**HW1**

**Brandon Walker**

**Introduction**

This document outlines the test plan for the dice-rolling software component. The purpose of this component is to "simultaneously" roll up to five six-sided dice, with the number of dice to be rolled specified by the user. The primary focus of this test plan is to rigorously verify the randomness and independence of the generated results, as per the project requirements.

**Assumptions:**

* The user input for the number of dice to roll will be a valid integer between 1 and 5, inclusive. Error checking for invalid inputs is outside the scope of this project.
* The output of the dice-rolling routine will be returned as output parameters to the calling routine, not printed directly.
* The software under test is a separate component, and a test harness will be used to automate the testing process.
* Being able to instantiate a die, and use that die to get whatever number of rolls I call for, in a real-world scenario I would thus know that the number of dice called is irrelevant, for instance, I could call six million rolls, and separate it into whatever partitions I wanted for analysis, and it would have exactly the same randomness as calling for six dice a million times. I will be working on the assumption that I am supposed to pretend not to know this.

**Instructions for Alice:**

* To execute the test plan, run the test harness code which will call the dice-rolling component and perform the checks outlined in the following sections.
* The harness is designed to automate the data collection and analysis, including the Chi-squared tests.
* The results will be logged and can be reviewed in the "Actual Output" sections of each test case.

**Tests to be used:**

**1. Distribution Tests**

These check if each outcome happens about as often as it should.

* For a single die, each face (1–6) should appear equally often.
* For two or more dice, each possible sum (like 2–12 for two dice) should appear with the correct probability based on the number of ways it can happen.
* We use the **chi-squared goodness-of-fit test**, which compares the counts we observed with the counts we expected.
* If the differences are small (chi-squared value below the cutoff), the test passes. If it is too large, it fails.

**2. Independence Tests**

These check whether the dice are truly independent — meaning that one die or one roll does not influence another. I check independence in two ways:

* **Horizontal independence (within a single roll of multiple dice):** This checks that the dice in the same roll does not depend on each other. For example, on a two-dice roll, die #1 showing a 2 should not make it more or less likely that die #2 also shows a 2. If the dice are independent, the chance of rolling doubles (both dice equal) will automatically come out correct (1/6). Will also check run length, and frequency of different lengths, for the horizontal runs it will check dice 1, to 2 etc, and then roll over to next roll if it’s in the middle of a run.
* **Vertical independence (across rolls):** This checks that rolls do not depend on previous rolls. For example, if you roll a 3, the number that comes next should be no more or less likely to be a 4, a 5, or anything else. We test this by building a 6×6 “transition table” that records how often each number follows another. If the dice are independent across time, the table will match what we’d expect for random dice. Will also check run lengths across runs and check frequency of different lengths of runs across specific die, as well as independence of sums of rolls.

We use the **chi-squared test of independence** to check both horizontal and vertical independence. This compares the observed 6×6 table with the expected table under the assumption of independence. If the chi-squared value is below the cutoff, the dice pass the independence test.

**Glossary of Formulas and Terms**

Here are the main mathematical tools we are using, explained in plain language:

1. **Observed Count (Oᵢ):**  
   The number of times a particular outcome happened in our test. Example: if we rolled a single die 3,600,000 times and got “4” exactly 600,200 times, then O₄ = 600,200.
2. **Expected Count (Eᵢ):**  
   The number of times we *should* expect a particular outcome if the dice are fair. Example: for one die rolled 3,600,000 times, each face should appear 600,000 times, so E₄ = 600,000.
3. **Chi-squared statistic (χ²):**  
   A measure of how different the observed counts are from the expected counts. The formula is:
   * If Oᵢ and Eᵢ are close, χ² will be small.
   * If Oᵢ and Eᵢ are very different, χ² will be large.
4. **Degrees of Freedom (df):**  
   A number used to determine the cutoff for the chi-squared test.
   * For a distribution test with k categories, df = k − 1.
   * For an independence test with an r\*c table, df = (r − 1) x (c − 1).
5. **Critical Value:**  
   The cutoff number we compare χ² against. It depends on the degrees of freedom and the significance level (1%). If χ² is larger than the critical value, the test fails.
6. **Significance Level (α = 0.01):**  
   The threshold we chose. It means we are willing to accept a 1% chance of mistakenly rejecting fair dice.
7. **Goodness-of-Fit Test (chi-squared version):**  
   A test to see if observed counts match expected counts for a single categorical variable (like die faces or sums).
8. **Test of Independence (chi-squared version):**  
   A test to see if two variables are independent (like die 1 and die 2 or roll t and roll t+1).
9. **Transition Table:**  
   A 6×6 table showing how often one roll is followed by another. Used in horizontal independence tests.
10. **Lag-1 Autocorrelation:**  
    A test that looks deeper than chi-squared, but only at linear relationships. Cutoff by with n being number of rolls.
11. **Permutation P-Value:**  
    A test that looks deeper than chi-squared, for relationships. Returns a value from 0 to 1 that represents percent chance of chi-value being as high as it is, generally if chance is above 5, or 1 when being strict, it is considered unrelated. Computationally EXTREMELY expensive.

**Test Plan**

The following test cases are designed to verify the randomness and independence of the dice-rolling software. The sample sizes for each test are chosen to be large enough to provide statistically significant results for Chi-squared analysis.

**Test Case 1: Single Die Distribution and Independence**

* **Input:** Roll a single die 3.6 million times.
* **Expected Outputs:**
  + **Distribution:** Each face (1, 2, 3, 4, 5, 6) is expected to appear approximately 600,000 times (3,600,000/6). The Chi-squared test should show a p-value that indicates a uniform distribution.
  + **Independence:** The frequency of any specific roll being followed by another specific roll (e.g., a "3" followed by a "4" vs. a "3" followed by a "5") should be approximately equal. For example, a "3" followed by a "4" is expected to occur approximately 100,000 times (3,600,000/36). The Chi-squared test on the sequential pairs should not show a dependency between consecutive rolls.

**Actual Output: Number of times each number was rolled:**

**1: 599689**

**2: 601080**

**3: 599593**

**4: 599883**

**5: 599712**

**6: 600043**

**Critical value (df=5, alpha=0.05): 11.07**

**Result: 2.54542 Pass**

**Transition Counts (for independence test):**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** |
| **1** | 100102 | 99980 | 99782 | 100122 | 99813 | 99889 |
| **2** | 99829 | 100023 | 100817 | 99885 | 100406 | 100120 |
| **3** | 99481 | 100300 | 99774 | 100327 | 100185 | 99526 |
| **4** | 100029 | 100303 | 99832 | 99831 | 99867 | 100021 |
| **5** | 99651 | 100462 | 99624 | 100047 | 99641 | 100287 |
| **6** | 100597 | 100012 | 99764 | 99671 | 99799 | 100200 |

**Critical value (df=25, alpha=0.05): 37.65**

**Result: 27.3578 Pass**

**Test Case 2: Two Dice Distribution, Sums, and Independence**

* **Input:** Roll two dice 3.6 million times.
* **Expected Outputs:**
  + **Distribution:**
    - **Faces:** Each face (1, 2, 3, 4, 5, 6) is expected to appear approximately 1,200,000 times (3,600,000/6x2). The Chi-squared test will be used to test for a uniform distribution.
    - **Sums:** The distribution of the sum of the dice should follow a normal distribution. The Chi-squared test will be used to test for this distribution.
  + **Independence:**
    - **Horizontal:** The results of the first die across all rolls should be independent of the results of the second die for the same roll. A Chi-squared test will be used to test for significant dependency.
    - **Vertical:** The outcome of a die from one roll should not influence the outcome of a die from the next roll. Chi-squared test of independence will be used to test for significant dependency.
      * **Sums:** The outcome of the sums of one roll should not influence the outcome of another roll. A Chi-squared test will be used to test for significant dependency.
* **Actual Output:** Number of times each number was rolled:

1: 1198987

2: 1200547

3: 1199698

4: 1200599

5: 1200121

6: 1200048

Critical value (df=5, alpha=0.05): 11.07

Result: 1.49361 Pass

Sum distribution:

2: 100090

3: 199692

4: 299917

5: 399991

6: 499949

7: 599620

8: 500009

9: 400681

10: 299924

11: 200414

12: 99713

Critical value (df=10, alpha=0.05): 18.31

Result: 3.68384 Pass

Horizontal Transition Counts (for independence test):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** |
| **1** | 199193 | 200812 | 199359 | 199801 | 200018 | 199804 |
| **2** | 200152 | 199646 | 199894 | 200772 | 200383 | 199700 |
| **3** | 200468 | 200511 | 199670 | 199994 | 199427 | 199628 |
| **4** | 199673 | 199908 | 200163 | 200093 | 200598 | 200163 |
| **5** | 199958 | 199351 | 200326 | 199895 | 200063 | 200527 |
| **6** | 199542 | 200319 | 200286 | 200043 | 199632 | 200226 |

Critical value (df=25, alpha=0.05): 37.65

Result: 25.9478 Pass

**Vertical Transition Counts (for independence test):**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** |
| **1** | 200180 | 199692 | 199638 | 199727 | 200280 | 199470 |
| **2** | 199692 | 200558 | 200264 | 200311 | 199811 | 199911 |
| **3** | 199638 | 200264 | 198716 | 200339 | 199958 | 200783 |
| **4** | 199727 | 200311 | 200339 | 200280 | 199898 | 200044 |
| **5** | 200280 | 199811 | 199958 | 199898 | 199760 | 200414 |
| **6** | 199470 | 199911 | 200783 | 200044 | 200414 | 199426 |

Critical value (df=25, alpha=0.05): 37.65

Result: 27.1421 Pass

Sum Transition Counts (for independence test):

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 2 | 2660 | 5597 | 8382 | 11112 | 13896 | 16736 | 13920 | 11122 | 8411 | 5540 | 2714 |
| 3 | 5498 | 11127 | 16745 | 22314 | 28086 | 32993 | 27780 | 22032 | 16548 | 11065 | 5504 |
| 4 | 8407 | 16707 | 25072 | 33262 | 41787 | 50273 | 41379 | 33417 | 24788 | 16551 | 8274 |
| 5 | 11238 | 22073 | 33382 | 44485 | 55494 | 66840 | 55510 | 44435 | 33277 | 22094 | 11163 |
| 6 | 13751 | 27583 | 41423 | 55579 | 69511 | 82780 | 69314 | 55685 | 42301 | 28048 | 13974 |
| 7 | 16669 | 33126 | 49949 | 66409 | 83017 | 100073 | 83765 | 66612 | 49972 | 33591 | 16437 |
| 8 | 14049 | 27852 | 41811 | 55410 | 69567 | 83176 | 69746 | 55687 | 41321 | 27755 | 13635 |
| 9 | 11087 | 22237 | 33174 | 44641 | 55529 | 66772 | 55322 | 45003 | 33539 | 22187 | 11189 |
| 10 | 8421 | 16726 | 25020 | 33279 | 41518 | 50052 | 41500 | 33247 | 24828 | 16925 | 8408 |
| 11 | 5536 | 11048 | 16598 | 22405 | 27832 | 33270 | 27791 | 22381 | 16714 | 11187 | 5652 |
| 12 | 2774 | 5616 | 8361 | 11094 | 13712 | 16655 | 13982 | 11060 | 8225 | 5471 | 2763 |

Critical value (df=100, alpha=0.01): 44.31

Result: 102.403 Fail

Lag-1 autocorrelation of sums(0.001054) : 0.000502873

Permutation empirical p-value for chi-square: 0.406593

**Test Case 3: Three Dice Distribution and Independence**

* **Inputs:** Roll three dice 4.32 million times.
* **Expected Outputs:**
  + **Distribution:**
    - **Faces:** Each face (1, 2, 3, 4, 5, 6) is expected to appear approximately 2,160,000 times (4,320,000/6x3). The Chi-squared test will be used to test for a uniform distribution.
    - **Sums:** The distribution of the sum of the dice should follow a normal distribution. The Chi-squared test will be used to test for this distribution.
  + **Independence:**
    - **Horizontal:** The results of any die should be independent of the results of any other die for the same roll. A Chi-squared test using an orthogonal array strength two will be used to test for significant dependency.
    - **Vertical:** The outcome of a die from one roll should not influence the outcome of a die from the next roll. Chi-squared test of independence on the sequential rolls will be used to test for significant dependency.
      * **Sums:** The outcome of the sums of one roll should not influence the outcome of another roll. A Chi-squared test will be used to test for significant dependency.

**Actual Output:** Number of times each number was rolled:

1: 2160094

2: 2158960

3: 2162155

4: 2160564

5: 2158216

6: 2160011

Critical value (df=5, alpha=0.05): 11.07

Result: 4.27562 Pass

Sum distribution:

3: 19779

4: 59795

5: 119819

6: 200843

7: 299954

8: 420043

9: 500180

10: 539206

11: 540617

12: 500762

13: 420186

14: 299290

15: 199674

16: 119690

17: 59966

18: 20196

Critical value (df=15, alpha=0.05): 26.12

Result: 15.1137 Pass

Horizontal Transition Counts (for independence test):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 360242 | 358794 | 361078 | 361355 | 359401 | 359224 |
| 2 | 360188 | 360219 | 359725 | 359963 | 359578 | 359286 |
| 3 | 360076 | 360459 | 361430 | 359014 | 360095 | 361081 |
| 4 | 359571 | 360085 | 360326 | 360710 | 359378 | 360492 |
| 5 | 360248 | 358921 | 359881 | 359469 | 359850 | 359847 |
| 6 | 359768 | 360482 | 359715 | 360052 | 359914 | 360080 |

Critical value (df=25, alpha=0.05): 37.65

Result: 30.6422 Pass

Vertical Transition Counts (for independence test):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 359568 | 359393 | 360542 | 360434 | 360094 | 360063 |
| 2 | 359704 | 359672 | 359566 | 360167 | 360249 | 359602 |
| 3 | 360422 | 360217 | 360881 | 360391 | 360185 | 360059 |
| 4 | 360600 | 359947 | 360852 | 359433 | 358971 | 360761 |
| 5 | 360136 | 359903 | 360223 | 360221 | 358798 | 358935 |
| 6 | 359664 | 359828 | 360091 | 359918 | 359919 | 360591 |

Critical value (df=25, alpha=0.05): 37.65

Result: 16.8437 Pass

Sum Transition Counts (for independence test):

A table of numbers

AI-generated content may be incorrect.

Critical value (df=225, alpha=0.01): 95.02

Result: 251.392 Fail

Lag-1 autocorrelation of sums (0.000962): 0.000807697

Permutation empirical p-value for chi-square: 0.106893

**Test Case 4: Four Dice Distribution and Independence**

* **Inputs:** Roll four dice 6.48 million times.
* **Expected Outputs:**
  + **Distribution:**
    - **Faces:** Each face (1, 2, 3, 4, 5, 6) is expected to appear approximately 4,320,000 times (4,320,000/6x4). The Chi-squared test will be used to test for a uniform distribution.
    - **Sums:** The distribution of the sum of the dice should follow a normal distribution. The Chi-squared test will be used to test for this distribution.
  + **Independence:**
    - **Horizontal:** The results of any die should be independent of the results of any other die for the same roll. A Chi-squared test using an orthogonal array strength two will be used to test for significant dependency.
    - **Vertical:** The outcome of a die from one roll should not influence the outcome of a die from the next roll. Chi-squared test of independence on the sequential rolls will be used to test for significant dependency.
      * **Sums:** The outcome of the sums of one roll should not influence the outcome of another roll. A Chi-squared test will be used to test for significant dependency.

