SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 3rd semester

Mobile Communication Network (PGI20G03J)- Lab Manual

Lab 1: Mobile Communication System Simulator

Title: Implementation of a Simple Mobile Communication System Simulator

Aim: To implement a basic mobile communication system simulator that demonstrates the fundamental concepts of frequency reuse, handover, and mobility management.

Procedure:

1. **Environment Setup:** Choose a simulation environment (e.g., Python with libraries like `matplotlib` for visualization, or MATLAB).
2. **Cell Definition:** Define a cellular layout with multiple hexagonal or square cells. Assign a unique base station (BS) to each cell.
3. **Frequency Reuse:** Implement a frequency reuse pattern (e.g., K=3 or K=7) by assigning distinct sets of frequencies to non-adjacent cells.
4. **Mobile User (MU) Initialization:** Create several mobile users with initial positions within the cellular network.
5. **Mobility Model:** Implement a simple mobility model for MUs (e.g., random walk, straight line movement with random turns).
6. **Signal Strength Calculation:** For each MU, calculate the signal strength received from nearby base stations based on a propagation model (e.g., path loss exponent).
7. **Handover Mechanism:**
   - Define a handover threshold.
   - When an MU's signal strength from its current serving BS drops below the threshold, and the signal strength from a neighboring BS becomes stronger, trigger a handover.
   - Update the MU's serving BS.
8. **Mobility Management:** Track the location of MUs and their serving base stations.
9. **Simulation Loop:** Run the simulation for a defined period, updating MU positions, signal strengths, and performing handovers as necessary.
10. **Visualization (Optional but Recommended):** Plot the cellular layout, MU positions, and indicate serving BSs and handover events.

Source Code:

```
# Conceptual Python code structure for Lab 1

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

```
# --- System Parameters ---
NUM_CELLS = 7 # Example for a K=7 reuse pattern
CELL_RADIUS = 1000 # meters
NUM_MOBILE_USERS = 5
HANDOVER_THRESHOLD_DBM = -90 # dBm
PATH_LOSS_EXPONENT = 3.5

# --- Cell Class (Conceptual) ---
class Cell:
    def __init__(self, id, center_x, center_y, frequencies):
        self.id = id
        self.center = (center_x, center_y)
        self.frequencies = frequencies # Frequencies assigned to this cell's
BS

# --- Mobile User Class (Conceptual) ---
class MobileUser:
    def __init__(self, id, start_x, start_y):
        self.id = id
        self.position = np.array([start_x, start_y])
        self.serving_bs = None
        self.current_signal_strength = -float('inf')

    def move(self):
        # Simple random walk for demonstration
        self.position += np.random.uniform(-50, 50, 2) # Move by +/- 50m

    def calculate_signal_strength(self, bs_position):
        # Simple path loss model: P_rx = P_tx - 10 * n * log10(d)
        distance = np.linalg.norm(self.position - bs_position)
        if distance == 0:
            return 0 # At the BS
        # Placeholder for actual power calculation
        return - (10 * PATH_LOSS_EXPONENT * np.log10(distance)) # Simplified
dBm

# --- Simulation Logic ---
def run_simulation():
    # Initialize cells and frequency assignments (e.g., K=7 pattern)
    cells = [
        Cell(0, 0, 0, [1, 2, 3]), # Central cell
        Cell(1, 1500, 0, [4, 5, 6]), # Neighboring cell
        # ... more cells with assigned frequencies based on reuse pattern
    ]

    # Initialize mobile users
    mobile_users = [MobileUser(i, np.random.uniform(-500, 500),
np.random.uniform(-500, 500)) for i in range(NUM_MOBILE_USERS)]

    for t in range(100): # Simulate 100 time steps
        print(f"--- Time Step {t} ---")
        for mu in mobile_users:
            mu.move()
            best_bs = None
            max_signal = -float('inf')

            # Find the best serving BS
            for cell in cells:
                signal = mu.calculate_signal_strength(cell.center)
                if signal > max_signal:
                    max_signal = signal
                    best_bs = cell

            # Handover logic
            if mu.serving_bs is None or (max_signal >
mu.current_signal_strength and max_signal > HANDOVER_THRESHOLD_DBM):
```

```
                if mu.serving_bs != best_bs:
                    print(f"MU {mu.id}: Handover from Cell {mu.serving_bs.id
if mu.serving_bs else 'None'} to Cell {best_bs.id}")
                mu.serving_bs = best_bs
                mu.current_signal_strength = max_signal
            else:
                print(f"MU {mu.id}: Stays with Cell {mu.serving_bs.id}")

            # Mobility Management: Update location and serving BS
            # In a real simulator, this would involve updating a central
database or registry

        # Optional: Add visualization code here to plot current state
        # plt.clf() # Clear figure
        # # Plot cells, BSs, MUs, and connections
        # plt.pause(0.1) # Pause for animation

# run_simulation()
```

Input:

- Number of cells and their positions.
- Frequency reuse factor (K).
- Number of mobile users and their initial positions.
- Mobility model parameters (e.g., speed, direction changes).
- Handover threshold.
- Path loss exponent.

Expected Output:

- A simulation log showing mobile user movements, signal strength changes, and triggered handover events.
- (Optional) A graphical representation of the cellular network, mobile user trajectories, and real-time indication of serving base stations and handovers.
- Demonstration of how frequency reuse prevents interference in co-channel cells.
- Observation of seamless connectivity during handovers.

Lab 2: Network Performance Analysis (1G to 5G)

Title: Performance Analysis of 1G, 2G, 3G, 4G, and 5G Networks

Aim: To analyze and compare the performance differences between 1G, 2G, 3G, 4G, and 5G networks using network simulation tools.

Procedure:

1. **Tool Selection:** Choose a network simulation tool (e.g., NS-3, MATLAB with communication toolboxes, or a simplified custom simulator in Python).
2. **Network Model Definition:** For each generation (1G, 2G, 3G, 4G, 5G), define simplified network models that capture their key characteristics:
   - **1G (Analog):** Focus on voice communication, limited capacity.
   - **2G (GSM/CDMA):** Digital voice, SMS, basic data (GPRS/EDGE).
   - **3G (UMTS/CDMA2000):** Higher data rates, multimedia, circuit-switched and packet-switched.
   - **4G (LTE/LTE-A):** All-IP network, high data rates, low latency.
   - **5G (NR):** Massive MIMO, beamforming, network slicing, ultra-low latency, high bandwidth.
3. **Traffic Generation:** Simulate various traffic types relevant to each generation (e.g., voice calls for 1G/2G, web browsing for 3G/4G, streaming/IoT for 5G).
4. **Performance Metrics:** Define key performance indicators (KPIs) to measure:
   - **Throughput:** Data rate achieved.
   - **Latency:** Delay in data transmission.
   - **Packet Loss Rate:** Percentage of packets lost.
   - **Call Drop Rate (for voice):** Percentage of calls disconnected.
   - **Spectral Efficiency:** Data rate per unit of bandwidth.
   - **Energy Efficiency (for 5G):** Power consumption.
5. **Simulation Execution:** Run simulations for each network generation under varying load conditions.
6. **Data Collection and Analysis:** Collect the performance metrics from the simulations.
7. **Comparison and Visualization:** Compare the results across different generations using graphs and charts (e.g., bar charts for throughput comparison, line graphs for latency vs. load).

Source Code:

```
# Conceptual Python code structure for Lab 2 (using simplified models)

# This would involve detailed network models for each generation,
# which are too complex to fully implement here.
# Instead, consider using a simulation library like SimPy for event-based
simulation
# or a statistical approach to model performance.

# Example: Simplified throughput comparison
network_generations = ['1G', '2G', '3G', '4G', '5G']
avg_throughput_mbps = [0.01, 0.1, 2, 50, 1000] # Hypothetical average
throughputs

plt.figure(figsize=(10, 6))
plt.bar(network_generations, avg_throughput_mbps, color='skyblue')
plt.xlabel('Network Generation')
plt.ylabel('Average Throughput (Mbps)')
plt.title('Average Throughput Comparison Across Network Generations')
plt.grid(axis='y', linestyle='--')
```

```
plt.show()

# Similar conceptual code for latency, packet loss, etc.
```

Input:

- Network configuration parameters for each generation (e.g., bandwidth, modulation schemes, access technologies).
- Traffic load profiles (e.g., number of users, data rates, call durations).
- Simulation duration.

Expected Output:

- Graphs and tables illustrating the comparative performance of 1G, 2G, 3G, 4G, and 5G networks in terms of throughput, latency, packet loss, and other relevant KPIs.
- A report summarizing the observed trends and performance improvements across generations.

Lab 3: Wireless Propagation and Channel Modelling

Title: Simulation of Multipath Propagation and Fading Effects

Aim: To simulate the effects of multipath propagation and fading using MATLAB or Python, and analyze their impact on signal quality and coverage.

Procedure:

1. **Environment Setup:** Use MATLAB or Python (with libraries like `numpy`, `scipy`, `matplotlib`).
2. **Signal Generation:** Generate a simple baseband signal (e.g., a complex exponential or a modulated signal).
3. **Multipath Channel Model:**
   - Implement a multipath channel model. This can be a simple two-ray model, a tapped delay line model, or a more complex Rayleigh or Rician fading channel.
   - For a simple two-ray model: consider a direct path and a reflected path with different delays and attenuations.
   - For fading: generate random complex gains for each path based on Rayleigh or Rician distributions.
4. **Signal Propagation:** Pass the generated signal through the simulated multipath fading channel. This involves convolving the signal with the channel impulse response or applying complex gains.
5. **Noise Addition:** Add Additive White Gaussian Noise (AWGN) to the received signal.
6. **Received Signal Analysis:**
   - Plot the transmitted and received signals (time domain and frequency domain).
   - Calculate the Signal-to-Noise Ratio (SNR) or Signal-to-Interference-plus-Noise Ratio (SINR).
   - Analyze the impact of fading on the signal envelope and phase.
7. **Performance Evaluation:**
   - Measure the Bit Error Rate (BER) or Symbol Error Rate (SER) for different SNR values.
   - Observe how fading causes deep fades and impacts the reliability of communication.
8. **Visualization:** Plot the fading envelope, constellation diagrams (if modulation is used), and BER vs. SNR curves.

Source Code:

