Problem Sheet 1 Notes

Luke Jones

Candidate Number: 229538

1 Using Random Numbers

```
// To compile I entered the following:
g++ -o q1 qOneMain.cpp randomDouble.cpp qOneFunctions.cpp argChecker.cpp
```

I used ***srand(time(NULL))*** for my pseudo-random seed. Providing 2 instances of the application are not executed within a second of one another, this is sufficiently random. To calculate sample variance, I used Bessel's correction of dividing by $(n{-}1)$ as seen in equation 1.

$$s^2 = \frac{\sum (x_i - \bar{x})^2}{n-1} \tag{1}$$

I chose to implement this because the question specified a sample variance and I decided this would be more appropriate when the value of n is small. When n is sufficiently large, the calculated sample variance $\approx 0.08\overline{3}$ which is $\frac{1}{12}$ suggesting the sample does have a uniform distribution. I tried to illustrate this in my programme by iterating through an array of increasing sizes of n from 10 to $1,000,000$ to show how the sample variance converged toward $0.08\overline{3}$ as the sample size increased.

2 Simulating Dice Throws

```
// To compile I entered the following:
g++ -o q2 qTwoMain.cpp randomDouble.cpp rollingDice.cpp argChecker.cpp
```

Because one of the constraints specified that the generate dice values had to be uniformly distributed, I had a problem finding a way to produce random numbers in the interval $[1, 6]$ without using the modulo operator. The common answer found when simulating dice rolls uses the standard C++ function ***rand()*** as seen below:

```
rand() % 6 + 1;
```

We need to understand that ***rand()*** produces an integer between 0 and 32767. The inherent problem with the use of the modulo operator is that it introduces bias towards smaller numbers which would therefore negatively impact the simulation of rolling a 'fair' dice. Whilst using rejection is an alternative solution to remove bias, I instead implemented the following functions:

```
// Returns double in interval [0,1)
double randDouble()
{
  return static_cast<double>(rand()) / RAND_MAX;
}

int dice()
{
  return randDouble() * 6 + 1;
}
```

By using my function ***dice()*** in my dice simulations, I was able to remove the modulo operation and thus eliminate the modulo bias. After running the simulation several times, each with an increased number of throws, I produced several graphs in Python that illustrated the distribution of scores when rolling 2 dice independently and summing their face values. By using this style of graph, it also highlights whether my dice rolling functions are correct; if the functions are working as intended, we should see a score distribution that corroborates with the known probabilities of rolling 2 dice and summing the score.

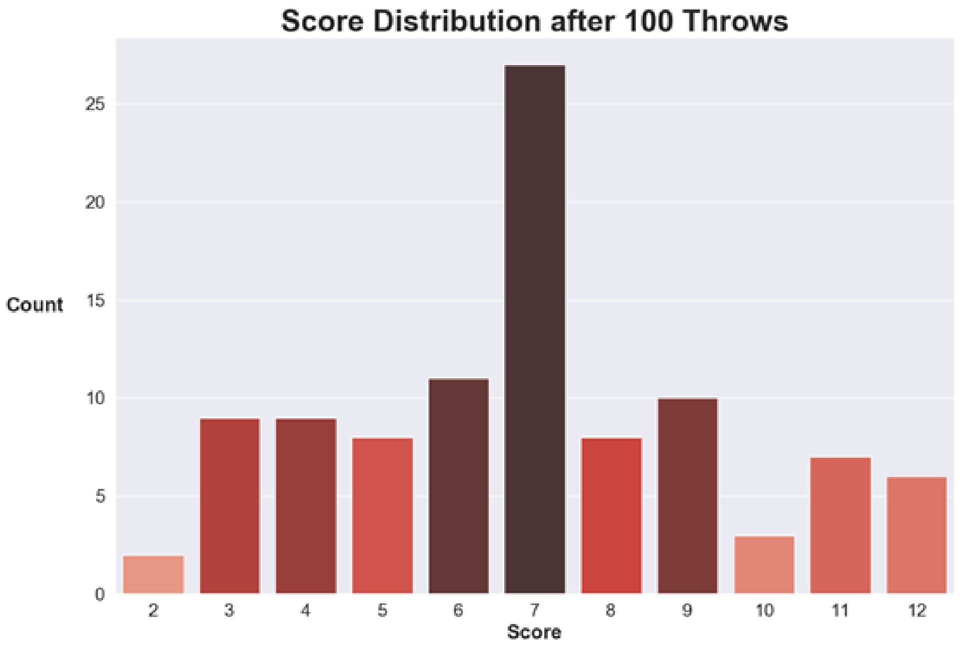


Figure 1: A graph showing the distribution after 100 dice throws.

