

浙江大学

量子计算理论基础与软件系统实验报告

课程名称: 量子计算理论基础与软件系统

作业名称: Quantum Circuit Simulation

姓名: 周楠

学号: 3220102535

电子邮箱: 3220102535@zju.edu.cn

联系电话: 19858621101

指导教师: 卢丽强

2024 年 9 月 27 日

一. 实验目的和要求

本次实验中，我们使用 qubit-simulator 研究量子电路模拟。qubit-simulator 是一个简单而轻量级的 Python 包，提供了一个用于模拟量子比特和量子门的量子模拟器，支持基本的量子操作，如 Hadamard 门、 $\pi/8$ 相位旋转门、受控非门和一般酉变换。

二. 实验环境

```
1   conda create -n quantum python=3.10
2   conda activate quantum
3   conda deactivate
4   conda env remove -n quantum
```

在命令行中使用如下指令安装 ‘qubit-simulator’

```
1   pip install qubit-simulator
```

三. 实验流程

3.1 qubit-simulator 源代码分析

根据 ‘qubit_simulator’ 中的源代码分析 ‘qubit-simulator’ 的基本原理、结构及运行流程。

3.1.1 Gate 类的解释

1. 预定义的常用量子门

- (a) Hadamard (H) 门: 用于将量子比特从基态 0 或 1 变为叠加态。

```
1 np.ndarray = np.array([[1, 1], [1, -1]]) / np.sqrt(2)
```

- (b) $\pi/8$ (T) 门: 是一个相移门, 对量子态应用相位变化。

```
1 np.ndarray = np.array([[1, 0], [0, np.exp(1j * np.pi / 8)]])
```

P门是一个相位旋转门, 它将量子比特的相位进行旋转, 带有一个输入参数用于确定具体相位; S门是P门的特例, 其中相位参数 ϕ 等于 $\frac{\pi}{2}$, T门是另一个相位旋转门, 它将量子比特的相位旋转 $\frac{\pi}{4}$ 。



$$P = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}, \quad S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}$$

图 1: Pauli-X (NOT) 门

- (c) Pauli-X (NOT) 门: 类似经典计算中的 NOT 操作, 将 0 和 1 互换。

```
1 np.ndarray = np.array([[0, 1], [1, 0]])
```

2. 通用 U 门

```
1 @staticmethod
2 def U(theta: float, phi: float, lambda_: float) -> np.ndarray:
3     """
4     Generic (U) gate.
5
6     :param theta: Angle theta.
7     :param phi: Angle phi.
```

```

8      :param lambda_: Angle lambda.
9      :return: Unitary matrix representing the U gate.
10     """
11     return np.array(
12         [
13             [np.cos(theta), -np.exp(1j * lambda_) * np.sin(theta)
14              ],
15             [np.exp(1j * phi) * np.sin(theta), np.exp(1j * (phi +
16                 lambda_)) * np.cos(theta)],
17         ]
18     )

```

U 方法用于创建一个通用的量子门，参数化的 U 门可以表示任意的单量子比特操作：theta, phi, lambda_ 是角度，用于定义旋转、相移等操作。返回一个 2x2 的幺正矩阵（量子计算中的基本操作矩阵）。

$$U(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta) & -e^{i\lambda} \sin(\theta) \\ e^{i\phi} \sin(\theta) & e^{i(\phi+\lambda)} \cos(\theta) \end{bmatrix}$$

3. 受控门的生成

```

1  @staticmethod
2  def create_controlled_gate(
3      gate: np.ndarray, control_qubit: int, target_qubit: int,
4      num_qubits: int
5  ) -> np.ndarray:
6      """
7      Creates a controlled gate.
8
9      :param gate: Matrix representing the gate.
10     :param control_qubit: Index of the control qubit.
11     :param target_qubit: Index of the target qubit.
12     :param num_qubits: Total number of qubits.
13     :return: Matrix representing the controlled gate.
14     """

```

create_controlled_gate 方法用于生成受控量子门（controlled quantum gate）。受控门是一种多比特操作，只有当控制比特为 1 时，目标比特才会执行某个操

