Introduction

The debate around computerised random number generators and algorithms capable of generating supposedly random collections of numbers has been around since the early 1950s. Mathematician and computer scientist John Von Neumann was once quoted saying: *"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin"* (**Von Neumann, 1951**). This is because arithmetical operations for creating random numbers, such as Von N's own middle-square method, are not capable of generating a true random combination of numbers, instead producing pseudorandom sequences.

A core issue faced historically by those looking to analyse random number generators can be explained by Park and Miller: strange and unpredictable is not necessarily random (**Park, Miller, 1988**). Even the most predictable pseudorandom generators, such as IBM's notorious RANDU, are often capable of creating a sequence that at a glance appears to be random. Attempting to define what a random sequence looks like using only human intuition is an impossible task. Everyone has their own biases on what random looks like, from the expected number of sequential repetitions in a sequence to the ratio of heads to tails when flipping coins. The equanimity of your average tosser of coins depends upon a law…which ensures that he will not upset himself by losing too much nor upset his opponent by winning too often (**Stoppard, 1966**) and this extends to all areas of random data evaluation. It is only through visualisation and empirical testing that faults in these generators can be identified.

This project aimed to evaluate a collection of pseudorandom algorithms, to compare them to true random number generators and to test their versatility, seeing how close a pseudorandom algorithm is capable of getting to a true random generator. These tests addressed a variety of algorithms, including methods such as the middle-square, and compared them to physical random number generators, such as a shuffled deck of cards, dice and coins. As well as comparing against true random generators, a collection of empirical tests originally described by Donald Knuth in his 1998 book: *The Art of Computer Programming Volume 2: Semi-numerical Algorithms* (**Knuth, 1998**) was used. An algorithm that ranked highly in these tests will be as 'true to life' as possible and be capable of generating a sequence that comes as close as possible to true random.

Literature Review

Before testing could begin, suitable background research about random number generators was required. The highlighted materials outlined in this section served as the main sources of information regarding areas such as pseudorandom algorithms, background on the creation of digital random number generation, the use of white noise for true random generation, the use of pseudorandom numbers in cryptography and empirical tests that can be used to evaluate randomness.

Donald Knuth's 1998 book *The Art of Computer Programming Volume 2: Semi-numerical Algorithms* (**Knuth, 1998**) served as a useful introduction to the theory behind computerised pseudorandom number generators as well as detailed

explanations of empirical tests capable of estimating the degree of randomness a sequence displays. These include but are not limited to Birthday Spacings, Serial Correlation and Poker tests. Information regarding these tests proved invaluable throughout the analysis portion of this investigation. The book itself is highly regarded among computer scientists and although it was released twenty-five years ago much of the fundamental theory remains valid with the book itself being a revision of Knuth's original works released over twenty years before.

Stephen Park and Keith Miller's 1988 paper *Random Number Generators: Good Ones are Hard to Find* (**Park, Miller, 1988**) was another valuable introduction to random number generators. This paper focused on algorithms used by computer scientists at the time, many of which being highly predictable and rigid in the structure of their 'random' outputs such as RANDU (**Wikipedia (1), 2023**) and the practical implementation of a new generator, later coined the Lehmer generator (**Wikipedia (2), 2023**), which aimed to provide a 'minimal standard' for a reliably random sequence of numbers with comparatively simple code written in Pascal. Although like Knuth's book this paper might seem dated, the Lehmer generator remains relevant today. After a review from computer scientists in 1993 (**Marsagilia, Sullivan, 1993**), the Lehmer generator from Park and Miller's original paper was updated to feature a new base multiplier value, which served to reduce reproducibility and predictability. Both versions of the Lehmer generator are provided as standard in modern languages such as C++11's minstd_rand functions (**Wikipedia (2), 2023**).

A pair of similar papers, *Random Bit Sequence Generation from Image Data* by Yas Abbas Alsultanny in 2008 (**Alsultanny, 2008**) and *Random Number Generated from White Noise of Webcam* by Jer-Min Tsai, I-Te Chen and Jengnan Tzeng in 2009 (**Tsai, Chen, Tzeng, 2009**) both focused on the concept of collecting white noise from a webcam image and using it to generate true random number sequences. These papers proved valuable to the project as the outline provided for each of their practical demonstrations on white noise collection, processing and use in generation. Although these papers made use of video data, whereas the white noise collected in this project is audio based, seeing the steps taken in both experiments gave an indication as to how similar steps could be taken regarding an alternate data source.

Likewise, I-Te Chen's 2013 paper *Random Numbers Generated from Audio and Video Sources* (**Chen, 2013**) also aims to collect and examine random number sequences derived from webcam white noise. The combination of webcam inputs leads to an audio-visual random number generator that produces random sequences from image data as well as taking advantage of audio's influence (**Chen, 2013**). This paper, while focusing on many of the same steps as the ones outlined previously, proved useful as it gave a demonstration of audio white noise being used to generate random number sequences.

Another area of interest is the future of pseudorandom algorithms. While many of the sources cited in this review focus on the history of random number generators, *Evaluation of splitable pseudo-random generators* released in 2015 by Hans Georg Schaathun (**Schaathun, 2015**) looks at future developments surrounding parallel

computing and the creation of parallel number generators. Although the research to be undertaken in this project will not focus on parallel algorithms, the generators outlined could still be adapted to a non-parallel function for testing purposes. The paper uses the serial empirical test described by Knuth (**1998**) which provided a valuable insight into how the test could be used on this project's data.

A new approach to analyze the independence of statistical tests of randomness by Elena Almaraz Luengo, et. al (**Luengo, et. al, 2022**) was a valuable asset when exploring test suite batteries for random numbers such as DIEHARD and DIEHARDER. The paper aimed to outline the main concepts of a variety of suites, including the tests included within them, evaluating the limits of each both in terms of analyzable sequence length and accuracy. Released in 2022, this research presents pros and cons for the most up to date test batteries currently in use, with a focus less on the specifics of each test and instead a wider focus on the performance of the batteries as a whole.

As outlined previously, the successful generation of non-replicable random number sequences is useful for more than just scientific simulations. Cryptography and encryption can both rely heavily on pseudo and true random generators and the paper *Cryptographically secure pseudo-random number generator IP-core based on SHA2 algorithm* by Luca Baldanzi, et. al (**Baldanzi, et. al, 2020**) aims to address this. Due to the sensitive nature of cyber security, the paper features useful details on the need for unpredictability and how random numbers provide secure applications to help combat attacks. This information proved valuable when looking to address potential impacts of this research outside of a purely academic context.

Frederick James and Lorenzo Moneta's 2020 work *Review of High-Quality Random Number Generators* (**James, Moneta, 2020**) helped to address the need for research such as this, and how testing to find high quality generators benefits mathematicians and computer scientists. As well as providing additional background on pseudorandom number generators, it also provides examples of high-quality algorithms including RANLUX and MIXMAX which show the prominence and need for reliable pseudorandom generators.

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.

All the code used for this project can be found at the following GitHub repository: https://github.com/AlexDenman47674/PROJ518

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 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.

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 (3), 2023**) and is used by companies such as Google, Dropbox, and Netflix as well as by data scientists alongside R for data analysis.

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 (4), 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 can't generate large batches of values at once.

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.

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 (5), 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.

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.

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.

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.

3.2 C# Implementation

The C# pseudorandom generators were created in a .NET framework project. 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.

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        //The Rand function provided within the C# IDE generates a seed based around the system clock of the PC being used
        //This can change if the application is a .NET core file, which generates a seed based on a thread specific pseudorandom generator within the system
        //For a closer match to true random, the file format used for this algorithm is a .NET framework
        //In addition, the Random function provided by C# also allows for an inputted seed that will be used in place of the system clock

        //Declares the lists to store generated values
        List<double> ReturnValues = new List<double>();
        List<string> ReturnCardValues = new List<string>();
        List<int> ReturnDiceValues = new List<int>();

        //Implementation 1 of the Random function
        RandImplementation1(ref ReturnValues);

        Console.WriteLine("Implementation 1");

        for (int i = 0; i <= ReturnValues.Count() - 1; i++)
        {
            Console.WriteLine(ReturnValues[i]);
        }

        //Writes data to JSON
        string Randjson1 = JsonConvert.SerializeObject(ReturnValues.ToArray());

        System.IO.File.WriteAllText(@"D:/Github/PROJ518/C#_Rand_Function/RandFunctionOutput/RandVer1.json", Randjson1);
```

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. 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.

```
1 reference
static void RandImplementation1(ref List<double>Values)
{
    //The random class can generate a variety of pseudorandom variables
    //The .next(x) function will generate positive integers between 0 and x

    Values.Clear();

    //By providing no seed value, the rand variable will generate a seed based on the current time
    Random rand = new Random();

    for (int i = 1; i <= 500; i++)
    {
        //Values between 0 and 100 are generated
        Values.Add(rand.Next(100));
    }
}
```

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. 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.

```
1 reference
static void RandImplementation2(ref List<double>Values)
{
    //As with Implementation 1, the .next() function will be used to generate random values
    //However when creating the Random class the control seed 30/10/2000 will be used

    Values.Clear();

    Random rand = new Random(30102000);

    for (int i = 1; i <= 500; i++)
    {
        //Values between 0 and 100 are generated
        Values.Add(rand.Next(100));
    }
}
```

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.

```
1 reference
static byte[] RandImplementation3(int size)
{
    //In additon to Random, C# also provides the RandomNumberGenerator function
    //Unlike the default Random, RandomNumberGenerator is a cryptographic generator
    //To access this generator the System.Security.Crytography library is required

    using (var generator = RandomNumberGenerator.Create())
    {
        //The array values stores the generated bytes created by RandomNumberGenerator
        var values = new byte[size];

        generator.GetBytes(values);

        return values;
    }
}
```

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. 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.

```
1 reference
static void RandCoinSimulation1(ref List<double> Values)
{
    //This function operates the same as the implementation 1 function
    //However the .next() values are adjusted to 0 (heads) and 1 (tails)

    Values.Clear();

    Random rand = new Random();

    for (int i = 1; i <= 500; i++)
    {
        //Values between 0 and 1 are generated
        Values.Add(rand.Next(2));
    }
}
```

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).

```
1 reference
static void RandCoinSimulation2(ref List<double> Values)
{
    //This function operates the same as the simulation 1 function
    //However the standard seed 30/10/2000 is used in place of the system generated seed

    Values.Clear();

    Random rand = new Random(30102000);

    for (int i = 1; i <= 500; i++)
    {
        //Values between 0 and 1 are generated
        Values.Add(rand.Next(2));
    }
}
```

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.

```
2 references
static void RandCardSim1(ref List<string> Values)
{
    //This function operates the same as the implementation 1 function
    //however the .next() values are adjusted after each generation as if picking from a deck of cards
    //Originally the values are generated between 0 and 51 simulating a standard deck of 52 cards
    //Each card is bought into a 'Deck' list by reading from a deck JSON file
    //As cards are being removed from the 'Deck' the .next() max value decreases

    Values.Clear();

    //Creates the Deck list
    List<string> Deck = new List<string>();

    Random rand = new Random();

    int ChosenCard;

    //Reads the Deck JSON into the 'Deck' list
    using (StreamReader r = new StreamReader("D:/Github/PROJ518/C#_Rand_Function/RandFunctionInput/Deck.json"))
    {
        string json = r.ReadToEnd();
        Deck = JsonConvert.DeserializeObject<List<string>>(json);
    }

    for (int i = 0; i <= 51; i++)
    {
        ChosenCard = rand.Next(Deck.Count()-1);
        Values.Add(Deck[ChosenCard]);

        //Once a card is added, it must be removed from the deck
        Deck.RemoveAt(ChosenCard);
    }
}
```

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. 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.

```
0 references
static void RandCardSim2(ref List<string> Values)
{
    //This function operates the same as simulation 1
    //however the control seed is used

    Values.Clear();

    //Creates the Deck list
    List<string> Deck = new List<string>();

    Random rand = new Random(30102000);

    int ChosenCard;

    //Reads the Deck JSON into the 'Deck' list
    using (StreamReader r = new StreamReader("D:/Github/PROJ518/C#_Rand_Function/RandFunctionInput/Deck.json"))
    {
        string json = r.ReadToEnd();
        Deck = JsonConvert.DeserializeObject<List<string>>(json);
    }

    for (int i = 0; i <= 51; i++)
    {
        ChosenCard = rand.Next(Deck.Count() - 1);
        Values.Add(Deck[ChosenCard]);

        //Once a card is added, it must be removed from the deck
        Deck.RemoveAt(ChosenCard);
    }
}
```

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.

```
1 reference
static void RandDiceRoll1(ref List<int> Values)
{
    //This function uses .Next() to simulate a dice roll
    Values.Clear();

    Random rand = new Random();

    for (int i = 1; i <= 500; i++)
    {
        //Values between 1 and 6 are generated
        Values.Add(rand.Next(1,7));
    }
}
```

Figure 9. A screenshot of the RandDiceRoll1 function

```
1 reference
static void RandDiceRoll2(ref List<int> Values)
{
    //This function uses .Next() to simulate a dice roll with the specified seed
    Values.Clear();

    Random rand = new Random(30102000);

    for (int i = 1; i <= 500; i++)
    {
        //Values between 1 and 6 are generated
        Values.Add(rand.Next(1, 7));
    }
}
```

Figure 10. A screenshot of the RandDiceRoll2 function

The dice roll simulations were developed with the same method as the coin simulations shown above. 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. 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.

```
#One Python alternative to C#'s Random is Randint()
#Randint functions almost identically to .Next()
#Two inputs are given, one to determine the minimum value and another the maximum value

import random
import json

#As with C# the ReturnValues array will hold data generated
ReturnValues = []

for x in range(500):
    #randint will generate values between 0 and 100 as with C#
    ReturnValues.append(random.randint(0,100))

print(ReturnValues)

# Serializing json
json_object = json.dumps(ReturnValues, indent=4)

# Writing to json
with open("PythonOutput1.json", "w") as outfile:
    outfile.write(json_object)
```

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. A 500 iteration for loop encapsulates an append command, which stores the result of a *Randint()* generation within *ReturnValues*. The Json library imported at the beginning of the program then serialises the *ReturnValues* list into a JSON compatible object which is then output as with the C# outputs into a JSON file.

```
#A second implementation of random generation in Python is the random module
#The seed can be given or if no seed is provided the system will designate one based on the current system time

from random import seed
from random import random
import json

#This implementation will use the system given seed
seed()

ReturnValues = []

for x in range(500):
    #random() will generate values between 0 and 1
    ReturnValues.append(random())

print(ReturnValues)

# Serializing json
json_object = json.dumps(ReturnValues, indent=4)

# Writing to json
with open("PythonOutput2.json", "w") as outfile:
    outfile.write(json_object)
```

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.

```python
#Repeating implementation 2 with the prechosen seed 30/10/2000

from random import seed
from random import random
import json

#This implementation will use the system given seed
seed(30102000)

ReturnValues = []

for x in range(500):
    #random() will generate values between 0 and 1
    ReturnValues.append(random())

print(ReturnValues)

# Serializing json
json_object = json.dumps(ReturnValues, indent=4)

# Writing to json
with open("PythonOutput3.json", "w") as outfile:
    outfile.write(json_object)
```

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 during declaration.

```python
#The last implementation of random generation is provided by the NumPy library
#While similar, NumPy uses its own implementation of pseudorandom generation

import numpy
from numpy.random import seed
from numpy.random import randint
import json

#This implementation will use the system given seed
seed()

ReturnValues = []

ReturnValues = randint(0,100,500)

print(ReturnValues)

# Serializing json
ReturnValuesList = ReturnValues.tolist()
json_object = json.dumps(ReturnValuesList, indent=4)

# Writing to json
with open("PythonOutput4.json", "w") as outfile:
    outfile.write(json_object)
```

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. 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.

```python
#Repeating implementation 3 with the prechosen seed 30/10/2000

import numpy
from numpy.random import seed
from numpy.random import randint
import json

seed(30102000)

ReturnValues = []

ReturnValues = randint(0,100,500)

print(ReturnValues)

# Serializing json
ReturnValuesList = ReturnValues.tolist()
json_object = json.dumps(ReturnValuesList, indent=4)

# Writing to json
with open("PythonOutput5.json", "w") as outfile:
    outfile.write(json_object)
```

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.