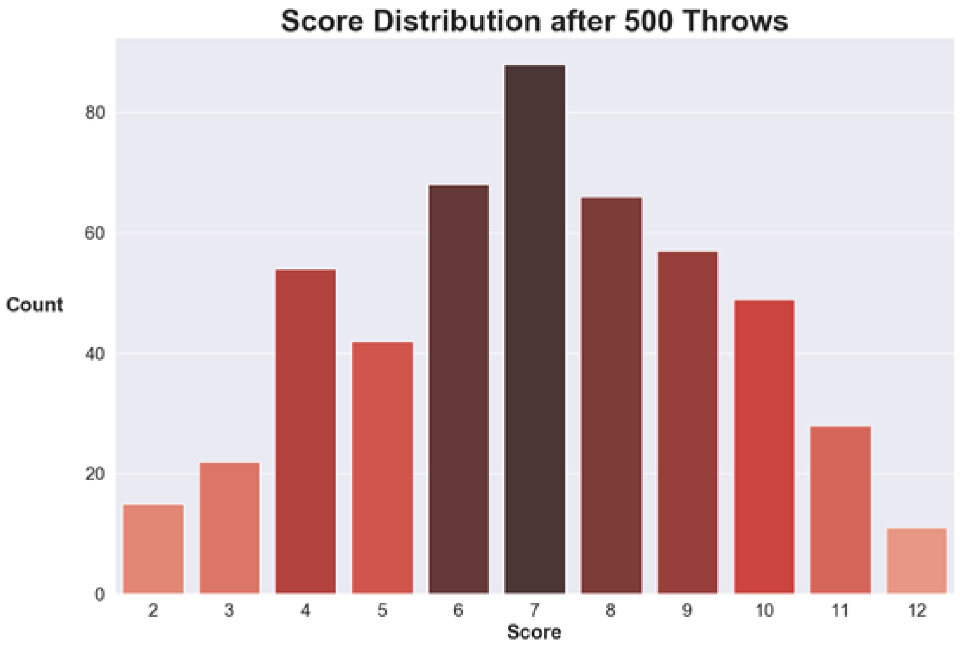


Figure 2: A graph showing the distribution after 500 dice throws.

