State-Vector Simulation and Tensor-Network simulation of Quantum Circuits

Hashmitha Sugumar

October 2024

1 Introduction

Efficient digital simulation of quantum systems holds significance in the era of NISQ (Noise Intermediate Scale Quantum Computing) . The term NISQ was coined by John Preskill and it refers to the current state of quantum computers. NISQ computers are made up of hundreds of qubits that are not error corrected, and they are subject to significant levels of quantum noise and error.
Quantum simulation is the concept of mimicking quantum behavior (quantum systems and their dynamics), that is, it refers to the process of simulating the behaviour of quantum systems using a quantum computer. On a broad scale there lies three main categories- Analog Quantum Simulators, Digital Quantum Simulators, and Quantum Annealers.

1. **Analog Quantum Simulation** : is the process of replicating quantum behavior through the physical implementation of quantum dynamics.
2. **Digital Quantum Simulation**: is the process of programming a digital quantum simulator using gates to replicate any other quantum system. These "gates" are norm-maintaining transformations of qubits in a quantum system.
3. **Quantum Annealers**: operate by representing optimization problems as energy landscapes, where the goal is to find the lowest energy state that corresponds to the optimal solution utilizing the principles like quantum superposition and quantum tunneling.

Runtime is in most cases the primary metric in comparing the simulation techniques. Runtime refers to how long the execution of a program takes. Here I'd be taking into account both the runtime and memory usage as metrics.
The different types of quantum simulations on the basis of their functionality for an n-qubit system is as follows:

- State-Vector Simulation (SV simulation)
- Tensor-Network simulation (TN simulation)
- Density-Matrix Simulation (DM simulation)

- Unitary Simulation.

NOTE: As per the given **first task**, the goal is to implement a state-vector simulator for quantum circuits from scratch. The task deals with defining a quantum circuit consisting of the gates -X, H, CNOT and applying the gates sequentially to the state-vector via matrix multiplication. Also I was asked to plot the runtime of the code as a function of the number of qubits and find how many qubits was it possible to simulate this way.
As per the given **second task**, it deals with defining a quantum circuit consisting of the 1- and 2-qubit matrix representations of X, H, CNOT (same as above) and applying them sequentially to the quantum state tensor via tensor multiplication. Additionally, the runtime of my code has to be plotted as a function of the number of qubits and find how many qubits was it possible to simulate this way, thereby comparing the current results with the previous ones!

---

2 State Vector Simulation

Step-01: Importing utilities:

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

Step-02: Initializing gates as matrices:

```
X= np.array([[0,1],[1,0]]) #NOT gate
H = np.array([[1, 1],
              [1, -1]]) / np.sqrt(2)  #Hadamard gate
CX= np.array([[1, 0, 0, 0],
                 [0, 1, 0, 0],
                 [0, 0, 0, 1],
                 [0, 0, 1, 0]]) #CNOT gate (2-qubit gate)
I= np.array([[1,0],[0,1]]) #Identity
```

Step-03: Defining the initial state:

```
def initialize_state(n):
 state = np.zeros(2**n) #an array of zeroes
 state[0] = 1 #setting the first element to 1 to create ket 0
 print(state)
```

Step-04: Defining a function to apply single qubit gates to a specific qubit in a n-qubit system:

```
def apply_single_qubit_gate(gate, state, qubit, n):
    full_gate = 1
    for i in range(n):
        if i == qubit:
            full_gate = np.kron(full_gate, gate)
        else:
            full_gate = np.kron(full_gate, I)
    return np.dot(full_gate, state)
```

Step-05: Defining a function to apply two-qubit gate (here,CNOT) to specific qubits in a n-qubit system:

```
def apply_two_qubit_gate(state, control, target, n):
    if n == 2 and control == 0 and target == 1:
        return np.dot(CNOT, state)
    else:
        raise NotImplementedError("Only a 2-qubit CNOT is implemented in this demo.")
```

Step-06: Quantum circuit simulation part for varying number of qubits:

```
def quantum_circuit(n):
    state = initialize_state(n)

    # Apply an H gate to the first qubit (qubit 0)
    state = apply_single_qubit_gate(H, state, 0, n)

    # Apply a CNOT gate (control=0, target=1) if n == 2
    if n == 2:
        state = apply_two_qubit_gate(state, 0, 1, n)

    return state
```

