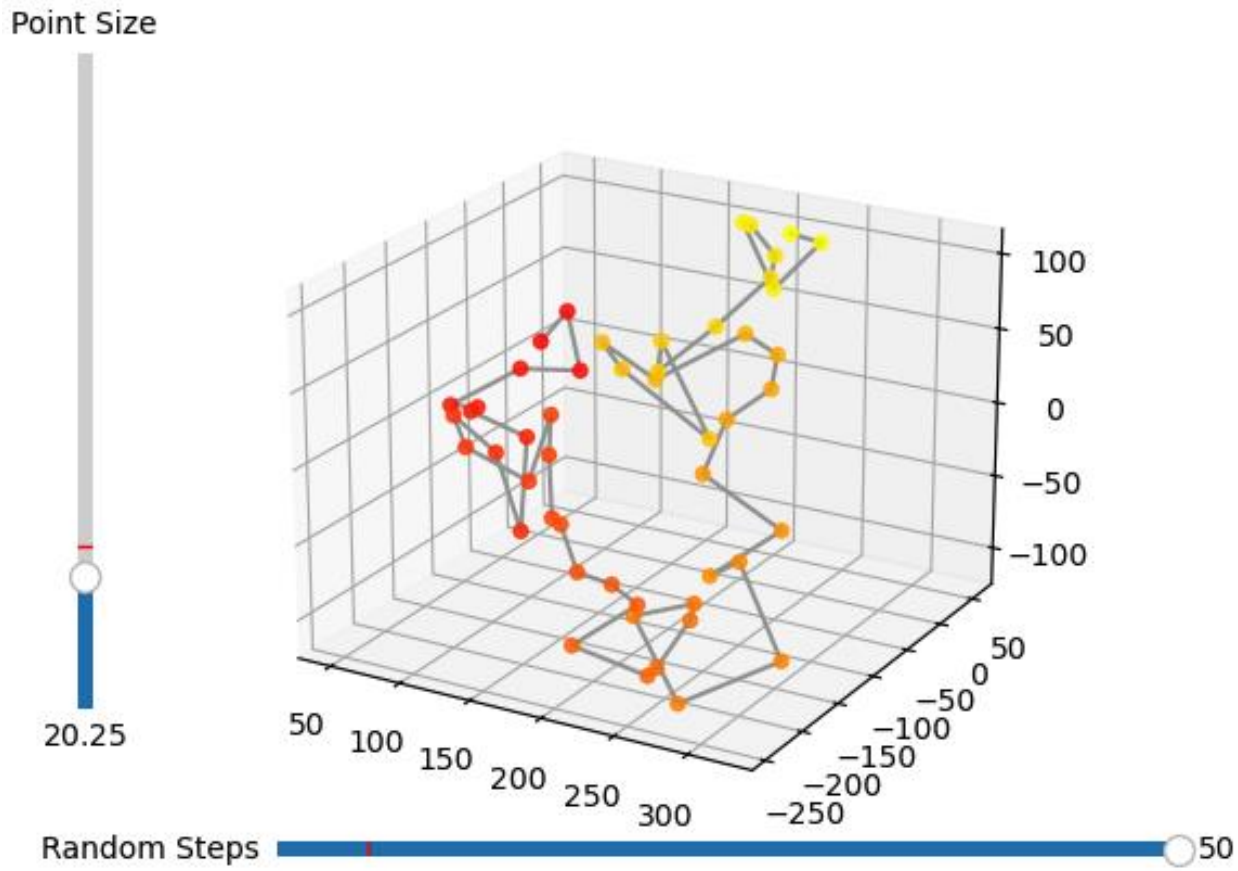

Figure 7: 3D Walk Heatmap Visualization example using command-line-interface arguments [algorithm = splitmix, visual = 3d_walk, seed = 17777] and visualization-specific arguments [num_steps = 50, step_size = 50, colormap = autumn]

Results

The results of the project are a completed command-line based application for generating heatmap art from different pseudorandom number generation algorithms. The final product, as delivered to the client, features nine PRNG algorithm options and eight visualization options, which all work interchangeably, with additional options and parameters for each visualization to allow the user as much customization as possible for generating art using pseudorandom numbers and heatmaps.

In terms of solving the problem of how to visualize pseudorandom data in a way that gives the user some easy-to-understand or intuitive sense of the shortcomings of different PRNG algorithms, the Distribution and Seed Evaluation Heatmap Visualizations serve this role for the purposes of this project. Additional visualizations such as 2D, 3D Scatterplot, and 3D Walk Heatmaps are also available for exploration of PRNG algorithms and producing interesting visual pieces.

