

ZJU Computational Physics: Homework #4

Due on Monday, December 13, 2024

Github: <https://github.com/NAKONAKO4/ZJU-computational-physics-NAKO>

管一泮

Problem 1

Random Number Generator

选取前十组数据进行随机数生成，采用线性同余生成器，数学公式表示为 $X_{n+1} = (a \cdot X_n + c) \bmod m$ ，生成 2500 个随机数。测试方法采用四种，分别为：分布均匀性测试、自相关性测试、Kolmogorov-Smirnov 测试、频谱分析（FFT 后观察各个频率分布是否均匀）

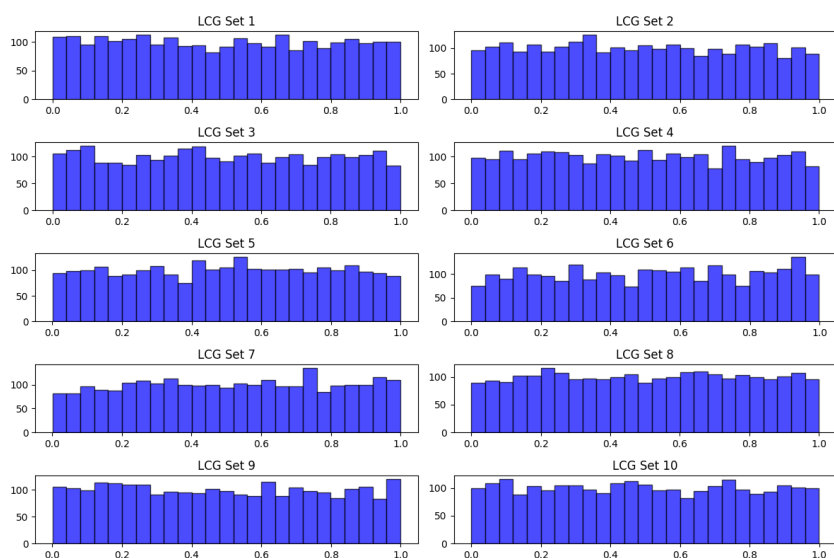


图 1: 在 10 组参数下生成的随机数分布的均匀性

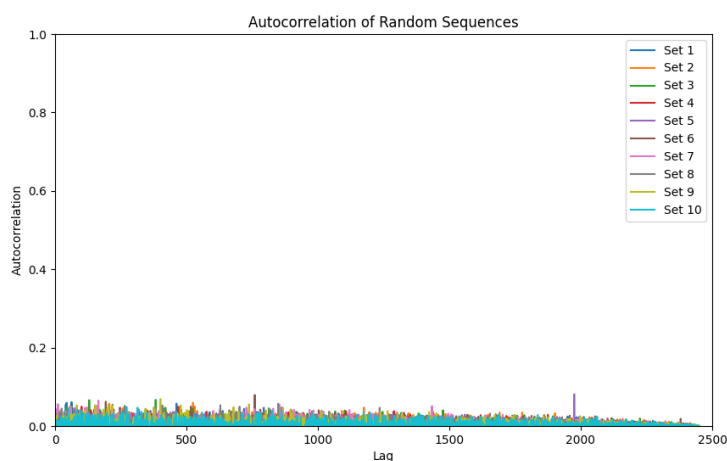


图 2: 在 10 组参数下生成的随机数分布的自相关性

Kolmogorov-Smirnov Test Results:

Set 1: KS Statistic = 0.02053, p-value = 0.23940
 Set 2: KS Statistic = 0.02132, p-value = 0.20305
 Set 3: KS Statistic = 0.01532, p-value = 0.59502
 Set 4: KS Statistic = 0.01214, p-value = 0.85017
 Set 5: KS Statistic = 0.02075, p-value = 0.22859
 Set 6: KS Statistic = 0.02566, p-value = 0.07304
 Set 7: KS Statistic = 0.02717, p-value = 0.04902
 Set 8: KS Statistic = 0.01238, p-value = 0.83373
 Set 9: KS Statistic = 0.02206, p-value = 0.17299
 Set 10: KS Statistic = 0.01439, p-value = 0.67335

