SKKU 2025 Quantum Challenge : Simulating Fluid Dynamics on a Quantum Computer

Sangyoon Woo

October 26, 2025

Problem 1: Initialization

The initialization step encodes the distribution $\phi(x, 0)$ into a quantum state. This can be achieved by using two quantum registers $|q\rangle$ for spatial dimension encoding, and $|d\rangle$ for the encoding the distribution functions. Consider a 1D lattice with M cells and Q microscopic velocities.

1-a) How many qubits in the register $|q\rangle$ are needed to encode $\phi(x, 0)$?

To distinguish M lattice positions as quantum states, each position $x \in \{0, 1, \ldots, M-1\}$ must be mapped to a distinct computational basis state $|x\rangle$. To verify this, let us consider a few examples.

First, consider the case of $M = 2$. In this case, the computational basis state $|x\rangle$ can take two possible forms, $|0\rangle$ or $|1\rangle$, and hence a single qubit is sufficient to represent all positions.

Next, consider $M = 64$. The possible states are $|000000\rangle$ to $|111111\rangle$, representing a total of 64 mutually orthogonal states. Thus, six qubits are required to encode all possible positions.

From these two examples, the number of qubits n_q required can be generally expressed as follows:

$$n_q = \log_2 M \tag{1}$$

However, this is not entirely correct. For example, if $M = 63$, then according to Eq. 1, $n_q = \log_2 63 \approx 5.977$. However, a '5.977 qubit' system has no physical meaning since the number of qubits must always be an integer.

Therefore, a ceiling function $\lceil \cdot \rceil$ must be applied to the right-hand side of Eq. 1. The ceiling function represents the smallest integer greater than or equal to $\log_2 M$. Accordingly, Eq. 2 can be expressed as:

$$n_q = \lceil \log_2 M \rceil \tag{2}$$

For instance, when $M = 63$, we obtain $n_q = \lceil \log_2 63 \rceil = 6$. With six qubits, $2^{n_q} = 64$ basis states can be represented, meaning that all 63 physical lattice sites can be uniquely encoded, with one remaining unused state.

If the number of qubits is chosen excessively large (e.g., using seven qubits for $M = 63$), efficiency decreases, and additional unused states may increase noise sensitivity. Therefore, it is recommended to use the minimum number of qubits determined by Eq. 2.

1-b) How many qubits in the register $|d\rangle$ are needed to encode distribution functions?

The number of qubits n_d required to encode the distribution functions can be determined in the same manner as in Problem 1-a. Since n qubits can represent up to 2^n distinct states, the value of n_d can be expressed as follows:

$$n_d = \lceil \log_2 Q \rceil \tag{3}$$

Here, Q represents the number of possible discrete choices that each particle can take. It is important to note that, unlike in the previous problem where M denoted the number of lattice sites, Q replaces M in the logarithmic term. As in Problem 1-a, the ceiling function must be applied to ensure that n_d is an integer.

1-c) Suppose $M = 64$, $Q = 3$, and the initial distribution is zero everywhere except at the point source $x_s = 32$, where $\phi(x_s, 0) = 1.0$. Using qiskit, write a function that returns a quantum circuit that initializes $\phi(x, 0)$).

Using Eq. 2 and 3 obtained from Problems 1-a and 1-b, we can calculate the total number of qubits required to initialize the quantum circuit. By substituting $M = 64$ into Eq. 2 and $Q = 3$ into Eq. 3, we find that the number of required qubits, n_q and n_d, are 6 and 2, respectively. Therefore, the overall configuration of the qubits can be represented as shown in Eq. 4.

$$|\psi\rangle = |q_5q_4q_3q_2q_1q_0\rangle \otimes |d_1d_0\rangle = |q_5q_4q_3q_2q_1q_0d_1d_0\rangle \tag{4}$$

The overall procedure of Problem 1 can be implemented using qiskit, as shown in Listing 1.

Although the initial position $x_s = 32$ can be represented by applying an X-gate to q_5 to invert the corresponding bit, for consistency in the iterative simulation process, it was implemented by applying an array in which the element corresponding to the index x_s is set to 1, while all other elements are set to 0.

