**Methodology**

**3.1 Sample Selection**

Before testing could begin, suitable true random and pseudorandom generators had to be chosen. As outlined previously (**See Section 1: Introduction**) all algorithms tested represented functions from in-use languages and sites as well as industry standard packages for said languages. An appropriate digital generator needed to:

* Be in active use in either a commercial or scientific environment.
* Be able to produce at least 500 values in a single generation.
* Be able to produce a double or integer output or a non-numeric output within a chosen format (e.g., Coin Faces or Cards from a Deck).

These requirements ensured that any generators chosen for testing would represent tools currently available to both professionals in industry and the public. This also ensured that the generators in question would provide outputs suitable for individual and grouped analysis. In the event where an output would differ from the conventional integer or double values, primarily when producing a ‘random’ combination of cards from a deck, suitable generators would be required to be able to have their outputs adjusted accordingly.

In total, six pseudorandom sources matching the criteria listed above were chosen for analysis. Three industry standard languages were chosen, these being C#, Python and JavaScript, as well as reproduced algorithms of the Lehmer Generator and Von Neumann’s Middle Square Method (both written in C#). Finally, data was taken from Random.Org, a site built in 1998 by the School of Computer Science and Statistics at Trinity College in Dublin (**Random.Org, 2023**) which serves as one of the most well-known sites for random number generation on the internet. Four physical true random sources were also chosen. Dice, coins and playing cards, three sources of random chance that have been used for thousands of years were sampled for comparison to their pseudorandom counterparts. White-noise data from three different locations across Plymouth was also recorded in hopes of providing true random sequences to compare against the integer and double sequences given by the pseudorandom generators.

C# is an object-oriented, component-oriented programming language (**Microsoft, 2023**) that is primarily used for the creation of applications using the .NET framework. It is used by a variety of companies including Microsoft, Stack Overflow and Trustpilot and can be utilised to create both web and desktop applications. For this reason, it was chosen as one of the pseudorandom data sources to be tested. Three different integer sequence samples were taken from C#, two from the *rand.Next()* function and the third from the *RandomNumberGenerator.GetBytes()* function provided to C# by the Cryptography library. When using the *rand.Next()* function, the seed given to the algorithm can be chosen either by the system itself using the PC clock or by being input by the user. The *rand.Next()* function also produced two outputs for dice roll, coin flip and card shuffle simulations.

Similar to C#, Python is a high-level, general purpose programming language. It supports multiple programming paradigms, including structured, object oriented and functional programming (**Wikipedia (X), 2023**) and is used by companies such as Google, Dropbox, and Netflix as well as by data scientists alongside R for data analysis. Five sequence samples were taken from Python, three sets of integer values and two sets of double values. This is because three different functions were sampled from Python: *Randint()*, *Random()* and from the NumPy library *Numpy.Random()*. As with the *rand.Next()* function in C#, both *Random()* and *Numpy.Random()* can have their seeds selected either by the system clock or manually by the user. In addition to the integer and double sequences the Python functions were also used to produce outputs for dice, coin, and card simulations, with these simulations also being adjusted to use either the system or user input seed.

JavaScript is a programming language that is one of the core technologies of the World Wide Web, alongside HTML and CSS. As of 2023, 98.7% of websites use JavaScript (**Wikipedia (X), 2023**) so including it as a data source for this investigation was deemed essential. In addition, this would allow for comparisons to be made regarding the quality of Google’s pseudorandom generator, which also uses the *Math.Random()* function provided by JavaScript but isn’t capable of generating large batches of values at once. Only one integer sequence value was taken from JavaScript as the *Math.Random()* function provided doesn’t allow the user to manually input a seed for generation and instead only uses the PC clock to determine seed. Similarly, only one set of outputs were given for the coin and dice simulations and due to the limitations of *Math.Random()* only being able to produce integer sequences, attempts to produce a suitable card simulation were not included.

As outlined in Section 2, the Lehmer Generator was first coined by Park and Miller in 1988 as a ‘minimum standard’ for a reliable pseudorandom generator. As its variants still see use in functions today like *minstd\_rand* for C++, including the generator in this investigation was an obvious choice. The double sequence outputs given for this data source are split into four: two using a ‘version 1’ integer based and real based algorithm and two using a ‘version 2’ integer based and real based algorithm. The use of either integers or real values in these algorithms refers to the variable types used to store the seed, seed multiplier and modifier values used during calculation. As with the JavaScript functions, due to the specific type of outputs given by the Lehmer Generator it was unsuitable to be used for coin, dice, or card shuffle simulations.