```
# Conceptual Python code structure for Lab 3

import numpy as np
import matplotlib.pyplot as plt

# --- System Parameters ---
CARRIER_FREQ = 2.4e9 # Hz
SAMPLING_RATE = 1e6 # Hz
NUM_SAMPLES = 1000
SNR_DB = 10 # dB

# --- Rayleigh Fading Channel (Conceptual) ---
def rayleigh_fading_channel(signal, num_paths=5):
    faded_signal = np.zeros_like(signal, dtype=complex)
    for _ in range(num_paths):
        # Generate random complex gain for each path (Rayleigh distributed)
        h = (np.random.randn() + 1j * np.random.randn()) / np.sqrt(2)
        # Apply delay (simplified, actual delay involves interpolation)
```

```
        delay_samples = np.random.randint(0, 10)
        delayed_signal = np.roll(signal, delay_samples)
        faded_signal += h * delayed_signal
    return faded_signal / np.sqrt(num_paths) # Normalize

# --- Simulation Logic ---
# Generate a simple BPSK signal
data_bits = np.random.randint(0, 2, NUM_SAMPLES)
bpsk_signal = 2 * data_bits - 1 # Map 0 to -1, 1 to 1

# Apply Rayleigh fading
received_signal_faded = rayleigh_fading_channel(bpsk_signal)

# Add AWGN
noise_power = np.var(received_signal_faded) / (10**(SNR_DB/10))
noise = np.sqrt(noise_power / 2) * (np.random.randn(NUM_SAMPLES) + 1j *
np.random.randn(NUM_SAMPLES))
received_signal_noisy = received_signal_faded + noise

# Plotting (conceptual)
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(np.abs(received_signal_faded), label='Faded Signal Envelope')
plt.title('Received Signal Envelope with Rayleigh Fading')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')
plt.legend()
plt.grid(True)

plt.subplot(2, 1, 2)
plt.plot(np.real(received_signal_noisy[:100]), label='Noisy Received Signal
(Real Part)')
plt.title('Noisy Received Signal (First 100 Samples)')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Further analysis would involve demodulation and BER calculation
```

Input:

- Signal parameters (e.g., frequency, modulation type).
- Channel model parameters (e.g., number of paths, delays, fading statistics).
- SNR values for BER analysis.

Expected Output:

- Plots showing the effects of multipath propagation (e.g., delayed and attenuated signal replicas).
- Plots illustrating the random fluctuations of the signal envelope due to fading.
- BER vs. SNR curves demonstrating the performance degradation caused by fading.
- Analysis of how fading affects signal quality and coverage, leading to potential outages.

Lab 4: Multiple Access Techniques Comparison

Title: Performance Evaluation of FDMA, TDMA, CDMA, and OFDMA

Aim: To develop a simulator to compare and evaluate the performance of FDMA, TDMA, CDMA, and OFDMA in terms of spectral efficiency and interference management.

Procedure:

1. **Environment Setup:** Use MATLAB or Python for simulation.
2. **System Model Definition:** For each multiple access technique, define a simplified system model:
   - **FDMA (Frequency Division Multiple Access):** Divide the total bandwidth into non-overlapping frequency channels.
   - **TDMA (Time Division Multiple Access):** Divide the time into slots, with each user transmitting in assigned slots.
   - **CDMA (Code Division Multiple Access):** All users transmit simultaneously over the same frequency, separated by unique spreading codes.
   - **OFDMA (Orthogonal Frequency Division Multiple Access):** Divide the bandwidth into orthogonal subcarriers, with users assigned specific subcarriers.
3. **User Allocation:** Implement mechanisms to allocate resources (frequency channels, time slots, spreading codes, subcarriers) to multiple users.
4. **Interference Modeling:**
   - **FDMA/TDMA/OFDMA:** Model inter-channel/inter-slot/inter-subcarrier interference (ideally minimal if orthogonal).
   - **CDMA:** Model multi-user interference (MUI) due to non-perfect orthogonality of codes.
5. **Spectral Efficiency Calculation:** For each technique, calculate spectral efficiency (bits/s/Hz) under varying user loads and interference levels.
6. **Throughput Measurement:** Measure the total system throughput and individual user throughput.
7. **Simulation Scenarios:**
   - Vary the number of active users.
   - Vary the signal-to-interference ratio (SIR) or SNR.
8. **Data Collection and Analysis:** Collect throughput and spectral efficiency data.
9. **Comparison and Visualization:** Plot and compare the performance metrics for each technique.

Source Code:

```
# Conceptual Python code structure for Lab 4

import numpy as np
import matplotlib.pyplot as plt

# --- Simulation Parameters ---
TOTAL_BANDWIDTH = 10e6 # Hz
NUM_USERS_RANGE = range(1, 21) # Simulate from 1 to 20 users
DATA_RATE_PER_USER_BPS = 100e3 # 100 kbps

# --- FDMA Simulation (Conceptual) ---
def simulate_FDMA(num_users, bandwidth_per_user):
    if num_users * bandwidth_per_user > TOTAL_BANDWIDTH:
        return 0 # Not enough bandwidth
    spectral_efficiency = (num_users * DATA_RATE_PER_USER_BPS) /
TOTAL_BANDWIDTH
    return spectral_efficiency
```

```
# --- TDMA Simulation (Conceptual) ---
def simulate_TDMA(num_users, time_slot_duration_per_user):
    # Assuming fixed frame duration, varying time slots
    spectral_efficiency = (num_users * DATA_RATE_PER_USER_BPS) /
TOTAL_BANDWIDTH # Simplified
    return spectral_efficiency

# --- CDMA Simulation (Conceptual) ---
def simulate_CDMA(num_users):
    # Performance degrades with more users due to MUI
    # This is a highly simplified model
    interference_factor = 1 + (num_users - 1) * 0.1 # Simple linear increase
in interference
    effective_data_rate = DATA_RATE_PER_USER_BPS / interference_factor
    spectral_efficiency = (num_users * effective_data_rate) / TOTAL_BANDWIDTH
    return spectral_efficiency

# --- OFDMA Simulation (Conceptual) ---
def simulate_OFDMA(num_users, subcarriers_per_user):
    # Assuming orthogonal subcarriers, high efficiency
    spectral_efficiency = (num_users * DATA_RATE_PER_USER_BPS) /
TOTAL_BANDWIDTH # Simplified
    return spectral_efficiency

# --- Running Simulations ---
spectral_efficiencies_FDMA = []
spectral_efficiencies_TDMA = []
spectral_efficiencies_CDMA = []
spectral_efficiencies_OFDMA = []

for num_users in NUM_USERS_RANGE:
    # Assuming fixed allocation for simplicity
    spectral_efficiencies_FDMA.append(simulate_FDMA(num_users,
TOTAL_BANDWIDTH / len(NUM_USERS_RANGE)))
    spectral_efficiencies_TDMA.append(simulate_TDMA(num_users, 1)) #
Placeholder
    spectral_efficiencies_CDMA.append(simulate_CDMA(num_users))
    spectral_efficiencies_OFDMA.append(simulate_OFDMA(num_users, 1)) #
Placeholder

# --- Plotting Results ---
plt.figure(figsize=(12, 7))
plt.plot(NUM_USERS_RANGE, spectral_efficiencies_FDMA, marker='o',
label='FDMA')
plt.plot(NUM_USERS_RANGE, spectral_efficiencies_TDMA, marker='s',
label='TDMA')
plt.plot(NUM_USERS_RANGE, spectral_efficiencies_CDMA, marker='^',
label='CDMA')
plt.plot(NUM_USERS_RANGE, spectral_efficiencies_OFDMA, marker='x',
label='OFDMA')

plt.xlabel('Number of Users')
plt.ylabel('Spectral Efficiency (bps/Hz)')
plt.title('Spectral Efficiency Comparison of Multiple Access Techniques')
plt.legend()
plt.grid(True)
plt.show()
```

Input:

- Total available bandwidth.
- Number of users.
- Spreading codes (for CDMA).

- Subcarrier allocation (for OFDMA).
- Interference models for each technique.

Expected Output:

- Graphs comparing the spectral efficiency of FDMA, TDMA, CDMA, and OFDMA as a function of the number of users or load.
- Analysis of how each technique manages interference and its impact on performance.
- Discussion on the trade-offs between the techniques (e.g., complexity vs. capacity).

Lab 5: Cellular Network Layout Design and Simulation

Title: Design and Simulation of a Cellular Network Layout

Aim: To design and simulate a cellular network layout considering cell planning, frequency reuse, and interference management strategies using specialized software.

Procedure:

1. **Tool Selection:** Utilize specialized cellular network planning software (e.g., OpenCellular, Atoll, or a custom simulator built in Python/MATLAB).
2. **Area Definition:** Define the geographical area for the cellular network (e.g., a city map, a rural area).
3. **Cell Planning:**
    - Place base stations (BSs) strategically to provide adequate coverage.
    - Consider different cell shapes (hexagonal, circular) and sizes.
    - Determine the optimal number of cells required to cover the area.
4. **Frequency Reuse Implementation:** Apply a chosen frequency reuse pattern (e.g., K=3, K=7) to assign frequency channels to cells, minimizing co-channel interference.
5. **Propagation Model:** Integrate a realistic propagation model (e.g., Okumura-Hata, COST 231-Hata) to estimate signal strength and coverage.
6. **Interference Analysis:**
    - Identify potential sources of co-channel interference and adjacent channel interference.
    - Calculate Carrier-to-Interference Ratio (C/I) for different locations.
7. **Coverage Analysis:** Generate coverage maps showing signal strength levels across the area.
8. **Capacity Planning:** Estimate the network capacity based on the number of users and traffic demand.
9. **Optimization (Optional):** Experiment with different BS placements, antenna heights, and power levels to optimize coverage and minimize interference.
10. **Visualization and Reporting:** Generate reports and visualizations of the network layout, coverage, and interference levels.

Source Code:

```
# Conceptual Python code structure for Lab 5 (simplified for demonstration)

import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import Voronoi, voronoi_vertices

# --- System Parameters ---
AREA_SIZE = 1000 # meters x 1000 meters
NUM_BASE_STATIONS = 10
CELL_RADIUS = 200 # meters (approximate for visualization)
FREQUENCY_REUSE_FACTOR = 3 # K=3 example

# --- Base Station Class ---
class BaseStation:
    def __init__(self, id, x, y, assigned_frequencies):
        self.id = id
        self.position = np.array([x, y])
        self.frequencies = assigned_frequencies

# --- Simple Propagation Model (Path Loss) ---
def calculate_signal_strength(tx_power_dbm, distance_m,
path_loss_exponent=3.0):
```

```
    # Simplified path loss model
    if distance_m < 1: # Avoid log(0)
        distance_m = 1
    return tx_power_dbm - (10 * path_loss_exponent * np.log10(distance_m))