**Actual Output:** Number of times each number was rolled:

1: 4317882

2: 4318930

3: 4318962

4: 4321903

5: 4319011

6: 4323312

Critical value (df=5, alpha=0.05): 11.07 Result: 5.15675 Pass

Sum distribution:

4: 4917

5: 19915

6: 50034

7: 100427

8: 173936

9: 279840

10: 399552

11: 519796

12: 625218

13: 699774

14: 729813

15: 700201

16: 625068

17: 520795

18: 400502

19: 279326

20: 175369

21: 100341

22: 50081

23: 20104

24: 4991

Critical value (df=20, alpha=0.05): 33.92 Result: 17.0868 Pass

Horizontal Transition Counts (for independence test):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 718867 | 720567 | 719826 | 719593 | 719924 | 719105 |
| 2 | 719555 | 718250 | 719618 | 720304 | 720572 | 720630 |
| 3 | 720053 | 719798 | 718908 | 720682 | 720056 | 719465 |
| 4 | 719291 | 721414 | 720128 | 720711 | 719005 | 721354 |
| 5 | 719868 | 718355 | 719994 | 720450 | 718886 | 721457 |
| 6 | 720247 | 720546 | 720487 | 720163 | 720567 | 721300 |

Critical value (df=25, alpha=0.05): 37.65 Result: 22.4572 Pass

Vertical Transition Counts (for independence test):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 720350 | 718457 | 719174 | 720132 | 718731 | 721038 |
| 2 | 719686 | 719054 | 720623 | 720031 | 719585 | 719951 |
| 3 | 719858 | 719540 | 719410 | 719933 | 720425 | 719796 |
| 4 | 718765 | 720682 | 719253 | 720532 | 721108 | 721563 |
| 5 | 719610 | 721144 | 719455 | 721255 | 718317 | 719230 |
| 6 | 719613 | 720053 | 721047 | 720020 | 720845 | 721734 |

Critical value (df=25, alpha=0.05): 37.65 Result: 26.4712 Pass

Sum Transition Counts (for independence test):

A white background with many numbers

AI-generated content may be incorrect.

Critical value (df=400, alpha=0.01): 163.65 Result: 411.805 Fail

Lag-1 autocorrelation of sums (0.000786): 0.000347751

Permutation empirical p-value for chi-square: 0.311688

**Test Case 5: Five Dice Distribution and Independence**

* **Inputs:** Roll five dice 7.776 million times.
* **Expected Outputs:**
  + **Distribution:**
    - **Faces:** Each face (1, 2, 3, 4, 5, 6) is expected to appear approximately 6,480,000 times (7,776,000/6x5). The Chi-squared test will be used to test for a uniform distribution.
    - **Sums:** The distribution of the sum of the dice should follow a normal distribution. The Chi-squared test will be used to test for this distribution.
  + **Independence:**
    - **Horizontal:** The results of any die should be independent of the results of any other die for the same roll. A Chi-squared test using an orthogonal array strength two will be used to test for significant dependency.
    - **Vertical:** The outcome of a die from one roll should not influence the outcome of a die from the next roll. Chi-squared test of independence on the sequential rolls will be used to test for significant dependency.
      * **Sums:** The outcome of the sums of one roll should not influence the outcome of another roll. A Chi-squared test will be used to test for significant dependency.

**Actual Output:** Number of times each number was rolled:

1: 6479348

2: 6482753

3: 6478801

4: 6477983

5: 6481762

6: 6479353

Critical value (df=5, alpha=0.05): 11.07 Result: 2.62859 Pass

Sum distribution:

5: 1060

6: 4948

7: 14989

8: 34967

9: 70108

10: 126174

11: 204113

12: 305431

13: 420579

14: 538996

15: 651972

16: 734553

17: 779812

18: 779783

19: 735995

20: 651279

21: 540178

22: 419091

23: 305413

24: 204919

25: 125878

26: 69889

27: 34769

28: 15079

29: 5023

30: 1002

Critical value (df=25, alpha=0.05): 41.64 Result: 19.956 Pass

Horizontal Transition Counts (for independence test):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1080240 | 1081803 | 1078448 | 1078579 | 1080734 | 1079544 |
| 1 | 1079612 | 1081379 | 1080669 | 1080604 | 1080700 | 1079788 |
| 2 | 1078908 | 1079909 | 1080000 | 1080184 | 1079856 | 1079943 |
| 3 | 1080141 | 1080514 | 1079200 | 1078225 | 1079339 | 1080564 |
| 4 | 1080819 | 1079588 | 1080817 | 1080947 | 1079663 | 1079927 |
| 5 | 1079626 | 1079558 | 1079667 | 1079443 | 1081470 | 1079587 |

Critical value (df=25, alpha=0.05): 37.65 Result: 16.58 Pass

Vertical Transition Counts (for independence test):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1079536 | 1080939 | 1079502 | 1079700 | 1080223 | 1079448 |
| 1 | 1081272 | 1080165 | 1080010 | 1080631 | 1080569 | 1080106 |
| 2 | 1079029 | 1078921 | 1080147 | 1079881 | 1081074 | 1079749 |
| 3 | 1079594 | 1079334 | 1080370 | 1078699 | 1079849 | 1080137 |
| 4 | 1079310 | 1082512 | 1079394 | 1079616 | 1079635 | 1081295 |
| 5 | 1080607 | 1080882 | 1079378 | 1079456 | 1080412 | 1078618 |

Critical value (df=25, alpha=0.05): 37.65 Result: 16.0609 Pass

Sum Transition Counts (for independence test):

A white grid with many small colored lines

AI-generated content may be incorrect.

Critical value (df=625, alpha=0.01): 486.36 Result: 589.15 Fail

Lag-1 autocorrelation of sums (0.000717): 0.000374124

Permutation empirical p-value for chi-square: 1

Top 20 cell contributions 2-dice (contrib, row(prevSum), col(currSum), obs, exp, resid):

0: 10.1163 (6,10) obs=42301 exp=41651.9 resid=3.18062

1: 5.41729 (2,2) obs=2660 exp=2782.78 resid=-2.32751

2: 4.51423 (3,6) obs=28086 exp=27732.2 resid=2.12467

3: 3.71796 (9,9) obs=45003 exp=44595.8 resid=1.9282

4: 3.31543 (8,12) obs=13635 exp=13849.3 resid=-1.82083

5: 3.11501 (10,11) obs=16925 exp=16696.9 resid=1.76494

6: 2.9079 (6,7) obs=82780 exp=83272.1 resid=-1.70526

7: 2.80029 (7,8) obs=83765 exp=83282.1 resid=1.67341

8: 2.70808 (8,10) obs=41321 exp=41656.9 resid=-1.64563

9: 2.15829 (3,7) obs=32993 exp=33260.9 resid=-1.46911

10: 2.0304 (4,7) obs=50273 exp=49954.5 resid=1.42492

11: 1.9452 (9,8) obs=55322 exp=55651 resid=-1.3947

12: 1.84065 (4,8) obs=41379 exp=41655.9 resid=-1.3567

13: 1.83476 (11,12) obs=5652 exp=5551.08 resid=1.35453

14: 1.76707 (7,12) obs=16437 exp=16608.3 resid=-1.32931

15: 1.68953 (3,9) obs=22032 exp=22225.8 resid=-1.29982

16: 1.6694 (6,11) obs=28048 exp=27832.4 resid=1.29205

17: 1.59666 (6,2) obs=13751 exp=13900 resid=-1.26359

18: 1.58104 (4,10) obs=24788 exp=24986.8 resid=-1.25739

19: 1.56198 (8,2) obs=14049 exp=13901.6 resid=1.24979

Top 20 cell contributions 3-dice (contrib, row(prevSum), col(currSum), obs, exp, resid):