With the distribution heatmap's resulting GIF animation, the user can watch the PRNG select random values over multiple iterations as it populates the grid heatmap. With the waterfall-esc visual effect, an efficient PRNG algorithm with sufficient iterations of selecting random values should blend into rows of smooth color mapping, indicating that the distribution of random values generated by the PRNG algorithm are mostly uniform. Vertical streaks of non-uniform color on a distribution heatmap indicate areas where the PRNG is either favoring or failing to reach, as can be seen in the example in Figure 4, where xorshift does begin to blend to a more uniform color across the row towards the bottom of the heatmap, but still has prevalent streaks of non-uniform color. This would indicate that, at least with the seed and number of values generated, the algorithm is not producing an even distribution of random values, suggesting a potential fault in its degree of 'randomness'.

Considering how many PRNG algorithms, especially the less efficient ones focused on in this project, vary heavily in their degree of 'randomness' depending on their seed value, the seed evaluation heatmap visualization allows the users to test a batch of seeds across a range of values. With each row of the grid heatmap produced corresponding to a unique seed and the cells of that row being mapped from the sequence of random values generated by the chosen algorithm, any sequence of repeating cells or sequence of same-colored cells indicates a seed that

produces a cycle or potentially gets stuck cycling on a single value indefinitely. An obvious example of this can be seen when the seed evaluation heatmap is generated using the middle square algorithm, which is a very poor PRNG that quickly produces cycles and gets stuck on all-zero states with many seed values.

Discussion

As previously stated, the aim of the project was not to implement the most efficient algorithms, but to explore a variety of PRNG algorithms and how they can be visualized to produce interesting pieces through heatmaps. If we expand the analysis to the additional algorithms included in the final project, that were implemented by my fellow group members, and compare some visualizations of different algorithms, it will become clearer just how much variation there is in the PRNG space. For example, the by far least efficient and evenly distributed PRNG algorithm included in the delivered project was called ‘four’. The only rule for the ’four’ implementation is that the algorithm will only return the number 4 [10]. This is a perfect example of a horrible algorithm for producing anything actually ‘random’, but it works well for the sake of comparison to other more legitimate PRNG’s as ‘four’ serves as a baseline for inefficiency. A comparison of the distributions of the ‘xorshift’ and ‘four’ algorithms can be seen in Figure 8. When viewing a distribution visualization, vertical streaks of abnormally light or dark colors indicate values where the algorithm is either cycling on repeatedly (lighter color vertical streaks) or cycling past (darker color vertical streaks), visually demonstrating the lack of uniformity in the distribution of the chosen PRNG algorithm with the seed specified.

