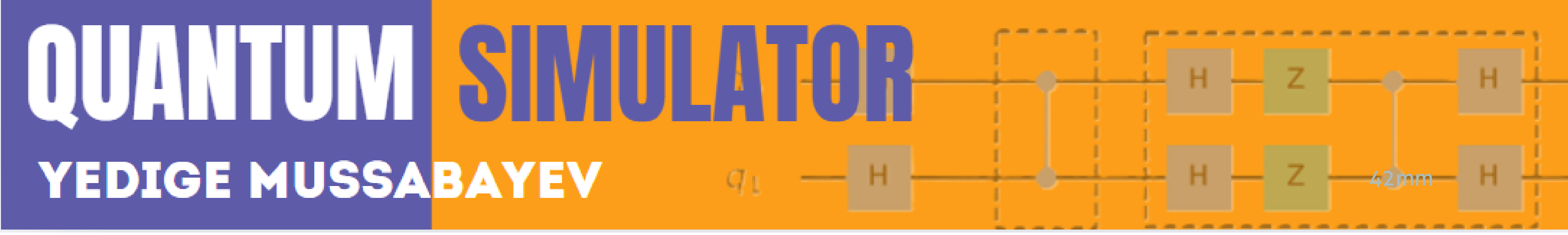


RESEARCH PAPER

Quantum Simulator. Algorithmization and modeling of foundational concepts from quantum physics on Python language.

Mussabayev Yedige[1]

[1]Second-year undergraduate student in Computer Systems and Software Engineering. International Information Technologies University. Almaty. Kazakhstan.

Received 1 Month 2000; revised 1 Month 2000; accepted 1 Month 2000

tabularx booktabs float placeins
listings enumitem amsmath, amssymb

This paper presents a Python-based quantum simulator designed to algorithmically model fundamental concepts of quantum mechanics. The simulator provides a programmable environment to visualize and test essential quantum operations such as superposition, entanglement, and quantum gate transformations, including Hadamard, Pauli, Grover's search, and the Quantum Fourier Transform. By leveraging matrix mechanics and quantum circuit formalism, the system enables step-by-step tracking of qubit states and their evolution. Designed for educational and research use, the simulator emphasizes accessibility and clarity, offering interactive outputs and modular code for experimentation.

OBJECTIVES The main goal of this work is to demonstrate the practical applicability of quantum computing concepts on classical computers that operate using binary bits. A secondary objective is the development of a fully functional quantum simulator.

METHODS In this work, foundational quantum computing concepts were algorithmized and modeled using classical computational techniques. These methods formed the core of the research, enabling the simulation of key quantum behaviors within a Python-based environment.

RESULTS The main outcome of this study is a working quantum simulator developed in Python, which supports essential quantum gates and algorithms. During the research, the implementation of crucial quantum operations was achieved, including the Hadamard gate, Pauli gates (X, Y, Z), and phase gates (S and T). Simulator optimization allows operation with 20 or more qubits without a critical increase in computing resource usage. To enhance interpretability, the simulator includes graphical visualizations of quantum processes.

CONCLUSIONS This work demonstrates the applicability of quantum computing principles in classical systems through the algorithmization and execution of quantum operations. The developed simulator serves both educational and practical purposes, validating the potential of quantum algorithms in a classical environment.

KEYWORDS Quantum simulator, Grover's algorithm, Hadamard gate, qubit, superposition, quantum entanglement

1. INTRODUCTION

Quantum computing opens up new prospects in information processing, but the complexity of its implementation on real hardware necessitates the development of effective modeling tools. This project aims to create a quantum simulator capable of replicating the fundamental concepts of quantum physics and executing basic quantum algorithms. The simulator enables the study of quantum system behavior, testing of algorithms prior to execution on real quantum computers, and detailed analysis of their performance. The project addresses the limited accessibility of quantum computing by providing researchers and students with a convenient tool for training and experimentation in the field of quantum information science. Quantum computing represents a rapidly evolving discipline focused on solving problems that are intractable for classical computers. Unlike classical systems that use binary bits, quantum computers operate with qubits, which can represent superpositions and entangled states. To explore these principles in accessible environments, quantum simulators act as practical tools for education, experimentation, and early-stage algorithm design. In this work, we present a *Python-based quantum simulator* that models foundational quantum computing concepts. The simulator supports essential quantum gates *(Hadamard, Pauli-X/Y/Z, Phase-S, Phase-T), multi-qubit operations such as CNOT, and quantum algorithms like Grover's search.* The simulator features a command-driven interface and includes graphical visualization of qubit states and measurement results. The goal of this work is to demonstrate that complex quantum operations can be effectively algorithmized and simulated using traditional programming techniques, thereby bridging theoretical quantum physics with practical computing.

*Correspondence: yedige.mussabayev@gmail.com

2. MATERIALS AND METHODS

The core of this research is the development of a quantum simulator implemented in Python. The simulator models fundamental quantum computing principles through algorithmic representations of qubits, gates, and measurement procedures. It was developed using only built-in Python libraries to ensure cross-platform compatibility and educational accessibility. The simulator defines an n-qubit system using a complex-valued state vector, initialized to the $|0\rangle^n$ state. Quantum gates are implemented as matrix operators, including the Hadamard, Pauli-X, Pauli-Y, Pauli-Z, Phase-S, Phase-T, and CNOT gates. Multi-qubit operations are handled through tensor product expansions and Kronecker manipulations to apply gates across specific qubits within a system of arbitrary size. Grover's search algorithm is implemented as a practical application of the simulator, demonstrating amplitude amplification and iterative inversion about the mean. The simulator includes functions for building an oracle, applying Grover iterations, and performing measurements with output probabilities. To facilitate user interaction, a command-driven interface allows users to write simple quantum scripts using predefined commands (e.g., 'h q[0];', 'cx q[0:1];', 'measure q[0];', etc.). The simulator parses these scripts, applies corresponding quantum transformations, and visualizes the system's state. Additionally, matplotlib-based visualization is provided to display the probability distribution of measured states and the real and imaginary parts of the system's state vector. These visual tools assist users in understanding quantum behaviors such as superposition and entanglement. Overall, the methods used involve algorithmic construction of quantum logic, efficient matrix handling for state evolution, and graphical feedback for interpretation—all realized without the use of external quantum frameworks or simulation libraries.

3. THEORETICAL BACKGROUND

3.1 How much the quantum computers are better in general?

Suppose we are given a Boolean function $f(x)$ that takes as input an integer from the set $\{1, 2, 3, \ldots, N-1\}$ and returns 'True' only for a single unknown value, say $n = 15$, and 'False' otherwise. In Python, such a function could be defined as:

```
def f(n):
    return n == 15
```

If we evaluate this function on a classical computer, the average number of evaluations required to find the correct n is approximately $\frac{1}{2}N$, since on average half of the inputs must be checked before finding the correct one.

In computer science, the symbol O (big-O notation) is commonly used to describe the upper bound of an algorithm's runtime, focusing on how the runtime scales with input size.

The **BBBV Theorem** (Bennett, Bernstein, Brassard, and Vazirani, 1994) established that no quantum algorithm can solve this unstructured search problem faster than:

$$\text{Quantum Search} \geq \Omega(\sqrt{N})$$

Later, it was shown that Grover's algorithm achieves exactly this bound, requiring approximately:

$$O(\sqrt{N}) \text{ evaluations}$$

This demonstrates a **quadratic speedup** over the classical approach, making Grover's algorithm one of the most significant results in quantum computing for unstructured search problems.

3.2 Quantum Computing Fundamentals

In traditional or classical computing, a single bit can be thought of as a piece of binary information, notated as either a 0 or a 1. Modern computers typically represent bits as either an electrical voltage or current pulse (or by the electrical state of a flip-flop circuit). In these systems, when there is no current flowing, the circuit can be considered to be off, and this state is represented as a 0. When current is flowing, the circuit is considered on, and this state is represented as a 1.

While quantum technologies do use binary code, the quantum data derived from a quantum system—such as a **qubit** — encodes data differently from traditional bits, with a few remarkable advantages Since each bit can represent either a 0 or a 1, by pairing two bits of information, we can create up to four unique binary combinations: 0 0 0 1 1 0 1 1 While each bit can be either a 0 or a 1, a single qubit can be either a 0, a 1, or a superposition. A quantum superposition can be described as both 0 and 1, or as all the possible states between 0 and 1 because it actually represents the probability of the qubit's state.

The abstraction levels of classical computers in **comparison** with quantum computers have well-known hardware elements(such as CPU, memory etc), its bits as the result of measurements of current for instance and data types in which bits are converted (numbers, letters etc). Quantum computers have

3.3 Abstraction Layers of Digital and Quantum Computers

To better understand quantum computing, we briefly compare it with classical computing by examining three abstraction layers in both systems:

1. **Hardware implementation**,
2. **Bits (classical) or qubits (quantum)** as measurement results,
3. **Data types** (e.g., numbers, characters).

Classical Computers

- **Hardware**: Classical computers use transistors, where electric current determines a bit's value.
- **Bits**: The transistor state (on/off) is converted into a classical bit (0 or 1) depending on either presence of absence of a current.
- **Data types**: Bits are processed into higher-level data types such as integers, floats, etc.

Quantum Computers

On the other hand, the hardware of a quantum computer strongly depends on the specific physical approach used to

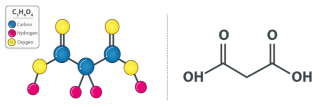


FIGURE 1. Malonic Acid

realize it. Among the various implementations, let us consider **Nuclear Magnetic Resonance (NMR)**, where an NMR spectrometer is used to measure and control the magnetic field interactions in the nuclei of molecules. The spectrometer contains a magnet that is cooled to extremely low temperatures using **liquid helium and nitrogen**. Inside the magnet, a **crystal of malonic acid** is placed, whose **carbon nuclei** serve as qubits. These carbon nuclei exhibit a quantum property known as **nuclear spin**, which causes them to behave like tiny magnets. This spins tend to align either parallel or antiparallel on the field due to Zeeman splitting. They align with the **Magnetic field** to create state $|0\rangle$ and against the Magnetic field to create state $|1\rangle$. Due to having magnetic properties, nuclei affect one another, which results in entanglement.

- **Hardware**:
  - A superconducting **magnet**, cooled with liquid nitrogen/helium.
  - A **malonic acid** crystal ($CH_2(COOH)_2$) with ^{13}C nuclei as qubits.
  - Nuclear spins align with the external magnetic field (Zeeman effect).
- **Qubits**:
  - Parallel alignment: $|1\rangle$ (higher energy).
  - Anti-parallel alignment: $|0\rangle$ (lower energy).
  - Exist in superpositions $\alpha|0\rangle + \beta|1\rangle$.
- **Measurement**:
  - Radiofrequency pulses manipulate states.
  - Final measurement yields probabilities for $|0\rangle$ and $|1\rangle$.

Key Differences

Aspect	**Classical**	**Quantum (NMR)**
Basic Unit	Transistor (bit)	Nuclear spin (qubit)
State	0 or 1	$\alpha|0\rangle + \beta|1\rangle$
Initialization	Voltage levels	Spin alignment
Measurement	Deterministic	Probabilistic

When you run a program on a quantum computer, the program doesn't necessarily define a specific output. Instead, it determines a probability distribution across possible outputs.

One important point for those new to quantum computing is that once you read (or measure) the quantum memory, you obtain a particular value, and the underlying state of the computer collapses—meaning all of the probability becomes concentrated on the value you just observed. As a result, if you continue reading (measuring) the system, you will keep getting the same value.

We might say that quantum programs create a very delicate and sensitive probability distribution, which collapses to a single outcome the moment you observe it—that is, the moment you sample from the distribution.

This leads us to the key question: **Where does this distribution come from?**

3.4 State Vector

The state of a quantum system is defined by its state vector. Each component of the state vector corresponds to one of the possible outcomes you might obtain upon measurement—that is, one of the possible bit strings. For example, if we are working with 5 qubits, the state vector has 32 components

$$2^5 = 32$$

each corresponding to one of the 32 possible 5-bit strings. Importantly, the state vector does not directly represent the probability distribution over the outputs. Instead, the probability of measuring a specific outcome (i.e., a specific bit string) is given by the squared magnitude (modulus squared) of the corresponding complex amplitude in the state vector. So, if we take the complex amplitude associated with a particular basis state and compute its modulus squared, we obtain the probability of observing that output upon measurement.

When a qubit is measured, its quantum state $|\psi\rangle$ collapses onto one of the computational basis states, either $|0\rangle$ or $|1\rangle$, with probabilities determined by the **Born Rule**:

$$P(\text{result}) = |\langle \text{result}|\psi\rangle|^2.$$

This means that if the qubit is in a superposition state

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

then the probability of obtaining $|0\rangle$ is $|\alpha|^2$ and the probability of obtaining $|1\rangle$ is $|\beta|^2$.

After the first measurement, the qubit's state becomes *exactly* the observed basis state, and any subsequent measurements in the same basis will yield the same result with 100% certainty. The original superposition can only be restored by applying further quantum gates to reorient the qubit state on the Bloch sphere.

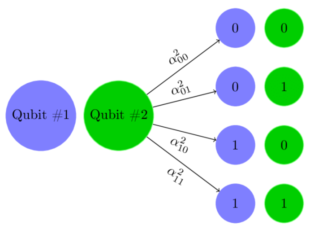


FIGURE 2. A two-qubit system can collapse into one of four states

This collapse phenomenon is not unique to abstract qubits. It has direct analogies in physical systems, such as:

- **Electron spin measurements**: once the spin is measured along the z-axis (up or down), it remains in that eigenstate unless manipulated by an external magnetic field.
- **Proton polarization**: once the polarization state is de-

termined, repeated measurements in the same direction yield identical results.

These examples highlight the key feature we will use computationally: a controllable two-level system whose state collapses to a measurement eigenstate and is repeatable under the same basis.

3.5 Qubits

A qubit is the abstract version of such a two-level system. We denote its computational basis by $\{|0\rangle, |1\rangle\}$, and write a general pure state as $\alpha|0\rangle + \beta|1\rangle$ with $|\alpha|^2 + |\beta|^2 = 1$.

The standard computational basis consists of two orthonormal states, denoted by *Dirac notation* (or *bra–ket* notation) as:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

A general pure state of a qubit is a linear combination (superposition) of these basis states:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

where α and β are complex numbers satisfying the normalization condition

$$|\alpha|^2 + |\beta|^2 = 1.$$

Here, the **ket** $|\psi\rangle$ represents a column vector describing the state of the system. The corresponding **bra** $\langle\psi|$ is its Hermitian conjugate (complex conjugate transpose), represented as a row vector:

$$\langle\psi| = \alpha^*\langle 0| + \beta^*\langle 1|, \quad \langle 0| = [\,1 \quad 0\,], \quad \langle 1| = [\,0 \quad 1\,].$$

The bra–ket notation is compact and powerful:

- **Inner product:** $\langle\phi|\psi\rangle$ produces a complex scalar measuring the overlap between two states.
- **Outer product:** $|\psi\rangle\langle\phi|$ produces an operator (matrix) that maps states to states.

