Documentation of Dictionary-based Sparse Block Encoding

Dictionary-based Sparse BLock Encoding (hereafter abbreviated as DSBLE) is a sparse block encoding protocol that relies on a novel dictionary data structure of sparse structure matrices and achieves low subnormalization and circuit depth [1]. The low-circuit-depth implementation requires extensive ancillary qubit resources, making practical quantum circuit simulations currently infeasible. Therefore, we provide Python code to verify the subnormalization properties of DSBLE, which is available in https://github.com/ChunlinYangHEU/DSBLE. The quantum circuit of DSBLE is built on top of MindQuanutm in Python.

DSBLE relies on the dictionary data structure of sparse matrices shown in Table 1.

Keys	Values
0	$\{(a_{ij}, i, j) : a_{ij} = A_0, (i,j) \in (c_0(j), S_c(0))\}$
1	$\{(a_{ij}, i, j) : a_{ij} = A_1, (i,j) \in (c_1(j), S_c(1))\}$
2	$\{(a_{ij}, i, j) : a_{ij} = A_2, (i,j) \in (c_2(j), S_c(2))\}$
$\vdots$	$\vdots$
s_0	$\{(a_{ij}, i, j) : a_{ij} = A_{s_0}, (i,j) \in (c_{s_0}(j), S_c(s_0))\}$

Table 1: Dictionary data structure of a sparse matrix.

Based on the dictionary, DSBLE can be implemented by using the quantum circuit shown in Figure 1.

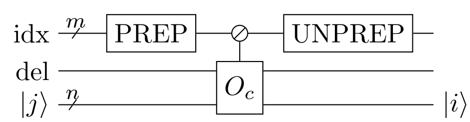


Figure 1: Basic framework of dictionary-based sparse block encoding.

1 Core Objectives

The main two tasks of DSBLE are presented in the following.

- **Task 1: State Preparation.** Implement quantum circuit for the oracles PREP and UNPREP, where

$$\mathrm{PREP}\, |0\rangle_{\mathrm{idx}}^{\otimes m} = \frac{1}{\sqrt{\sum_{l=0}^{s_0-1} |A_l|}} \left(\sum_{l=0}^{s_0-1} \sqrt{A_l}\, |l\rangle_{\mathrm{idx}} + \sum_{l=s_0}^{2^m-1} 0\, |l\rangle_{\mathrm{idx}} \right),$$

$$\mathrm{UNPREP}^{\dagger}\, |0\rangle_{\mathrm{idx}}^{\otimes m} = \frac{1}{\sqrt{\sum_{l=0}^{s_0-1} |A_l|}} \left(\sum_{l=0}^{s_0-1} \sqrt{A_l}^{*}\, |l\rangle_{\mathrm{idx}} + \sum_{l=s_0}^{2^m-1} 0\, |l\rangle_{\mathrm{idx}} \right).$$

- **Task 2: Index Mapping.** Implement quantum circuit for the oracle O_c, where

$$O_c\, |l\rangle_{\mathrm{idx}} |0\rangle_{\mathrm{del}} |j\rangle = \begin{cases} |l\rangle_{\mathrm{idx}} |0\rangle_{\mathrm{del}} |c_l(j)\rangle, & \text{if } l \in [0, s_0-1] \text{ and } j \in S_c(l), \\ |l\rangle_{\mathrm{idx}} |1\rangle_{\mathrm{del}} |j\rangle, & \text{if } l \in [s_0, 2^m-1] \text{ or } j \notin S_c(l). \end{cases}$$

We assume that the data functions $i = c_l(j)$ have the expression

$$i = c_l(j) = j \pm k_l, \tag{1}$$

where $j \in S_c(l)$, k_l is a non-negative integer, $l \in [0, s_0-1]$.

