ZJU Computational Physics: Homework #4

Due on Monday, December 13, 2024

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

NAKO

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 $X_{n+1} = (a \cdot X_n + c) \mod m$，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）

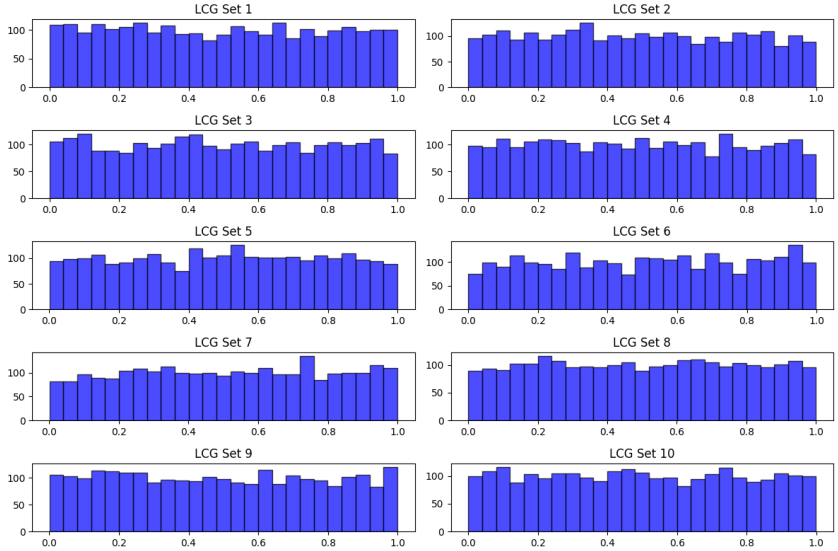


图 1: 在 10 组参数下生成的随机数分布的均匀性

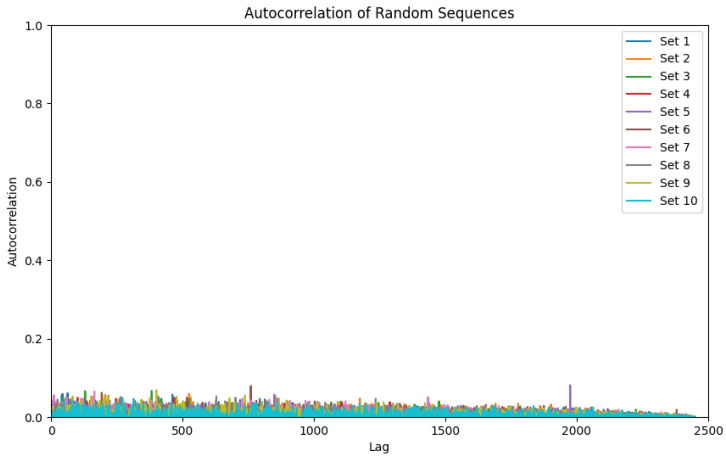


图 2: 在 10 组参数下生成的随机数分布的自相关性

Problem 1 continued on next page...

```
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

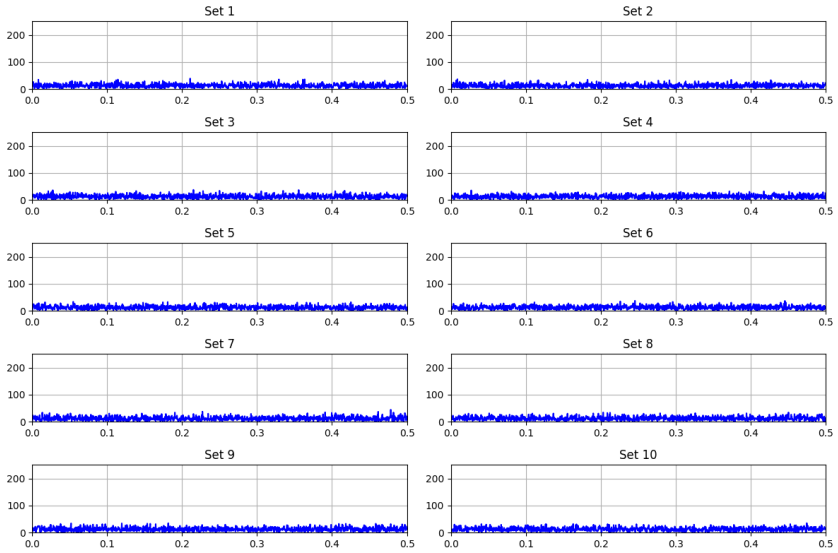
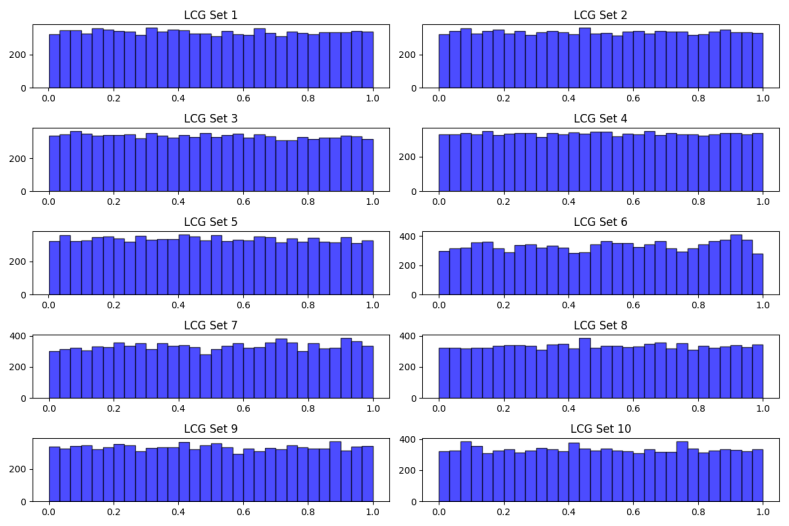


图 4: 在 10 组参数下生成的随机数分布的频谱分析

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：

Problem 1 continued on next page...



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。