Physically, measuring a qubit in the computational basis yields $|0\rangle$ with probability $|\alpha|^2$ and $|1\rangle$ with probability $|\beta|^2$. The act of measurement collapses the qubit into one of these eigenstates, similar to the spin or polarization examples discussed earlier.

To build intuition for the probabilistic nature of qubits, consider an ideal classical coin. Before observing the result of a toss, we can only assign probabilities for the outcomes *heads* (H) or *tails* (T). For example:

$$P(H) = 0.5, \quad P(T) = 0.5.$$

Once the coin lands and we look at it, the outcome becomes definite — either H or T — and repeated observations without flipping again will always give the same result.

A qubit measured in the computational basis $\{|0\rangle, |1\rangle\}$ behaves similarly in the sense that it produces one of two possible results with certain probabilities:

$$P(0) = |\alpha|^2, \quad P(1) = |\beta|^2.$$

However, the quantum case has a crucial difference: **before measurement**, the qubit is not simply in one state or the other with some hidden probability, but in a genuine superposition

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

which can give rise to interference effects not possible with a classical coin. Furthermore, we can choose to measure the qubit in other bases (analogous to tossing the coin onto an edge), where the probabilities change according to the qubit's amplitudes.

3.6 Quantum Algorithms

While everyone is familiar with logic gates in classical computers, quantum computers operate using analogous elements known as **quantum gates**. These gates are the fundamental building blocks of quantum circuits and are represented by unitary matrices that evolve the state of qubits in a reversible manner.

Unlike classical gates that operate on bits (0 or 1), quantum gates manipulate qubits in superposition, allowing for operations across multiple states simultaneously. This is the source of the quantum computer's power in solving specific classes of problems more efficiently than classical algorithms.

Let us now explore some of the most well-known quantum algorithms and how they exploit quantum principles such as superposition, entanglement, and interference.

3.6.1 1. Deutsch–Jozsa Algorithm

This is one of the earliest examples demonstrating exponential speed-up. The goal is to determine whether a given function $f : \{0,1\}^n \to \{0,1\}$ is **constant** (same output for all inputs) or **balanced** (returns 0 for half the inputs, 1 for the other half).

Classically, one needs to evaluate the function on up to $2^{n-1} + 1$ inputs in the worst case.

Quantumly, the Deutsch–Jozsa algorithm determines the answer with only one function evaluation using:

- Superposition of all inputs
- A quantum oracle that encodes the function f
- Quantum interference to amplify the correct outcome

3.6.2 2. Grover's Search Algorithm

Grover's algorithm allows us to search an unsorted database of N entries in only $O(\sqrt{N})$ time, compared to $O(N)$ classically.

It works by:

- Initializing a superposition of all possible inputs
- Applying an oracle to mark the correct solution
- Using the Grover diffusion operator to amplify the probability of the correct result

This algorithm provides a quadratic speed-up and is useful for a wide range of search-type problems.

3.6.3 3. Shor's Algorithm

Shor's algorithm solves the problem of integer factorization exponentially faster than the best known classical algorithms.

Given an integer N, it finds its prime factors by reducing the problem to period finding and using the Quantum Fourier Transform (QFT).

- Classically: No known polynomial-time algorithm
- Quantumly: Shor's algorithm runs in polynomial time ($O((\log N)^3)$)

Its potential to break widely used cryptographic systems (e.g., RSA) makes Shor's algorithm one of the most significant results in quantum computing.

3.6.4 4. Quantum Fourier Transform (QFT)

The QFT is the quantum analog of the classical discrete Fourier transform (DFT). It plays a central role in many quantum algorithms, including Shor's.

The QFT transforms amplitudes of a quantum state and is defined as:

$$|x\rangle \rightarrow \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{2\pi i x y/2^n} |y\rangle.$$

Efficiently implementable with $O(n^2)$ gates, it is used to detect periodicities in quantum states.

3.6.5 5. Quantum Phase Estimation

This algorithm estimates the phase ϕ in the eigenvalue $e^{2\pi i\phi}$ of a unitary operator U, given an eigenvector $|\psi\rangle$ such that $U|\psi\rangle = e^{2\pi i\phi}|\psi\rangle$.

It is a key subroutine in:

- Shor's algorithm
- Quantum simulations
- Solving linear systems (as in HHL algorithm)

3.7 Theoretical Foundations of Gates in the Simulator

This section provides a conceptual and mathematical overview of all quantum gates and algorithmic building blocks implemented in the simulator. The goal is to understand how each gate transforms quantum states, how multi-qubit operations create entanglement or phase relationships, and how algorithmic primitives are constructed from fundamental quantum principles. Quantum logic gates are the building blocks of quantum circuits, like classical logic gates are for conventional digital circuits. An important mention of quantum gates is that they are **reversible**. Moreover, all quantum operations have to be reversible because the wave function must evolve without loss of information. Quantum operations are described by unitary operations.

1. Unitary Operators and Information Preservation Let a qubit or a system of qubits be in a state $|\psi\rangle$ in a Hilbert space $\mathcal{H}$. When a quantum gate U acts on this state, it transforms it as:

$$|\psi'\rangle = U|\psi\rangle.$$

The operator U is **unitary**, meaning it satisfies:

$$U^\dagger U = UU^\dagger = I,$$

where $U^\dagger$ is the conjugate transpose (Hermitian adjoint) of U, and I is the identity matrix.

This condition implies: - Norm preservation: $\|U|\psi\rangle\| = \||\psi\rangle\|$, so probabilities remain valid. - Invertibility: Every unitary operation has an inverse, $U^{-1} = U^\dagger$.

Thus, given the output of a unitary operation, one can always apply the inverse and recover the original input state without any information loss.

2. Why Quantum Gates Must Be Reversible The reversibility of quantum gates arises from two key physical requirements:

- **Deterministic evolution**: In a closed system (without measurement), the wave function evolves smoothly and predictably over time.
- **Conservation of probability**: The total probability (i.e., the sum of $|\alpha_i|^2$ over all components) must remain equal to 1. This is preserved by unitary operations but not by arbitrary functions.

As a result, quantum gates must be described by unitary matrices to ensure the overall evolution remains reversible and probability-preserving.

3. Contrast with Classical Gates Most classical logic gates are not reversible: - The AND gate maps $(1, 1) \rightarrow 1$, but the output 1 could come from many different inputs. - Therefore, we cannot uniquely recover the input from the output.

In contrast, all quantum gates (e.g., Hadamard, Pauli, CNOT) are reversible:

$$H^\dagger = H, \quad X^\dagger = X, \quad \text{CNOT}^\dagger = \text{CNOT}.$$

4. Connection to the Wave Function and Lossless Evolution The state of a quantum system is encoded in its **wave function** (or state vector), which contains all information about the system. To preserve the validity of quantum mechanics, this wave function must evolve in a way that does not lose or overwrite information — otherwise, we would violate the principle of quantum determinism.

This is why quantum gates must be: - Linear: to ensure superposition is respected. - Unitary: to ensure no information is lost or created, and evolution remains reversible.

Therefore, the use of unitary operators to model gates is not a choice but a necessity — it guarantees the evolution of the wave function is both norm-preserving and invertible.

Single-Qubit Gates

Quantum gates are unitary operations that evolve the quantum state vector in Hilbert space. For single qubits, they are represented as 2×2 unitary matrices.

Pauli Gates These gates represent fundamental operations corresponding to the Pauli matrices:

- **Pauli-X (X)**: Bit-flip gate

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad X|0\rangle = |1\rangle, \quad X|1\rangle = |0\rangle.$$

- **Pauli-Y (Y)**: Bit-flip + phase-flip

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Y|0\rangle = i|1\rangle, \quad Y|1\rangle = -i|0\rangle.$$

- **Pauli-Z (Z)**: Phase-flip gate

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad Z|0\rangle = |0\rangle, \quad Z|1\rangle = -|1\rangle.$$

These form the basis of all single-qubit quantum operations and correspond to π rotations around the Bloch sphere axes.

Hadamard Gate (H) The Hadamard gate creates superposition by rotating the state vector around the $x + z$ axis:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

It maps basis states as:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

This gate is critical in creating quantum parallelism.

Phase Gates: S, T, and General Phase These gates rotate the phase of the $|1\rangle$ state:

- $S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$ applies a $\pi/2$ phase.
- $T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$ applies a $\pi/4$ phase.
- $S^\dagger$ and $T^\dagger$ are their respective inverses.
- **General phase shift** $S(\theta)$:

$$P(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}, \quad \text{with } \theta = \frac{\pi}{k}.$$

These gates change the relative phase of superposed states, which is crucial for interference.

Identity Gate The identity gate $I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ leaves the qubit unchanged. It is often used for padding or timing analysis.

Two-Qubit Gates

Controlled-NOT (CNOT) Gate The CNOT gate flips the target qubit q_j only if the control qubit q_i is in state $|1\rangle$:

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

It creates entanglement when applied after a Hadamard gate, such as in Bell state preparation:

$$H|0\rangle \xrightarrow{\text{CNOT}} \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

Controlled Phase Shift $C(S(\theta))$ This gate applies a phase shift $e^{i\theta}$ to the target qubit, conditioned on the control qubit being $|1\rangle$. It has the form:

$$\text{Controlled-}P(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{bmatrix}.$$

These gates are essential for building interference patterns and controlled rotations in algorithms such as Quantum Phase Estimation.

Quantum Fourier Transform (QFT)

The QFT maps a computational basis state $|x\rangle$ to a superposition:

$$|x\rangle \to \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{2\pi i x y / 2^n} |y\rangle.$$

It consists of a sequence of Hadamard and controlled-phase gates and is used in algorithms like Shor's for period finding.

The inverse QFT (IQFT) is simply the Hermitian adjoint of QFT and is used for uncomputation of phases.

Algorithmic Primitives

Bell State Preparation Entanglement is achieved by:

$$Hq[0]; \quad \texttt{cx q[0], q[1]} \quad \Rightarrow \quad \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

Phase Estimation This algorithm estimates the phase ϕ in $U|\psi\rangle = e^{2\pi i\phi}|\psi\rangle$ using:

- A control register in superposition
- Controlled-U^{2^k} operations
- Application of QFT

Grover's Oracle and Diffusion Grover's algorithm includes two key steps:

- **Oracle:** Flips the sign of the target state: $|x\rangle \to -|x\rangle$
- **Diffusion:** Reflects amplitudes about their average using:

  $$D = 2|\psi\rangle\langle\psi| - I$$

  where $|\psi\rangle$ is the uniform superposition.

Quantum Arithmetic Using general and controlled phase shifts (such as $P(\pi/k)$), the simulator allows implementation of modular addition and phase-based arithmetic circuits, foundational to Shor's algorithm.

Measurement and Collapse

Measurement collapses a superposed state to one of the computational basis states. If a qubit is in state $\alpha|0\rangle + \beta|1\rangle$, measurement in the computational basis yields:

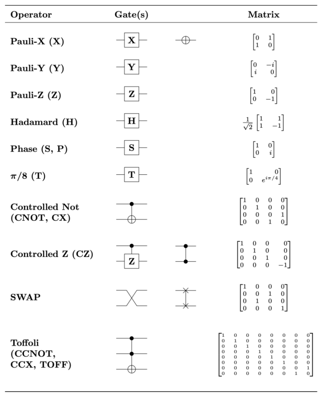
$$\text{Outcome } |0\rangle \text{ with probability } |\alpha|^2, \quad |1\rangle \text{ with probability } |\beta|^2.$$

After measurement, the qubit collapses to the observed eigenstate.

Noise Modeling

To simulate realistic behavior, the simulator introduces decoherence via random Pauli gates. These mimic real-world imperfections:

- Bit-flip (X), phase-flip (Z), or both (Y) applied stochastically.
- Useful for testing error resilience.



Operator	Gate(s)		Matrix
Pauli-X (X)	X		$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$
Pauli-Y (Y)	Y		$\begin{bmatrix}0 & -i\\i & 0\end{bmatrix}$
Pauli-Z (Z)	Z		$\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$
Hadamard (H)	H		$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$
Phase (S, P)	S		$\begin{bmatrix}1 & 0\\0 & i\end{bmatrix}$
$\pi/8$ (T)	T		$\begin{bmatrix}1 & 0\\0 & e^{i\pi/4}\end{bmatrix}$
Controlled Not (CNOT, CX)			$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$
Controlled Z (CZ)	Z		$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&-1\end{bmatrix}$
SWAP			$\begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)			$\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\end{bmatrix}$

FIGURE 3. Common quantum logic gates by name (including abbreviation), circuit form(s) and the corresponding unitary matrices

3.8 Theoretical Foundations of Algorithms in the Simulator

Although the simulator operates purely at the gate level and does not implement complete algorithms as predefined routines, it provides the necessary quantum gate primitives to construct and simulate the core components of well-known quantum algorithms. These building blocks demonstrate key quantum phenomena such as superposition, entanglement, interference, and phase estimation.

Bell State Preparation

One of the simplest yet fundamental quantum circuits is the creation of a **Bell state**, which represents an entangled two-qubit system. The Bell's states are a form of entangled and normalized basis vectors. This normalization implies that the overall probability of the particles being in one of the mentioned states is 1: [illegible]|[illegible]=1

The circuit consists of:

- Applying a Hadamard gate to qubit 0:

$$H|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

- Followed by a CNOT gate with control qubit 0 and target qubit 1:

$$\text{CNOT}\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle\right) \rightarrow \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

This state exhibits perfect entanglement: measurement of one qubit instantaneously determines the outcome of the other, regardless of distance. Bell states are essential in quantum teleportation and superdense coding.

Phase Estimation Component

The simulator enables manual construction of the core elements of the **Quantum Phase Estimation (QPE)** algorithm. This procedure estimates the eigenvalue (phase) ϕ from a unitary operator U such that:

$$U|\psi\rangle = e^{2\pi i\phi}|\psi\rangle.$$

The key steps implemented via simulator gates include:

- Initialization of control qubits in a superposition using Hadamard gates.
- Application of controlled unitary operations U^{2^k} via repeated controlled phase gates (`csk`).
- Inverse Quantum Fourier Transform (`IQFT`) to extract the phase from the interference pattern.

This process is foundational in many quantum algorithms, including Shor's algorithm and quantum simulations of physical systems.