作：

gate: 受控门作用的基本门（例如 Pauli-X）。

control_qubit: 控制量子比特的索引（哪一位量子比特作为控制）。

target_qubit: 目标量子比特的索引（哪一位量子比特作为目标）。

num_qubits: 系统中量子比特的总数。

4. 逆量子门

```
1 @staticmethod
2 def create_inverse_gate(gate: np.ndarray) -> np.ndarray:
3     """
4     Creates an inverse gate.
5
6     :param gate: Matrix representing the gate.
7     :return: Matrix representing the inverse gate.
8     """
9     return np.conjugate(gate.T)
```

该方法返回给定矩阵的复共轭转置，代表其逆矩阵。量子门的一个重要性质是么正性，即量子门的逆等于其共轭转置。

5. 量子门的验证: 该方法通过检查 $U^\dagger U = I$ 来确保门是么正的。如果不符合这个条件，它会抛出一个 `ValueError` 错误。

3.1.2 量子比特模拟器 QubitSimulator

1. `__init__` 方法是 QubitSimulator 类的构造函数，用于初始化量子比特模拟器。

(a) `self.state_vector = np.zeros(2**num_qubits, dtype=complex)`: 状态向量 (`state_vector`) 是一个复杂数数组，表示整个量子系统的状态。对于 n 个量子比特，状态向量的维度是 2^n ，因为每个量子比特都有两种可能的状态，整个系统的可能状态组合数是 2^n 。

这里使用了 numpy 的 `np.zeros` 函数初始化了一个大小为 2^n 的复数向量，初始时所有状态的概率幅值都设置为 0。

`self.state_vector[0] = 1`:

设置初始状态为 $|0 \dots 0\rangle$ 状态，即第一个基态。在经典计算中，这就像是系统处于一个“全为 0”的初始状态（所有量子比特都为 $|0\rangle$ ）。

此操作将 `self.state_vector` 的第一个元素设置为 1，表示整个系统的初始状态是基态 $|000\dots\rangle$ ，其概率幅为 1。

2. `_apply_gate` 方法的作用是将指定的量子门应用到量子比特系统的状态向量上，并将应用的门记录在电路历史中。

```
1 def _apply_gate(  
2     self,  
3     gate_name: str,  
4     gate: np.ndarray,  
5     target_qubit: int,  
6     control_qubit: Optional[int] = None,  
7 ):  
8     """  
9     Applies the given gate to the target qubit.  
10  
11     :param gate_name: Name of the gate.  
12     :param gate: Matrix representing the gate.  
13     :param target_qubit: Index of the target qubit.  
14     :param control_qubit: Index of the control qubit (if  
15         controlled gate).  
16     """  
17     # Validate the target and control qubit indices  
18     self._validate_qubit_index(target_qubit, control_qubit)
```

```

18     # Validate the gate
19     Gates._validate_gate(gate)
20     if control_qubit is not None:
21         operator = Gates.create_controlled_gate(
22             gate, control_qubit, target_qubit, self.num_qubits
23         )
24     else:
25         operator = np.eye(1)
26         for qubit in range(self.num_qubits):
27             operator = np.kron(
28                 operator,
29                 gate if qubit == target_qubit else np.eye(2),
30             )
31     self.state_vector = operator @ self.state_vector
32     self.circuit.append((gate_name, target_qubit, control_qubit)
33         )

```

(a) 验证量子比特索引: 确认目标和控制量子比特索引有效

(b) 验证量子门: 验证门是否是一个合法的幺正矩阵

(c) 处理受控门和非受控门的区别:

- 受控门: 如果 `control_qubit` 不是 `None`, 意味着这是一个受控门 (比如 CNOT), 那么调用 `Gates.create_controlled_gate` 方法创建受控量子门的整体操作符矩阵。该操作符会在系统中的所有量子比特上执行受控门的操作。
- 非受控门: 如果 `control_qubit` 是 `None`, 则表示这是一个单量子比特门。通过逐量子比特地构建操作符矩阵来处理该门:
`np.eye(1)` 初始化一个单位矩阵。
 使用 Kronecker 积 `np.kron` 依次构造每个量子比特的操作。如果当前处理的是目标量子比特 (`qubit == target_qubit`), 那么将门 `gate` 应用于该比特; 否则应用单位矩阵 `np.eye(2)`, 表示该比特上无操作。

(d) 更新状态向量: 在计算出完整的操作符矩阵后, 将其应用于系统的状态向量 `self.state_vector`。这相当于执行矩阵乘法 `operator state_vector`, 从而更新量子系统的状态。