Listing 1: Python function for initializing the quantum circuit with the initial fluid distribution $\phi(x, 0)$.

```python
initial_phi = np.zeros(M)
initial_phi[x_s] = 1.0

concentration_qc = qc.copy()
concentration_qc.initialize(initial_phi, q_reg)

concentration_qc.draw('mpl')
```

At this stage, qc represents an empty quantum circuit that contains no applied operations. It only defines the number of qubits and classical registers required for the simulation. Specifically, the circuit consists of two qubits d_0 and d_1 forming the direction register, six qubits q_0 through q_5 representing the lattice positions, and two classical registers: c_q (6 bits) for the position measurements and c_d (2 bits) for the direction measurements.

The circuit defined as `concentration_qc` in the code is shown in Fig. 1. This structure is used repeatedly during the simulation to initialize and configure the fluid distribution.

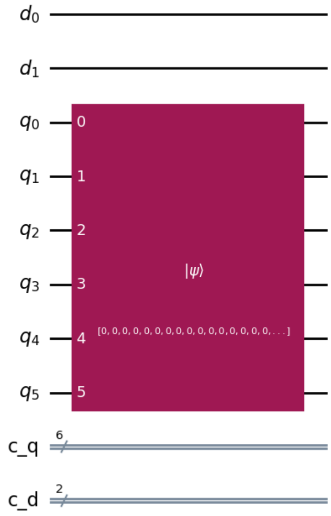

Figure 1: Quantum circuit defining the initial fluid distribution.

Problem 2: Collision

For uniform velocity field, the collision step amounts to preparing the state in the direction register $|d\rangle$ [1]:

$$|k\rangle = \frac{1}{\sqrt{2^{n_d}}} \sum_i k_i \, |i\rangle_d \tag{5}$$

where n_d is the number of qubits in the $|d\rangle$ register, and constants k_i are given by Eq. 6.

$$k_i = w_i \left[1 + \frac{e_i u_i}{c_s^2} \right] \tag{6}$$

2-a) Consider the 1D case described in the initialization part above (part c) with weight coefficients $w = [2/3, 1/6, 1/6]$, particle speeds $[0, 1, -1]$, advection speed $u = 0.2$, and the speed of sound $c_s = 1$. Using qiskit, write a function that initializes the state $\phi(x, 0)$ and applies the collision operator.

The parameters given in the problem are summarized in Table 1.

Table 1: Parameters used in Problem 2-a.

Description	**Symbol**	**Value**
Weight coefficients	w	$\left[\frac{2}{3}, \frac{1}{6}, \frac{1}{6}\right]$
Particle speeds	e	$[0,\ 1,\ -1]$
Advection speed	u	0.2
Speed of sound	c_s	1

Hence, by applying Eq. 6, the state $|k\rangle$ is obtained as $k = [0.667, 0.200, 0.133]$. The d register must also be configured such that each direction has an equal probability of being represented. However, it is important to note that the amplitudes assigned to the register should not directly correspond to the probability values [0.667, 0.200, 0.133].

In a quantum state, each amplitude represents the square root of the corresponding probability, and the sum of the squared amplitudes must be normalized to 1. Therefore, the magnitudes of the amplitudes should be computed as the square roots of the respective probabilities, yielding the following normalized amplitudes: $[0.8165, 0.4472, 0.3651]$.

In qiskit, since $Q = 3$ corresponds to a two-qubit $(2^2 = 4)$ d-register, the unused state $|11\rangle$ is extended and its amplitude is set to zero.

Listing 2 shows the code that initializes the direction register, including the normalization process. Finally, the initial quantum circuit after applying the collision operator is shown in Fig. 2. In this case, the value of $|k\rangle$ is $|k\rangle = 0.816\,|00\rangle + 0.447\,|01\rangle + 0.365\,|10\rangle$.

