Functional Paradigms in Quantum Computing

Saanvi Vishal
IMT2021043
Saanvi.Vishal@iiitb.ac.in

Arjun Subhedar
IMT2021069
Arjun.Subhedar@iiitb.ac.in

Ritik Kumar Gupta
IMT2021098
RitikKumar.Gupta@iiitb.ac.in

Pandey Shourya Prasad
IMT2021535
Shourya.Prasad@iiitb.ac.in

***Abstract*—Quantum computers are the new emerging and exciting field of computer science. Quantum computer technology is based on the laws of quantum physics and utilizes the properties of quantum particles i.e. superposition and entanglement to perform computation. In contrast to classical computers, where information is stored in bits that can only be either 0 or 1, quantum computers store information on qubits, which can exist either in state 0 or 1 or in the superposition of both states. Quantum has high processing power using the capability to be in multiple states, and simultaneously perform all possible permutations. Computers equipped with quantum computing will make it very easy to solve many complex problems like breaking RSA Encryption, searching a target number in O($\sqrt{N}$) operations, etc. This report will give an overview of quantum algorithms like Shor's algorithm and Grover's algorithm by including its classical, qiskit implementation and description that how it works.**

I. Introduction

Quantum algorithms are computational procedures designed to operate on quantum computers, leveraging the principles of quantum mechanics to perform certain tasks more efficiently than classical algorithms. Unlike classical computers, which process data using bits that are either 0 or 1, quantum computers use quantum bits or qubits, which can exist in a superposition of states, allowing them to represent and process multiple values simultaneously.

These algorithms are being researched for applications across diverse fields such as cryptography, optimization, machine learning, quantum chemistry, communication, and finance

Quantum algorithms exploit unique quantum phenomena such as superposition, entanglement, and interference to perform computations. One of the most famous quantum algorithms is Shor's algorithm, which efficiently factors large integers, a problem that is believed to be intractable for classical computers. Another significant quantum algorithm is Grover's algorithm, which can search an unsorted database quadratically faster than the best classical algorithms.

II. Shor's Algorithm

Shor's factoring algorithm finds one of two unknown variables that are crucial for efficiently factoring an integer. With two unknowns in one equation, finding both values quickly becomes classically intractable as the target integer gets larger.

Integer factorization is the decomposition of an composite integer into a product of smaller integers, for example, the integer 100 can be factored into 10×10. If these factors are restricted to prime numbers, the process is called prime factorization, for example, the prime factorization of 100 is $2 \times 2 \times 5 \times 5$.

When the integers are very large, no efficient classical integer factorization algorithm is known. The hardest factorization problems are semiprime numbers, the product of two prime numbers. The presumed difficulty of this semiprime factorization problem underlines many encryption algorithms, such as RSA, which is used in online credit card transactions, amongst other applications.

Shor's algorithm, named after mathematician Peter Shor, is a polynomial time quantum algorithm for integer factorization formulated in 1994.

A. Mathematics of the Algorithm

The number theory that underlines Shor's algorithm relates to periodic modulo sequences. Let's have a look at an example of such a sequence. Consider the sequence of the powers of two:

$$1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ...$$

Now let's look at the same sequence 'modulo 15', that is, the remainder after fifteen divides each of these powers of two:

$$1, 2, 4, 8, 1, 2, 4, 8, 1, 2, 4, ...$$

This is a modulo sequence that repeats every four numbers, that is, a periodic modulo sequence with a period of four. Reduction of factorization of N to the problem of finding the period of an integer x less than N and greater than 1 depends on the following result from number theory:

Theorem: The function $\mathcal{F}(a) = x^a \bmod N$ is a periodic function, where x is an integer coprime to N and $a \geq 0$.

Two numbers are coprime, if the only positive integer that divides both of them is 1. This is equivalent to their greatest common divisor being 1.