In contrast to the Lehmer Generator, Von Neumann’s 1949 Middle Square Method is considered to be a highly flawed method for many practical purposes (**Wikipedia (X), 2023**). While other pseudorandom algorithms were included in this investigation due to their continued use in commercial or scientific fields, the Middle Square Method was included due to the ease in recreating it and as an example of a generator proven to be flawed in producing random numbers. One integer sequence was outputted from the Middle Square algorithm used and much like the Lehmer Generator, due to the specific output of the algorithm, which also doubles as the seed for its next iteration, dice, coin, and card shuffle simulations could not be performed.

Random.org, created by Dr Mads Haahr in 1998, aims to offer true random numbers to anyone on the Internet (**Random.org, 2023**). This is done using an algorithm based on atmospheric noise, a principle that was used as the inspiration for the white noise data collected for this investigation. The claim, that through the use of atmospheric noise this digital generator was capable of producing true random instead of pseudorandom outputs, made the inclusion of Random.org data invaluable as not only did the generator meet the criteria specified above but if it could be proved through empirical testing that this generator is capable of producing true random outputs then it would serve as a benchmark when comparing the other pseudorandom generators featured. Three outputs were taken from Random.org: an integer sequence, a dice roll simulation, and a coin flip simulation. Due to the limitations in the format of possible outputs, a card shuffle simulation wasn’t considered to be viably possible with this generator.

While the data provided by Random.org could be used for comparison between true random and pseudorandom numeric sequences, the best tools for aiding in the evaluation of the ability of pseudorandom generators to replicate true random sequences were physical generators. Much like many of the pseudorandom generators used in this investigation, physical generators have been used in both commercial and scientific environments and are able to produce outputs in a variety of formats. Dice, coins and playing cards, although more commonly seen as components in games rather than number generators for study were chosen for this investigation due to the large number of commercial tools available that are designed to replicate them in a digital environment. All the programming language functions, as well as the data given by Random.org, evaluated in this investigation were capable of simulating one or more of these physical generators. Dice rolls were simulated to test a generator’s effectiveness when the range of possible results were limited from 0-100 to 1-6. Coin flip simulations expanded upon this idea by limiting possible results to a binary 0 or 1 while card shuffle simulations aimed to evaluate a generator’s effectiveness when both the number of iterations was reduced (from 500 to 52) and the range of possible results was reduced.

The use of white noise for the purpose of random number generation has been explored across several papers (**See Section 2: Literature Review**). In these examples, the most common method involved identifying the colour code of a sequence of pixels, then correlating the colour ids to bit values in a true random output. For this investigation a similar method was used; however, in place of video data, audio data was used instead. This audio data was then converted to a waveform and the audio levels were used to form a true random sequence of integers. Three sets of audio data were collected for this investigation, with each being recorded at a different location around the Plymouth area. By collecting from three distinct locations, three noticeably different waveforms could be created, as the white noise from all these areas are from different sources. The first location sampled was a busy roundabout, the second was a public park, and the third was next to the sea.

**3.2 C# Implementation**

The C# pseudorandom generators were created in a .NET framework project. Because the development environment uses a different method for generating a base seed in a .NET framework file compared to a .NET core file, it was important that a framework project was used. This was because .NET framework files, much like files in other languages, use the system clock for generating a ‘random’ base seed and being able to use the same method ensured consistency between languages. Due to the object-oriented nature of C#, individual functions were used to house each of the generators. These functions would then be called by the program upon being run. In total nine functions were created. To be able to analyse generator outputs not only by the C# implementations but also by all pseudorandom and true random generators external file storage was a necessity. While there are many file types capable of storing the data given including notepad, CSV, or excel it was decided that JavaScript Object Notation (JSON) files would be used. JSON is standard in many areas of the computing industry and can be read and written to by all the languages sampled in this investigation. These files are also designed to be used for storage of data structures such as arrays or lists which were also used throughout the implementation of the pseudorandom generators.

A screenshot of a computer program

Description automatically generated

*Figure 1. A screenshot of the Main body of the program and the Implementation 1 call*

Before any functions are called, a set of lists are created to hold the various outputs generated. When the function is called, as seen with *RandImplementation1* in Figure 1, the relevant output list is passed by reference into the function, allowing it to be modified outside of the program’s Main body. *ReturnValues*, *ReturnCardValues*, and *ReturnDiceValues* are all lists of types double, string and int that store outputs from the generators. Once a function has been called, a for loop is used to print the contents of the output list to the user before the output list is converted to a JSON object and exported as a JSON file.

