Computerized Simulation
Exercise No. 3

name : Seyed Mohammad Ghoreishy
teacher : Seyed Amirhossein Tabatabaei

Date: 1403.09.26

Contents

1 Exercise 1

Write a program using the **accept-reject method** to generate random numbers following a custom probability distribution defined by the piecewise function as follows:

$$f(x) = \begin{cases} 2x & 0 \leq x \leq 0.5 \\ 2(1-x) & 0.5 < x \leq 1. \end{cases}$$

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

def piecewise_function(x):
    return np.where((0 <= x) & (x <= 0.5), 2 * x,
            np.where((0.5 < x) & (x <= 1), 2 * (1 - x), 0))

def accept_reject_sampling(target_func, domain, n_samples, max_val):
    samples = []
    x_min, x_max = domain
    while len(samples) < n_samples:
        x = np.random.uniform(x_min, x_max)
        u = np.random.uniform(0, 1)
        if u <= target_func(x) / max_val:
            samples.append(x)
    return np.array(samples)

def plot_results(samples, target_func, domain, bins=50):
    x_vals = np.linspace(domain[0], domain[1], 1000)
    f_vals = target_func(x_vals)
    plt.figure(figsize=(8, 6))
    plt.hist(samples, bins=bins, density=True, alpha=0.6, color='g', label='Generated Samples')
    plt.plot(x_vals, f_vals, 'r-', linewidth=2, label='f(x) (Target Distribution)')
    plt.xlabel('x')
    plt.ylabel('Probability Density')
    plt.title('Generated Random Numbers vs Target Distribution')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

n_samples = 10000
domain = (0, 1)
max_val = 1
np.random.seed(0)
random_numbers = accept_reject_sampling(piecewise_function, domain, n_samples, max_val)
plot_results(random_numbers, piecewise_function, domain)
```


Generated Random Numbers vs Target Distribution
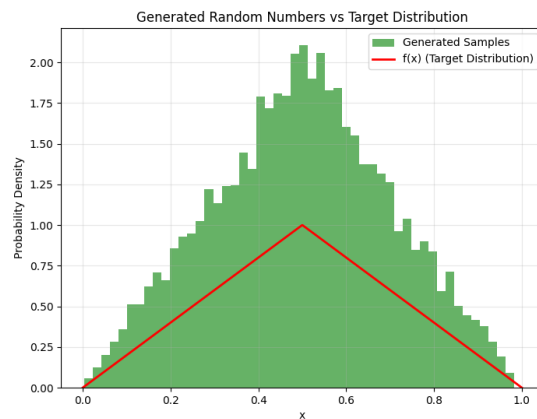
2 Exercise 2

Simulate the distribution of angles at which particles scatter when the probability distribution of scattering angles is proportional to $\cos^2(x)$. Use the **accept-reject method** to generate the angles.
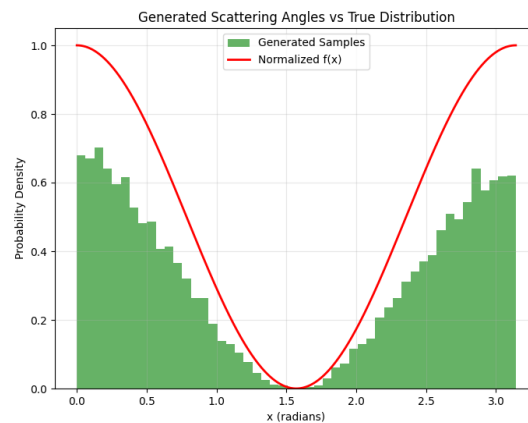
```python
import numpy as np
import matplotlib.pyplot as plt

def scattering_distribution(x):
    return np.cos(x)**2

def accept_reject_sampling(target_func, domain, n_samples, max_val):
    samples = []
    x_min, x_max = domain
    while len(samples) < n_samples:
        x = np.random.uniform(x_min, x_max)
        u = np.random.uniform(0, 1)
        if u <= target_func(x) / max_val:
            samples.append(x)
    return np.array(samples)