Since $\mathcal{F}(a)$ is a periodic function, it has some period r. Knowing that $x^0 \bmod N = 1$, this means that $x^r \bmod N = 1$ since the function is periodic, and thus r is just the first nonzero power where $x^r = 1 (\text{mod} N)$. Given this information and through the following algebraic manipulation:

$$x^r \equiv 1 \bmod N$$

$$x^r = (x^{r/2})^2 \equiv 1 \bmod N$$

$$(x^{r/2})^2 - 1 \equiv 0 \bmod N$$

and if r is an even number:

$$(x^{r/2} + 1)(x^{r/2} - 1) \equiv 0 \bmod N$$

From this, the product $(x^{r/2}+1)(x^{r/2}-1)$ is an integer multiple of N, the number to be factored. Thus, so long as $(x^{r/2}+1)$ or $(x^{r/2}-1)$ is not a multiple of N, then at least one of $(x^{r/2}+1)$ or $(x^{r/2}-1)$ must have a nontrivial factor in common with N.

So computing $\gcd(x^{r/2}-1,N)$ and $\gcd(x^{r/2}+1,N)$ will obtain a factor of N, where gcd is the greatest common denominator function, which can be calculated by the polynomial time

B. Classical Implementation

Algorithm 1 Classical Implementation of Shor's Algorithm

```
function SHORSALGORITHMCLASSICAL($N$)
    $x \leftarrow$ random$(0, N)$ {Step one: choose a random integer $x$}
    if $\gcd(x,N) \neq 1$ then {Step two: check if $x$ and $N$ are coprime}
        return $(x, 0, \gcd(x,N), N//\gcd(x,N))$
    end if
    $r \leftarrow$ findPeriodClassical$(x,N)$ {Step three: find the period of $x^k \mod N$}
    while $r \mod 2 \neq 0$ do {Ensure the period is even}
        $r \leftarrow$ findPeriodClassical$(x,N)$
    end while
    $p \leftarrow \gcd(x^{\frac{r}{2}}+1,N)$ {Step four: compute the factors}
    $q \leftarrow \gcd(x^{\frac{r}{2}}-1,N)$
    return $(x, r, p, q)$
end function

function FINDPERIODCLASSICAL($x, N$)
    $n \leftarrow 1$
    $t \leftarrow x$
    while $t \neq 1$ do
        $t \leftarrow (t \times x) \mod N$
        $n \leftarrow n + 1$
    end while
    return $n$
end function
```

1) Explanation: In Algorithm 1, we present a classical implementation of Shor's algorithm, which aims to factorize an integer N. Here's a breakdown of the steps involved:

1) **Random Selection:** Begin by selecting a random integer x between 0 and N.
2) **Check Coprimality:** Verify if x and N are coprime, i.e., their greatest common divisor (gcd) is 1. If not, return x along with its gcd with N.
3) **Finding Period:** Utilizing $findPeriodClassical(x,N)$ to determine the period of $x^k \mod N$, denoted by r. Ensure the period is even by repeating the process until $r \mod 2 = 0$.
4) **Compute Factors:** Calculate the factors p and q using the formulas: - $p = \gcd(x^{r/2}+1,N)$ - $q = \gcd(x^{r/2}-1,N)$

The $findPeriodClassical(x,N)$ function is responsible for finding the period of x modulo N. It iteratively calculates $t = (t \times x) \mod N$ until t equals 1, counting the number of iterations as n, which represents the period.

This classical implementation acts as a baseline for understanding Shor's algorithm and its quantum counterparts. While effective for smaller inputs, its time complexity grows exponentially with the size of N, making it inefficient for large integers.

C. Qiskit Implementation

Algorithm 2 Shor's Algorithm Pseudo Code

```
procedure CIRCUIT_AMOD15($qc, qr, cr, a$)
    if $a == 2$ then
        Apply controlled swaps
    else if $a == 7$ then
        Apply controlled swaps and controlled X gates
    else if $a == 8$ then
        Apply controlled swaps
    else if $a == 11$ then
        Apply controlled swaps and controlled X gates
    else if $a == 13$ then
        Apply controlled swaps and controlled X gates
    end if
end procedure
procedure CIRCUIT_APERIOD15($qc, qr, cr, a$)
    if $a == 11$ then
        Call Circuit_11period15
        return
    end if
    Initialize $q[0]$ to $|1\rangle$
    Apply $a^4$ mod 15
    Apply $a^2$ mod 15
    Apply $a$ mod 15
end procedure
procedure CIRCUIT_11PERIOD15($qc, qr, cr$)
    Initialize $q[0]$ to $|1\rangle$
    Apply $a^4$ mod 15
    Apply $a^2$ mod 15
    Apply 11 mod 15
end procedure
```

III. GROVER'S ALGORITHM

Grover's algorithm is a quantum search algorithm which in a given unsorted list of N elements enables us to find a target element with $O(\sqrt{N})$ operations, whereas a classical algorithm requires $O(N)$ operations. Therefore, it provides a quadratic speedup over its classical counterparts. Also, it has been applied as a subroutine for other quantum algorithms.