Listing 2: Implementation of the collision operator initializing the direction register.

```python
w = np.array([2/3, 1/6, 1/6])
e = np.array([0, 1, -1])
u = 0.2
cs = 1

def calculate_k(w, e, u, cs):
    k = w*(1+(e*u)/(cs**2))
    return k

def get_collision_amps(w, e, u, cs, n_d):
    k = calculate_k(w, e, u, cs)

    k_probabilities = np.zeros(2 ** n_d)
    k_probabilities[:len(k)] = k

    amps = np.sqrt(k_probabilities)

    return amps.tolist()

```

```python

k_amps = get_collision_amps(w, e, u, cs, n_d)

print(f'Calculated probability amplitude: {np.round(k_amps, 4)}')

amp_qc = qc.copy()
amp_qc.initialize(k_amps, d_reg)

amp_qc.draw('mpl')
```

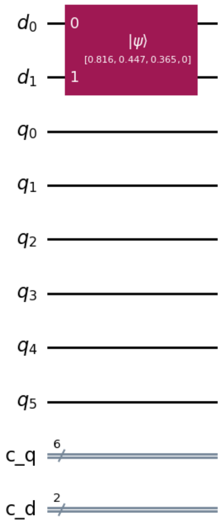

Figure 2: Collision circuit initializing the direction register with $k = [0.816, 0.447, 0.365]$.

Problem 3: Streaming

The distribution functions f_i are propagated in the directions corresponding velocities e_i in the streaming step. In the streaming step, each of the distributions f_i is shifted in the direction along link i with speed e_i. Consider the 1D case with velocity vectors 0,1,-1 for the distributions f_0,f_1, and f_2 respectively and periodic boundary conditions. The streaming operator will shift f_1 to the right, and f_2 to the left. This can be implemented on a quantum circuit applying the right and left operators R, L on the qubit register $|q\rangle$. The R operator performs the operations $R|i\rangle = |i+1\rangle$, and the L operator performs the operations $L|i\rangle = |i-1\rangle$ with periodic boundary conditions.

3-a) What is the product RL?

The operator R moves the particle one site to the right in a 1D lattice, while the operator L moves it one site to the left. We now examine the relationship between these two operators when they are applied in succession.

First, we apply the operator R to an arbitrary state $|i\rangle$. The process and the result are shown in Eq. 7:

$$R\,|i\rangle = |i+1\rangle \tag{7}$$

Next, we apply the operator L to this state. Since L performs the inverse operation of R, we obtain Eq. 8.

$$L(R\,|i\rangle) = L\,|i+1\rangle = |i\rangle \tag{8}$$

Even if we change the order of the operators and apply L first and then R, the result remains unchanged.

Therefore, the composition RL causes no change to the state $|i\rangle$. Mathematically, this means that the product of the two operators is identical to the identity operator I, as shown in Eq. 9:

$$RL = LR = I \tag{9}$$

This indicates that R and L are mutual inverses, and both operators are unitary. Consequently, they are also Hermitian conjugates of each other, as expressed in Eq. 10:

$$R^\dagger = R^{-1} = L, \qquad L^\dagger = L^{-1} = R \tag{10}$$

3-b) Using qiskit, write a function that implements the operators R, and L for n qubit states $|i\rangle$.

Each state can be expressed as a column vector of dimension 2^n, where only the element corresponding to its own index is 1 and all others are 0, as shown in Eq. 11.

$$|a\rangle = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}, \quad |b\rangle = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}, \quad |ba\rangle = |b\rangle \otimes |a\rangle = \begin{bmatrix} b_0a_0 \\ b_0a_1 \\ b_1a_0 \\ b_1a_1 \end{bmatrix}. \tag{11}$$

Let us now derive the R and L operators through an example. Here, we consider the 3-qubit state $|000\rangle$ for which $M = 8$. The state $|000\rangle$ can be represented as $|000\rangle = [10000000]^T$. Therefore, applying R gives $R\,|000\rangle = [01000000]^T$.

This process can be interpreted as selecting the next column in R, and under periodic boundary conditions, the operator R can be represented by the following matrix in Eq. 12:

$$R_8 = \begin{bmatrix} 0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \end{bmatrix}. \tag{12}$$

Using the same approach, we can construct the R operator for our environment with $M = 64$ as

$$R_{64} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix}_{64\times 64} . \tag{13}$$

Since the L operator performs the inverse action of R, we can easily derive its matrix as follows:

$$L_{64} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix}_{64\times 64} . \tag{14}$$

Since both R and L are permutation matrices (each row and column contains exactly one element equal to 1), they satisfy the following relationship:

$$RL = LR = I, \tag{15}$$

which confirms that they are inverses of each other. The code that implements R and L operators is equal to Listing 3.