Step-07: Measuring runtime and memory usage as a function of number of qubits:

```
def measure_runtime_memory(max_qubits):
    qubit_counts = list(range(1, max_qubits + 1))
    runtimes = []
    memory_usages = []

    process = psutil.Process()  # Get the current process for memory usage tracking

    for n in qubit_counts:
        # Measure runtime
        start_time = time.time()
        quantum_circuit(n)
```

```
        end_time = time.time()
        runtimes.append(end_time - start_time)

        # Measure memory usage in MB
        memory_info = process.memory_info().rss / (1024 ** 2)  # Memory in MB
        memory_usages.append(memory_info)
        print(f"Qubits: {n}, Runtime: {end_time - start_time:.6f} seconds, Memory Usage: {me

    return qubit_counts, runtimes, memory_usages
```

Step-08: Plotting the runtime and memory usage:

```
def plot_runtime_memory(max_qubits):
    qubit_counts, runtimes, memory_usages = measure_runtime_memory(max_qubits)

    fig, ax1 = plt.subplots()

    # Plot runtime
    ax1.set_xlabel('Number of Qubits')
    ax1.set_ylabel('Runtime (seconds)', color='tab:blue')
    ax1.plot(qubit_counts, runtimes, marker='o', color='tab:blue')
    ax1.tick_params(axis='y', labelcolor='tab:blue')

    # Create a second y-axis to plot memory usage
    ax2 = ax1.twinx()
    ax2.set_ylabel('Memory Usage (MB)', color='tab:red')
    ax2.plot(qubit_counts, memory_usages, marker='x', color='tab:red')
    ax2.tick_params(axis='y', labelcolor='tab:red')

    plt.title('Quantum Circuit Simulation: Runtime and Memory Usage vs Number of Qubits')
    fig.tight_layout()
    plt.grid(True)
    plt.show()
```

Step-09: Running and plotting for n=10:

```
plot_runtime_memory(10)
```

The graphs I obtained are shown in figure- 1,figure-2 and figure-3 for n=10,15,16 respectively. It is clear from the figures that the session crashed when n=16 was given. Two important inferences can be made from the graphs :

- The maximum number of qubits which can be simulated this way is 15. This means that the classical simulation of quantum systems becomes impractical for large numbers of qubits due to the exponential growth in memory and processing power required.

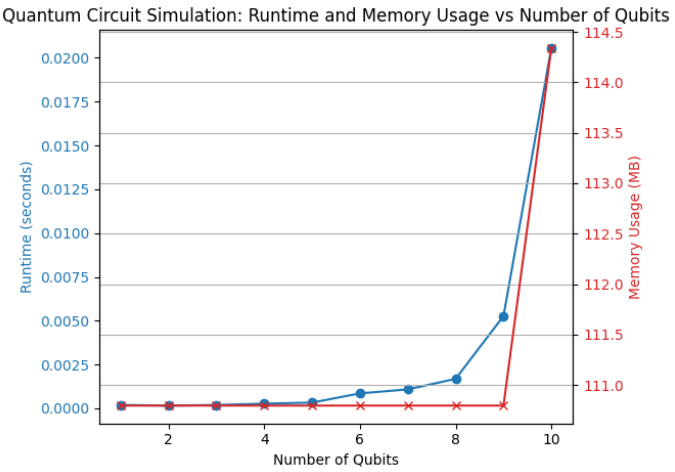

Figure 1: *For n=10 qubits*

- The runtime increases exponentially with the increase in the number of qubits.This highlights a significant challenge in simulating quantum circuits classically and demonstrates the need for specialized hardware like quantum computers to handle larger qubit systems efficiently.

This was the case only when I ran the code on my laptop. When I ran the same code on my desktop for the same number of qubits (n=15), the graph experienced slight changes. Even though the runtime more or less remained the same, the memory usage differed in both the cases (see figure 2 and 4). One similarity with both the devices is that the session when ran crashed for n=16. This could be because of the differences in system architecture (CPU, RAM, and cache), but both systems hit their limit at around 16 qubits because they both faced the same fundamental exponential growth in memory requirements.

2.0.1 Conclusion

The conclusions which can be drawn from the above simulation is that :

- The memory usage remains relatively constant for a small number of qubits (around 1-10 qubits) because the state space of a quantum system grows exponentially with the number of qubits. However, the increase is manageable up to around 12 qubits. From 12 qubits onward, the memory usage spikes sharply, reflecting the exponential growth in the number of possible quantum states. From 12 qubits onward, the memory usage