Grover's Oracle Step and Diffusion Operator

While a full implementation of **Grover's algorithm** requires several iterations, the simulator supports its key components:

- **Oracle Construction:** The oracle inverts the amplitude of a marked state:

$$|i\rangle \rightarrow -|i\rangle,$$

  implemented as a sign-flipping operation (`Sign i`).
- **Diffusion Step:** The inversion-about-the-mean operation, essential for amplitude amplification, is constructed using:
  - Global Hadamard transforms
  - Conditional sign flips
  - Another global Hadamard

These two operations form the core iterative loop in Grover's search algorithm, which achieves quadratic speedup over classical search.

Quantum Arithmetic and Modular Circuits

The simulator supports **quantum arithmetic** through custom phase rotation gates:

- `sk q[i], k` and `csk q[i], q[j], k` implement rotations by angles $\theta = \pi/k$ and controlled-phase shifts.
- These gates can be combined to construct:
  - Modular adders
  - Controlled multipliers
  - Exponentiation circuits

Such arithmetic circuits are critical subcomponents in Shor's algorithm for integer factorization, where modular exponentiation is performed under superposition.

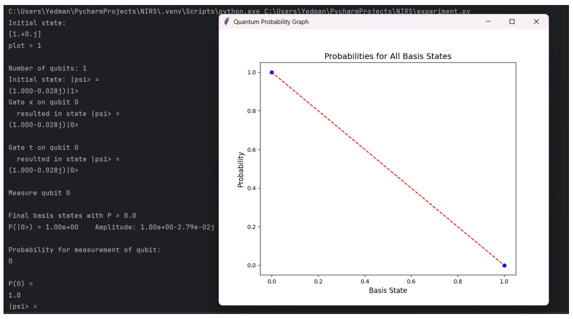
Noise Simulation

To approximate realistic quantum behavior, the simulator can introduce artificial noise by injecting random single-qubit Pauli gates:

```
x q[i],   y q[i],   z q[i].
```

These simulate:

- **Bit-flip errors (X)**
- **Phase-flip errors (Z)**
- **Combined bit and phase flips (Y)**



```
C:\Users\Yedman\PycharmProjects\NIRS\.venv\Scripts\python.exe C:\Users\Yedman\PycharmProjects\NIRS\experiment.py
Initial state:
[1.+0.j]
plot = 1

Number of qubits: 1
Initial state: |psi> =
(1.000-0.028j)|1>
Gate x on qubit 0
  resulted in state |psi> =
(1.000-0.028j)|0>

Gate t on qubit 0
  resulted in state |psi> =
(1.000-0.028j)|0>

Measure qubit 0

Final basis states with P > 0.0
P(|0>) = 1.00e+00    Amplitude: 1.00e+00-2.79e-02j

Probability for measurement of qubit:
0

P(0) =
1.0
|psi> =
```

FIGURE 4. Noise instructions simulator and graph

This functionality allows users to test the resilience of their circuits under decoherence and gate imperfections — a key concern in near-term quantum devices.

4. QUANTUM SIMULATOR

4.1 NOTE

Until the point where the quantum simulator itself is introduced, all preceding material was intended to provide background, clarify theoretical foundations, and familiarize the reader with the basic principles of quantum computing. This project was originally prepared as part of the Scientific Research Work of Students (НИРС 2025), where it was awarded 1st place in the intra-university competition and received a letter of appreciation from the Ministry at the republican-level contest. The work was carried out under the supervision of Artem Bykov, Candidate of Technical Sciences, Associate Professor of the Department of Computer Engineering.

4.2 SRWS's Overview

4.2.1 Abstract

Quantum computing is one of the most promising areas of modern science and technology, opening up new opportunities in information processing, modeling complex physical systems and solving problems that are inaccessible to classical computers. However, due to the high cost and technical limitations of real quantum computers, quantum simulators play an important role in their study and training of specialists.The quantum simulator being developed is designed to process tests and execute specified sequences of commands, simulating fundamental concepts of quantum mechanics, such as the superposition principle, quantum entanglement, quantum measurement, interference, amplitude gain (Grover's algorithm), quantum teleport and quantum Fourier transform.

4.2.2 General concept of the project

Project goal: To develop a quantum simulator capable of simulating and analyzing basic quantum physics concepts and algorithms in Python and proving quantum physics concepts in a computing environment. Project objectives. To achieve the project objective – development of a quantum simulator for modeling basic concepts of quantum physics – it is necessary to solve the following interrelated tasks:

1. Implementation of basic quantum operations (gates).
2. Initialization and measurement of quantum states.
3. Implementation of quantum entanglement
4. Grover's Algorithm
5. Quantum Teleport
6. Quantum Fourier Transform (QFT).
7. Visualization of quantum states.
8. User Interface Development

4.2.3 Scientific novelty and significance of the project

The development of quantum computing in recent years has led to an active study of its potential in various fields of science and technology. The main obstacle to the implementation of quantum algorithms remains the complexity of their implementation and limited access to real quantum processors. That is why quantum simulators play an important role in the educational process, scientific research and testing quantum algorithms. In addition, quantum programming has become a priority area, as it can significantly speed up AI computing, work with Big Data and complex modeling.

4.3 Quantum Simulator Overview

Overall, the given quantum simulator processes text-based quantum commands by parsing gate operations (including single-qubit gates like H/X/Y/Z, two-qubit gates like CNOT, and multi-qubit operations like QFT), dynamically updates a state-vector representation of qubit states, and simulates quantum measurement through probabilistic collapse, implementing core quantum algorithms through sequential unitary transformations while tracking superposition and entanglement. Also, in some cases depending on the original instruction's set simulator plots probabilities line graph for all basis states.

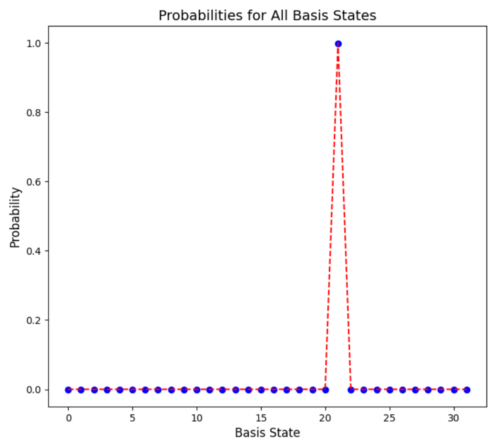
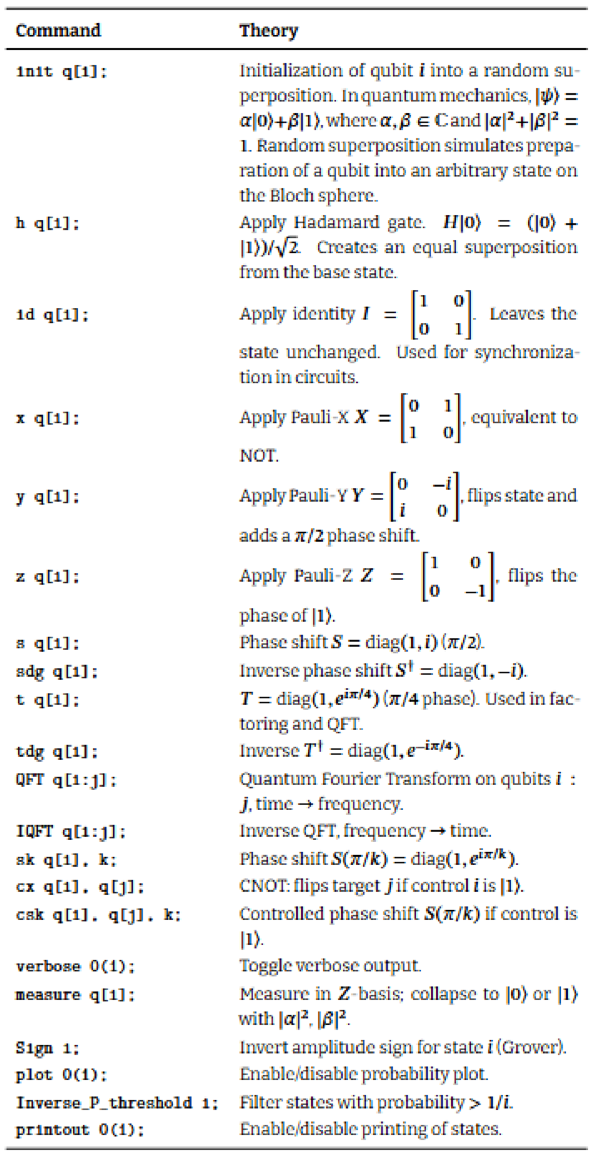


FIGURE 5. Grover Algorithms for 5 qubits

4.4 Commands Processing



Command	Theory
`init q[i];`	Initialization of qubit i into a random superposition. In quantum mechanics, $\|\psi\rangle = \alpha\|0\rangle+\beta\|1\rangle$, where $\alpha,\beta \in \mathbb{C}$ and $\|\alpha\|^2+\|\beta\|^2 = 1$. Random superposition simulates preparation of a qubit into an arbitrary state on the Bloch sphere.
`h q[i];`	Apply Hadamard gate. $H\|0\rangle = (\|0\rangle + \|1\rangle)/\sqrt{2}$. Creates an equal superposition from the base state.
`id q[i];`	Apply identity $I = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$. Leaves the state unchanged. Used for synchronization in circuits.
`x q[i];`	Apply Pauli-X $X = \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$, equivalent to NOT.
`y q[i];`	Apply Pauli-Y $Y = \begin{bmatrix}0 & -i\\i & 0\end{bmatrix}$, flips state and adds a $\pi/2$ phase shift.
`z q[i];`	Apply Pauli-Z $Z = \begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$, flips the phase of $\|1\rangle$.
`s q[i];`	Phase shift $S = \mathrm{diag}(1,i)$ $(\pi/2)$.
`sdg q[i];`	Inverse phase shift $S^\dagger = \mathrm{diag}(1,-i)$.
`t q[i];`	$T = \mathrm{diag}(1,e^{i\pi/4})$ ($\pi/4$ phase). Used in factoring and QFT.
`tdg q[i];`	Inverse $T^\dagger = \mathrm{diag}(1,e^{-i\pi/4})$.
`QFT q[i:j];`	Quantum Fourier Transform on qubits $i : j$, time → frequency.
`IQFT q[i:j];`	Inverse QFT, frequency → time.
`sk q[i], k;`	Phase shift $S(\pi/k) = \mathrm{diag}(1,e^{i\pi/k})$.
`cx q[i], q[j];`	CNOT: flips target j if control i is $\|1\rangle$.
`csk q[i], q[j], k;`	Controlled phase shift $S(\pi/k)$ if control is $\|1\rangle$.
`verbose 0(1);`	Toggle verbose output.
`measure q[i];`	Measure in Z-basis; collapse to $\|0\rangle$ or $\|1\rangle$ with $\|\alpha\|^2, \|\beta\|^2$.
`Sign i;`	Invert amplitude sign for state i (Grover).
`plot 0(1);`	Enable/disable probability plot.
`Inverse_P_threshold i;`	Filter states with probability $> 1/i$.
`printout 0(1);`	Enable/disable printing of states.

TABLE 1. Quantum gate commands and theory

FIGURE 6. Main Commands

Initialization Commands

init q[i] — command prepares qubits in a specific quantum state.

Single-qubit Gate Commands

h q[i] (Hadamard Gate) - creates superposition(|0⟩ → (|0⟩ + |1⟩)/√2) and modifies amplitudes for all states containing the target qubit **x/y/z q[i]** (Pauli gates) - X gate responds to bit-flip (|0⟩ ↔ |1⟩); Y gate responds to bit and phase flip ; Z gate responds to phase flip (|1⟩ → -|1⟩) **s/sdg q[i]** (Phase gates) -S gate responds to (|1⟩ → i|1⟩) ; S† responds to inverse phase shift **t/tdg q[i]** (π/4 phase gates) - T gate responds to (|1⟩ → $e^(i\pi/4)|1\rangle$)

Two-Qubit Gates

cx q[i],q[j] (CNOT) - creates entanglement |00⟩ → |00⟩, |10⟩ → |11⟩

Multi-Qubit Operations

QFT q[i:j] (Fourier Algorithm) - Quantum Fourier Transform on qubit range Key for: Shor's algorithm, phase estimation **IQFT q[i:j]** (Inverse Fourier Algorithm) - Inverse QFT for uncomputation **reverse** - Reverses qubit order in the state vector

Measurement Control

measure q[i] - collapses the qubit to |0⟩ or |1⟩ **Sign i** - flips the amplitude of state |i⟩ **verbose 1/plot 1** - toggles debugging output and visualizations

Special Algorithm Commands

Nm - prepares |m⟩ mod N⟩ states for Shor's algorithm **sk q[i],k** - custom phase gate (S(π/k)) for advanced algorithms

Overall command processing has such a workflow:

1. Parse command and qubit operands
2. Validate qubit indices
3. Update the state vector:
4. Apply unitary matrices (for gates)
5. Modify amplitudes (for measurements)
6. Track measurement probabilities
7. Output results (state vector/probabilities)

4.5 State Vector Determination

```
if initial_state == -1:
    state_vector[0] = 1
elif initial_state == -2:
    for k in range(2 ** num_qubits):
        if qubit_start <= k <= qubit_end:
            state_vector[k] = random.uniform(-1, 1) + 1j *
     random.uniform(-1, 1)
if initial_state != -1:
    state_vector /= np.sqrt(np.sum(np.abs(state_vector) ** 2)
    )

print('Initial state:')
print(state_vector)
```

We provide initial default state ⟩00⟩ what means that there is no superposition yet. Also let's remind that state vector in this case is [1,0,0,0] and for such meaning we have standard computational basis:

$$|00\rangle = \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \quad |01\rangle = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \quad |10\rangle = \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \quad |11\rangle = \begin{pmatrix}0\\0\\0\\1\end{pmatrix}$$

FIGURE 7. Computational Basis for two qubits

Thus, the vector [1,0,0,0] corresponds to the |0⟩ state. The question now is how to do it in a program? In the quantum simulator in order to determine how qubits initialized we make "control flags".

```
if initial_state == -1:
    state_vector[0] = 1
```

As can be seen '-1' is a control flag for the |0⟩ state.