1.1 Task 1: State Preparation

An n-fold controlled rotation $R_{\boldsymbol{\alpha}}(\boldsymbol{\theta}_{[0,2^n-1]})$ is a multiplexor operation composed of 1-qubit unitaries, defined as

$$R_{\boldsymbol{\alpha}}(\boldsymbol{\theta}_{[0,2^n-1]}) = \sum_{l=0}^{2^n-1} |l\rangle \langle l| \otimes R_{\boldsymbol{\alpha}}(\theta_l) = \begin{pmatrix} R_{\boldsymbol{\alpha}}(\theta_0) & & & \\ & R_{\boldsymbol{\alpha}}(\theta_1) & & \\ & & \ddots & \\ & & & R_{\boldsymbol{\alpha}}(\theta_{2^n-1}) \end{pmatrix},$$

where $R_{\boldsymbol{\alpha}}(\theta_l) \in \mathbb{C}^{2\times 2}$ denotes a rotation by angle $\frac{\theta}{2}$ about the axis $\boldsymbol{\alpha}\cdot\boldsymbol{\sigma} = a_x X + a_y Y + a_z Z$ [2] as

$$R_{\boldsymbol{\alpha}}(\theta) = e^{i\boldsymbol{\alpha}\cdot\boldsymbol{\sigma}\theta/2} = I\cos\frac{\theta}{2} + i\boldsymbol{\alpha}\cdot\boldsymbol{\sigma}\sin\frac{\theta}{2},$$

with $(a_x, a_y, a_z) \in \mathbb{R}^3$. A 2-fold controlled rotation is depicted in Figure 2.

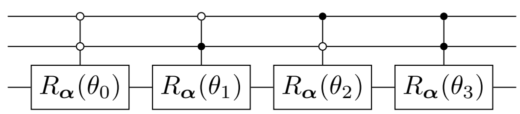


Figure 2: 2-fold controlled rotation $R_{\boldsymbol{\alpha}}(\boldsymbol{\theta}_{[0,3]})$, where $\boldsymbol{\theta}_{[0,3]} = (\theta_0, \theta_1, \theta_2, \theta_3)^{\mathrm{T}}$.

An n-fold controlled rotation $R_{\boldsymbol{\alpha}}(\boldsymbol{\theta}_{[0,2^n-1]})$ can be decomposed into 2^n single-qubit rotations $R_{\boldsymbol{\alpha}}(\tilde{\theta})$ and 2^n C-NOT gates [2]. The rotation angles $\tilde{\boldsymbol{\theta}} = \left(\tilde{\theta}_0, \cdots, \tilde{\theta}_{2^n-1}\right)^{\mathrm{T}}$ and $\boldsymbol{\theta} = (\theta_0, \cdots, \theta_{2^n-1})^{\mathrm{T}}$ satisfy the linear system

$$M\tilde{\boldsymbol{\theta}} = \boldsymbol{\theta},$$

where $M_{ij} = (-1)^{b_{i-1}\cdot g_{j-1}}$, b_i, g_i are the standard binary code representation of the integer i and the i-th Binary reflected 2-bit Gray code, and the dot in the exponent denotes the bitwise inner product of the binary vectors [2–4]. The uniformly controlled rotation decomposition of Figure 2 is shown in Figure 3.

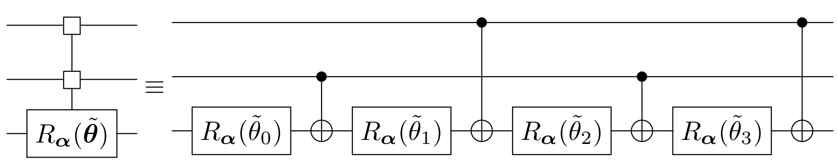


Figure 3: The uniformly controlled decomposition of 2-fold controlled rotation $R_{\boldsymbol{\alpha}}(\boldsymbol{\theta}_{[0,3]})$.

For an arbitrary m-qubit quantum state

$$|\psi\rangle = \frac{1}{\sqrt{\sum_{i=0}^{2^m-1} |a_i|^2}} \sum_{i=0}^{2^m-1} a_i |i\rangle,$$

with $a_i \in \mathbb{C}$, it can be prepared without introducing additional ancillary qubits using the quantum circuit in Figure 4 [4]. It contains only single-qubit R_Y, R_Z rotation gates and C-NOT gates. If the amplitudes are real, only R_Y rotation gates and C-NOT gates are enough.

The rotation angles of R_Y and R_Z rotations in Figure 4 can be computed by following two processes [4]:

1. Calculate the rotation angles $\boldsymbol{\varphi}^{(i)}$ and $\boldsymbol{\theta}^{(i)}$ of multiplexor rotations from classcial data by rotation-Y and rotation-Z binary trees.