A computer screen with green text

Description automatically generated

*Figure 2. A screenshot of the RandImplementation1 function*

*RandImplementation1* uses the *rand.Next()* function and a system generated seed to produce 500 values between 0 and 100. When first called, the *Values* list which is a reference to the *ReturnValues* list is emptied. Then a new Random variable is created which contains the *.Next()* method. By leaving rand without an integer value during declaration, the system will automatically produce a seed value based on the system clock of the PC. A for loop is then set to iterate 500 times and during each iteration a random integer is added to the *Values* list.

A computer screen with green and white text

Description automatically generated

*Figure 3. A screenshot of the RandImplementation2 function*

*RandImplementation2* is nearly identical to the former implementation. The only difference between both functions is in the declaration of *rand*. When deciding on a seed to provide as an alternative to the system clock, the value 30102000 was chosen. This was because the more traditional algorithms such as the Lehmer Generator or the Middle Square Method used six to ten character seeds for optimal calculations. As only the length of the seed determined effectiveness and not the number itself, the author’s birthday (30/10/2000) was considered valid.

A computer screen shot of a program

Description automatically generated

*Figure 4. A screenshot of the RandImplementation3 function*

Unlike the previous rand functions, *RandImplementation3* utilises the C# Cryptography library. Included in this library is the *RandomNumberGenerator* class which provides a cryptographically secure set of bytes. In addition, where previous functions required the ReturnValues list to be provided, RandImplementation3 requires the user to provide the number of iterations desired which is stored in the *size* integer variable. When called, the function will create the *values* list to store the generator output, then by calling the *.GetBytes()* function the generator will produce a *size* length collection of random bytes. Once generation is complete, the values list is returned to the program Main body where the bytes are stored inside the emptied *ReturnValues* list and then exported as a JSON file.

A screenshot of a computer program

Description automatically generated

*Figure 5. A screenshot of the RandCoinSimulation1 function*

*RandCoinSimulation1* used the same *.Next()* function as seen in *RandImplementation1* however by adjusting the desired threshold from 100 to 2, the possible outputs became only 0 (heads) or 1 (tails). As seen previously, upon being called *RandCoinSimulation1* emptied the *Values* list and used a for loop to iterate 500 times. Due to the use of the system clock for seed creation, this function and implementation 1 cannot be completely reproduced, unless a user knows exactly when the code was run and how the clock data can be formatted into a usable seed.

A computer screen with green and white text

Description automatically generated

*Figure 6. A screenshot of the RandCoinSimulation2 function*

As with implementation 2, *RandCoinSimulation2* operates the same as previously, however instead uses the predetermined 30102000 seed. Although an essentially binary output is given for both coin simulations, the ReturnValues double list is used for data storage as the outputs given, unlike in other simulations such as card shuffles, do not require specialised storage. Only two coin simulations were created for the C# implementation. This was due to the output given by the cryptographic generator not being compatible with the specific output limitations of the simulations. While able to produce bytes, the cryptographic method shown in *RandImplementation3* could not be altered to only generate values between 1 and 6.

A computer screen shot of a program

Description automatically generated

*Figure 7. A screenshot of the RandCardSim1 function*

The decision to use JSON files proved valuable again when designing the card shuffle simulations. As well as needing to output to JSON, the unshuffled ‘deck’ used by the card simulation functions was able to be read into the program as a JSON file. The Newtonsoft.JSON library for C# allowed for the creation of the StreamReader object *r*, which took the input deck data and stored it in a manipulatable string list. After this, a for loop was iterated through that would randomly select a *ChosenCard* from the deck and then add that card to the ‘shuffled’ deck. Once the new card had been added to the shuffled deck, it was removed from the input deck via its list position to prevent it from being selected again.

A screen shot of a computer program

Description automatically generated

*Figure 8. A screenshot of the RandCardSim2 function*

Figure 8 shows the *RandCardSim2* function, which much like RandCardSim1 used an input deck JSON file to produce a ‘shuffled’ collection of 52 cards. As with the previous method, the .*Next()* function allowed for pseudorandom card selection, however the 30102000 user given seed was used in place of the system clock determined seed. Only two card simulations were created in the C# implementation because, much like with the coin simulations, the cryptographic method provided was unable to produce outputs suitable for card shuffling. In addition, both card simulations used the same deck input file.