In order to generate quantum superposition there is a control flag '-2'.

```
elif initial_state == -2:
    for k in range(2 ** num_qubits):
        if qubit_start <= k <= qubit_end:
```

```
        state_vector[k] = random.uniform(-1, 1) + 1j *
    random.uniform(-1, 1)
```

In range of total number of basis states, where for two qubits four states correspond, the k is the index of each state. After we loop through all possible states the program check the qubit range and modifies only the state where the binary representation of k includes the target qubits. On the next step the state vector takes the value of a random complex number for the amplitude (Real and Imaginary part with random values [-1; 1])

```python
if initial_state != -1:
    state_vector /= np.sqrt(np.sum(np.abs(state_vector) ** 2)
    )
```

This condition checks whether the quantum state should not be initialized to the default |00...0| state. Thus,the given condition states that all other amplitudes remains 0

4.6 Qubits Extraction

```python
def extract_qubits(command):
    before, sep, after = command.rpartition(";")
    gate = before.split()[0]
    if gate not in ['cx', 'sk', 'csk', 'N&m']:
        before1, sep1, after1 = before.rpartition(":")
        if sep1 == ':':
            digits = [int(s) for s in before1 if s.isdigit()
    ]
            qubit_start = digits[0] if len(digits) == 1 else
     10 * digits[0] + digits[1]
            digits = [int(s) for s in after1 if s.isdigit()]
            qubit_end = digits[0] if len(digits) == 1 else
    10 * digits[0] + digits[1]
        else:
            digits = [int(s) for s in before if s.isdigit()]
            qubit_start = sum(d * 10 ** (len(digits) - i - 1)
     for i, d in enumerate(digits))
            qubit_end = qubit_start
        control_start = qubit_start
        control_end = qubit_end
        target_start = -1
        target_end = -1
```

Explanation. This routine converts a textual quantum command into numeric qubit bounds for all "simple" gates (i.e., any *not* in `['cx','sk','csk','N&m']`, which are handled elsewhere). It first splits the command at the last semicolon to ignore trailing decorations and obtains `before`, then extracts the gate mnemonic as the first token of `before`. For simple gates, it checks whether a qubit *range* is present by splitting `before` at the last colon: if a colon exists, digits to its left and right are collected with `s.isdigit()` and interpreted as `qubit_start` and `qubit_end`, supporting one- or two-digit indices via either a single digit or `10*digits[0]+digits[1]` (so `"7"`→7, `"12"`→12). If no colon is found, the code treats the instruction as a single-index form, gathering all digits from `before` and reconstructing the integer positionally (base-10) into `qubit_start`; it then sets `qubit_end = qubit_start` so the gate targets exactly one qubit. Finally, the function normalizes outputs into a common four-field layout used across the simulator: `control_start`/`control_end` store the inclusive qubit span to apply the gate on, while `target_start`/`target_end` are filled with -1 as sentinels because simple gates in this branch do not require a secondary range or parameter.

Parsing the `sk` Gate

```python
elif gate == 'sk':
    before2, sep2, after2 = before.rpartition(",")
    before1, sep1, after1 = before2.rpartition(":")
    if sep1 == ':':
        digits = [int(s) for s in before1 if s.isdigit()]
        qubit_start = digits[0] if len(digits) == 1 else 10 *
      digits[0] + digits[1]
        digits = [int(s) for s in after1 if s.isdigit()]
        qubit_end = digits[0] if len(digits) == 1 else 10 *
     digits[0] + digits[1]
    else:
        digits = [int(s) for s in before2 if s.isdigit()]
        qubit_start = digits[0] if len(digits) == 1 else 10 *
      digits[0] + digits[1]
        qubit_end = qubit_start
    digits = [int(s) for s in after2 if s.isdigit()]
    k = sum(d * 10 ** (len(digits) - i - 1) for i, d in
     enumerate(digits))
    control_start = qubit_start
    control_end = qubit_end
    target_start = k
    target_end = -1
```

Explanation. This block parses instructions of the form `sk q[i], sk q[i:j], or sk q[i:j], k`. First, the string is split at the last comma: the left part (`before2`) holds the qubit specification, and the right part (`after2`) holds the integer parameter k. Then `before2` is checked for a colon: if present, digits to the left and right are extracted as the range `qubit_start:qubit_end`; otherwise, a single index is read and used for both. Digits are gathered with a comprehension `[int(s) for s in ... if s.isdigit()]`, supporting one- or two-digit indices (e.g. `"7"` → 7, `"12"` → 12). The parameter k is reconstructed from `after2` by multiplying digits by their positional weights in base ten. Finally, the results are placed in a uniform structure: `control_start` and `control_end` contain the qubit range, `target_start` stores k, and `target_end` is fixed at -1 as a placeholder. In effect, `sk q[2:4], 15` applies the `sk` operation to qubits $2, 3, 4$ with parameter 15, while `sk q[5], 13` applies it only to qubit 5. The code assumes canonical syntax, ignores non-digit characters, and reliably handles indices up to two digits.

4.7 Parsing the `cx` Gate

```python
elif gate == 'cx':
    before2, sep2, after2 = before.rpartition(",")
    before1, sep1, after1 = before2.rpartition(":")
    if sep1 == ':':
        digits = [int(s) for s in before1 if s.isdigit()]
        control_start = digits[0] if len(digits) == 1 else
     10 * digits[0] + digits[1]
        digits = [int(s) for s in after1 if s.isdigit()]
        control_end = digits[0] if len(digits) == 1 else 10 *
      digits[0] + digits[1]
    else:
        digits = [int(s) for s in before2 if s.isdigit()]
        control_start = digits[0] if len(digits) == 1 else
     10 * digits[0] + digits[1]
        control_end = control_start
    before1, sep1, after1 = after2.rpartition(":")
    if sep1 == ':':
        digits = [int(s) for s in before1 if s.isdigit()]
        target_start = digits[0] if len(digits) == 1 else 10
      * digits[0] + digits[1]
        digits = [int(s) for s in after1 if s.isdigit()]
        target_end = digits[0] if len(digits) == 1 else 10 *
      digits[0] + digits[1]
    else:
        digits = [int(s) for s in after2 if s.isdigit()]
        target_start = digits[0] if len(digits) == 1 else 10
      * digits[0] + digits[1]
        target_end = target_start
```

Explanation. This branch parses a controlled-`x` instruction of the canonical form `cx control_spec, target_spec;`, where each spec can be a single index `q[i]` or an inclusive range `q[i:j]`. It first splits at the last comma so that `before2` holds the control specification and `after2` holds the target specification. For the control side, `before2.rpartition(":")` detects whether a range is present; if yes, digits to the left and right of the colon are collected with `s.isdigit()` and converted into integers supporting one- or two-digit indices via either a single digit or `10*digits[0]+digits[1]`, yielding `control_start` and `control_end`; if no colon exists, all digits in `before2` are read as a single index and both `control_start` and `control_end` are set to that value. The same procedure is then applied to the target side by operating on `after2`: a colon yields `target_start` and `target_end` as a range, otherwise a single index is used for both. In effect, `cx q[1:3], q[5:7];` produces `control_start=1, control_end=3` and `target_start=5, target_end=7`, while `cx q[2], q[9];` yields `control_start=control_end=2` and `target_start=target_end=9`. Non-digit characters are ignored, only the last comma and last colon are considered (so the syntax must be canonical), and indices beyond two digits are not fully supported by this simple digit-combining heuristic.

4.8 Parsing the `csk` Gate

```
elif gate == 'csk':
    before1, sep1, after1 = before.rpartition(":")
    if sep1 == ':':
        sys.exit('The csk gate does not allow expansion of
      range of qubits')
    before2, sep2, after2 = before.rpartition(",")
    before3, sep3, after3 = before2.rpartition(",")
    digits = [int(s) for s in before3 if s.isdigit()]
    control_qubit = digits[0] if len(digits) == 1 else 10 *
      digits[0] + digits[1]
    digits = [int(s) for s in after3 if s.isdigit()]
    target_qubit = digits[0] if len(digits) == 1 else 10 *
      digits[0] + digits[1]
    digits = [int(s) for s in after2 if s.isdigit()]
    k = sum(d * 10 ** (len(digits) - i - 1) for i, d in
      enumerate(digits))
    control_start = control_qubit
    control_end = target_qubit
    target_start = k
    target_end = -1
```

Explanation. This branch parses the controlled-`sk` instruction in the canonical form `csk q[c], q[t], k;` and explicitly forbids ranges. It first checks for a colon in `before`; if found, the code terminates with an error, enforcing that both control and target must be single indices. Next, it splits by the last comma to isolate the scalar parameter region (`after2` contains `k`) and by the preceding comma to separate the two qubit specs: `before3` holds the control spec and `after3` the target spec. For each qubit spec, digits are collected with `s.isdigit()` and interpreted as either a one-digit index or a two-digit index via `10*digits[0]+digits[1]` (so `"7"`→7, `"12"`→12). The parameter `k` is reconstructed from `after2` by positional weighting in base ten (e.g., `"123"`→123). Finally, values are normalized into the simulator's common four-field layout: `control_start` receives the control qubit, `control_end` receives the target qubit (reusing the "control/end" pair as a two-qubit holder), `target_start` stores the integer `k`, and `target_end` is set to `-1` as a sentinel since no second range exists here. In effect, `csk q[2], q[9], 15;` yields control 2, target 9, and parameter $k = 15$. The parser ignores non-digit characters, assumes the last-two-commas canonical order, and supports qubit indices up to two digits with this simple digit-combining heuristic.

4.9 Parsing the `N&m` Instruction

```
elif gate == 'N&m':
    before1, sep1, after1 = before.rpartition(":")
    before2, sep2, after2 = before.rpartition(",")
    digits = [int(s) for s in before2 if s.isdigit()]
    N = sum(d * 10 ** (len(digits) - i - 1) for i, d in
      enumerate(digits))
    digits = [int(s) for s in after2 if s.isdigit()]
    m = sum(d * 10 ** (len(digits) - i - 1) for i, d in
      enumerate(digits))
    control_start = N
    control_end = m
    target_start = -1
    target_end = -1
return control_start, control_end, target_start, target_end
```

Explanation. This branch parses an instruction of the canonical form `N&m N_value, m_value;` and converts the two scalar parameters into integers. It first calls `before.rpartition(":")` (the result is unused here, kept for interface symmetry with other gates) and then splits `before` at the last comma via `rpartition(",")` so that `before2` contains everything to the left of the comma (the N field) and `after2` contains everything to the right (the m field). For each field, digits are collected with `s.isdigit()` and reconstructed into an integer using positional base-10 weighting: $\sum d_i\, 10^{n-i-1}$. Unlike the simple two-digit heuristic used in other branches, this reconstruction handles arbitrarily many digits (e.g., `"16"`→16, `"1024"`→1024). The parsed values are then mapped into the common four-slot return layout used across the simulator: `control_start` receives N, `control_end` receives m, while `target_start` and `target_end` are set to -1 as sentinels since this opcode has no target range. Non-digit characters (such as `N&m`, spaces, or brackets) are ignored during extraction, the last comma determines the split, and the syntax is assumed to be canonical; for example, `N&m 16, 4;` yields `control_start` $= 16$, `control_end` $= 4$, `target_start` $=$ `target_end` $= -1$.

4.10 Reading Commands from a File

```
if len(sys.argv) > 1:
    input_file = sys.argv[1]
with open(input_file, "r") as file:
    command_list = [line.strip() for line in file]
```

Explanation. This snippet expects the first command-line argument (`sys.argv[1]`; note `sys.argv[0]` is the script name) to be a path to the input text file; if at least one argument is provided it assigns that path to `input_file`, then opens it in text mode and builds `command_list` by stripping leading/trailing whitespace (including newlines) from each line. The `with` statement ensures the file handle is closed automatically even on exceptions. Because the `open` happens unconditionally, omitting the CLI argument will leave `input_file` undefined and raise a `NameError`; robust code typically adds an `else` branch (usage/help or default path) and may specify an explicit encoding (e.g., `encoding='utf-8'`). The comprehension preserves line order; blank lines become empty strings `''` (filter with `if line.strip()` to skip them).

4.11 Main Command Loop

```python
for i in range(len(command_list)):
    command = command_list[i]
    before, sep, after = command.rpartition(";")
    if before.split() != []:
        gate = before.split()[0]
    else:
        gate = ''

    if gate in ['id', 'h', 'x', 'y', 'z', 's', 'sdg', 't', 'tdg', 'measure', 'QFT', 'IQFT']:
        qubit_start, qubit_end, _, _ = extract_qubits(command)
        num_qubits = max(num_qubits, qubit_start + 1)
        num_qubits = max(num_qubits, qubit_end + 1)
    elif gate in ['init', 'verbose', 'plot', 'printout', 'Inverse_P_threshold', 'N&m']:
        qubit_start, qubit_end, _, _ = extract_qubits(command)
        if gate == 'init':
            initial_state = -2
            qubit_start = qubit_start
            qubit_end = qubit_end
        elif gate == 'verbose':
            verbose_mode = qubit_start
        elif gate == 'plot':
            plot_enabled = qubit_start
            print('plot =', plot_enabled)
        elif gate == 'printout':
            print_enabled = qubit_start
            print('printout =', print_enabled)
        elif gate == 'Inverse_P_threshold':
            if qubit_start > 0: probability_threshold = float(1.0 / qubit_start)
            print('Inverse_P_threshold =', qubit_start)
            print('P_threshold =', probability_threshold)
        elif gate == 'N&m':
            initial_state = -3
            N = float(qubit_start)
            m = float(qubit_end)
    elif gate == 'sk':
        qubit_start, qubit_end, k, _ = extract_qubits(command)
        num_qubits = max(num_qubits, qubit_end + 1)
        num_qubits = max(num_qubits, qubit_start + 1)
    elif gate == 'cx':
        control_start, control_end, target_start, target_end = extract_qubits(command)
        num_qubits = max(num_qubits, control_start + 1)
        num_qubits = max(num_qubits, control_end + 1)
        num_qubits = max(num_qubits, target_start + 1)
        num_qubits = max(num_qubits, target_end + 1)
    elif gate == 'csk':
        control_qubit, target_qubit, k, _ = extract_qubits(command)
        num_qubits = max(num_qubits, control_qubit + 1)
        num_qubits = max(num_qubits, target_qubit + 1)
```

Explanation. The loop processes each parsed line `command`, isolates the portion before the final semicolon with `rpartition(";")`, and extracts the gate mnemonic as the first token of `before` (empty or whitespace-only lines yield `gate=''` and fall through). For single-qubit and range-capable "simple" gates (`id,h,x,y,z,s,sdg,t,tdg,measure,QFT,IQFT`), it delegates to `extract_qubits` to obtain `qubit_start` and `qubit_end` and expands the simulated register size by maintaining `num_qubits = max(num_qubits, index+1)`, using `+1` because qubits are zero-indexed. Configuration/meta opcodes are grouped under `['init','verbose','plot','printout','Inverse_P','N&m']` : each still calls `extract_qubits` (reusing its integer parsing), then applies side effects—`init` sets a sentinel `initial_state=-2` (range retained but not otherwise used here); `verbose`, `plot`, and `printout` set mode flags (and print the current setting for the latter two); `Inverse_P_threshold` optionally computes `probability_threshold = 1.0 / qubit_start` when positive and prints both the inverse and the derived threshold; `N&m` switches to `initial_state=-3` and stores two scalars `N` and `m` as floats from the parsed pair. The parameterized single-qubit/range gate `sk` obtains `k` as its third return and updates `num_qubits` from the start/end bounds. The two-operand `cx` gate receives control and target ranges and updates `num_qubits` against all four endpoints, while the controlled-`sk` (`csk`) variant reads single control/target indices plus `k` and grows `num_qubits` accordingly. Overall, this loop is the dispatcher: it canonicalizes gate names, leverages a unified `extract_qubits` contract to turn text into indices and parameters, updates the required register size lazily as the maximum seen index plus one, and applies per-opcode state changes that drive later simulation stages.

