RPG Dice Simulator

John Karasinski

God's dice always have a lucky roll.

Sophocles (497/6 — 406/5 BC)

God does not play dice.

Albert Einstein (1879 — 1955)

God not only plays dice, but also sometimes throws them where they cannot be seen.

Stephen Hawking (1942 —)

1 Introduction

God's use of dice may be disputed, but archaeological evidence shows that humans have been using dice for many thousands of years [1]. Dice are often used to insert randomness into games of chance, which can lead to wonderfully unpredictable situations. Due to the randomness associated with them, dice are also excellent for bringing everyone to the same level. A king and a peasant have the same odds when rolling dice, and the dice do not care who is more wealthy, talented, or desperate. While the result of any individual roll of the dice may be random and unpredictable, a basic application of statistics can lead us to make more informed decisions about the likelihood of events.

The project outline states: "Create a function that will allow me to specify the number of dice I want to use per roll, the number of faces on each die (assume all dice used in a roll have the same number of faces), whether I will sum the scores of each die or will have a minimum value needed to count as a success per die, and whether I will include botches or not as an outcome for rolls." Several additional statistics were requested to describe the nature of the outcome of 1,000 repetitions of various dice rolls.

2 Simulation Method

According to standard die notation, die rolls are given in the form AdX. A and X are variables, separated by the letter d, which stands for die or dice [2]. A represents the number of dice, and X represents the number of faces on each die. 1d20, for example, represents one twenty-sided die, and 2d10 represents two ten-sided die.

The primary assumption going forward is that we are considering perfect dice. Perfect dice have an equal probability of landing on each face. A six-sided die (1d6), for example, would then have a $1/6$ probability of resulting in a 1, 2, 3, 4, 5 or 6. Cheap dice purchased for gaming will generally not have this property, and small imperfections will instead slightly favor different outcomes. As we do not want to spend time throwing dice, we will instead use software to simulate the results of their throws. For the purposes of the simulations here, you can pretend that we have purchased *very, very expensive* dice which behave perfectly. The joy of the software used for this project though, is that it is all 100% open source and free.

This simulation will make use of Python 3.5.0 and NumPy 1.10.1. The core idea behind the simulation is to generate pseudorandom numbers to simulate the throwing of dice. The rules given in the project outline (summing, botching) can then be applied to deduce the result of the simulated dice throw.

The basic functionality behind the simulation is the `dice()` function. This function accepts three arguments, `A`, `X`, and an optional `n`, where `A` and `X` take on the same meaning as in standard dice notation, and `n` is the number of times to simulate the throw. Calling `dice(3, 6, n=2)`, for example, could return `array([[2, 2, 6], [5, 3, 4]])`. This example array has 2 rows of 3 simulated 6-sided dice rolls. The first row (representing the first throw) contains the values 2, 2, and 6, and the second row contains the values 5, 3, and 4.

The `dice()` function is essentially a helper function, which calls NumPy's random integer function. NumPy uses the Mersenne twister sequence to generate pseudorandom numbers [3]. As these are psuedo-random numbers and not true random numbers, knowing the seed of the algorithm allows you to reproduce the results, and predict future events with 100% accuracy. For the purposes of this project, however, we will confirm that pseudorandom numbers function sufficiently.

The primary function for this simulation is the `throw()` function, which parses user input and gives raw output, plots, and/or statistics about throws. There are three different ways to use this program. The first way to use this is to simply call `throw('1d6', n=1000)`. This will return the results of 1,000 1d6 dice rolls. The second way to run this is to call `throw('1d6', n=1000, plot=True)`, which will return a plot of the the outcomes of 1,000 1d6 dice rolls (a flat distribution from 1 to 6). The third way to use this program is to call `throw('1d6', n=1000, stats=True)`, which will return the value counts of each outcome of the 1,000 1d6 rolls.

A separate function, `expected()`, was used to enumerate all possible outcomes for each of the 4 primary scenarios we were asked to investigate. Once enumerated, the exact probability of each outcome was calculated. χ^2 tests were used to confirm the output of this simulation is statistically the same as the exact probabilities.