(e) 记录电路历史:

3. 各种门操作方法（如 `h`, `t`, `x`, `cx`, `u`, `cu`）

每个门操作调用 `_apply_gate`，并传递相应的门矩阵和量子比特索引。

在受控门操作中（如 `cx` 和 `cu`），需要同时传递控制比特和目标比特。

通用门（`u` 和 `cu`）通过参数控制门的行为，支持逆操作。

4. `measure` 方法用于测量量子状态并返回测量结果。它模拟了量子测量的概率性，基于当前量子态返回指定次数的测量结果。

(a) 检查参数合法性: 确保测量次数（`shots`）必须是非负数

(b) 处理基变换: 如果提供了测量基 `basis`，则首先通过 `Gates._validate_gate` 方法验证这个基是否为有效的幺正矩阵。然后执行基变换，计算出在该基下的状态向量

(c) 计算测量概率: 通过取状态向量每个元素的幅值（绝对值的平方），得到每个量子状态的测量概率。概率与量子态的振幅平方成正比。

(d) 计算测量结果的期望次数: 根据测量概率以及测量次数 `shots`，对每个量子状态的测量结果进行计数，得到每个状态在 `shots` 次测量中出现的期望次数。

(e) 生成测量结果:

5. `run` 方法运行整个模拟，并返回每个可能态的测量结果次数。通过调用 `measure` 方法多次测量系统状态，并返回一个状态计数的字典。

6. `plot_wavefunction(self)` 绘制当前量子态的波函数，包括振幅和相位。使用极坐标图展示每个量子态的振幅和相位关系。

3.2 构造 5-qubit GHZ 电路

使用 ‘qubit-simulator’ 构造如下图所示的 5-qubit GHZ 电路，并模拟运行，画出结果概率分布直方图。

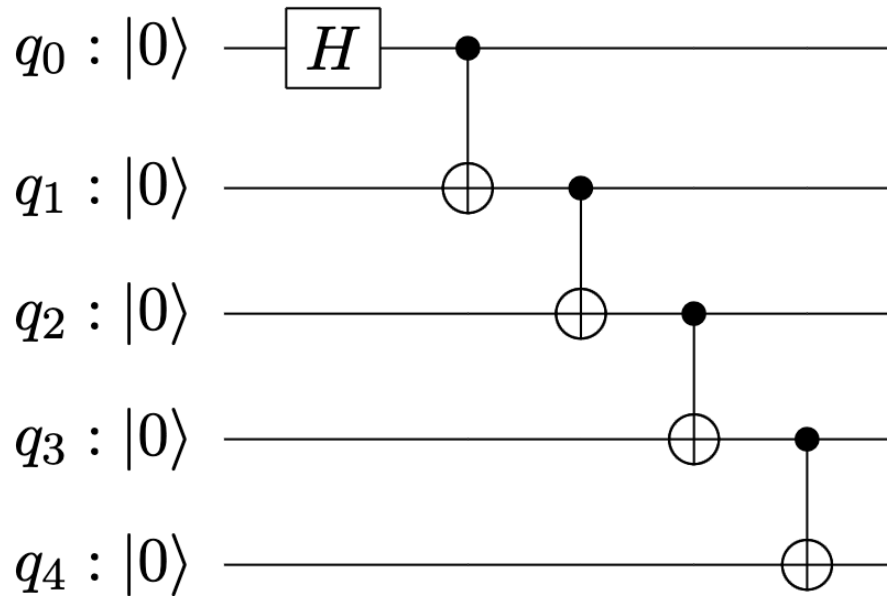


图 2: 5-qubit GHZ 电路

```
1  from qubit_simulator import QubitSimulator
2  # 用于生成 5 个量子比特的量子模拟器
3  n_qubits = 5
4  simulator = QubitSimulator(n_qubits)
5  simulator.h(0)
6  simulator.cx(0, 1)
7  simulator.cx(1, 2)
8  simulator.cx(2, 3)
9  simulator.cx(3, 4)
10 counts = simulator.run(shots=10000)
11 print(simulator)
12 print(counts)
```

```
1  from qiskit import QuantumCircuit, transpile
2  from qiskit_aer import AerSimulator
```

```

3  from qiskit.visualization import plot_histogram
4  import matplotlib.pyplot as plt
5
6  # use Aer's AerSimulator
7  simulator = AerSimulator()
8  # Create a Quantum Circuit acting on the q register
9  circuit = QuantumCircuit(5, 5)
10 # Add a H gate on qubit 0
11 circuit.h(0)
12 # Add a CX (CNOT) gate on control qubit 0 and target qubit 1
13 circuit.cx(0, 1)
14 circuit.cx(1, 2)
15 circuit.cx(2, 3)
16 circuit.cx(3, 4)
17
18 circuit.measure([0, 1, 2, 3, 4], [0, 1, 2, 3, 4])
19 # Draw the circuit
20 circuit.draw("mpl").savefig("./lab1/circuit.png")
21 # compile the circuit for the simulator
22 compiled_circuit = transpile(circuit, simulator)
23 # execute the circuit on the simulator
24 job = simulator.run(compiled_circuit, shots=1000)
25 # get the result from the job
26 result = job.result()
27 # Returns counts
28 counts = result.get_counts(circuit)
29 # 绘制直方图
30 plot_histogram(counts)
31 # 设置 x 轴标签水平显示
32 plt.xticks(rotation=0)
33 # 保存直方图
34 plt.savefig("./lab1/histogram.png")
35 print("\n Total count for 00000 and 11111 are:", counts)

```

使用了两个不同的量子模拟器来执行相同的量子电路。一个是 Qiskit 中的 Aer-Simulator，另一个是 qubit_simulator 模块中的自定义模拟器。分别得到如下相同的电路图：