4.12 Bit Manipulation Helpers

```python
def set_bit(value, bit_index):
    return value | (1 << bit_index)

def clear_bit(value, bit_index):
    return value & ~(1 << bit_index)
```

Explanation. Both helpers manipulate the bit at zero-based position `bit_index` using standard masks. In `set_bit`, `1 << bit_index` builds a mask whose only 1 is at that position; OR-ing it with `value` (`value` mask|) guarantees that bit becomes 1 while all other bits remain unchanged (idempotent if it was already 1). In `clear_bit`, the mask is negated first: `~(1 << bit_index)` has 0 at the target position and 1s elsewhere, so AND-ing with `value` (`value & ~mask`) forces that bit to 0, preserving all others (idempotent if it was already 0). Runtime is $O(1)$. Valid `bit_index` must be ≥ 0; Python raises `ValueError` for a negative shift. Arbitrarily large `bit_index` values are fine (Python integers are unbounded). These operations are typically used on non-negative integers; applying `clear_bit` to negative numbers can yield unintuitive results due to two's-complement style semantics of ~ on Python ints (`~n == -n-1`). A quick check like `assert bit_index >= 0` (and optionally `value >= 0`) makes intent explicit.

4.13 State Printing (`print_state`)

```
def print_state(gate, num_qubits, verbose_mode, state):
    if gate not in ['cx', 'sk', 'csk', 'Sign', 'QFT', 'IQFT', 'h']:
        print(f'Gate {gate} on qubit {qubit}'),
    if verbose_mode == 1:
        print('  resulted in state |psi> = '),
        k1 = 0
        psi = ''
        for k in range(2 ** num_qubits):
            binary_str = ("{:0%db}" % num_qubits).format(k)[::-1]
            if state[k] != 0:
                k1 += 1
                if k1 == 1:
                    psi += f'({state[k]:.3f})|{binary_str}>'
                else:
                    psi += f'+ ({state[k]:.3f})|{binary_str}>'
        psi = psi.replace('+ -', '- ')
        print(psi)
        print()
    return state, state
```

Explanation. This routine conditionally prints a human-readable snapshot of the simulator state. For gates *not* in `['cx','sk','csk','Sign','QFT','IQFT','h']` it emits a short header `Gate {gate} on qubit {qubit}` (note: `qubit` must exist in the surrounding scope or be passed as a parameter; otherwise a `NameError` occurs). When `verbose_mode == 1` it builds Dirac notation for the nonzero amplitudes of $|\psi\rangle$. It iterates over all computational basis indices `k = 0..2**num_qubits-1`, formats each as a zero-padded binary string of width `num_qubits`, then reverses it with `[::-1]` so that the least-significant bit (qubit 0) appears on the left—i.e., little-endian internal indexing is printed in a left-to-right qubit order. Only entries with `state[k] != 0` are included. A counter `k1` is used to prepend `'+ '` from the second term onward so the final string has the form `(amp)`$|b_{q0} \dots b_{qn-1}\rangle$ with pluses between terms; afterwards `psi.replace('+ -','- ')` cleans up signs for negative amplitudes. Amplitudes are formatted with `{state[k]:.3f}`, which is appropriate for real-valued states; for complex amplitudes a custom formatter (e.g., `f'({state[k].real:.3f}{state[k].imag:+.3f}i)'`) would be safer. The comma after each `print(...)` is a leftover Python 2 idiom that in Python 3 simply evaluates a tuple and has no effect; removing the trailing commas is recommended. Finally, the function returns `(state, state)`—two references to the same object—matching a caller convention of "(old, new) state" without copying; for large registers this avoids unnecessary memory use. Exponentially many basis states are scanned, so enabling verbose mode is practical only for small `num_qubits`.

4.14 DFT Basis Vector (`dft_j`)

```
def dft_j(type, N, j):
    dft_coeff = np.zeros(N, dtype=np.complex_)
    for k in range(N):
        dft_coeff[k] = np.exp(type * 2 * np.pi * 1j * j * k / N)
    return dft_coeff / np.sqrt(N)
```

Explanation. This routine builds the length-N normalized complex sinusoid corresponding to the j-th column (or row) of a DFT/IDFT matrix under the sign convention controlled by `type`. It allocates a complex vector `dft_coeff` and for each frequency sample $k = 0, \dots, N-1$ writes

$$(\texttt{dft\_coeff})[k] \;=\; \exp\!\Big(\sigma\, i\, 2\pi\, \frac{jk}{N}\Big), \qquad \sigma = \texttt{type} \in \{+1, -1\},$$

so $\sigma = -1$ yields the usual forward DFT kernel $e^{-i2\pi jk/N}$ and $\sigma = +1$ the inverse kernel. The final division by $\sqrt{N}$ enforces unitary normalization, making the set of such vectors orthonormal (columns/rows of a unitary Fourier matrix), which is essential in quantum-style or numerically stable Fourier operations. The dtype `np.complex_` (typically `complex128`) preserves precision; `1j` supplies the imaginary unit. Runtime is $O(N)$. Inputs should satisfy $N \in \mathbb{N}_{>0}$ and $j \in \{0, \dots, N-1\}$ (other integers map periodically modulo N).

4.15 Partial DFT on a Qubit Block (`dft`)

```
def dft(num_qubits, qubit_start, qubit_end, type, state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.complex128)
    N1 = 2 ** qubit_start                    # qubits below the DFT block
    N2 = 2 ** (qubit_end - qubit_start + 1)  # size of the DFT block
    N3 = 2 ** (num_qubits - qubit_end - 1)   # qubits above the DFT block
    for j3 in range(N3):
        for j2 in range(N2):
            for j1 in range(N1):
                j = (j3 << qubit_end + 1) + (j2 << qubit_start) + j1
                if np.absolute(state[j]) > 0:
                    dft_coeff = dft_j(type, N2, j2)
                    for jj in range(len(dft_coeff)):
                        j4 = (j3 << qubit_end + 1) + (jj << qubit_start) + j1
                        result_state[j4] += dft_coeff[jj] * state[j]
    return result_state
```

Explanation. This routine applies a unitary DFT/IDFT (sign controlled by `type` as in `dft_j`) to a *contiguous* register slice [`qubit_start`, `qubit_end`] while leaving all other qubits untouched. The computational basis index is interpreted in little-endian order (qubit 0 is the least significant bit). The state vector is logically partitioned into three bit fields: j_1 for the $q <$ `qubit_start` "below" block ($N_1 = 2^{\texttt{qubit\_start}}$), j_2 for the DFT block itself ($N_2 = 2^{\texttt{width}}$ with `width` $=$ `qubit_end` $-$ `qubit_start` $+ 1$), and j_3 for the $q >$ `qubit_end` "above" block ($N_3 = 2^{\texttt{num\_qubits}-\texttt{qubit\_end}-1}$). The nested loops enumerate all triples (j_3, j_2, j_1), reconstructing the flat index $j = (j_3 \ll (\texttt{qubit\_end} + 1)) + (j_2 \ll \texttt{qubit\_start}) + j_1$; in Python, `<<` has lower precedence than `+`, so `j3 << qubit_end + 1` is interpreted as $j_3 \ll (\texttt{qubit\_end} + 1)$. For each occupied basis component $state[j] \neq 0$, the code fetches the length-N_2 Fourier column $\mathrm{DFT}_{:,j_2}$ via `dft_j(type, N2, j2)` and scatters its contributions across all "frequency" indices jj *within the* *block* while keeping j_1 and j_3 fixed; the destination index is $j_4 = (j_3 \ll (\texttt{qubit\_end} + 1)) + (jj \ll \texttt{qubit\_start}) + j_1$, so only the middle bit field changes. Because `dft_j` uses $1/\sqrt{N_2}$ normalization, the transform on the selected slice is unitary; qubits outside the slice act as independent "batch" dimensions. Complexity is $O(2^{\texttt{num\_qubits}} \cdot N_2)$ since the outer three loops cover all basis states ($N_1 N_2 N_3 = 2^{\texttt{num\_qubits}}$) and the inner accumulation iterates over N_2.

4.16 Identity Gate (identity_gate)

```python
def identity_gate(num_qubits, qubit, state):
    return state
```

Explanation. This is a no-op gate: it returns the very same `state` object it received, leaving amplitudes unchanged and ignoring `num_qubits` and `qubit`. It is idempotent and $O(1)$, performs no validation (e.g., that `len(state)==2**num_qubits`), and does not copy—so downstream code sees the original array (use `state.copy()` if a defensive copy is desired). In the simulator's pipeline it provides a consistent hook for the `id` opcode, useful for testing control flow, timing, or verbose printing without altering the quantum state; shape and dtype are preserved exactly.

4.17 Hadamard Gate (hadamard_gate)

```python
def hadamard_gate(num_qubits, qubit, state):
    print(f'Hadamard gate applied to qubit {qubit}')
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    isq2 = 1 / np.sqrt(2)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            bit_parity = (j >> qubit) % 2
            if bit_parity == 0:
                result_state[j] += isq2 * state[j]
                result_state[set_bit(j, qubit)] += isq2 *
     state[j]
            elif bit_parity == 1:
                result_state[clear_bit(j, qubit)] += isq2 *
     state[j]
                result_state[j] += -isq2 * state[j]
    return result_state
```

Explanation. Applies the unitary Hadamard $H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ to a single qubit in an n-qubit state (length $2^{\texttt{num\_qubits}}$). For each basis index j, it tests the selected qubit by $\texttt{bit\_parity} = (j \gg \texttt{qubit}) \bmod 2$ (with little-endian convention: qubit 0 is LSB). If that bit is 0, amplitude ψ_j contributes $+\psi_j/\sqrt{2}$ to the same basis state j (interpreting $|0\rangle \mapsto (|0\rangle + |1\rangle)/\sqrt{2}$) and $+\psi_j/\sqrt{2}$ to the flipped index $\texttt{set\_bit}(j, \texttt{qubit}) = j \oplus (1 \ll \texttt{qubit})$. If the bit is 1, ψ_j contributes $+\psi_j/\sqrt{2}$ to the cleared/flipped index $\texttt{clear\_bit}(j, \texttt{qubit})$ and $-\psi_j/\sqrt{2}$ to j (i.e., $|1\rangle \mapsto (|0\rangle - |1\rangle)/\sqrt{2}$). Thus, within each two-state pair $\{|0_q\rangle, |1_q\rangle\}$, the new amplitudes are $a_0' = (a_0 + a_1)/\sqrt{2}$ and $a_1' = (a_0 - a_1)/\sqrt{2}$ while all other qubits (the higher and lower bit fields) remain fixed. The loop skips zero amplitudes for speed; a small tolerance could be used if needed. It allocates and returns a fresh `result_state` (complex128), leaving the input array unchanged; runtime is $O(2^{\texttt{num\_qubits}})$.

4.18 Multi-qubit Hadamard (hadamard_n)

```python
def hadamard_n(num_qubits, state):
    print(f'Hadamard gate applied to qubits from 0 to {
     num_qubits - 1}')
    isq2 = 1 / np.sqrt(2)
    for qubit in range(num_qubits):
        result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
        for j in range(2 ** num_qubits):
            if state[j] != 0:
                bit_parity = (j >> qubit) % 2
                if bit_parity == 0:
                    result_state[j] += isq2 * state[j]
                    result_state[set_bit(j, qubit)] += isq2 *
      state[j]
                elif bit_parity == 1:
                    result_state[clear_bit(j, qubit)] +=
     isq2 * state[j]
                    result_state[j] += -isq2 * state[j]
        state = result_state
    return result_state
```

Explanation. This routine applies the Hadamard gate sequentially to every qubit $q = 0, \ldots, \texttt{num\_qubits} - 1$, realizing the tensor product $H^{\otimes n}$ (Walsh–Hadamard transform) on the whole register. For each target `qubit` it allocates a fresh `result_state` (length $2^{\texttt{num\_qubits}}$, complex128) and scans all basis indices `j`; using little-endian indexing, the selected bit is `(j >> qubit) % 2`. If that bit is 0, amplitude ψ_j contributes $+\psi_j/\sqrt{2}$ to the same index j and $+\psi_j/\sqrt{2}$ to the bit-flipped index `set_bit(j, qubit)`; if it is 1, ψ_j contributes $+\psi_j/\sqrt{2}$ to `clear_bit(j, qubit)` and $-\psi_j/\sqrt{2}$ to j. This is exactly the pairwise butterfly $(a_0, a_1) \mapsto ((a_0 + a_1)/\sqrt{2}, (a_0 - a_1)/\sqrt{2})$ within each two-state subspace spanned by qubit q, while all other qubits remain fixed. After processing a qubit, `state` is replaced by `result_state` so each subsequent Hadamard acts on the updated amplitudes; after the final iteration the function returns the last `result_state`, which equals the fully transformed state. The printed message is informational; removing it has no effect on correctness. Complexity is $O(n\,2^n)$ (the inner test `state[j] != 0` skips exact zeros for speed), memory overhead is one extra vector per qubit step, and numerically this implementation preserves unitarity via the `isq2 = 1/√2` factor.

4.19 Pauli-X (pauli_x)

Listing 1. Single-qubit Pauli-X applied on a target qubit across the full register.

```python
def pauli_x(num_qubits, qubit, state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            bit_parity = (j >> qubit) % 2
            if bit_parity == 0:
                result_state[set_bit(j, qubit)] += state[j]
            if bit_parity == 1:
                result_state[clear_bit(j, qubit)] += state[j
      ]
    return result_state
```

Explanation. This part applies the Pauli-X gate to all qubits in a chosen interval [`qubit_start`, `qubit_end`]. If the interval is written in ascending order, the loop runs forward with `range(qubit_start, qubit_end + 1)`. If the interval is written in reverse, the loop respects that direction with `range(qubit_start, qubit_end - 1, -1)`. For each qubit in the interval, the function `pauli_x` is called, which flips the amplitudes between the $|0\rangle$ and $|1\rangle$ states of that qubit. The new state vector is then passed to `print_state`, and its output replaces the current state so that successive gates are applied in sequence. Although X gates on different qubits commute (the final result does not depend on order), the code deliberately follows the order provided by the user, keeping behavior consistent with other range handlers. Both forward and backward loops include the end index, so no qubit is skipped. The cost is proportional to the number of qubits in the interval, that is $O((|\texttt{qubit\_end}-\texttt{qubit\_start}|+1)\,2^n)$, and each step requires one extra state vector of length 2^n.