def plot_results(samples, target_func, domain, bins=50):
    x_vals = np.linspace(domain[0], domain[1], 1000)
    f_vals = target_func(x_vals)
    plt.figure(figsize=(8, 6))
    plt.hist(samples, bins=bins, density=True, alpha=0.6, color='g', label='Generated Samples')
    plt.plot(x_vals, f_vals / np.max(f_vals), 'r-', linewidth=2, label='Normalized f(x)')
    plt.xlabel('x (radians)')
    plt.ylabel('Probability Density')
    plt.title('Generated Scattering Angles vs True Distribution')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

n_samples = 10000
domain = (0, np.pi)
max_val = 1
np.random.seed(0)
scattering_angles = accept_reject_sampling(scattering_distribution, domain, n_samples, max_val)
plot_results(scattering_angles, scattering_distribution, domain)
```

3 Exercise 3

Simulate service times for customers in a queue. Assume service times follow a specific distribution (e.g., $f(x) = xe^{-x}$), and use the **accept-reject method** to generate the times. Analyze the average waiting time.

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

def service_time_distribution(x):
    return x * np.exp(-x) if x > 0 else 0

def exponential_distribution(x):
    return np.exp(-x) if x > 0 else 0

def accept_reject_sampling(target_func, proposal_func, domain, n_samples, max_val):
    samples = []
    while len(samples) < n_samples:
        x = np.random.exponential(1)
        u = np.random.uniform(0, 1)
        if u <= target_func(x) / (max_val * proposal_func(x)):
            samples.append(x)
    return np.array(samples)

def plot_results(samples, target_func, bins=50):
    x_vals = np.linspace(0, max(samples), 1000)
    f_vals = np.array([target_func(x) for x in x_vals])
    plt.figure(figsize=(8, 6))
    plt.hist(samples, bins=bins, density=True, alpha=0.6, color='g', label='Generated Samples')
    plt.plot(x_vals, f_vals / f_vals.max(), 'r-', linewidth=2, label='Normalized f(x)')
    plt.xlabel('x (Service Time)')
    plt.ylabel('Probability Density')
    plt.title(f'Service Times vs True Distribution \n mean:{np.mean(samples)} max:{np.max(samples)}
min:{np.min(samples)}')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

n_samples = 10000
domain = (0, np.inf)
max_val = 1 / np.exp(1)
np.random.seed(0)
service_times = accept_reject_sampling(service_time_distribution, exponential_distribution, domain,
n_samples, max_val)
plot_results(service_times, service_time_distribution)
```
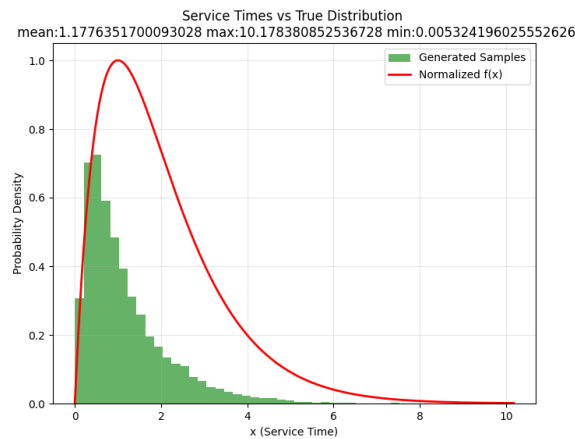

Service Times vs True Distribution
mean:1.1776351700093028 max:10.178380852536728 min:0.005324196025552626

4   Exercise 4

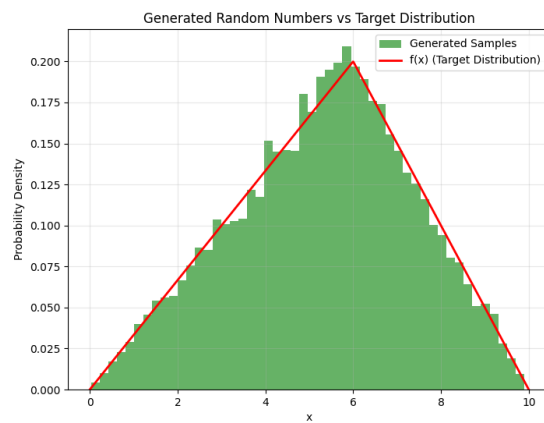Generate random numbers for a triangular distribution defined on $[a, b]$ with mode c.