The search problem we consider is to find the index of the target element among the list of $N = 2^n$ elements, where n is the number of qubits and N is the size of the list. The procedure of Grover's algorithm is as follows:

1) Prepare $|0\rangle^{\otimes n}$ where $\otimes$ means tensor, i.e., $|0\rangle^{\otimes n}$ is equivalent to $|0\rangle \otimes |0\rangle \otimes \cdots \otimes |0\rangle$ with n terms.
2) Apply $H^{\otimes n}$ to create a superposition.

3) Apply the oracle O to mark the target element by negating its sign, i.e., $O|x\rangle = -|x\rangle$ where $|x\rangle$ is the target.
4) Apply the Grover diffusion operator D to amplify the probability amplitude of the target element.
5) Repeat Steps 3) and 4) for about $\sqrt{N}$ times.
6) Perform measurements.

After about $\sqrt{N}$ iterations of Steps 3) and 4), we will find the target element with a high probability. Grover's diffusion operator can be expressed as

$$2|\psi\rangle\langle\psi| - IN = H^{\otimes n}(2|0\rangle\langle 0| - IN)H^{\otimes n},$$

where $|\psi\rangle$ is the uniform superposition of states and IN is the N-dimensional identity matrix. As $2|\psi\rangle\langle\psi| - IN$ operates a reflection about the $|\psi\rangle$, $2|0\rangle\langle 0| - IN$ operates a reflection about the $|0\rangle$. It turns out that Grover's diffusion can be implemented on a quantum circuit with a phase shift operator that negates all the states except for $|0\rangle$ sandwiched between $H^{\otimes n}$ gates.

A. Analysis

Let θ be the angle between m and the current state a before any iteration of the inner loop. Suppose $\alpha \leq \theta < \frac{\pi}{2}$; note that initially $\theta = \alpha$. Because m is our x-axis, the Grover reflection puts a at angle $-\theta$, which is stepping back to spring forward. Because its distance from j is now $\alpha + \theta$, the reflection about j doubles that and adds it to $-\theta$. The new angle is hence

$$\theta' = -\theta + 2\alpha + 2\theta = \theta + 2\alpha,$$

which means that the two reflections have effected a positive rotation by 2α.

Now, in each iteration of the inner loop, θ increases by 2α. The probability of measuring the target state is maximized after every iterations.
Given r iterations of Grover's algorithm, the angle θ_r after the iterations will be $\alpha + r \cdot (2\alpha) = \frac{\pi}{2}$, where $\alpha = \frac{1}{\sqrt{N}}$.

So, we have:

$$\frac{1}{\sqrt{N}} + r \cdot 2\frac{1}{\sqrt{N}} = \frac{\pi}{2}$$

$$\frac{1+2r}{\sqrt{N}} = \frac{\pi}{2}$$

Solving for r:

$$r = \frac{\pi}{4\alpha} - \frac{1}{2}$$

Now, substitute $\alpha = \frac{1}{\sqrt{N}}$:

$$r = \frac{\pi}{4} \cdot \sqrt{N} - \frac{1}{2}$$

Therefore, the number of iterations r required for $\theta_r = \frac{\pi}{2}$ is approximately $\frac{\pi}{4}\sqrt{N}$, which makes it order of $\sqrt{N} operations$

B. Classical Implementation

1) Explanation:

1) **Initialization of Quantum State:** The function initialize_state initializes the quantum state (superposition) by assigning equal amplitudes to all states.
2) **Applying the Oracle Operation:** The function apply_oracle applies the oracle operation to mark the target states. It negates the amplitude of each marked state in the amplitudes array.
3) **Inversion About the Mean Operation:** The function invert_about_mean performs the inversion about the mean operation. It calculates the mean amplitude of all states and reflects each amplitude about this mean.
4) **Grover's Algorithm Simulation:** The function grover_algorithm simulates Grover's algorithm by iteratively applying the oracle and inversion operations for a specified number of iterations.
5) **Main Program Execution:** The main program initializes the number of states and qubits, and defines the marked states. Grover's algorithm is then run for a specified number of iterations. Finally, the program prints the final quantum state after the iterations.