Listing 3: Implementation of right-shift (R) and left-shift (L) operators acting on the position register in the streaming step.

```python
def R_matrix(n_q):
    N = 2**n_q
    R = np.zeros((N, N), dtype=complex)
    for i in range(N):
        R[(i+1) % N, i] = 1
    return R

def L_matrix(n_q):
    N = 2**n_q
    L = np.zeros((N, N), dtype=complex)
    for i in range(N):
        L[(i-1) % N, i] = 1
    return L
```

3-c) After collision, the R and L operators are applied to shift the right distribution. As a result, the operators must be applied conditioned on the direction register. Suppose the distribution functions f_0,f_1, and f_2 are encoded by the states $|00\rangle$, $|01\rangle$, and $|10\rangle$ respectively in the register $|d\rangle$. Use qiskit to write a function that applies the streaming step after the collision step.

The operators R and L obtained in Section 3-b act conditionally depending on the direction state $|d\rangle$. Thus, these operators must be applied according to the state of $|d\rangle$, which can be represented in the following block matrix form:

$$U_{\text{stream}} = \begin{bmatrix} I & 0 & 0 & 0 \\ 0 & R & 0 & 0 \\ 0 & 0 & L & 0 \\ 0 & 0 & 0 & I \end{bmatrix}. \tag{16}$$

Here, when the direction state is $|00\rangle$, the identity operator I is applied (no change). For $|01\rangle$, the right-shift operator R is applied, and for $|10\rangle$, the left-shift operator L is applied. The state $|11\rangle$ is not used and is therefore assigned to I.

This matrix can also be expressed using tensor products as shown in Eq. 17:

$$U_{\text{stream}} = |00\rangle\langle 00| \otimes I + |01\rangle\langle 01| \otimes R + |10\rangle\langle 10| \otimes L. \tag{17}$$

Each term represents a conditional operation based on the direction qubits, and in qiskit, this can be implemented using controlled-unitary gates. Listing 4 shows the implementation of the streaming operator.

Listing 4: Qiskit implementation of the conditional streaming operator U_{stream} controlled by the direction register.

```python
streaming_qc = qc.copy()

R_gate = UnitaryGate(R_matrix(n_q), label='R')
L_gate = UnitaryGate(L_matrix(n_q), label='L')

d = QuantumRegister(n_d, 'd')
q = QuantumRegister(n_q, 'q')

streaming_qc.append(R_gate.control(2, ctrl_state='01'), [d[0],
    d[1], *q])
streaming_qc.append(L_gate.control(2, ctrl_state='10'), [d[0],
    d[1], *q])

streaming_qc.draw('mpl')
```

The quantum circuit after applying U_{stream} is shown in Fig. 3. As can be seen from the circuit, the R and L operators are conditionally applied according to the values of d_0 and d_1.

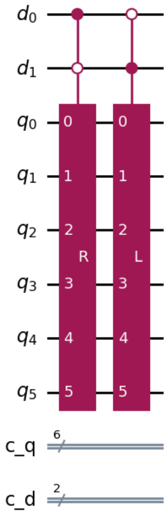

Figure 3: Streaming step implementing the R and L operators conditioned on the direction register.

Problem 4: Addition

4-a) The addition step calculates the macroscopic density according to Eq. 18. This is achieved by applying Hadamard gates on the qubits in the $|d\rangle$ register and post selection. Which state on the $|d\rangle$ register will give us the desired state $\phi(x, t = 1)$?

$$\phi(x,t) = \sum_i f_i(x,t) \tag{18}$$

In this step, Hadamard gates are applied to all qubits in the direction register $|d\rangle$, which stores the directional information of the particle distribution. Since $|d\rangle$ consists of two qubits, the tensor product of Hadamard gates is used, $H^{\otimes 2} = H \otimes H$. The two-qubit Hadamard matrix is derived as follows:

$$H^{\otimes 2} = H \otimes H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}. \tag{19}$$

Let us now apply this operator to an arbitrary two-qubit state represented as

$$|\phi\rangle = c_{00}\,|00\rangle + c_{01}\,|01\rangle + c_{10}\,|10\rangle + c_{11}\,|11\rangle\,. \tag{20}$$