0: 8.2728 (17,8) obs=5611 exp=5830.63 resid=-2.87625

1: 7.31532 (14,13) obs=29572 exp=29110.5 resid=2.70469

2: 6.54191 (11,14) obs=37949 exp=37454 resid=2.55771

3: 6.16194 (11,9) obs=63215 exp=62594 resid=2.48233

4: 6.09483 (6,13) obs=19190 exp=19535.1 resid=-2.46877

5: 5.10757 (17,3) obs=312 exp=274.553 resid=2.25999

6: 5.06919 (8,12) obs=49187 exp=48690.2 resid=2.25149

7: 4.9008 (3,9) obs=2396 exp=2290.06 resid=2.21377

8: 4.79082 (14,15) obs=13576 exp=13833.4 resid=-2.18879

9: 4.70996 (13,3) obs=2019 exp=1923.81 resid=2.17024

10: 4.3133 (5,18) obs=511 exp=560.154 resid=-2.07685

11: 4.1703 (10,11) obs=68008 exp=67477.5 resid=2.04213

12: 3.75244 (11,18) obs=2430 exp=2527.39 resid=-1.93712

13: 3.65434 (8,11) obs=52127 exp=52565.3 resid=-1.91163

14: 3.5755 (6,18) obs=881 exp=938.941 resid=-1.8909

15: 3.53257 (10,4) obs=7301 exp=7463.37 resid=-1.87951

16: 3.46555 (15,11) obs=25282 exp=24987.7 resid=1.8616

17: 3.39216 (13,18) obs=2046 exp=1964.37 resid=1.84178

18: 3.36634 (11,11) obs=67177 exp=67654.2 resid=-1.83476

19: 2.94493 (11,10) obs=67032 exp=67477.8 resid=-1.71608

Top 20 cell contributions 4-dice (contrib, row(prevSum), col(currSum), obs, exp, resid):

0: 11.9537 (24,6) obs=60 exp=38.537 resid=3.45742

1: 9.60872 (21,17) obs=7786 exp=8064.37 resid=-3.09979

2: 8.07167 (5,6) obs=189 exp=153.77 resid=2.84107

3: 6.01545 (7,8) obs=2823 exp=2695.66 resid=2.45264

4: 5.66557 (22,17) obs=4176 exp=4024.99 resid=2.38025

5: 5.6287 (12,9) obs=27390 exp=27000.2 resid=2.37249

6: 5.56309 (21,15) obs=11088 exp=10842.4 resid=2.35862

7: 5.1403 (21,9) obs=4184 exp=4333.25 resid=-2.26722

8: 4.68639 (4,24) obs=8 exp=3.78715 resid=2.16481

9: 4.61414 (19,21) obs=4184 exp=4325.27 resid=-2.14806

10: 4.38124 (10,8) obs=10508 exp=10724.8 resid=-2.09314

11: 4.28164 (20,5) obs=587 exp=538.962 resid=2.06921

12: 4.24079 (18,4) obs=268 exp=303.899 resid=-2.05932

13: 4.04776 (8,11) obs=14190 exp=13952.4 resid=2.01191

14: 3.9422 (16,5) obs=1834 exp=1921.02 resid=-1.9855

15: 3.88399 (17,4) obs=356 exp=395.177 resid=-1.97079

16: 3.65921 (18,14) obs=45513 exp=45106.7 resid=1.91291

17: 3.62687 (13,13) obs=76092 exp=75568.5 resid=1.90443

18: 3.61059 (10,15) obs=42779 exp=43173.8 resid=-1.90015

19: 3.60752 (22,12) obs=4700 exp=4832.03 resid=-1.89935

Top 20 cell contributions 5-dice (contrib, row(prevSum), col(currSum), obs, exp, resid):

0: 17.7321 (6,30) obs=4 exp=0.63759 resid=4.21095

1: 8.47062 (28,30) obs=6 exp=1.94305 resid=2.91043

2: 6.55633 (11,25) obs=3157 exp=3304.18 resid=-2.56053

3: 6.54805 (6,25) obs=103 exp=80.0983 resid=2.55892

4: 5.80527 (19,8) obs=3171 exp=3309.61 resid=-2.40941

5: 5.80355 (19,24) obs=19731 exp=19395.5 resid=2.40906

6: 5.32826 (5,11) obs=40 exp=27.824 resid=2.3083

7: 4.80017 (29,8) obs=33 exp=22.5874 resid=2.19093

8: 4.76848 (25,8) obs=618 exp=566.046 resid=2.18368

9: 4.73637 (17,10) obs=12898 exp=12653.2 resid=2.17632

10: 4.5231 (6,12) obs=224 exp=194.351 resid=2.12676

11: 4.51899 (18,29) obs=456 exp=503.71 resid=-2.12579

12: 4.40846 (27,6) obs=32 exp=22.1241 resid=2.09963

13: 4.39791 (26,5) obs=16 exp=9.52705 resid=2.09712

14: 4.37845 (7,6) obs=16 exp=9.53775 resid=2.09248

15: 4.34856 (24,11) obs=5226 exp=5378.94 resid=-2.08532

16: 4.31511 (25,7) obs=275 exp=242.642 resid=2.07729

17: 4.31032 (18,16) obs=74225 exp=73661.5 resid=2.07613

18: 4.30525 (18,18) obs=77617 exp=78197.2 resid=-2.07491

19: 4.2845 (10,27) obs=515 exp=564.165 resid=-2.0699

// main.cpp

// This file simulates rolling dice, analyzes the results for fairness and independence, and prints detailed statistics.

#include <iostream>

#include <thread>

#include <chrono>

#include <vector>

#include <iomanip>

#include <random>

#include <algorithm>

#include <tuple>

#include <future>

#include <cmath>

#include "DiceRoller.h"

using namespace std;

double chiSquareDist(int *observed*[], int *totalRolls*) {

// Checks if each die face (1-6) appears about as often as expected (fairness test).

    double expected = *totalRolls* / 6.0;

    double chiSquared = 0.0;

    for(int i = 0; i < 6; ++i) {

        double diff = *observed*[i] - expected;

        chiSquared += (diff \* diff) / expected;

    }

    return chiSquared;

}

// Helper to count ways to get each sum for n dice

std::vector<int> diceSumWays(int *numDice*) {

// Calculates how many ways you can get each possible sum when rolling numDice dice.

    int minSum = *numDice*;

    int maxSum = *numDice* \* 6;

    std::vector<int> ways(maxSum + 1, 0);

    // Dynamic programming: ways[s][d] = ways to get sum s with d dice

    std::vector<std::vector<int>> dp(*numDice* + 1, std::vector<int>(maxSum + 1, 0));

    dp[0][0] = 1;

    for (int d = 1; d <= *numDice*; ++d) {

        for (int s = 0; s <= maxSum; ++s) {

            for (int face = 1; face <= 6; ++face) {

                if (s - face >= 0)

                    dp[d][s] += dp[d-1][s-face];

            }

        }

    }

    for (int s = minSum; s <= maxSum; ++s) {

        ways[s] = dp[*numDice*][s];

    }

    return ways;

}

// Chi-squared test for dice sums (non-uniform expected)

double chiSquareSum(const std::vector<int>& *observed*, int *numDice*, int *numRolls*) {

// Checks if the distribution of dice sums matches what you'd expect from random dice.

    int minSum = *numDice*;

    int maxSum = *numDice* \* 6;

    std::vector<double> expectedCounts(maxSum + 1, 0.0);

    std::vector<int> ways = diceSumWays(*numDice*);

    int totalWays = 0;

    for (int s = minSum; s <= maxSum; ++s) totalWays += ways[s];

    for (int s = minSum; s <= maxSum; ++s) {

        expectedCounts[s] = static\_cast<double>(ways[s]) / totalWays \* *numRolls*;

    }

    double chiSquared = 0.0;

    for (int s = minSum; s <= maxSum; ++s) {

        double expected = expectedCounts[s];

        double obs = *observed*[s];

        if (expected > 0) {

            double diff = obs - expected;

            chiSquared += (diff \* diff) / expected;