A computer screen shot of a program code

Description automatically generated

*Figure 9. A screenshot of the RandDiceRoll1 function*

A computer screen shot of a code

Description automatically generated

*Figure 10. A screenshot of the RandDiceRoll2 function*

The dice roll simulations were developed with the same method as the coin simulations shown above. A *rand* variable was created using the Random class, first in *RandDiceRoll1* with the system clock seed and then again in *RandDiceRoll2* with the user given seed. A for loop allowed for 500 consecutive iterations, with each iteration producing a value between 1 and 7 (not including 7). The maximum bound available for the generator had to be increased due to a particular quirk of the function, which when given the bounds 1-6 would only generate values up to 5.

**3.3 Python Implementation**

The Python pseudorandom generators were created in a Jupyter Notebook file using Anaconda. The main advantage of working within Jupyter Notebook is that the code can be written into individual cells, which function entirely independently of each other, much the same as the functions used in the previous implementation stage which allowed for a more organised and easily modifiable solution. As with C#, Python uses a procedure to produce seeds for random number generation based on the PC’s system clock. In addition, libraries were used to import the necessary functions for pseudorandom generation. Two main libraries were used in this stage of implementation, Random and NumPy. Random houses all of Python’s standard RNG functions including *Randint()* and *Random()* while NumPy is a staple open source Python library centered around mathematics and data structures. A JSON library was included to allow for reading and writing of JSON files which served to both allow the program to export outputs produced by the generators and import the unshuffled deck data for card simulations. In contrast to the cryptography library used in the C# implementation, all the functions provided by Random and NumPy were able to have their seeds adjusted manually. Five numerical sequences were produced by the Python generators and three sequences were produced for each of the three physical generator simulations.

A screenshot of a computer program

Description automatically generated

*Figure 11. A screenshot of the Python Randint implementation*

Being called almost identically to C#’s *.Next()* method, the *Randint()* function provided by Python is capable of generating any integer between a set of min and max values. This function is provided by the random library which is imported into the program alongside the Json library before any generations begin. In order to store outputs, the *ReturnValues* list is again used. However, unlike in C# when declaring a list, the type of data it stores does not have to be declared, meaning that *ReturnValues* can be used to store any output generated in Python. A 500 iteration for loop encapsulates an append command, which stores the result of a *Randint()* generation within *ReturnValues*. As with previous generators, the minimum and maximum possible values given are 0 and 100. After all iterations are completed, the program displays the contents of *ReturnValues* to the user using a print command. The Json library imported at the beginning of the program then serialises the *ReturnValues* list into a JSON compatible object which is then outputted as with the C# outputs into a JSON file.

A screenshot of a computer

Description automatically generated

*Figure 12. A screenshot of the Python Random implementation*

Python’s random library contains more than just the *Randint()* function. Figure 12 shows the implementation of the *Random()* function, which generates decimal values between 0 and 1. In order to use this function, the seed and random sub-libraries must be imported from the random library used in implementation 1. As this implementation focussed on the system given seed, no value is given when declaring the seed. After declaring the *ReturnValues* list, another 500 iteration for loop is used to fill the list with the outputs from *Random()*. These outputs are again given to the user with a print command and exported to a JSON file. The most noticeable difference between implementation 1 and 2 is that where implementation 1 focused on a limited set of purely integer values, each value given by implementation 2 has 16 decimal places, meaning that the number of unique possible outcomes is significantly higher. Another difference between the functions was that while *Randint()* required user given constraints to determine minimum and maximum values, *Random()* requires no such constraints and will by default produce real numbers between 0 and 1.

A screenshot of a computer

Description automatically generated

*Figure 13. A screenshot of the Python Seeded Random implementation*

The seeded implementation of Python’s *Random()* function was largely unaltered from implementation 2, with the only difference being the use of the user given seed 30102000 when declaring the seed. While other alternative seeds were considered when designing this investigation, especially as with some algorithms such as the middle square having certain seeds that provide more reliably ‘random’ outputs, ultimately the use of a single seed that conformed to all the generators in use was chosen. While it was possible that different values may yield improved sequences, the difference between these outputs when using more modern algorithms found in languages like Python were considered negligible and unnecessary for comparison and evaluation.

A screenshot of a computer code

Description automatically generated

*Figure 14. A screenshot of the NumPy Randint implementation*

Returning to integer based numeric sequences, the second version of the Randint implementation used the NumPy mathematics library. From this library, the sub-libraries seed and Randint were imported, similarly to the method seen in the seeded random implementation. The seed and *ReturnValues* variables were then declared. This implementation did not require a for loop in order to produce 500 generations as the *Randint()* function given by NumPy outputs an array of length n, which can be specified by the user. To conform with the outputs given by previous generators, the *Randint()* function was given a minimum value of 0, maximum value of 100, and number of iterations of 500. The outputs were given to the user using a print command and then converted to a JSON compatible list before being exported. As the structure of the output given by NumPy was an array, and up to this point all generations had been produced as single values, an extra step was needed during this export stage. The use of *.toList()* allowed the program to convert ReturnValues from a non-compatible Python array to *ReturnValuesList*, a list in the same format as seen in previous implementations.