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

def triangular_distribution_function(x, a, b, c):
    return np.where((a <= x) & (x < c), 2 * (x - a) / ((b - a) * (c - a)),
        np.where((c <= x) & (x <= b), 2 * (b - x) / ((b - a) * (b - c)), 0))

def accept_reject_sampling(target_func, domain, n_samples, max_val, *params):
    samples = []
    x_min, x_max = domain
    while len(samples) < n_samples:
        x = np.random.uniform(x_min, x_max)
        u = np.random.uniform(0, 1)
        if u <= target_func(x, *params) / max_val:
            samples.append(x)
    return np.array(samples)

def plot_results(samples, target_func, domain, params, bins=50):
    x_vals = np.linspace(domain[0], domain[1], 1000)
    f_vals = target_func(x_vals, *params)
    plt.figure(figsize=(8, 6))
    plt.hist(samples, bins=bins, density=True, alpha=0.6, color='g', label='Generated Samples')
    plt.plot(x_vals, f_vals, 'r-', linewidth=2, label='f(x) (Target Distribution)')
    plt.xlabel('x')
    plt.ylabel('Probability Density')
    plt.title('Generated Random Numbers vs Target Distribution')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

a, b, c = 0, 10, 6
n_samples = 10000
max_val = 2 / (b - a)
domain = (a, b)
np.random.seed(0)
random_numbers = accept_reject_sampling(triangular_distribution_function, domain, n_samples, max_val,
a, b, c)
plot_results(random_numbers, triangular_distribution_function, domain, (a, b, c))
```

5 Exercise 5

Implement a random number generator for the Rayleigh distribution with variance σ^2.

The Rayleigh distribution is a continuous probability distribution commonly used in signal processing, radar systems, and various statistical applications.

$$PDF : f(x; \sigma) = \frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}}, \quad x \geq 0,$$

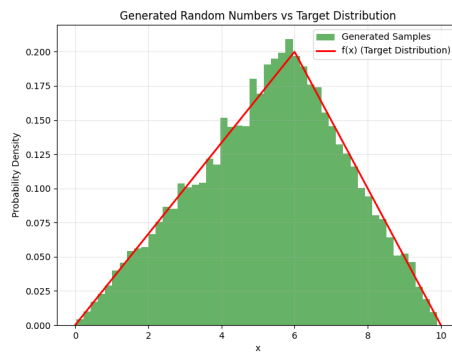$$CDF : F(x; \sigma) = 1 - e^{-\frac{x^2}{2\sigma^2}}, \quad x \geq 0.$$

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

def triangular_distribution_function(x, a, b, c):
    return np.where((a <= x) & (x < c), 2 * (x - a) / ((b - a) * (c - a)),
        np.where((c <= x) & (x <= b), 2 * (b - x) / ((b - a) * (b - c)), 0))

def accept_reject_sampling(target_func, domain, n_samples, max_val, *params):
    samples = []
    x_min, x_max = domain
    while len(samples) < n_samples:
        x = np.random.uniform(x_min, x_max)
        u = np.random.uniform(0, 1)
        if u <= target_func(x, *params) / max_val:
            samples.append(x)
    return np.array(samples)

def plot_results(samples, target_func, domain, params, bins=50):
    x_vals = np.linspace(domain[0], domain[1], 1000)
    f_vals = target_func(x_vals, *params)
    plt.figure(figsize=(8, 6))
    plt.hist(samples, bins=bins, density=True, alpha=0.6, color='g', label='Generated Samples')
    plt.plot(x_vals, f_vals, 'r-', linewidth=2, label='f(x) (Target Distribution)')
    plt.xlabel('x')
    plt.ylabel('Probability Density')
    plt.title('Generated Random Numbers vs Target Distribution')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

a, b, c = 0, 10, 6
n_samples = 10000
max_val = 2 / (b - a)
domain = (a, b)
np.random.seed(0)
random_numbers = accept_reject_sampling(triangular_distribution_function, domain, n_samples, max_val,
a, b, c)
plot_results(random_numbers, triangular_distribution_function, domain, (a, b, c))
```

6 Exercise 6

Simulate a bank with multiple counters where:

- Customer arrival times follow an exponential distribution.