图 3: 在 10 组参数下生成的随机数分布的 Kolmogorov-Smirnov 测试

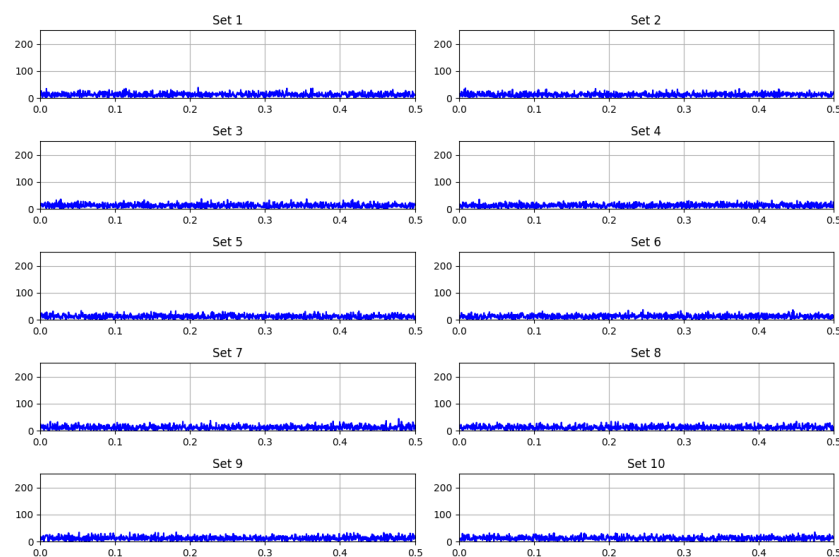
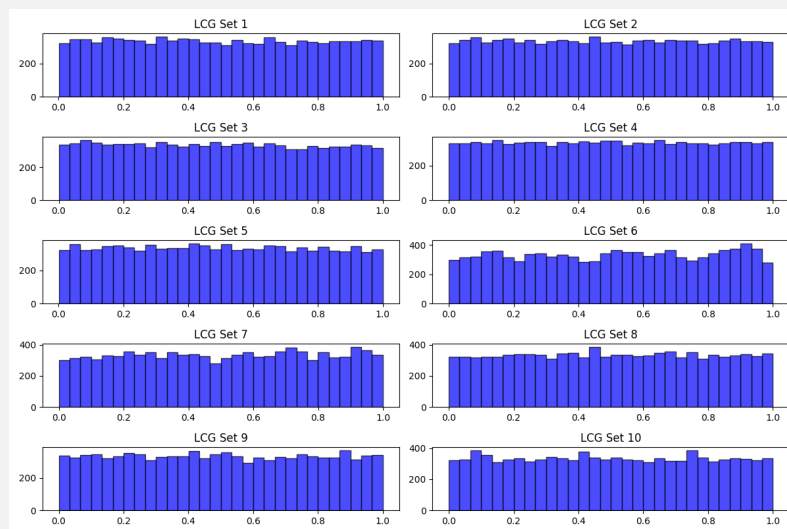


图 4: 在 10 组参数下生成的随机数分布的频谱分析

可以看到，分布均匀性较好，自相关性基本为 0，Kolmogorov-Smirnov 测试的 stat 较小且 p-value 较大，说明随机数序列接近均匀分布，频谱分析可以看到十组数据在各个频率上分布接近均匀。事实上 2500 个随机数相对并不多，但由于在较多随机数生成时存在绘图问题，只进行了生成 10000 个随机数时的分布均匀性测试，得到结果为：



可以看到随着生成随机数越多，随机数生成器的均匀性能体现的越好，这说明随机数生成器的性能是好的，能够生成较随机的伪随机数。

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def lcg(m, a, c, seed, n):
5     x = seed
6     numbers = []
7     for _ in range(n):
8         x = (a * x + c) % m
9         numbers.append(x / m)
10    return numbers
11
12 params = [
13     (214326, 1807, 45289),
14     (244944, 1597, 51749),
15     (233280, 1861, 49297),
16     (175000, 2661, 26979),
17     (121500, 4081, 25673),
18     (145800, 3661, 30809),
19     (139968, 3877, 29573),
20     (214326, 3613, 45289),
21     (714025, 1366, 150889),
22     (134456, 8121, 28411),
23 ]
24
25 seed = 42
26 num_samples = 2500
27
28 random_sequences = []

```