```python
#Randint() can also be used to simulate coin flips
#This is achieved by changing the min and max values to 0 (heads) and 1 (tails)

import random
import json

#As with C# the ReturnValues array will hold data generated
ReturnValues = []

for x in range(500):
    #randint will generate values between 0 and 1 as with C#
    ReturnValues.append(random.randint(0,1))

print(ReturnValues)

# Serializing json
json_object = json.dumps(ReturnValues, indent=4)

# Writing to json
with open("PythonCoinSim1.json", "w") as outfile:
    outfile.write(json_object)
```

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 with the previous generators, Numpy can be used to replicate coin flips

import numpy
from numpy.random import seed
from numpy.random import randint
import json

#This implementation will use the system given seed
seed()

ReturnValues = []

ReturnValues = randint(0,2,500)

print(ReturnValues)

# Serializing json
ReturnValuesList = ReturnValues.tolist()
json_object = json.dumps(ReturnValuesList, indent=4)

# Writing to json
with open("PythonCoinSim2.json", "w") as outfile:
    outfile.write(json_object)
```

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.

```
#Repeating simulation 2 with the prechosen seed 30/10/2000

import numpy
from numpy.random import seed
from numpy.random import randint
import json

seed(30102000)

ReturnValues = []

ReturnValues = randint(0,2,500)

print(ReturnValues)

# Serializing json
ReturnValuesList = ReturnValues.tolist()
json_object = json.dumps(ReturnValuesList, indent=4)

# Writing to json
with open("PythonCoinSim3.json", "w") as outfile:
    outfile.write(json_object)
```

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

As seen in figure 18, the simulation was repeated with the chosen 30102000 seed.

```python
#Randint can be used for simulating card draws as well as coin flips
import random
import json

ReturnValues = []

#As with C# the deck must be read into the 'Deck' list
Deck = []

LoadList = open('Deck.json')
Deck = json.load(LoadList)

SelectedCard = 0

#The deck is then looped through in order to draw every card
for x in range(52):
    SelectedCard = random.randint(0,len(Deck)-1)
    ReturnValues.append(Deck[SelectedCard])

    #The selected card is then removed from the deck
    Deck.pop(SelectedCard)

print(ReturnValues)

# Serializing json
json_object = json.dumps(ReturnValues, indent=4)

# Writing to json
with open("PythonCardSim1.json", "w") as outfile:
    outfile.write(json_object)

LoadList.close()
```

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.

```python
#Numpy can also be used for simulating card draws
import numpy
from numpy.random import seed
from numpy.random import randint
import json

ReturnValues = []
seed()

#As with C# the deck must be read into the 'Deck' list
Deck = []

LoadList = open('Deck.json')
Deck = json.load(LoadList)

SelectedCard = 0

#The deck is then looped through in order to draw every card
for x in range(52):
    #Due to a logic error with the numpy randint function, an if/else statement is used to allow the final value to be stored
    if len(Deck)-1 == 0:
        ReturnValues.append(Deck[0])
    else:
        SelectedCard = randint(0,len(Deck)-1,1)
        ReturnValues.append(Deck[SelectedCard[0]])

    #The selected card is then removed from the deck
    Deck.pop(SelectedCard[0])

print(ReturnValues)

# Serializing json
json_object = json.dumps(ReturnValues, indent=4)

# Writing to json
with open("PythonCardSim2.json", "w") as outfile:
    outfile.write(json_object)

LoadList.close()
```

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 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.

```
#Numpy can also be used for simulating card draws
import numpy
from numpy.random import seed
from numpy.random import randint
import json

ReturnValues = []
#The control seed is used for this simulation
seed(30102000)

#As with C# the deck must be read into the 'Deck' list
Deck = []

LoadList = open('Deck.json')
Deck = json.load(LoadList)

SelectedCard = 0

#The deck is then looped through in order to draw every card
for x in range(52):
    #Due to a logic error with the numpy randint function, an if/else statement is used to allow the final value to be stored
    if len(Deck)-1 == 0:
        ReturnValues.append(Deck[0])
    else:
        SelectedCard = randint(0,len(Deck)-1,1)
        ReturnValues.append(Deck[SelectedCard[0]])

    #The selected card is then removed from the deck
    Deck.pop(SelectedCard[0])

print(ReturnValues)

# Serializing json
json_object = json.dumps(ReturnValues, indent=4)

# Writing to json
with open("PythonCardSim3.json", "w") as outfile:
    outfile.write(json_object)

LoadList.close()
```

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.

```
#By adjusting the parameters randint can be used for dice simulation
import random
import json

#As with C# the ReturnValues array will hold data generated
ReturnValues = []

for x in range(500):
    #randint will generate values between 1 and 6 as with C#
    ReturnValues.append(random.randint(1,6))

print(ReturnValues)

# Serializing json
json_object = json.dumps(ReturnValues, indent=4)

# Writing to json
with open("PythonDiceOutput1.json", "w") as outfile:
    outfile.write(json_object)
```

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

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.

```
#As with the previous generators, Numpy can be used to replicate dice rolls

import numpy
from numpy.random import seed
from numpy.random import randint
import json

#This implementation will use the system given seed
seed()

ReturnValues = []

ReturnValues = randint(1,7,500)

print(ReturnValues)

# Serializing json
ReturnValuesList = ReturnValues.tolist()
json_object = json.dumps(ReturnValuesList, indent=4)

# Writing to json
with open("PythonDiceOutput2.json", "w") as outfile:
    outfile.write(json_object)
```

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.

```
#As with the previous generators, Numpy can be used to replicate dice rolls

import numpy
from numpy.random import seed
from numpy.random import randint
import json

#This implementation will use the pre-chosen seed
seed(30102000)

ReturnValues = []

ReturnValues = randint(1,7,500)

print(ReturnValues)

# Serializing json
ReturnValuesList = ReturnValues.tolist()
json_object = json.dumps(ReturnValuesList, indent=4)

# Writing to json
with open("PythonDiceOutput3.json", "w") as outfile:
    outfile.write(json_object)
```

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.

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math.Random Function</h2>

<p>Math.floor(Math.random() * 100) + 1) returns a random integer between 1 and
100 (both included). The function has been modified to produce 500 iterations</p>

<p>This script is designed to mirror the random number generator provided by Google as both use the Math.random function</p>

<p>Standard JavaScript has no export method to JSON, so instead the array is converted to JSON format using JSON.stringify in console and then manually copied to the output file</p>

<p id="demo"></p>

<p id="output"></p>
```

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.

```
<script>
const OutputValues = [];
for (let i = 1; i <= 500; i++) {

    OutputValues.push(Math.floor(Math.random() * 101));
}

let text = "<ul>";
for (let j = 0; j <= 499; j++) {
  text += "<li>" + j + " : " + OutputValues[j] + "</li>";
}
text += "</ul>";
document.getElementById("output").innerHTML = text;

console.log(JSON.stringify(OutputValues));
```

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()*. As with previous languages implementations, a for loop was used to iterate the *Math.Random()* function and store the results 500 times. 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()*.

```
const CoinSimValues = [];
for (let i = 1; i <= 500; i++) {

    CoinSimValues.push(Math.round(Math.random()));
}

console.log(JSON.stringify(CoinSimValues));
```

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.

```
const DiceSimValues = [];
for (let i = 1; i <= 500; i++) {

  DiceSimValues.push(Math.floor(Math.random() * 6) + 1);
}

console.log(JSON.stringify(DiceSimValues));

</script>
```

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.

3.5 Lehmer Generator Implementation

The Lehmer Generator 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++.

```
0 references
static void Main(string[] args)
{
    //Park and Miller's implementation of the Lehmer Generator or minimal standard generator consists of 4 algorithms
    //Two algorithms dedicated to an integer based generator
    //And another two dedicated to a real number based generator

    //Declares the value to be given
    List<double> ReturnValues = new List<double>();

    //Integer Version 1
    IntegerVer1(ref ReturnValues);

    //Output to JSON
    string Intjson1 = JsonConvert.SerializeObject(ReturnValues.ToArray());

    System.IO.File.WriteAllText(@"D:/Github/PROJ518/Lehmer_Generator/Lehmer_Generator_Output/IntegerVer1Output.json", Intjson1);

    Console.WriteLine("Integer Version 1");
    for (int i = 0; i <= ReturnValues.Count()-1; i++)
    {
        Console.WriteLine(ReturnValues[i]);
    }
```

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

When each function is called, as seen with IntegerVer1 in figure 29, *ReturnValues* is called by reference meaning that each function has direct access to the list instead of needing four different lists to store each generator's output.

```
1 reference
public static void IntegerVer1(ref List<double>Values)
{
    //Integer Versions of the algorithm use an integer based seed instead of a double or float
    //All values are between 1 and -1

    //Empties the list before generation
    Values.Clear();

    //Declares the seed multiplier (A) and the modifier (M) as integers
    //For the algorithms used the original 1988 value of A has been replaced by Park and Miller's updated value to ensure greater accuracy
    const int A = 48271;
    const int M = 214783647;

    //The base seed used for all algorithms featured will remain the same
    //The specific base seed used for testing is the authors date of birth (30/10/2000) as it provided a value large enough for authentic generation
    int seed = 30102000;

    //500 Generations are stored within ReturnValues
    for (int i = 1; i <= 500; i++)
    {
        seed = (A * seed) % M;

        //Since an int seed is used, conversion must take place otherwise all decimals are truncated to 0 before storage
        Values.Add((Convert.ToDouble(seed) / M));
    }
}
```

Figure 30. A screenshot of the IntegerVer1 Implementation

Figure 30 shows IntegerVer1, a function housing the first integer based Lehmer algorithm. 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. 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.

```
1 reference
public static void RealVer1(ref List<double>Values)
{
    //Real Versions of the algorithm use a double type variables
    //All values are between 1 and 0

    //Empties the list before generation
    Values.Clear();

    //As before constant values A and M are created
    //However double is used in place of int
    const double A = 48271.0;
    const double M = 2147483647.0;

    //Double is also used for both the seed and temp values
    double seed = 30102000.0;
    double temp = 0.0;

    //500 Generations are stored within ReturnValues
    for (int i = 1; i <= 500; i++)
    {
        temp = A * seed;
        seed = temp - M * Math.Truncate(temp / M);
        Values.Add(seed / M);
    }
}
```

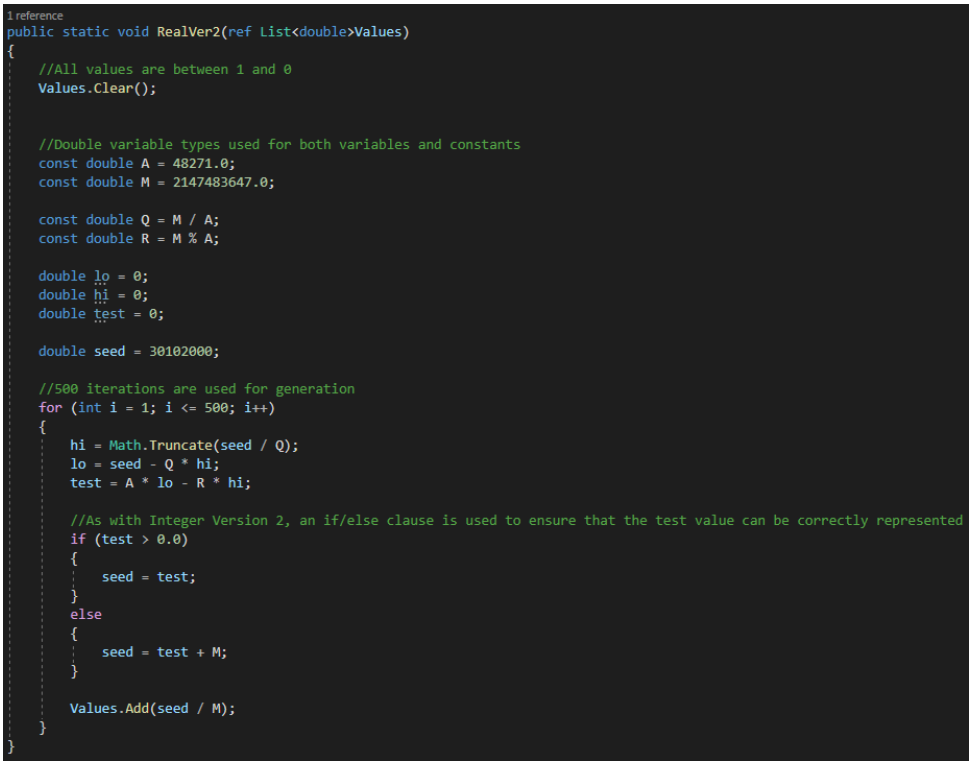
Figure 31. A screenshot of the RealVer1 Implementation

The variables used for the RealVer1 function, besides being updated from integers to doubles, were the same as in the IntegerVer1 function. A for loop containing a modified Lehmer algorithm was then called.

```
1 reference
public static void IntegerVer2(ref List<double>Values)
{
    //All values are between 10 and -10
    Values.Clear();

    const int A = 48271;
    const int M = 214783647;
    int seed = 30102000;

    //Q and R are combinations of A and M that are used in the seed and test modification process
    const int Q = M / A; //Q is the result of M div A
    const int R = M % A; //R is the result of M mod A

    int lo = 0;
    int hi = 0;
    int test = 0;

    for (int i = 1; i <= 500; i++)
    {
        //Q relates to the modification of the base seed into a high and low value
        hi = seed / Q;
        lo = seed % Q;

        //R relates to the creation of test, based on further modification of hi and lo
        test = A * lo - R * hi;

        //If statement used to ensure that test will never be a value that cannot be correctly represented with 32 bits
        if (test > 0)
        {
            seed = test;
        }
        else
        {
            seed = test + M;
        }

        //Again due to the nature of int variables, conversion must be made to double to correctly store results
        Values.Add(Convert.ToDouble(seed) / M);
    }
}
```

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.



```
1 reference
public static void RealVer2(ref List<double>Values)
{
    //All values are between 1 and 0
    Values.Clear();


    //Double variable types used for both variables and constants
    const double A = 48271.0;
    const double M = 2147483647.0;

    const double Q = M / A;
    const double R = M % A;

    double lo = 0;
    double hi = 0;
    double test = 0;

    double seed = 30102000;

    //500 iterations are used for generation
    for (int i = 1; i <= 500; i++)
    {
        hi = Math.Truncate(seed / Q);
        lo = seed - Q * hi;
        test = A * lo - R * hi;

        //As with Integer Version 2, an if/else clause is used to ensure that the test value can be correctly represented
        if (test > 0.0)
        {
            seed = test;
        }
        else
        {
            seed = test + M;
        }

        Values.Add(seed / M);
    }
}
```

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.

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.

```
0 references
static void Main(string[] args)
{
    //The Middle Square Method works by multiplying a number by itself then selecting n digits from the result to be the generated value
    //This value is then multiplied by itself to continue the algorithm
    //Values generated are between 1 and 100

    //Set up variables before generation
    //A shorter seed value was required as multiplication of 30102000 led to an overflow error
    const int seed = 301020;
    int value = 0;
    int valueLength = 0;
    int valueMiddle = 0;
    string valueString = "";
    List<int> ReturnValues = new List<int>();

    value = seed;
```

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.

