**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

Many of these tests would have been provided in the Dieharder test suite, including the Kolmogorov-Smirnov and Birthday Spacings tests, however after technical issues regarding the set-up of a test battery the test suite shown above had to be produced manually within R Studio. 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 within R Studio. Although similar tools existed within languages like Python or MATLAB, the ggplot2 library available with R could produce much higher quality figures compared to Python alternatives like Matplotlib or Plotly while also providing a substantial amount of control to the user.

A screenshot of a computer code

Description automatically generated

*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. However, for coin and dice data that had a maximum number of possible outputs of either 2 or 6, these datasets were easily mappable to the distribution table.

A math equation with numbers and symbols

Description automatically generated

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

A table of numbers with numbers

Description automatically generated

*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**).

A computer code with text

Description automatically generated with medium confidence

*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. The *CDice1* list contains the frequency of observed values (Yn) produced by the first implementation of the C# dice simulation while the expected values (np) of each dice face occurring is set to 83.333… (Total Iterations (500) \* Probability of Outcome (1/6)). Every outcome in each dataset has its observed value compared against its expected value producing a V value for each dataset. Eight V values were produced from the collected dice data, two from C#, three from Python, one from JavaScript, one from Random.org, and one from physical dice.

A graph of different colored lines

Description automatically generated

*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.

The C# datasets scored V values of 1.192 (unseeded) and 4.024 (seeded). To compare these values 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.

The Python datasets scored V values of 3.064 (Randint), 2.536 (NumPy unseeded), and 3.88 (NumPy seeded). 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 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 scored a V value of 37.12 which placed it 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 scored a V value of 4.936 which placed it between the .9 and .1 distributions. Considering the claim of Random.org that all results produced through their site can be considered true random, this distribution placing was not unexpected. 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 scored a V value of 6.28 which placed it 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.

A graph of blue and pink bars

Description automatically generated

*Figure 48. 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. Eight V values were produced from the coin flip datasets, two from C#, three from Python, one from JavaScript, one from Random.org, and one from a physical coin.

The frequency of observed values for each data source is shown in Figure 48. Interestingly, with the exception of the real coin values, every generator sampled produced more tails than heads. However, while some datasets had a noticeable difference in total heads vs tails such as the C# unseeded dataset, the NumPy datasets produced an almost even split of heads to tails.

The C# datasets scored V values of 1.352 (unseeded) and 0.8 (seeded). These V values placed both implementations between the .9 and .1 distribution again allowing the dataset to be considered acceptably random. While both implementations remained valid methods of pseudorandom generation 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 scored V values of 0.968 (Randint), 0.008 (NumPy unseeded), and 0.008 (NumPy seeded). This placed all three implementations between the .9 and .1 distribution which remained consistent with the dice simulation results. Unlike 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 scored a V value of 0.128 placing it 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. It was unclear why the dice simulation produced a sequence that contained such a significant decrease in 1s and 6s, but it is evident that this irregular pattern did not persist in the coin simulation. 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 scored a V value of 0.512 placing it 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 a V value of 0.128, a score identical 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. While this does aid in showing the validity of the JavaScript generator, it must be considered that the limited possible outcomes of a coin flip presents a scenario where similar or matching outputs between data sources is far more likely.

**4.3 Kolmogorov-Smirnov Test of Equal 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 equal distribution, and as such an equal weighting, of all data provided. This is given by a test value which must not score above 0.5 to be classified as equally 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.

A screenshot of a computer code

Description automatically generated

*Figure 49. A screenshot of the KS function using C# rand data*

Figure 49 shows the implementation of the Kolmogorov-Smirnov test with the C# data. When a data source contained more than one data set, both 1 sample and 2 sample tests were performed. The results of the testing are shown in figure 50. To visualise the equal 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# 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 50. A table of results for the Kolmogorov-Smirnov Test*