**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.

|  |  |
| --- | --- |
| 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. 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 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.

A graph of blue and pink bars

Description automatically generated

*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. 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 49. 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.

|  |  |
| --- | --- |
| 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. 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 were all placed 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 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. 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 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. 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 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, and as such an empirical weighting, of all data provided. This is given by a 1-sample test value which must not score above the critical value of 0.501 (0.5 + 0.001 (the upper bound of N=100)) and with datasets containing multiple implementations 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.

A screenshot of a computer code

Description automatically generated

*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. 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 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*

A graph of a number of data

Description automatically generated with medium confidence

*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.

A graph of a number of numbers and a number of numbers

Description automatically generated with medium confidence

*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. The reason for this can be seen in figure 54, which shows the Real implementations following the expected upwards trend seen in figure 53 while the Integer version 1 begins around 0.4 instead of 0 and Integer version 2 shows almost no upwards trend at all.

A graph of a number of data

Description automatically generated with medium confidence

*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. Although they failed the 1-sample test, the unseeded and seeded Random functions did pass the 2-sample test and as shown in figure 55 all five of the Python implementations showed an empirical distribution.

A graph with blue dots

Description automatically generated

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

The JavaScript dataset passed the 1-sample KS test although was unable to perform the 2-sample test due to lack of alternate implementations available within the language. Figure 56 shows the distribution of the JavaScript data which followed the expected trend. The empirical distribution shown, in addition to the results of the coin simulation data, helps to confirm the potential validity of the JavaScript implementation despite its original poorer performance when simulating dice rolls.

A graph with blue dots and white text

Description automatically generated

*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 and was unable to complete a 2-sample test due to lack of additional data. 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. While other unsuitable generators sampled in this investigation such as the integer based Lehmer generators show a similar horizontal instead of vertical trend, the Middle Square dataset is the best example of this.

A graph with blue dots

Description automatically generated

*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, these being at 0, 0.2, 0.7 and 0.9, which could explain why the dataset was only just beyond the point of passing. Overall, the data is visibly similar to that seen in figures 56 or 53.

A graph showing a number of different colored lines

Description automatically generated

*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. All three trend upwards, however only the Park and Roundabout data reach the max Fn(x) value of 1.0 as the Sea data achieves a maximum of 0.8. All three datasets show a section of horizontal trending although this trend is most significant in the Park and Sea data.

A graph of a graph

Description automatically generated

*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. This was likely due to the differences in format that these generators produced, with the white noise data covering a far larger range of possible outputs and the cryptographic generator providing a secure sequence that shouldn’t match any results from reproducible pseudorandom generators. 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*

A graph of colored rectangular bars

Description automatically generated with medium confidence

*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. The most likely explanation for this is that the cryptographic sequence produced focused on avoiding expected or predictable sequencing to allow for its use in secure systems.

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 meaning the serial sets it had produced were highly unlikely.

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. This overall decent performance does help to reinforce the claim that Random.org is a true random number generator, with sequences being considered satisfactorily random across multiple tests.

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. This was obviously due to the repetitive nature of the method which would begin to iterate when the seed used for calculations would constantly loop back to itself. 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. Although this meant that these datasets couldn’t be compared to the other implementations, they could still be compared against each other. The Roundabout white noise data performed best in both test statistic and p-value while the Park white noise was the worst out of the three. 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. Although it is possible for generators both random and pseudorandom to yield sequences comprising of repeated sequential values or sequences of completely unique non-repeating values, the expected likelihood of this happening is low. 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. Additionally, the datasets being evaluated in this test were also compared against each other. 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*

A graph of different colored bars

Description automatically generated

*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 however this was expected for the Gap test as a cryptographically secure sequence was unlikely to feature expected intervals between values.

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, only 0.04 away from unseeded random. 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. This was unexpected because of the randomness shown by the dataset in other tests however one explanation for this could be that the source of numbers used by Random.org, atmospheric noise, is more variable and as such is likely to feature longer gaps between recurring values than sequences produced by algorithm.

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. It was made clear throughout these tests that the integer versions of the Lehmer Generator were more than likely unsuitable for pseudorandom number generation, with the real based generators able to fill the requirements of a pseudorandom generator more consistently.