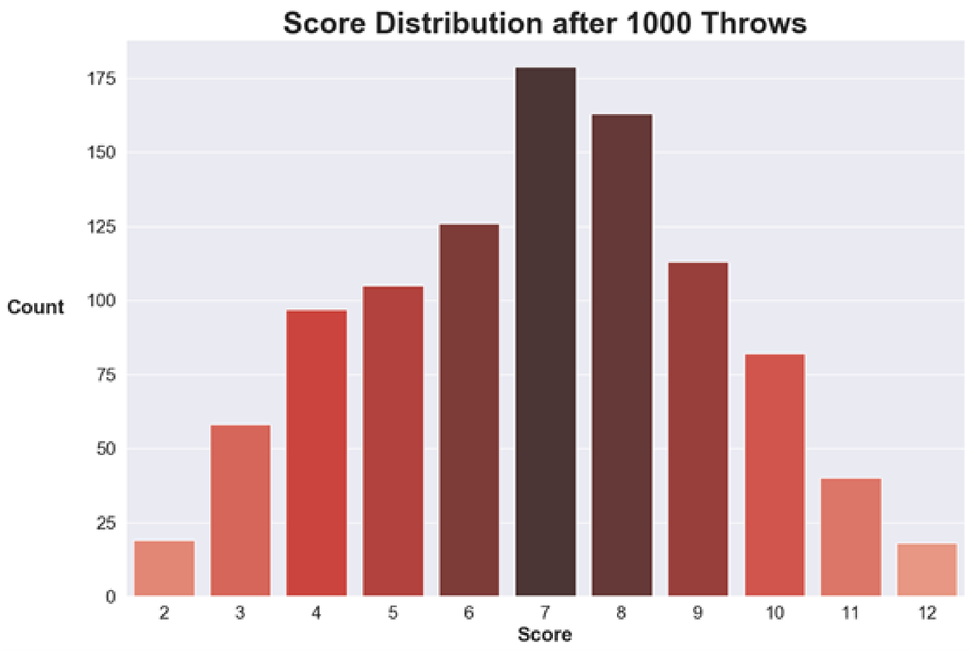


Figure 3: A graph showing the distribution after 1,000 dice throws.

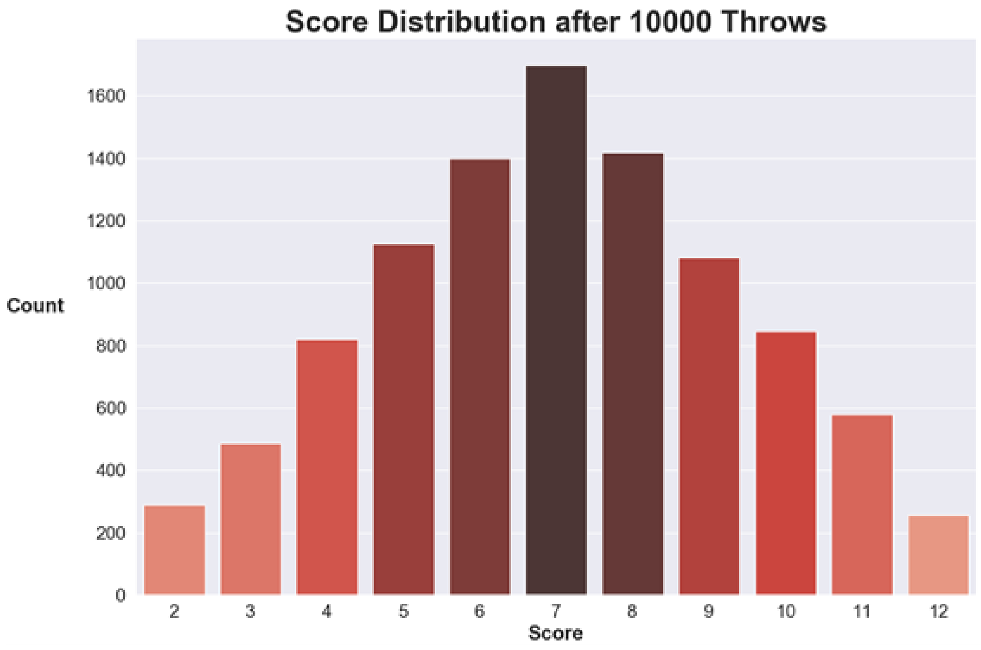


Figure 4: A graph showing the distribution after 10,000 dice throws.