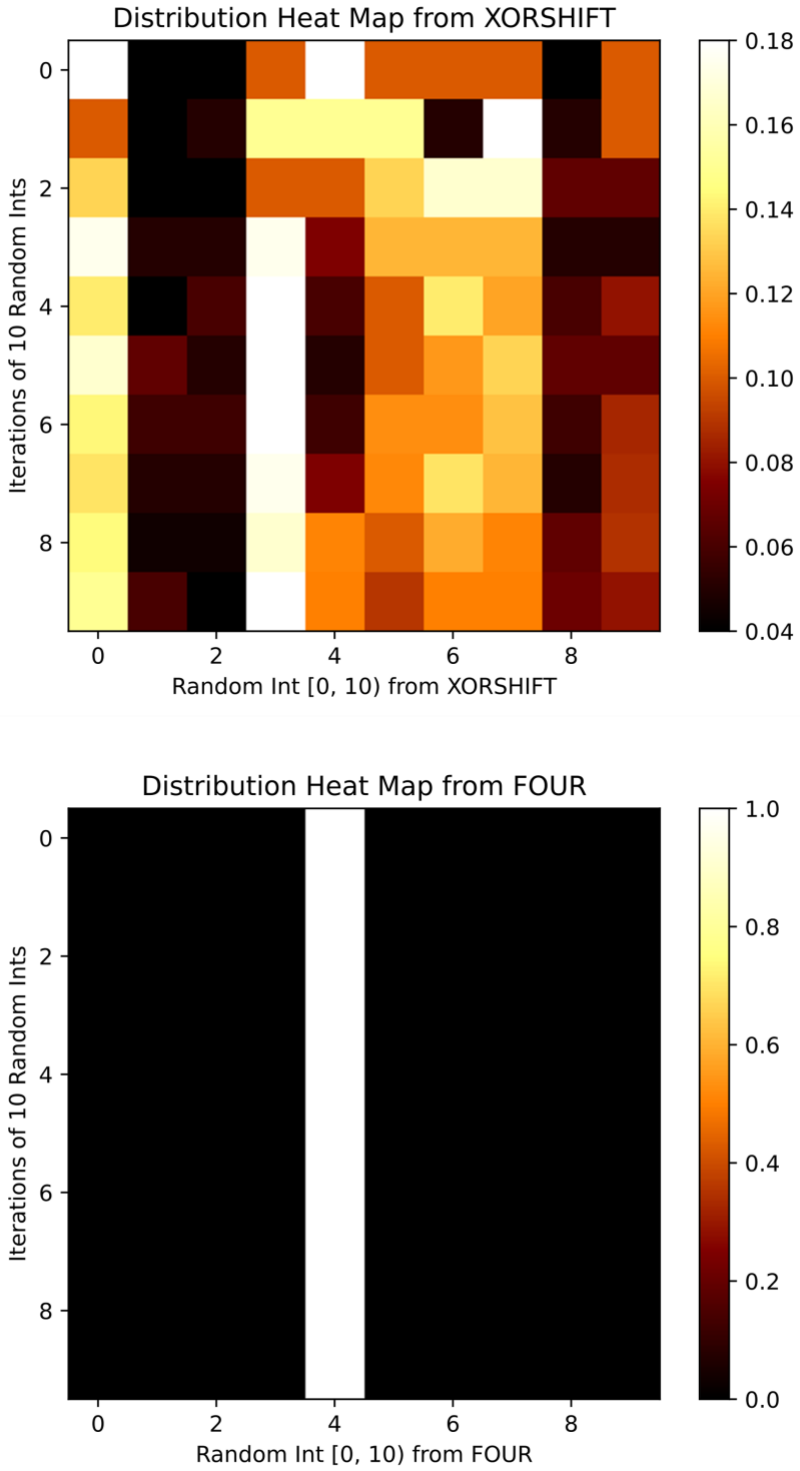

Figure 8: A comparison of 'xorshift' and 'four' PRNG algorithms using the Distribution Heatmap Visualization with command-line-interface arguments [algorithm = xorshift/four, visual = distribution, seed = 123] and visualization-specific arguments [N = 10, X= 10, Z = 10, colormap = afmhot]

Another example of a not-so-efficient algorithm that can be explored visually and is included in the final project is the middle-square method, which produces an n-digit random value by squaring the current state then taking only the middle n-digits from the result to be the next state [8]. The results of using middle-square are heavily dependent on the initial seed, as it will inevitably reach a point where it cycles either to a state that is all zeros, in which it will every subsequent state will be all zeros, or to a state that is equivalent to the initial seed, in which it will repeat the same sequence of values indefinitely. This can be demonstrated visually using the Seed Evaluation Heatmap Visualization that was previously mentioned and can be seen in Figure 9 below. With each row of the seed evaluation visual corresponding to a unique seed in the range specified by the user, it can be seen how the middle square algorithm's performance varies severely depending on the seed. Rows that appear as noise, without repeating color patterns or solid color representing all-zeros, represent seed values for which the middle square algorithm performs effectively as a source of algorithmic entropy. Specifically, it can be seen that the row corresponding to the seed 2998 gets stuck in a cycle of all zero values after producing a sequence of about thirty values, indicated by the column position where the row begins a consistent light teal color, corresponding to zero on the color map. The row just below seed 2998, which corresponds to seed 1999 (not labeled), also contains uniquely inefficient behavior, which can be identified by the repeated pattern of colors across the row, also starting about thirty values into the sequence. The repeated pattern indicates that the middle square algorithm is stuck in a cycle of values, which is highly undesirable behavior for any application that demands unpredictability.