```
//Also to keep results fair, all generators use the same number of iterations
for (int i = 1; i <= 500; i++)
{
    value = value * value;
    valueLength = value.ToString().Length;

    //Middle values cannot be extracted from odd length squares
    //Padding using 0 to make the length even is required for successful generation
    valueString = Convert.ToString(value);

    if (valueLength % 2 != 0)
    {
        valueString = valueString.Insert(0, "0");
        valueLength = valueString.Length;
    }

    //Using substrings, the middle 2 digits of the valueString variable can be obtained
    valueMiddle = Convert.ToInt32(valueString.Substring((valueLength / 2) - 1, 2));

    Console.WriteLine(valueMiddle);

    //All generated values are stored in the ReturnValues list
    //On inspection of the outputs given, the flaws of this algorithm become clear
    //Repetitions begin with this seed after just 11 iterations
    ReturnValues.Add(valueMiddle);

    value = valueMiddle;
}
```

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. If the number was of odd length, then an additional 0 could be added to the left-hand side. A substring function could then be used to obtain the middle two digits and convert them back to integer form.

3.7 Random.org Data Collection

Ensuring that a range of in-use solutions to digital random number generation were used 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.

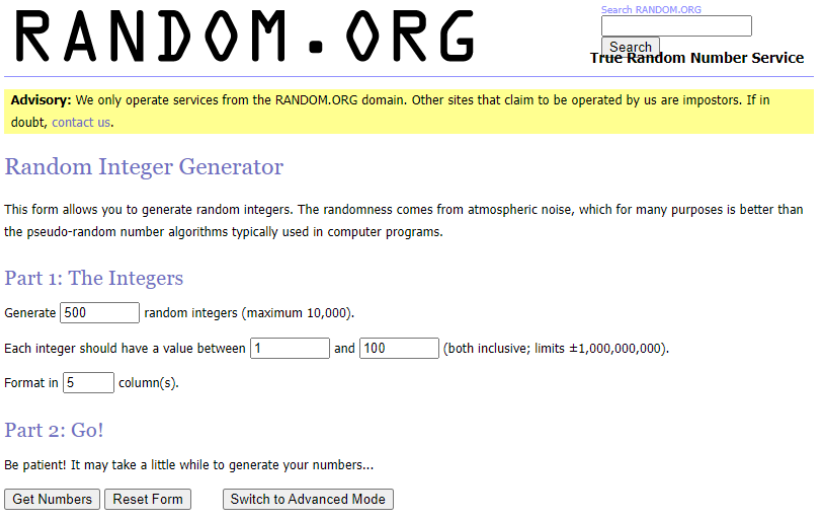


Figure 36. A screenshot of the Random integer Generator page from Random.org

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.

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 in the investigation.

Figure 37. An image of the coin used for true random generation

To produce valid coin flip data a standard 50 pence piece was used. The coin itself was in good condition, with no rust or artifacts present that could unfairly add weight to either side. A result of either 0 (Heads) or 1 (Tails) was recorded after the coin was flipped.

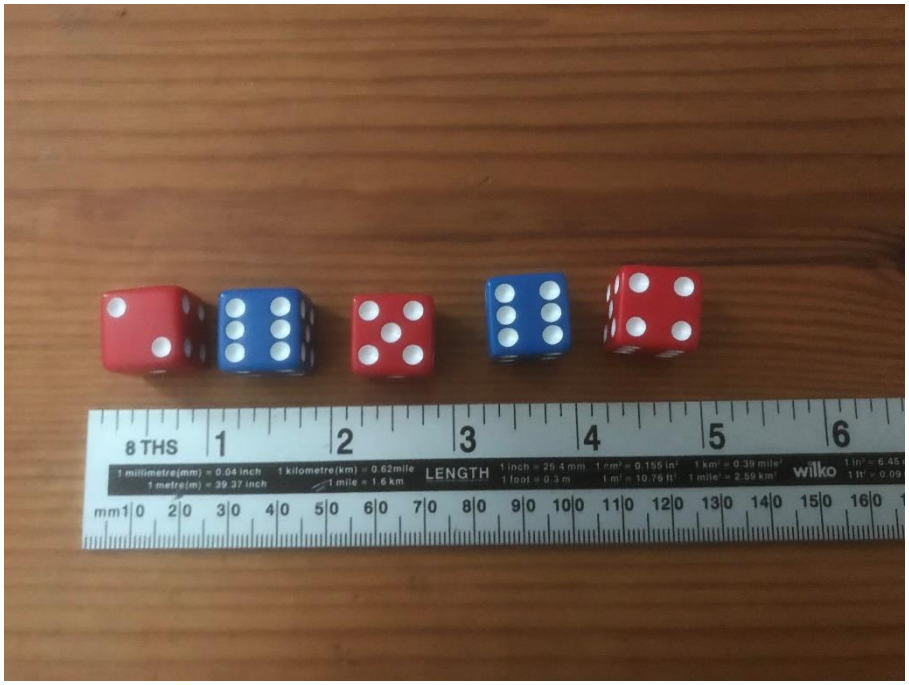


Figure 38. An image of the dice used for true random generation

To produce valid dice roll data five plastic classic 6-sided dice were used. These dice were taken from the box of the game Risk and were all in good condition. During data collection dice rolls were performed in batches of five to substantially reduce the time spent collecting data. While this could have altered the possible results as each

roll was not being made independently, the data being collected both physically and digitally was designed to simulate gameplay conditions in which batch rolling dice is commonplace.

Figure 39. An image of the cards used for true random generation

To produce valid card shuffle data a regular 52 card pack of playing cards were used. No cards were removed, weighted, or marked in any way. Before any shuffle data was collected the pack was sorted into a designated order, matching the input deck file provided for the digital generators. The deck was sorted into suits (the order being Spades, Hearts, Clubs, Diamonds) and each suit was arranged in ascending order from Ace to King. This ‘input’ deck was then given to three individuals with different shuffle techniques and those individuals were given 20 seconds to shuffle the deck as thoroughly as possible. After each shuffle the new card order was recorded, and the deck returned to pre-shuffle order.

3.9 White Noise Data Collection

In addition to the previous physical generators, white noise was collected for this investigation to provide another source of true random numeric sequences. To do this audio was recorded for a minute at three locations around the Plymouth area, one by a busy roundabout, one by the sea, and lastly one in a public park. The audio samples were then downloaded as wav files and converted into waveforms using Python.

```python
#In order to use white noise for random number generation, first the wav files must be read and processed
#Python features a suite of tools that enable the reading of wav files and generation of waveforms
#so will be the chosen language for this stage of data collection
import wave
import numpy as np
import sys
import json

#The roundabout whitenoise file is read first
Roundabout_wav = wave.open('Roundabout.wav', 'r')

#Extracts the raw audio from the Roundabout wav file
signal = Roundabout_wav.readframes(-1)
signal = np.frombuffer(signal, "Int16")
fs = Roundabout_wav.getframerate()

Time = np.linspace(0, len(signal) / fs, num=len(signal))

#Plotting the waveform
import matplotlib.pyplot as plt
plt.figure(1)
plt.title("Signal Wave for Roundabout.wav")
plt.plot(Time, signal)
plt.show()

#Extracting the data from the wav file
print(signal)
```

Figure 40. A screenshot of the Waveform Conversion algorithm

Figure 40 shows the conversion of the Roundabout.wav file which had its signal data and framerate collected to produce a graphable waveform. The *signal* variable holds the audio levels of the file which make up an extractable true random integer sequence.

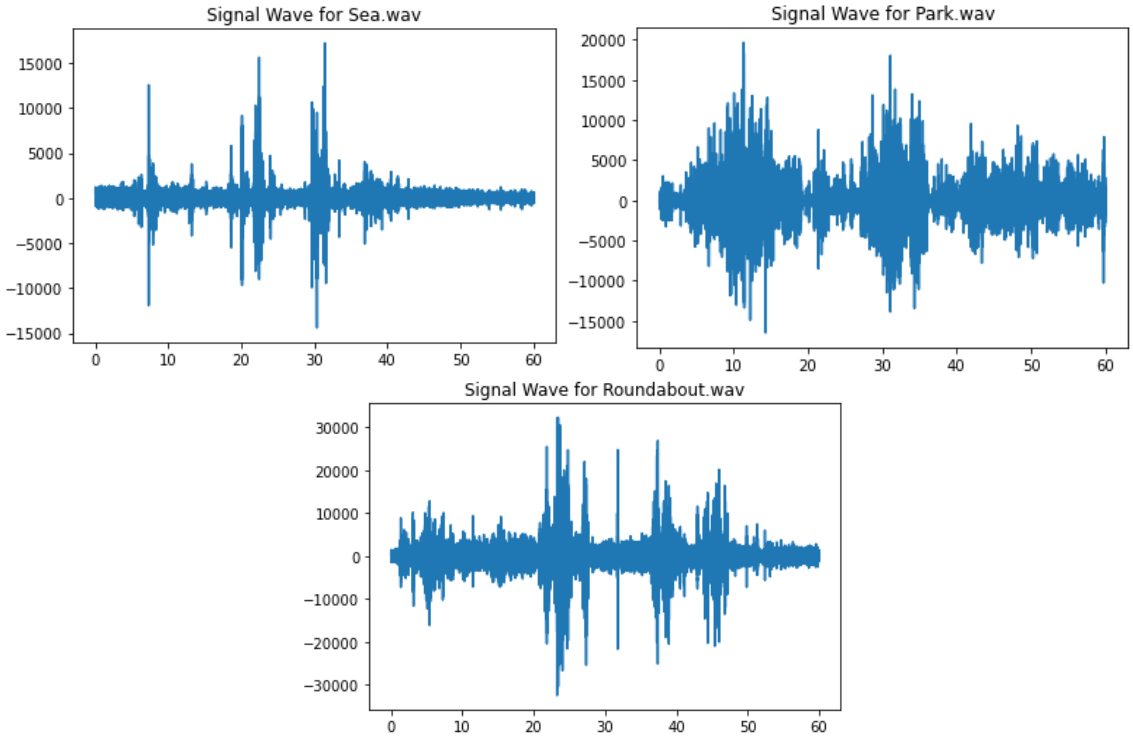


Figure 41. The waveforms for the Roundabout, Sea, and Park data

Figure 41 shows the three waveforms produced by the program. The Sea and Park signal wave values vary between 15000 and -15000 while unsurprisingly the Roundabout signal wave values have a higher range, with values between 30000 and -30000.

```python
# Serializing json
SignalList = signal.tolist()

#The signal array contains far more data than any of the other generators have produced
#Two different outputs are given, a cut list which contains 500 numbers and an uncut list which contains all generated values
SignalListCut = SignalList[:500]
json_object = json.dumps(SignalList, indent=4)
json_object_cut = json.dumps(SignalListCut, indent=4)

# Writing to json
with open("RoundaboutData.json", "w") as outfile:
    outfile.write(json_object)

with open("RoundaboutDataCut.json", "w") as outfile:
    outfile.write(json_object_cut)
```

Figure 42. A screenshot of the Waveform output algorithm

Once the signal data had been graphed, it could be serialised into a JSON compatible list. Figure 42 shows this process being completed with the Roundabout data set.

Data Analysis

4.1 Analysis Methodology

With all required data collected the next step was analysis. In total 8 empirical tests of randomness were performed. These consisted of:

- The Chi-Squared Test
- The Kolmogorov-Smirnov Test
- The Serial Test
- The Gap Test
- The Poker Test
- The Runs Test
- The Serial Correlation Test
- The Birthday Spacings Test

The data analysis was completed in R primarily due to the facilities provided by the language for handling and visualising datasets as well as its wide array of test libraries that provided the functions necessary to produce the test suite shown above. Additional libraries such as ggplot2 and rjson also made R the obvious choice for analysis as the JSON datasets could easily be imported, processed, and graphed.

```r
library(rjson)
library(plyr)
library(dplyr)
library(ggplot2)
library(dgof)
library(randtoolbox)
library(randtests)
library(EnvStats)
library(CryptRndTest)

#The working directory is set to allow access to stored JSON files
setwd("D:/Github/PROJ518/C#_Rand_Function/RandFunctionOutput")

#Retrieving the Dice Sim data from the C# output
CDiceSim1values <- fromJSON(file = "C#DiceSim1.json")
CDiceSim2values <- fromJSON(file = "C#DiceSim2.json")

print(CDiceSim1values)
print(CDiceSim2values)

#Retrieving the Dice Sim data from the Python output
setwd("D:/Github/PROJ518/Python_Rand_Function/Python_Output")
PDiceSim1values <- fromJSON(file = "PythonDiceOutput1.json")
PDiceSim2values <- fromJSON(file = "PythonDiceOutput2.json")
PDiceSim3values <- fromJSON(file = "PythonDiceOutput3.json")

print(PDiceSim1values)
print(PDiceSim2values)
print(PDiceSim3values)
```

Figure 43. The R analysis program importing libraries and JSON data

4.2 Chi-Squared Test of Dice and Coin Simulation Data

The first test used in this investigation was the Chi-Squared test, which is an empirical test designed to calculate a V value from a sequence of random numbers and compare that value to a distribution table to determine the probability that such a sequence could be produced. The main caveat of the Chi-Squared test that limited its use to only dice and coin data is the distribution table which only considers sequences with up to 101 exactly potential outputs. As all the pseudorandom sequences sampled in this investigation had a minimum of 100 potential outputs, it wasn't feasible to apply Chi-Squared testing to them.

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

χ^2 = chi squared

O_i = observed value

E_i = expected value

*Figure 44. The Chi-Squared Equation (**Google, 2023**)*

Chi-square Distribution Table

d.f.	.995	.99	.975	.95	.9	.1	.05	.025	.01
1	0.00	0.00	0.00	0.00	0.02	2.71	3.84	5.02	6.63
2	0.01	0.02	0.05	0.10	0.21	4.61	5.99	7.38	9.21
3	0.07	0.11	0.22	0.35	0.58	6.25	7.81	9.35	11.34
4	0.21	0.30	0.48	0.71	1.06	7.78	9.49	11.14	13.28
5	0.41	0.55	0.83	1.15	1.61	9.24	11.07	12.83	15.09
6	0.68	0.87	1.24	1.64	2.20	10.64	12.59	14.45	16.81
7	0.99	1.24	1.69	2.17	2.83	12.02	14.07	16.01	18.48
8	1.34	1.65	2.18	2.73	3.49	13.36	15.51	17.53	20.09
9	1.73	2.09	2.70	3.33	4.17	14.68	16.92	19.02	21.67
10	2.16	2.56	3.25	3.94	4.87	15.99	18.31	20.48	23.21
11	2.60	3.05	3.82	4.57	5.58	17.28	19.68	21.92	24.72
12	3.07	3.57	4.40	5.23	6.30	18.55	21.03	23.34	26.22
13	3.57	4.11	5.01	5.89	7.04	19.81	22.36	24.74	27.69
14	4.07	4.66	5.63	6.57	7.79	21.06	23.68	26.12	29.14
15	4.60	5.23	6.26	7.26	8.55	22.31	25.00	27.49	30.58
16	5.14	5.81	6.91	7.96	9.31	23.54	26.30	28.85	32.00
17	5.70	6.41	7.56	8.67	10.09	24.77	27.59	30.19	33.41
18	6.26	7.01	8.23	9.39	10.86	25.99	28.87	31.53	34.81
19	6.84	7.63	8.91	10.12	11.65	27.20	30.14	32.85	36.19
20	7.43	8.26	9.59	10.85	12.44	28.41	31.41	34.17	37.57
22	8.64	9.54	10.98	12.34	14.04	30.81	33.92	36.78	40.29
24	9.89	10.86	12.40	13.85	15.66	33.20	36.42	39.36	42.98
26	11.16	12.20	13.84	15.38	17.29	35.56	38.89	41.92	45.64
28	12.46	13.56	15.31	16.93	18.94	37.92	41.34	44.46	48.28
30	13.79	14.95	16.79	18.49	20.60	40.26	43.77	46.98	50.89
32	15.13	16.36	18.29	20.07	22.27	42.58	46.19	49.48	53.49
34	16.50	17.79	19.81	21.66	23.95	44.90	48.60	51.97	56.06
38	19.29	20.69	22.88	24.88	27.34	49.51	53.38	56.90	61.16
42	22.14	23.65	26.00	28.14	30.77	54.09	58.12	61.78	66.21
46	25.04	26.66	29.16	31.44	34.22	58.64	62.83	66.62	71.20
50	27.99	29.71	32.36	34.76	37.69	63.17	67.50	71.42	76.15

*Figure 45. The Chi-Squared Distribution Table (**University of Queensland, 2023**)*

The equation to determine the V value, seen in figure 44, was produced in R manually following the method seen in *The Art of Computer Programming Volume 2: Semi-numerical algorithms* (**Knuth, 1998**).

```r
#Performing the Chi Squared equation
#The Chi Squared equation: V = ((Yn - np)^2 / np) + ((Yn+1 - np)^2 / np)  + ...