4.20 Pauli-Y (`pauli_y`)

Listing 2. Single-qubit Pauli-Y applied on a target qubit across the full register.

```python
def pauli_y(num_qubits, qubit, state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            bit_parity = (j >> qubit) % 2
            if bit_parity == 0:
                result_state[set_bit(j, qubit)] += 1j *
     state[j]
            if bit_parity == 1:
                result_state[clear_bit(j, qubit)] += -1j *
     state[j]
    return result_state
```

Explanation. This function applies the Pauli-Y gate to the specified `qubit` of an n-qubit state vector (length 2^n). Little-endian indexing is used: qubit 0 is the least significant bit of the basis index j. For each j, the target bit is read as $(j \gg \texttt{qubit}) \bmod 2$. If that bit is 0, the amplitude is moved to the bit-flipped index and multiplied by $+i$; if the bit is 1, it is moved to the bit-flipped index and multiplied by $-i$. Thus, within each two-level subspace of the target qubit, the mapping is $(\psi_0, \psi_1) \to (-i\,\psi_1,\; i\,\psi_0)$, which is exactly the Pauli-Y action. A new array `result_state` is allocated to avoid in-place overwrites; the operation preserves the ℓ_2 norm because it is a permutation with unit-modulus phases. Time complexity is $O(2^n)$ and the temporary memory is one additional vector of size 2^n.

4.21 Pauli-Z (`pauli_z`)

Listing 3. Single-qubit Pauli-Z applied on a target qubit across the full register.

```python
def pauli_z(num_qubits, qubit, state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            bit_parity = (j >> qubit) % 2
            if bit_parity == 0:
                result_state[j] += state[j]
            if bit_parity == 1:
                result_state[j] += -state[j]
    return result_state
```

Explanation. This function applies the Pauli-Z gate (phase flip) to the chosen `qubit` of an n-qubit state vector. The basis index j is scanned, and the target bit is obtained by $(j \gg \texttt{qubit}) \bmod 2$. If that bit is 0, the amplitude ψ_j is copied unchanged; if the bit is 1, the amplitude is negated. The effect on the two-level subspace of the target qubit is $(\psi_0, \psi_1) \to (\psi_0, -\psi_1)$, which corresponds to the unitary matrix $Z = \mathrm{diag}(1, -1)$. Other qubits remain unaffected. A new `result_state` is created to avoid modifying the input in place. The operation preserves the vector norm since only phases (± 1) are applied. Runtime is $O(2^n)$ and memory overhead is one additional vector of length 2^n.

4.22 Phase Gate (`phase_gate`)

Listing 4. Single-qubit Phase (S) gate applied on a target qubit across the full register.

```python
def phase_gate(num_qubits, qubit, state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            bit_parity = (j >> qubit) % 2
            if bit_parity == 0:
                result_state[j] += state[j]
            if bit_parity == 1:
                result_state[j] += 1j * state[j]
    return result_state
```

Explanation. This function applies the Phase gate (S) to the specified `qubit` of an n-qubit state. For each basis index j, the bit value of the target qubit is read as $(j \gg \texttt{qubit}) \bmod 2$. If the bit is 0, the amplitude ψ_j is left unchanged; if the bit is 1, it is multiplied by i. The resulting action on the two-level subspace of the target qubit is

$$(\psi_0, \psi_1) \longrightarrow (\psi_0,\; i\psi_1),$$

which matches the unitary $S = \mathrm{diag}(1, i)$. All other qubits remain fixed. A fresh `result_state` buffer of length 2^n is created, ensuring the input vector is not modified in place. Since the multiplier is a unit-modulus phase, the norm of the quantum state is preserved. Runtime is $O(2^n)$ with memory overhead of one additional vector.

4.23 Inverse Phase Gate (`phase_dagger_gate`)

Listing 5. Single-qubit inverse Phase ($S^\dagger$) gate applied on a target qubit.

```python
def phase_dagger_gate(num_qubits, qubit, state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            bit_parity = (j >> qubit) % 2
            if bit_parity == 0:
                result_state[j] += state[j]
            if bit_parity == 1:
                result_state[j] += -1j * state[j]
    return result_state
```

Explanation. This function applies the inverse of the Phase gate ($S^\dagger$) to the chosen `qubit` in an n-qubit state vector. For each basis index j, the target bit is determined as $(j \gg \texttt{qubit}) \bmod 2$. If the bit equals 0, the amplitude ψ_j is copied unchanged; if the bit equals 1, the amplitude is multiplied by $-i$. Within the two-dimensional subspace of the target qubit, the transformation is

$$(\psi_0, \psi_1) \longrightarrow (\psi_0, -i\,\psi_1),$$

which corresponds to the unitary $S^\dagger = \mathrm{diag}(1, -i)$. All other qubits remain unchanged. A new state vector `result_state` is allocated, preventing in-place modification of the input. Since multiplication by $-i$ has unit modulus, the norm of the quantum state is preserved. The procedure runs in $O(2^n)$ time and uses $O(2^n)$ additional memory.

4.24 Inverse Phase Gate (`phase_dagger_gate`)

Listing 6. Single-qubit inverse Phase ($S^\dagger$) gate applied on a target qubit.

```python
def phase_dagger_gate(num_qubits, qubit, state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            bit_parity = (j >> qubit) % 2
            if bit_parity == 0:
                result_state[j] += state[j]
            if bit_parity == 1:
                result_state[j] += -1j * state[j]
    return result_state
```

Explanation. This function applies the inverse of the Phase gate ($S^\dagger$) to the chosen `qubit` in an n-qubit state vector. For each basis index j, the target bit is determined as $(j \gg \texttt{qubit}) \bmod 2$. If the bit equals 0, the amplitude ψ_j is copied unchanged; if the bit equals 1, the amplitude is multiplied by $-i$. Within the two-dimensional subspace of the target qubit, the transformation is

$$(\psi_0, \psi_1) \longrightarrow (\psi_0, -i\,\psi_1),$$

which corresponds to the unitary $S^\dagger = \mathrm{diag}(1, -i)$. All other qubits remain unchanged A new state vector `result_state` is allocated, preventing in-place modification of the input. Since multiplication by $-i$ has unit modulus, the norm of the quantum state is preserved. The procedure runs in $O(2^n)$ time and uses $O(2^n)$ additional memory.

4.25 T Gate (`t_gate`)

Listing 7. Single-qubit T gate applied on a target qubit across the full register.

```python
def t_gate(num_qubits, qubit, state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            bit_parity = (j >> qubit) % 2
            if bit_parity == 0:
                result_state[j] += state[j]
            if bit_parity == 1:
                result_state[j] += (1 + 1j) / np.sqrt(2) *
     state[j]
    return result_state
```

Explanation. This function applies the T gate to the selected `qubit` of an n-qubit state. The loop iterates over all basis indices j, and the target bit is extracted as $(j \gg \texttt{qubit}) \bmod 2$. If that bit is 0, the amplitude ψ_j is left unchanged. If the bit is 1, the amplitude is multiplied by

$$\frac{1+i}{\sqrt{2}} = e^{i\pi/4}.$$

Thus, within the two-dimensional subspace of the target qubit, the mapping is

$$(\psi_0, \psi_1) \longrightarrow (\psi_0, e^{i\pi/4}\psi_1),$$

which corresponds to the unitary $T = \mathrm{diag}(1, e^{i\pi/4})$.

A fresh array `result_state` of size 2^n is allocated, ensuring the input vector remains unchanged. The operation preserves the norm because the factor $e^{i\pi/4}$ has unit modulus. Runtime is $O(2^n)$ and memory overhead is $O(2^n)$.

4.26 Inverse T Gate (`t_dagger_gate`)

Listing 8. Single-qubit inverse T gate ($T^\dagger$) applied on a target qubit.

```python
def t_dagger_gate(num_qubits, qubit, state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            bit_parity = (j >> qubit) % 2
            if bit_parity == 0:
                result_state[j] += state[j]
            if bit_parity == 1:
                result_state[j] += (1 - 1j) / np.sqrt(2) *
     state[j]
    return result_state
```

Explanation. This function applies the inverse of the T gate ($T^\dagger$) to the specified `qubit` of an n-qubit state vector. For each basis index j, the bit value of the target qubit is obtained as $(j \gg \texttt{qubit}) \bmod 2$. If the bit equals 0, the amplitude ψ_j is unchanged. If the bit equals 1, the amplitude is multiplied by

$$\frac{1-i}{\sqrt{2}} = e^{-i\pi/4}.$$

Therefore, on the two-level subspace of the target qubit, the mapping is

$$(\psi_0, \psi_1) \longrightarrow (\psi_0, e^{-i\pi/4}\psi_1),$$

corresponding to the unitary $T^\dagger = \mathrm{diag}(1, e^{-i\pi/4})$. The procedure allocates a fresh state vector `result_state` (size 2^n), which prevents overwriting the input. Since the multiplier $e^{-i\pi/4}$ has unit modulus, the norm of the quantum state is preserved. Runtime is $O(2^n)$ with $O(2^n)$ extra memory.

4.27 Parameterized Phase Gate (`sk_gate`)

Listing 9. Single-qubit parameterized phase gate with factor $e^{i\pi/k}$.

```python
def sk_gate(num_qubits, qubit, k, state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    phase_factor = np.exp(np.pi * 1j / k)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            bit_parity = (j >> qubit) % 2
            if bit_parity == 0:
                result_state[j] += state[j]
            if bit_parity == 1:
                result_state[j] += phase_factor * state[j]
    print(f'Gate sk on qubit {qubit} with k = {k}'),
    return result_state
```

Explanation. This function applies a parameterized single-qubit phase gate to the chosen `qubit`. The parameter k determines the phase multiplier

$$\texttt{phase\_factor} = e^{i\pi/k}.$$

For each basis index j, the bit of the target qubit is read as $(j \gg \texttt{qubit}) \bmod 2$. If the bit is 0, the amplitude ψ_j is copied unchanged. If the bit is 1, the amplitude is multiplied by $e^{i\pi/k}$. The transformation on the two-level subspace of the target qubit is

$$(\psi_0, \psi_1) \longrightarrow (\psi_0, e^{i\pi/k}\psi_1),$$

which corresponds to the unitary $\mathrm{diag}(1, e^{i\pi/k})$.

4.28 Controlled-X Gate (`controlled_x`)

Listing 10. Two-qubit Controlled-X (CNOT) gate with one control and one target qubit.

```
def controlled_x(num_qubits, control_qubit, target_qubit,
    state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            control_parity = (j >> control_qubit) % 2
            target_parity = (j >> target_qubit) % 2
            if control_parity == 0:
                result_state[j] += state[j]
            else:
                if target_parity == 0:
                    result_state[set_bit(j, target_qubit)]
    += state[j]
                else:
                    result_state[clear_bit(j, target_qubit)]
     += state[j]
    print(f'Gate cx on control qubit {control_qubit} and
     target qubit {target_qubit}'),
    return result_state
```

Explanation. This function applies the controlled-X (CNOT) gate to an n-qubit state vector. The `control_qubit` determines whether the X operation is triggered on the `target_qubit`. For each basis index j:

- If the control bit $(j \gg \texttt{control\_qubit}) \bmod 2$ equals 0, the amplitude ψ_j is copied unchanged.
- If the control bit equals 1, then the target bit is flipped:
  - If target bit = 0, amplitude is sent to $\texttt{set\_bit}(j, \texttt{target\_qubit})$.
  - If target bit = 1, amplitude is sent to $\texttt{clear\_bit}(j, \texttt{target\_qubit})$.

This is equivalent to $j \mapsto j \oplus 2^{\text{target\_qubit}}$ when the control bit is 1, and $j \mapsto j$ otherwise. The action on the two-qubit subspace $(\texttt{control}, \texttt{target})$ is

$$|00\rangle \mapsto |00\rangle, \quad |01\rangle \mapsto |01\rangle, \quad |10\rangle \mapsto |11\rangle, \quad |11\rangle \mapsto |10\rangle,$$

which matches the unitary matrix of CNOT.

4.29 Controlled-Parameterized Phase Gate (`controlled_sk`)

Listing 11. Two-qubit controlled phase gate with parameter k.

```
def controlled_sk(num_qubits, control_qubit, target_qubit, k,
     state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
    phase_factor = np.exp(np.pi * 1j / k)
    for j in range(2 ** num_qubits):
        if state[j] != 0:
            control_parity = (j >> control_qubit) % 2
            target_parity = (j >> target_qubit) % 2
            if control_parity == 0:
                result_state[j] += state[j]
            else:
                if target_parity == 0:
                    result_state[j] += state[j]
                if target_parity == 1:
                    result_state[j] += phase_factor * state[
     j]
    print(f'Gate csk on control qubit {control_qubit} and
     target qubit {target_qubit} with k = {k}'),
    return result_state
```

Explanation. This function applies a controlled phase gate that depends on the parameter k. The gate multiplies amplitudes by $e^{i\pi/k}$ only when both the control and target qubits are 1. For each basis index j:

- If the control bit is 0, the amplitude ψ_j is copied unchanged.
- If the control bit is 1, then:
  - If the target bit is 0, the amplitude is unchanged.
  - If the target bit is 1, the amplitude is multiplied by $e^{i\pi/k}$.

Within the two-qubit subspace spanned by the control and target qubits, the action is

$$|00\rangle \mapsto |00\rangle, \quad |01\rangle \mapsto |01\rangle, \quad |10\rangle \mapsto |10\rangle, \quad |11\rangle \mapsto e^{i\pi/k}|11\rangle,$$

which corresponds to the unitary matrix $\mathrm{diag}(1, 1, 1, e^{i\pi/k})$.

4.30 Sign Flip (`sign_flip`)

Listing 12. Negate the amplitude at a single basis index (in-place).

```
def sign_flip(num_qubits, index, state):
    result_state = state
    result_state[index] = -state[index]
    print(f'Sign flip on index {index}'),
    return result_state
```

Explanation. Negates the amplitude stored at `state[index]` and returns the same array object. The assignment `result_state = state` only creates an alias, so the modification happens *in place*. This is $O(1)$ time and uses no extra memory.