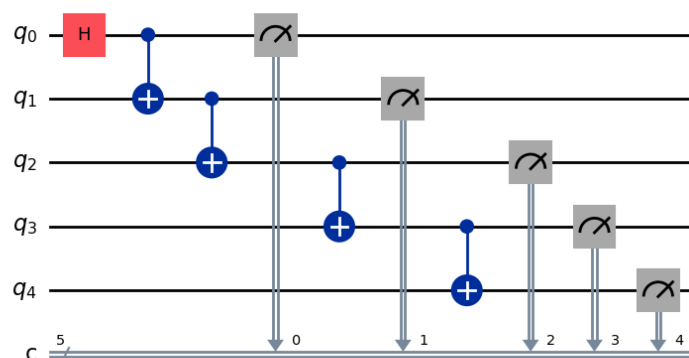


图 3: AerSimulator 电路

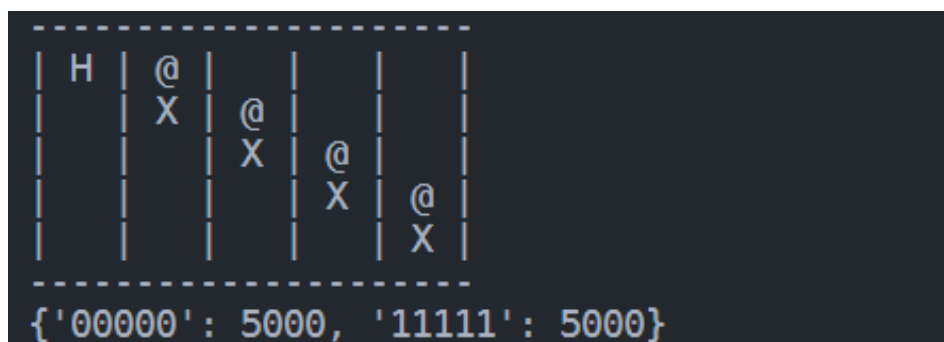


图 4: qubit_simulator 电路

3.3 观察量子电路模拟的运行时间

3.3.1 绘制运行时间与量子比特数的关系图

下段代码构造了一个量子电路，运行这段代码，并尝试调整代码中 `n_qubits` 的值，然后观察量子电路模拟的运行时间。绘制量子电路模拟运行时间与量子电路比特数的关系图，并分析 `qubit-simulator` 进行量子电路模拟的复杂度。

```
1  import random #用于生成随机浮点数，以模拟量子门中的参数
2  import time #用于测量代码运行的时间
3  import matplotlib.pyplot as plt
4  from qubit_simulator import QubitSimulator
5  from qiskit.visualization import plot_histogram
6
7  def apply_circuit(circuit, n):
8      # 在编号为 n-1 的量子比特上施加 Hadamard 门。
9      # Hadamard 门用于将量子比特从基态 |0> 或 |1> 变为叠加态。
10
11     circuit.h(n - 1)
12     for qubit in range(n - 1):
13         # 对剩下的每一对相邻的量子比特 (qubit, qubit + 1) 应用受控
14         # U 门 (cu)
15         # random.random() 生成的随机值乘以 3.14，来模拟不同的相移、
16         # 旋转或任意角度操作。
17         circuit.cu(qubit, qubit + 1, random.random() * 3.14,
18                   random.random() * 3.14, random.random() * 3.14)
19
20     # 记录运行时间
21     times = []
22     for i in range(2, 16):
23
24         n_qubits = i # change this value (<=16)
25         # QubitSimulator(n_qubits) 初始化量子模拟器，生成具有 n_qubits
26         # 个量子比特的系统。
27         simulator = QubitSimulator(n_qubits)
28
29         t = time.time()
```

```

26     apply_circuit(simulator, n_qubits)
27     run_time = time.time() - t
28     print("n_qubits = {}, run_time = {}".format(n_qubits, run_time
29           ))
30
31     times.append(run_time)
32
33     job = simulator.run(shots=1000)
34     counts = job
35     print("n_qubits = {}, counts = {}".format(n_qubits, counts))
36     # plot_histogram(counts).savefig("./lab1/histogram_{}.png".
37     #   format(n_qubits))
38
39     print(times)
40     # 绘制运行时间与量子比特数的关系图
41     plt.figure()
42     plt.plot(range(2, 16), times)
43     plt.xlabel("n_qubits")
44     plt.ylabel("run_time")
45     plt.savefig("./lab1/run_time.png")
46
47     plt.figure()
48     plt.plot(range(2, 16), times, marker='o')
49     plt.xlabel("n_qubits")
50     plt.ylabel("run_time (log scale)")
51     plt.yscale('log') # 设置 y 轴为对数刻度
52     plt.grid(True, which="both", ls="--") # 添加网格线便于阅读对数图
53     plt.savefig("./lab1/run_time_log_scale.png")

```

1. 定义量子电路的函数

在 `apply_circuit(circuit, n)` 函数中, `circuit.h(n-1)`: 在最后一个量子比特 (编号为 `n-1`) 上应用 Hadamard 门, 这会将其从基态转换为叠加态。for `qubit in range(n - 1)`: 对从 0 到 `n-2` 的每个量子比特进行操作。对于每一对相邻的量子比特 (`qubit, qubit + 1`), 应用一个受控 U 门 `cu`, 随机生成三个参数来模拟不同的旋转角度。