        }

    }

    return chiSquared;

}

double chiSquareInd(const vector<vector<int>>& *observed*, int *totalTransitions*) {

// Tests if the transitions between die faces (from one roll to the next) are independent.

    vector<int> rowTotals(6, 0);

    vector<int> colTotals(6, 0);

    int grandTotal = 0;

    for (int i = 0; i < 6; ++i) {

        for (int j = 0; j < 6; ++j) {

            rowTotals[i] += *observed*[i][j];

            colTotals[j] += *observed*[i][j];

            grandTotal += *observed*[i][j];

        }

    }

    double chiSquared = 0.0;

    for (int i = 0; i < 6; ++i) {

        for (int j = 0; j < 6; ++j) {

            double expected\_ij = (double)rowTotals[i] \* colTotals[j] / grandTotal;

            if (expected\_ij > 0) {

                double diff = *observed*[i][j] - expected\_ij;

                chiSquared += (diff \* diff) / expected\_ij;

            }

        }

    }

    return chiSquared;

}

// double chiSquareIndSum(const std::vector<std::vector<int>>& transMatrix) {

// // Tests if the transitions between sums (from one roll to the next) are independent.

//     int n = transMatrix.size();

//     std::vector<int> rowTotals(n, 0);

//     std::vector<int> colTotals(n, 0);

//     int grandTotal = 0;

//     // Calculate row, column, and grand totals

//     for (int i = 0; i < n; ++i) {

//         for (int j = 0; j < n; ++j) {

//             rowTotals[i] += transMatrix[i][j];

//             colTotals[j] += transMatrix[i][j];

//             grandTotal += transMatrix[i][j];

//         }

//     }

//     double chiSquared = 0.0;

//     for (int i = 0; i < n; ++i) {

//         for (int j = 0; j < n; ++j) {

//             double expected = (double)rowTotals[i] \* colTotals[j] / grandTotal;

//             if (expected > 0) {

//                 double diff = transMatrix[i][j] - expected;

//                 chiSquared += (diff \* diff) / expected;

//             }

//         }

//     }

//     return chiSquared;

// }

// return contributors list: tuple(contribution, i, j, observed, expected, residual)

std::vector<std::tuple<double,int,int,int,double,double>>

chiSquareIndSumWithDiagnostics(const std::vector<std::vector<int>>& *transMatrix*,

                               double &*chiOut*) {

// Like chiSquareIndSum, but also records which transitions contribute most to the test statistic.

    int n = (int)*transMatrix*.size();

    std::vector<int> rowTotals(n, 0);

    std::vector<int> colTotals(n, 0);

    int grandTotal = 0;

    for (int i = 0; i < n; ++i)

        for (int j = 0; j < n; ++j) {

            rowTotals[i] += *transMatrix*[i][j];

            colTotals[j] += *transMatrix*[i][j];

            grandTotal += *transMatrix*[i][j];

        }

*chiOut* = 0.0;

    std::vector<std::tuple<double,int,int,int,double,double>> contribs;

    contribs.reserve(static\_cast<size\_t>(n) \* static\_cast<size\_t>(n));

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            double expected = 0.0;

            if (grandTotal > 0)

                expected = (double)rowTotals[i] \* (double)colTotals[j] / (double)grandTotal;

            if (expected > 0.0) {

                double diff = *transMatrix*[i][j] - expected;

                double c = (diff\*diff) / expected;

                double resid = diff / sqrt(expected);

*chiOut* += c;

                contribs.emplace\_back(c, i, j, *transMatrix*[i][j], expected, resid);

            }

        }

    }

    // sort contributions descending

    std::sort(contribs.begin(), contribs.end(),

              [](auto &*a*, auto &*b*){ return std::get<0>(*a*) > std::get<0>(*b*); });

    return contribs;

}

// compute lag-1 autocorrelation of sums vector

double lag1\_autocorr(const std::vector<int>& *sums*) {

// Measures how much each sum is correlated with the previous sum (lag-1 autocorrelation).

    if (*sums*.size() < 2) return 0.0;

    double n = (double)*sums*.size();

    double mean = 0.0;

    for (int v : *sums*) mean += v;

    mean /= n;

    double num = 0.0, den = 0.0;

    for (size\_t i = 0; i+1 < *sums*.size(); ++i)

        num += (*sums*[i] - mean) \* (*sums*[i+1] - mean);

    for (size\_t i = 0; i < *sums*.size(); ++i)

        den += (*sums*[i] - mean) \* (*sums*[i] - mean);

    if (den == 0.0) return 0.0;

    return num / den;

}

// This version was a test with minor changes to reduce memory usage and improve speed

// permutation test: permute prior sums and recompute chi-square many times

// double permutation\_pvalue\_chi\_parallel2(const std::vector<int>& prevSums, const std::vector<int>& currSums, int maxSum, int nperms, double observedChi, mt19937\_64& rng, int nthreads=0) {

//     // Shuffles the previous sums many times to see how often you get a chi-squared value as extreme as observed (permutation test for independence).

//     // build original contingency as counts

//     const size\_t N = prevSums.size();

//     if (N == 0 || nperms <= 0) return 1.0;

//     // Determine actual sum range [base..maxVal]; base should equal numDice in your setup

//     int base = std::numeric\_limits<int>::max();

//     int maxVal = std::numeric\_limits<int>::min();

//     for (int v : prevSums) { base = std::min(base, v); maxVal = std::max(maxVal, v); }

//     for (int v : currSums) { base = std::min(base, v); maxVal = std::max(maxVal, v); }

//     // Fall back to provided maxSum if needed

//     maxVal = std::max(maxVal, maxSum);

//     int S = maxVal - base + 1;

//     // Precompute row/col totals once (constant across permutations)

//     std::vector<int> rowCounts(S, 0), colCounts(S, 0);

//     for (int v : prevSums) rowCounts[v - base]++;

//     for (int v : currSums) colCounts[v - base]++;

//     // Decide thread count and partition work

//     unsigned hw = std::thread::hardware\_concurrency();

//     int T = nthreads > 0 ? nthreads : (hw ? (int)hw : 2);

//     T = std::max(1, std::min(T, nperms));

//     std::vector<int> chunk(T, nperms / T);

//     for (int r = 0; r < nperms % T; ++r) ++chunk[r];

//     // Derive per-thread seeds from the provided rng (single-threaded draw)

//     std::vector<uint64\_t> seeds(T);

//     for (int t = 0; t < T; ++t) seeds[t] = rng();

//     std::vector<int> ge\_counts(T, 0);

//     std::vector<std::thread> threads;

//     threads.reserve(T);

//     for (int ti = 0; ti < T; ++ti) {

//         threads.emplace\_back([&, ti]{

//             std::mt19937\_64 trng(seeds[ti]);

//             // Local working buffers

//             std::vector<int> prior = prevSums;

//             std::vector<std::vector<int>> Tp(S, std::vector<int>(S, 0));

//             int ge = 0;

//             for (int p = 0; p < chunk[ti]; ++p) {

//                 // Reset

//                 for (int i = 0; i < S; ++i) {

//                     std::fill(Tp[i].begin(), Tp[i].end(), 0);

//                 }

//                 // Permute and build contingency in compact index space

//                 std::shuffle(prior.begin(), prior.end(), trng);

//                 for (size\_t k = 0; k < N; ++k) {

//                     int r = prior[k] - base;

//                     int c = currSums[k] - base;

//                     Tp[r][c]++;

//                 }

//                 // Compute chi using precomputed marginals

//                 double chi = 0.0;

//                 for (int i = 0; i < S; ++i) {

//                     for (int j = 0; j < S; ++j) {

//                         double exp = (N > 0) ? (double)rowCounts[i] \* colCounts[j] / (double)N : 0.0;

//                         if (exp > 0.0) {

//                             double d = Tp[i][j] - exp;

//                             chi += (d \* d) / exp;

//                         }

//                     }

//                 }

//                 if (chi >= observedChi) ++ge;

//             }

//             ge\_counts[ti] = ge;

//         });

//     }

//     for (auto& th : threads) th.join();

//     long long ge\_total = 0;

//     for (int v : ge\_counts) ge\_total += v;

//     return (double)(ge\_total + 1) / (double)(nperms + 1); // add-one correction

// }

// permutation test: permute prior sums and recompute chi-square many times

double permutation\_pvalue\_chi\_parallel(const std::vector<int>& *prevSums*, const std::vector<int>& *currSums*, int *maxSum*, int *nperms*, double *observedChi*, mt19937\_64& *rng*, int *nthreads*=0) {

    // Shuffles the previous sums many times to see how often you get a chi-squared value as extreme as observed (permutation test for independence).