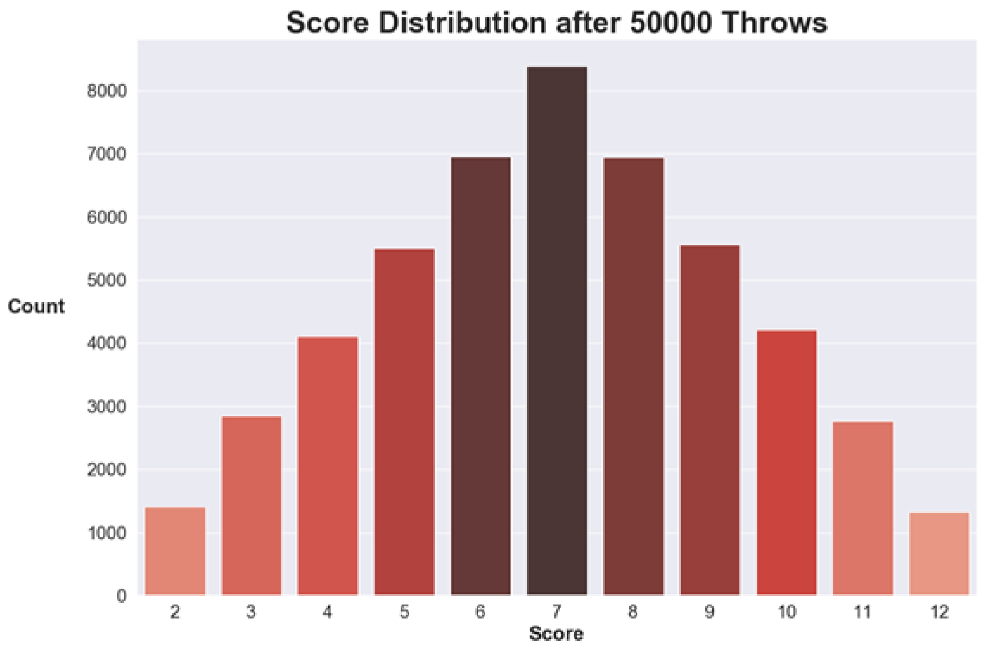
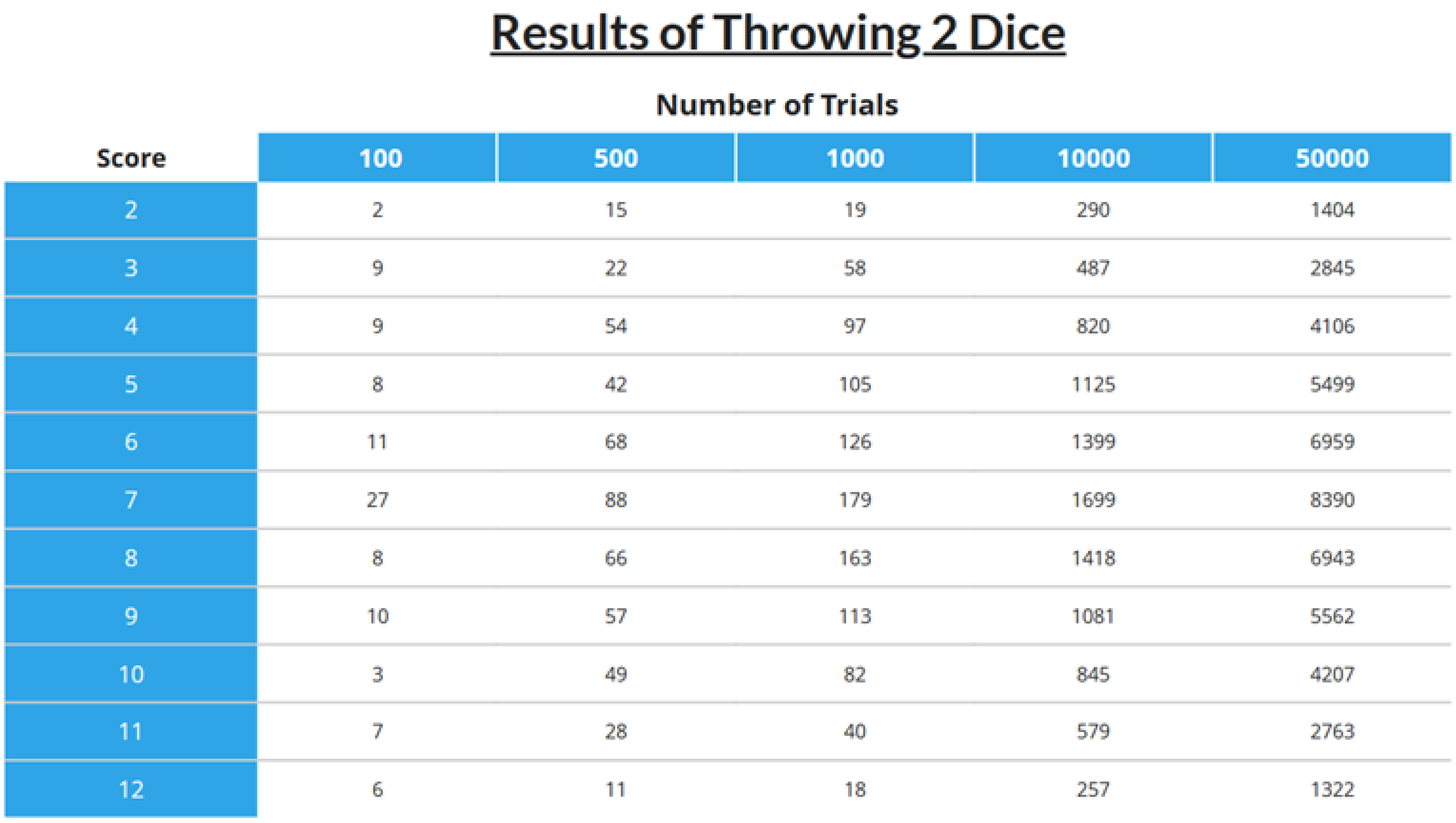