A screenshot of a computer

Description automatically generated

*Figure 15. A screenshot of the NumPy Seeded Randint implementation*

In addition to the Randint implementation, a seeded implementation of the NumPy *Randint()* function was also created, which can be seen in figure 15. Besides the inclusion of the 30102000 seed, no further alterations needed to be made to the Randint implementation.

A screenshot of a computer program

Description automatically generated

*Figure 16. A screenshot of the Randint Coin Simulation implementation*

Figure 16 shows the first method used for simulating coin flips in Python, by adjusting the *Randint()* function to produce outputs of either 0 (heads) or 1 (tails). As this function doesn’t allow for user input when creating a seed, this simulation could only be completed using the PC clock. In addition, as the desired output of this simulation could only be 0 or 1, the *Random()* function could also not be used here, as it would be unable to correctly produce the required values. Besides the adjustment of the maximum value, this implementation featured code identical to that seen in implementation 1.

A screenshot of a computer program

Description automatically generated

*Figure 17. A screenshot of the NumPy Coin Simulation implementation*

As the pseudorandom generator provided by NumPy could produce outputs in the necessary format for coin flips, simulations were made using its version of Randint. Much like with the previous coin simulation implementation, little had to be changed in the code to facilitate the new desired outputs. By changing the maximum value of *Randint()* from 100 to 2, the program would produce a sequence of only 1s and 0s. This sequence was then given to the user and exported to a JSON file.

A screenshot of a computer

Description automatically generated

*Figure 18. A screenshot of the NumPy Seeded Coin Simulation Implementation*

As seen in figure 18, the NumPy *Randint()* function allowed for user customisable seeds, the simulation was repeated as seen previously with the chosen 30102000 seed. No other parameters were adjusted for this simulation, as the min, max, and iterations values stayed identical to the non-seeded implementation. Once generation was completed, the NumPy array was converted to a JSON serialisable list and then exported.

A screenshot of a computer program

Description automatically generated

*Figure 19. A screenshot of the Randint Card Shuffle Simulation Implementation*

Figure 19 shows the implementation of the Randint card shuffle simulation which, when given an unshuffled deck of 52 cards, would draw cards at random until a new shuffled deck was produced. Unlike the other physical generator simulations implemented into Python, the card shuffle sim required an input file before any generations could be run. The unshuffled ‘Deck.JSON’ file seen in all card shuffle implementations throughout this investigation was used. In order to access the contents of the file two commands given by the Json library, open and Json.load, were used. In order to use *Randint()* to replicate a card draw a *SelectedCard* variable was used that would hold the output of *Randint()*, which used a minimum value of 0 and a maximum value of input Deck length -1*.* This *SelectedCard* value was related to the position of each card in the Deck list. The *ReturnValues* list was then appended with the card found in the input Deck at position *SelectedCard*. Following this a pop command was used to remove the card from the Deck of available cards, to prevent it from being chosen again. After this process was completed for all available cards in the input Deck, the shuffled deck was given to the user with a print command. Finally, *ReturnValues* was serialised and exported to a JSON file.

A screenshot of a computer program

Description automatically generated

*Figure 20. A screenshot of the NumPy Card Shuffle Simulation Implementation*

The method used to produce a card shuffle simulation using the NumPy pseudorandom generators was mostly like that of the Randint card shuffle simulation. The input Deck was read into the program, and a *SelectedCard* variable was created. Then, using a 52 iteration for loop, cards were chosen from the input Deck and moved to the *ReturnValues* list. The most noticeable difference in implementation between NumPy and Randint is the use of an if/else statement within the for loop. This statement was required due to a logic error within the NumPy *Randint()* function, which would cause the program to error when trying to move the last card from the input Deck. As such, a check needed to be included that ensured the length of the input Deck was greater than 1, and when the length was equal to 1 the program would manually move the last card.

A screenshot of a computer program

Description automatically generated

*Figure 21. A screenshot of the NumPy Seeded Card Shuffle Simulation Implementation*

Figure 21 shows the NumPy implementation for the seeded card shuffle simulation. Functionally, this method operated the same as the previous implementation, with *SelectedCard* being used as a reference for a cards position in the Deck list. No other factors were adjusted besides the user given value of 30102000 for the generator’s seed.