```

29
30 for m, a, c in params:
31     random_sequences.append(lcg(m, a, c, seed, num_samples))
32
33 def plot_histograms(sequences, n_cols=2):
34     n_rows = len(sequences) // n_cols + (len(sequences) % n_cols > 0)
35     fig, axs = plt.subplots(n_rows, n_cols, figsize=(12, 8))
36     axs = axs.flatten()
37     for i, seq in enumerate(sequences):
38         axs[i].hist(seq, bins=25, alpha=0.7, color='blue', edgecolor='black')
39         axs[i].set_title(f"LCG_Set_{i+1}")
40     plt.tight_layout()
41     plt.show()
42
43 plot_histograms(random_sequences)
44
45 def autocorrelation(sequence):
46     n = len(sequence)
47     mean = np.mean(sequence)
48     autocorr = np.correlate(sequence - mean, sequence - mean, mode='full')
49     autocorr = autocorr[n - 1:] / autocorr[n - 1]
50     return autocorr
51
52 plt.figure(figsize=(10, 6))
53 for i, seq in enumerate(random_sequences):
54     acorr = autocorrelation(seq)
55     plt.plot(acorr[50:], label=f"Set_{i+1}")
56 plt.ylim([0, 1])
57 plt.xlim([0, 2500])
58 plt.title("Autocorrelation of Random Sequences")
59 plt.xlabel("Lag")
60 plt.ylabel("Autocorrelation")
61 plt.legend()
62 plt.show()
63
64 from scipy.stats import kstest
65
66 print("Kolmogorov-Smirnov Test Results:")
67 for i, seq in enumerate(random_sequences):
68     stat, p_value = kstest(seq, 'uniform')
69     print(f"Set_{i+1}: KS Statistic={stat:.5f}, p-value={p_value:.5f}")
70
71 def spectrum_analysis_all(sequences):
72     n_cols = 2
73     n_rows = len(sequences) // n_cols + (len(sequences) % n_cols > 0)
74     fig, axs = plt.subplots(n_rows, n_cols, figsize=(12, 8))
75     axs = axs.flatten()

```

```

76
77     freq_range = (0, 0.5)
78     magnitude_max = 250
79
80     for i, sequence in enumerate(sequences):
81         sequence = sequence - np.mean(sequence)
82         fft_result = np.fft.fft(sequence)
83         magnitude = np.abs(fft_result)
84         freq = np.fft.fftfreq(len(sequence))
85
86         axs[i].plot(freq[:len(freq)//2], magnitude[:len(magnitude)//2], color='blue')
87         axs[i].set_title(f"Set_{i+1}")
88         axs[i].set_xlim(freq_range)
89         axs[i].set_ylim(0, magnitude_max)
90         axs[i].grid()
91
92     for ax in axs[len(sequences):]:
93         ax.axis('off')
94
95     plt.tight_layout()
96     plt.show()
97
98 spectrum_analysis_all(random_sequences)
99
100 def find_period(sequence):
101     seen = {}
102     for i, value in enumerate(sequence):
103         if value in seen:
104             return i - seen[value]
105         seen[value] = i
106     return None
107
108 print("Period Lengths:")
109 for i, seq in enumerate(random_sequences):
110     period = find_period(seq)
111     print(f"Set_{i+1}: {period if period else 'No Period Detected'}")

```

Problem 2

Random walks in two dimensions

代码运行结果为:

```

Simple random walks:
<x(N)> = 0.000, <y(N)> = 0.000, <ΔR^2(N)> = 4.000

non-reversal random walks:
<x(N)> = 0.000, <y(N)> = -0.000, <ΔR^2(N)> = 4.971

self-avoiding random walks:
<x(N)> = 0.000, <y(N)> = 0.000, <ΔR^2(N)> = 6.350
Total number of self-avoiding walks: 618

```

值得注意的是, 在实验过程中我发现由于浮点误差, 代码中的“is_self_avoiding()”函数会因为设置的随机行走方向存在浮点数, 而导致判断错误得到有 634 步 self-avoiding random walks, 但这是错误的, 所以我修改随机行走方向全部为整数用来输入到“is_non_reversal()”和“is_self_avoiding()”函数中用来判断随机行走类型, 然后在输入到“calculate_statistics()”函数中进行 x 和 y 相关的计算时, 对 y 数据乘以 $\sqrt{3}$ 来将随机行走方向回到真实数值。