# --- Simulation Logic ---
def design_cellular_layout():
    # Randomly place base stations for demonstration
    bs_positions = np.random.rand(NUM_BASE_STATIONS, 2) * AREA_SIZE

    # Assign frequencies based on a simple reuse pattern (conceptual)
    # In a real scenario, this would be more sophisticated (e.g., graph
coloring)
    frequency_groups = [
        [1, 2, 3], # Group A
        [4, 5, 6], # Group B
        [7, 8, 9]  # Group C
    ]
    base_stations = []
    for i, pos in enumerate(bs_positions):
        freq_group_index = i % FREQUENCY_REUSE_FACTOR
        base_stations.append(BaseStation(i, pos[0], pos[1],
frequency_groups[freq_group_index]))

    # --- Coverage Visualization ---
    x_coords = np.linspace(0, AREA_SIZE, 50)
    y_coords = np.linspace(0, AREA_SIZE, 50)
    coverage_map = np.zeros((len(y_coords), len(x_coords)))

    for i, y in enumerate(y_coords):
        for j, x in enumerate(x_coords):
            max_signal = -float('inf')
            for bs in base_stations:
                distance = np.linalg.norm(np.array([x, y]) - bs.position)
                signal = calculate_signal_strength(40, distance) # Assuming
40 dBm Tx power
                if signal > max_signal:
                    max_signal = signal
            coverage_map[i, j] = max_signal

    plt.figure(figsize=(10, 8))
    plt.imshow(coverage_map, extent=[0, AREA_SIZE, 0, AREA_SIZE],
origin='lower', cmap='viridis')
    plt.colorbar(label='Signal Strength (dBm)')
    plt.scatter(bs_positions[:, 0], bs_positions[:, 1], color='red',
marker='^', s=100, label='Base Stations')

    # Add cell boundaries (conceptual - using Voronoi for illustration)
    if NUM_BASE_STATIONS > 2:
        vor = Voronoi(bs_positions)
        plt.plot(vor.vertices[:,0], vor.vertices[:,1], 'ko', markersize=2)
        for simplex in vor.ridge_vertices:
            simplex = np.asarray(simplex)
            if np.all(simplex >= 0):
                plt.plot(vor.vertices[simplex,0], vor.vertices[simplex,1],
'k-')

    plt.title('Cellular Network Coverage Map')
    plt.xlabel('X-coordinate (m)')
    plt.ylabel('Y-coordinate (m)')
    plt.legend()
    plt.grid(True)
    plt.show()

# design_cellular_layout()
```

Input:

- Geographical area dimensions.
- Number of base stations.
- Base station locations.
- Frequency reuse pattern (e.g., K=3, K=7).
- Propagation model parameters (e.g., path loss exponent, antenna height).
- Traffic demand (for capacity planning).

Expected Output:

- A visual representation of the cellular network layout, including base station locations and cell boundaries.
- Coverage maps indicating signal strength distribution.
- Interference maps highlighting areas with high interference.
- Analysis of how different cell planning and frequency reuse strategies impact coverage and interference.

Lab 6: Resource Allocation Optimization

Title: Algorithm Development for Resource Allocation Optimization in Cellular Networks

Aim: To develop an algorithm to optimize the allocation of resources (frequency channels, time slots) in a cellular network to maximize capacity and coverage while minimizing interference.

Procedure:

1. **Problem Formulation:** Clearly define the resource allocation problem. This involves:
   - **Objective Function:** Maximize total system throughput, maximize number of served users, minimize interference, etc.
   - **Constraints:** Available frequency channels, time slots, power limits, QoS requirements (e.g., minimum data rate).
2. **Network Model:** Use a simplified cellular network model (e.g., a few cells, multiple users).
3. **Algorithm Design:** Choose an optimization approach:
   - **Heuristic Algorithms:** Greedy algorithms, iterative refinement.
   - **Optimization Techniques:** Linear programming, integer linear programming (if applicable), genetic algorithms, simulated annealing.
   - **For simpler cases:** A greedy approach where resources are assigned to users with the best channel conditions or highest demand first.
4. **Resource Allocation Implementation:** Implement the chosen algorithm to assign frequency channels or time slots to users.
5. **Performance Evaluation:**
   - Calculate the achieved capacity (total throughput).
   - Measure the average SINR for users.
   - Quantify the interference levels.
   - Evaluate coverage based on minimum SINR requirements.
6. **Comparison:** Compare the performance of the optimized allocation with a non-optimized (e.g., random) allocation.
7. **Scenario Variation:** Test the algorithm under different network loads, user distributions, and channel conditions.

Source Code:

```
# Conceptual Python code structure for Lab 6

import numpy as np

# --- System Parameters ---
NUM_CHANNELS = 10
NUM_USERS = 20
CHANNEL_BANDWIDTH = 200e3 # Hz per channel
MAX_TX_POWER_DBM = 30 # dBm

# --- User Class (Simplified) ---
class User:
    def __init__(self, id, desired_data_rate_bps):
        self.id = id
        self.desired_data_rate = desired_data_rate_bps
        self.assigned_channel = None
        self.sinr = 0 # Placeholder for calculated SINR

# --- Channel Conditions (Conceptual) ---
# Simulate channel gains/path loss from a base station to each user for each
channel
# Higher value means better channel
```

```
channel_gains = np.random.rand(NUM_USERS, NUM_CHANNELS) * 10 # Random gains
0-10 (conceptual)

# --- Simple Greedy Resource Allocation Algorithm ---
def greedy_resource_allocation(users, channel_gains):
    allocated_channels = [None] * NUM_CHANNELS # Track if a channel is used
    user_assignments = {} # User ID -> assigned channel

    # Sort users by some priority (e.g., desired data rate, or just process
in order)
    # For simplicity, we'll iterate through users and assign the best
available channel

    for user_id, user in enumerate(users):
        best_channel = -1
        max_gain = -1

        # Find the best available channel for the current user
        for channel_id in range(NUM_CHANNELS):
            if allocated_channels[channel_id] is None: # If channel is free
                if channel_gains[user_id, channel_id] > max_gain:
                    max_gain = channel_gains[user_id, channel_id]
                    best_channel = channel_id

        if best_channel != -1:
            user.assigned_channel = best_channel
            allocated_channels[best_channel] = user.id
            user_assignments[user.id] = best_channel
            # In a real scenario, calculate SINR based on assigned channel
and interference
            user.sinr = max_gain * 10 # Conceptual SINR
            print(f"User {user.id} assigned to Channel {best_channel} with
gain {max_gain:.2f}")
        else:
            print(f"User {user.id} could not be assigned a channel.")

    return user_assignments, users

# --- Simulation Execution ---
users = [User(i, np.random.uniform(50e3, 200e3)) for i in range(NUM_USERS)] #
Users with random data rates

print("--- Running Greedy Resource Allocation ---")
assignments, updated_users = greedy_resource_allocation(users, channel_gains)

# --- Performance Evaluation (Conceptual) ---
total_served_users = sum(1 for u in updated_users if u.assigned_channel is
not None)
total_capacity = sum(u.desired_data_rate for u in updated_users if
u.assigned_channel is not None) # Simplified
avg_sinr = np.mean([u.sinr for u in updated_users if u.assigned_channel is
not None])

print(f"\nTotal served users: {total_served_users}")
print(f"Total theoretical capacity served: {total_capacity / 1e6:.2f} Mbps")
print(f"Average conceptual SINR for served users: {avg_sinr:.2f}")

# Compare with a random allocation (conceptual)
# In a real lab, you'd implement a random allocation and compare metrics.
```

Input:

- Number of available frequency channels/time slots.
- Number of users and their QoS requirements (e.g., desired data rate).

- Channel state information (e.g., channel gains, path loss) between users and base stations.
- Interference model.

Expected Output:

- The optimized resource allocation scheme (which user gets which resource).
- Metrics demonstrating the improvement in capacity, coverage, and interference reduction compared to a non-optimized approach.
- Analysis of the algorithm's performance under different network conditions.

Lab 7: Interference Mitigation Techniques

Title: Implementation of Interference Mitigation Techniques in a Simulated Wireless Network

Aim: To implement interference mitigation techniques such as power control, adaptive beamforming, or interference cancellation in a simulated wireless network environment.

Procedure:

1. **Environment Setup:** Use MATLAB or Python for simulation.
2. **Network Model:** Set up a simple wireless network scenario with multiple interfering users/cells.
    - **Scenario 1 (Power Control):** Multiple users transmitting to a single base station (uplink) or a single base station transmitting to multiple users (downlink), with varying distances and channel conditions.
    - **Scenario 2 (Adaptive Beamforming):** A base station with multiple antennas serving multiple users, where interference is a concern.
    - **Scenario 3 (Interference Cancellation):** A receiver trying to decode a desired signal in the presence of strong interference.
3. **Interference Modeling:** Model the interference caused by co-channel users or adjacent cells.
4. **Technique Implementation:**
    - **Power Control:** Implement an iterative power control algorithm (e.g., distributed power control) where users adjust their transmit power to meet a target SINR while minimizing interference.
    - **Adaptive Beamforming:** Implement a beamforming algorithm (e.g., Minimum Variance Distortionless Response - MVDR, or simple array weighting) to direct antenna beams towards desired users and nulls towards interferers.
    - **Interference Cancellation:** Implement a successive interference cancellation (SIC) or parallel interference cancellation (PIC) scheme at the receiver to remove known interference components.
5. **Performance Evaluation:**
    - Measure the SINR improvement with the mitigation technique.
    - Evaluate throughput or BER improvement.
    - Analyze the reduction in interference levels.
6. **Comparison:** Compare the performance with and without the interference mitigation technique.
7. **Visualization:** Plot SINR improvements, beam patterns (for beamforming), or signal constellations.

Source Code:

```
# Conceptual Python code structure for Lab 7 (Example: Simple Power Control)

import numpy as np
import matplotlib.pyplot as plt

# --- System Parameters ---
NUM_USERS = 5
TARGET_SINR_DB = 10 # dB
NOISE_POWER_DBM = -100 # dBm
MAX_ITERATIONS = 50
CONVERGENCE_THRESHOLD = 0.1 # dB

# Convert to linear scale
TARGET_SINR_LINEAR = 10**(TARGET_SINR_DB / 10)
```

```
NOISE_POWER_LINEAR = 10**(NOISE_POWER_DBM / 10)

# --- Channel Gains (Conceptual: User to BS) ---
# Each row is a user, each column is a gain from another user (interference)
or self (desired)
# For simplicity, let's assume a single BS and users interfering with each
other
# Diagonal elements are desired channel gains, off-diagonal are interference
gains
channel_gains = np.random.rand(NUM_USERS, NUM_USERS) * 0.1 +
np.eye(NUM_USERS) * 1 # Self-gain is stronger