In matrix form, the state can be written as

$$|\phi\rangle = \begin{bmatrix} c_{00} \\ c_{01} \\ c_{10} \\ c_{11} \end{bmatrix}. \tag{21}$$

Applying $H^{\otimes 2}$ to $|\phi\rangle$ gives the following transformed amplitudes:

$$\tilde{c}_{00} = \frac{1}{2}(c_{00} + c_{01} + c_{10} + c_{11}) \tag{22}$$

$$\tilde{c}_{01} = \frac{1}{2}(c_{00} - c_{01} + c_{10} - c_{11}) \tag{23}$$

$$\tilde{c}_{10} = \frac{1}{2}(c_{00} + c_{01} - c_{10} - c_{11}) \tag{24}$$

$$\tilde{c}_{11} = \frac{1}{2}(c_{00} - c_{01} - c_{10} + c_{11}) \tag{25}$$

Among these four components, $\tilde{c}_{00}$ represents the combination where all coefficients have positive signs, corresponding to the summation of all directional components. According to Eq. 18, the macroscopic density $\phi(x,t)$ is obtained by summing over all microscopic distributions. Therefore, selecting the state $|d\rangle = |00\rangle$ after applying $H^{\otimes 2}$ corresponds to the desired macroscopic density state $\phi(x,1)$.

Next, we integrate all the gates and circuits implemented above to perform the full simulation. The merged code is shown in Listing 5, the full appearance of the integrated circuit is shown in Fig. 4, For readability, each stage of the process is separated by barriers.

Listing 5: Construction of the complete quantum circuit integrating initialization, collision, streaming, addition, and measurement steps.

```
main_qc = concentration_qc.copy()
main_qc.barrier()

main_qc.compose(amp_qc, inplace=True)
main_qc.barrier()

main_qc.compose(streaming_qc, inplace=True)
main_qc.barrier()

main_qc.h(d_reg)
main_qc.barrier()

main_qc.measure(q_reg, c_q)
main_qc.measure(d_reg, c_d)

main_qc.draw('mpl')
```

After applying the Hadamard gates to the d-register, all qubits are measured. Based on the measurement results, only the states of the q-register corresponding to the d-register being in the $|00\rangle$ state are filtered to compute the distribution function for the next time step.

The code used to simulate the circuit is shown in Listing 6, and it returns both the distribution function and the probability amplitudes for the next time step.

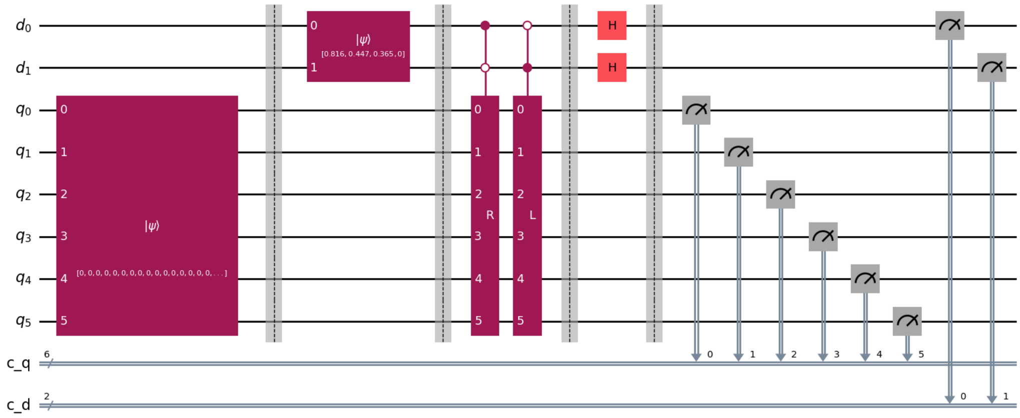

Figure 4: Integrated quantum circuit combining initialization, collision, streaming, addition, and measurement steps.