V_CDice1 <- ((CDice1[1, 2] - np)^2 / np) + ((CDice1[2, 2] - np)^2 / np) + ((CDice1[3, 2] - np)^2 / np) +
print(V_CDice1)
V_CDice2 <- ((CDice2[1, 2] - np)^2 / np) + ((CDice2[2, 2] - np)^2 / np) + ((CDice2[3, 2] - np)^2 / np) +
print(V_CDice2)
```

Figure 46. A screenshot of the Chi-Squared equation using C# Dice Simulation data

Figure 46 shows the implementation of the Chi-Squared equation into R using dice simulation data. Every outcome in each dataset has its observed value compared against its expected value producing a V value for each dataset.

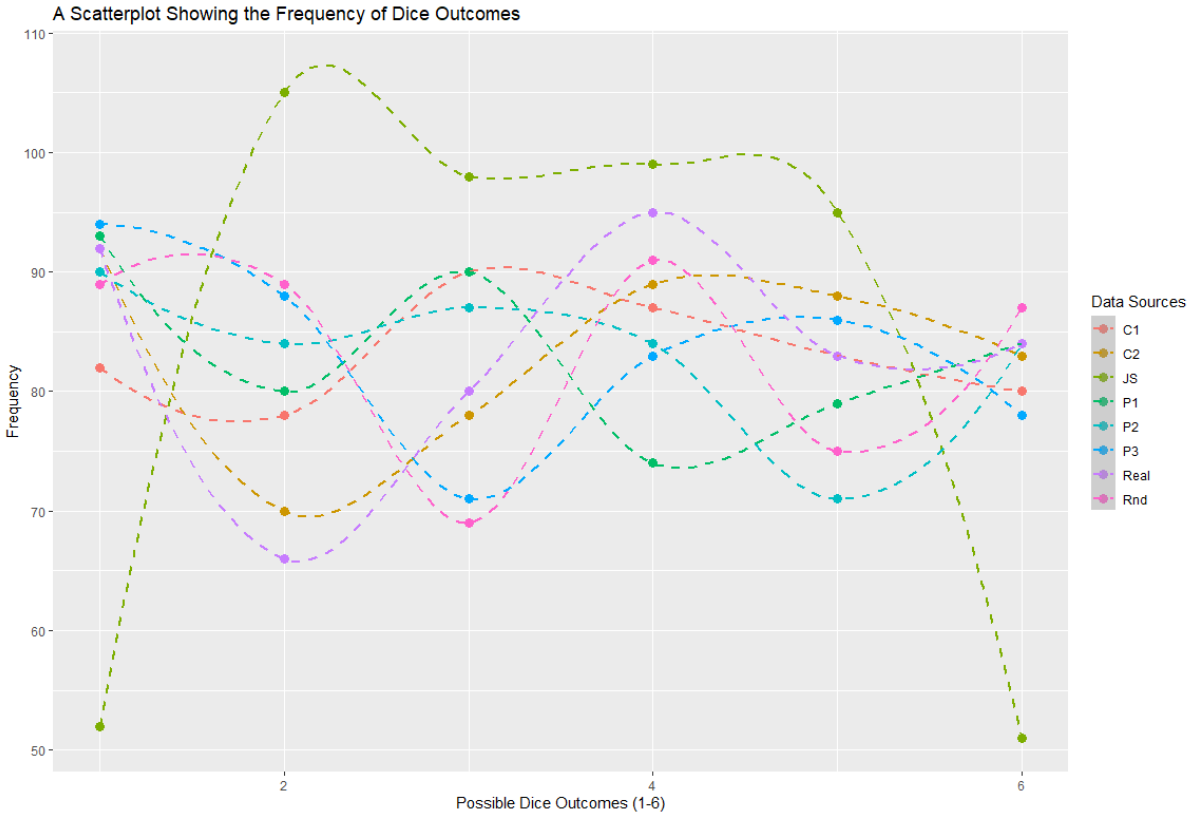


Figure 47. A Scatterplot Showing the Frequency of Dice Outcomes

The frequency of observed values for each of the dice datasets is shown in figure 47. The most immediately noticeable trend is given by the JavaScript dataset which features a noticeably lower frequency of 1s and 6s but the largest frequency of 2s,3s,4s and 5s with all these outcomes occurring significantly more than expected. The remaining datasets all followed a similar trend, with the observed frequency of all possible outcomes occurring between 65 and 95 times.

Data Source	V Value
C# Unseeded Rand	1.192
C# Seeded Rand	4.024
Python Randint	3.064
NumPy Unseeded Randint	2.536
NumPy Seeded Randint	3.88
JavaScript Rand	37.12

Random.org Data	4.936
Real Data	6.28

Figure 48. A Table of Results for the Chi-Squared Test of Dice Roll Data

To compare the V values shown in figure 48 to the distribution table the degrees of freedom must be calculated. This can be done simply by subtracting 1 from the number of possible outcomes (k), in this case producing 5 degrees of freedom. A suitable random sequence is found between the .95 and .1 distributions while V values closer to .995 or .01 are considered too likely or too unlikely to be viable.

The C# unseeded rand implementation placed between the .95 and .9 distributions allowing it to be considered satisfactorily random. The seeded rand implementation placed between .9 and .1 which while closer to the expected values is still satisfactorily random.

In comparison to the C# data, all three Python sequences rated between the unseeded and seeded Rand data with no value scoring above or below the previous results. The Python Randint implementation placed between the .9 and .1 distributions classifying it as suitable. Both versions of the NumPy implementation also placed between the .9 and .1 distributions. Interestingly for Python even the mathematics centric NumPy library was unable to match C# and reach a distribution between .9 and .95.

The JavaScript dataset placed in the .01 distribution. This was by far the largest V value scored by any of the generators surveyed and as a result the dataset was considered unsuitable as a random number source. The highly unlikely nature of this data could also be seen in figure 47 in which the cause of this value can be assumed to be from the low frequencies recorded for outputs of 1 or 6.

The Random.org dataset was placed between the .9 and .1 distributions. This dataset's V value being only slightly higher than the Python or C# generators also presents an interesting argument regarding the validity of the pseudorandom generators. Assuming the data given by Random.org is truly random, then it's possible the pseudorandom generators surveyed are capable of effectively simulating almost-true random conditions.

The real dice dataset placed between the .9 and .1 distributions. Much like the Random.org dataset, this placing was not unexpected as a true random generator will more often than not produce results with a good level of 'reliable' randomness. Besides JavaScript, the real dice scored the highest V value out of the generators sampled although since only one set of 500 samples were collected, it is possible that different rolling methods could have produced a noticeably different V value. Regardless the dice and rolling method used can be classified as satisfactorily random.

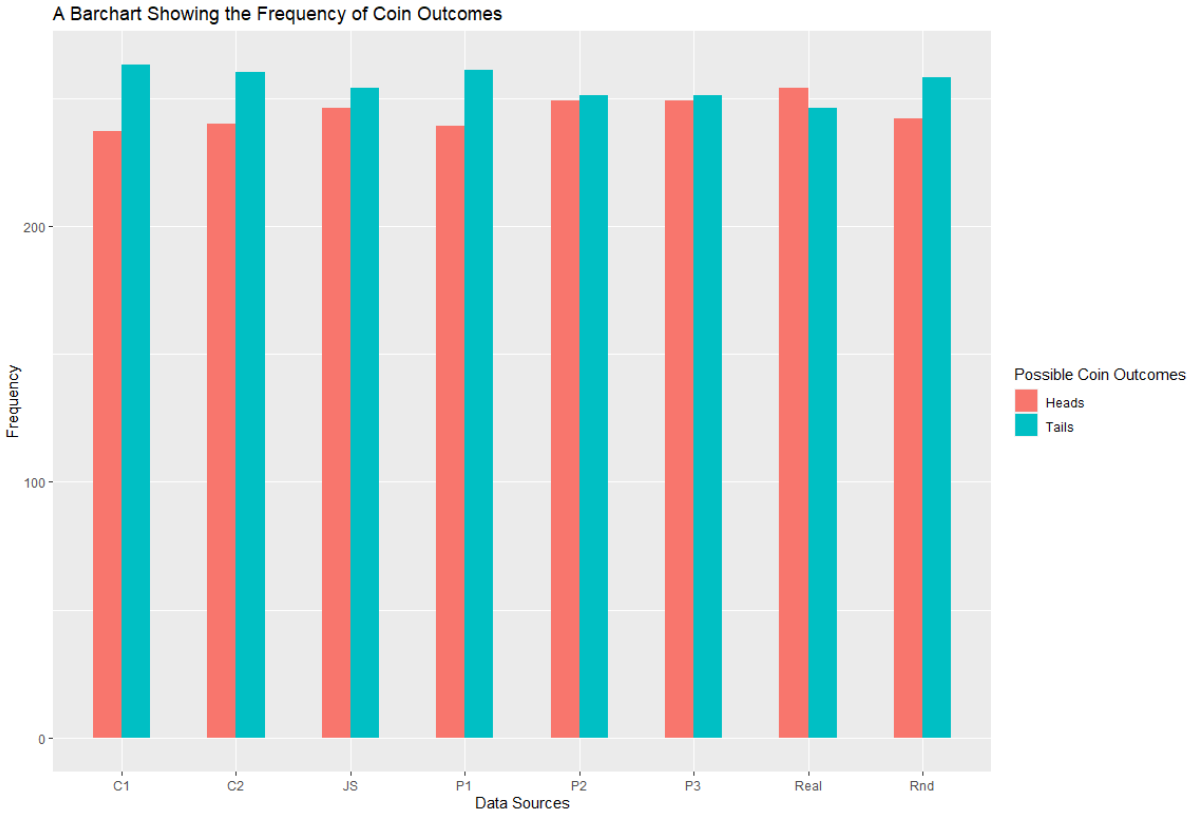


Figure 49. A Bar Chart Showing the Frequency of Coin Outcomes

The Chi-Squared testing of the coin data was completed the same as with the dice data. Observed values of heads and tails for each data source were compared against the expected values (np = 250) and when analysing this result with the distribution table 1 degree of freedom was used.

The frequency of observed values for each data source is shown in Figure 49. Interestingly, with the exception of the real coin values, every generator sampled produced more tails than heads.

Data Source	V Value
C# Unseeded Rand	1.352
C# Seeded Rand	0.8
Python Randint	0.968
NumPy Unseeded Randint	0.008
NumPy Seeded Randint	0.008
JavaScript Rand	0.128
Random.org Data	0.512
Real Data	0.128

Figure 50. A Table of Results for the Chi-Squared Test of Coin Flip Data

The C# datasets were both placed between the .9 and .1 distribution again allowing the dataset to be considered acceptably random. The use of the binary restraint on possible outputs causing neither to fall into the lower .95-.9 distribution could imply that the use of limitations on the generator does impact performance.

The Python datasets were all placed between the .9 and .1 distribution which remained consistent with the dice simulation results. The NumPy implementations were able to produce the exact same results and had the same V value which enforces the idea that the restriction on possible outputs has a noticeable impact on generator performance.

The JavaScript dataset was placed between the .9 and .1 distribution. Compared to the dice simulation V this is a significant improvement for the JavaScript implementation as in this test it was considered satisfactorily random. No pseudorandom generator is designed to perform optimally in all tests or simulations and the improvement in distribution showed that the JavaScript generator could still be considered valid for random number generation.

The Random.org dataset placed between the .9 and .1 distribution. As with the dice simulation data, this result was not unexpected for this data source and the consistent satisfactorily random output shown supports the idea that the data gathered is from a true random source.

The real coin dataset scored identically to the JavaScript dataset. As seen with the Python data repeated V values are entirely possible however unlike what was shown before these values came from completely different sources. It must be considered that the limited possible outcomes of a coin flip present a scenario where similar or matching outputs between data sources is far more likely.

4.3 Kolmogorov-Smirnov Test of Empirical Distribution

After the analysis of the dice and coin simulation data, the investigation moved to focus on the numeric sequence data provided by the pseudorandom generators. The first test used on this data was the Kolmogorov-Smirnov test which focused on the distribution of data between the minimum and maximum potential values. To pass this test a generator must show an empirical distribution of all data provided. This is given by a 1-sample test value which must not score above the critical value of 0.501. Datasets containing multiple implementations also produced a 2-sample test value which must not score above the value of α (0.05) to be classified as empirically distributed. The main caveat of this test was that it was designed for data between 0 and 1 so in order to adjust the outputs provided, division by 100 was used to ensure data normally in integer form between 0 and 100 was in the correct format.

```
#Retrieving the C# rand data from the C# output
setwd("D:/Github/PROJ518/C#_Rand_Function/RandFunctionOutput")
CRand1Values <- fromJSON(file = "RandVer1.json")
CRand2Values <- fromJSON(file = "RandVer2.json")
CRand3Values <- fromJSON(file = "RandVer3.json")
print(CRand1Values)
print(CRand2Values)
print(CRand3Values)

#1 sample tests
ks.test(CRand1Values/100, "pnorm")
ks.test(CRand2Values/100, "pnorm")
ks.test(CRand3Values/100, "pnorm")