- Service times at counters also follow an exponential distribution.

The simulation was conducted using the following input parameters:

- Number of Counters: 3

- Arrival Rate (λ_a): 2 customers per unit time

- Service Rate (λ_s): 3 customers per unit time

- Simulation Time: 5000 time units

Based on the input parameters, the simulation produced the following results:

- Average Waiting Time: 0.14 time units.

- Counter Usage: 40.03% , 19.41% ,6.67%

- Average Time in System: 0.34 time units.

7 Exercise 7

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

NUM_DRONES = 3
AREA_SIZE = 10
BATTERY_RANGE = 15
MAX_PAYLOAD = 5
DRONE_SPEED = 50
NUM_PACKAGES = 50


def generate_packages(num_packages, area_size):
    locations = np.random.uniform(0, area_size, size=(num_packages, 2))
    weights = np.random.uniform(1, MAX_PAYLOAD, num_packages)
    arrival_times = np.cumsum(np.random.exponential(scale=1, size=num_packages))
    return locations, weights, arrival_times


def calculate_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))


def assign_drone(drone_positions, drone_battery, package_location, package_weight):
    for idx, (position, battery) in enumerate(zip(drone_positions, drone_battery)):
        distance = calculate_distance(position, package_location)
        if battery >= 2 * distance and package_weight <= MAX_PAYLOAD:
            return idx, distance
    return None, None


def simulate_drones(num_drones, packages, battery_range, speed):
    locations, weights, arrival_times = packages
    drone_positions = np.zeros((num_drones, 2))
    drone_battery = np.full(num_drones, battery_range)
    total_distance = 0
    delivery_times = []
    current_time = 0

    for package_idx, package_time in enumerate(arrival_times):
        current_time = package_time
        package_location = locations[package_idx]
        package_weight = weights[package_idx]

        drone_idx, distance = assign_drone(drone_positions, drone_battery, package_location,
package_weight)
        if drone_idx is not None:
            travel_time = (2 * distance) / speed
            total_distance += 2 * distance
            delivery_times.append(current_time + travel_time)
            drone_positions[drone_idx] = package_location
            drone_battery[drone_idx] -= 2 * distance

    avg_delivery_time = np.mean(delivery_times)
    return avg_delivery_time, total_distance, locations


def plot_simulation(locations, area_size):
    plt.figure(figsize=(8, 8))
    plt.scatter(locations[:, 0], locations[:, 1], c='blue', label='Packages', alpha=0.7)
    plt.scatter(0, 0, c='red', label='Drone Base', s=100)
    plt.xlim(0, area_size)
    plt.ylim(0, area_size)
    plt.title('Package Delivery Map')
    plt.xlabel('X (km)')
    plt.ylabel('Y (km)')
    plt.legend()
    plt.grid(True)
    plt.show()


packages = generate_packages(NUM_PACKAGES, AREA_SIZE)
avg_delivery_time, total_distance, locations = simulate_drones(NUM_DRONES, packages, BATTERY_RANGE,
DRONE_SPEED)
print(f"Average Delivery Time: {avg_delivery_time:.2f} hours")
print(f"Total Distance Traveled by All Drones: {total_distance:.2f} km")
plot_simulation(locations, AREA_SIZE)
```
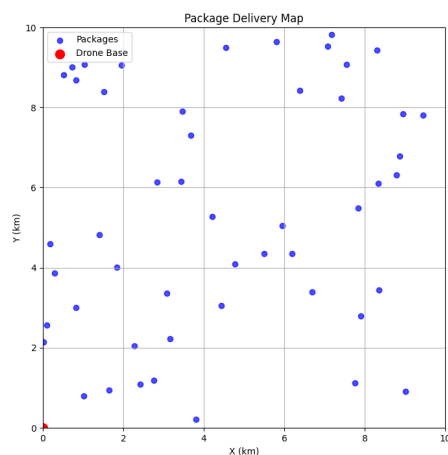

- Average Delivery Time: 11.06 hours

- Total Distance Traveled by All Drones: 41.04 km