A screenshot of a computer program

Description automatically generated

*Figure 22. A screenshot of the Randint Dice Roll Simulation Implementation*

In order to effectively evaluate the pseudorandom functions provided in Python, as many requirement-limiting simulations were run as possible. This included simulations for dice rolls, which limit the available outcomes of the generators from 0-100 to 1-6. Figure 22 shows the implementation of a dice roll simulation using *Randint()*. The implementation was based on the *Randint()* numeric sequence generator seen in figure 11, however when appending *ReturnValues* using the *random.randint()* command, the values 1 and 6 were given.

A screenshot of a computer program

Description automatically generated

*Figure 23. A screenshot of the NumPy Dice Roll Simulation Implementation*

This process was repeated with the NumPy *Randint()* function. For this implementation, the minimum and maximum values were adjusted to 1 and 7, with the number of iterations remaining at 500. A max of 7 was needed for this implementation due to the *Randint()* function only producing values between the minimum and the maximum value -1.

A screenshot of a computer program

Description automatically generated

*Figure 24. A screenshot of the NumPy Seeded Dice Roll Simulation Implementation*

As well as limiting the possible outcomes of the NumPy Randint() function, figure 24 shows the seeded simulation implementation which also restricted the seed to exclusively use a user given value.

**3.4 JavaScript Implementation**

The JavaScript pseudorandom generators were programmed in Visual Studio Code and run in a Google Chrome browser. The main body of the program was made in HTML with a section for the generators marked with *<script>* tags that housed the JavaScript code itself. The default function provided by JavaScript being tested in this program was *Math.Random()* which returns an integer output between 1 and x where x is a user given constraint. The use of *Math.Random()* also came with several limitations compared to other languages featured in this investigation. Firstly, the function used can only operate with a system generated seed, not a user given one which limited the amount of testing able to be performed using it. In addition, the generator uses 0 as an unchangeable minimum value for all integer generation, which meant the adjustment of outputs given to accommodate the restrictions given for simulations. Finally, while able to ‘stringify’ lists to JSON format, JavaScript cannot read in or write to JSON files stored locally. Therefore, all outputs had to be copied and written into JSON files manually and the card shuffle simulations could not be conducted due to the inability to read in the unshuffled Deck file. However, despite these limitations, the *Math.Random()* function was able to be tested on its ability to produce numeric integer sequences, simulate dice rolls, and simulate coin flips. While in other implementations functions or unique cells were used to separate the generators featured, due to the reduced number of outputs and format of the language, all three generators were run in the same *<script>* section. In total three outputs were given from the JavaScript program, all of which used the Math.Random() function with various limitations.

A black background with white text

Description automatically generated

*Figure 25. A screenshot of the HTML body of the JavaScript Implementation*

As mentioned above, the main body of the JavaScript Implementation was written in HTML which, when run, would produce a simple web page capable of displaying generator output to the user. Although no self-created functions were used in this program, the main body and the generator code were kept separate both to avoid confusion during implementation and due to the necessity to hold the generator code within separate *<script>* tags. The body of the program contains a header tag (h2) detailing the purpose of the web page and several paragraph tags (p) which describe the generator in use and display the outputs given.

A computer screen shot of a code

Description automatically generated

*Figure 26. A screenshot of the JavaScript Random Integer Implementation*

The first generator held within the *<script>* tags is the random integer generator which used *Math.Random()*. To store outputs given the *OutputValues* list was created. As with previous languages implementations, a for loop was used to iterate the *Math.Random()* function and store the results 500 times. Only one parameter was given to the function, a maximum value of 101 which would cause the program to produce integer values between 0 and 100. After that a for loop was used to fill the variable *text* with the contents of *OutputValues*. This was done so that the *output* paragraph tag could be updated to contain *text* and, by extension, the results of *Math.Random()*. Once the values were printed to the user, a stringify command allowed *OutputValues* to be given in a JSON compatible form which was copied manually to an output JSON file. Although only the integer generator outputs would be shown on the web page, all other JavaScript implementations would use the stringify method to produce exportable data.

A computer screen with text

Description automatically generated

*Figure 27. A screenshot of the JavaScript Coin Simulation Implementation*

Figure 27 shows the implementation of a coin flip simulation in JavaScript using the *Math.Random()* function. Although the method used remained the same, there was a core difference when creating the generator. While the integer generator used *Math.floor* the coin simulation used *Math.round*. This was because by default if *Math.Random()* isn’t given any parameters it will produce decimal values between 0 and 1. This would mean that outputs given originally wouldn’t be usable for this simulation, but the results could be rounded to the nearest integer to correctly identify them as either heads or tails. Using this method for determining heads or tails meant that an additional form of *Math.Random()* could be examined. Once the outputs had been adjusted to fit the requirements of the simulation, stringify was used to produce a JSON compatible file.