```python
import numpy as np
import matplotlib.pyplot as plt

def lcg(m, a, c, seed, n):
    x = seed
    numbers = []
    for _ in range(n):
        x = (a * x + c) % m
        numbers.append(x / m)
    return numbers

params = [
    (214326, 1807, 45289),
    (244944, 1597, 51749),
    (233280, 1861, 49297),
    (175000, 2661, 26979),
    (121500, 4081, 25673),
    (145800, 3661, 30809),
    (139968, 3877, 29573),
    (214326, 3613, 45289),
    (714025, 1366, 150889),
    (134456, 8121, 28411),
]

seed = 42
num_samples = 2500

random_sequences = []
```

Problem 1 continued on next page...

```
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()
```

Problem 1 continued on next page...

```
    freq_range = (0, 0.5)
    magnitude_max = 250

    for i, sequence in enumerate(sequences):
        sequence = sequence - np.mean(sequence)
        fft_result = np.fft.fft(sequence)
        magnitude = np.abs(fft_result)
        freq = np.fft.fftfreq(len(sequence))

        axs[i].plot(freq[:len(freq)//2], magnitude[:len(magnitude)//2], color='blue')
        axs[i].set_title(f"Set {i+1}")
        axs[i].set_xlim(freq_range)
        axs[i].set_ylim(0, magnitude_max)
        axs[i].grid()

    for ax in axs[len(sequences):]:
        ax.axis('off')

    plt.tight_layout()
    plt.show()

spectrum_analysis_all(random_sequences)

def find_period(sequence):
    seen = {}
    for i, value in enumerate(sequence):
        if value in seen:
            return i - seen[value]
        seen[value] = i
    return None

print("Period Lengths:")
for i, seq in enumerate(random_sequences):
    period = find_period(seq)
    print(f"Set {i+1}: Period Length = {period if period else 'No Period Detected'}")
```

Problem 2

Random walks in two dimensions

代码运行结果为：

Problem 2 continued on next page...

```
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。

```python
import itertools
import numpy as np

directions = [
    (1, 0),
    (-1, 0),
    (0.5, 1),
    (-0.5, 1),
    (0.5, -1),
    (-0.5, -1)
]


def enumerate_walks(N):
    return list(itertools.product(range(6), repeat=N))

def calculate_trajectory(steps):
    x, y = 0, 0
    trajectory = [(x, y)]
    for step in steps:
        dx, dy = directions[step]
        x += dx
        y += dy
        trajectory.append((x, y))
    return trajectory


def is_non_reversal(steps):
```