Figure 5: A graph showing the distribution after 50,000 dice throws.



Results of Throwing 2 Dice

	Number of Trials				
Score	100	500	1000	10000	50000
2	2	15	19	290	1404
3	9	22	58	487	2845
4	9	54	97	820	4106
5	8	42	105	1125	5499
6	11	68	126	1399	6959
7	27	88	179	1699	8390
8	8	66	163	1418	6943
9	10	57	113	1081	5562
10	3	49	82	845	4207
11	7	28	40	579	2763
12	6	11	18	257	1322

Figure 6: A frequency table of scores as the number of trials increase.

In each graph, the colour of the bars act as an intuitive way to gauge and compare frequencies - as the count increases, the colour of the bar becomes darker. My graphs indicate that as the number of trials increase, the distribution more accurately represents the probability distribution of rolling 2, 6 sided dice and summing their face values. After 100 throws (Fig. 1), a score of 7 is clearly the most common score - this coincides with the known probabilities of rolling 2 dice because a score of 7 has the highest probability of being rolled: $\frac{6}{36} = \frac{1}{6}$. Figure 5. illustrates a clear distribution of scores that does match the expected probabilities of each respective score; this makes me confident in the validity of my dice simulation. I have included a frequency table (Fig 6.) to see precise frequencies of each respective score at varying sizes of n.

In my programme, I decided to utilise the C++ ***stringstreams*** as a neat and efficient way to automatically use the command line argument for n - the number of samples - in the output file.

3 The Game of Craps

I decided to output my simulated games in the form of a .csv file because I found it easier to manipulate .csv files with Python and its graphing libraries. At the bottom of the .csv file, you will find the total number of wins and losses of the simulated games; for the 10,000 games analysed in the report I observed 4846 wins and 5154 losses.

```
// To compile I entered the following:
g++ -o q3 qThreeMain.cpp randomDouble.cpp rollingDice.cpp
```

3.1 How many games are won and lost on the 1st, 2nd, 3rd...20th roll and after the 20th roll?

Figure 7. illustrates the win / loss frequency for each roll. This is then shown more precisely in Figure 8. which is a table of the wins and losses for each subsequent throw, as well as the total number of games and the calculated chance to win on each throw.

3.2 What are the chances of winning in a Game of Craps?

In the 10,000 games explored in this report, the chance to win was 48.46%. In the real Game of Craps the chance to win is 49.3%. If we compare these two values, we could say that in my simulation, you had a lower than expected chance to win. We could also interpret this difference in win rate as error suggesting that these 10,000 games had an absolute error of 0.84%. When applying this to the sample size, however, this is an incredibly low error rate of 0.000084 or 8.4×10^{-5} per game.

3.3 What is the average length of a Game of Craps?

The average in these 10,000 games was 3.361 throws with a standard error of ± 0.03. Using 95% confidence intervals, I attained the range $[3.302, 3.421]$. Therefore, using rounding, I am comfortable concluding that 95% of the games would finish on the 3rd throw.

3.4 Do the chances of winning improve with the length of the game?

No. The player has a higher chance to win on the very first roll than they do to lose; $\frac{2}{9}$ to win compared to $\frac{1}{9}$ to lose. This is then reflected in both Figure 7. and 8. where the calculated win rate on the very first roll is 65.6% whereas the overall chance to win at the Game of Craps is a considerably lower 49.3%. This would suggest that past the first roll, the chance of winning decreases and my evidence in Figure 7 and 8 supports this.