3 Results

The script for this project, `dice.py`, accomplishes all of the scenarios set out in the project outline (see Appendix). The first four scenarios are to perform basic calculates on single rolls of the dice. These were:

1. "Roll 1d20 and I just want to know the score."
2. "Roll 2d6 and I want to know my summed scores."
3. "Roll 4d10 where I need to roll 5s or greater such that each roll of 5 or greater counts as a success."
4. "The same scenario as above, but now each 1 rolled detracts from the number of successes."

These can be accomplished by running the following code:

```
throw('1d20')
# 15
throw('2d6')
# 9
throw('4d10', success_min=5)
# 4
throw('4d10', success_min=5, botching=True)
# 1
```

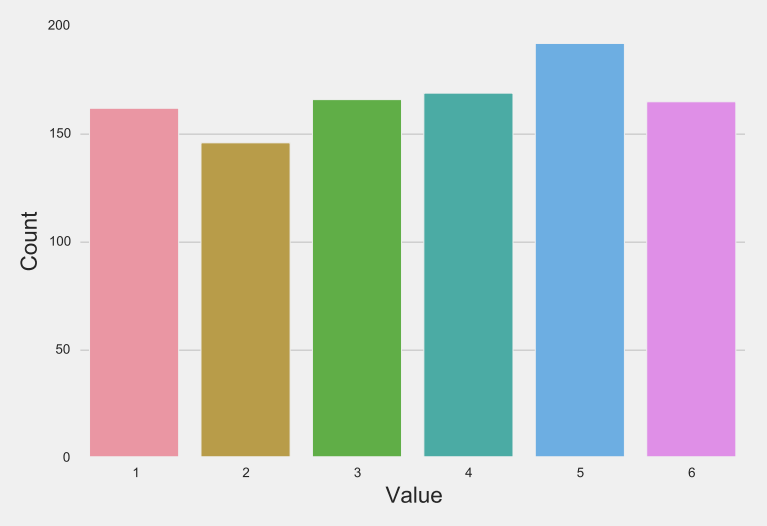
The second set of four scenarios was to plot and table several statistics from various dice roll scenarios. χ^2 tests were used to compare the expected value counts to the observed value counts. These exact probabilities were used along with χ^2 tests to statistically confirm that the values from this simulation are the same at the 5% level of significance. H_0 is that the frequencies of our results from simulated dice rolls is equivalent to the frequencies of the results from the exact probabilities. H_1 is simply that H_0 is false—the resulting frequencies are not equivalent. Four individual tests were run to confirm that the simulation's outcomes are the same as those expected from the calculated exact outcomes. These four results are combined using Fisher's method to test if H_0 is true. The results of these tests are shown below.

3.1 Roll 1d6 1,000 times, frequency of each outcome?

This is a very basic scenario. As we are just rolling a single die, we should expect a flat distribution of all the possible outcomes (1-6). The exact probability of each outcome is $1/6 \approx 16.67\%$. The observed dice rolls showed no statistical difference from the exact probability frequencies $[\chi^2 = 6.60, p = 0.25]$.

The command to run this scenario is:

```
throw('1d6', n=1000, plot=True, stats=True)
```



Value	Observed	Expected
1	162	166.67
2	146	166.67
3	166	166.67
4	169	166.67
5	192	166.67
6	165	166.67

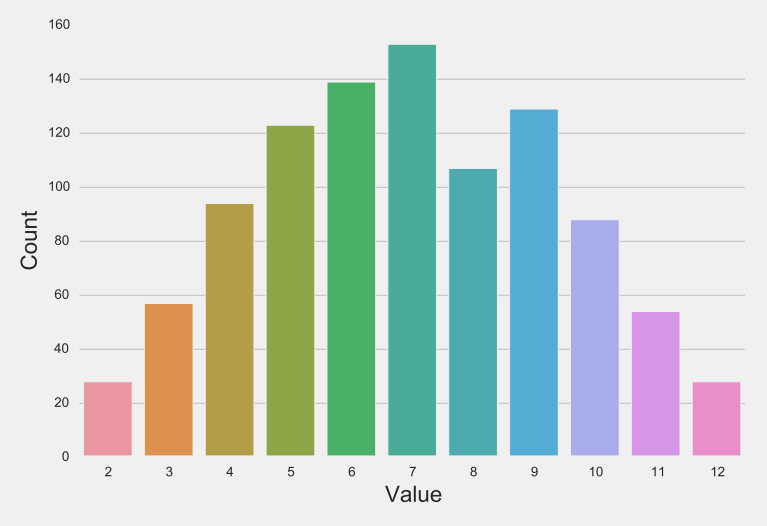
Figure 1: Results of 1,000 1d6 rolls. Observed results are compared to expected results from exact probabilities, and show no differences.