2. 主循环，测量运行时间

循环过程：从 2 到 15 个量子比特，逐步增加系统中量子比特的数量，每次增加一个比特。

量子电路模拟器初始化：使用 `QubitSimulator(n_qubits)` 创建一个包含 `n_qubits` 个量子比特的模拟器。

应用电路：调用 `apply_circuit` 函数，对模拟器施加量子门操作。

测量运行时间：使用 `time.time()` 计算量子电路的执行时间，保存到列表 `times` 中。

3. 绘制图表

生成的图像文件保存在 `./lab1/` 目录下，分别为：

`run_time.png`：普通运行时间图。

`run_time_log_scale.png`：带有对数 y 轴的运行时间图。

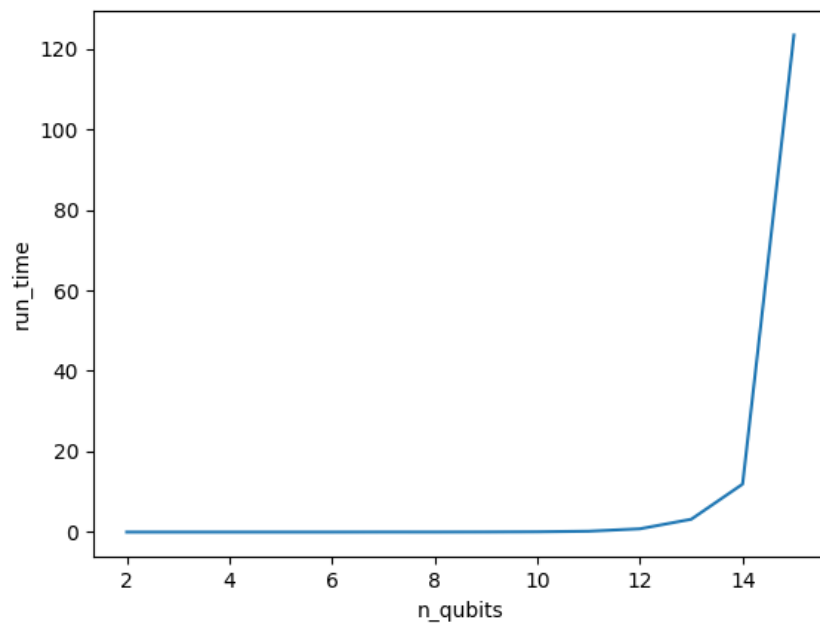


图 5: 普通运行时间图

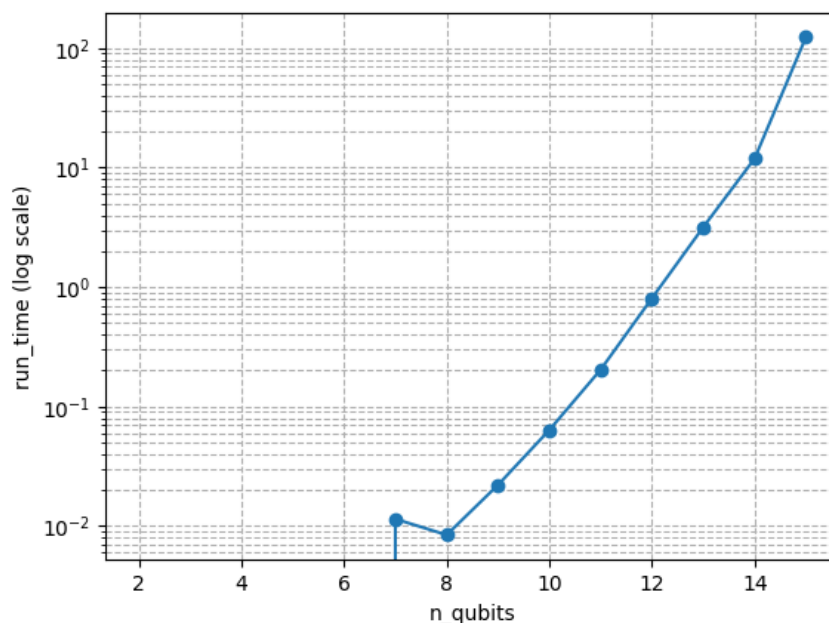


图 6: 带有对数 y 轴的运行时间图

3.3.2 量子电路模拟的复杂度

1. 量子比特数对内存需求的影响：在经典计算机中模拟量子电路的内存需求是指数级增长的。具体来说，模拟 n 个量子比特需要 2^n 个复数来描述量子态。对于每个量子比特的增加，所需的存储空间都会翻倍。因此，随着量子比特数增加，系统的内存使用量会急剧上升。