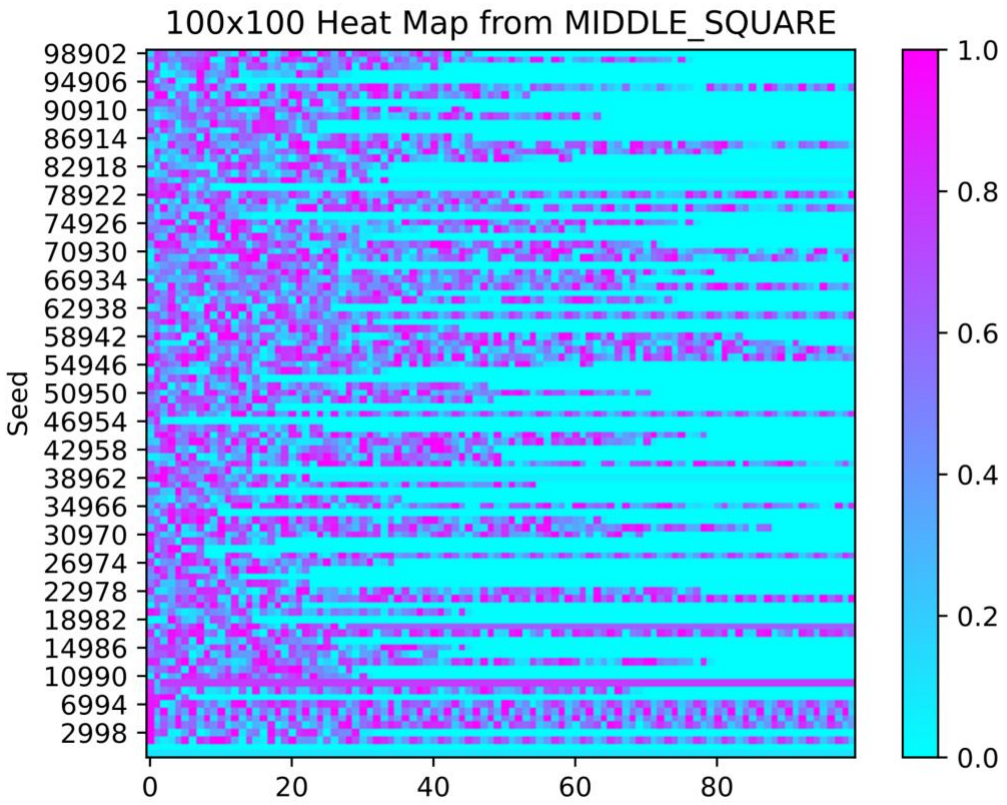

Figure 9: Seed Evaluation Heatmap Visualization of Middle Square PRNG algorithm using visualization-specific arguments [rows = 100, columns = 100, min_seed = 1, max_seed = 100000, colormap = cool]. Rows are labeled with the seed they correspond to.

Even when we look at an algorithm far more 'random' than the four or middle square algorithms, such as xorshift, we can use the implemented heatmap visuals to see its shortcomings. As previously mentioned, when used in practical or more sophisticated applications, xorshift is often initialized with another differing PRNG algorithm such as splitmix in order to produce more evenly random values without being as dependent on the starting seed value. The need for this first-state initialization with another algorithm can be seen with another use of the Seed Evaluation Heatmap Visualization seen in Figure 10. The first value produced by each seed is effectively zero, which is why xorshift when used in practice is often initialized

using another PRNG algorithm to avoid this quirk. However, for the sake of this project, this quirk is exactly what we were looking to explore.