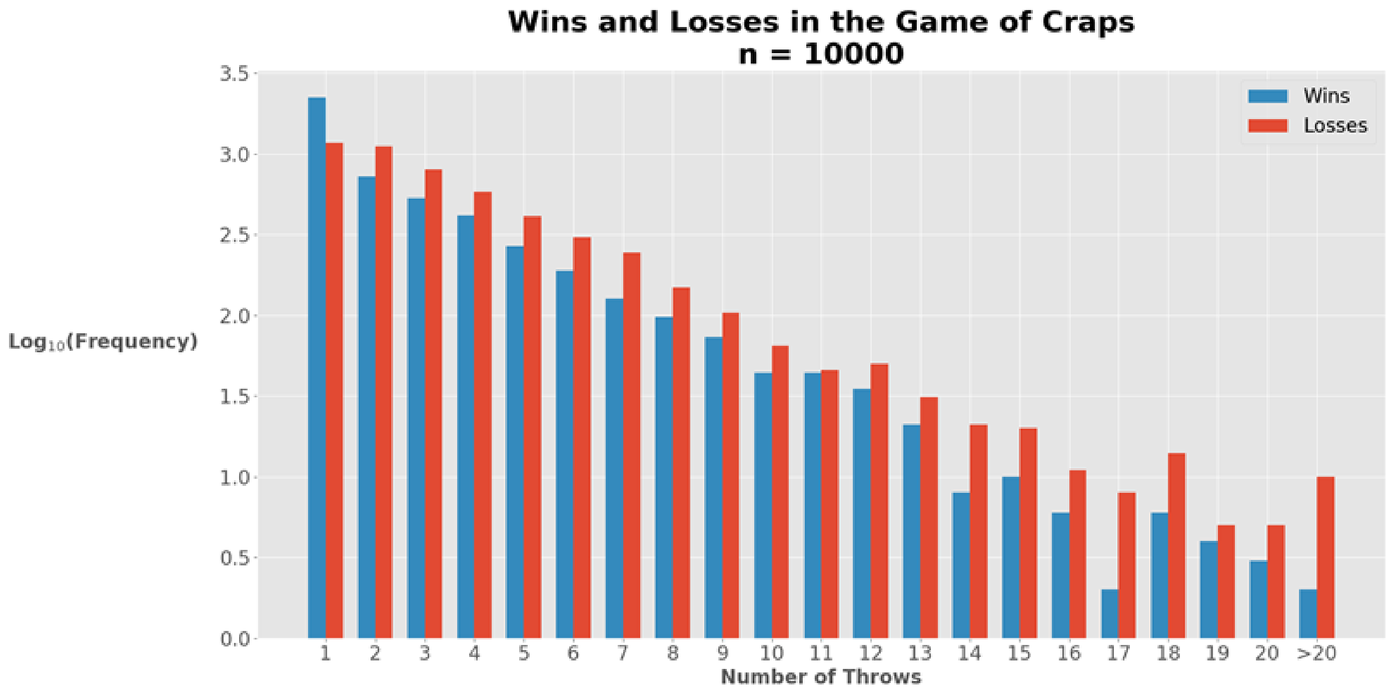
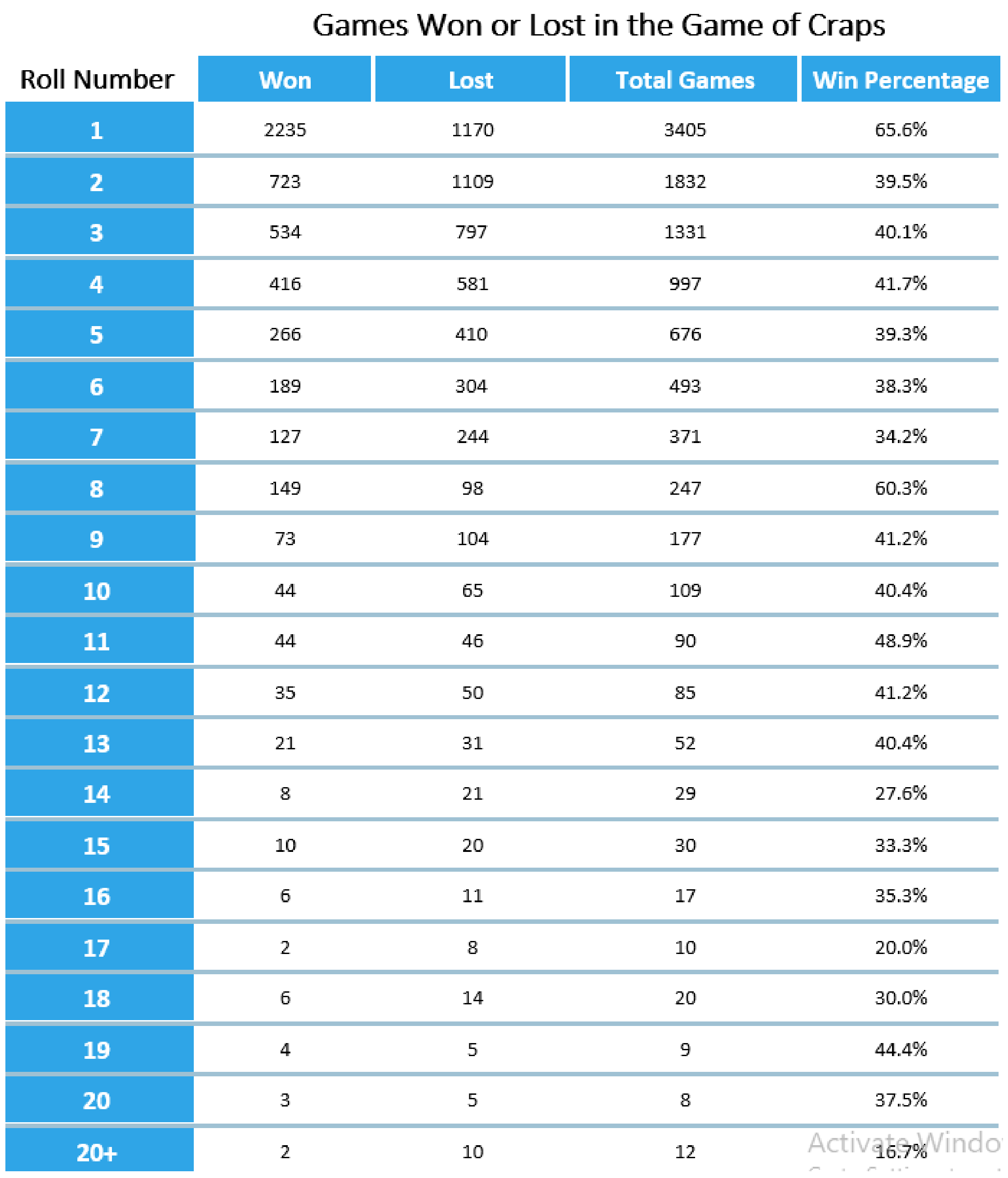


Figure 7: A graph that compares the frequency of wins and losses in the Game of Craps for each throw. A log scale was used because for these 10,000 games the number of games lasting near or exceeding 20 rolls was incredibly small. Notice that the only time where wins exceeded losses was on the very first roll.



Games Won or Lost in the Game of Craps

Roll Number	Won	Lost	Total Games	Win Percentage
1	2235	1170	3405	65.6%
2	723	1109	1832	39.5%
3	534	797	1331	40.1%
4	416	581	997	41.7%
5	266	410	676	39.3%
6	189	304	493	38.3%
7	127	244	371	34.2%
8	149	98	247	60.3%
9	73	104	177	41.2%
10	44	65	109	40.4%
11	44	46	90	48.9%
12	35	50	85	41.2%
13	21	31	52	40.4%
14	8	21	29	27.6%
15	10	20	30	33.3%
16	6	11	17	35.3%
17	2	8	10	20.0%
18	6	14	20	30.0%
19	4	5	9	44.4%
20	3	5	8	37.5%
20+	2	10	12	16.7%

Figure 8: A table overview of how many games were won or lost for each successive throw