3.2 Roll 2d6 1,000 times, frequency of each outcome? Use the summed scores of each die as your outcome.

This is a slightly more interesting scenario. As we are rolling a two die and calculating the sum, we should expect a triangle distribution centered around the most likely combination, 7, and ranging all the possible values, 2-12. We can calculate each possible outcome and the probability of each. The observed dice rolls showed no statistical difference from the exact probability frequencies, with some minor variation due to the low n $[\chi^2 = 14.31, p = 0.16]$.

The command to run this scenario is:

```
throw('2d6', n=1000, plot=True, stats=True)
```



Value	Observed	Expected
2	28	27.7
3	57	55.5
4	94	83.3
5	123	111.1
6	139	138.8
7	153	166.6
8	107	138.8
9	129	111.1
10	88	83.3
11	54	55.5
12	28	27.7

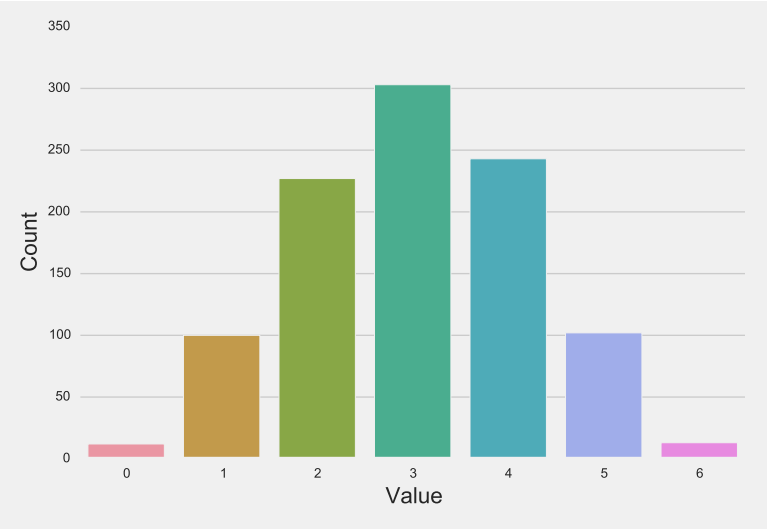
Figure 2: Results of summed values from 1,000 2d6 rolls. Observed results are compared to expected results from exact probabilities, and show no differences.

3.3 Roll 6d10 1,000 times, frequency of each outcome? Treat a score of 6 or better as a success and scores less than 6 as nothing, the outcome is total number of successes (include the frequency of no successes).

Now we no longer care about the exact outcome of each die, we only care whether or not it is greater than or equal to 6. For a 10 sided die, exactly half of the possible outcomes are counted as successes. We can calculate each possible outcome to generate the exact probabilities. The observed dice rolls showed no statistical difference from the exact probability frequencies $[\chi^2 = 3.26, p = 0.78]$.

The following line calculates the statistics for this scenario:

```
throw('6d10', n=1000, success_min=6, plot=True, stats=True)
```



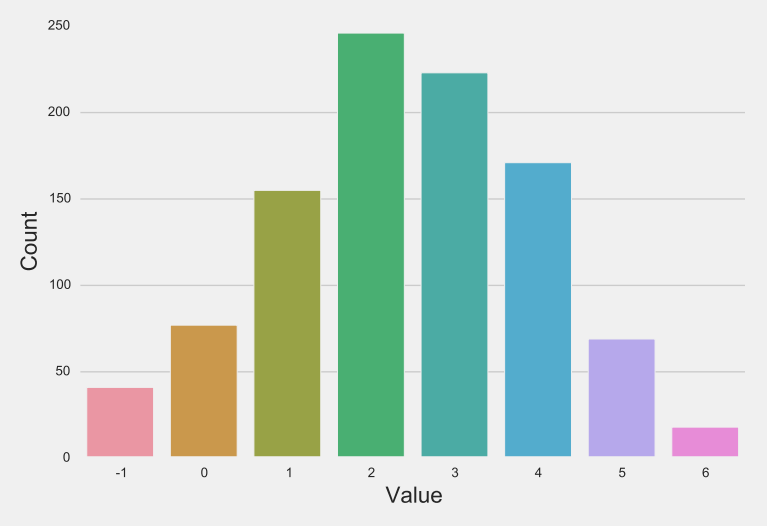
Value	Observed	Expected
0	12	15.6
1	100	93.7
2	227	234.3
3	303	312.5
4	243	234.3
5	102	93.7
6	13	15.6