A computer screen with text and numbers

Description automatically generated

*Figure 28. A screenshot of the JavaScript Dice Simulation Implementation*

The JavaScript implementation of a dice roll simulation returned to Math.floor when creating its generator. A maximum value of 6 was given in order to produce values between 0-5. These values were then increased by 1 when being added to the *DiceSimValues* list to correctly replicate the traditional 1-6 die faces.

**3.5 Lehmer Generator Implementation**

The Lehmer Generator, first created by Park and Miller in 1988, was originally written in Pascal. Due to its widespread use in languages like C++ today and the fact that the algorithm used by the generator which, when replicated correctly, will not produce different outcomes depending on the language used it was decided that creation of the Lehmer Generator could be completed in any language of choice. C# was the chosen language for this implementation, because of both familiarity regarding the language and the similarities between it and the main home of Lehmer generation currently, C++. The program was formatted much the same way as the C# implementations, with a main body responsible for making function calls and printing outputs for the user and a series of functions each housing a different implementation of the minimal standard generator. In total four different implementations were created, two based on integer values and two based on double or real values. All outputs given from these generators were in double form and with the exception of Integer Version 2, that output values between 10 and -10, were all between 1 and -1.

A computer screen with text on it

Description automatically generated

*Figure 29. A screenshot of the Lehmer Generator Implementation Main Body*

Much like previously, the *ReturnValues* list was used here to store the outputs given from the functions used. When called each function, as seen with IntegerVer1 in figure 29, calls *ReturnValues* by reference meaning that each function has direct access to the list instead of needing four different lists to store each generator’s output. *ReturnValues* is also given to the user through a print command and exported to a JSON file using the serialise functions provided by the Newtonsoft.Json library.

A screenshot of a computer program

Description automatically generated

*Figure 30. A screenshot of the IntegerVer1 Implementation*

Figure 30 shows IntegerVer1, a function housing the first integer based Lehmer algorithm. After ensuring that the *Values* list, which is linked to *ReturnValues*, is empty the constants A (the seed multiplier) and M (the modifier) are declared. As mentioned previously (**see Section X**) the given seed multiplier used for this investigation is the newer 48271 because this value provides a more accurate simulation of true random sequencing. The seed is also hard coded for all versions of the Lehmer algorithm used as the algorithm doesn’t have a default seed and the method outlined by Park and Miller states that the only requirement for a reliable seed is a length of 8-10 characters. Within a 500 iteration for loop, the algorithm declares a random value by taking the result of A multiplied by the seed then dividing by M to determine the modulus. This value is then converted to a double (this was done due to issues with C#’s mathematics logic causing all non-converted values to equal 0), divided by M, and stored in the *Values* list. As the seed variable was interacted with and modified during these calculations, during subsequent iterations this value becomes the new base seed and is modified by further calculations leading to a constantly changing pseudorandom sequence of values and seeds.

A computer screen shot of a program

Description automatically generated

*Figure 31. A screenshot of the RealVer1 Implementation*

The variables used for the RealVer1 function, besides being updated from integers to doubles, were the same as in the IntegerVer1 function. When first called, the Values list was emptied and in addition to A, M, and seed a temp variable was declared for use in calculations. A for loop containing a modified Lehmer algorithm was then called. First, temp was altered to hold the result of A \* seed. Then the seed variable was used to store the results of temp - M \* the truncated result of temp / M. Further division of seed by M was calculated and held within *Values* as the first of 500 pseudo-randomly generated numbers.

A computer screen shot of a program

Description automatically generated

*Figure 32. A screenshot of the IntegerVer2 Implementation*

Figure 32 shows the IntegerVer2 function which, compared to its version 1 counterpart, featured a noticeably more complex algorithm. Additional integer variables such as Q and R which served as combinations of A and M were primarily used in calculations for the new *test* variable. The purpose of *test* within this algorithm was to provide error checking facilities in conjunction with an if/else statement that determined if the seed value being used for pseudorandom generation could be correctly represented by a PC in 32 bits, theoretically improving the performance of the generator. If the test was greater than 0, then the seed would take on the value of test, else the seed would take on the value of test + M. The remainder of the algorithm remained unchanged, with a converted seed / M being stored in the *Values* list.