Algorithm 3 Classical Grover Algorithm

```
function InitializeState(num_states)
    $sqrt\_n \leftarrow \frac{1.0}{\sqrt{num\_states}}$
    return Array of size $num\_states$ with amplitude $sqrt\_n$
end function
function ApplyOracle(amplitudes, marked_states)
    for $i$ from 0 to $num\_states - 1$ do
        if marked_states[$i$] is $true$ then
            $amplitudes[i] \leftarrow -amplitudes[i]$
        end if
    end for
end function
function InvertAboutMean(amplitudes)
    $mean \leftarrow \frac{\sum_{i=0}^{num\_states-1} amplitudes[i]}{num\_states}$
    for $i$ from 0 to $num\_states - 1$ do
        $amplitudes[i] \leftarrow 2.0 \times mean - amplitudes[i]$
    end for
end function
function GroverAlgorithm(amplitudes, marked_states, num_iterations)
    for $k$ from 1 to $num\_iterations$ do
        ApplyOracle(amplitudes, marked_states)
        InvertAboutMean(amplitudes)
    end for
end function
```

C. Qiskit Implementation

The below pseudo-code implements the Grover search algorithm using the Qiskit quantum computing framework. It initializes quantum and classical registers, constructs an oracle to mark target solutions, and iterates through Grover's algorithm to amplify the probability amplitudes of these solutions. After measurement, the results are visualized using histograms. This code serves as a practical demonstration of Grover's algorithm for solving unstructured search problems in quantum computing.

Algorithm 4 Grover Search Algorithm

```
Initialize quantum registers and classical registers
Apply Hadamard gate to the first 6 qubits
Apply X gate to the 6th qubit
Apply barrier
Building Oracle:
Apply X gate to the 2nd and 4th qubits
Apply barrier
Apply CCX gates to create the oracle
Apply X gate to the 2nd and 4th qubits
Apply barrier
Grover Iterations:
for $i$ in range(iterations) do
    Apply Hadamard gate to the first 5 qubits
    Apply X gate to the first 5 qubits
    Apply Hadamard gate to the last qubit
    Apply barrier
    Apply CCX gates
    Apply CX gate
    Apply CCX gates
    Apply Hadamard gate to the last qubit
    Apply barrier
    Apply X gate to the first 5 qubits
    Apply Hadamard gate to the first 5 qubits
    Apply barrier
end for
Measure the first 5 qubits and store the result in classical registers
```

IV. Functional Quantum Algorithm Framework for Shor's Algorithm

A. Core Vision

Our project aims to simplify quantum computing by providing a framework that abstracts away the intricacies of low-level quantum mechanics. Instead of getting bogged down in the complexities of quantum theory, developers can focus on building applications at a higher level, making quantum computing more accessible and manageable for a wider audience.

B. Key Features

Flexible Gate Implementation: Our framework allows developers to implement an arbitrary number of quantum gates, enabling the construction of diverse quantum circuits tailored to specific tasks or algorithms.

Scalable State Handling: With the capability to process arbitrary n-qubit states, our framework accommodates quantum computations of varying complexities, limited only by the memory resources of the host computer.

Shor's Algorithm Support: We offer pre-made functions designed specifically for executing Shor's algorithm, demonstrating the effectiveness and versatility of our framework in tackling complex quantum algorithms.

C. Description

Quantum computing is gaining more interest because of algorithms like Shor's, which dramatically speed up solving big math problems. We're creating a framework using OCaml that can mimic any quantum algorithm, including Shor's, given enough computer memory.

This tool works by taking a starting quantum state and a string that describes the quantum process you want to simulate. It then uses different kinds of gates to carry out the process and gives you the final quantum state. But, because quantum stuff needs a lot of memory, there is a limitation on the number of qubits it can handle.

To make building circuits easier, it provides basic gates like CNOT, along with more complex ones. Again there is a limitation for big numbers, we can't do it because it needs too much memory.

Additionally, Shor's algorithm needs some regular math steps like integer factorization, integer exponentiation,and converting to base 10 representation, so we'll include those too.