Figure 3: Results of successes from 1,000 6d10 rolls, where scores of 6 or greater are considered successes. Observed results are compared to expected results from exact probabilities, and show no differences.

3.4 Same as (c) above except now scores of 1 count as a botch. Each botch is subtracted from each success. If there are more botches than success (i.e., -1 successes) then the overall outcome is a botch. Now depict the frequency of botching, and total number of successes (as before, include the frequency of no successes).

This final scenario is very similar to the previous one. For our 10 sided die, there is now a $1/10$ probability of a botch, a $2/5$ probability of a failure, and a $1/2$ probability of a success. We can calculate each possible outcome to generate the exact probabilities. The observed dice rolls showed no statistical difference from the exact probability frequencies $[\chi^2 = 3.64, p = 0.82]$.

```
throw('6d10', n=1000, success_min=6, botching=True, plot=True, stats=True)
```



Value	Observed	Expected
-1	41	42.5
0	77	80.9
1	155	156.7
2	246	225.3
3	223	235.0
4	171	168.7
5	69	75.0
6	18	15.6

Figure 4: Results of succeses from 1,000 6d10 rolls, where scores of 6 or greater are considered successes, and scores of 1 are considered botches. Observed results are compared to expected results from exact probabilities, and show no differences.

3.5 Fisher's method

Using Fisher's method, the results of our four independent χ^2 tests can be combined to determine if H_0 holds. In essence, we are simple combining the independently calculated p-values to test if our overall hypothesis holds. In Python, this can be quite easily computed by:

```
from scipy.stats import combine_pvalues
combine_pvalues([.25,.16,.78,.82])
# (7.3315762457810782, 0.50131680258503031)
```

We conclude that H_0 still holds—the simulation outputs are statistically equivalent to what would be found from truly random choices from their exact probability frequencies [$\chi^2 = 7.33, p = 0.50$].

4 Summary

A Python script was created to allow a user to specify the properties of various dice rolls. These properties include the number of dice per roll, the number of faces on each die (assuming all dice used in a roll have the same number of faces), whether the scores of each die will be summed or will have a minimum value needed to count as a success per die, and whether to include botches or not as an outcome for rolls. Several statistics were calculated to describe the nature of the outcome of 1,000 repetitions of various dice rolls compared to the expected outcome calculated from exact probabilities.

All four χ^2 tests showed that H_0, that the psuedorandom outcomes from the simulation scenarios was equivalent to what would be expected from truly random outcomes from the exact probability frequencies, holds. Fisher's method was used to combine the p-values from these tests, and also suggests that H_0 holds at the .05 level [$\chi^2 = 7.33, p = 0.50$].

References

[1] Carlisle, Rodney P., ed. Encyclopedia of play in today's society. Vol. 1. Sage, 2009.

[2] “Standard Dice Notation”. dice-play. 2006-04-06. Archived from the original on 2007-04-26. https://web.archive.org/web/20070426013749/http://homepage.ntlworld.com/dice-play/Notation.htm

[3] Matsumoto, Makoto, and Takuji Nishimura. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator.” ACM Transactions on Modeling and Computer Simulation (TOMACS) 8.1 (1998): 3-30.