    // build original contingency as counts

    const size\_t N = *prevSums*.size();

    if (N == 0 || *nperms* <= 0) return 1.0;

    // Decide thread count

    unsigned hw = std::thread::hardware\_concurrency();

    int T = *nthreads* > 0 ? *nthreads* : (hw ? (int)hw : 2);

    T = std::max(1, std::min(T, *nperms*));

    // Derive per-thread seeds from the provided rng (single-threaded) to avoid sharing rng

    std::vector<uint64\_t> seeds(T);

    for (int t = 0; t < T; ++t) {

        seeds[t] = *rng*();

    }

    // Partition work across threads

    std::vector<int> chunk(T, *nperms* / T);

    for (int r = 0; r < *nperms* % T; ++r) ++chunk[r];

    std::vector<int> ge\_counts(T, 0);

    std::vector<std::thread> threads;

    threads.reserve(T);

    for (int ti = 0; ti < T; ++ti) {

        threads.emplace\_back([&, *ti*]{

            std::mt19937\_64 trng(seeds[ti]);

            // Local working buffers to avoid contention

            std::vector<int> prior = *prevSums*;

            std::vector<std::vector<int>> Tp(*maxSum* + 1, std::vector<int>(*maxSum* + 1, 0));

            std::vector<int> rtot(*maxSum* + 1, 0), ctot(*maxSum* + 1, 0);

            int ge = 0;

            for (int p = 0; p < chunk[ti]; ++p) {

                // Reset

                for (int i = 0; i <= *maxSum*; ++i) {

                    std::fill(Tp[i].begin(), Tp[i].end(), 0);

                }

                std::fill(rtot.begin(), rtot.end(), 0);

                std::fill(ctot.begin(), ctot.end(), 0);

                // Permute and build contingency

                std::shuffle(prior.begin(), prior.end(), trng);

                for (size\_t k = 0; k < N; ++k) Tp[ prior[k] ][ *currSums*[k] ]++;

                // Compute chi-square

                double chi = 0.0;

                int grand = 0;

                for (int i = 0; i <= *maxSum*; ++i) for (int j = 0; j <= *maxSum*; ++j) {

                    rtot[i] += Tp[i][j];

                    ctot[j] += Tp[i][j];

                    grand += Tp[i][j];

                }

                for (int i = 0; i <= *maxSum*; ++i) for (int j = 0; j <= *maxSum*; ++j) {

                    double exp = (grand > 0) ? (double)rtot[i] \* ctot[j] / (double)grand : 0.0;

                    if (exp > 0.0) {

                        double d = Tp[i][j] - exp;

                        chi += (d \* d) / exp;

                    }

                }

                if (chi >= *observedChi*) ++ge;

            }

            ge\_counts[ti] = ge;

        });

    }

    for (auto& th : threads) th.join();

    long long ge\_total = 0;

    for (int v : ge\_counts) ge\_total += v;

    return (double)(ge\_total + 1) / (double)(*nperms* + 1); // add-one correction

}

// Old slow single-threaded version

// double permutation\_pvalue\_chi(const std::vector<int>& prevSums, const std::vector<int>& currSums, int maxSum, int nperms, double observedChi, mt19937\_64& rng) {

//     std::vector<std::vector<int>> T(maxSum+1, std::vector<int>(maxSum+1,0));

//     size\_t N = prevSums.size();

//     for (size\_t k = 0; k < N; ++k) {

//         T[prevSums[k]][currSums[k]]++;

//     }

//     int ge = 0;

//     // make a vector of prior-sum values to shuffle

//     std::vector<int> prior = prevSums;

//     std::vector<std::vector<int>> Tp(maxSum+1, std::vector<int>(maxSum+1,0));

//     std::vector<int> rtot(maxSum+1,0), ctot(maxSum+1,0);

//     for (int p = 0; p < nperms; ++p) {

//         for (int i=0;i<=maxSum;++i) { std::fill(Tp[i].begin(), Tp[i].end(), 0); }

//         std::fill(rtot.begin(), rtot.end(), 0);

//         std::fill(ctot.begin(), ctot.end(), 0);

//         std::shuffle(prior.begin(), prior.end(), rng);

//         for (size\_t k = 0; k < N; ++k) Tp[ prior[k] ][ currSums[k] ]++;

//         double chi = 0.0;

//         int grand=0;

//         for (int i=0;i<=maxSum;++i) for (int j=0;j<=maxSum;++j) {

//             rtot[i]+=Tp[i][j];

//             ctot[j]+=Tp[i][j];

//             grand+=Tp[i][j];

//         }

//         for (int i=0;i<=maxSum;++i) for (int j=0;j<=maxSum;++j) {

//             double exp = (grand>0) ? (double)rtot[i]\*ctot[j]/(double)grand : 0.0;

//             if (exp>0.0) {

//                 double d = Tp[i][j]-exp;

//                 chi += (d\*d)/exp;

//             }

//         }

//         if (chi >= observedChi) ++ge;

//     }

//     return (double)(ge+1) / (double)(nperms+1); // add-one correction

// }

// Old slow single-threaded version with changes to reduce memory usage and improve speed

// double permutation\_pvalue\_chi(const std::vector<int>& prevSums, const std::vector<int>& currSums, int maxSum, int nperms, double observedChi, mt19937\_64& rng) {

//     // Determine actual sum range [base..maxVal]; base should equal numDice in your setup

//     int base = std::numeric\_limits<int>::max();

//     int maxVal = std::numeric\_limits<int>::min();

//     for (int v : prevSums) { base = std::min(base, v); maxVal = std::max(maxVal, v); }

//     for (int v : currSums) { base = std::min(base, v); maxVal = std::max(maxVal, v); }

//     // Fall back to provided maxSum if needed

//     maxVal = std::max(maxVal, maxSum);

//     int S = maxVal - base + 1;

//     // Precompute row/col totals once (constant across permutations)

//     std::vector<int> rowCounts(S, 0), colCounts(S, 0);

//     for (int v : prevSums) rowCounts[v - base]++;

//     for (int v : currSums) colCounts[v - base]++;

//     // Local buffers

//     std::vector<int> prior = prevSums;

//     std::vector<std::vector<int>> Tp(S, std::vector<int>(S, 0));

//     int ge = 0;

//     for (int p = 0; p < nperms; ++p) {

//         // Reset

//         for (int i = 0; i < S; ++i) {

//             std::fill(Tp[i].begin(), Tp[i].end(), 0);

//         }

//         std::shuffle(prior.begin(), prior.end(), rng);

//         // Build contingency in compact index space

//         for (size\_t k = 0; k < N; ++k) {

//             int r = prior[k] - base;

//             int c = currSums[k] - base;

//             Tp[r][c]++;

//         }

//         // Compute chi using precomputed marginals (expected is constant per cell)

//         double chi = 0.0;

//         for (int i = 0; i < S; ++i) {

//             for (int j = 0; j < S; ++j) {

//                 double exp = (N > 0) ? (double)rowCounts[i] \* colCounts[j] / (double)N : 0.0;

