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

```
# Step 1: Clifford Gates
def get_clifford_gates():
    I = np.array([[1, 0], [0, 1]])
    X = np.array([[0, 1], [1, 0]])
    Y = np.array([[0, -1j], [1j, 0]])
    Z = np.array([[1, 0], [0, -1]])
    H = np.array([[1, 1], [1, -1]]) / np.sqrt(2)
    S = np.array([[1, 0], [0, 1j]])
    return [I, X, Y, Z, H, S]
```

```
# Step 2: Random Clifford Sequence
def generate_random_sequence(length):
    clifford_gates = get_clifford_gates()
    return [random.choice(clifford_gates) for _ in range(length)]
```

```
def compute_inverse_sequence(sequence):
    """Compute inverse to return to |0⟩"""
    # Multiply all gates and find inverse
    total = np.eye(2)
    for gate in sequence:
        total = gate @ total

    # Find closest Clifford inverse (simplified)
    clifford_gates = get_clifford_gates()
    best_gate = clifford_gates[0]
    best_fidelity = 0

    for gate in clifford_gates:
        test_result = gate @ total
        fidelity = abs(test_result[0, 0])**2  # Probability of |0⟩
        if fidelity > best_fidelity:
            best_fidelity = fidelity
            best_gate = gate

    return best_gate
```

```
# Step 3: Benchmarking Circuit
def run_rb_sequence(sequence_length, noise_level=0.01):
    """Run single RB sequence"""
    # Start with |0⟩
    state = np.array([1, 0])
```

```
    # Generate random sequence
    sequence = generate_random_sequence(sequence_length)

    # Apply sequence with noise
    for gate in sequence:
        state = gate @ state
        # Add noise (depolarizing)
        if random.random() < noise_level:
            noise_gate = random.choice(get_clifford_gates())
            state = noise_gate @ state

    # Apply inverse
    inverse_gate = compute_inverse_sequence(sequence)
    state = inverse_gate @ state

    # Return fidelity (probability of measuring |0⟩)
    return abs(state[0])**2
```

```
# Step 4: RB Protocol
def randomized_benchmarking(max_length=20, num_sequences=30):
    """Full randomized benchmarking protocol"""
    print("=== Randomized Benchmarking Protocol ===")

    lengths = range(1, max_length + 1, 2)
    fidelities = []

    for length in lengths:
        # Average over multiple random sequences
        sequence_fidelities = []
        for _ in range(num_sequences):
            fidelity = run_rb_sequence(length)