2. Calculate the rotation angles $\tilde{\boldsymbol{\varphi}}^{(i)}$ and $\tilde{\boldsymbol{\theta}}^{(i)}$ of single-qubit controlled rotations by the permutative demultiplexor.

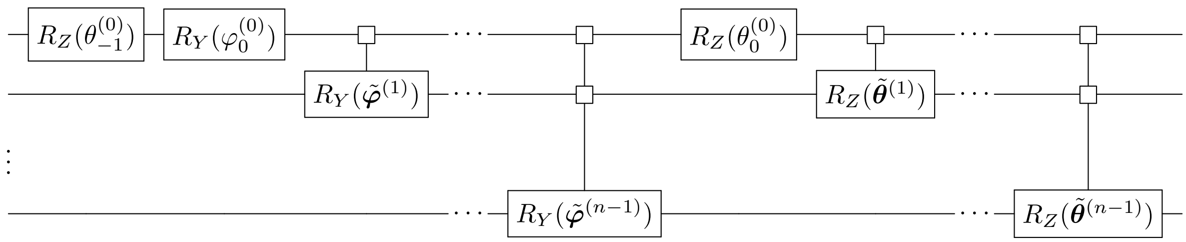


Figure 4: Quantum circuit for the state preparation of a state $|\psi\rangle = \frac{1}{\sqrt{\sum_{i=0}^{2^n-1}|a_i|^2}} \sum_{i=0}^{2^n-1} a_i |i\rangle$ with $a_i \in \mathbb{C}$. The rotation angles $\tilde{\boldsymbol{\varphi}}^{(i)} = \left(\tilde{\varphi}_0^{(i)}, \cdots, \tilde{\varphi}_{2^i-1}^{(i)}\right)^{\mathrm{T}}$ and $\tilde{\boldsymbol{\theta}}^{(i)} = \left(\tilde{\theta}_0^{(i)}, \cdots, \tilde{\theta}_{2^i-1}^{(i)}\right)^{\mathrm{T}}$, where $i \in [n-1]$.

1.2 Task 2: Index mapping

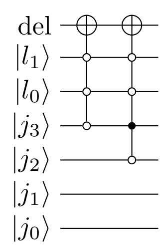
To implement the data functions in Equation (1), two subtasks must be completed.

- **Subtask 1:** Determine the defining domains $S_c(l)$;
- **Subtask 2:** Implement the mappings $j \pm k_l$.

1.2.1 Subtask 1: Determine the defining domains

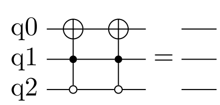
For each $l \in [0, s_0 - 1]$, we use multi-qubit controlled NOT (hereafter abbreviated as MC-NOT) gates to flag all in-range column indices $j \in S_c(l)$. Specifically, the MC-NOT gate is controlled by the registers idx and $|j\rangle$ and act on the qubit del. In the register idx, the control qubits are all qubits and the control states are the binary bit string of l with length m. In the register $|j\rangle$, the control qubits and control states are determined by $S_c(l)$.

For example, if $m = 2$, $l = 0$, $n = 4$ and $S_c(0) = [0, 11]$, then the following circuit flags the defining domain $S_c(0)$, where the first MC-NOT gate flags column indices $j \in [0, 7]$ and the second MC-NOT gate flags column indices $j \in [8, 11]$.

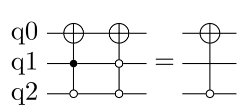


Next, we give several simplifications of the MC-NOT gates.

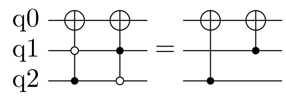
- If two MC-NOT gates have the same target qubit, control qubits, and control states, then the two MC-NOT gates cancel out. An example is presented below.



- If two MC-NOT gates have the same target qubit and control qubits, but their control states differ by one qubit, then they are equal to a MC-NOT gate eliminating the control of that qubit. An example is presented below.



- If two MC-NOT gates have the same target qubit and control qubits, but their control states differ by two qubits q_{l_1} and q_{l_2}, where $(q_{l_1}, q_{l_2}) = (0, 1)$ for one MC-NOT gate and $(q_{l_1}, q_{l_2}) = (1, 0)$ for the other MC-NOT gate, then one MC-NOT gate can eliminate the control of qubit q_{l_1} and the other MC-NOT gate can eliminate the control of qubit q_{l_2}. An example is presented below.