D. Architecture

The tool will adopt a pipe and filter system, initially processing user input into two possible pipelines: **utilizing Shor's algorithm for factoring** or creating a custom circuit via a **REPL** interface. If opting for a pre-built quantum algorithm, the program will accept algorithm parameters and provide both the final outcome and the required algorithm details.

Alternatively, users selecting the option to construct and execute their custom quantum circuit will be prompted to first design and save the circuit. Subsequently, they will input the initial state and the saved circuit's name. It will then execute the algorithm and give the circuit's output state alongside relevant performance metrics. Subsequently, users will return to the REPL interface, allowing for iterative exploration of quantum processes.

E. System and Module Design

So we've structured our system into five key modules, each serving a distinct purpose and interacting with one another as illustrated in the dependency diagram. We will take a look at each of the module below:

1) Arithmetic Module: This module contains essential arithmetic and number-theoretic functions crucial for implementing specific gates and components of Shor's factoring algorithm within our quantum computing framework. Below are the functions planned to be implemented in this module:

- Integer Exponentiation:
  Description: Computes the exponentiation of an integer.
  Function Signature: $int\_exp : int \to int \to int$

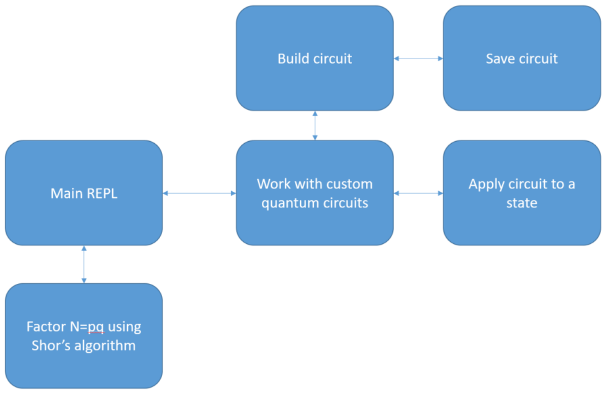
Build circuit
Save circuit
Main REPL
Work with custom quantum circuits
Apply circuit to a state
Factor N=pq using Shor's algorithm

Fig. 1. Workflow of the tool

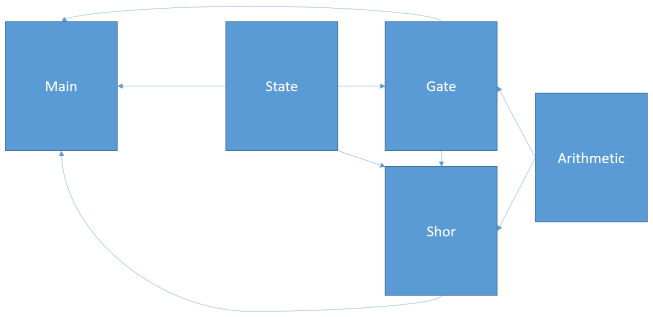

Fig. 2. Dependency graph of the module

- Integer Exponentiation with Modulus
  Description: Computes the exponentiation of an integer with a modulus.
  Function Signature: $int\_exp\_mod : int \rightarrow int \rightarrow int \rightarrow int$.

- Bitwise XOR Operation
  Description: Performs a bitwise XOR operation on two digit lists.
  Function Signature: $add\_mod\_2 : dlist \rightarrow dlist \rightarrow dlist$

- Splitting Digit List
  Description: Splits a digit list into two parts, with the length of the first part specified.
  Function Signature: $split : int \rightarrow dlist \rightarrow dlist * dlist$

- Base 10 Representation
  Description: Converts a number from base b to its base 10 representation.
  Function Signature: $base\_10\_rep : int \rightarrow dlist \rightarrow int$

- Base Conversion
  Description: Converts a number from one base to another represented by the digit list.
  Function Signature: $base\_convert : int \rightarrow int \rightarrow dlist \rightarrow dlist$

- Digit List to String
  Description: Converts a digit list to a string.
  Function Signature: $dlst\_to\_string : dlist \rightarrow string$

- Zero Padding
  Description: Adds zeros to the head of a digit list until it reaches the specified length.
  Function Signature: $pad\_zeros : int \rightarrow dlist \rightarrow dlist$

2) State: This module is dedicated for managing quantum states, defines necessary types and functions for representing and manipulating qubits within the quantum computer. Its functions are designed to interface with other modules for seamless processing.

