IEOR 4404 Simulation **Fall 2024**

Homework 7

Instructor: Henry Lam

Problem 1 A Markov chain $\{X_n, n \geq 0\}$ with states $0, 1, 2$, has the transition probability matrix

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{3} & \frac{1}{6} \\ 0 & \frac{1}{3} & \frac{2}{3} \\ \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix}$$

Write the pseudo code for simulating the Markov chain up to time N (where N can be potentially a random time decided by a certain criterion; examples are in parts (e) and (f) below), by assuming each of the following:

(a) The computer has the capability to generate discrete random variables with a finite number of values, as long as we specify these values, say x_i, and probabilities, say p_i.

(b) The capability to generate $Unif(0, 1)$.

By simulating the Markov chain 100 times (i.e., 100 trajectories) using any of the above pseudo codes (i.e., you may use any available built-in routine for generating discrete random variables, or use $Unif(0, 1)$ generator), give point estimates of:

(c) $P(X_{10} = 1|X_0 = 0)$

(d) $E[X_{10}|X_0 = 0]$

(e) $P(T \leq 10|X_0 = 0)$, where $T = \min\{n : X_n = 1\}$ is the first time to hit state 1.

(f) $E[T|X_0 = 0]$, where $T = \min\{n : X_n = 1\}$ is the first time to hit state 1.

Finally, simulate the Markov chain starting from state 0, for 10000 steps. Regarding the first 100 as the burn-in steps, give point estimates of:

(g) Long-run proportions of visits to states 0, 1 and 2.

(h) Long-run average cost, where the cost is 1 for state 0, 3 for state 1, and 2 for state 2.

Problem 2 Each individual in a population of size N is, in each period, either active or inactive. If an individual is active in a period then, independent of all else, that individual will be active in the next period with probability α. Similarly, if an individual is inactive in a period then, independent of all else, that individual will be inactive in the next period with probability β.

(a) Let X_n denote the number of individuals that are active in period n. Argue that $X_n, n \geq 0$ is a Markov chain.

(b) Continuing from part (a), suppose $N = 100$, $\alpha = 0.2$ and $\beta = 0.4$. Initially there are 10 active individuals. Simulate the Markov chain $X_n, n \geq 0$ for 10000 time steps, with a burn-in period of 1000 steps, to estimate the long-run average number of active individuals. In simulating the Markov chain, you can either use any available built-in routine for generating discrete random variables, or the $Unif(0, 1)$ generator.

(c) Let Y_n and Z_n denote the numbers of individuals that are active in period n that were initially active and inactive, respectively. That is, $X_n = Y_n + Z_n$ and $Y_0 = 10$ and $Z_0 = 0$. Argue that the tuple $(Y_n, Z_n), n \geq 0$ is a Markov chain.

(d) Continuing from part (c), suppose $N = 100$, $\alpha = 0.2$ and $\beta = 0.4$. Simulate the Markov chain $(Y_n, Z_n), n \geq 0$ for 10000 time steps, with a burn-in period of 1000 steps, to estimate the long-run probability that more active individuals come from the initially active group than the initially inactive group, (i.e., $\mathbb{P}(Y_n > Z_n)$). In simulating the Markov chain, you can either use any available built-in routine for generating discrete random variables, or the $Unif(0,1)$ generator.

Problem 3

(a) Write the pseudo-code of the Metropolis-Hastings algorithm to approximate the conditional distribution of 10 independent exponential random variables $X_1, \ldots, X_{10}$ with common mean 1 given that $\prod_{i=1}^{10} X_i > 20$. In the algorithm, use a proposal transition of moving by an independent $Unif(-1,1)$ in each coordinate. (Note that it is fine to have the transition shooting outside the support of the target distribution, in which case we regard the target density as 0 at that position and the acceptance probability would become 0 by definition.)

(b) Implement the algorithm using 1000 time steps, with 100 burn-in steps, and initial value of $20^{1/10}$ for each X_i. Plot the distribution of X_1.