1.2.2 Subtask 2: Implement the mappings

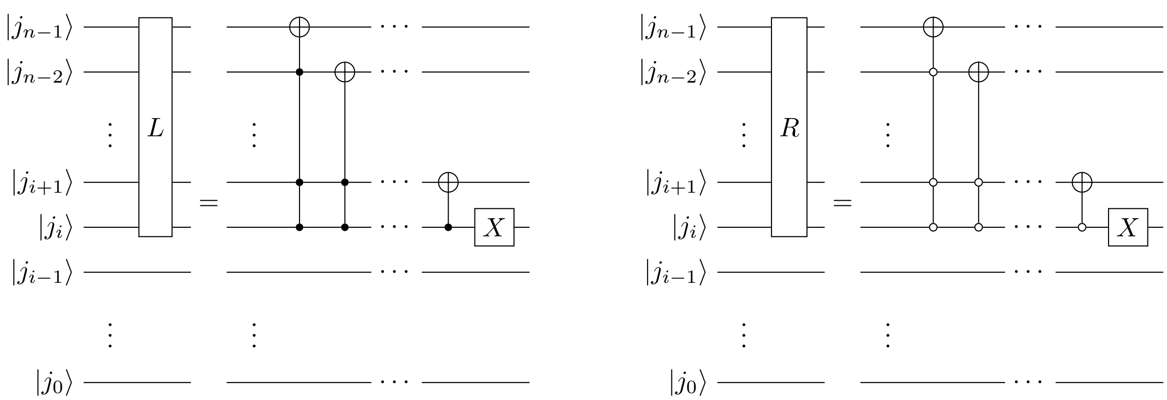
For the non-negative integers k_l in Equation (1), they have the binary representation

$$k_l = \sum_{i=0}^{n-1} k_l^{(i)} 2^i,$$

where $k_l^{(i)} \in \{0, 1\}$. So, the data functions (1) are also expressed as

$$i = j \pm \sum_{k_l^{(i)}=1} 2^i.$$

For each $+2^i$ and -2^i, they can be implemented using an L^i-shift and R^i-shift gates [5], respectively, which are shown in Figure 5.



(a) The L^i-shift gate.

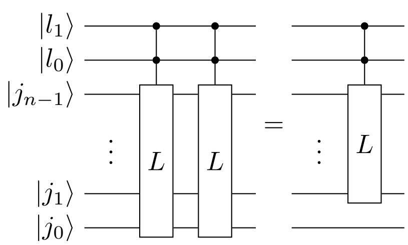
(b) The R^i-shift gate.

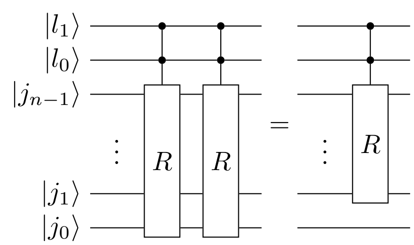
Figure 5: Quantum circuits of the L^i-shift and R^i-shift gates.

Therefore, the data function $i = j + k_l$ can be implemented using a sequence of L^i-shift gates, where $k_l^{(i)} = 1$, $i \in [0, n-1]$. And the data function $i = j - k_l$ can be implemented using a sequence of R^i-shift gates, where $k_l^{(i)} = 1$, $i \in [0, n-1]$.

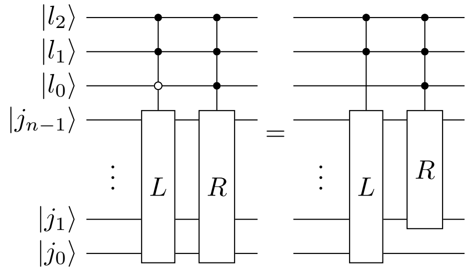
Next, we give several simplifications of the shift gates.

- If two L^i-shift (or R^i-shift) gates have the same target qubits, control qubits and control states, then they are equal to a L^{i+1}-shift (or R^{i+1}-shift) gate. Two examples are presented below.