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. Having the same few values occurring in the same order was punished heavily in this test and due to the middle square data predominantly containing the previously mentioned looping values it was not possible for this implementation to pass.

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. These datasets most likely failed the Gap test for the opposite reason to the middle square data. Where the Middle Square data failed due to the lack of gap between repetitions, the White Noise data failed because of the far larger than expected gaps between repetitions. By featuring mostly unique non-repeating values, the datasets were unable to match the expected intervals required by the Gap test.

**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. For this test eight datasets were provided, two from C#, three from Python, and three physically shuffled decks. In order 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.

A graph of different colored dots

Description automatically generated

*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 the 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 are 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 while the other decks only had up to three matching cards grouped at a time. 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 (for example a group of Clubs from position 15 to 22 in Real Card Shuffle 1 and a group of Diamonds from position 8 to 16 in Real Card Shuffle 3). 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.

A graph of different colored bars

Description automatically generated

*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. Each of these collections of 5 would be considered a player’s hand and any combination of those cards that formed a playable outcome, based only on suit, would be recorded. 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. After seeing the distribution of suits in figure 65 the frequencies of outcomes were as expected. 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. This wasn’t expected considering the distribution of suits shown in figure 65, in which groups rarely went above 2 cards. However, when drawn sequentially this can be explained as groups of these pairs can be drawn together, allowing for Three or Four of a Kind hands. The NumPy implementations gave results similar to those seen with the C# implementations, primarily a majority of Two Pair and Three of a Kind hands although the seeded dataset contained two Four of a Kind hands and no Three of a Kind hands.

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. Based on the distributions seen in figure 65, these frequencies match what was expected, with the exception that the number of Full House results were lower than expected given the number of groups of two or three that were found in the physical datasets.

A graph of different colored bars

Description automatically generated

*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. The hands available in this simulation extended beyond the suits, utilising the numeric or face value of a card which brought new hand outcomes including High Card (where a hand contains no matching cards, and the players highest card is put forward instead) and Straight (where the five cards all increase/decrease sequentially). As before all dealing was done through a custom-made function. The incorporation of the multiple hands also meant that the wins and loses of each player could also be tallied, to ensure that no position at the table had an advantage over the others.

|  |  |  |  |
| --- | --- | --- | --- |
| Player | Wins | Loses | 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, even the lowest scoring player at the table would have likely achieved at least one win, as the total number of hands that could have won the round but only drew because they were matched by another player was high.

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 of matching suit, and the incorporation of the number value of a card being needed as well as the suit to form a pair which makes acquiring the cards needed to form a hand substantially more difficult.

This trend repeated in the Python datasets, with none of the implementations able to achieve hands greater than a Pair. Again, the already small clusters of suits divided amongst the six players at the table and the need for matching numeric or face values made forming viable hands harder.

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.

A graph of different colored bars

Description automatically generated

*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’ and players may use a combination of cards in their hand and the communal cards on the river to form a winning hand. Texas Hold ‘Em poker sees high levels of play digitally and is the more popular ruleset used when playing poker 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/loses of each player were recorded to ensure that no position at the table gave an unfair advantage.

|  |  |  |  |
| --- | --- | --- | --- |
| Player | Wins | Loses | 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’s possible that the results given in figures 68 and 70 could indicate that player 5 was at an advantage, especially to player 3 who was unable to win a single round in both games played, however in Texas Hold ‘Em players besides 3 were also unable to win a round and player 5 only had one more win than player 1, meaning any advantage offered was minimal. 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. Pair, Two Pair and Three Pair are more common and in the unseeded dataset no High Cards were used. 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. This allows for groups of two or three to stay together more often, leading to more pairs being available for the players.

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. Again, while unlikely these results do not automatically deem the Real Card Shuffle 3 dataset invalid as they only cover one round of play. The dataset should only be considered invalid if a result like this was seen on a more frequent basis in the remainder of the deck which, based on figure 65, it does not.

**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*

A graph of different colored bars

Description automatically generated

*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. This was an unexpected result, as in other tests both Rand datasets performed similarly.

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. As was seen in the Poker test, the sequences produced by the generators using an unseeded method follow a more concise pattern, usually shown as small groups of similar values. Because of the Runs test checking for values that increase/decrease sequentially, sequences that are arranged in that type of pattern will be less likely to meet the expected numbers predicted by the test.

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