Problem 4 Suppose the joint density of X, Y, Z is given by

$$f(x, y, z) = Ce^{-(x+y+z+axy+bxz+cyz)}, x > 0, y > 0, z > 0$$

where a, b, c are specified nonnegative constants, and C does not depend on x, y, z. We want to estimate $E[XYZ]$ when $a = b = c = 1$ by the following procedures:

(a) The Metropolis-Hastings algorithm with the proposal transition of moving by an independent $Unif(-1,1)$ for each of x, y, z.

(b) The Gibbs sampler.

For each approach above, write the pseudo-code and implement the algorithm using 1000 time steps, with 100 burn-in steps and initial value of 1 for all of x, y, z.

Alexandre
Duhamel
afd2153
MSOR

HW 7: Simulation

Problem 1:

a) 1. Initialize $X_0 = 0$

2. For n in $1 \dots N$:

$$\begin{cases} \text{if } X_n = 0, \; X_{n+1} = \begin{cases} 0 & \text{wp } 1/2 \\ 1 & \text{wp } 1/3 \\ 2 & \text{wp } 1/6 \end{cases} \\ \text{if } X_n = 1, \; X_{n+1} = \begin{cases} \cancel{0 \;\; \text{wp} \;\; 0} \\ 1 & \text{wp } 1/3 \\ 2 & \text{wp } 2/3 \end{cases} \\ \text{if } X_n = 2, \; X_{n+1} = \begin{cases} 0 & \text{wp } 1/2 \\ \cancel{1 \;\; \text{wp} \;\; 0} \\ 2 & \text{wp } 1/2 \end{cases} \end{cases}$$

b) 1. Initialize $X_0 = 0$

2. For n in $1 \dots N$:

$$\begin{cases} \text{Generate } U \sim \text{Unif}(0,1) \\ \text{if } X_n = 0 : \begin{cases} \text{if } U \leq 1/2, \; X_{n+1} = 0 \\ \text{if } 1/2 < U \leq 5/6, \; X_{n+1} = 1 \\ \text{if } 5/6 < U, \; X_{n+1} = 2 \end{cases} \\ \text{if } X_n = 0 : \begin{cases} \cancel{\text{if } U \leq 0, \; X_{n+1} = 0} \\ \text{if } U \leq 1/3, \; X_{n+1} = 1 \\ \text{if } 1/3 < U, \; X_{n+1} = 2 \end{cases} \\ \text{if } X_n = 0 : \begin{cases} \text{if } U \leq 1/2, \; X_{n+1} = 0 \\ \cancel{\text{if } 1/2 < U \leq 1/2, \; X_{n+1} = 1} \\ \text{if } 1/2 < U, \; X_{n+1} = 2 \end{cases} \end{cases}$$

c)

```python
P = np.array([
    [0.5, 0.33, 0.17],
    [0, 0.33, 0.67],
    [0.5, 0, 0.5]
])

def next_state(current_state):
    return np.random.choice([0, 1, 2], p=P[current_state])

np.random.seed(0)
```

```python
# (c) P(X10 = 1 | X0 = 0)

def estimate_prob_X10_equals_1(initial_state=0, N=100, time_step=10):
    count = 0
    for _ in range(N):
        state = initial_state
        for _ in range(time_step):
            state = next_state(state)
        if state == 1:
            count += 1
    return count / N

prob_X10_equals_1 = estimate_prob_X10_equals_1()
print(f"P(X10 = 1 | X0 = 0): {prob_X10_equals_1:.4f}")
```

```
P(X10 = 1 | X0 = 0): 0.1600
```

d)

```python
# Part (d): Estimate E[X10 | X0 = 0]
def estimate_expected_X10(initial_state=0, N=100, time_step=10):
    total = 0
    for _ in range(N):
        state = initial_state
        for _ in range(time_step):
            state = next_state(state)
        total += state
    return total / N

expected_X10 = estimate_expected_X10()
print(f"E[X10 | X0 = 0]: {expected_X10:.4f}")
```

```
E[X10 | X0 = 0]: 1.1600
```