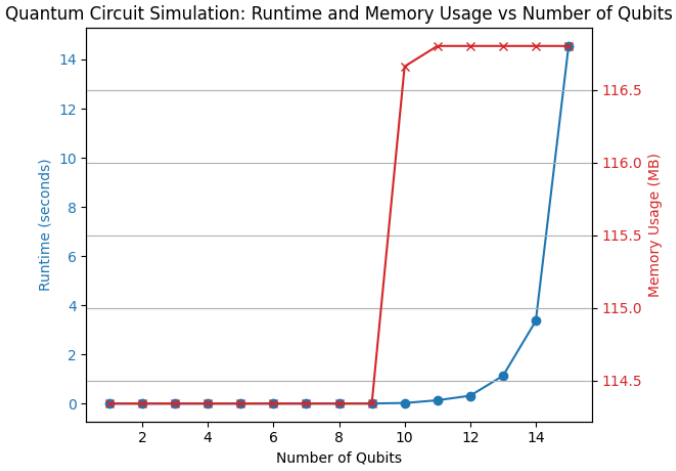

Figure 2: *For n=15 qubits*

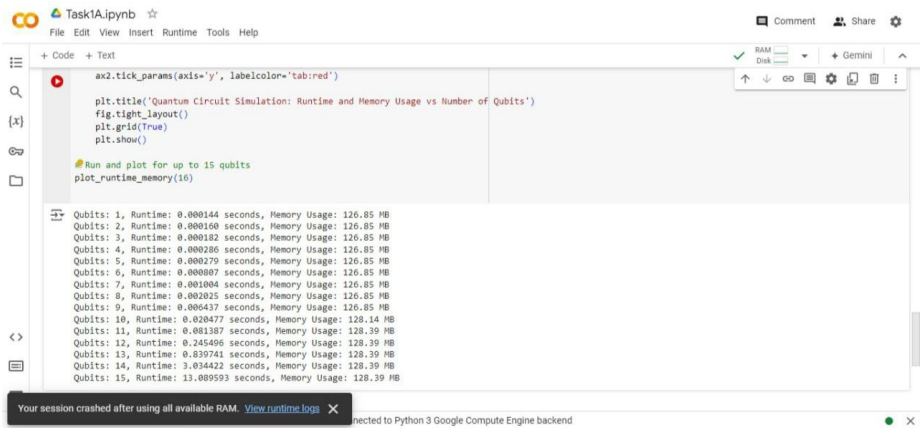

Figure 3: *For n=16 qubits*

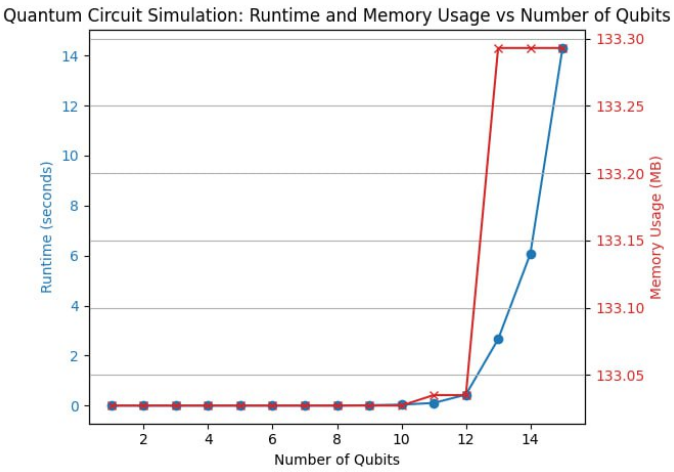

Figure 4: For n=15 on desktop

spikes sharply, reflecting the exponential growth in the number of possible quantum states 2^n, which requires storing more data.

- Both devices crashed when I tried to simulate 16 qubits. This is due to the fact that the memory requirements become excessive beyond 15 qubits. The number of elements required to represent the state of an **n-qubit** system is 2^n and for 16 qubits it would become 65,536 complex numbers. As a result, the system ran out of available memory or computational resources to handle the large matrix and vector sizes, leading to a crash.

- It is good to note that when a system runs out of physical RAM, it uses virtual memory (swap space) on the hard drive. This process is much slower, and memory management mechanisms might behave differently depending on the system's configuration. Hence, a system with more RAM will show lower memory usage for the same task because it doesn't need to swap as much which directly coincides with the fact that my laptop has got more RAM than my desktop!

The size of the SV grows exponentially, limiting simulation to the compute node's RAM.