# --- Initial Transmit Powers (Linear) ---
transmit_powers = np.ones(NUM_USERS) * 1e-3 # 1 mW initial power for all
users

# --- Power Control Algorithm (Iterative) ---
def power_control_iteration(tx_powers, channel_gains, target_sinr,
noise_power):
    new_tx_powers = np.zeros_like(tx_powers)
    current_sinrs_db = []

    for i in range(NUM_USERS):
        desired_signal = tx_powers[i] * channel_gains[i, i]
        interference = 0
        for j in range(NUM_USERS):
            if i != j:
                interference += tx_powers[j] * channel_gains[i, j]

        current_sinr_linear = desired_signal / (interference + noise_power)
        current_sinrs_db.append(10 * np.log10(current_sinr_linear))

        # Update rule for power control (e.g., based on target SINR)
        new_tx_powers[i] = tx_powers[i] * (target_sinr / current_sinr_linear)

    return new_tx_powers, current_sinrs_db

# --- Simulation Execution ---
sinr_history = []
for iteration in range(MAX_ITERATIONS):
    old_transmit_powers = np.copy(transmit_powers)
    transmit_powers, current_sinrs = power_control_iteration(transmit_powers,
channel_gains, TARGET_SINR_LINEAR, NOISE_POWER_LINEAR)
    sinr_history.append(current_sinrs)

    # Check for convergence
    if iteration > 0:
        avg_sinr_diff = np.mean(np.abs(np.array(sinr_history[-1]) -
np.array(sinr_history[-2])))
        if avg_sinr_diff < CONVERGENCE_THRESHOLD:
            print(f"Converged at iteration {iteration}")
            break

print("\n--- Final Transmit Powers (mW) ---")
for i, p in enumerate(transmit_powers):
    print(f"User {i}: {p * 1000:.2f} mW")

print("\n--- Final SINRs (dB) ---")
for i, sinr_db in enumerate(current_sinrs):
    print(f"User {i}: {sinr_db:.2f} dB (Target: {TARGET_SINR_DB} dB)")

# Plot SINR convergence
sinr_history_array = np.array(sinr_history)
plt.figure(figsize=(10, 6))
for i in range(NUM_USERS):
    plt.plot(sinr_history_array[:, i], label=f'User {i} SINR')
```

```
plt.axhline(y=TARGET_SINR_DB, color='r', linestyle='--', label='Target SINR')
plt.xlabel('Iteration')
plt.ylabel('SINR (dB)')
plt.title('SINR Convergence with Power Control')
plt.legend()
plt.grid(True)
plt.show()
```

Input:

- Network topology (e.g., number of users, base stations).
- Channel conditions (e.g., path loss, fading coefficients).
- Interference sources and their characteristics.
- Target SINR (for power control).
- Antenna array configuration (for beamforming).

Expected Output:

- Demonstration of improved SINR or reduced BER/throughput with the mitigation technique.
- Plots showing the convergence of power levels (for power control) or beam patterns (for beamforming).
- Analysis of the effectiveness and trade-offs of the chosen interference mitigation technique.

Lab 8: OSI Protocol Stack Implementation

Title: Implementation of a Simplified OSI Protocol Stack

Aim: To implement a simplified version of the OSI protocol stack, including physical, data link, network, and transport layers, and demonstrate data transmission between mobile devices.

Procedure:

1. **Environment Setup:** Use Python or C++ for implementation.
2. **Layer Definition:** Define classes or modules for each layer:
   - **Physical Layer:** Simulates bit transmission (e.g., converting bytes to bit streams, adding noise/errors).
   - **Data Link Layer:** Handles framing, error detection (e.g., CRC), and flow control.
   - **Network Layer:** Handles addressing (e.g., IP-like addresses) and routing.
   - **Transport Layer:** Handles end-to-end communication, segmentation, reassembly, and basic reliability (e.g., ACK/NACK).
3. **Data Flow:**
   - **Encapsulation:** Implement the process of adding headers at each layer as data flows down the stack (from application to physical).
   - **Decapsulation:** Implement the process of removing headers and passing data up the stack at the receiver.
4. **Inter-Layer Communication:** Define clear interfaces for communication between adjacent layers (e.g., service access points).
5. **Simulation Scenario:**
   - Set up two "mobile devices" (simulated as separate processes or threads, or just functions calling each other).
   - Simulate sending a message from one device to another through the implemented stack.
6. **Error Simulation (Optional):** Introduce errors (e.g., bit errors at physical layer, packet loss at network layer) to test error detection and retransmission mechanisms.
7. **Verification:** Print the data at each layer during encapsulation and decapsulation to verify correct operation.

Source Code:

```
# Conceptual Python code structure for Lab 8

# --- Constants ---
MAX_PAYLOAD_SIZE = 10 # bytes
CRC_POLYNOMIAL = 0x1021 # Example CRC-16 polynomial

# --- Helper Function for CRC (Simplified) ---
def calculate_crc(data_bytes):
    # This is a highly simplified placeholder for CRC calculation
    # In a real implementation, use a proper CRC algorithm
    return sum(data_bytes) % 256 # Simple checksum as a stand-in

# --- Layer Classes ---

class PhysicalLayer:
    def transmit(self, bits):
        print(f"[Physical Layer] Transmitting {len(bits)} bits...")
        # Simulate transmission, potentially adding noise/errors
        return bits # No errors for this simplified example
```

```
    def receive(self, bits):
        print(f"[Physical Layer] Receiving {len(bits)} bits...")
        return bits

class DataLinkLayer:
    def __init__(self, physical_layer):
        self.physical_layer = physical_layer

    def send_frame(self, data_bytes):
        # Framing: Add header (e.g., length, CRC)
        frame_header = len(data_bytes).to_bytes(1, 'big') # 1 byte for length
        crc = calculate_crc(data_bytes).to_bytes(1, 'big') # 1 byte for CRC
        frame = b"START" + frame_header + data_bytes + crc + b"END"
        print(f"[Data Link Layer] Sending frame: {frame}")
        bits = ''.join(format(byte, '08b') for byte in frame) # Convert to
bit string
        self.physical_layer.transmit(bits)

    def receive_frame(self):
        # This would involve listening on the physical layer and parsing
incoming bits
        # For simplicity, we'll assume the physical layer returns the
original bits
        received_bits = self.physical_layer.receive("") # Placeholder for
actual reception
        # In a real scenario, parse bits back to bytes and validate frame
        # For this example, we'll just return a dummy data
        print("[Data Link Layer] Frame received and processed.")
        return b"Simulated Data Link Payload" # Placeholder

class NetworkLayer:
    def __init__(self, datalink_layer, own_address):
        self.datalink_layer = datalink_layer
        self.own_address = own_address

    def send_packet(self, destination_address, payload_bytes):
        # Add IP-like header (source, destination, protocol)
        packet_header = f"{self.own_address}-{destination_address}-
TCP".encode('utf-8')
        packet = packet_header + payload_bytes
        print(f"[Network Layer] Sending packet from {self.own_address} to
{destination_address}: {packet}")
        self.datalink_layer.send_frame(packet)

    def receive_packet(self):
        received_frame_payload = self.datalink_layer.receive_frame()
        # Parse packet header to extract payload
        print(f"[Network Layer] Packet received: {received_frame_payload}")
        return received_frame_payload # Placeholder for actual payload
extraction

class TransportLayer:
    def __init__(self, network_layer, own_port):
        self.network_layer = network_layer
        self.own_port = own_port

    def send_segment(self, destination_address, destination_port, message):
        # Segmentation if message is large (not implemented here)
        # Add TCP/UDP-like header (source port, destination port)
        segment_header = f"{self.own_port}-{destination_port}".encode('utf-
8')
        segment = segment_header + message.encode('utf-8')
        print(f"[Transport Layer] Sending segment from port {self.own_port}
to {destination_port}: {segment}")
        self.network_layer.send_packet(destination_address, segment)
```

```
    def receive_segment(self):
        received_packet_payload = self.network_layer.receive_packet()
        # Parse segment header to extract message
        print(f"[Transport Layer] Segment received:
{received_packet_payload}")
        return received_packet_payload # Placeholder

# --- Simulation Flow ---
if __name__ == "__main__":
    # Device A
    physical_a = PhysicalLayer()
    datalink_a = DataLinkLayer(physical_a)
    network_a = NetworkLayer(datalink_a, "192.168.1.10")
    transport_a = TransportLayer(network_a, 12345)

    # Device B (simplified, directly receiving from A's physical layer for
demonstration)
    physical_b = PhysicalLayer()
    datalink_b = DataLinkLayer(physical_b)
    network_b = NetworkLayer(datalink_b, "192.168.1.20")
    transport_b = TransportLayer(network_b, 8080)

    # Simulate sending a message from Device A to Device B
    print("\n--- Device A: Sending Message ---")
    message_to_send = "Hello from Device A!"
    transport_a.send_segment("192.168.1.20", 8080, message_to_send)

    print("\n--- Device B: Receiving Message (Conceptual) ---")
    # In a real scenario, Device B's physical layer would receive bits,
    # then pass them up through its own stack.
    # For this simplified example, we'll simulate the reception flow.
    received_data_at_b = transport_b.receive_segment()
    print(f"\n[Application Layer] Device B received:
{received_data_at_b.decode('utf-8')}")
```

Input:

- A message string to be transmitted.
- Source and destination addresses/ports.
- (Optional) Parameters to introduce errors (e.g., bit error rate).

Expected Output:

- Console output showing the encapsulation process at the sender (data being passed down through layers with headers added).
- Console output showing the decapsulation process at the receiver (headers being removed and data passed up).
- The original message successfully received at the destination.
- (Optional) Demonstration of error detection and retransmission if errors are introduced.

Lab 9: GSM Protocol Stack Simulator

Title: Development of a GSM Protocol Stack Simulator

Aim: To develop a GSM protocol stack simulator to handle functions such as call setup, SMS messaging, and handover between base stations.

Procedure:

1. **Environment Setup:** Use Python or Java for simulation.
2. **GSM Architecture Simplification:** Focus on key components relevant to call setup, SMS, and handover:
   - Mobile Station (MS) / User Equipment (UE)
   - Base Transceiver Station (BTS)
   - Base Station Controller (BSC)
   - Mobile Switching Center (MSC)
   - Home Location Register (HLR) / Visitor Location Register (VLR)
3. **Protocol Layer Implementation (Simplified):**
   - **Radio Resource (RR) Management:** Handles channel allocation, handover.
   - **Mobility Management (MM):** Handles location updates, authentication.
   - **Call Management (CM):** Handles call setup, release.
   - **Short Message Service (SMS) Layer:** Handles SMS transmission.
4. **Scenario Implementation:**
   - **Call Setup:** Simulate the sequence of messages exchanged between MS, BTS, BSC, and MSC to establish a voice call.
   - **SMS Messaging:** Simulate the process of sending and receiving an SMS.
   - **Handover:** Simulate an MS moving between two cells, triggering a handover procedure involving BTSs and BSC.
5. **State Machines:** Implement simplified state machines for MS and network elements to manage the various call/SMS/mobility states.
6. **Message Exchange:** Define and simulate the exchange of control messages (e.g., `CM SERVICE REQUEST`, `ASSIGNMENT COMMAND`, `HANDOVER COMMAND`) between the simulated entities.
7. **Verification:** Log the message flow and state changes to verify correct protocol operation.

Source Code:

```
# Conceptual Python code structure for Lab 9

# --- Simplified GSM Entities ---

class MobileStation:
    def __init__(self, id, current_cell_id):
        self.id = id
        self.current_cell_id = current_cell_id
        self.call_state = "IDLE" # IDLE, CALLING, ACTIVE, HANGUP
        self.location_area = None
        print(f"MS {self.id} initialized in Cell {self.current_cell_id}")

    def initiate_call(self, target_number):
        print(f"MS {self.id}: Initiating call to {target_number}...")
        self.call_state = "CALLING"
        # Simulate sending CM SERVICE REQUEST to network
        return "CM_SERVICE_REQUEST"

    def receive_call(self, caller_id):
```

```
        print(f"MS {self.id}: Incoming call from {caller_id}")
        self.call_state = "ACTIVE"
        return "CALL_ACCEPTED"

    def send_sms(self, recipient, message):
        print(f"MS {self.id}: Sending SMS to {recipient}: '{message}'")
        # Simulate sending SMS message to network
        return "SMS_DELIVERY_REQUEST"

    def move_to_cell(self, new_cell_id):
        print(f"MS {self.id}: Moving from Cell {self.current_cell_id} to Cell
{new_cell_id}")
        self.current_cell_id = new_cell_id
        # Simulate location update or potential handover trigger
        return "LOCATION_UPDATE_REQUEST"

class BaseStationController:
    def __init__(self, id, controlled_cells):
        self.id = id
        self.controlled_cells = controlled_cells # List of cell IDs
        print(f"BSC {self.id} initialized, controlling cells:
{controlled_cells}")

    def handle_cm_service_request(self, ms_id, call_target):
        print(f"BSC {self.id}: Received CM SERVICE REQUEST from MS {ms_id}
for {call_target}")
        # Forward to MSC
        return "FORWARD_TO_MSC"

    def handle_handover_request(self, ms_id, old_cell, new_cell):
        print(f"BSC {self.id}: Handling Handover Request for MS {ms_id} from
{old_cell} to {new_cell}")
        # Coordinate with MSC and new BTS
        return "HANDOVER_COMMAND"

# --- Simulation Logic ---
def simulate_gsm_scenario():
    ms1 = MobileStation(1, 101) # MS in Cell 101
    bsc1 = BaseStationController(1, [101, 102]) # BSC controlling cells 101,
102

    # Scenario 1: Call Setup
    print("\n--- Scenario: Call Setup ---")
    ms_request = ms1.initiate_call("555-1234")
    if ms_request == "CM_SERVICE_REQUEST":
        bsc_action = bsc1.handle_cm_service_request(ms1.id, "555-1234")
        if bsc_action == "FORWARD_TO_MSC":
            print("MSC (Conceptual): Processing call setup...")
            print(f"MS {ms1.id}: Call state is now {ms1.call_state}")
            ms1.call_state = "ACTIVE" # Call connected
            print(f"MS {ms1.id}: Call state is now {ms1.call_state}")

    # Scenario 2: SMS Messaging
    print("\n--- Scenario: SMS Messaging ---")
    sms_request = ms1.send_sms("555-5678", "Hello from GSM sim!")
    if sms_request == "SMS_DELIVERY_REQUEST":
        print("SMSC (Conceptual): Delivering SMS...")
        print("SMS delivered successfully.")

    # Scenario 3: Handover
    print("\n--- Scenario: Handover ---")
    ms1.move_to_cell(102) # MS moves to a new cell
    # In a real sim, this would trigger signal strength monitoring and
handover decision
    handover_trigger = bsc1.handle_handover_request(ms1.id, 101, 102)
    if handover_trigger == "HANDOVER_COMMAND":
```

```
        print(f"MS {ms1.id}: Handover complete. Now in Cell
{ms1.current_cell_id}")

# simulate_gsm_scenario()
```

Input:

- Initial state of mobile stations (e.g., current cell).
- Call destination numbers, SMS recipients and messages.
- Movement patterns for handover scenarios.

Expected Output:

- Console output detailing the sequence of messages exchanged between simulated GSM entities for call setup, SMS, and handover.
- Verification of state transitions for mobile stations and network elements.
- Demonstration of successful call establishment, SMS delivery, and seamless handover.

Lab 10: CDMA Protocol Implementation

Title: Implementation and Performance Analysis of a CDMA-Based Communication System

Aim: To implement a CDMA-based communication system simulator and analyze its performance in handling multiple users and mitigating interference.

Procedure:

1. **Environment Setup:** Use MATLAB or Python for simulation.
2. **CDMA Fundamentals:**
   - **Spreading Codes:** Generate orthogonal (e.g., Walsh-Hadamard) or pseudo-random noise (PN) spreading codes for multiple users.
   - **Spreading:** Implement the process of spreading the user data by multiplying it with the assigned spreading code.
   - **Despreading:** Implement the process of despreading the received signal by multiplying it with the desired user's code.
3. **Multi-User Scenario:**
   - Simulate multiple users transmitting simultaneously.
   - Sum the spread signals from all active users.
4. **Noise and Interference:** Add AWGN and model multi-user interference (MUI) due to non-perfect orthogonality or asynchronous transmissions.
5. **Receiver Implementation:** Implement a simple CDMA receiver that despreads the signal and attempts to recover the desired user's data.
6. **Performance Metrics:**
   - **BER vs. Eb/No:** Plot the Bit Error Rate (BER) against the Energy per Bit to Noise Power Spectral Density Ratio (Eb/No) for different numbers of active users.
   - **Capacity:** Determine the maximum number of users that can be supported for a given BER target.
   - **Interference Mitigation:** Observe how the spreading gain helps in mitigating interference.
7. **Scenario Variation:** Vary the number of active users, spreading gain, and channel conditions.

Source Code:

```
# Conceptual Python code structure for Lab 10

import numpy as np
import matplotlib.pyplot as plt

# --- System Parameters ---
NUM_USERS = 4 # Number of active users
SPREADING_FACTOR = 8 # Length of Walsh code (e.g., 8 for 8 users max with
orthogonal codes)
BITS_PER_USER = 1000 # Number of bits to transmit per user
EB_NO_DB_RANGE = np.arange(0, 15, 1) # Eb/No range for BER plot

# --- Walsh-Hadamard Codes (Simplified for illustration) ---
def walsh_codes(n):
    if n == 1:
        return np.array([[1]])
    else:
        h_prev = walsh_codes(n // 2)
        top_left = h_prev
        top_right = h_prev
```

```
        bottom_left = h_prev
        bottom_right = -h_prev
        return np.vstack([np.hstack([top_left, top_right]),
np.hstack([bottom_left, bottom_right])])

# Generate Walsh codes for the given spreading factor
if SPREADING_FACTOR > 0 and (SPREADING_FACTOR & (SPREADING_FACTOR - 1) == 0):
# Check if power of 2
    codes = walsh_codes(SPREADING_FACTOR)
else:
    print("Spreading factor must be a power of 2 for Walsh codes. Using dummy
codes.")
    codes = np.eye(NUM_USERS) # Fallback to identity matrix for conceptual

# --- CDMA Simulation Function ---
def simulate_cdma(num_users, spreading_factor, eb_no_db, bits_per_user,
codes):
    eb_no_linear = 10**(eb_no_db / 10)
    noise_variance = 1 / (2 * eb_no_linear * spreading_factor) # Eb/No
definition

    total_errors = 0
    for _ in range(bits_per_user):
        # Generate random bits for all users (+1 or -1)
        user_bits = 2 * np.random.randint(0, 2, num_users) - 1

        # Spread each user's bit
        spread_signals = np.zeros((num_users, spreading_factor))
        for i in range(num_users):
            if i < codes.shape[0]: # Ensure code exists
                spread_signals[i, :] = user_bits[i] * codes[i, :]

        # Sum of all spread signals (multi-user interference)
        received_signal = np.sum(spread_signals, axis=0)

        # Add AWGN
        noise = np.sqrt(noise_variance) * np.random.randn(spreading_factor)
        received_signal_noisy = received_signal + noise

        # Despread and detect for User 0 (desired user)
        if 0 < codes.shape[0]:
            despread_signal = np.dot(received_signal_noisy, codes[0, :])
            detected_bit = 1 if despread_signal > 0 else -1
            if detected_bit != user_bits[0]:
                total_errors += 1
        else:
            # Handle case where no code is available for user 0
            pass

    return total_errors / bits_per_user # BER

# --- Running Simulations ---
ber_results = []
for eb_no_db in EB_NO_DB_RANGE:
    ber = simulate_cdma(NUM_USERS, SPREADING_FACTOR, eb_no_db, BITS_PER_USER,
codes)
    ber_results.append(ber)
    print(f"Eb/No: {eb_no_db} dB, BER: {ber:.4f}")

# --- Plotting Results ---
plt.figure(figsize=(10, 6))
plt.semilogy(EB_NO_DB_RANGE, ber_results, marker='o', label=f'CDMA
(Users={NUM_USERS}, SF={SPREADING_FACTOR})')
plt.xlabel('Eb/No (dB)')
plt.ylabel('Bit Error Rate (BER)')
plt.title('CDMA Performance: BER vs. Eb/No')
```