#2 sample tests
ks.test(CRand1Values/100, CRand2Values/100)
ks.test(CRand1Values/100, CRand3Values/100)
ks.test(CRand2Values/100, CRand3Values/100)
```

Figure 51. A screenshot of the KS function using C# rand data

Figure 51 shows the implementation of the Kolmogorov-Smirnov test with the C# data. The results of the testing are shown in figure 50. To visualise the empirical distribution of the data sets, the *ecdf* function provided by R was also used to calculate plottable empirical distribution data points.

Data Sources	1 Sample Test Statistic	2 Sample Test Statistic 1	2 Sample Test Statistic 2
C# Unseeded Rand	0.5	0.046	0.592
C# Seeded Rand	0.5	0.592	0.046
C# Cryptographic Rand	0.51842	0.592	0.592
Lehmer Int Version 1	0.15921	0.42	N/A
Lehmer Int Version 2	0.3444	0.42	N/A
Lehmer Real Version 1	0.50074	0.042	N/A
Lehmer Real Version 2	0.50002	0.042	N/A
Python Randint	0.5	0.048	0.042
Python Unseeded Random	0.50118	0.048	N/A
Python Seeded Random	0.50215	0.048	N/A
NumPy Unseeded Randint	0.5	0.038	N/A
NumPy Seeded Randint	0.5	0.048	0.038
JavaScript Rand	0.5	N/A	N/A
Middle Square Method	0.51594	N/A	N/A
Random.org Data	0.50399	N/A	N/A
Park White Noise Data	0.47854	0.338	0.192
Sea White Noise Data	0.49299	0.192	0.456
Roundabout White Noise Data	0.57062	0.338	0.456

Figure 52. A Table of Results for the Kolmogorov-Smirnov Test

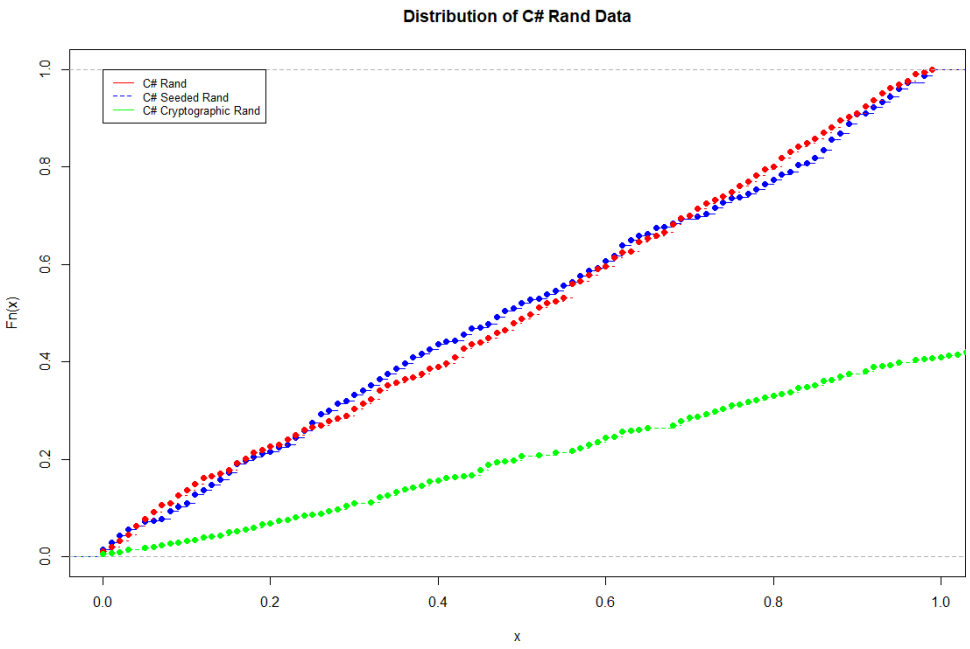


Figure 53. A Scatter Graph Showing the Distribution of C# Rand Data

Both the unseeded and seeded C# implementations of rand were shown to have an empirical distribution of results while the cryptographic implementation failed both 1-sample and 2-sample KS testing. Figure 53 shows the empirical distribution of all three implementations. While the unseeded and seeded implementations follow an expected upwards trend from 0 to 1, the cryptographic implementation, while still following an upwards trend, shows far less distributions between 0.2 and 1.0 on the Y axis.

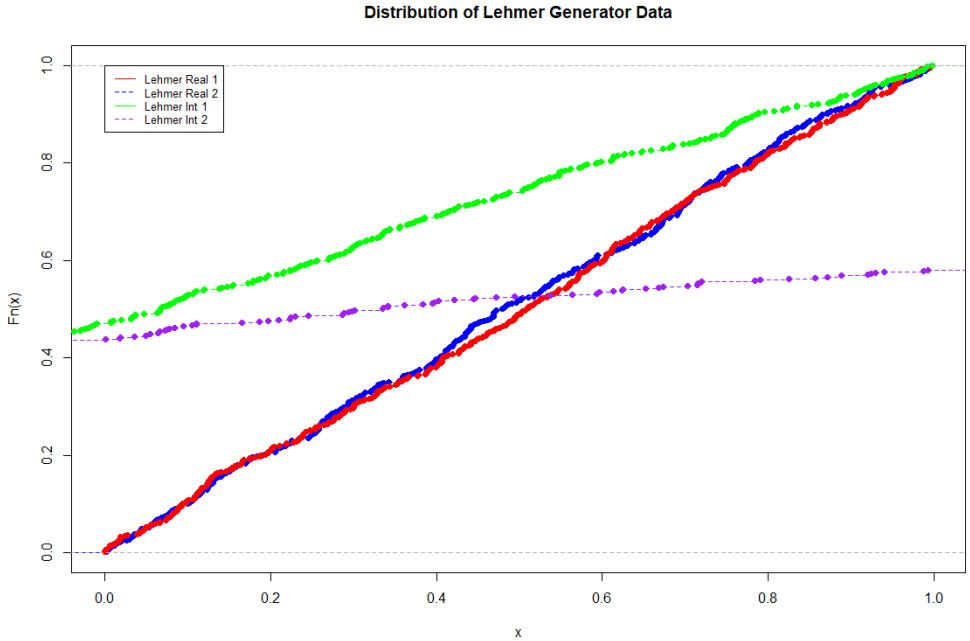


Figure 54. A Scatter Graph Showing the Distribution of Lehmer Generator Data

By far the most successful implementations of the Lehmer Generator were versions 1 and 2 of the Real based generators which passed both the 1-sample and 2-sample KS tests. Although able to pass the 1-sample tests, version 1 and 2 of the Integer based generators failed the 2-sample tests.

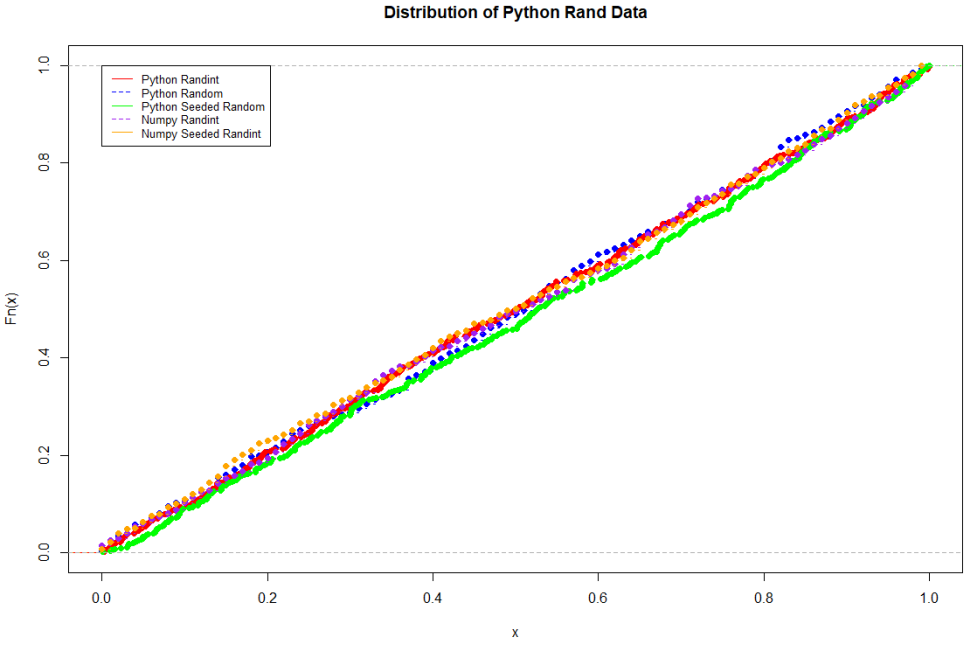


Figure 55. A Scatter Graph Showing the Distribution of Python Rand Data

The integer sequence Randint generators within Python and NumPy performed better than the real/decimal sequence Random generators, with all three of the Randint functions passing the 1-sample and 2-sample KS tests. As shown in figure 55 all five of the Python implementations showed an empirical distribution.

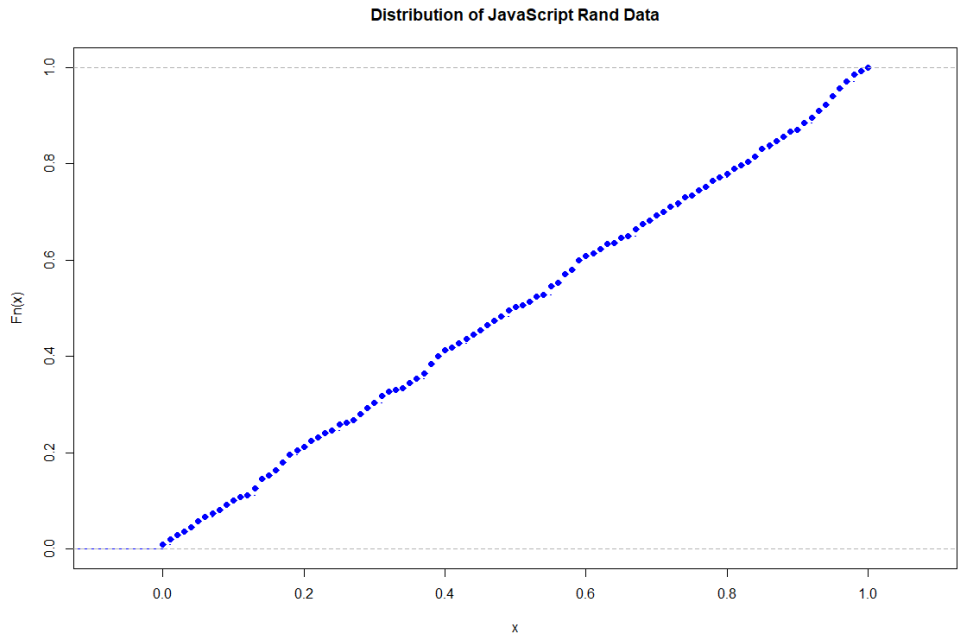


Figure 56. A Scatter Graph Showing the Distribution of JavaScript Rand Data

The JavaScript dataset passed the 1-sample KS test. Figure 56 shows the distribution of the JavaScript data which followed the expected trend.

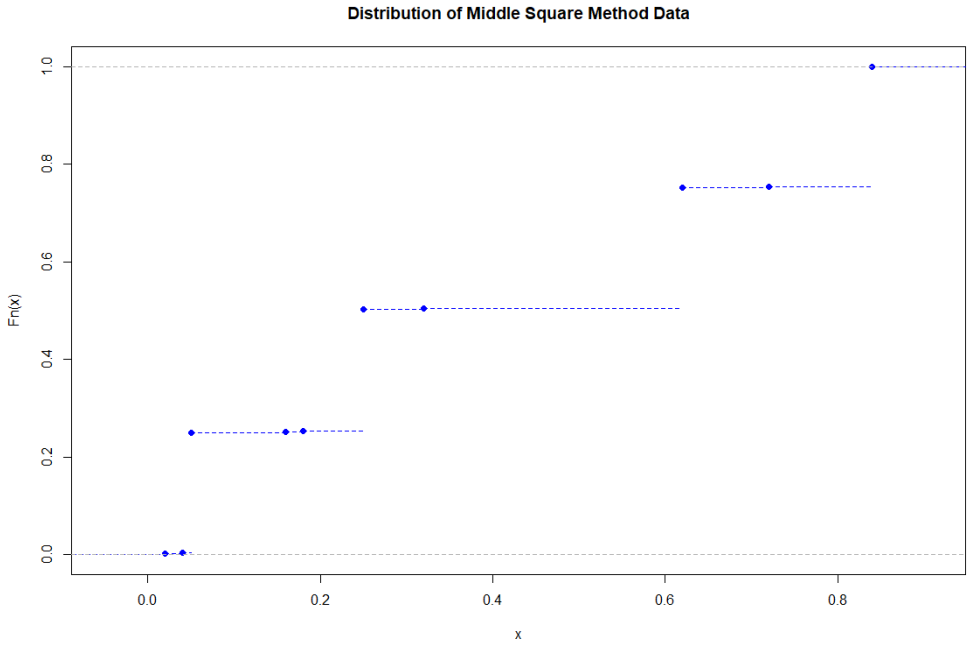


Figure 57. A Scatter Graph Showing the Distribution of Middle Square Method Data

As expected, the Middle Square dataset did not pass the 1-sample KS test. The distribution shown in figure 57 also confirms the unsuitable nature of the Middle Square method as a pseudorandom generator, in which the methods repeating cycle of values are seen as a series of unconnected lines rather than a trend of data points.

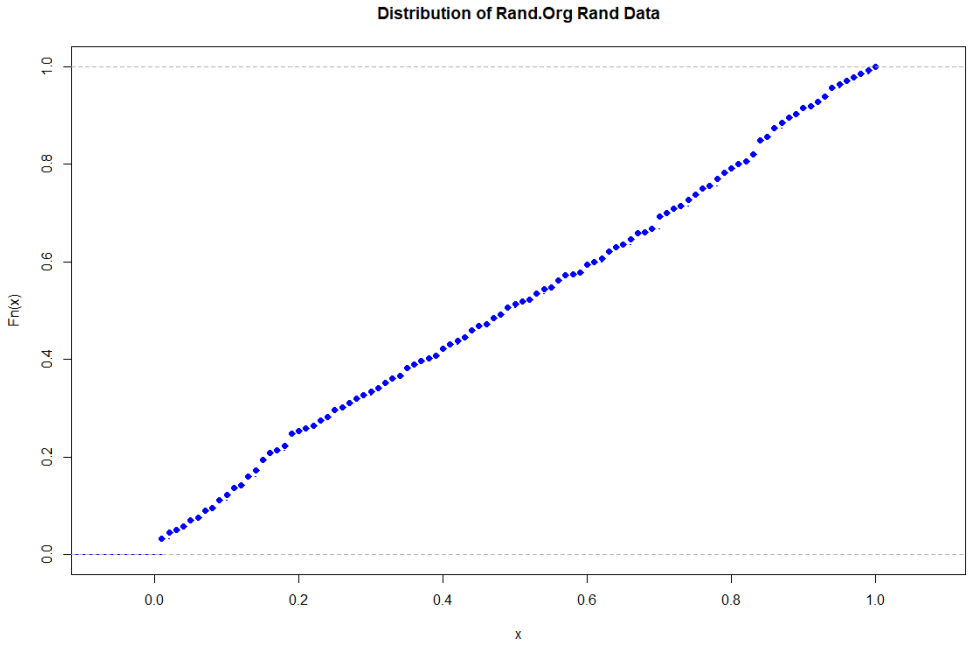


Figure 58. A Scatter Graph Showing the Distribution of Random.org Data

The Random.org dataset failed the 1-sample KS test, with a test statistic just 0.0299 away from the critical value. When visualised in figure 58, the trend shown by the data mostly corresponds with what is expected however features several breaks throughout the distribution. Overall, the data is visibly similar to that seen in figures 56 or 53.

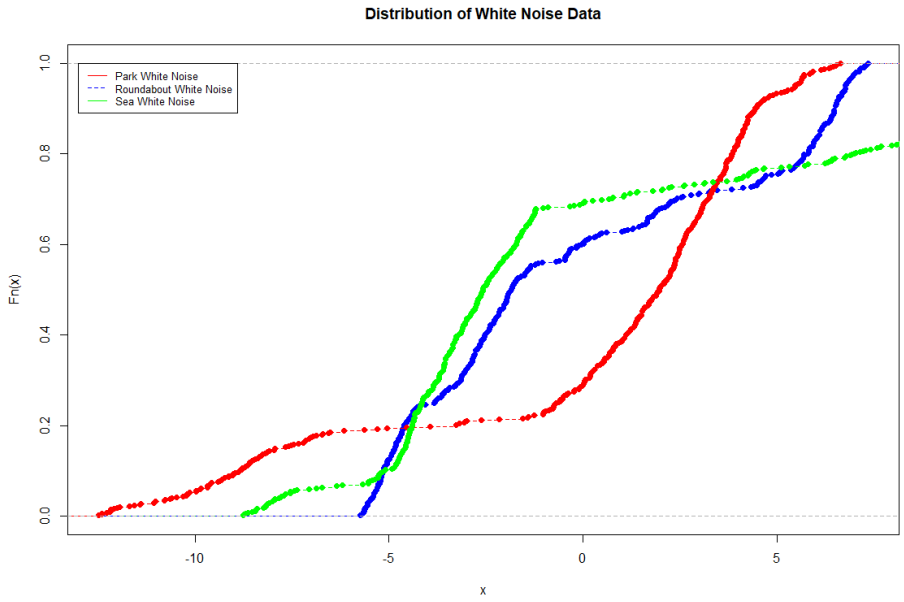


Figure 59. A Scatter Graph Showing the Distribution of White Noise Data

The Park and Sea white noise datasets passed both the 1-sample and 2-sample KS tests while the Roundabout dataset could only pass the 2-sample tests. When viewing the distribution of the white noise data in figure 59 all three datasets show unique trends unseen with any of the other generators.

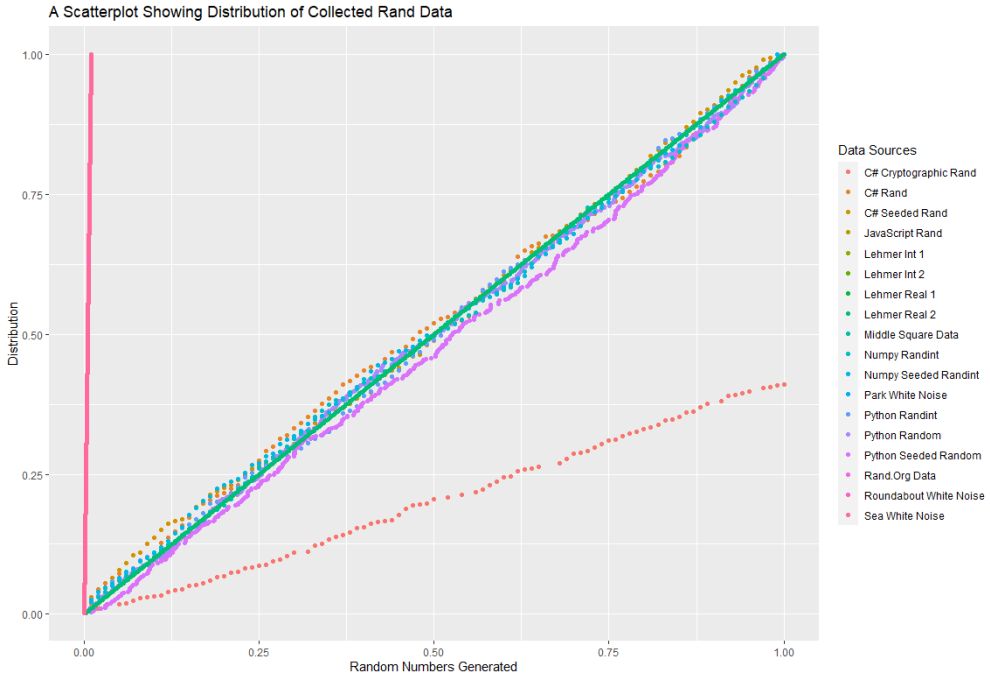


Figure 60. A Scatter Graph Showing the Distribution of all Collected Rand Data

When all the data is graphed together, the most noticeable outliers are the white noise data and the C# cryptographic data. The remaining generators all mostly conform to an empirical distribution and this is reflected both in figure 60 and the results within figure 52.

4.4 Serial Test of Empirical Distribution

While the Kolmogorov-Smirnov test focused on the distribution of whole datasets, the Serial test instead compares the empirical distribution of sets within the data. This test provides two output statistics: a test statistic, and a P-value. While the test statistic can be used for comparison between generators, where a lower value indicates a dataset where the observed number of sets more closely match the expected number of sets, the P-value or Probability value represents the likelihood that the data provided was able to achieve its test statistic. Much like with the Chi-Squared test an ideal generator will provide a P-value that is neither too large nor too small, with the best performing generators being the ones able to score the value closest to 0.5.

Data Sources	Test Statistic	P-Value
C# Unseeded Rand	1.2	0.76
C# Seeded Rand	1.5	0.69
C# Cryptographic Rand	139	7.1e-30
Python Randint	5.4	0.14
Python Unseeded Rand	3.9	0.27
Python Seeded Rand	9	0.03
NumPy Unseeded Randint	2.8	0.43
NumPy Seeded Randint	1.4	0.7
JavaScript Rand	5	0.17
Random.org Data	3.4	0.33
Lehmer Int Version 1	87	0.1e-19
Lehmer Int Version 2	223	4.9e-48
Lehmer Real Version 1	3.9	0.28
Lehmer Real Version 2	0.34	0.95
Middle Square Data	726	4.2e-157
Park White Noise Data	250	6.5e-54
Roundabout White Noise Data	230	1.1e-49
Sea White Noise Data	237	5.2e-51

Figure 61. A Table of Results for the Serial Test

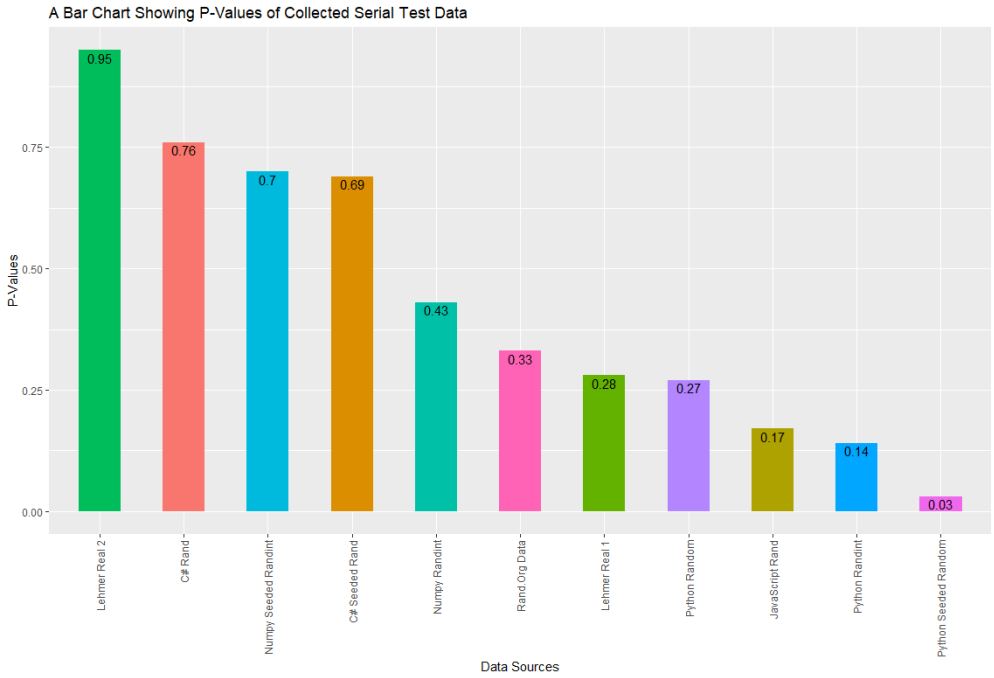


Figure 62. A Bar Chart Showing P-Values of Collected Serial Test Data

The unseeded and seeded Rand data from C# performed well in the Serial test, scoring close to the optimal P-value and being among the lowest test statistics scored. However, the same couldn't be said about the cryptographic data which failed to pass the test with both its test statistic and P-value.

All Python datasets passed the Serial test, with the NumPy unseeded implementation achieving the best P-value of any dataset and the NumPy seeded implementation scoring one of the lowest test statistics. An obvious outlier within the Python data was the Seeded Rand implementation which had a high test statistic and the lowest non-anomalous P-value.

The JavaScript dataset passed the Serial test, however performed poorer than other implementations in regard to both test statistic and P-value. The values given for the JavaScript generator were close to those of the Python Randint implementation, with JavaScript scoring a lower test statistic and a P-value only 0.03 higher.

The Random.org data also passed, with the second-best P-value recorded. This was not mirrored in the test statistic however, which while still low was beaten by the NumPy implementations and the C# implementations.

As with the KS tests previously, the integer versions of the Lehmer Generator failed to pass the Serial test, producing not only incredibly high test statistics but also anomalous P-values so small the sequences produced had to be considered too unlikely to be valid. In contrast, the real versions of the Lehmer Generator passed the Serial test, although version 2 did produce the highest P-value of any implementation, making it too likely to be considered valid. Version 1 performed better, however, with the fourth best P-value recorded.

The Middle Square data, as expected, failed the Serial test with the worst test statistic and P-value recorded for any dataset. Due to this being a test of distributed sets within the data, the hundreds of repeated pairs were penalised heavily.

The white noise datasets performed poorly in the Serial test, with all three failing due to both their test statistics and P-values. All three datasets came close to matching the performance seen by the integer version 2 Lehmer Generator and despite failing were still considerably more effective than the Middle Square method.

4.5 Gap Test of Interval Recurrence

The Gap test focusses on measuring the ‘gaps’ between recurring values within the datasets. This test was chosen as much like empirical distribution, the space between recurring values provides an insight into the ‘randomness’ of a sequence. Much like the Serial test it produces two outputs: a test statistic and a P-value. An effective generator was one that would provide both a low test statistic and a P-value as close as possible to 0.5. All the datasets featured in the Serial test were used in this test with the results for each generator featured in a table of results seen in figure 63.

Data Sources	Test Statistic	P-Value
C# Unseeded Rand	11	0.3
C# Seeded Rand	14	0.12
C# Cryptographic Rand	32	0.00023
Python Randint	5.3	0.8
Python Unseeded Rand	7.7	0.56
Python Seeded Rand	6.1	0.73
NumPy Unseeded Randint	7.3	0.6
NumPy Seeded Randint	12	0.22
JavaScript Rand	6.7	0.67
Random.org Data	17	0.042
Lehmer Int Version 1	27	0.0012
Lehmer Int Version 2	79	2.6e-13
Lehmer Real Version 1	18	0.036
Lehmer Real Version 2	4.8	0.85
Middle Square Data	1.6e+147	0
Park White Noise Data	120	1.3e-21
Roundabout White Noise Data	109	2e-19
Sea White Noise Data	121	8.5e-22

Figure 63. A Table of results for the Gap Test

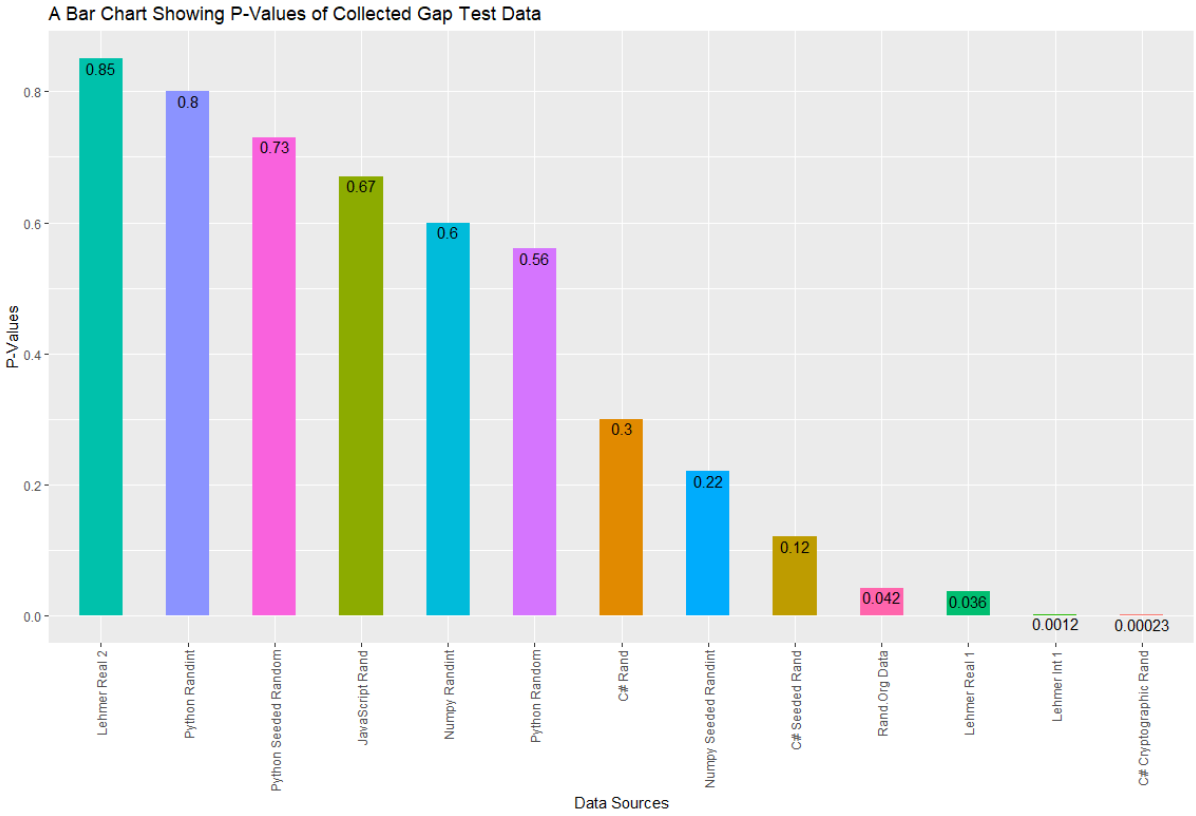


Figure 64. A Bar Chart Showing P-Values of Collected Gap Test Data

The C# datasets performed worse in the Gap test than in the previous Serial test, with only the unseeded rand implementation scoring highly, coming in as the fourth best P-value. The cryptographic rand implementation was again the lowest performing C# dataset.

Being only 0.06 away from the optimal P-value, the Python unseeded random dataset was the top scoring implementation seen in figure 64, with the NumPy unseeded dataset again placing highly as the second-best implementation. All other Python datasets passed the Gap test, with the NumPy seeded implementation having the lowest Python performance.

JavaScript placed well in the Gap test and performed better than in the previous Serial test, being the third-best implementation featured by P-value. The JavaScript test statistic also showed improvement, with the generator outperforming the C# implementations and many of the Python implementations.