3 Tensor Network Simulation

Tensor networks are smart factorizations intended for such large vectors, because a gate is limited to a factor of the previously-huge vector, "contracting" it with another tensor (the gate tensor has a rank equal to the number of qubits it transforms), which reduces computation at the expense of accuracy. The code for the second subtask is as follows:

```
import numpy as np
import time
import matplotlib.pyplot as plt
import psutil  # For measuring memory usage

# Define basic quantum gates as matrices
X = np.array([[0, 1],
              [1, 0]])  # X gate (Pauli-X)

H = np.array([[1, 1],
              [1, -1]]) / np.sqrt(2)  # H gate (Hadamard)

I = np.array([[1, 0],
              [0, 1]])  # Identity gate

CNOT = np.array([[1, 0, 0, 0],
                 [0, 1, 0, 0],
                 [0, 0, 0, 1],
                 [0, 0, 1, 0]])  # CNOT gate

# Function to initialize n-qubit state |0...0>
def initialize_state(n):
    state = np.zeros((2,) * n)
    state[0] = 1  # The |0...0> state
    return state

# Function to apply a single-qubit gate using np.einsum
def apply_single_qubit_gate_einsum(gate, state, qubit):
    # Reshape state to apply gate on the specific qubit
    state_shape = list(state.shape)
    state_reshaped = state.reshape((-1, 2, *state_shape[qubit+1:]))  # Rearranging axes
    state_reshaped = np.moveaxis(state_reshaped, 1, 0)  # Move qubit axis to front
    new_state = np.einsum('ij...,j...->i...', gate, state_reshaped)
    return new_state.reshape(state.shape)

# Function to apply a two-qubit gate (e.g., CNOT) using np.einsum
def apply_two_qubit_gate_einsum(state, control, target):
    # Reshape state to apply CNOT on the specific qubits
    state_shape = list(state.shape)
    # Reshape the state to have 4 dimensions for the CNOT gate
    state_reshaped = state.reshape(
        (*state_shape[:control], 2, *state_shape[control + 1:target], 2, *state_shape[target
    )

    # Move the control and target qubit dimensions to the front
    state_reshaped = np.moveaxis(state_reshaped, [control, target], [0, 1])
```

```
    state_reshaped = state_reshaped.reshape((4, -1))

    # Apply CNOT using matrix multiplication and reshape back
    new_state = (CNOT @ state_reshaped).reshape(
        (*state_shape[:control], 2, *state_shape[control + 1:target], 2, *state_shape[target
    )

    # Move the control and target qubit dimensions back to their original positions
    new_state = np.moveaxis(new_state, [0, 1], [control, target])

    return new_state.reshape(state.shape)
# Quantum circuit simulation for varying number of qubits using np.einsum
def quantum_circuit_einsum(n):
    state = initialize_state(n)

    # Apply an H gate to the first qubit (qubit 0)
    state = apply_single_qubit_gate_einsum(H, state, 0)

    # Apply a CNOT gate (control=0, target=1) if n >= 2
    if n >= 2:
        state = apply_two_qubit_gate_einsum(state, 0, 1)

    return state

# Measure runtime and memory usage as a function of number of qubits
def measure_runtime_memory_einsum(max_qubits):
    qubit_counts = list(range(1, max_qubits + 1))
    runtimes = []
    memory_usages = []

    process = psutil.Process()  # Get the current process for memory usage tracking

    for n in qubit_counts:
        # Measure runtime
        start_time = time.time()
        quantum_circuit_einsum(n)
        end_time = time.time()
        runtimes.append(end_time - start_time)

        # Measure memory usage in MB
        memory_info = process.memory_info().rss / (1024 ** 2)  # Memory in MB
        memory_usages.append(memory_info)
        print(f"Qubits: {n}, Runtime: {end_time - start_time:.6f} seconds, Memory Usage: {me

    return qubit_counts, runtimes, memory_usages
```

```
# Plotting the runtime and memory usage
def plot_runtime_memory_einsum(max_qubits):
    qubit_counts, runtimes, memory_usages = measure_runtime_memory_einsum(max_qubits)

    fig, ax1 = plt.subplots()

    # Plot runtime
    ax1.set_xlabel('Number of Qubits')
    ax1.set_ylabel('Runtime (seconds)', color='tab:blue')
    ax1.plot(qubit_counts, runtimes, marker='o', color='tab:blue', label='Runtime')
    ax1.tick_params(axis='y', labelcolor='tab:blue')

    # Create a second y-axis to plot memory usage
    ax2 = ax1.twinx()
    ax2.set_ylabel('Memory Usage (MB)', color='tab:red')
    ax2.plot(qubit_counts, memory_usages, marker='x', color='tab:red', label='Memory Usage')
    ax2.tick_params(axis='y', labelcolor='tab:red')

    plt.title('Quantum Circuit Simulation: Runtime and Memory Usage vs Number of Qubits')
    fig.tight_layout()
    plt.grid(True)
    plt.show()

# Run and plot for up to n qubits
plot_runtime_memory_einsum(29)
```

The results obtained were similar to the previous case but this time the number of qubits that could be simulated this way increased to 29. The session crashed when n=30. The resulting graph is as shown in figure 5.