//                 if (exp > 0.0) {

//                     double d = Tp[i][j] - exp;

//                     chi += (d \* d) / exp;

//                 }

//             }

//         }

//         if (chi >= observedChi) ++ge;

//     }

//     return (double)(ge + 1) / (double)(nperms + 1); // add-one correction

// }

struct RollTestResults {

// Holds all the results and statistics from a dice rolling experiment.

    std::vector<int> rollCount;

    double chiDist;

    std::vector<int> rollSum;

    double chiSums;

    std::vector<std::vector<int>> horizTrans;

    double chiHorInd;

    std::vector<std::vector<int>> vertTrans;

    double chiVertInd;

    std::vector<std::vector<int>> sumTrans;

    std::vector<int> prevSumsList;

    std::vector<int> currSumsList;

    double chiObserved;

    std::vector<std::tuple<double,int,int,int,double,double>> topContribs;

    double lag1Autocorr;

    double permPValue;

};

RollTestResults rollTest(int *numRolls*, int *numDice*){

// Simulates rolling numDice dice numRolls times, collects statistics, and runs all tests in parallel threads.

    using clock = std::chrono::high\_resolution\_clock;

    DiceRoller roller(*numDice*);

    std::vector<int> rollCount(6, 0);

    int minSum = *numDice*;

    int maxSum = *numDice* \* 6;

    std::vector<int> rollSum(maxSum + 1, 0);

    std::vector<std::vector<int>> sumTrans(maxSum + 1, std::vector<int>(maxSum + 1, 0));

    std::vector<std::vector<int>> vertTrans(6, std::vector<int>(6, 0));

    std::vector<std::vector<int>> horizTrans(6, std::vector<int>(6, 0));

    std::vector<int> prevRoll(*numDice*, 0);

    //std::vector<std::vector<int>> results;

    std::vector<int> prevSumsList;

    std::vector<int> currSumsList;

    prevSumsList.reserve(std::max(0, *numRolls*-1));

    currSumsList.reserve(std::max(0, *numRolls*-1));

    for(int i = 0; i < *numRolls*; ++i) {

        vector<int> roll = roller.rollDice();

        //results.push\_back(roll);

        int sum = 0;

        int prevSum = 0;

        // Sequential transitions within a roll

        for(int j = 0; j < *numDice*; ++j) {

            rollCount[roll[j] - 1]++;

            sum += roll[j];

            // Compare die j to die (j+1)%numDice

            int nextIdx = (j + 1) % *numDice*;

            int curr = roll[j];

            int next = roll[nextIdx];

            vertTrans[curr - 1][next - 1]++;

            if(i > 0) {

                int prev = prevRoll[j];

                int curr = roll[j];

                horizTrans[prev - 1][curr - 1]++;

                prevSum += prevRoll[j];

            }

        }

        if (sum >= minSum && sum <= maxSum) {

            rollSum[sum]++;

        }

        // Track transitions between sums

        if (i > 0 && prevSum >= minSum && prevSum <= maxSum && sum >= minSum && sum <= maxSum) {

            sumTrans[prevSum][sum]++;

            prevSumsList.push\_back(prevSum);

            currSumsList.push\_back(sum);

        }

        prevRoll = std::move(roll);

    }

    double chiDist = 0.0;

    double chiSums = 0.0;

    double chiHorInd = 0.0;

    double chiVertInd = 0.0;

    double chiObserved = 0.0;

    double lag1Autocorr = 0.0;

    double permPValue = 0.0;

    std::vector<std::tuple<double,int,int,int,double,double>> topContribs;

    // chiObserved = chiSquareIndSum(sumTrans);  Old version that didnt have diagnostics for my failing sum transitions

    topContribs = chiSquareIndSumWithDiagnostics(sumTrans, chiObserved);

    topContribs.resize(std::min(20, (int)topContribs.size()));

    std::thread t1([&]{ permPValue = permutation\_pvalue\_chi\_parallel(prevSumsList, currSumsList, maxSum, 1000, chiObserved, roller.getEngine()); });

    lag1Autocorr = lag1\_autocorr(currSumsList);

    chiDist = chiSquareDist(rollCount.data(), *numRolls* \* *numDice*);

    chiHorInd = chiSquareInd(horizTrans, *numRolls* - 1);

    chiVertInd = chiSquareInd(vertTrans, *numRolls* - 1);

    chiSums = chiSquareSum(rollSum, *numDice*, *numRolls*);

    t1.join();

    return RollTestResults{

        rollCount,

        chiDist,

        rollSum,

        chiSums,

        horizTrans,

        chiHorInd,

        vertTrans,

        chiVertInd,

        sumTrans,

        prevSumsList,

        currSumsList,

        chiObserved,

        topContribs,

        lag1Autocorr,

        permPValue

    };

}

void printMatrix(const std::vector<std::vector<int>>& *matrix*, int *minIdx*, int *maxIdx*) {

// Nicely prints a matrix (like transition counts) for inspection.

    cout << "        ";

    for (int j = *minIdx*; j <= *maxIdx*; ++j) cout << setw(7) << j;

    cout << endl;

    for (int i = *minIdx*; i <= *maxIdx*; ++i) {

        cout << setw(7) << i << " |";

        for (int j = *minIdx*; j <= *maxIdx*; ++j) {

            cout << setw(7) << *matrix*[i][j];

        }

        cout << endl;

    }

}

void printRollTestResults(const RollTestResults& *results*, int *numDice*) {

// Prints all the results and statistics for a dice rolling experiment, including pass/fail for each test.

    const double chiCritSumsArr[4] = {18.31, 26.12, 33.92, 41.64}; // 2-5 dice

    const double chiCritSumIndArr[4] = {44.31, 95.02, 163.65, 486.36}; // 2-5 dice, alpha=0.01

    cout << "\nResults for " << *numDice* << " dice:" << endl;

    // Uniformity distribution

    cout << "Number of times each number was rolled:\n";

    for (size\_t i = 0; i < *results*.rollCount.size(); ++i) {

        cout << (i + 1) << ": " << *results*.rollCount[i] << endl;

    }

    cout << "\nCritical value (df=5, alpha=0.05): 11.07\n";

    if (*results*.chiDist > 11.07)

        cout << "\nResult: " << *results*.chiDist << " Fail\n";

    else

        cout << "\nResult: " << *results*.chiDist << " Pass\n";

    // Sums distribution

    cout << "\nSum distribution:\n";

    int minSum = *numDice*;

    int maxSum = *numDice* \* 6;

    for (int s = minSum; s <= maxSum; ++s) {

        cout << s << ": " << *results*.rollSum[s] << endl;

    }

    cout << "\nCritical value (df=" << (maxSum-minSum) << ", alpha=0.05): ";

    if (*numDice* >= 2 && *numDice* <= 5) {

        cout << chiCritSumsArr[*numDice*-2] << "\n";

        if (*results*.chiSums > chiCritSumsArr[*numDice*-2])

            cout << "\nResult: " << *results*.chiSums << " Fail\n";

        else

            cout << "\nResult: " << *results*.chiSums << " Pass\n";

    } else {

        cout << "[see chi-squared table]\n";

        cout << "\nResult: " << *results*.chiSums << " [no pass/fail]\n";

    }

    // Horizontal independence

    cout << "\nHorizontal Transition Counts (for independence test):\n";

    printMatrix(*results*.horizTrans, 0, 5);

    cout << "\nCritical value (df=25, alpha=0.05): 37.65" << endl;

    if (*results*.chiHorInd > 37.65)

        cout << "\nResult: " << *results*.chiHorInd << " Fail\n";

    else

        cout << "\nResult: " << *results*.chiHorInd << " Pass\n";

    // Vertical independence

    cout << "\nVertical Transition Counts (for independence test):\n";

    printMatrix(*results*.vertTrans, 0, 5);

    cout << "\nCritical value (df=25, alpha=0.05): 37.65" << endl;

    if (*results*.chiVertInd > 37.65)

        cout << "\nResult: " << *results*.chiVertInd << " Fail\n";

    else

        cout << "\nResult: " << *results*.chiVertInd << " Pass\n";