```
plt.grid(True, which="both", ls="-")
plt.legend()
plt.show()
```

Input:

- Number of active users.
- Spreading factor (length of spreading codes).
- Type of spreading codes (e.g., Walsh, PN).
- Eb/No range for BER analysis.
- Number of bits to simulate.

Expected Output:

- BER vs. Eb/No curves for different numbers of active users, demonstrating the impact of MUI.
- Analysis of how spreading gain affects the system's capacity and interference rejection capability.
- Comparison of performance with and without CDMA (e.g., a simple BPSK link for reference).

Lab 11: Mobile IP Protocol Stack

Title: Design and Implementation of a Mobile IP Protocol Stack

Aim: To design and implement a Mobile IP protocol stack to support seamless mobility of devices across different IP networks, and evaluate its effectiveness in real-world scenarios.

Procedure:

1. **Environment Setup:** Use Python or C++ for implementation.
2. **Mobile IP Components:** Define and simulate the key entities in Mobile IP:
   - **Mobile Node (MN):** The mobile device.
   - **Home Agent (HA):** A router on the MN's home network.
   - **Foreign Agent (FA):** A router on the visited network.
   - **Correspondent Node (CN):** A node communicating with the MN.
3. **Key Processes Implementation:**
   - **Agent Discovery:** Simulate how the MN discovers HA and FA.
   - **Registration:** Implement the registration process where the MN informs its HA about its current Care-of Address (CoA) via the FA.
   - **Tunneling:** Implement IP tunneling (encapsulation) to forward packets from the HA to the MN's CoA.
   - **Route Optimization (Optional):** Implement how CNs can directly send packets to the MN's CoA after learning it.
4. **Packet Flow Simulation:**
   - **MN to CN:** Simulate MN sending packets to a CN (standard IP routing).
   - **CN to MN (HA-FA Tunneling):** Simulate a CN sending packets to the MN's home address, which are then intercepted by the HA and tunneled to the FA, and finally delivered to the MN.
5. **Mobility Scenario:** Simulate the MN moving from its home network to a foreign network, triggering the Mobile IP registration process.
6. **Performance Evaluation:**
   - Measure latency during handover (when MN moves).
   - Analyze packet loss during handover.
   - Compare with standard IP routing (without Mobile IP) during mobility.
7. **Verification:** Print packet headers and routing decisions at each simulated entity.

Source Code:

```python
# Conceptual Python code structure for Lab 11

# --- Constants ---
HOME_NETWORK_PREFIX = "10.0.0."
FOREIGN_NETWORK_PREFIX = "20.0.0."
HOME_AGENT_IP = "10.0.0.1"
FOREIGN_AGENT_IP = "20.0.0.1"
MOBILE_NODE_HOME_IP = "10.0.0.10"
CORRESPONDENT_NODE_IP = "10.0.0.20"

# --- Packet Class (Simplified) ---
class Packet:
    def __init__(self, source_ip, dest_ip, payload, encapsulated_by=None):
        self.source_ip = source_ip
        self.dest_ip = dest_ip
        self.payload = payload
        self.encapsulated_by = encapsulated_by # To track tunneling

    def __str__(self):
```

```
        encap_info = f" (Encapsulated by: {self.encapsulated_by})" if
self.encapsulated_by else ""
        return f"Packet from {self.source_ip} to {self.dest_ip}, Payload:
'{self.payload}'{encap_info}"

# --- Mobile IP Entities (Simplified) ---

class MobileNode:
    def __init__(self, home_ip):
        self.home_ip = home_ip
        self.current_ip = home_ip
        self.care_of_address = None
        self.home_agent = None
        print(f"MN {self.home_ip} initialized.")

    def register(self, home_agent, foreign_agent_ip):
        self.home_agent = home_agent
        self.care_of_address = foreign_agent_ip # FA's address as CoA
        print(f"MN {self.home_ip}: Registering CoA {self.care_of_address}
with HA {self.home_agent.ip}")
        self.home_agent.receive_registration(self.home_ip,
self.care_of_address)

    def send_packet(self, dest_ip, payload):
        packet = Packet(self.current_ip, dest_ip, payload)
        print(f"MN {self.home_ip}: Sending {packet}")
        return packet # Simulate sending to network

    def receive_packet(self, packet):
        print(f"MN {self.home_ip}: Received {packet}")

class HomeAgent:
    def __init__(self, ip):
        self.ip = ip
        self.binding_cache = {} # Mobile Node Home IP -> Care-of Address
        print(f"HA {self.ip} initialized.")

    def receive_registration(self, mn_home_ip, care_of_address):
        self.binding_cache[mn_home_ip] = care_of_address
        print(f"HA {self.ip}: Registered {mn_home_ip} with CoA
{care_of_address}")

    def process_packet_for_mn(self, packet):
        if packet.dest_ip in self.binding_cache:
            coa = self.binding_cache[packet.dest_ip]
            # Encapsulate packet
            tunneled_packet = Packet(self.ip, coa, packet.payload,
encapsulated_by=packet.dest_ip)
            print(f"HA {self.ip}: Tunneling {packet.dest_ip}'s packet to CoA
{coa}: {tunneled_packet}")
            return tunneled_packet # Simulate sending tunneled packet
        else:
            print(f"HA {self.ip}: No binding for {packet.dest_ip}. Standard
routing.")
            return packet # Standard routing if not registered

class ForeignAgent:
    def __init__(self, ip):
        self.ip = ip
        print(f"FA {self.ip} initialized.")

    def receive_tunneled_packet(self, tunneled_packet):
        # Decapsulate packet
        original_dest_ip = tunneled_packet.encapsulated_by
        original_packet = Packet(tunneled_packet.source_ip, original_dest_ip,
tunneled_packet.payload)
```

```
        print(f"FA {self.ip}: Decapsulated packet for {original_dest_ip}:
{original_packet}")
        return original_packet # Deliver to MN

# --- Simulation Flow ---
if __name__ == "__main__":
    mn = MobileNode(MOBILE_NODE_HOME_IP)
    ha = HomeAgent(HOME_AGENT_IP)
    fa = ForeignAgent(FOREIGN_AGENT_IP)

    cn_packet_to_mn = Packet(CORRESPONDENT_NODE_IP, MOBILE_NODE_HOME_IP,
"Hello MN!")

    print("\n--- Scenario 1: MN in Home Network (no Mobile IP action) ---")
    mn.current_ip = mn.home_ip
    mn.receive_packet(cn_packet_to_mn) # CN sends directly to MN

    print("\n--- Scenario 2: MN moves to Foreign Network and Registers ---")
    mn.current_ip = FOREIGN_NETWORK_PREFIX + "100" # MN gets new IP on
foreign network
    mn.register(ha, fa.ip)

    print("\n--- Scenario 3: CN sends packet to MN (HA-FA Tunneling) ---")
    # CN sends packet to MN's home address
    print(f"CN {CORRESPONDENT_NODE_IP}: Sending {cn_packet_to_mn}")

    # HA intercepts and tunnels
    tunneled_packet = ha.process_packet_for_mn(cn_packet_to_mn)
    if tunneled_packet:
        # FA receives and decapsulates
        original_packet_at_fa = fa.receive_tunneled_packet(tunneled_packet)
        # FA delivers to MN
        mn.receive_packet(original_packet_at_fa)
```

Input:

- IP addresses for MN, HA, FA, and CN.
- Messages to be sent.
- Simulated network changes (e.g., MN changing its network attachment point).

Expected Output:

- Console output showing the agent discovery, registration, and tunneling processes.
- Demonstration of packets being correctly forwarded to the mobile node's current location.
- Analysis of how Mobile IP provides seamless connectivity during mobility, even if it introduces some overhead.

Lab 12: Mobile Communication Network Security Framework

Title: Development of a Security Framework for Mobile Communication Networks

Aim: To develop a security framework for mobile communication networks, including encryption algorithms, authentication protocols, and intrusion detection mechanisms.

Procedure:

1. **Environment Setup:** Use Python or Java for implementation.
2. **Threat Model:** Identify common security threats in mobile networks (e.g., eavesdropping, impersonation, denial-of-service, man-in-the-middle attacks).
3. **Component Definition:** Define simplified entities:
   - **User/Mobile Device**
   - **Base Station/Access Point**
   - **Authentication Server (e.g., AAA server)**
   - **Key Distribution Center (KDC)**
4. **Security Mechanism Implementation:**
   - **Encryption:** Implement a symmetric encryption algorithm (e.g., AES in a simplified form, or a simple XOR cipher for demonstration) to protect data confidentiality.
   - **Authentication:** Implement a basic challenge-response authentication protocol (e.g., based on shared secrets or public-key cryptography principles).
   - **Intrusion Detection (Simplified):** Implement a simple mechanism to detect suspicious activity (e.g., too many failed login attempts, unusual traffic patterns).
5. **Scenario Simulation:**
   - **Secure Communication:** Simulate a user communicating securely with a base station using encryption.
   - **Authentication Process:** Simulate a user authenticating with the network.
   - **Intrusion Attempt:** Simulate an attacker attempting to impersonate a legitimate user or launch a DoS attack, and observe the IDS response.
6. **Key Management (Simplified):** Demonstrate a basic key exchange mechanism or pre-shared keys.
7. **Verification:**
   - Show that encrypted data is unreadable without the key.
   - Verify successful authentication for legitimate users and rejection for unauthorized attempts.
   - Log detected intrusions.

Source Code:

```
# Conceptual Python code structure for Lab 12

import hashlib # For hashing in authentication
from cryptography.fernet import Fernet # For simplified encryption (requires
'cryptography' library)

# --- Constants ---
SHARED_SECRET = "supersecretkey" # For authentication
ENCRYPTION_KEY = Fernet.generate_key() # Generate a new key for each run
CIPHER_SUITE = Fernet(ENCRYPTION_KEY)

# --- Security Functions ---

def encrypt_data(data):
    encrypted_data = CIPHER_SUITE.encrypt(data.encode('utf-8'))
    print(f"[Encryption] Encrypted data: {encrypted_data.decode('utf-8')}")
```

```
    return encrypted_data

def decrypt_data(encrypted_data):
    try:
        decrypted_data = CIPHER_SUITE.decrypt(encrypted_data).decode('utf-8')
        print(f"[Decryption] Decrypted data: {decrypted_data}")
        return decrypted_data
    except Exception as e:
        print(f"[Decryption] Decryption failed: {e}")
        return None

def generate_challenge():
    return hashlib.sha256(str(np.random.rand()).encode('utf-8')).hexdigest()

def authenticate_user(username, password, challenge, shared_secret):
    # Simulate a simple challenge-response using a hash
    expected_response = hashlib.sha256(f"{username}-{password}-{challenge}-
{shared_secret}".encode('utf-8')).hexdigest()
    print(f"[Authentication] Expected response: {expected_response}")
    return expected_response

# --- Intrusion Detection (Simplified) ---
LOGIN_ATTEMPTS = {}
MAX_FAILED_ATTEMPTS = 3

def detect_intrusion(username, success):
    if not success:
        LOGIN_ATTEMPTS[username] = LOGIN_ATTEMPTS.get(username, 0) + 1
        if LOGIN_ATTEMPTS[username] >= MAX_FAILED_ATTEMPTS:
            print(f"[IDS] ALERT: Multiple failed login attempts for
{username}! Possible intrusion detected.")
            return True
    else:
        LOGIN_ATTEMPTS[username] = 0 # Reset on success
    return False

# --- Simulation Flow ---
if __name__ == "__main__":
    # --- Scenario 1: Secure Communication ---
    print("\n--- Scenario: Secure Communication ---")
    original_message = "This is a confidential message."
    encrypted_message = encrypt_data(original_message)
    decrypted_message = decrypt_data(encrypted_message)

    if decrypted_message == original_message:
        print("Secure communication successful: Data integrity maintained.")
    else:
        print("Secure communication failed.")