The Random.org dataset had a surprisingly poor performance in the Gap test, especially considering its high performance in the Serial test. The dataset's test statistic placed behind the C#, Python and JavaScript implementations and had the fourth lowest non-anomalous P-value collected.

Again, the real versions of the Lehmer Generator performed noticeably better than the integer versions. While version 1 of the real generator had the third lowest P-value recorded and had a test statistic only 1 higher than the Random.org data, version 2 of the real generator had the largest P-value and the smallest test statistic.

Integer version 1 of the generator passed the Gap test and scored the second lowest P-Value although as previously version 2 failed to pass the test.

The Middle Square data did not pass the Gap test with either its test statistic or its P-value. This was as expected due to the unavoidable loop the seed values form after a certain number of iterations.

The White Noise data performed poorly again in the Gap test, with all three datasets failing in regard to their test statistics and P-values. As seen previously, the Roundabout dataset was the best performing out of the three while in this test the Sea dataset performed the worst.

4.6 Poker Test of Card Shuffle Simulation Data

Unlike the dice roll or coin flip simulation data, the card shuffle data wasn't made up of simply numeric values that could be tested using the standard empirical tests shown above. Instead, the specialised Poker test was used. Much like the Kolmogorov-Smirnov test, the Poker test is designed to evaluate the distribution of values. This is particularly prudent for card shuffle simulations as their effectiveness in digital card games revolves around their ability to fairly shuffle a deck based not only on the values of the cards but the suits as well. To prove whether these datasets could be considered effective, the Poker test analysed the distribution of suits in each deck, the frequency of possible poker hands achievable with a sequential distribution of each deck, the frequency of possible poker hands achievable with a poker style distribution of each deck, and the frequency of possible poker hands achievable with a 'Texas Hold 'Em' style distribution of each deck. As with other randomness tests used, the most effective implementations were the ones able to consistently achieve a believable 'random' distribution of values.

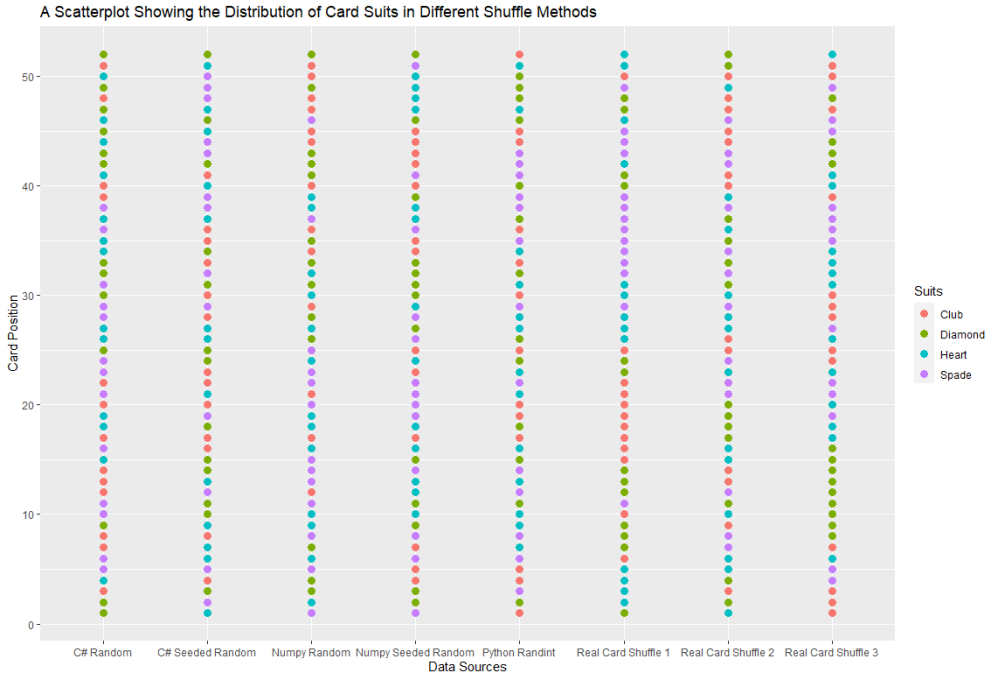


Figure 65. A Scatterplot Showing the Distribution of Card Suits in Different Shuffle Methods

Before the data was analysed using any distribution style the cards in each deck were categorised into four suits. A scatterplot such as the one in figure 65 could then be used to display the distribution of suits in each shuffled deck.

Both C# implementations showed an acceptable distribution of suits, often with pairs and the occasional triple of matching suits grouped together. Unlike in other implementations there were no large groups of one suit. The Python implementations showed similar patterns although the NumPy seeded deck contained multiple sets of four matching suits in a row. This systematic distribution of the suits, with often no more than two or three cards together was the most noticeable feature of the pseudorandom generators. The physically shuffled decks showed immediate diversity compared to their digital counterparts. Real Card Shuffle 1 and 3 both featured large groups of matching suits. The distribution in Real Card Shuffle 2, which used a standard card dealers shuffle technique, showed a distribution more similar to the pseudorandom implementations than the other physical implementations, with groups only occasionally exceeding two or three matching cards.

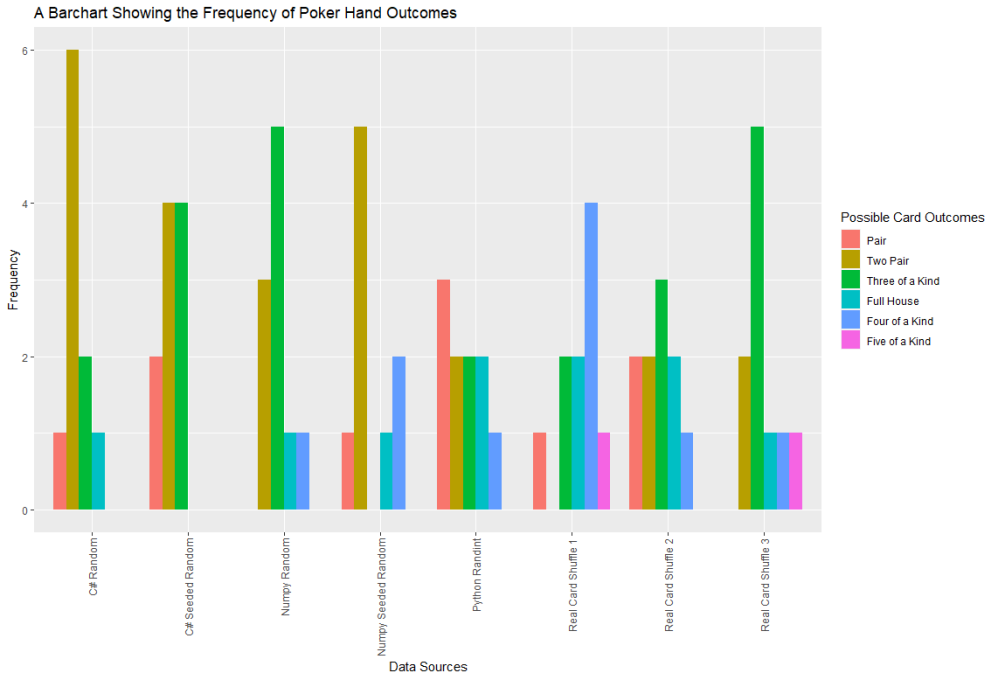


Figure 66. A Bar Chart Showing the Frequency of Poker Hand Outcomes from Sequential Card Draws

The next test performed on the data was to calculate the frequency of possible poker hands each shuffled deck is capable of producing when drawn through sequentially. This was done with a custom-made function in R, which would take a deck as an input, then using a for loop iterate through the deck in rounds of 5. The possible outcomes of 5 cards considering only suit are: Pair, Two Pair, Three of a Kind, Full House, Four of a Kind, or Five of a Kind. Figure 66 shows the recorded frequencies for each data set.

The most frequent outcome for the C# implementations was Two Pair which occurred 50% of the time across both datasets. Three of a kind also occurred

frequently, particularly in the seeded random implementation. Across both implementations Full House only occurred once and no Four or Five of a Kinds were recorded. Without larger concentrations of suits in groups of 4 or more the likelihood of sequentially drawing a hand consisting of four or five cards of the same suit was unlikely.

The Python Randint implementation showed the most diversity out of all the Python datasets analysed, with at least 1 occurrence of each possible outcome except Five of a Kind. The NumPy implementations gave results similar to those seen with the C# implementations.

The physical datasets gave far more diverse outcomes compared to the C# implementations, with the large groups of suits found in Real Card Shuffle 1 and Real Card Shuffle 3 allowing for hands of Four and Five of a Kind. Three of a Kind hands remained the most seen outcome in the majority of datasets.

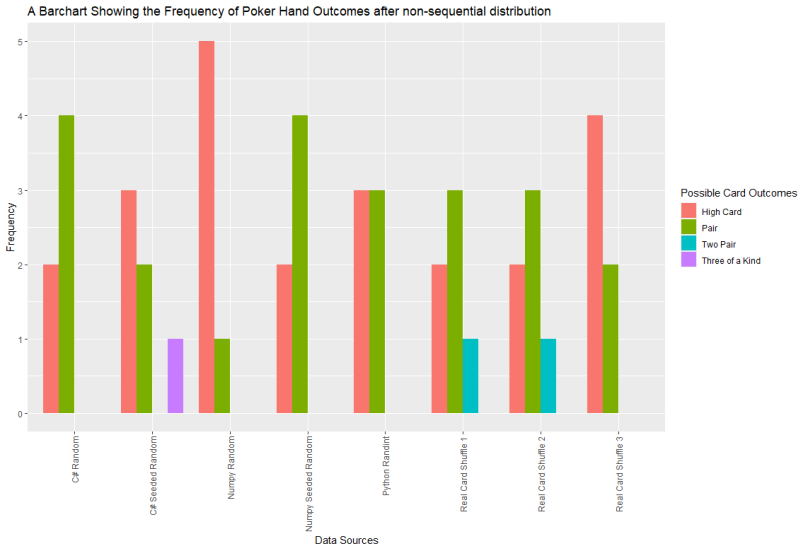


Figure 67. A Bar Chart Showing the Frequency of Poker Hand Outcomes after Non-Sequential Distribution

Moving beyond purely suits for determining outcomes, the next test used the non-sequential method for dealing hands seen in a game of poker. For this simulation, the deck was dealt between six ‘players’ sitting around a table in a clockwise direction until each player had a hand of five cards. As before all dealing was done through a custom-made function. The incorporation of the multiple hands also meant that the wins and losses of each player could also be tallied, to ensure that no position at the table had an advantage over the others.

Player	Wins	Losses	Draws
Player 1	1	5	2
Player 2	2	4	2
Player 3	0	4	4
Player 4	1	3	4
Player 5	2	5	1
Player 6	1	5	2

Figure 68. A Table of Results for Player Wins/Loses in Poker

Figure 68 shows that across the eight rounds played of Poker, player 2 and player 5 had the highest number of wins, while player 3 had the fewest, achieving a total of 0 wins and 4 draws. The results given show no clear indication that any position at the table was preferable to any of the others, especially given if play had continued.

A clear difference between the C# data shown in figure 66 and 67 is the lack of hands with combinations above a Pair. Between both implementations only a single Three of a Kind hand was achieved. This could be explained by both the distribution of cards between players, in which the previously seen groups of two or three cards from the same suit have been divided further and often leave players with few cards in the hand that can match.

This trend repeated in the Python datasets, with none of the implementations able to achieve hands greater than a Pair.

The diversity seen in the physical datasets also suffered because of the new distribution method. Only Real Card Shuffle 1 and 2 were able to achieve a hand above a Pair while for Real Card Shuffle 3 the most commonly seen hand was a High Card.

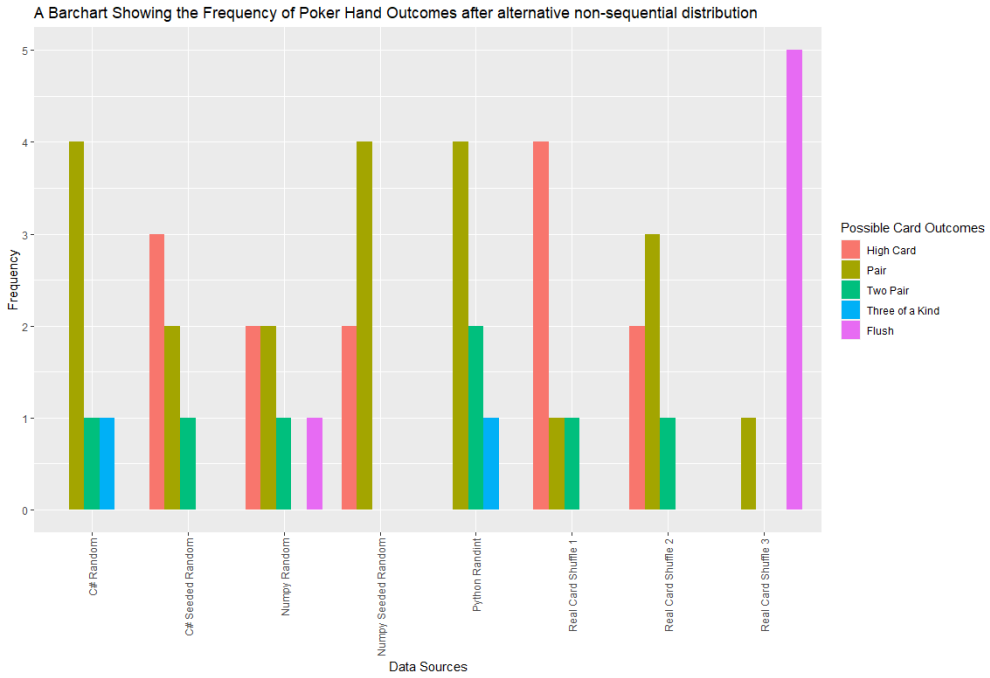


Figure 69. A Bar Chart Showing the Frequency of Poker Hand Outcomes after Alternative Non-Sequential Distribution

The final test performed on the data was similar to the test shown above, however instead the cards were distributed using the ‘Texas Hold ‘Em’ poker ruleset. In this version of the game, each player is dealt cards one at a time clockwise as previously, but each player only receives two cards. Once the hands are distributed, five cards are placed on the ‘river’. Texas Hold ‘Em poker sees high levels of play digitally and is the more popular ruleset used which made this test valuable not only since it provided an alternative form of non-sequential distribution but also as it allowed for evaluation on the effectiveness of these generators in another real game

scenario. As with the previous test, the wins/losses of each player were recorded to ensure that no position at the table gave an unfair advantage.

Player	Wins	Losses	Draws
Player 1	2	5	1
Player 2	0	6	2
Player 3	0	7	1
Player 4	1	5	2
Player 5	3	4	1
Player 6	0	6	2

Figure 70. A Table of Results for Player Wins/Loses in Texas Hold 'Em Poker

Figure 70 shows that across the eight rounds played, player 5 had the highest number of wins while players 2,3 and 6 were unable to win with any of the hands available to them. It is difficult to say whether the poorer performance of player 3 and the better performance of player 5 was due only because of their position at the table or whether this was caused by the selection of cards offered to the players. Because of the results given in figure 68, it cannot be stated with certainty that the position of the players at the table gave an inherent advantage.

The C# implementations seen in figure 69 feature a more diverse set of outcomes than those seen in figure 67. This increase in diversity for outcomes was most likely caused by the new distribution method providing not only two additional cards to form hands with but also by the addition of the river which draws cards sequentially from the deck instead of being dealt among all the players at the table.

The NumPy unseeded implementation and the Python random implementation also followed this trend, with the appearance of Two Pair, Three of a Kind, and Flush hands, although the same couldn't be said about the seeded NumPy dataset which still contained only High Card and Pair outcomes.

The physical datasets Real Card Shuffle 1 and Real Card Shuffle 2 remained mostly unchanged compared to their poker results, although Shuffle 1 showed a far higher frequency of High Card hands. Real Card Shuffle 3 presented what at first appeared to be a purely anomalous set of results, with five out of the six hands being Flushes. While this is incredibly unlikely, this can be explained by examining the distributions shown in figure 65. The larger groups of matching suits, especially the group of Diamonds near the beginning of the deck, meant that any player dealt at least one Diamond was able to achieve a Flush using the cards on the river.

4.7 Runs Test

Moving back to numeric sequence testing, the Runs test examines a dataset for 'runs' or sequentially increasing/decreasing sets of values. Two types of runs can be found in data, 'runs up' where the sequential values in the sequence move in ascending order and 'runs down' where the values move in descending order. The test divides the dataset into sections of runs and then produces a P-value representing the likelihood that such a ratio of ascending/descending sequences could occur. A generator passes the Runs test if its P-value is greater than the significance value of α (0.05).