Listing 6: Simulation routine that measures the circuit, performs post-selection on $|d\rangle = |00\rangle$, and returns $\phi(x, t+1)$.

```python
def simulate_circuit(qc, sim, n_shots):
    compiled_qc = transpile(qc, sim)
    result = sim.run(compiled_qc, shots=n_shots).result()
    counts = result.get_counts()

    filtered_counts = Counter()
    for key, val in counts.items():
        try:
            d_state, q_state = key.split(' ')
        except ValueError:
            continue

        if d_state == '00':
            filtered_counts[q_state] += val

    phi_next_step = np.zeros(2**n_q)
    total_filtered_counts = sum(filtered_counts.values())

    if total_filtered_counts > 0:
        for state_str, count in filtered_counts.items():
            idx = int(state_str, 2)
            phi_next_step[idx] = count
        phi_next_step /= total_filtered_counts

    phi_next_amplitudes = np.sqrt(phi_next_step)

    return phi_next_step, phi_next_amplitudes
```

Although the number of measurements was not explicitly specified in the problem statement, it was assumed that the number of usable results would decrease exponentially as the simulation progressed, since only the $|d\rangle = |00\rangle$ state was filtered in each time step. Therefore, to ensure sufficient statistical reliability, a total of 50,000 measurement shots were performed.

Since the measurement process was implemented as a function, the results can be obtained through a simplified execution code as shown in Listing 7.

Listing 7: Simplified execution script to obtain the first time-step distribution $\phi(x, 1)$ using the measurement function.

```
sim = AerSimulator()

next_phi, phi_amp = simulate_circuit(main_qc, sim, n_shots)

print(f'phi(x, 1) :\n{np.round(next_phi, 4)}')
```

The execution result of this code is shown below. It can be observed that the fluid begins to flow to both sides from the initial position $N = 32$. The obtained values converge to the probabilities k that were computed earlier.

Output 1: $\phi(x, 1)$ distribution

```
phi(x, 1) :
[0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
 0.     0.1321 0.6703 0.1975 0.     0.     0.     0.     0.     0.
 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
 0.     0.     0.     0.    ]
```

Problem 5: Simulation

5-a) Using $\phi(x, t = 1)$ from the addition step above as the input state, repeat the steps above to get $\phi(x, t = 2)$

In this quantum circuit system, the qubits q_0–q_5 must be initialized with the distribution function obtained from the previous time step. By including this step, the overall circuit integration process can be generalized into a single function. The implementation of this function is shown in Listing 8.

Listing 8: Generalized function that constructs the full circuit for any given time step, initializing the q-register with the previous distribution.

```
def gen_circuit(phi_amp):
    main_qc = qc.copy()

    main_qc.initialize(phi_amp, q_reg)
    main_qc.barrier()

    main_qc.compose(amp_qc, inplace=True)
    main_qc.barrier()

    main_qc.compose(streaming_qc, inplace=True)
    main_qc.barrier()
```

```python

    main_qc.h(d_reg)
    main_qc.barrier()

    main_qc.measure(q_reg, c_q)
    main_qc.measure(d_reg, c_d)

    return main_qc
```

The code used to compute the distribution function for the second time step is shown in Listing 9.

Listing 9: Execution script for calculating the second time-step distribution $\phi(x, 2)$.

```python
main_qc = gen_circuit(phi_amp)
next_phi, phi_amp = simulate_circuit(main_qc, sim, n_shots)

print(f'phi(x, 2) :\n{np.round(next_phi, 4)}')
```

The output result is shown below. It clearly indicates that the distribution function becomes more dispersed than in the previous timestep.

Output 2: $\phi(x, 2)$ distribution

```
phi(x, 2) :
[0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
 0.002  0.1824 0.5307 0.2643 0.0205 0.     0.     0.     0.     0.
 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
 0.     0.     0.     0.    ]
```

The result corresponds to the value obtained by multiplying each position of $\phi(x, 1)$ by the corresponding transition probability k and then summing the contributions.

As an example, the value at $x = 33$ can be derived by summing the contributions from the neighboring positions. Specifically, the value at $x = 32$ (0.667) contributes through the rightward transition with probability $k = 0.200$, while the value at $x = 33$ (0.200) contributes through the resting probability $k = 0.667$. The resulting theoretical value, 0.267, closely matches the simulated result.

5-b) Repeat the steps above to simulate fluid flow for 20 stpes.