- Make State:
  Description: Creates a quantum state from a list of complex numbers, where each element in the list corresponds to a coefficient of a state for its respective index.
  Function Signature: $val\, make : Complex.tlist \rightarrow st$

- State of
  Description: Returns the state of a specific qubit within a quantum system.
  Function Signature: $val\, state\_of : int \rightarrow int \rightarrow st.$

- Empty
  Description: Returns an empty quantum state with all coefficients set to zero.
  Function Signature: $val\, empty : int \rightarrow st$

- Splitting Digit List
  Description: Splits a digit list into two parts, with the length of the first part specified.
  Function Signature: $split : int \rightarrow dlist \rightarrow dlist * dlist$

- Number of Bits
  Description: Returns the number of qubits in a quantum state.
  Function Signature: $val\, n\_bits : st \rightarrow int$

- Zero
  Description: Returns a quantum state representing the zero qubit.
  Function Signature: $val\, zero : st$

- One
  Description: Returns a quantum state representing the one qubit.
  Function Signature: $val\, one : st$

- Tensor Product
  Description: Computes the tensor product of multiple quantum states.

Function Signature: $val\,tensor : stlist \rightarrow st$

- Normalize
  Description: Normalizes a quantum state based on its coefficients.
  Function Signature: $val\,normalize : st \rightarrow st$

- Add
  Description: Adds two quantum states.
  Function Signature: $val\,add : st \rightarrow st \rightarrow st$

- Phase
  Description: Adds an overall phase to all elements in a quantum state.
  Function Signature: $val\,phase : float \rightarrow st \rightarrow st$

- Coefficients
  Description: Returns the coefficients of a quantum state as a list of complex numbers.
  Function Signature: $val\,coeff : st \rightarrow Complex.tlist$

- Value String
  Description: Formats the coefficients of a quantum state into a string.
  Function Signature: $val\,val\_string : st \rightarrow string$

3) Gate: Now in this module we will handle quantum gates, which are operators responsible for transforming quantum states. Drawing parallels to classical logic gates, this module defines various gate types and functions for their composition.

- Gate Creation:
  Description: Creates a quantum gate based on an input string that represents the gate type and the qubits it acts on.
  Function Signature: $val\,gate\_of : string \rightarrow t$

- Controlled Gate
  Description: Generates a controlled gate where activation depends on the state of specified qubits.
  Function Signature: $val\,control : intlist \rightarrow t \rightarrow t$

- Apply Gate
  Description: Applies a gate to a quantum state and returns the resulting state.
  Function Signature: $val\,apply : t \rightarrow st \rightarrow st$

- Tensor Product
  Description: Computes the tensor product of multiple gates, representing their sequential application.
  Function Signature: $val\,product : tlist \rightarrow t$

- Gate to String
  Description: Converts a gate to a string representation for storage or serialization.
  Function Signature: $val\,gate\_to\_string : t \rightarrow string$

- String to Gate
  Description: Converts a string representation of a gate back into its gate object.
  Function Signature: $val\,string\_to\_gate : string \rightarrow t$

4) Shor: This module is focused on implementing Shor's factoring algorithm, this module integrates quantum states and gates to execute the algorithm efficiently. Leveraging functionalities from the Arithmetic module, it encapsulates the essential functions required for factoring integers. Further details are outlined in the "shor.mli" file.

- Factorization:
  Description: Simulates factoring an integer using Shor's algorithm and efficient classical algorithms, printing the prime factors and efficiency statistics.
  Function Signature: $val\,factor : int \rightarrow unit$

- Quantum Fourier Transform (QFT) Gate
  Description: Generates a Quantum Fourier Transform (QFT) gate, which applies the QFT to a subset of qubits out of a total number of qubits.
  Function Signature: $val\,qft\_gate : int \rightarrow int \rightarrow Gate.t$

5) Main: Main module acts the central coordinator, it has the following functionalities -

- REPL (Read-Eval-Print Loop):
  The REPL provides users with two primary choices:
  1) Building quantum circuits and applying states to them
  2) Factoring integers using Shor's algorithm

- Building Quantum Circuits:
  Users can build quantum circuits by listing out gates or sequences of gates separated by spaces. They can save the circuits they've built and view currently available circuits. The format for entering gates is provided, guiding users on how to construct quantum circuits.