e)

```python
# Part (e): Estimate P(T <= 10 | X0 = 0), where T = min{n : Xn = 1}
def estimate_prob_T_leq_10(initial_state=0, N=100, time_limit=10):
    count = 0
    for _ in range(N):
        state = initial_state
        for step in range(time_limit + 1):
            if state == 1:
                count += 1
                break
            state = next_state(state)
    return count / N

prob_T_leq_10 = estimate_prob_T_leq_10()
print(f"P(T <= 10 | X0 = 0): {prob_T_leq_10:.4f}")
```

```
P(T <= 10 | X0 = 0): 0.8900
```

f)

```python
# Part (f): Estimate E[T | X0 = 0], where T = min{n : Xn = 1}
def estimate_expected_T(initial_state=0, N=100):
    total_steps = 0
    count = 0
    for _ in range(N):
        state = initial_state
        steps = 0
        while state != 1:
            state = next_state(state)
            steps += 1
            if steps > 100:  # Avoid infinite loop in case it never reaches state 1
                break
        if steps <= 100:
            total_steps += steps
            count += 1
    return total_steps / count if count > 0 else None

expected_T = estimate_expected_T()
print(f"E[T | X0 = 0]: {expected_T:.4f}")
```

```
E[T | X0 = 0]: 4.3500
```

g)

```python
# Part (g): Long-run proportions of visits to states 0, 1, and 2
def long_run_proportions(initial_state=0, steps=10000, burn_in=100):
    state_counts = np.zeros(3)
    state = initial_state
    for step in range(steps):
        state = next_state(state)
        if step >= burn_in:
            state_counts[state] += 1
    return state_counts / (steps - burn_in)

proportions = long_run_proportions()
print(f"Long-run proportions: State 0: {proportions[0]:.4f}, State 1: {p
```

```
Long-run proportions: State 0: 0.4031, State 1: 0.2058, State 2: 0.3911
```

h)

```python
# Part (h): Long-run average cost with costs 1, 3, and 2 for states 0, 1, and 2
def long_run_average_cost(initial_state=0, steps=10000, burn_in=100):
    costs = {0: 1, 1: 3, 2: 2}
    total_cost = 0
    state = initial_state
    for step in range(steps):
        state = next_state(state)
        if step >= burn_in:
            total_cost += costs[state]
    return total_cost / (steps - burn_in)

average_cost = long_run_average_cost()
print(f"Long-run average cost: {average_cost:.4f}")
```

```
Long-run average cost: 1.7922
```

Problem 2:

a) We can model the problem as: $\begin{cases} \mathbb{P}(X_{n+1}=1 \mid X_n=1, X_{n-1}=\cdots, \dots, X_0=0) = \mathbb{P}(X_{n+1}=1 \mid X_n=1) = \alpha \\ \mathbb{P}(X_{n+1}=0 \mid X_n=0) = \beta \quad \text{similarly} \end{cases}$

Here, the future state X_{n+1} only depends on the current state X_n.

So this is a Markov chain where we can deduce the next step based on the current state and parameters α and β.

b)

```python
def simulate_X_chain(N, alpha, beta, initial_active, total_steps, burn_in):
    X_n = initial_active
    active_counts = []

    for step in range(total_steps):
        new_active_count = 0

        # Currently active individua
        for _ in range(X_n):
            if np.random.uniform(0, 1) < alpha:
                # If active individual remains active
                new_active_count += 1  # Increment active count
            elif np.random.uniform(0, 1) > alpha:
                # If this condition is not met, the individual becomes inactive, so no increment
                new_active_count += 0

        # Currently inactive individual
        for _ in range(N - X_n):
            if np.random.uniform(0, 1) >= beta:
                # If inactive individual becomes active
                new_active_count += 1  # Increment active count
            elif np.random.uniform(0, 1) < beta:
                # If this condition is not met, the individual remains inactive, so no increment
                new_active_count += 0

        X_n = new_active_count

        if step >= burn_in:
            active_counts.append(X_n)

    long_run_average = np.mean(active_counts)
    return long_run_average

long_run_average_active = simulate_X_chain(N, alpha, beta, initial_active, total_steps, burn_in)
print(f"Long-run average number of active individuals: {long_run_average_active:.2f}")
```

```python
N = 100
alpha = 0.2
beta = 0.4
initial_active = 10
total_steps = 10000
burn_in = 1000
```

```
Long-run average number of active individuals: 42.86
```