Using the generalized function defined above, the simulation can be performed up to any desired time step $t = N$. The code for performing the simulation up to $t = 20$ is shown in Listing 10, where the distribution results at each time step are stored in the variable `phi_history`.

Listing 10: Iterative simulation loop performing 20 time steps, storing each distribution in `phi_history`.

```python
phi_history = []
phi_history.append(initial_phi)

phi_amp = initial_phi.copy()

for i in range(20):
    main_qc = gen_circuit(phi_amp)
    next_phi, phi_amp = simulate_circuit(main_qc, sim, n_shots)
    print(f'Timestep {i+1} finished')

    phi_history.append(next_phi)
```

5-c) Plot the states $\phi(x,0)$, $\phi(x,1)$, $\phi(x,2)$, and $\phi(x,20)$.

Based on the stored distribution functions at each time step, the fluid motion can be visualized. The code used for visualizing the previously stored data is shown in Listing 11.

Listing 11: Visualization code plotting $\phi(x,t)$ at $t = 0, 1, 2, 20$.

```python
plt.figure(figsize=(14, 6))
x_axis = np.arange(2**n_q)

for i in range(len(phi_history)):
    if i in [0, 1, 2, 20]:
        plt.plot(x_axis, phi_history[i], '-', label=f'$\phi(x, t={i})$')

plt.xticks(np.arange(0, 64, 2))
plt.title(f'QLBM Simulation Results')
plt.xlabel('Lattice Position (x)')
plt.ylabel('Probability Density $\phi(x, t)$')
plt.legend()
plt.grid(True)

plt.show()
```

The visualization was performed only for $t = 0, 1, 2, 20$, and the resulting plot is shown in Fig. 5. It can be clearly observed that the fluid gradually diffuses over time, exhibiting a profile similar to a Gaussian distribution.

Through this process, a one-dimensional fluid simulation using the QLBM framework was successfully implemented. Increasing the number of measurement shots can further improve accuracy, and by experimenting with various initial conditions and parameters, different fluid characteristics can be explored.

Furthermore, this result suggests that the same approach can be extended to two-dimensional or three-dimensional fluid simulations.

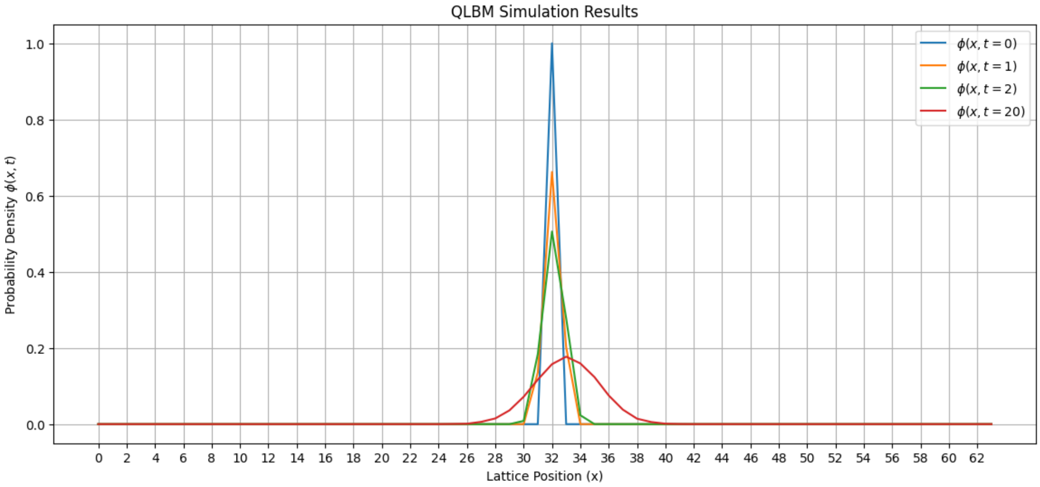

Figure 5: Visualization of the fluid distribution at $t = 0, 1, 2, 20$, showing Gaussian-like diffusion behavior.

References

[1] Apurva Tiwari, Jason Iaconis, Jezer Jojo, Sayonee Ray, Martin Roetteler, Chris Hill, and Jay Pathak. Algorithmic advances towards a realizable quantum lattice boltzmann method. *arXiv preprint arXiv:2504.10870*, 2025.