Data Source	P-Value
C# Unseeded Rand	0.08693
C# Seeded Rand	0.2243
C# Cryptographic Rand	0.3229
Python Randint	0.713
Python Unseeded Rand	0.6544
Python Seeded Rand	0.5308
NumPy Unseeded Randint	0.9997
NumPy Seeded Randint	0.3706
JavaScript Rand	0.3018
Random.org Data	0.05256
Lehmer Int Version 1	0.7882
Lehmer Int Version 2	0.4887
Lehmer Real Version 1	0.7202
Lehmer Real Version 2	0.6544
Middle Square Data	2.2e-16
Park White Noise Data	2.2e-16
Roundabout White Noise Data	2.2e-16
Sea White Noise Data	2.2e-16

Figure 71. A Table of Results for the Runs Test

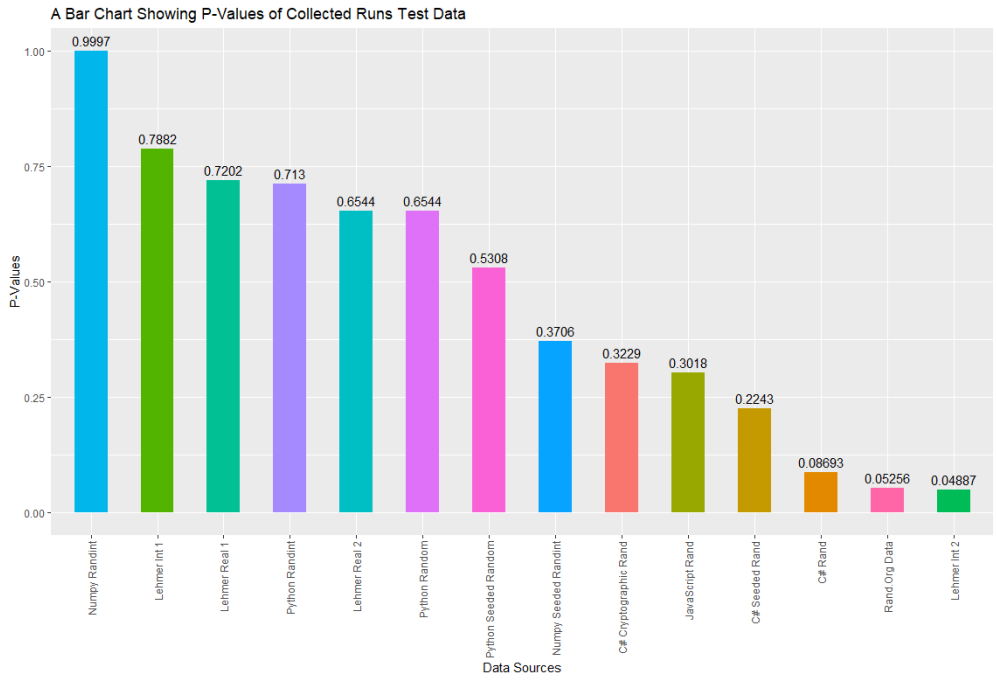


Figure 72. A Bar Chart Showing P-Values of Collected Runs Test Data

All three C# implementations passed the Runs test. The cryptographic dataset performed far better in this test than previously, achieving a much more central P-value while the Unseeded Rand implementation had the weakest performance out of the C# datasets, with the third lowest non-anomalous value.

The Python and NumPy datasets all passed the Runs test, with the Seeded Rand implementation achieving the most central P-value recorded. Other datasets also performed well, particularly the Python Randint and Unseeded Rand however the

NumPy Unseeded Randint had the highest P-value of any dataset. Much like the C# implementations, this was an unexpected outcome as in previous tests there was never a large disparity between seeded and unseeded implementations. One explanation for this was the impact of a system generated seed value on the sequences.

The JavaScript implementation passed the Runs test, performing better than the C# Rand implementations and some of the above-mentioned Python implementations.

The Random.org dataset also passed the Runs test, with a P-value only 0.00256 above the significance value. This score placed it as the second lowest non-anomalous P-value recorded.

Except for the Int Version 2 implementation, all versions of the Lehmer Generator passed the Runs test. As expected, the Real versions of the generator were the best performing implementations with the Real Version 2 dataset achieving the same P-value as the Python Unseeded Rand.

The Middle Square and White Noise datasets all failed the Runs test with matching P-values. This was not unexpected as the data provided in these datasets either follow too rigid a pattern, as with the Middle Square method, or are too irregular to form runs of the expected length, as with the White Noise implementations.

4.8 The Serial Correlation Test of Dependency

The Serial Correlation test is designed to evaluate the serial coefficient of a data set, which is a measure of dependency using the Yule-Walker estimation method. Two values are given by the Serial Correlation test, a test statistic, and a P-value. An effective generator was one able to provide a test statistic closest to 0.

Data Sources	Test Statistic	P-Value
C# Unseeded Rand	1.6	0.11
C# Seeded Rand	-0.84	0.4
C# Cryptographic Rand	-0.96	0.34
Python Randint	0.68	0.5
Python Unseeded Rand	-0.082	0.93
Python Seeded Rand	-0.43	0.67
NumPy Unseeded Randint	1.2	0.24
NumPy Seeded Randint	1.8	0.079
JavaScript Rand	-0.7	0.49
Random.org Data	-2.9	0.0039
Lehmer Int Version 1	0.6	0.55
Lehmer Int Version 2	-0.039	0.97
Lehmer Real Version 1	-0.14	0.89
Lehmer Real Version 2	0.29	0.78
Middle Square Data	2.2	0.025
Park White Noise Data	-22	0
Roundabout White Noise Data	-22	0
Sea White Noise Data	-22	0

Figure 73. A Table of Results for the Serial Correlation Test

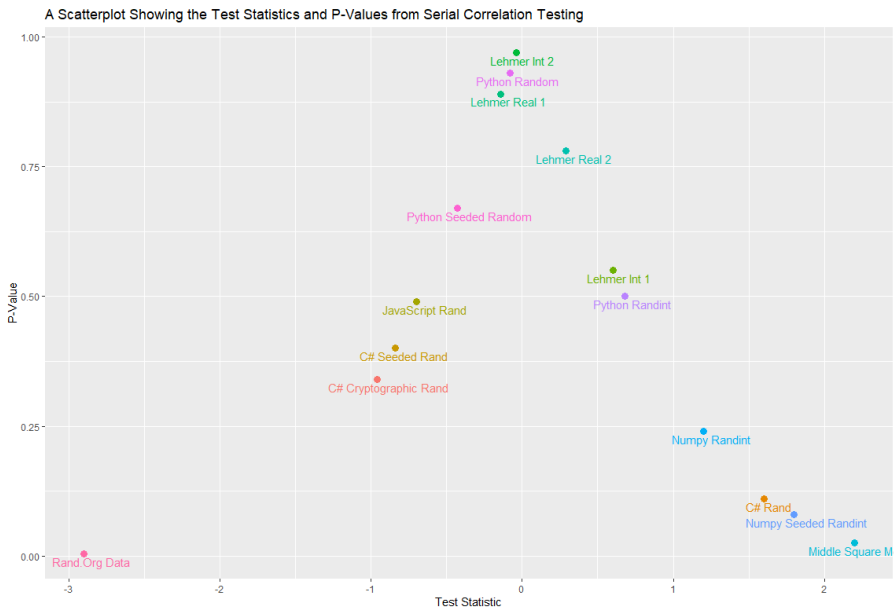


Figure 74. A Scatterplot Showing the Test Statistics and P-Values from Serial Correlation Testing

When viewing the C# data within figure 74, the plot shows that the Seeded and Cryptographic implementations performed better than the Unseeded implementation. Both scored a test statistic close to 0 while also having a reasonable P-value.

The Python datasets performed well in the Serial Correlation test, with the Unseeded, Seeded and Randint implementations producing low test statistics. The NumPy datasets performed significantly worse than their Python counterparts, with high test statistics and low P-values.

The JavaScript dataset performed well in the Serial Correlation test, producing a test statistic closer to 0 than C# or NumPy. JavaScript also scored relatively highly with its P-value.

The Random.org dataset performed far worse than expected, with the lowest test statistic and P-value. As mentioned previously, Random.org collects data using atmospheric noise which may lead to the data collected being considered too independent thus causing the test to negatively grade the dataset.

Unsurprisingly, the Lehmer Generator implementations were among the best performing datasets in figure 74. This was expected with the Lehmer algorithm, which is built with a dependency on the modified seed values used for output generation.

When viewing the Middle Square method test statistic, it is clear why the method performed so poorly. By using an algorithm designed around the previous output value, dependency between the data will naturally be high.

While it would be expected for the White Noise datasets, which provide data not sampled from any algorithm allowing for completely independent data, to score well in the Serial Correlation test but much like the Random.org data the datasets were too independent to be considered viable.

4.9 Birthday Spacings Test

One of the classic tests of randomness, the Birthday Spacings test evaluates the extent to which a dataset conforms to a normal distribution. This is done by choosing *m* birthdays from a year of *n* length then comparing the expected frequency of repeated spacing values to the observed frequency found in each dataset. Two results were produced by the Birthday Spacings test, an Anderson-Darling Distribution test statistic and Anderson-Darling result of 1 or 0. A result of 1 indicates that a dataset conforms to a normal distribution while a result of 0 indicates a lack of normal distribution.

Data Source	Test Statistic	Result
C# Unseeded Rand	15.172	0
C# Seeded Rand	15.013	0
C# Cryptographic Rand	16.221	0
Python Randint	14.739	0
Python Unseeded Rand	2	1
Python Seeded Rand	2	1
NumPy Unseeded Randint	15.504	0
NumPy Seeded Randint	15.221	0
JavaScript Rand	14.583	0
Random.org Data	14.837	0
Lehmer Int Version 1	2	1
Lehmer Int Version 2	2	1
Lehmer Real Version 1	2	1
Lehmer Real Version 2	2	1
Middle Square Data	12.816	0
Park White Noise Data	16.813	0
Roundabout White Noise Data	16.73	0
Sea White Noise Data	16.858	0

Figure 75. A Table of Results for the Birthday Spacings Test

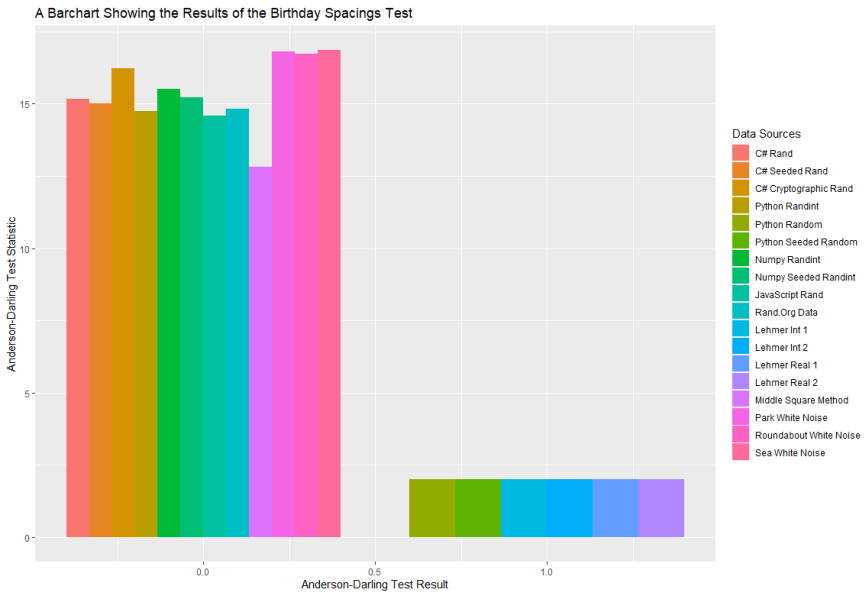


Figure 76. A Bar Chart Showing the Results of the Birthday Spacings Test

All three C# implementations passed the Birthday Spacings test and were among the highest test statistics achieved. The cryptographic implementation performed best out of the three datasets.

Only the Python Rand implementations failed the Birthday Spacings test, with the other generators including the NumPy implementations producing relatively high test statistics.

The JavaScript and Random.org datasets also both passed the Birthday Spacings test. These two implementations were some of the lowest scoring pass results, implying that while both contained non-normal distribution of values, they were more uniformly distributed than generators like C# or Python.

None of the Lehmer Generator implementations passed the Birthday Spacings test, which was highly unexpected. As with the failed Python implementations the most likely cause of this was that the particular pattern the generator followed when producing values was picked up on by the test and punished where in other cases such an algorithm would have shined. Many of the results shown in the Birthday Spacings test mirror those seen in the Serial Correlation test, with many of the highest-ranking generators seen previously such as the Lehmer and Python Rand implementations failing to pass.

The Middle Square dataset passed the Birthday Spacings test, while achieving the lowest test statistic of any passed generator. This result was expected because the structure of the Middle Square data is so often presented as anomalous in other tests that there was little chance it would be considered normally distributed in this one.

All three White Noise datasets passed the Birthday Spacings test and had the largest test statistics recorded. Much like the Middle Square dataset, these results weren't unexpected as there was little chance the data recorded would match a normal distribution.

Discussion

After collecting and analysing the data provided by the real and pseudorandom generators sampled, the final step was to evaluate each implementation's overall performance. A key point that was considered during this discussion was that when reviewing all the test data gathered and attempting to compare suitable generators the datasets able to not necessarily outperform but remain constantly effective across many tests should be considered the more effective solutions.

The C# implementations were broken down into two main categories: the Rand functions and the Cryptographic function. The Rand implementations, both seeded and unseeded, performed well in the dice, coin, and card simulations which showed that the generators effectiveness remained relatively unaffected by input and output limitations. During Chi-Squared testing, all the Rand implementation datasets ranked between the desired distribution ranges, with the dice simulation data performing just slightly better than the coin simulation data. When producing shuffled deck data, the distribution of suits for both implementations was acceptable, with most groups being

between two or three cards large. The distribution of values, seen for instance in the Kolmogorov-Smirnov test, with the numeric sequence Rand datasets followed the expected trend of pseudorandom data and while never being the top performers in any of the tests conducted, both the seeded and unseeded Rand implementations consistently passed. Unfortunately, the same couldn't be said about the Cryptographic implementation which performed poorly in many of the numeric sequence tests conducted. Despite this however, it is still believed that the function can produce a valid pseudorandom sequence and the main issue presented by the tests was that they weren't designed for the cryptographically secure sequences being produced. The purpose of such a function is to offer the user a non-reproducible collection of bits that can be used for encryption, not a mass-produced collection of values simulating randomness.