A Source Code

```
import itertools
from collections import Counter
from scipy.stats import chisquare
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('white')
plt.style.use('fivethirtyeight')
np.random.seed(np.array([ord(s) for s in 'John Karasinski']).sum())


def dice(A, X, n=1):
    ''' Generate n instances of an AdX dice roll.'''
    return np.random.randint(low=1, high=X+1, size=(n, A))


def throw(AdX, n=1, success_min=None, botching=False, stats=False, plot=False):
    '''
    Throw dice with optional repetitions, minimum value for success, and
    botching. The results can also be plotted by setting the plot argument.
    '''
    # Pull the values for your roll.
    A, X = np.array(AdX.split('d'), dtype=int)

    if n == 1 and success_min is None:
        # Just the results of the dice roll.
        if A == 1:
            return dice(A, X)[0][0]
        else:
            return dice(A, X).sum()
    elif success_min is None:
        # Statistics on the result of many dice rolls.
        scores = dice(A, X, n).sum(axis=1)
    elif success_min and not botching:
        # Statistics of the results of many dice rolls with a minimum value for
        # success.
        scores = (dice(A, X, n) >= success_min).sum(axis=1)

        # Just the number of successes for this roll.
        if n == 1:
            return scores[0]
    elif botching:
        # Statistics of the results of many dice rolls with a minimum value for
        # success and the possibility of botching.
        d = dice(A, X, n)
        d[d == 1] = -1
        d[np.logical_and(d > 0, d < success_min)] = 0
        d[d > 0] = 1
        scores = d.sum(axis=1)
        scores[scores < 0] = -1
        if n == 1:
            return scores[0]
    else:
```

```
        raise ValueError('Sorry, these values are not supported.')

    if plot:
        filename = AdX
        if success_min: filename += '_min' + str(success_min)
        if botching: filename += '_botching'
        statistics(scores, plot=True, filename=filename)
    if stats:
        return statistics(scores)

    return scores


def statistics(scores, plot=False, filename=False):
    s = pd.Series(scores).value_counts().reset_index()
    s.columns = ['Value', 'Count']
    s['Frequency'] = s['Count']/s['Count'].sum() * 100
    s = s.sort('Value')

    if plot:
        sns.countplot(scores, order=np.unique(scores))
        plt.xlabel('Value')
        plt.ylabel('Count')
        plt.tight_layout()
        if filename:
            plt.savefig(filename + '.pdf')
        else:
            plt.show()
        plt.close('all')
    return s


def expected(section):
    if section == 'a':
        possibilities = np.array(list(itertools.product(range(1, 7))))
    elif section == 'b':
        possibilities = np.array(list(itertools.product(range(1, 7), repeat=2)))
    elif section == 'c':
        possibilities = np.array(list(itertools.product([0, 1], repeat=6)))
    elif section == 'd':
        possibilities = np.array(list(itertools.product([-1, 0, 0, 0, 0, 1, 1, 1, 1, 1],
    repeat=6)))
    else:
        raise ValueError

    counts = Counter(possibilities.sum(axis=1))
    d = pd.DataFrame(counts, index=['Counts']).T.reset_index()
    if section == 'd':
        botches = d[d['index'] < 0].sum()
        botches['index'] = -1
        d = d[d['index'] >= 0].append(botches, ignore_index=True)

    d = d.sort('index')
    d['Expected'] = d.Counts/d.Counts.sum() * 1000
    return d


def main():
    # %timeit throw('1d20')
    # 100000 loops, best of 3: 9.98 s per loop

    # %timeit throw('2d6')
    # 100000 loops, best of 3: 15.4 s per loop

    # %timeit throw('4d10', success_min=5)
    # 10000 loops, best of 3: 20.2 s per loop
```

```
120
121     # %timeit throw('4d10', success_min=5, botching=True)
122     # 10000 loops, best of 3: 31.7 s per loop
123
124     a = throw('1d6', n=1000, plot=True, stats=True)
125     print(a)
126     print(chisquare(a.Count, expected('a').Expected))
127
128     b = throw('2d6', n=1000, plot=True, stats=True)
129     print(b)
130     print(chisquare(b.Count, expected('b').Expected))
131
132     c = throw('6d10', n=1000, success_min=6, plot=True, stats=True)
133     print(c)
134     print(chisquare(c.Count, expected('c').Expected))
135
136     d = throw('6d10', n=1000, success_min=6, botching=True, plot=True, stats=True)
137     print(d)
138     print(chisquare(d.Count, expected('d').Expected))
139
140 if __name__ == '__main__':
141     main()
```