            sequence_fidelities.append(fidelity)

        avg_fidelity = np.mean(sequence_fidelities)
        fidelities.append(avg_fidelity)
        print(f"Length {length}: Fidelity = {avg_fidelity:.4f}")

    return lengths, fidelities
```

```
# Step 5: Analyze Results
def analyze_rb_results(lengths, fidelities):
    """Analyze RB decay and extract error rate"""
    # Fit exponential decay: F(m) = A * p^m + B

    # Simple linear fit in log space
    log_fidelities = np.log(np.maximum(fidelities, 1e-10))
    coeffs = np.polyfit(lengths, log_fidelities, 1)

    # Extract decay rate
    decay_rate = -coeffs[0]
    error_per_gate = 1 - np.exp(-decay_rate)

    print(f"\n=== Analysis Results ===")
```

```
    print(f"Decay rate: {decay_rate:.6f}")
    print(f"Error per gate: {error_per_gate:.6f}")
    print(f"Gate fidelity: {1 - error_per_gate:.6f}")

    return decay_rate, error_per_gate
```

```
# Step 6: Visualization
def plot_rb_results(lengths, fidelities, decay_rate):
    """Plot RB decay curve"""
    plt.figure(figsize=(10, 6))

    # Plot data points
    plt.plot(lengths, fidelities, 'bo-', label='Measured Fidelity', markersize=8)

    # Plot fitted curve
    fit_curve = np.exp(-decay_rate * np.array(lengths))
    plt.plot(lengths, fit_curve, 'r--', label='Exponential Fit', linewidth=2)

    plt.xlabel('Sequence Length')
    plt.ylabel('Fidelity')
    plt.title('Randomized Benchmarking Decay Curve')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.ylim(0, 1.1)
    plt.show()
```

```
def test_different_noise_levels():
   """Test RB with different noise levels"""
   print("\n=== Testing Different Noise Levels ===")

   noise_levels = [0.001, 0.01, 0.05, 0.1]
   colors = ['g', 'b', 'orange', 'r']

   plt.figure(figsize=(10, 6))

   for i, noise in enumerate(noise_levels):
       lengths = range(1, 16, 2)
       fidelities = []

       for length in lengths:
           avg_fidelity = np.mean([run_rb_sequence(length, noise) for _ in range(
           fidelities.append(avg_fidelity)

       plt.plot(lengths, fidelities, color=colors[i], marker='o', linestyle='-',
                label=f'Noise = {noise}', markersize=6)

   plt.xlabel('Sequence Length')
   plt.ylabel('Fidelity')
   plt.title('RB with Different Noise Levels')
   plt.legend()
   plt.grid(True, alpha=0.3)
   plt.show()
```

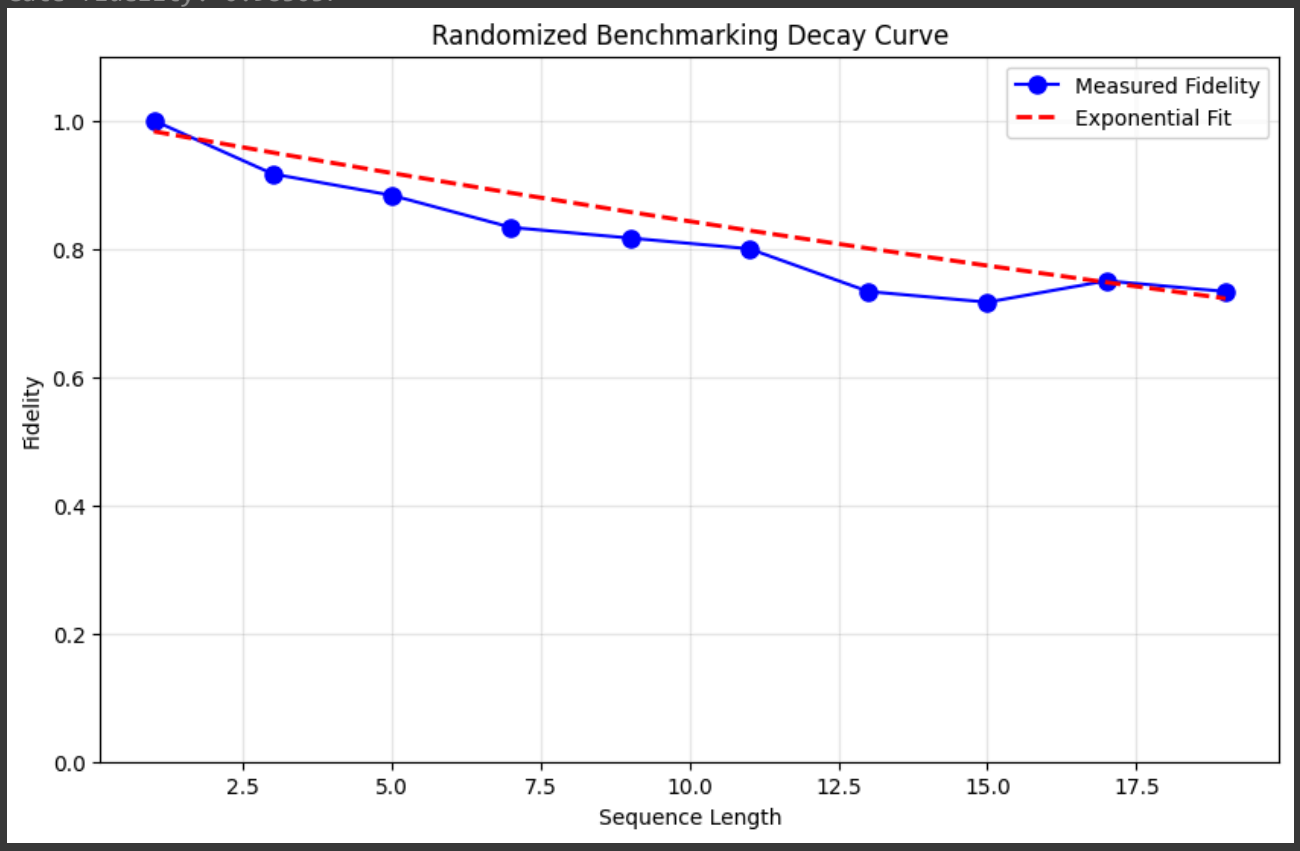
```