```

1 import itertools
2 import numpy as np
3
4 directions = [
5     (1, 0),
6     (-1, 0),
7     (0.5, 1),
8     (-0.5, 1),
9     (0.5, -1),
10    (-0.5, -1)
11 ]
12
13
14 def enumerate_walks(N):
15     return list(itertools.product(range(6), repeat=N))
16
17 def calculate_trajectory(steps):
18     x, y = 0, 0
19     trajectory = [(x, y)]
20     for step in steps:
21         dx, dy = directions[step]
22         x += dx
23         y += dy
24         trajectory.append((x, y))
25     return trajectory
26
27
28 def is_non_reversal(steps):

```

```

29     for i in range(1, len(steps)):
30         if steps[i] == (steps[i - 1] + 3) % 6:
31             return False
32     return True
33
34
35 def is_self_avoiding(trajectory):
36     visited = set()
37     for point in trajectory:
38         if point in visited:
39             return False
40         visited.add(point)
41     return True
42
43 def is_self_avoiding_1(trajectory):
44     visited = set()
45     a=0
46     for point in trajectory:
47         if point in visited:
48             return False
49         visited.add(point)
50         a += 1
51     return True, a
52
53
54
55 def calculate_statistics(trajectories):
56     x_vals = [traj[-1][0] for traj in trajectories]
57     y_vals = [traj[-1][1]*np.sqrt(3)/2 for traj in trajectories]
58     # TIMES \SQRT{3}/2 TO RETURN THE DIRECTION TO REAL VALUE.
59
60     mean_x = np.mean(x_vals)
61     mean_y = np.mean(y_vals)
62     mean_x2 = np.mean(np.array(x_vals) ** 2)
63     mean_y2 = np.mean(np.array(y_vals) ** 2)
64     mean_delta_r2 = mean_x2 + mean_y2 - mean_x ** 2 - mean_y ** 2
65     return mean_x, mean_y, mean_delta_r2
66
67
68 def main():
69     N = 4
70     all_walks = enumerate_walks(N)
71
72     simple_trajectories = [calculate_trajectory(steps) for steps in all_walks]
73     #PRINT(SIMPLE_TRAJECTORIES)
74     simple_mean_x, simple_mean_y, simple_mean_delta_r2 = calculate_statistics(
        simple_trajectories)

```



```

75     print("Simple_random_walks:")
76     print(f"<x(N)>={simple_mean_x:.3f}, <y(N)>={simple_mean_y:.3f}, <ΔR^2(N)>={
        simple_mean_delta_r2:.3f}")
77
78     non_reversal_walks = [steps for steps in all_walks if is_non_reversal(steps)]
79     non_reversal_trajectories = [calculate_trajectory(steps) for steps in non_reversal_walks
        ]
80     non_reversal_mean_x, non_reversal_mean_y, non_reversal_mean_delta_r2 =
        calculate_statistics(
81         non_reversal_trajectories)
82     print("\nnon-reversal_random_walks:")
83     print(
84         f"<x(N)>={non_reversal_mean_x:.3f}, <y(N)>={non_reversal_mean_y:.3f}, <ΔR^2(N)>=
        {non_reversal_mean_delta_r2:.3f}")
85
86     self_avoiding_trajectories = [traj for traj in simple_trajectories if is_self_avoiding(
        traj)]
87     self_avoiding_mean_x, self_avoiding_mean_y, self_avoiding_mean_delta_r2 =
        calculate_statistics(
88         self_avoiding_trajectories)
89     print("\nself-avoiding_random_walks:")
90     print(
91         f"<x(N)>={self_avoiding_mean_x:.3f}, <y(N)>={self_avoiding_mean_y:.3f}, <ΔR^2(N)>=
        {self_avoiding_mean_delta_r2:.3f}")
92
93     a=0
94     for traj in simple_trajectories:
95         if is_self_avoiding(traj):
96             a+=1
97     print(f"Total_number_of_self-avoiding_walks:{a}")
98
99 if __name__ == "__main__":
100     main()

```

Problem 3

Numerical solution of the potential within a rectangular region.

a. 对 $n_x = n_y = 9, n_x = n_y = 45, n_x = n_y = 72$ 进行分析, 结果如下, 其中 $n_x = n_y = 45$ 是为了与 c 题进行对比。