A screenshot of a computer program

Description automatically generated

*Figure 33. A screenshot of the RealVer2 Implementation*

The real number implementation was also altered significantly compared to the previous version. As with IntegerVer2 additional variables were provided for seed modification and an if/else statement was used to ensure correct bit representation by the PC. Unlike the integer equivalent of this algorithm, truncation was used during the calculation of hi, one of the variables used to determine the value of test. The if/else statement was largely unchanged, only featuring a double value of 0.0 to compare against test instead of an integer 0. As C# didn’t present any logic or syntactic errors during decimal calculation, no conversions needed to be made to ensure that a correct output was given to the *Values* list.

**3.6 Middle Square Method Implementation**

The Middle Square Method originally began as an arithmetic method not a programmed function, unlike other methods presented in this investigation, which meant that there was no specific language deemed preferable for implementation over any other. For the same reasons given in the Lehmer implementation (**see Section 3.5: Lehmer Generator Implementation**), C# was the language chosen when programming this method. The main body of the program contained the entirety of the Middle Square implementation due to the fact that multiple functions weren’t required and there was no need to be able to call the generator independently from the rest of the program. The Json library Newtonsoft.Json was implemented to allow for exporting of data however because of the simplicity of the algorithm no other libraries were needed. Because of the method used by the algorithm for seed manipulation, specific constraints were required to prevent errors (see below) meaning a system given seed would not be compatible for testing. In total only one output was given by the Middle Square implementation which made use of a modified version of the user given seed featured throughout the investigation.

A computer screen with green text

Description automatically generated

*Figure 34. A screenshot of the variables declared for Middle Square Implementation*

As seen in figure 34, one of the main limitations for a viable seed was length due to the risk of overflow error should a value too large be given. For this reason the user given seed 30102000 was shortened for this implementation to 301020. In addition, variables *valueLength*, *valueMiddle* and *valueString* were declared to allow for the extraction of pseudorandom numbers from the Middle Square value (see below). To store all 500 outputs, the *ReturnValues* list was implemented and before any generations could begin *value* was set to the base seed 301020.

A computer screen shot of a program

Description automatically generated

*Figure 35. A screenshot of the Middle Square Implementation*

The first step in the implementation was, as expected, to square *value*. The length of the squared *value* was then recorded, and *value* was converted to a string. This was done so that the length could be checked using an if statement to ensure that it was even as in Von N’s method when generating two-digit results there is no extractable ‘middle’ in an odd length number. The Middle Square method is also able to produce three-digit results, in which case this requirement is flipped, and odd length squares are required. If the number was of odd length, then an additional 0 could be added to the left-hand side. It was this addition that required *value* to be in string form, as a 0 cannot be added to the left-hand side of a numeric variable in any programming language. A substring function could then be used to obtain the middle two digits and convert them back to integer form, so in the case that a pair of middle digits such as ‘06’ are selected the program can store them correctly. Figure 35 also outlined a key issue in the Middle Square implementation caused by the repetition of two-digit seeds leading to the same sequence of numbers repeating in the remaining iterations.

**3.7 Random.org Data Collection**

Ensuring that a range of in-use solutions to digital random number generation was a necessity for this investigation. For this reason, data collection from a site like Random.org that focused on providing consumers a reliable source of ‘random’ sequences was obvious. Unlike with previous sources, no programming had to be done in order to produce the data required. Instead, the 500 iterations could be produced and displayed by the algorithm provided. This sequence could then be formatted and copied into an empty JSON file. In total, three outputs were gathered from Random.org: a random numeric sequence, a coin flip simulation and a dice roll simulation.

A screenshot of a computer

Description automatically generated

*Figure 36. A screenshot of the Random integer Generator page from Random.org*

The Random Integer Generator feature of Random.org was used for this investigation. Much like with the programmed functions seen previously the generator required a number of iterations, a minimum value, and a maximum value. For the numeric sequence a min and max of 1 and 100 was used, for the coin simulation a min and max of 0 and 1 was used and for the dice simulation a min and max of 1 and 6 was used. All three of these sequences were also set to iterate 500 times.

**3.8 Physical True Random Number Generator Data Collection**

The main issue presented when collecting data from physical generators was not the set-up but instead the need to repeatedly perform generations manually. For dice rolls and coin simulations both had to be rolled/flipped 500 times each and the deck of cards needed to be sorted, shuffled, and drawn three times. Although time consuming, this method was ultimately necessary as producing data sets with the exact same dimensions as their digital counterparts allowed for fair evaluation later on in the investigation.