Python provided two different options for pseudorandom generation, default Rand and NumPy. Both performed well in the simulation tests, achieving sufficiently random results and a sufficiently random distribution of shuffled cards. In the poker and Texas Hold 'Em distribution tests, both the Python and NumPy implementations provided a likely variety of hands, featuring Three and Four of a Kind. In the numeric sequence tests both implementations performed very well, with all showing an expected trend in the Kolmogorov-Smirnov test and achieving the closest P-values to the optimal 0.5 in the Serial, Gap, Runs, and Serial Correlation tests. This is counterbalanced however by a poor performance by the Rand functions in the Birthday Spacings test and low test statistics for NumPy in the Serial Correlation test. Seeding also played a more significant role in the Python implementations than in the C# implementations, often with a large difference between values depending on the seed provided to the algorithms.

Testing began poorly for JavaScript, with anomalous frequencies of dice outcomes and an unsuitable Chi-Squared statistic. However, this method improved in further tests with far more expected values in the coin simulation. The trend seen in the Kolmogorov-Smirnov test was as expected. Although the dataset featured no standout performances, and the generator lacks any ability to alter the seed used, almost all JavaScript results were above average. In tests such as the Serial Correlation and Gap, the dataset was among the top performing implementations. While the results of testing indicated that limitations on output can notably affect performance and the generator performs less effectively than Python, JavaScript can still be considered a valid pseudorandom number generator for most commercial projects, although its use in a scientific environment wouldn't be recommended.

Considering the claim that Random.org used atmospheric noise to produce true random number sequences, expectations regarding its performance were high. This dataset, perhaps even more so than the JavaScript implementation, embodied the statement that a consistently well performing generator will be better overall than a generator with a few notably exceptional results. In all tests performed, both simulation based and numeric sequence based, the Random.org dataset remained a not top performing generator but a repeatedly well scoring one. The only exceptions to this being in the Gap and Serial Correlation tests. While these results cannot say definitively whether the data can be considered true random, they can say that the

data collected serves as a highly effective pseudorandom source, comparable to JavaScript or C#.

The Lehmer Generator data was split into two main types of implementations, integer based and real based. Throughout testing it became clear that the real implementations were significantly more effective than the integer implementations. The trends shown in the Kolmogorov-Smirnov test highlight this, with the integer trends completely different to the expected and the real trends. In other numeric sequence tests like the Serial and Gap tests the integer versions were shown to consistently either fail or score substantially worse than their counterparts. The two most notable exceptions to this were the Birthday Spacings test, in which all four versions failed, and the Serial Correlation test where the Integer Version 2 implementation came the closest to the target of 0. Overall, the Real versions of the Lehmer Generator performed acceptably, with results on par with those found in JavaScript.

Going into testing, it was assumed that the Middle Square Method would perform poorly. Throughout the numeric sequence tests performed as expected the Middle Square data did fail and was easily the worst performing dataset evaluated. Apart from Birthday Spacings, it was unable to pass a single test. Even when passing, the Middle Square data was the lowest performing passed dataset sampled. Again, this was not unexpected. The Middle Square Method served as the starting point for pseudorandom generation and as such was going to perform worse than its more modern and better designed contemporaries. It was not included in this investigation because it was considered a viable alternative to other more popular generation methods, but because it was prudent to evaluate its effectiveness in comparison to what's on the market now, to show the improvements made after almost 75 years.

The final datasets to discuss were the White Noise implementations. These were quite different from the other datasets sampled, as the data provided came directly from a source rather than through an algorithm. This is reflected in the test results, which unfortunately were often poor. Much like the Middle Square Method, the White Noise data failed most of the tests they were surveyed in, except for the Birthday Spacings test in which they were the highest performing implementations. The reason for this is two-fold. For one the data sampled wasn't given the same limitations as other implementations, meaning the data had a far wider range of possible values to sample from. Secondly, many of the tests were designed with likely possibilities in mind and so when evaluating a dataset that contained completely random sequences, this true random nature was seen as too unlikely to be valid in the majority of circumstances.

Conclusion

The main aim of this investigation was to evaluate a collection of pseudorandom algorithms, comparing them against true random generators to test their versatility. To do this the data collected from each of these generators was examined by a collection of empirical tests to decern both randomness and value distribution. These results were then graphed and analysed to determine generator performance. After testing, it was clear that Python, along with the other pseudorandom generators

available to both the public and scientific community, was a suitable sources of pseudorandom number generation with outputs able to closely resemble, although not fully replicate, true random functionality. Having observed the capabilities of the earliest attempts to produce random sequences digitally, the advancements made in this field and the improvement in accuracy of modern pseudorandom generators is clear.

The execution of this investigation did, at times, leave much to be desired. Should work like this be undertaken again, additional research into the effects of different seed values should be considered and by extension an increase in the amount of alternate data collected for each implementation. Perhaps the investigation's greatest weakness was the data used, as each implementation only provided a single set of generated values for each required dataset. As many of the functions used incorporate the system clock for seed production it is possible to record data at different points in time in order to produce a different selection of values. In addition, the methods used to sample and format the White Noise datasets were unideal. Microphone quality and recording compression methods altered the data before it could be transformed into a suitable wavelength. Data cleaning was also ignored in this implementation to better achieve a true random sequence, however adjusting the sound levels to fall within a certain range could allow for more testable datasets.

Overall, the investigation can be considered a success, as the primary goals set out were achieved. Should this work be undertaken again in the future, the main focus would be on increasing the scope to incorporate a greater variety of pre-existing digital generators, and addressing the issues mentioned above.

References

Alsultanny, Y A. (2008) 'Random-bit sequence generation from image data', Image and Vision Computing, 26(4), pp. 592-601, accessed online at: https://www.sciencedirect.com/science/article/pii/S026288560700114X

Baldanzi, L, Crocetti, L, Falaschi, F, Bertolucci, M, Belli, J, Fanucci, L and Saponara, S. (2020) 'Cryptographically Secure Pseudo-Random Number Generator IP-Core Based on SHA2 Algorithm', Sensors, 20(7), accessed online at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7180860/

Chen, I-T. (2013) 'Random Numbers Generated from Audio and Video Sources', Mathematical Problems in Engineering, 2013, accessed online at: https://www.hindawi.com/journals/mpe/2013/285373/

James, F and Moneta, L. (2020) 'Review of High-Quality Random Number Generators', Computing and Software for Big Science, 4, accessed online at: https://link.springer.com/article/10.1007/s41781-019-0034-3#citeas

Knuth, D. (1998) The Art of Computer Programming Volume 2: Semi-numerical Algorithms. Third Edition. Addison-Wesley Professional

Luengo, E, Cerna, M, Villalba, L and Hernandez-Castro, J. (2022) 'A new approach to analyze the independence of statistical tests of randomness', Applied Mathematics

and Computation, 426, accessed online at: https://www.sciencedirect.com/science/article/pii/S0096300322002004

Marsaglia, G. (1993) ‘Technical Correspondence: Remarks on Choosing and Implementing Random Number Generators’, Communications of the ACM, 36, accessed online at: https://www.firstpr.com.au/dsp/rand31/p105-crawford.pdf

Microsoft, (2023) A tour of the C# language https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

Park, S and Miller, W. (1988) 'Random Number Generators: Good Ones Are Hard To Find', Communications of the ACM, 31, accessed online at: https://dl.acm.org/doi/pdf/10.1145/63039.63042

Random.org, (2023) https://www.random.org/

Robert G. Brown's General Tools Page, (2023) DIEHARDER test suite https://webhome.phy.duke.edu/~rgb/General/dieharder.php

Schaathun, H. (2015) 'Evaluation of splittable pseudo-random generators', Journal of Functional Programming, 25, accessed online at: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/evaluation-of-splittable-pseudorandom-generators/3EBAA9F14939C5BB5560E32D1A132637

Stoppard, T. (1966) Rosencrantz and Guildenstern Are Dead. Faber & Faber

Tsai, J-M, Chen, I-T and Tzeng, J. (2009) 'Random Number Generated from White Noise of Webcam', accessed online at: https://www.researchgate.net/publication/221566107_Random_Number_Generated_from_White_Noise_of_Webcam

Von Neumann, J. (1951) Monte Carlo Method. Washington, DC: National Bureau of Standards Applied Mathematics Series

Wikipedia (1), (2023) RANDU https://en.wikipedia.org/wiki/RANDU

Wikipedia (2), (2023) Lehmer Random Number Generator https://en.wikipedia.org/wiki/Lehmer_random_number_generator

Wikipedia (3), (2023) Python (programming language) https://en.wikipedia.org/wiki/Python_(programming_language)

Wikipedia (4), (2023) JavaScript https://en.wikipedia.org/wiki/JavaScript

Wikipedia (5), (2023) Middle-square method https://en.wikipedia.org/wiki/Middle-square_method