```

Iterations for 9x9 grid: 57
Iterations for 45x45 grid: 393
Iterations for 72x72 grid: 257

```

```

1 import numpy as np
2
3 def initialize_grid(nx, ny, boundary_top, boundary_bottom, boundary_left, boundary_right):
4     grid = np.zeros((ny, nx))
5     grid[0, :] = boundary_top
6     grid[-1, :] = boundary_bottom
7     grid[:, 0] = boundary_left
8     grid[:, -1] = boundary_right
9     return grid
10
11 def jacobi_relaxation(grid, tol=1e-2, max_iter=10000):
12     ny, nx = grid.shape
13     new_grid = grid.copy()
14     for iteration in range(max_iter):
15         for i in range(1, ny-1):
16             for j in range(1, nx-1):
17                 new_grid[i, j] = 0.25 * (grid[i+1, j] + grid[i-1, j] + grid[i, j+1] + grid[i,
18                     j-1])
19             diff = np.abs(new_grid - grid).max()
20             if diff < tol:
21                 return new_grid, iteration
22             grid[:, :] = new_grid
23     return new_grid, max_iter
24
25 # FOR NX=NY=9
26 nx_9, ny_9 = 9, 9
27 boundary_top, boundary_bottom = 9, 9
28 boundary_left, boundary_right = 5, 5
29 grid_9x9 = initialize_grid(nx_9, ny_9, boundary_top, boundary_bottom, boundary_left,
30     boundary_right)
31 final_grid_9x9, iterations_9x9 = jacobi_relaxation(grid_9x9, tol=0.01)
32
33 # FOR NX=NY=45
34 nx_45, ny_45 = 45, 45
35 grid_45x45 = initialize_grid(nx_45, ny_45, boundary_top, boundary_bottom, boundary_left,
36     boundary_right)
37 final_grid_45x45, iterations_45x45 = jacobi_relaxation(grid_45x45, tol=0.01)
38
39 # FOR NX=NY=72
40 nx_72, ny_72 = 72, 72
41 grid_72x72 = initialize_grid(nx_72, ny_72, boundary_top, boundary_bottom, boundary_left,
42     boundary_right)
43 final_grid_72x72, iterations_72x72 = jacobi_relaxation(grid_72x72, tol=0.01)
44
45 print(f"Iterations_for_9x9_grid:{iterations_9x9}")
46 print(f"Iterations_for_45x45_grid:{iterations_45x45}")

```

```
43 print(f"Iterations_for_72x72_grid:{iterations_72x72}")
```

b. 结果如下，可以看到进行边界值平均作为初始猜测值的方法是更好的，需要的迭代次数远少于随机猜测初始值的方法。对两种方法得到的结果进行差值分析，可以看到两种方法的结果相差很小。

```
Iterations with average guess: 9
Iterations with random guess: 83
Mean Absolute Difference: 0.005835184233804118
Max Absolute Difference: 0.027777134676482795
```