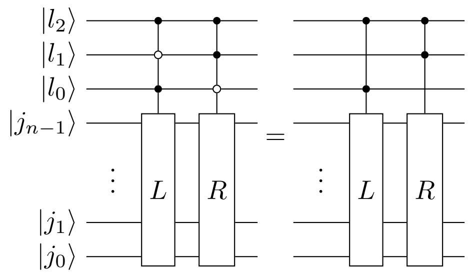




- If a L^i-shift gate and a R^i-shift gate have the same target qubits and control qubits, but their control states differ by only one qubit, then they are equal to a L^i-shift gate (or R^i-shift) eliminating the control of that qubit and a R^{i+1}-shift (or L^{i+1}-shift) gate. An example is presented below.



- If a L^i-shift gate and a R^i-shift gate have the same target qubits and control qubits, but their control states differ by two qubits q_{l_1} and q_{l_2}, where $(q_{l_1}, q_{l_2}) = (0, 1)$ for the L^i-shift gate and $(q_{l_1}, q_{l_2}) = (1, 0)$ for the R^i-shift gate, then the L^i-shift gate can eliminate the control of qubit q_{l_1} and the R^i-shift gate can eliminate the control of qubit q_{l_2}. An example is presented below.

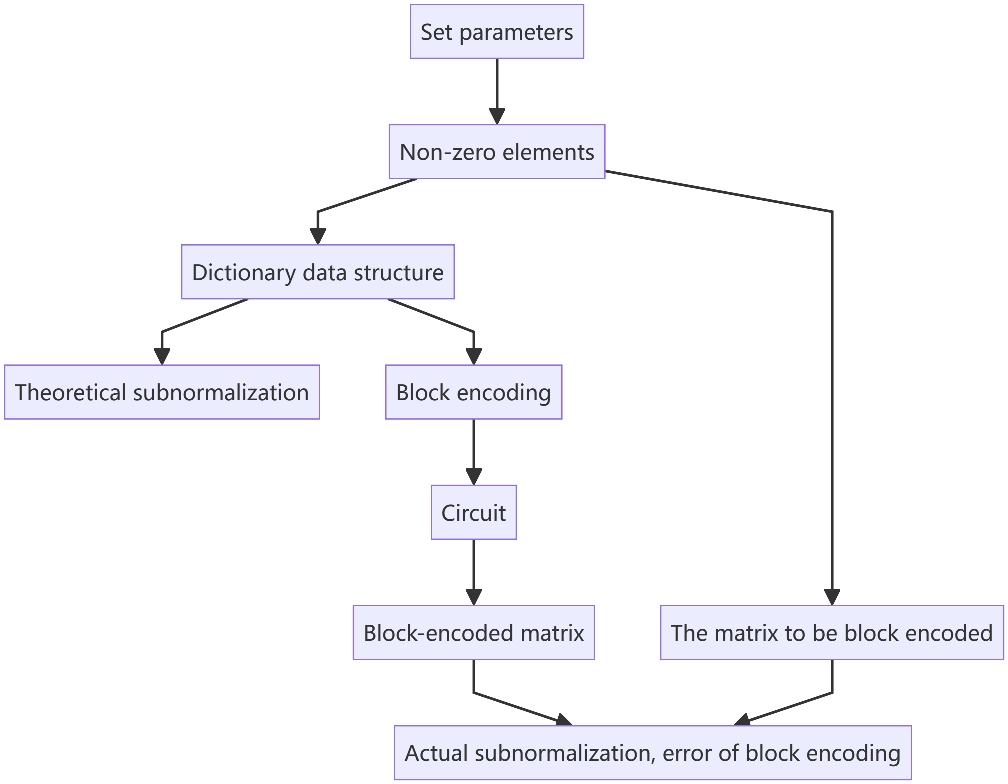


2 Technology Stack

- **Programming Language:** Python 3.7-3.9
- **Quantum Framework:** MindQuantum
- **Math Library:** Numpy, Pandas
- **Core Modules:**
  - `anglecompute.py`
    This file is used to compute the rotation angles of R_Y and R_Z rotation gates for state preparation in sec. 1.1. It follows the codes in https://github.com/ChunlinYangHEU/BITBLE_python.
  - `blockencoding.py`
    This file is used to:
    (1) construct the quantum circuit of DSBLE;
    (2) construct the quantum circuits of oracles O_c and PREP;
    (3) obtain the block-encoded matrix of the block-encoding circuit.
  - `qgates.py`
    This file is used to:
    (1) implement several (controlled) quantum gates used in DSBLE, including X, Y, Z, H, SWAP, R_X, R_Y, R_Z, compressed uniformly rotation, left-shift and right-shift gates;
    (2) simplify the multi-qubit controlled X gates;
    (3) simplify the left-shift and right-shift gates.
  - `tools.py`
    This file provides several helpful functions used in constructing the quantum circuit of DSBLE.
  - `test_*.py`
    These files provides the test cases for the signless Laplacian matrix, two-dimensional discrete Laplacian and matrices in ocean acoustic generalized eigenvalue problems.