3.0.1 Conclusion

So now, *why is the difference in the number of qubits that can be simulated between matrix multiplication and tensor multiplication?*
When I used tensor multiplication instead of matrix multiplication, I was able to simulate 29 qubits, compared to 15 qubits with matrix multiplication. This can be attributed to:

- **Higher Dimensionality:** Tensor multiplication naturally handles the increased dimensionality without exponentially larger matrices. It treats the qubit states as independent entities and combines them seamlessly.
- **Memory Efficiency:** Tensor operations require less memory overhead because one can manipulate higher-dimensional structures without creating massive matrices for each operation, thus avoiding memory limits

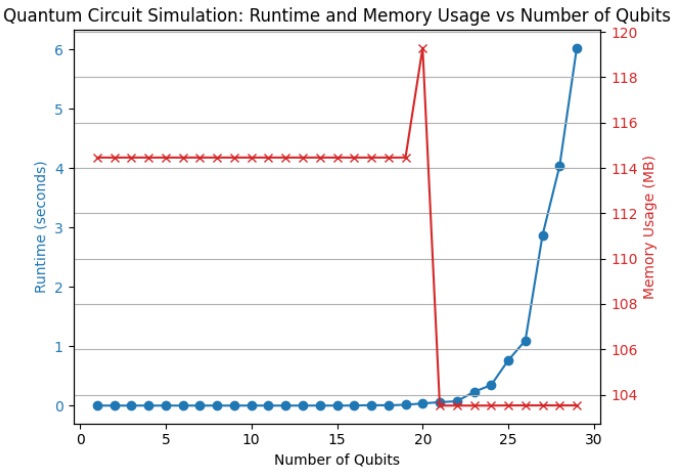

Figure 5: For n=29 qubits

more effectively.

Matrix multiplication complicates operations and limits scalability when compared to tensor multiplications.

4 Conclusion

- Both matrix multiplication and tensor multiplication demonstrate exponential growth in runtime as the number of qubits increases, reflecting the inherent complexity of quantum systems.
- The matrix multiplication approach limits scalability, becoming impractical for simulating more than 15 qubits due to the rapid increase in matrix size and corresponding resource demands. In contrast, the tensor multiplication method significantly enhances scalability, allowing for the successful simulation of up to 29 qubits. This is attributed to its ability to manipulate smaller tensors, resulting in reduced computational and memory overhead pointing towards the fact that tensor multiplication is more efficient and effective than matrix multiplication.
- The findings underscore the importance of efficient algorithms in quantum computing, especially as research progresses toward larger quantum systems. The ability to scale simulations is vital for practical applications in quantum algorithms, quantum machine learning, and other advanced quantum technologies.
- This work opens avenues for further exploration of tensor networks and

other advanced computational techniques to enhance quantum simulations. Future research could focus on optimizing tensor operations further and exploring hybrid methods that combine the strengths of both matrix and tensor approaches.

5 Bonus Questions:

1. *How would you sample from the final states in the state vector or tensor representations?*
   Sampling refers to the act of measuring a quantum state to obtain to a classical result. The probabilities of obtaining particular outcomes are derived from the amplitudes of the state vector.

   Consider a 2-qubit system as follows: Assuming that the state is normalized,

   $$|\psi >= a|00 > +b|01 > +c|10 > +d|11 > \tag{1}$$

   On measuring this state, the probability of measuring the first state, second state, third state and the fourth state would be : $|a|^2$, $|b|^2$, $|c|^2$, and $|d|^2$ respectively. The probabilities are first calculated and then sample an outcome based on the distribution. For example, if P(00)= 0.3, P(01)=0.4, P(10)=0.2 and P(11)=0.1 , then one would likely get —00¿ more often when sampling repeatedly.

```
import numpy as np
# Assuming ‘state‘ as the given statevector
probabilities = np.abs(state) ** 2
states = np.arange(len(state))
sampled_state = np.random.choice(states, p=probabilities)
```

2. *And how about computing exact expectation values in the form:*

   $$< \Psi|Op|\psi > \tag{2}$$

   In order to compute the exact expectation value it's a wise choice to break up the equation as follows:

   $$< \Psi|op|\Psi >=< \psi|\Omega > \tag{3}$$

   where,

   $$|\Omega >= Op|\psi > \tag{4}$$

```
operator = np.array([[...], [...], ...])  # Defining the operator
Omega = np.dot(operator, psi)             # psi here refers to the state
expectation_value = np.vdot(psi, Omega)
```