    // Sum independence

    cout << "\nSum Transition Counts (for independence test):\n";

    printMatrix(*results*.sumTrans, minSum, maxSum);

    // This function is modified to use the dianostics version

    cout << "\nCritical value (df=" << (maxSum-minSum)\*(maxSum-minSum) << ", alpha=0.01): ";

    if (*numDice* >= 2 && *numDice* <= 5) {

        cout << chiCritSumIndArr[*numDice*-2] << "\n";

        if (*results*.chiObserved > chiCritSumIndArr[*numDice*-2])

            cout << "\nResult: " << *results*.chiObserved << " Fail\n";

        else

            cout << "\nResult: " << *results*.chiObserved << " Pass\n";

    } else {

        cout << "[see chi-squared table]\n";

        cout << "\nResult: " << *results*.chiObserved << " [no pass/fail]\n";

    }

    cout << "Lag-1 autocorrelation of sums: " << *results*.lag1Autocorr << endl;

    cout << "Permutation empirical p-value for chi-square: " << *results*.permPValue << endl;

    cout << "\nTop " << *results*.topContribs.size() << " cell contributions (contrib, row(prevSum), col(currSum), obs, exp, resid):\n";

    for (int i = 0; i < *results*.topContribs.size(); ++i) {

        auto [c, r, col, obs, exp, resid] = *results*.topContribs[i];

        cout << i << ": " << c << "  (" << r << "," << col << ")  obs=" << obs << " exp=" << exp << " resid=" << resid << "\n";

    }

}

int main() {

    // Thread pool using std::async for parallel rollTest

    auto fut5 = std::async(std::launch::async, rollTest, 7776000, 5);

    auto fut4 = std::async(std::launch::async, rollTest, 6480000, 4);

    auto fut3 = std::async(std::launch::async, rollTest, 4320000, 3);

    auto fut2 = std::async(std::launch::async, rollTest, 3600000, 2);

    constexpr int numRolls = 3600000;

    int rollCount[6] = {0};

    vector<vector<int>> transCount(6, vector<int>(6, 0));

    DiceRoller roller(1);

    int prev = 0;

    // Batch random generation for single die

    for(int i = 0; i < numRolls; ++i) {

        int val = roller.rollOne();

        ++rollCount[val - 1];

        if(i > 0) {

            int curr = val;

            transCount[prev - 1][curr - 1]++;

        }

        prev = val;

    }

    double chiDist = 0.0;

    double chiInd = 0.0;

    auto futDist = std::async(std::launch::async, chiSquareDist, rollCount, numRolls);

    auto futInd = std::async(std::launch::async, chiSquareInd, transCount, numRolls - 1);

    cout << "Number of times each number was rolled:\n";

    for (int i = 0; i < 6; ++i) {

        cout << (i + 1) << ": " << rollCount[i] << '\n';

    }

    chiDist = futDist.get();

    cout << "\nCritical value (df=5, alpha=0.05): 11.07\n";

    if (\_\_builtin\_expect(chiDist > 11.07, 0)) // [[unlikely]]

        cout << "\nResult: " << chiDist << " Fail\n" << endl;

    else

        cout << "\nResult: " << chiDist << " Pass\n" << endl;

    printMatrix(transCount, 0, 5);

    chiInd = futInd.get();

    cout << "\nCritical value (df=25, alpha=0.05): 37.65" << endl;

    if (\_\_builtin\_expect(chiInd > 37.65, 0)) // [[unlikely]]

        cout << "\nResult: " << chiInd << " Fail\n";

    else

        cout << "\nResult: " << chiInd << " Pass\n";

    // Wait for all dice results

    printRollTestResults(fut2.get(), 2);

    printRollTestResults(fut3.get(), 3);

    printRollTestResults(fut4.get(), 4);

    printRollTestResults(fut5.get(), 5);

    return 0;

}

#pragma once

#include <cstdint>

#include <random>

#include <chrono>

#include <array>

// xoshiro256pp code retained for future use:

class xoshiro256pp {

public:

    using result\_type = uint64\_t;

    std::array<uint64\_t, 4> s;

    explicit xoshiro256pp(const uint64\_t *seed1*, const uint64\_t *seed2*, const uint64\_t *seed3*, const uint64\_t *seed4*) noexcept

        : s{*seed1*, *seed2*, *seed3*, *seed4*} {}

    static constexpr uint64\_t rotl(const uint64\_t *x*, const int *k*) noexcept {

        return (*x* << *k*) | (*x* >> (64 - *k*));

    }

    inline uint64\_t operator()() noexcept {

        const uint64\_t result = rotl(s[0] + s[3], 23) + s[0];

        const uint64\_t t = s[1] << 17;

        s[2] ^= s[0];

        s[3] ^= s[1];

        s[1] ^= s[2];

        s[0] ^= s[3];

        s[2] ^= t;

        s[3] = rotl(s[3], 45);

        return result;

    }

    static constexpr uint64\_t min() noexcept { return 0; }

    static constexpr uint64\_t max() noexcept { return UINT64\_MAX; }

};

class DiceRoller {

public:

    DiceRoller(int *numDice*) noexcept;

    inline std::vector<int> rollDice() noexcept {

        std::vector<int> results;

        results.reserve(engines.size());

        for (auto& eng : engines) {

            results.push\_back(dist(eng));

        }

        return results;

    }

    inline int rollOne() noexcept {

        return dist(engines[0]);

    }

    // inline std::vector<int> rollDice(const int numDice) noexcept {

    //     static thread\_local std::mt19937\_64 rng(std::random\_device{}());

    //     std::vector<int> results;

    //     results.reserve(numDice);

    //     for (int i = 0; i < numDice; ++i) {

    //         results.push\_back(dist(rng));

    //     }

    //     return results;

    // }

    std::mt19937\_64& getEngine(int *idx* = 0) noexcept { return engines[*idx*]; }

    static thread\_local std::uniform\_int\_distribution<int> dist;

private:

    std::vector<std::mt19937\_64> engines;

};

// DiceRoller.cpp

// This file provides random dice rolling using multiple sources of randomness for extra unpredictability.

#include <thread>

#include "DiceRoller.h"

using namespace std;

DiceRoller::DiceRoller(int *numDice*) noexcept {

    engines.reserve(*numDice*);

    // auto hash64 = [](uint64\_t x) {

    //     x ^= (x >> 33);

    //     x \*= 0xff51afd7ed558ccdULL;

    //     x ^= (x >> 33);

    //     x \*= 0xc4ceb9fe1a85ec53ULL;

    //     x ^= (x >> 33);

    //     return x;

    // };

    for (int i = 0; i < *numDice*; ++i) {

        std::random\_device rd;

        uint64\_t seed = ((uint64\_t)rd()) << 32 | rd();

        engines.emplace\_back(std::mt19937\_64(seed));

        // uint64\_t rd\_seed = ((uint64\_t)rd()) << 32 | rd();

        // uint64\_t time = std::chrono::duration\_cast<std::chrono::milliseconds>(

        //     std::chrono::system\_clock::now().time\_since\_epoch()).count();

        // uint64\_t mixed = time ^ rd\_seed ^ (i \* 0x9e3779b97f4a7c15ULL);

        // uint64\_t h1 = hash64(mixed);

        // uint64\_t h2 = hash64(mixed + i);

        // uint64\_t h3 = hash64(mixed ^ (i << 16));

        // uint64\_t h4 = hash64(mixed \* (i+1));

        // engines.emplace\_back(h1, h2, h3, h4);

        // std::this\_thread::sleep\_for(std::chrono::microseconds(1100));

    }

}

thread\_local std::uniform\_int\_distribution<int> DiceRoller::dist(1, 6);