c) (Y_m, Z_m) is a Markov chain because the future state (Y_{m+1}, Z_{m+1}) only depends on the current state (Y_m, Z_m) and not the previous states.

For Y_m: the probability of staying active is α, and becoming inactive wp $1-\alpha$.

For Z_m: the probability of staying inactive is β, and becoming active wp $1-\beta$.

d)

```
initial_Y = 10
initial_Z = 0

def simulate_YZ_chain(N, alpha, beta, initial_Y, initial_Z, total_steps, burn_in):
    Y_n = initial_Y
    Z_n = initial_Z
    Y_greater_than_Z_count = 0

    for step in range(total_steps):
        new_Y_n = 0
        new_Z_n = 0

        # Initially active individual
        for _ in range(Y_n):
            if np.random.uniform(0, 1) < alpha:
                # Remains active
                new_Y_n += 1

        # Initially inactive individual who became active (in Zn)
        for _ in range(Z_n):
            if np.random.uniform(0, 1) < beta:
                # Remains inactive, so not counted in new_Z_n
                continue
            else:
                # Becomes active
                new_Z_n += 1

        # Initially inactive individuals who might become active
        for _ in range(N - initial_Y - Z_n):
            if np.random.uniform(0, 1) >= beta:
                # Becomes active
                new_Z_n += 1

        Y_n = new_Y_n
        Z_n = new_Z_n

        if step >= burn_in:
            if Y_n > Z_n:
                Y_greater_than_Z_count += 1

    long_run_probability = Y_greater_than_Z_count / (total_steps - burn_in)
    return long_run_probability

long_run_probability_Y_greater_Z = simulate_YZ_chain(N, alpha, beta, initial_Y, initial_Z, total_steps, burn_in)
print(f"Long-run probability that more active individuals come from the initially active group than the initially in
```

```
Long-run probability that more active individuals come from the initially active group than the initially inactive
group: 0.0000
```

Problem 3:

a) 1. Initialize: $X = \{X_1, \dots, X_{10}\}$ derived from $\exp(1)$ and $\prod X_i > 20$

2. For i in $1, \dots, 10$:

$$\begin{cases} Y_i = X_i + \Delta \text{ where } \Delta \sim Unif(-1,1) \\ \text{Set } Y = \{X_1, \dots, X_{i-1}, Y_i, X_{i+1}, \dots, X_{10}\} : \text{updating the } i\text{-th value} \\ \text{Compute } P_{new} = \prod_{j=1}^{10} Y_j \\ \text{If } P_{new} \le 20, \text{ set } \alpha = 0 \\ \text{Otherwise: } \alpha = \min\left(\dfrac{f(Y)\cancel{q(Y,X)}}{f(X)\cancel{q(X,Y)}}, 1\right) = \min\left(e^{-(Y_i - X_i)}, 1\right) \\ \text{Generate } U \sim Unif(0,1) : \begin{cases} \text{If } U \le \alpha : \text{accept } Y_i \text{ by setting } X_i = Y_i \\ \text{If } U > \alpha : \text{reject} : X_i = X_i \end{cases} \end{cases}$$