2. 门操作的复杂度：施加量子门时（例如 h 或 cu 门），量子比特的状态空间被更新。对于全局操作，更新每个量子比特的状态都会涉及计算所有量子态的幅值。门的复杂度随比特数增加，所需计算的数量通常呈指数级。

复杂度分析结果

- 时间复杂度：在经典计算机上，模拟 n 个量子比特的时间复杂度通常为 $O(2^n)$ 。量子比特数增加时，电路模拟所需的时间将指数级增加，这在实验结果中的运行时间上也体现了出来。
- 空间复杂度：空间复杂度同样是 $O(2^n)$ ，模拟的每一个额外量子比特会使存储所需的内存翻倍，这在大规模量子模拟时成为主要瓶颈。

四. 遇到的困难及解决方法

本次实验没有遇到任何困难，一帆风顺。

五. 总结与心得

在本次实验中，我们使用 qubit-simulator 研究量子电路模拟。qubit-simulator 是一个简单而轻量级的 Python 包，提供了一个用于模拟量子比特和量子门的量子模拟器。

通过分析 qubit-simulator 源代码，我了解了 qubit-simulator 的基本原理、结构及运行流程。结合代码中的门矩阵表示方法和课堂上的数学公式，我进一步的理解了从理论到实现的过程。

在构造的 5-qubit GHZ 电路中，我使用了两种不同的量子模拟器：AerSimulator, qubit_simulator。在两者相互验证中，更加熟悉如何应用量子门。