# Multi-Qubit RB (simplified)
def two_qubit_rb():
    """Simplified 2-qubit RB"""
    print("\n=== 2-Qubit RB (Simplified) ===")

    # 2-qubit Clifford gates (simplified set)
    gates_2q = [
        np.kron(np.eye(2), np.eye(2)),  # II
        np.kron(get_clifford_gates()[4], np.eye(2)),  # HI
        np.kron(np.eye(2), get_clifford_gates()[4]),  # IH
        np.array([[1,0,0,0],[0,1,0,0],[0,0,0,1],[0,0,1,0]])  # CNOT
    ]

    lengths = range(1, 11)
    fidelities = []

    for length in lengths:
        total_fidelity = 0
        for _ in range(20):
            # Start with |00⟩
            state = np.array([1, 0, 0, 0])

            # Apply random 2-qubit sequence
            for _ in range(length):
                gate = random.choice(gates_2q)
                state = gate @ state
                # Add noise
                if random.random() < 0.02:
                    noise = random.choice(gates_2q)
                    state = noise @ state

            # Simplified return to |00⟩
            fidelity = abs(state[0])**2
            total_fidelity += fidelity

        avg_fidelity = total_fidelity / 20
        fidelities.append(avg_fidelity)

    plt.figure(figsize=(8, 5))
    plt.plot(lengths, fidelities, 'mo-', linewidth=2, markersize=6)
    plt.xlabel('Sequence Length')
    plt.ylabel('2-Qubit Fidelity')
    plt.title('2-Qubit Randomized Benchmarking')
    plt.grid(True, alpha=0.3)
    plt.show()
```

```
def main():
    print("🎯 RANDOMIZED BENCHMARKING PROTOCOL 🎯")
    print("=" * 45)

    # Single qubit RB
    lengths, fidelities = randomized_benchmarking()
    decay_rate, error_rate = analyze_rb_results(lengths, fidelities)
    plot_rb_results(lengths, fidelities, decay_rate)
```

```
    # Test different noise
    test_different_noise_levels()

    # 2-qubit RB
    two_qubit_rb()

    print("\n" + "=" * 45)
    print("✅ CONCLUSION: Successfully implemented RB Protocol!")
    print(f"✅ Measured gate error rate: {error_rate:.6f}")
    print("✅ Characterized quantum gate performance")
```

```
main()
```

```
Length 9: Fidelity = 0.8167
Length 11: Fidelity = 0.8000
Length 13: Fidelity = 0.7333
Length 15: Fidelity = 0.7167
Length 17: Fidelity = 0.7500
Length 19: Fidelity = 0.7333

=== Analysis Results ===
Decay rate: 0.017108
Error per gate: 0.016963
Gate fidelity: 0.983037
```




```
=== Testing Different Noise Levels ===
```