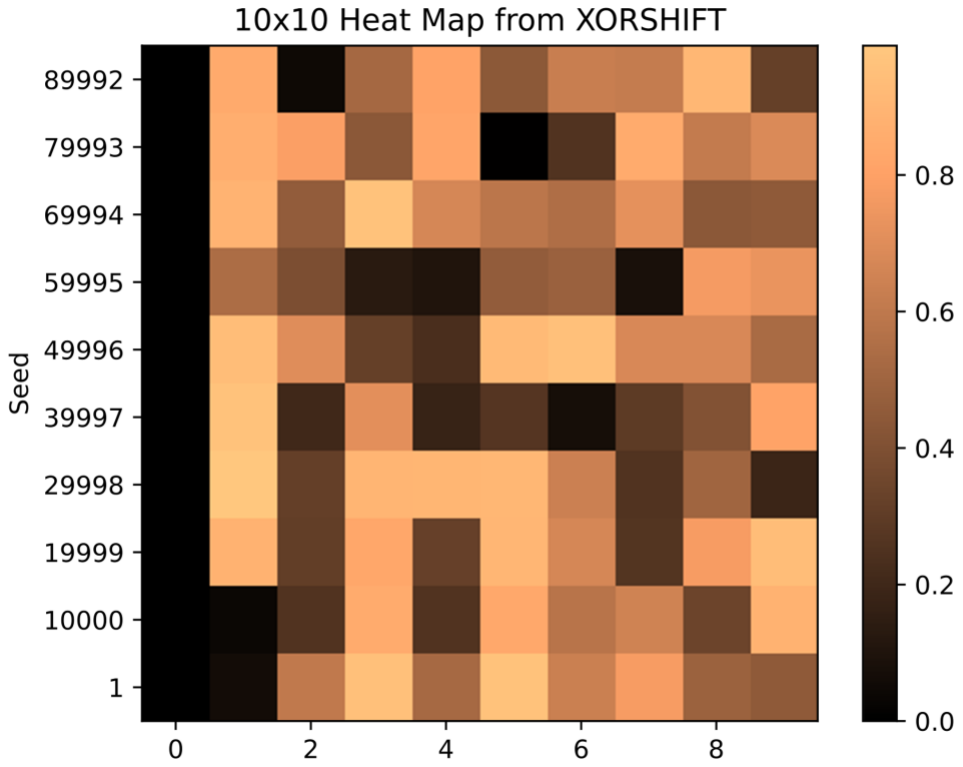

Figure 10: Seed Evaluation Heatmap Visualization of Xorshift PRNG algorithm using visualization-specific arguments [rows = 10, columns = 10, min_seed = 1, max_seed = 99999, colormap = copper].

Conclusion

The completed PRNG Heatmap Visualization project effectively fulfills the client's wishes of a command-line based application for generating a variety of heatmap art from a variety of pseudorandom number generation algorithms. The delivered project allows the user to explore nine different pseudorandom number generation algorithms including: xorshift, splitmix, linear congruential generator, lagged fibonnaci, lehmer, middle square, blum blum shub, rule30, and four. The user can select a PRNG to be visualized using eight different methods including: 2d, distribution, frequency, 3d scatterplot, 3d walk, 3d, shaded relief, and seed evaluation. The implemented visualizations provide a variety of ways to explore the quirks of different ways of deterministically generating pseudorandom values through heatmap art.

The problem of how to visualize the shortcomings and quirks of different PRNG algorithms was solved primary through the implementation of both the Distribution and Seed Evaluation Heatmap Visualizations, which give the user different methods of assessing the efficiency of the included algorithms in a visually intuitive manner. It was demonstrated how both the distribution and seed evaluation heatmaps could be used to get a visually intuitive representation of the shortcomings of different pseudorandom number generation algorithms, such as middle square's seed dependency issue discussed with Figure 9 or xorshift's first-value issue discussed with Figure 10. Other visualizations included in the project also give the user the freedom to explore a variety of algorithms in unique ways to produce interesting pieces of heatmap art, as requested by the client in the project description.

References

[1] “Hardware Random Number Generator.” *Wikipedia*, Wikimedia Foundation, 10 May 2024, en.wikipedia.org/wiki/Hardware_random_number_generator.

[2] “Entropy (Computing).” *Wikipedia*, Wikimedia Foundation, 12 Mar. 2024, en.wikipedia.org/wiki/Entropy_(computing).

[3] Tashian, Carl. “A Brief History of Random Numbers.” *A Brief History of Random Numbers - *, 25 Nov. 2023, tashian.com/articles/a-brief-history-of-random-numbers/.

[4] “Pseudorandom Number Generator.” *Wikipedia*, Wikimedia Foundation, 25 Apr. 2024, en.wikipedia.org/wiki/Pseudorandom_number_generator.

[5] “Heat Map.” *Wikipedia*, Wikimedia Foundation, 15 Mar. 2024, en.wikipedia.org/wiki/Heat_map.

[6] “Xorshift.” *Wikipedia*, Wikimedia Foundation, 21 Apr. 2024, en.wikipedia.org/wiki/Xorshift.

[7] “Pseudo-Random Numbers/Splitmix64.” *Rosetta Code*, Rosetta Code, 9 May 2024, rosettacode.org/wiki/Pseudo-random_numbers/Splitmix64.

[8] “Middle-Square Method.” *Wikipedia*, Wikimedia Foundation, 24 Apr. 2024, en.wikipedia.org/wiki/Middle-square_method.

[9] Thomas, et al. *Heatmap* (2024), GitHub repository, https://github.com/UMB-Heatmap/heatmap

[10] Lee, et al. *Heatmap* (2024), GitHub repository, https://github.com/UMB-Heatmap/heatmap

[11] Rukhin, Andrew, et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, 15 May 2001, pp. 1–3, https://doi.org/10.6028/nist.sp.800-22.

[12] Gu, Zuguang. “Complex heatmap visualization.” *iMeta*, vol. 1, no. 3, Aug. 2022, https://doi.org/10.1002/imt2.43.

[13] “Random Numbers.” *Simul8*, Simul8, www.simul8.com/support/help/doku.php?id=features%3Arandom_numbers#:~:text=Random%20numbers%20enable%20a%20simulation,separate%20stream%20of%20random%20numbers. Accessed 16 May 2024.