4.31 Bit reversal and setup (`reverse_state`)

Listing 13. Reverse bit order of basis indices; initialize state and measurement arrays.

```
def reverse_state(num_qubits, state):
    result_state = np.zeros(2 ** num_qubits, dtype=np.
     complex128)
```

```python
    for j in range(2 ** num_qubits):
        binary_str = ("{:0%db}" % num_qubits).format(j)[::]
        reversed_binary_str = ("{:0%db}" % num_qubits).format(j)[::-1]
        result_state[int(reversed_binary_str, 2)] = state[int(binary_str, 2)]
    return result_state

print(f'\nNumber of qubits: {num_qubits}')
state_vector = np.zeros(2 ** num_qubits, dtype=np.complex128)

measurement_array = np.zeros(num_qubits)
```

Explanation. `reverse_state` permutes the state vector by reversing the n-bit binary label of each basis index. For each index j in `0..2**num_qubits-1`, the code builds its zero-padded n-bit string, creates a reversed copy, converts that back to an integer, and copies the amplitude from `state[j]` to the reversed index in `result_state`. This implements an endianness swap (bit-reversal permutation) commonly used after QFT/IQFT-style routines. The function returns a new array and does not modify the input in place.

After the function, the script prints the qubit count, allocates a zero-filled complex state vector of length `2**num_qubits`, and creates a `measurement_array` of length `num_qubits`.

4.32 Initial state setup

Listing 14. Initial state preparation modes (-1, -2, -3).

```python
if initial_state == -1:
    state_vector[0] = 1
elif initial_state == -2:
    for k in range(2 ** num_qubits):
        if k >= qubit_start and k <= qubit_end:
            state_vector[k] = random.uniform(-1, 1) + 1j * random.uniform(-1, 1)
elif initial_state == -3:
    for k in range(2 ** num_qubits):
        state_vector[k] = (m ** k) % N

if initial_state != -1:
    norm = np.sqrt(np.sum(np.abs(state_vector) ** 2))
    state_vector /= norm

print('Initial state: |psi> = ')
```

Explanation. Prepares the starting state vector (length $2^{\text{num\_qubits}}$). Mode `-1`: sets $|0\cdots0\rangle$ by placing 1 at index 0 (already normalized). Mode `-2`: fills a contiguous index window `[qubit_start, qubit_end]` with i.i.d. random complex entries drawn from `Uniform(-1,1)` for real and imaginary parts; other entries stay 0. Mode `-3`: assigns each entry the modular value $(m^k) \bmod N$ (real), one per index k. For any mode other than `-1`, the vector is normalized by $\psi \leftarrow \psi/\|\psi\|$, where $\|\psi\| = \sqrt{\sum_k |\psi_k|^2}$. Finally, a header string for printing the state is emitted.

4.33 Printing the initial state

Listing 15. Pretty-print nonzero amplitudes of the prepared state vector.

```python
if initial_state == -1:
    psi = ''
    for k in range(1):
        binary_str = ("{:0%db}" % num_qubits).format(k)[::-1]
        if state_vector[k] != 0:
            psi += f'({state_vector[k]:.3f})|{binary_str}> '
    print(psi)
else:
    psi = ''
    for k in range(2 ** num_qubits):
        binary_str = ("{:0%db}" % num_qubits).format(k)[::-1]
        if state_vector[k] != 0:
            psi += f'({state_vector[k]:.3f})|{binary_str}> '
    print(psi)
```

Explanation. Builds a human-readable string for the state vector by concatenating each nonzero amplitude and its basis label. The bitstring for index `k` is produced by zero-padding to `num_qubits` bits and then reversing with `[::-1]` so qubit 0 (LSB) appears on the left. For `initial_state == -1`, only $k = 0$ is considered; otherwise all $2^{\text{num\_qubits}}$ indices are scanned. The final string `psi` is printed once.

4.34 Command parsing and simple-gate dispatch

Listing 16. Parse gate mnemonic and, for simple gates, extract the target qubit interval.

```python
for i in range(len(command_list)):
    command = command_list[i]
    before, sep, after = command.rpartition(";")
    if before.split() != []:
        gate = before.split()[0]
    else:
        gate = ''

    if gate in ['id', 'h', 'hn', 'x', 'y', 'z', 's', 'sdg', 't', 'tdg', 'measure']:
        qubit_start, qubit_end, _, _ = extract_qubits(command)
```

Explanation. Iterates over each source line `command`. The call `rpartition(";")` splits at the last semicolon, giving the part before it in `before`. If `before` has tokens, the first token is taken as the gate mnemonic in `gate`; otherwise `gate` is set to the empty string. For recognized simple gates (`id, h, hn, x, y, z, s, sdg, t, tdg, measure`), the code calls `extract_qubits(command)` to obtain the inclusive target interval `[qubit_start, qubit_end]` (extra return values are ignored here). Subsequent logic (shown elsewhere) applies the corresponding kernel over that interval.

4.35 Hadamard range handling

Listing 17. Hadamard range handling

```python
if gate == 'h':
    if qubit_start == 0 and qubit_end == num_qubits - 1:
        result_state = hadamard_n(num_qubits, state_vector)
        state_vector, _ = print_state(gate, num_qubits, verbose_mode, result_state)
    elif qubit_end >= qubit_start:
        for qubit in range(qubit_start, qubit_end + 1):
            result_state = hadamard_gate(num_qubits, qubit, state_vector)
            state_vector, _ = print_state(gate, num_qubits, verbose_mode, result_state)
    elif qubit_end < qubit_start:
        for qubit in range(qubit_start, qubit_end - 1, -1):
```

```
        result_state = hadamard_gate(num_qubits, qubit,
  state_vector)
        state_vector, _ = print_state(gate, num_qubits,
  verbose_mode, result_state)
```

Explanation. Dispatches the Hadamard operation based on the requested interval. If the interval spans the whole register (`0..num_qubits-1`), it calls `hadamard_n` once to apply $H^{\otimes n}$. Otherwise it sweeps the interval either forward (`range(qubit_start, qubit_end + 1)`) or backward (`range(qubit_start, qubit_end - 1, -1)`), calling `hadamard_gate` on each target qubit. After each application, the new vector is passed to `print_state` and its first return replaces `state_vector`, so effects accumulate in sequence. Endpoints are inclusive in both directions.

4.36 Pauli-X range handling

Listing 18. Apply X over an inclusive qubit interval (respects forward/backward order).

```
if gate == 'x':
    if qubit_end >= qubit_start:
        for qubit in range(qubit_start, qubit_end + 1):
            result_state = pauli_x(num_qubits, qubit,
  state_vector)
            state_vector, _ = print_state(gate, num_qubits,
  verbose_mode, result_state)
    elif qubit_end < qubit_start:
        for qubit in range(qubit_start, qubit_end - 1, -1):
            result_state = pauli_x(num_qubits, qubit,
  state_vector)
            state_vector, _ = print_state(gate, num_qubits,
  verbose_mode, result_state)
```

Explanation. Applies the Pauli-X gate to every qubit in the interval `[qubit_start, qubit_end]`. If the bounds are ascending, it iterates forward with `range(qubit_start, qubit_end + 1)`; if the bounds are reversed, it iterates backward with `range(qubit_start, qubit_end - 1, -1)`. For each target qubit, `pauli_x` returns a new vector with that qubit flipped on all basis components; the new vector is passed to `print_state` and then assigned back to `state_vector` so effects accumulate.

4.37 Pauli-Y range handling

Listing 19. Apply Y over an inclusive qubit interval (respects forward/backward order)

```
if gate == 'y':
    if qubit_end >= qubit_start:
        for qubit in range(qubit_start, qubit_end + 1):
            result_state = pauli_y(num_qubits, qubit,
  state_vector)
            state_vector, _ = print_state(gate, num_qubits,
  verbose_mode, result_state)
    elif qubit_end < qubit_start:
        for qubit in range(qubit_start, qubit_end - 1, -1):
            result_state = pauli_y(num_qubits, qubit,
  state_vector)
            state_vector, _ = print_state(gate, num_qubits,
  verbose_mode, result_state)
```

Explanation. Applies the Pauli-Y gate to every qubit in the interval `[qubit_start, qubit_end]`. If the bounds are ascending, it iterates forward with `range(qubit_start, qubit_end + 1)`; if the bounds are reversed, it iterates backward with `range(qubit_start, qubit_end - 1, -1)`. For each target qubit, `pauli_y` returns a new state where that qubit is flipped with the appropriate $\pm i$ phase; the result is passed to `print_state` and then assigned back to `state_vector` so effects accumulate.

4.38 Pauli-Z range handling

Listing 20. Apply Z over an inclusive qubit interval (respects forward/backward order)

```
if gate == 'z':
    if qubit_end >= qubit_start:
        for qubit in range(qubit_start, qubit_end + 1):
            result_state = pauli_z(num_qubits, qubit,
  state_vector)
            state_vector, _ = print_state(gate, num_qubits,
  verbose_mode, result_state)
    elif qubit_end < qubit_start:
        for qubit in range(qubit_start, qubit_end - 1, -1):
            result_state = pauli_z(num_qubits, qubit,
  state_vector)
            state_vector, _ = print_state(gate, num_qubits,
  verbose_mode, result_state)
```

Explanation. Applies the Pauli-Z gate to every qubit in `[qubit_start, qubit_end]`. If the bounds are ascending, it iterates forward with `range(qubit_start, qubit_end + 1)`; if the bounds are reversed, it iterates backward with `range(qubit_start, qubit_end - 1, -1)`. For each target qubit, `pauli_z` multiplies amplitudes with that bit equal to 1 by -1 (bit 0 unchanged). The updated vector is passed to `print_state` and then assigned back to `state_vector` so effects accumulate.

4.39 Phase-S range handling

Listing 21. Apply S over an inclusive qubit interval (respects forward/backward order)

```
if gate == 's':
    if qubit_end >= qubit_start:
        for qubit in range(qubit_start, qubit_end + 1):
            result_state = phase_gate(num_qubits, qubit,
  state_vector)
            state_vector, _ = print_state(gate, num_qubits,
  verbose_mode, result_state)
    elif qubit_end < qubit_start:
        for qubit in range(qubit_start, qubit_end - 1, -1):
            result_state = phase_gate(num_qubits, qubit,
  state_vector)
            state_vector, _ = print_state(gate, num_qubits,
  verbose_mode, result_state)
```

Explanation. Applies the Phase-S gate to every qubit in `[qubit_start, qubit_end]`. If the bounds are ascending, it iterates with `range(qubit_start, qubit_end + 1)`; if the bounds are reversed, it iterates with `range(qubit_start, qubit_end - 1, -1)`. For each target `qubit`, `phase_gate` multiplies amplitudes with that bit equal to 1 by i (bit 0 unchanged). The returned vector is printed (optionally) and then assigned back to `state_vector` so effects compose across the sweep.

4.40 Phase-sdg range handling

Listing 22. Apply sdg over an inclusive qubit interval (respects forward/backward order)

```python
if gate == 'sdg':
    if qubit_end >= qubit_start:
        for qubit in range(qubit_start, qubit_end + 1):
            result_state = phase_dagger_gate(num_qubits,
     qubit, state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
    elif qubit_end < qubit_start:
        for qubit in range(qubit_start, qubit_end - 1, -1):
            result_state = phase_dagger_gate(num_qubits,
     qubit, state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
```

Explanation. Applies the inverse Phase gate $S^\dagger$ to every qubit in `[qubit_start, qubit_end]`. If the bounds are ascending, it iterates with `range(qubit_start, qubit_end + 1)`; if they are reversed, it iterates with `range(qubit_start, qubit_end - 1, -1)`. For each target `qubit`, `phase_dagger_gate` multiplies amplitudes whose bit value is 1 by $-i$ (bit 0 unchanged). The returned vector is optionally printed and then assigned back to `state_vector` so effects compose across the sweep.

4.41 T gate range handling

Listing 23. Apply T over an inclusive qubit interval (respects forward/backward order)

```python
if gate == 't':
    if qubit_end >= qubit_start:
        for qubit in range(qubit_start, qubit_end + 1):
            result_state = t_gate(num_qubits, qubit,
     state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
    elif qubit_end < qubit_start:
        for qubit in range(qubit_start, qubit_end - 1, -1):
            result_state = t_gate(num_qubits, qubit,
     state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
```

Explanation. Applies the T gate to every qubit in `[qubit_start, qubit_end]`. If the bounds are ascending, it iterates with `range(qubit_start, qubit_end + 1)`; if the bounds are reversed, it iterates with `range(qubit_start, qubit_end - 1, -1)`. For each target `qubit`, `t_gate` multiplies amplitudes whose bit value is 1 by $e^{i\pi/4}$ (bit 0 unchanged). The returned vector is optionally printed and then assigned back to `state_vector` so phases compose across the sweep.

4.42 T-dagger gate range handling

Listing 24. Apply tdg over an inclusive qubit interval (respects forward/backward order)

```python
if gate == 'tdg':
    if qubit_end >= qubit_start:
        for qubit in range(qubit_start, qubit_end + 1):
            result_state = t_dagger_gate(num_qubits, qubit,
     state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
    elif qubit_end < qubit_start:
        for qubit in range(qubit_start, qubit_end - 1, -1):
            result_state = t_dagger_gate(num_qubits, qubit,
     state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
```

Explanation. Applies the inverse T gate to every qubit in `[qubit_start, qubit_end]`. If the bounds are ascending, it iterates with `range(qubit_start, qubit_end + 1)`; if the bounds are reversed, it iterates with `range(qubit_start, qubit_end - 1,  -1)`. For each target `qubit`, `t_dagger_gate` multiplies amplitudes with that bit equal to 1 by $e^{-i\pi/4}$ (bit 0 unchanged). The returned vector is optionally printed and then assigned back to `state_vector` so phases compose across the sweep.

4.43 `sk` range handling

Listing 25. Apply sk over an inclusive qubit interval (respects forward/backward order)

```python
if gate == 'sk':
    qubit_start, qubit_end, k, _ = extract_qubits(command)
    k_log = int(np.log2(abs(k)))
    if qubit_end >= qubit_start:
        for qubit in range(qubit_start, qubit_end + 1):
            result_state = sk_gate(num_qubits, qubit, k,
     state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
    elif qubit_end < qubit_start:
        for qubit in range(qubit_start, qubit_end - 1, -1):
            result_state = sk_gate(num_qubits, qubit, k,
     state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
```

Explanation. Parses the inclusive target interval `[qubit_start, qubit_end]` and the parameter `k`, then applies `sk_gate` to each qubit in that interval. If the bounds are ascending it iterates forward with `range(qubit_start, qubit_end + 1)`; if reversed it iterates with `range(qubit_start, qubit_end - 1, -1)`. Each call to `sk_gate` multiplies amplitudes with the target bit equal to 1 by a phase factor while leaving bit 0 unchanged. The freshly computed `result_state` is passed to `print_state` and then assigned back to `state_vector` so effects accumulate across the sweep.