Problem 2 continued on next page...

```
    for i in range(1, len(steps)):
        if steps[i] == (steps[i - 1] + 3) % 6:
            return False
    return True


def is_self_avoiding(trajectory):
    visited = set()
    for point in trajectory:
        if point in visited:
            return False
        visited.add(point)
    return True

def is_self_avoiding_1(trajectory):
    visited = set()
    a=0
    for point in trajectory:
        if point in visited:
            return False
        visited.add(point)
        a += 1
    return True, a



def calculate_statistics(trajectories):
    x_vals = [traj[-1][0] for traj in trajectories]
    y_vals = [traj[-1][1]*np.sqrt(3)/2 for traj in trajectories]
    # TIMES \SQRT{3}/2 TO RETURN THE DIRECTION TO REAL VALUE.

    mean_x = np.mean(x_vals)
    mean_y = np.mean(y_vals)
    mean_x2 = np.mean(np.array(x_vals) ** 2)
    mean_y2 = np.mean(np.array(y_vals) ** 2)
    mean_delta_r2 = mean_x2 + mean_y2 - mean_x ** 2 - mean_y ** 2
    return mean_x, mean_y, mean_delta_r2


def main():
    N = 4
    all_walks = enumerate_walks(N)

    simple_trajectories = [calculate_trajectory(steps) for steps in all_walks]
    #PRINT(SIMPLE_TRAJECTORIES)
    simple_mean_x, simple_mean_y, simple_mean_delta_r2 = calculate_statistics(
        simple_trajectories)
```

Problem 2 continued on next page...

```
    print("Simple␣random␣walks:")
    print(f"<x(N)>␣=␣{simple_mean_x:.3f},␣<y(N)>␣=␣{simple_mean_y:.3f},␣<ΔR^2(N)>␣=␣{
        simple_mean_delta_r2:.3f}")

    non_reversal_walks = [steps for steps in all_walks if is_non_reversal(steps)]
    non_reversal_trajectories = [calculate_trajectory(steps) for steps in non_reversal_walks
        ]
    non_reversal_mean_x, non_reversal_mean_y, non_reversal_mean_delta_r2 =
        calculate_statistics(
       non_reversal_trajectories)
    print("\nnon-reversal␣random␣walks:")
    print(
       f"<x(N)>␣=␣{non_reversal_mean_x:.3f},␣<y(N)>␣=␣{non_reversal_mean_y:.3f},␣<ΔR^2(N)>␣=
           ␣{non_reversal_mean_delta_r2:.3f}")

    self_avoiding_trajectories = [traj for traj in simple_trajectories if is_self_avoiding(
        traj)]
    self_avoiding_mean_x, self_avoiding_mean_y, self_avoiding_mean_delta_r2 =
        calculate_statistics(
       self_avoiding_trajectories)
    print("\nself-avoiding␣random␣walks:")
    print(
       f"<x(N)>␣=␣{self_avoiding_mean_x:.3f},␣<y(N)>␣=␣{self_avoiding_mean_y:.3f},␣<ΔR^2(N)>
           ␣=␣{self_avoiding_mean_delta_r2:.3f}")
    a=0
    for traj in simple_trajectories:
       if is_self_avoiding(traj):
          a+=1
    print(f"Total␣number␣of␣self-avoiding␣walks:␣{a}")

if __name__ == "__main__":
    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
```

Problem 3 continued on next page...

```python
import numpy as np

def initialize_grid(nx, ny, boundary_top, boundary_bottom, boundary_left, boundary_right):
    grid = np.zeros((ny, nx))
    grid[0, :] = boundary_top
    grid[-1, :] = boundary_bottom
    grid[:, 0] = boundary_left
    grid[:, -1] = boundary_right
    return grid

def jacobi_relaxation(grid, tol=1e-2, max_iter=10000):
    ny, nx = grid.shape
    new_grid = grid.copy()
    for iteration in range(max_iter):
        for i in range(1, ny-1):
            for j in range(1, nx-1):
                new_grid[i, j] = 0.25 * (grid[i+1, j] + grid[i-1, j] + grid[i, j+1] + grid[i,
                    j-1])
        diff = np.abs(new_grid - grid).max()
        if diff < tol:
            return new_grid, iteration
        grid[:, :] = new_grid
    return new_grid, max_iter

# For nx=ny=9
nx_9, ny_9 = 9, 9
boundary_top, boundary_bottom = 9, 9
boundary_left, boundary_right = 5, 5
grid_9x9 = initialize_grid(nx_9, ny_9, boundary_top, boundary_bottom, boundary_left,
    boundary_right)
final_grid_9x9, iterations_9x9 = jacobi_relaxation(grid_9x9, tol=0.01)

# For nx=ny=45
nx_45, ny_45 = 45, 45
grid_45x45 = initialize_grid(nx_45, ny_45, boundary_top, boundary_bottom, boundary_left,
    boundary_right)
final_grid_45x45, iterations_45x45 = jacobi_relaxation(grid_45x45, tol=0.01)

# For nx=ny=72
nx_72, ny_72 = 72, 72
grid_72x72 = initialize_grid(nx_72, ny_72, boundary_top, boundary_bottom, boundary_left,
    boundary_right)
final_grid_72x72, iterations_72x72 = jacobi_relaxation(grid_72x72, tol=0.01)

print(f"Iterations for 9x9 grid: {iterations_9x9}")
print(f"Iterations for 45x45 grid: {iterations_45x45}")
```