```
1 import numpy as np
2
3 def initialize_grid_with_guess(nx, ny, boundary_top, boundary_bottom, boundary_left,
    boundary_right, guess_type, vmax=20):
4     grid = np.zeros((ny, nx))
5     grid[0, :] = boundary_top
6     grid[-1, :] = boundary_bottom
7     grid[:, 0] = boundary_left
8     grid[:, -1] = boundary_right
9
10    if guess_type == "average":
11        avg_value = (boundary_top + boundary_bottom + boundary_left + boundary_right) / 4
12        grid[1:-1, 1:-1] = avg_value
13    elif guess_type == "random":
14        grid[1:-1, 1:-1] = np.random.uniform(-vmax, vmax, size=(ny-2, nx-2))
15
16    return grid
17
18 def jacobi_relaxation(grid, tol=1e-2, max_iter=10000):
19     ny, nx = grid.shape
20     new_grid = grid.copy()
21     for iteration in range(max_iter):
22         for i in range(1, ny-1):
23             for j in range(1, nx-1):
24                 new_grid[i, j] = 0.25 * (grid[i+1, j] + grid[i-1, j] + grid[i, j+1] + grid[i,
                    j-1])
25         diff = np.abs(new_grid - grid).max()
26         if diff < tol:
27             return new_grid, iteration
28         grid[:, :] = new_grid
29     return new_grid, max_iter
30
31 nx, ny = 9, 9
32 boundary_top, boundary_bottom = 9, 9
```

```

33 boundary_left, boundary_right = 5, 5
34 vmax = 20
35
36 # CASE 1: AVERAGE BOUNDARY VALUES GUESS
37 grid_average = initialize_grid_with_guess(nx, ny, boundary_top, boundary_bottom,
      boundary_left, boundary_right, "average")
38 final_grid_avg, iterations_avg = jacobi_relaxation(grid_average, tol=0.01)
39
40 # CASE 2: RANDOM GUESS
41 grid_random = initialize_grid_with_guess(nx, ny, boundary_top, boundary_bottom,
      boundary_left, boundary_right, "random", vmax=vmax)
42 final_grid_random, iterations_random = jacobi_relaxation(grid_random, tol=0.01)
43
44 print(f"Iterations with average guess: {iterations_avg}")
45 print(f"Iterations with random guess: {iterations_random}")
46
47 difference = final_grid_avg - final_grid_random
48 abs_difference = np.abs(difference)
49
50 mean_diff = np.mean(abs_difference)
51 max_diff = np.max(abs_difference)
52
53 print(f"Mean Absolute Difference of two methods: {mean_diff}")
54 print(f"Max Absolute Difference of two methods: {max_diff}")

```

c. 结果如下，可以看到比 a 中的 $n_x = n_y = 45$ 情况运算更快，迭代次数更少。

Checkerboard methods' iterations for grid 45x45: 344

```

1 import numpy as np
2
3 def initialize_grid(nx, ny, boundary_top, boundary_bottom, boundary_left, boundary_right):
4     grid = np.zeros((ny, nx))
5     grid[0, :] = boundary_top
6     grid[-1, :] = boundary_bottom
7     grid[:, 0] = boundary_left
8     grid[:, -1] = boundary_right
9     return grid
10
11 def checkerboard_jacobi_relaxation(grid, tol=1e-2, max_iter=10000):
12     ny, nx = grid.shape
13     new_grid = grid.copy()
14
15     for iteration in range(max_iter):

```

```
16     for i in range(1, ny-1):
17         for j in range(1, nx-1):
18             if (i + j) % 2 == 0: # RED POINT
19                 new_grid[i, j] = 0.25 * (grid[i+1, j] + grid[i-1, j] + grid[i, j+1] + grid
20                     [i, j-1])
21
22     for i in range(1, ny-1):
23         for j in range(1, nx-1):
24             if (i + j) % 2 == 1: # BLACK POINT
25                 new_grid[i, j] = 0.25 * (new_grid[i+1, j] + new_grid[i-1, j] + new_grid[i,
26                     j+1] + new_grid[i, j-1])
27
28     diff = np.abs(new_grid - grid).max()
29     if diff < tol:
30         return new_grid, iteration
31
32     grid[:, :] = new_grid
33
34     return new_grid, max_iter
35
36 nx, ny = 45, 45
37 boundary_top, boundary_bottom = 9, 9
38 boundary_left, boundary_right = 5, 5
39
40 grid = initialize_grid(nx, ny, boundary_top, boundary_bottom, boundary_left, boundary_right)
41
42 final_grid, iterations = checkerboard_jacobi_relaxation(grid, tol=0.01)
43
44 print(f"Checkerboard_methods' iterations for grid 45x45: {iterations}")
```