- Applying Gates to States:
  Users can apply saved circuits to states, observing the resulting state after passing through the circuit. They can view available circuits, apply gates, or quit this mode.

- Integration with Shor's Algorithm:
  Users can factor natural numbers using Shor's algorithm, with the program providing statistics comparing its efficiency with classical algorithms. This functionality relies on the Shor module.

V. Comparative Analysis: Comparing Functional & Imperative Implementations in QC

A. Advantages of Functional Programming in Quantum Computing:

1) **Immutable State Management:** Leveraging OCaml's immutable data structures (e.g., in the State module) ensures quantum state integrity throughout computations. See state.mli for implementations.
2) **Higher-Order Functions:** Functional constructs in the Gate module ensures concise representation of quantum gates and operations.
3) **Pattern Matching:** OCaml's pattern matching simplifies quantum algorithm implementation, notably in the Main module's REPL interactions. We can use pattern matching for user input in the REPL.

B. Challenges with Functional Paradigms:

1) **Learning Curve:** Understanding OCaml's functional concepts, especially in quantum computing, may require time and effort. Explore the project's documentation and codebase to understand foundational quantum computing concepts and functional programming techniques.
2) **Efficient Implementation:** While functional programming offers good algorithm design, optimizing quantum algorithms for performance (e.g., in the Shor module) requires expertise in both quantum computing and functional programming which further adds a layer of complexity.
3) **Concurrency and Parallelism:** Functional programming also supports concurrency and parallelism, but effectively using these features in quantum computing needs careful planning. We can use concurrency in the Main module for circuit construction.

C. Comparative Performance Analysis:

1) **Resource Utilization:** We can compare memory usage and processing time between functional and imperative implementations that we will get after executing the circuit. Also we can analyze benchmarks and alter data from modules like Gate and Shor to see how the resource utilization changes.
2) **Scalability:** Evaluating performance across varying problem sizes and computational requirements using benchmarks from the Main module. Also analyze scalability differences between functional and imperative approaches for different quantum algorithms.

D. Insights into Imperative Implementation:

1) **Mutable State Management:** Imperative programming allows for mutable state management, which can sometimes be more intuitive for developers to analyze bugs or algorithms. Direct manipulation of states and gates may offer performance advantages in specific scenarios, especially for low-level optimizations or the legacy code.
2) **Explicit Control Flow:** Imperative programming also offers explicit control flow mechanisms, offering finer-grained control over program execution. This can be beneficial for optimizing critical sections of quantum algorithms and addressing specific performance bottlenecks.

E. Conclusion: Choosing the Right Approach for Quantum Computing:

1) **Hybrid Approaches:** While both functional and imperative programming paradigms have their advantages and challenges in quantum computing, a hybrid approach that combines the strengths of both can offer the best of both paradigms. Using functional programming for high-level abstraction and declarative modeling, combined with imperative techniques for low-level optimization and performance optimization, can lead to robust and efficient quantum computing solutions.
2) **Adaptability and Flexibility:** Ultimately, the choice between functional and imperative programming in quantum computing depends on the specific requirements of the application, the expertise of the development team, and considerations of performance, scalability, and maintainability.

VI. Challenges Faced

1) **Learning Hurdles:** Initially, we thought we'd jump straight into building, but we quickly realized there was a lot to learn. Quantum computing is complex, and we spent a good chunk of time studying before we could even think about coding. ***It felt more like a study project than an implementation one!***
2) **Conceptual Understanding:** We found quantum computing differs significantly from classical computing. Understanding even the basic concept in QC was a challenging task for any one who is new to QC, ***concepts such as superposition, entanglement, and quantum gates required careful study before they could be effectively applied in the project.***
3) **Internal Working of Qiskit**: While familiar with the input and output of Qiskit, IBM's quantum computing library, but trying to understand how it internally works is a challenge to us. As ***without understanding the underlying mechanism and algorithm within Qiskit it's impossible to implement those low-level functions.***

   Despite these challenges, the project provided valuable insights into quantum computing principles and methodologies, contributing to the ongoing advancement of quantum computing research and development. Through perseverance, collaboration, and innovative problem-solving, the project team successfully navigated these challenges, ultimately delivering a functional and impactful solution in the field of quantum computing.