4.44 CNOT (`cx`) range handling

Listing 26. Apply cx when either control or target is a range (other end fixed)

```python
if gate == 'cx':
    control_start, control_end, target_start, target_end =
     extract_qubits(command)
    if control_end > control_start and target_start ==
     target_end:
        target_qubit = target_start
        for control_qubit in range(control_start,
     control_end + 1):
            result_state = controlled_x(num_qubits,
     control_qubit, target_qubit, state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
    elif control_end < control_start and target_start ==
     target_end:
        target_qubit = target_start
        for control_qubit in range(control_start,
     control_end - 1, -1):
            result_state = controlled_x(num_qubits,
     control_qubit, target_qubit, state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
    elif control_end == control_start and target_end >=
     target_start:
        control_qubit = control_start
        for target_qubit in range(target_start, target_end +
      1):
            result_state = controlled_x(num_qubits,
     control_qubit, target_qubit, state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
    elif control_end == control_start and target_start >=
     target_end:
        control_qubit = control_start
        for target_qubit in range(target_start, target_end -
      1, -1):
            result_state = controlled_x(num_qubits,
     control_qubit, target_qubit, state_vector)
            state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
```

Explanation. Handles CNOT sweeps where exactly one endpoint varies: either a range of controls with a fixed target, or a fixed control with a range of targets. The code honors the user's order: forward ranges use `range(a, b + 1)`, reversed ranges use `range(a, b - 1, -1)`. Each call to `controlled_x` produces a new state vector; `print_state` can log it, and the first return replaces `state_vector` so effects accumulate.Time $O(m \cdot 2^{\texttt{num\_qubits}})$ where m is the number of controls or targets iterated; one extra state vector is allocated per step.

4.45 `csk` dispatch

Listing 27. Execute a single controlled parameterized phase (csk)

```
if gate == 'csk':
    control_qubit, target_qubit, k, _ = extract_qubits(
     command)
    k_log = int(np.log2(abs(k)))
    result_state = controlled_sk(num_qubits, control_qubit,
     target_qubit, k, state_vector)
    state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
```

Explanation. Parses `control_qubit`, `target_qubit`, and the scalar parameter `k` from `command`, then applies one `controlled_sk` operation. The helper computes `k_log = int(log2(|k|))` (not used further here). The call to `controlled_sk` multiplies amplitudes by $e^{i\pi/k}$ only on basis states where both the control and target bits are 1; all other amplitudes pass through unchanged. The new vector is optionally printed via `print_state` and then adopted as `state_vector`.

4.46 `Sign` operator handling

Listing 28. Execute a single Sign operator (in-place amplitude negation)

```
if gate == 'Sign':
    index, _, _, _ = extract_qubits(command)
    result_state = sign_flip(num_qubits, index, state_vector)

    state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
```

Explanation. Parses the target `index` from the command and applies `sign_flip`, which negates the amplitude at that basis index. The helper performs an *in-place* update (it returns the same array object), after which `print_state` may log the new state and the first return is assigned back to `state_vector` for consistency with other handlers.

4.47 `reverse` operator handling

Listing 29. Apply reverse (bit-order permutation) to entire register

```
if gate == 'reverse':
    result_state = reverse_state(num_qubits, state_vector)
    state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
```

Explanation. Invokes `reverse_state` to permute amplitudes by reversing the bit order of basis indices (endianness swap). The function returns a new vector; `print_state` may log it, and the first return replaces `state_vector` so subsequent commands see the permuted state. The permutation is unitary and norm-preserving.

4.48 QFT / IQFT handling

Listing 30. Apply QFT or IQFT over an inclusive qubit range

```
if gate in ['QFT', 'IQFT']:
    qubit_start, qubit_end, _, _ = extract_qubits(command)
    if gate == 'QFT':
        print(f'Starting QFT from qubit {qubit_start} to
     qubit {qubit_end}')
        type = 1
    if gate == 'IQFT':
        print(f'Starting IQFT from qubit {qubit_start} to
     qubit {qubit_end}')
        type = -1
    result_state = dft(num_qubits, qubit_start, qubit_end,
     type, state_vector)
    if gate == 'QFT':
        print('Ending QFT ..')
    elif gate == 'IQFT':
        print('Ending IQFT ..')
    state_vector, _ = print_state(gate, num_qubits,
     verbose_mode, result_state)
```

Explanation. Dispatches the Quantum Fourier Transform or its inverse on the inclusive interval `[qubit_start, qubit_end]` parsed from the command. It prints a start message, sets a sign flag (`type = +1` for QFT, `type = -1` for IQFT), and calls `dft` to apply the transform on that contiguous block while leaving other qubits fixed (per the implementation of `dft`). A matching end message is printed, and the resulting vector is passed to `print_state`; its first return replaces `state_vector` so subsequent commands see the transformed state.

4.49 Measurement range handling

Listing 31. Mark a qubit interval for measurement

```
if gate == 'measure':
    if qubit_end >= qubit_start:
        for qubit in range(qubit_start, qubit_end + 1):
            measurement_array[qubit] = 1
            print(f'Measure qubit {qubit}')
    elif qubit_end < qubit_start:
        for qubit in range(qubit_start, qubit_end - 1, -1):
            measurement_array[qubit] = 1
            print(f'Measure qubit {qubit}')
```

Explanation. Flags every qubit in the inclusive interval `[qubit_start, qubit_end]` for measurement by setting `measurement_array[qubit] = 1`. The code honors forward and reverse ranges; order only affects the print messages and not the final mask. This step *does not* collapse the state or sample outcomes—it merely records which qubits should be measured later.

4.50 Measurement post-processing: probabilities

Listing 32. calculate$_r$*esults* : *aggregateoutcomeprobabilities*

```
def calculate_results(num_qubits, state, measurement_array):
    probabilities = np.zeros(int(2 ** np.sum(
      measurement_array)))
    amplitudes = ['' for _ in range(int(2 ** np.sum(
      measurement_array)))]

    for i in range(2 ** num_qubits):
        num = 0
        k = 0
        for j in range(num_qubits):
            if measurement_array[j] == 1:
                num += ((i >> j) & 1) * 2 ** k
                k += 1
        probabilities[num] += np.absolute(state[i]) ** 2
```

Explanation. Allocates arrays for $m = \sum$ `measurement_array` measured qubits: `probabilities` and `amplitudes` of length 2^m. The first loop buckets each basis index `i` into an outcome index `num` by packing the measured bits in **little-endian order of qubit indices** (lower qubit indices contribute to less-significant bits). It then accumulates $|\psi_i|^2$ into `probabilities[num]`.

4.51 Measurement post-processing: conditional states

Listing 33. calculate$_r$*esults* : *buildnormalizedconditionalstates*

```
    for i in range(2 ** num_qubits):
        num = 0
        k = 0
        for j in range(num_qubits):
            if measurement_array[j] == 1:
                num += ((i >> j) & 1) * 2 ** k
                k += 1
        binary_str = ("{:0%db}" % num_qubits).format(i)
      [::-1]
        if np.absolute(state[i]) > 0.0001:
            if len(amplitudes[num]) == 0:
                amplitudes[num] = amplitudes[num] + f'({
      state[i] / np.sqrt(probabilities[num]):.3f})|{
      binary_str}>'
            else:
                amplitudes[num] = amplitudes[num] + f' + ({
      state[i] / np.sqrt(probabilities[num]):.3f})|{
      binary_str}>'
```

Explanation. Repeats the same bucketing to append each significant component ($|\psi_i| > 10^{-4}$) to the string for that outcome `num`. Each appended amplitude is **normalized conditionally** by dividing by $\sqrt{P(\texttt{num})}$ so that the printed ψ is unit norm within that outcome.

4.52 Measurement post-processing: reporting

Listing 34. calculate$_r$*esults* : *printprobabilitiesandstates*

```
    if np.sum(measurement_array) > 0:
        if np.sum(measurement_array) > 1:
            print('\nProbabilities for measurements of
      qubits: '),
        else:
            print('\nProbability for measurement of qubit: ')
      ,
        for i in range(num_qubits):
            if measurement_array[i] == 1: print(f'{i} '),

    for i in range(int(2 ** np.sum(measurement_array))):
        binary_str = ("{:0%db}" % np.sum(measurement_array)).
      format(i)[::-1]
        if probabilities[i] > 0.00000000001:
            if np.sum(measurement_array) > 0:
                print(f'\nP({binary_str}) = '),
                print(probabilities[i])
            print('|psi> = '),
            print(amplitudes[i])

    return
```

Explanation. If any qubits are flagged for measurement, prints the list of qubit indices. Then, for each outcome label `i` with non-negligible probability ($> 10^{-11}$), prints the probability and the assembled conditional state string.

4.53 Final results (`calculate_final_results`)

Listing 35. Print basis states above a probability threshold

```
def calculate_final_results(num_qubits, state,
      probability_threshold):
    print(f'\nFinal basis states with P > {
      probability_threshold}')
    probabilities = np.absolute(state) ** 2
    for k in range(2 ** num_qubits):
        if probabilities[k] > probability_threshold:
            binary_str = ("{:0%db}" % num_qubits).format(k)
      [::-1]
            psi = f'P(|{binary_str}>) = {probabilities[k]:.2
      e}\t Amplitude: {state[k]:.2e}'
            print(psi)
    return
```

Explanation. Computes the per-basis probabilities $P_k = |\psi_k|^2$ and prints only those indices k whose P_k exceeds `probability_threshold`. Each printed line includes the little-endian bit label for k and shows both P_k and the raw amplitude ψ_k in scientific notation.

4.54 Plotting results (`plot_results`)

Listing 36. Plot basis-state probability envelope

```
def plot_results(num_qubits, state):
    probabilities = np.absolute(state) ** 2
    R = 1
    if num_qubits > 20:
        R = 2 ** (num_qubits - 20)
    y1 = np.reshape(probabilities, (-1, R)).max(axis=1)
    x = np.arange(len(y1)) * R

    fig, ax = plt.subplots(figsize=(8, 7))
    ax.set_title("Probabilities for All Basis States",
      fontsize=14)
    ax.set_xlabel("Basis State", fontsize=12)
    ax.set_ylabel("Probability", fontsize=12)
    ax.plot(x, y1, 'bo', x, y1, 'r--')

    plt.savefig('plot.png')
    plt.close(fig)
    print("Graph saved as plot.png!")
```

Explanation. Computes $P_k = |\psi_k|^2$ and, for large registers, downsamples along the basis index by a factor $R = 2^{\max(0,n-20)}$. The probability array is reshaped to `(2**num_qubits / R, R)` and the blockwise maximum is taken to form an envelope `y1`; `x` is scaled by `R` so indices remain on the original scale. The plot overlays blue markers and a red dashed line, saves to `plot.png`, then closes the figure.

4.55 Interactive plotting with Tkinter

Listing 37. Tkinter plotting wrapper (embeds a Matplotlib figure)

```
def plot_tkinter(num_qubits, state):

    probabilities = np.absolute(state) ** 2
    R = 1
    if num_qubits > 20:
        R = 2 ** (num_qubits - 20)
    y1 = np.reshape(probabilities, (-1, R)).max(axis=1)
    x = np.arange(len(y1)) * R

    fig, ax = plt.subplots(figsize=(8, 7))
    ax.set_title("Probabilities for All Basis States",
     fontsize=14)
    ax.set_xlabel("Basis State", fontsize=12)
    ax.set_ylabel("Probability", fontsize=12)
    ax.plot(x, y1, 'bo', x, y1, 'r--')

    root = tk.Tk()
    root.title("Quantum Probability Graph")

    canvas = FigureCanvasTkAgg(fig, master=root)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

    close_button = tk.Button(root, text="Close", command=
     lambda: quit_tkinter(root))
    close_button.pack()

    print("Graph successfully displayed in Tkinter!")

    root.mainloop()
```

Explanation. Computes probabilities `abs(state)**2`, downsamples along the basis index by a factor `R` when `num_qubits` > 20 via blockwise maximum to form an envelope, and plots the result with Matplotlib. The figure is then embedded into a Tk window using `FigureCanvasTkAgg`; a "Close" button calls `quit_tkinter(root)` (defined elsewhere) to terminate the GUI. The function prints a confirmation and enters `root.mainloop()` until the window is closed.

5. CONCLUSION AND RESULTS

We presented a lightweight, state–vector quantum simulator written in Python that emphasizes clarity, explicit bit–level semantics, and paper–friendly reproducibility. The implementation covers a practical gate set for algorithm sketches: single–qubit Clifford and T gates ($X, Y, Z, S, S^\dagger, T, T^\dagger$), a parameterized phase S_k, multi–qubit Hadamard via $H^{\otimes n}$, two–qubit primitives (CNOT and controlled–S_k), index–space utilities (`reverse`, `Sign`), measurement masking with post–processing, and simple visualization through file and GUI plots. All kernels act on a flat array of length 2^n using little–endian indexing (qubit 0 is the least–significant bit), and the "range semantics" preserve user–specified forward or backward sweeps across contiguous qubit intervals.

What this enables. The simulator is well–suited for:

- *Didactic experiments:* step–by–step inspection of amplitudes after each gate; explicit control over endianness and gate ordering.
- *Algorithm prototypes:* compact implementations of QFT/IQFT blocks, phase–kickback patterns, and Grover–style sign inversions.
- *Deterministic reporting:* printable state summaries, outcome marginals over arbitrary measured subsets, and quick probability plots.

Correctness considerations. Each gate kernel is norm–preserving by construction (unitary permutations and unit–modulus phases), and fresh output buffers avoid in–place overwrite hazards (except the intentional in–place `Sign`). Measurement post–processing aggregates marginals consistently with the chosen endianness and prints normalized conditional states.

Takeaway. Within its intended regime (small to moderate n), the simulator offers a transparent reference for algorithmization: every amplitude move is explicit, every phase is visible, and every measurement marginal can be inspected. This makes it a useful bridge between textbook derivations and executable prototypes, and a solid baseline for future optimized or physically richer simulators.

REFERENCES

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information* (10th Anniversary ed.), Cambridge University Press, 2010. doi:10.1017/CBO9780511976667.

[2] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM J. Comput.* **26**(5), 1484–1509 (1997). doi:10.1137/S0097539795293172.

[3] L. K. Grover, "Quantum Mechanics Helps in Searching for a Needle in a Haystack," *Phys. Rev. Lett.* **79**, 325–328 (1997). doi:10.1103/PhysRevLett.79.325.

[4] D. Coppersmith, "An Approximate Fourier Transform Useful in Quantum Factoring," IBM Research Report RC19642 (1994); arXiv:quant-ph/0201067 (2002).

[5] S. Abbas *et al.*, "Quantum computing with Qiskit," arXiv:2405.08810 (2024).

[6] Google Quantum AI, "Cirq basics," online documentation (accessed 26 Aug 2025). https://quantumai.google/cirq

[7] C. R. Harris *et al.*, "Array programming with NumPy," *Nature* **585**, 357–362 (2020). doi:10.1038/s41586-020-2649-2.