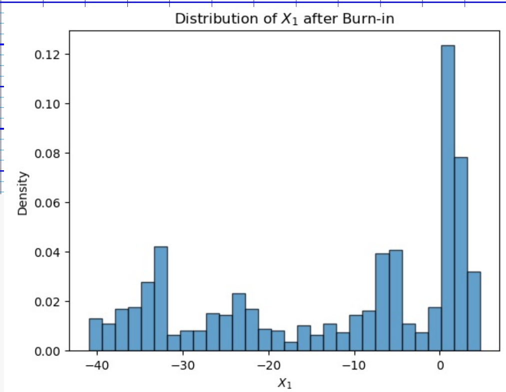
symmetric proposal due to Unif $(-1,1)$

3. Return $X = \{X_1, \dots, X_{10}\}$

b) We have $f_{X_i}(x_i) = e^{-x_i}$ ($\lambda = 1$)

So $f(X) = \prod_{i=1}^{10} f_{X_i}(x_i) = e^{-\sum_{i=1}^{10} x_i}$

independent X_i's




```python
num_steps = 1000
burn_in = 100
initial_value = 20**(1/10)
num_variables = 10

def target_density(X):
    if np.prod(X) > 20:
        return np.exp(-np.sum(X))
    else:
        return 0

def propose_new_state(current_state):
    return current_state + np.random.uniform(-1, 1, size=len(current_state))

X = np.full(num_variables, initial_value)
samples_X1 = []

# Run Metropolis-Hastings algorithm
for step in range(num_steps):
    proposed_X = propose_new_state(X)

    current_density = target_density(X)
    proposed_density = target_density(proposed_X)

    acceptance_prob = min(proposed_density / current_density, 1) if current_density > 0 else 0

    if np.random.uniform(0, 1) < acceptance_prob:
        X = proposed_X

    if step >= burn_in:
        samples_X1.append(X[0])

plt.hist(samples_X1, bins=30, density=True, alpha=0.7, edgecolor='black')
plt.title('Distribution of $X_1$ after Burn-in')
plt.xlabel('$X_1$')
plt.ylabel('Density')
plt.show()
```

Problem 4:

a)

```python
def joint_density(x, y, z):
    if x > 0 and y > 0 and z > 0:
        return np.exp(-(x + y + z + x*y + x*z + y*z))
    return 0

def metropolis_hastings(num_steps=1000, burn_in=100):
    x, y, z = 1.0, 1.0, 1.0
    xyz_values = []

    for step in range(num_steps):
        x_prime = x + np.random.uniform(-1, 1)
        y_prime = y + np.random.uniform(-1, 1)
        z_prime = z + np.random.uniform(-1, 1)

        current_density = joint_density(x, y, z)
        proposed_density = joint_density(x_prime, y_prime, z_prime)

        alpha = min(proposed_density / current_density, 1) if current_density > 0 else 0
        if np.random.uniform(0, 1) < alpha:
            x, y, z = x_prime, y_prime, z_prime

        if step >= burn_in:
            xyz_values.append(x * y * z)

    return np.mean(xyz_values)

estimated_expectation_mh = metropolis_hastings()
print(f"Estimated E[XYZ] using Metropolis-Hastings: {estimated_expectation_mh:.4f}")
```

```
Estimated E[XYZ] using Metropolis-Hastings: 0.0839
```

b)

```python
def gibbs_sampler(num_steps=1000, burn_in=100):
    x, y, z = 1.0, 1.0, 1.0
    xyz_values = []

    for step in range(num_steps):
        # Sample x given y, z
        x = np.random.exponential(scale=1 / (1 + y + z))

        # Sample y given x, z
        y = np.random.exponential(scale=1 / (1 + x + z))

        # Sample z given x, y
        z = np.random.exponential(scale=1 / (1 + x + y))

        if step >= burn_in:
            xyz_values.append(x * y * z)

    return np.mean(xyz_values)

estimated_expectation_gibbs = gibbs_sampler()
print(f"Estimated E[XYZ] using Gibbs Sampler: {estimated_expectation_gibbs:.4f}")
```

```
Estimated E[XYZ] using Gibbs Sampler: 0.0806
```