Problem 3 continued on next page...

```
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
```

```
import numpy as np

def initialize_grid_with_guess(nx, ny, boundary_top, boundary_bottom, boundary_left,
    boundary_right, guess_type, vmax=20):
    grid = np.zeros((ny, nx))
    grid[0, :] = boundary_top
    grid[-1, :] = boundary_bottom
    grid[:, 0] = boundary_left
    grid[:, -1] = boundary_right

    if guess_type == "average":
        avg_value = (boundary_top + boundary_bottom + boundary_left + boundary_right) / 4
        grid[1:-1, 1:-1] = avg_value
    elif guess_type == "random":
        grid[1:-1, 1:-1] = np.random.uniform(-vmax, vmax, size=(ny-2, nx-2))

    return grid

def jacobi_relaxation(grid, tol=1e-2, max_iter=10000):
    ny, nx = grid.shape
    new_grid = grid.copy()
    for iteration in range(max_iter):
        for i in range(1, ny-1):
            for j in range(1, nx-1):
                new_grid[i, j] = 0.25 * (grid[i+1, j] + grid[i-1, j] + grid[i, j+1] + grid[i,
                    j-1])
        diff = np.abs(new_grid - grid).max()
        if diff < tol:
            return new_grid, iteration
        grid[:, :] = new_grid
    return new_grid, max_iter

nx, ny = 9, 9
boundary_top, boundary_bottom = 9, 9
```

Problem 3 continued on next page...

```
boundary_left, boundary_right = 5, 5
vmax = 20

# Case 1: Average boundary values guess
grid_average = initialize_grid_with_guess(nx, ny, boundary_top, boundary_bottom,
    boundary_left, boundary_right, "average")
final_grid_avg, iterations_avg = jacobi_relaxation(grid_average, tol=0.01)

# Case 2: Random guess
grid_random = initialize_grid_with_guess(nx, ny, boundary_top, boundary_bottom,
    boundary_left, boundary_right, "random", vmax=vmax)
final_grid_random, iterations_random = jacobi_relaxation(grid_random, tol=0.01)

print(f"Iterations with average guess: {iterations_avg}")
print(f"Iterations with random guess: {iterations_random}")

difference = final_grid_avg - final_grid_random
abs_difference = np.abs(difference)

mean_diff = np.mean(abs_difference)
max_diff = np.max(abs_difference)

print(f"Mean Absolute Difference of two methods: {mean_diff}")
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
```

```
import numpy as np

def initialize_grid(nx, ny, boundary_top, boundary_bottom, boundary_left, boundary_right):
    grid = np.zeros((ny, nx))
    grid[0, :] = boundary_top
    grid[-1, :] = boundary_bottom
    grid[:, 0] = boundary_left
    grid[:, -1] = boundary_right
    return grid

def checkerboard_jacobi_relaxation(grid, tol=1e-2, max_iter=10000):
    ny, nx = grid.shape
    new_grid = grid.copy()

    for iteration in range(max_iter):
```

Problem 3 continued on next page...

```
        for i in range(1, ny-1):
            for j in range(1, nx-1):
                if (i + j) % 2 == 0: # Red point
                    new_grid[i, j] = 0.25 * (grid[i+1, j] + grid[i-1, j] + grid[i, j+1] + grid
                        [i, j-1])

        for i in range(1, ny-1):
            for j in range(1, nx-1):
                if (i + j) % 2 == 1: # Black point
                    new_grid[i, j] = 0.25 * (new_grid[i+1, j] + new_grid[i-1, j] + new_grid[i,
                         j+1] + new_grid[i, j-1])

        diff = np.abs(new_grid - grid).max()
        if diff < tol:
            return new_grid, iteration

        grid[:, :] = new_grid

    return new_grid, max_iter

nx, ny = 45, 45
boundary_top, boundary_bottom = 9, 9
boundary_left, boundary_right = 5, 5

grid = initialize_grid(nx, ny, boundary_top, boundary_bottom, boundary_left, boundary_right)

final_grid, iterations = checkerboard_jacobi_relaxation(grid, tol=0.01)

print(f"Checkerboard␣methods'␣iterations␣for␣grid␣45x45: {iterations}")
```