    # --- Scenario 2: Authentication Protocol ---
    print("\n--- Scenario: Authentication Protocol ---")
    user = "alice"
    correct_pass = "password123"
    wrong_pass = "wrongpass"

    # Legitimate attempt
    challenge1 = generate_challenge()
    print(f"Server (Conceptual): Challenge for {user}: {challenge1}")
    user_response1 = authenticate_user(user, correct_pass, challenge1,
SHARED_SECRET)
    # Server verifies
    if user_response1 == hashlib.sha256(f"{user}-{correct_pass}-{challenge1}-
{SHARED_SECRET}".encode('utf-8')).hexdigest():
        print(f"Server (Conceptual): {user} authenticated successfully.")
        detect_intrusion(user, True)
    else:
```

```
        print(f"Server (Conceptual): {user} authentication failed.")
        detect_intrusion(user, False)

    # Malicious attempt
    print("\n--- Scenario: Malicious Authentication Attempt ---")
    for i in range(MAX_FAILED_ATTEMPTS + 1):
        challenge_mal = generate_challenge()
        print(f"Server (Conceptual): Challenge for 'attacker':
{challenge_mal}")
        attacker_response = authenticate_user("attacker", wrong_pass,
challenge_mal, SHARED_SECRET)
        if attacker_response == hashlib.sha256(f"attacker-{correct_pass}-
{challenge_mal}-{SHARED_SECRET}".encode('utf-8')).hexdigest():
            print(f"Server (Conceptual): Attacker authenticated successfully
(ERROR!).")
            detect_intrusion("attacker", True)
        else:
            print(f"Server (Conceptual): Attacker authentication failed.")
            if detect_intrusion("attacker", False):
                break # IDS detected intrusion
```

Input:

- Data to be encrypted/decrypted.
- User credentials (username, password).
- Simulated attacker attempts (e.g., incorrect passwords).

Expected Output:

- Demonstration of data encryption and successful decryption.
- Successful authentication for valid credentials and rejection for invalid ones.
- Logs indicating detection of intrusion attempts (e.g., multiple failed logins).
- Analysis of the strengths and weaknesses of the implemented security mechanisms.

Lab 13: QoS Mechanisms in Mobile Networks

Title: Design and Implementation of QoS Mechanisms in a Simulated Mobile Network

Aim: To design and implement Quality of Service (QoS) mechanisms to prioritize traffic, ensure bandwidth allocation, and manage latency in a simulated mobile network environment.

Procedure:

1. **Environment Setup:** Use Python or MATLAB for simulation.
2. **Traffic Classification:** Define different types of traffic with varying QoS requirements (e.g., Voice (high priority, low latency), Video Streaming (medium priority, high bandwidth), Best-Effort Data (low priority)).
3. **Network Model:** Set up a simplified network model with a bottleneck link or a shared resource (e.g., a base station serving multiple users).
4. **QoS Mechanism Implementation:**
   - **Packet Scheduling:** Implement a scheduling algorithm at the base station or router:
     - **Priority Queuing:** High-priority traffic is served first.
     - **Weighted Fair Queuing (WFQ):** Each traffic class gets a fair share of bandwidth based on its weight.
     - **Strict Priority:** Voice traffic always goes first.
   - **Bandwidth Allocation:** Implement a mechanism to reserve or guarantee a certain amount of bandwidth for specific traffic classes.
   - **Admission Control (Optional):** Implement a simple admission control policy to prevent network overload by rejecting new connections if QoS cannot be guaranteed.
5. **Performance Evaluation:**
   - Measure **latency** for different traffic classes.
   - Measure **throughput** for different traffic classes.
   - Measure **packet loss rate** for different traffic classes.
   - Observe how traffic prioritization affects the performance of various applications.
6. **Scenario Variation:** Vary the network load, mix of traffic types, and QoS policy parameters.

Source Code:

```
# Conceptual Python code structure for Lab 13

import collections
import time
import random
import matplotlib.pyplot as plt

# --- Constants ---
SIMULATION_DURATION = 10 # seconds
LINK_CAPACITY_BPS = 1000000 # 1 Mbps
PACKET_SIZE_BITS = 1000 # 1000 bits per packet

# --- Traffic Classes ---
class TrafficClass:
    VOICE = {'priority': 3, 'weight': 0.5, 'label': 'Voice', 'color': 'red'}
    VIDEO = {'priority': 2, 'weight': 0.3, 'label': 'Video', 'color': 'blue'}
    DATA = {'priority': 1, 'weight': 0.2, 'label': 'Data', 'color': 'green'}

# --- Packet Class ---
class Packet:
```

```
    def __init__(self, traffic_type, creation_time):
        self.traffic_type = traffic_type
        self.creation_time = creation_time
        self.size_bits = PACKET_SIZE_BITS

# --- Router/Scheduler (Simplified) ---
class QosRouter:
    def __init__(self, link_capacity_bps):
        self.link_capacity_bps = link_capacity_bps
        self.queues = {
            TrafficClass.VOICE['label']: collections.deque(),
            TrafficClass.VIDEO['label']: collections.deque(),
            TrafficClass.DATA['label']: collections.deque()
        }
        self.transmitted_packets = []
        self.current_time = 0.0

    def enqueue(self, packet):
        self.queues[packet.traffic_type['label']].append(packet)

    def process_packets(self, time_step):
        bits_transmitted_this_step = 0
        packets_to_transmit = []

        # Priority Scheduling: Iterate through queues by priority (highest
first)
        sorted_queues = sorted(self.queues.items(), key=lambda item:
TrafficClass[item[0].upper()]['priority'], reverse=True)

        for queue_label, queue in sorted_queues:
            while queue and bits_transmitted_this_step + PACKET_SIZE_BITS <=
self.link_capacity_bps * time_step:
                packet = queue.popleft()
                packets_to_transmit.append(packet)
                bits_transmitted_this_step += PACKET_SIZE_BITS

        for packet in packets_to_transmit:
            self.transmitted_packets.append(packet)
            latency = self.current_time - packet.creation_time
            print(f"Time {self.current_time:.2f}s: Transmitted
{packet.traffic_type['label']} packet. Latency: {latency:.4f}s")

        self.current_time += time_step

# --- Simulation Logic ---
def run_qos_simulation():
    router = QosRouter(LINK_CAPACITY_BPS)
    packet_arrival_rate_per_sec = 100 # packets/sec

    all_latencies = {
        TrafficClass.VOICE['label']: [],
        TrafficClass.VIDEO['label']: [],
        TrafficClass.DATA['label']: []
    }

    for t in np.arange(0, SIMULATION_DURATION, 0.01): # Simulate in 10ms
steps
        router.current_time = t
        # Generate new packets
        for _ in range(int(packet_arrival_rate_per_sec * 0.01)): # packets
per time step
            rand_val = random.random()
            if rand_val < 0.2: # 20% voice
                router.enqueue(Packet(TrafficClass.VOICE, t))
            elif rand_val < 0.5: # 30% video
                router.enqueue(Packet(TrafficClass.VIDEO, t))
```

```
            else: # 50% data
                router.enqueue(Packet(TrafficClass.DATA, t))

        router.process_packets(0.01) # Process for 10ms

    # Collect latencies
    for packet in router.transmitted_packets:
        latency = router.current_time - packet.creation_time # Final latency
calculation
        all_latencies[packet.traffic_type['label']].append(latency)

    # --- Plotting Results ---
    plt.figure(figsize=(10, 6))
    for traffic_label, latencies in all_latencies.items():
        if latencies:
            plt.hist(latencies, bins=20, alpha=0.7, label=f'{traffic_label}
(Avg: {np.mean(latencies):.4f}s)',
                     color=TrafficClass[traffic_label.upper()]['color'])
    plt.xlabel('Latency (seconds)')
    plt.ylabel('Number of Packets')
    plt.title('Packet Latency Distribution with Priority Queuing')
    plt.legend()
    plt.grid(True)
    plt.show()

# run_qos_simulation()
```

Input:

- Link capacity.
- Traffic arrival rates for different classes.
- QoS policy parameters (e.g., priority levels, weights).
- Simulation duration.

Expected Output:

- Graphs showing latency distribution for different traffic classes, demonstrating that high-priority traffic experiences lower latency.
- Throughput measurements for each traffic class.
- Analysis of how QoS mechanisms effectively manage network resources to meet application requirements.

Lab 14: Location-Based Services (LBS) Applications

Title: Development of Location-Based Services (LBS) Applications

Aim: To develop location-based services (LBS) applications using GPS or cell tower triangulation techniques, and explore their use cases and practical implementations.

Procedure:

1. **Environment Setup:** Use Python (e.g., with `geopy` for distance calculations, `matplotlib` for visualization) or a web-based approach with JavaScript and mapping APIs (e.g., Google Maps API).
2. **Location Data Simulation/Acquisition:**
   - **GPS Simulation:** Simulate GPS coordinates (latitude, longitude) for a mobile device moving along a path.
   - **Cell Tower Triangulation Simulation:** Define positions of multiple cell towers. Simulate a mobile device receiving signal strengths from these towers. Implement a triangulation algorithm to estimate the device's location.
3. **LBS Application Development:** Choose a simple LBS application to implement:
   - **"Find Nearest" Service:** Given a user's location, find the nearest point of interest (e.g., restaurant, hospital) from a predefined list.
   - **"Geo-fencing" Alert:** Trigger an alert when a user enters or exits a predefined geographical area.
   - **Location Tracking:** Display a user's simulated movement on a map.
4. **Core LBS Logic:**
   - **Distance Calculation:** Implement functions to calculate distances between two geographical points (e.g., Haversine formula).
   - **Triangulation Algorithm (if applicable):** Implement a basic trilateration/triangulation algorithm based on signal strength or time difference of arrival.
5. **User Interface/Visualization:**
   - For Python: Plot locations on a simple 2D graph.
   - For Web: Use a mapping library to display locations and interactions.
6. **Use Case Exploration:** Discuss potential real-world use cases for the implemented LBS.

Source Code:

```
# Conceptual Python code structure for Lab 14 (Find Nearest Service with GPS
simulation)

import numpy as np
import matplotlib.pyplot as plt
from math import radians, sin, cos, sqrt, atan2

# --- Constants ---
EARTH_RADIUS_KM = 6371.0

# --- Haversine Distance Function ---
def haversine_distance(lat1, lon1, lat2, lon2):
    lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon / 2)**2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    distance = EARTH_RADIUS_KM * c
    return distance
```

```
# --- Points of Interest (POIs) ---
pois = {
    "Cafe": (12.9716, 77.5946), # Bangalore coordinates example
    "Hospital": (12.9720, 77.5950),
    "Park": (12.9700, 77.5930),
    "Library": (12.9730, 77.5960)
}

# --- LBS Application: Find Nearest POI ---
def find_nearest_poi(user_lat, user_lon, poi_list):
    nearest_poi = None
    min_distance = float('inf')

    for name, coords in poi_list.items():
        lat, lon = coords
        distance = haversine_distance(user_lat, user_lon, lat, lon)
        if distance < min_distance:
            min_distance = distance
            nearest_poi = name
    return nearest_poi, min_distance

# --- Simulation Flow ---
if __name__ == "__main__":
    # Simulate user's current GPS location
    user_location_lat = 12.9718
    user_location_lon = 77.5948

    print(f"User is at: Lat {user_location_lat}, Lon {user_location_lon}")

    nearest, distance = find_nearest_poi(user_location_lat,
user_location_lon, pois)
    print(f"The nearest POI is '{nearest}' at a distance of {distance:.2f}
km.")

    # --- Visualization ---
    plt.figure(figsize=(8, 8))
    plt.scatter(user_location_lon, user_location_lat, color='red',
marker='*', s=200, label='User Location')

    for name, coords in pois.items():
        plt.scatter(coords[1], coords[0], color='blue', marker='o', s=100,
label=f'POI: {name}')
        plt.text(coords[1] + 0.0001, coords[0] + 0.0001, name, fontsize=9)

    # Draw a line to the nearest POI
    if nearest:
        nearest_coords = pois[nearest]
        plt.plot([user_location_lon, nearest_coords[1]], [user_location_lat,
nearest_coords[0]], 'k--', linewidth=1)

    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.title('Location-Based Services: Nearest POI')
    plt.grid(True)
    plt.legend()
    plt.show()
```

Input:

- User's current location (simulated GPS coordinates or cell tower signal strengths).
- List of points of interest with their coordinates.
- Geo-fence boundaries (for geo-fencing application).

Expected Output:

- The identified nearest point of interest and its distance.
- (For geo-fencing) Alerts triggered when entering/exiting a defined area.
- A visual representation of user location, POIs, and geo-fences on a map or plot.
- Discussion of various LBS use cases (e.g., navigation, emergency services, targeted advertising).

Lab 15: Emerging 5G Technologies Experimentation

Title: Experimentation with Emerging 5G Technologies

Aim: To experiment with emerging 5G technologies such as massive MIMO, beamforming, and network slicing by prototyping and testing various network configurations in a laboratory setting.

Procedure:

1. **Environment Setup:** This lab typically requires specialized hardware/software platforms (e.g., SDR kits like USRP, OpenAirInterface, or advanced simulation tools like MATLAB's 5G Toolbox, NS-3 with 5G modules). For a conceptual lab, a Python/MATLAB simulation will be used.
2. **Technology Focus:** Choose one or two key 5G technologies to focus on:
    - **Massive MIMO:** Simulate a base station with a large number of antennas serving multiple users simultaneously.
    - **Beamforming:** Implement algorithms to steer narrow beams towards individual users.
    - **Network Slicing:** Conceptually demonstrate how network resources can be logically partitioned for different services.
3. **Simulation Model:**
    - **Massive MIMO/Beamforming:** Model a multi-antenna base station and multiple single-antenna users. Implement channel models (e.g., Rayleigh fading).
    - **Network Slicing:** Define different "slices" with distinct QoS requirements (e.g., eMBB, URLLC, mMTC). Simulate resource allocation for these slices.
4. **Algorithm Implementation:**
    - **Massive MIMO:** Implement precoding techniques (e.g., Zero-Forcing, MRT) to manage interference and enhance signal strength for multiple users.
    - **Beamforming:** Implement a simple beamforming algorithm (e.g., phase array steering) to direct energy.
    - **Network Slicing:** Implement a resource orchestrator that allocates virtualized resources (bandwidth, computing power) to different slices based on their SLAs.
5. **Performance Evaluation:**
    - **Massive MIMO/Beamforming:** Measure sum throughput, individual user throughput, SINR improvement, and spatial multiplexing gain.
    - **Network Slicing:** Demonstrate how different slices meet their QoS requirements even under varying loads.
6. **Scenario Variation:**
    - Vary the number of antennas at the BS.
    - Vary the number of users.
    - Change traffic profiles for different slices.
7. **Visualization:** Plot beam patterns, throughput comparisons, or resource utilization per slice.

Source Code:

```
# Conceptual Python code structure for Lab 15 (Simple Beamforming)

import numpy as np
import matplotlib.pyplot as plt

# --- System Parameters ---
NUM_ANTENNAS = 8 # Number of antenna elements at the base station
NUM_USERS = 2
```

```
CARRIER_FREQUENCY = 2.4e9 # Hz
WAVELENGTH = 3e8 / CARRIER_FREQUENCY
ANTENNA_SPACING = WAVELENGTH / 2 # Half-wavelength spacing

# --- User Angles (relative to array broadside) ---
USER_ANGLES_DEG = [30, -20] # Degrees

# --- Beamforming Function (Uniform Linear Array - ULA) ---
def calculate_array_factor(theta_deg, num_antennas, antenna_spacing,
wavelength):
    theta_rad = np.radians(theta_deg)
    k = 2 * np.pi / wavelength
    array_factor = np.zeros_like(theta_rad, dtype=complex)

    for i in range(num_antennas):
        # Apply phase shift for steering (e.g., to steer towards user 0)
        # For a simple steered beam, phase shift for antenna 'i' is -k * i *
d * sin(target_angle)
        # Here, we're just calculating the array factor for a given angle
        phase_shift = k * i * antenna_spacing * np.sin(theta_rad)
        array_factor += np.exp(1j * phase_shift)
    return np.abs(array_factor) / num_antennas # Normalized

def apply_beamforming_weights(num_antennas, antenna_spacing, wavelength,
target_angle_deg):
    target_angle_rad = np.radians(target_angle_deg)
    k = 2 * np.pi / wavelength
    weights = np.zeros(num_antennas, dtype=complex)
    for i in range(num_antennas):
        weights[i] = np.exp(-1j * k * i * antenna_spacing *
np.sin(target_angle_rad))
    return weights

# --- Simulation Flow ---
if __name__ == "__main__":
    # Calculate beam pattern for a range of angles
    angles_to_plot = np.arange(-90, 91, 1)

    # --- Scenario 1: Simple Beamforming (Steering to User 0) ---
    print("\n--- Scenario: Simple Beamforming ---")
    target_user_angle = USER_ANGLES_DEG[0]
    beamforming_weights = apply_beamforming_weights(NUM_ANTENNAS,
ANTENNA_SPACING, WAVELENGTH, target_user_angle)

    # Calculate the array factor with these weights
    array_factor_steered = np.zeros_like(angles_to_plot, dtype=float)
    for i, angle_deg in enumerate(angles_to_plot):
        angle_rad = np.radians(angle_deg)
        k = 2 * np.pi / WAVELENGTH
        current_array_response = 0j
        for ant_idx in range(NUM_ANTENNAS):
            current_array_response += beamforming_weights[ant_idx] *
np.exp(1j * k * ant_idx * ANTENNA_SPACING * np.sin(angle_rad))
        array_factor_steered[i] = np.abs(current_array_response) /
NUM_ANTENNAS # Normalized

    plt.figure(figsize=(10, 6))
    plt.plot(angles_to_plot, 20 * np.log10(array_factor_steered + 1e-10),
label=f'Beam steered to {target_user_angle}°')
    plt.axvline(x=target_user_angle, color='r', linestyle='--',
label=f'Target User 0 ({target_user_angle}°)')
    if len(USER_ANGLES_DEG) > 1:
        plt.axvline(x=USER_ANGLES_DEG[1], color='g', linestyle=':',
label=f'Other User 1 ({USER_ANGLES_DEG[1]}°)')

    plt.xlabel('Angle (degrees)')
```

```
    plt.ylabel('Normalized Array Factor (dB)')
    plt.title('Beamforming Pattern for a Uniform Linear Array')
    plt.ylim([-30, 0]) # Show main lobe and side lobes
    plt.grid(True)
    plt.legend()
    plt.show()

    # --- Scenario 2: Conceptual Network Slicing ---
    print("\n--- Scenario: Conceptual Network Slicing ---")
    # This would involve defining resource pools and allocating them to
different slices
    # E.g., a dictionary representing resource allocation
    network_slices = {
        "eMBB_Slice": {"bandwidth_Mbps": 500, "latency_ms": 10, "users":
100},
        "URLLC_Slice": {"bandwidth_Mbps": 50, "latency_ms": 1, "users": 5},
        "mMTC_Slice": {"bandwidth_Mbps": 10, "latency_ms": 500, "users":
1000}
    }
    print("Simulating resource allocation for different 5G network slices:")
    for slice_name, params in network_slices.items():
        print(f"- {slice_name}: Bandwidth={params['bandwidth_Mbps']} Mbps,
Latency={params['latency_ms']} ms, Users={params['users']}")
        # In a real simulation, you'd run traffic through these slices and
verify QoS
```

Input:

- Number of antennas at the base station.
- Number of users and their angular positions.
- Carrier frequency.
- Target angles for beamforming.
- QoS requirements for different network slices (e.g., bandwidth, latency, reliability).

Expected Output:

- Plots of beam patterns showing the main lobe directed towards the desired user and nulls towards interferers.
- (For Massive MIMO) Throughput gains and SINR improvements as the number of antennas increases.
- (For Network Slicing) Conceptual demonstration of how resources are partitioned and how different slices meet their distinct QoS requirements.
- Analysis of the benefits and challenges of implementing these 5G technologies.