3 A Test Case of Two-dimensional Discrete Laplacian

3.1 Functional Architecture



3.2 Full Code

Code:

```
import numpy as np
import pandas as pd
from dsble import blockencoding, tools
from mindquantum import *


def get_non_zero_elements(parameters):
    """
    Computes all distinct non-zero elements of the two-dimensional discrete Laplacian.

    Args:
        parameters (dict): A dictionary containing the discretization parameters.
            - 'delta_x': The discretization step size in the x-direction.
            - 'delta_y': The discretization step size in the y-direction.

    Returns:
        list: A list of distinct non-zero elements [A0, A1, A2].
    """
    delta_x = parameters['delta_x']
    delta_y = parameters['delta_y']
    A0 = - 2 * (1 / delta_x ** 2 + 1 / delta_y ** 2)
    A1 = 1 / delta_x ** 2
    A2 = 1 / delta_y ** 2
```

```python
    non_zero_elements = [A0, A1, A2]

    return non_zero_elements


# Get the two_dimensional discrete Laplacian
def two_dimensional_discrete_laplacian(non_zero_elements, nx, ny):
    """
    Constructs the two-dimensional discrete Laplacian.

    Args:
        non_zero_elements (list): A list of three non-zero elements [A0, A1, A2].
            - A0: The diagonal element.
            - A1: The horizontal off-diagonal element.
            - A2: The vertical off-diagonal element.
        nx (int): The number of grid points in the x-direction.
        ny (int): The number of grid points in the y-direction.

    Returns:
        numpy.array: The two-dimensional discrete Laplacian.
    """
    matrix = np.zeros((nx * ny, nx * ny))
    for i1 in range(nx):
        for i2 in range(nx):
            for j1 in range(ny):
                for j2 in range(ny):
                    if i1 == j1 and i2 == j2:
                        matrix[i1 + i2 * nx, j1 + j2 * ny] = non_zero_elements[0]
                    elif abs(i1 - j1) == 1 and i2 == j2:
                        matrix[i1 + i2 * nx, j1 + j2 * ny] = non_zero_elements[1]
                    elif abs(i2 - j2) == 1 and i1 == j1:
                        matrix[i1 + i2 * nx, j1 + j2 * ny] = non_zero_elements[2]

    return matrix


def get_data_item(non_zero_elements, nx, ny):
    """
    Constructs the dictionary for the two-dimensional discrete Laplacian.

    Args:
        non_zero_elements (list): A list of distinct non-zero elements [A0, A1, A2] in
            the two-dimensional discrete Laplacian.
            - A0: The diagonal element.
            - A1: The horizontal off-diagonal element.
            - A2: The vertical off-diagonal element.
        dim (int): The dimension of the two-dimensional discrete Laplacian, which must
            be a power of two.

    Returns:
        dict: The dictionary consisting of data items.
    """
    dim = nx * ny
    n = int(np.log2(dim))
    data_item = {
        0: [non_zero_elements[0],
            0,
            tools.binary_range(0, dim - 1, n, True, right_close=True)],
        1: [non_zero_elements[1],
            -1,
            tools.binary_range(0, dim - 1, n, True, True, 1)
            + tools.binary_range(0, dim - 1, n, True, True, 2)
```

```
            + tools.binary_range(0, dim - 1, n, True, True, 3)],
        2: [non_zero_elements[1],
            1,
            tools.binary_range(0, dim - 1, n, True, True, 0)
            + tools.binary_range(0, dim - 1, n, True, True, 1)
            + tools.binary_range(0, dim - 1, n, True, True, 2)],
        3: [non_zero_elements[2],
            -4,
            tools.binary_range(nx, dim - 1, n, True, True)],
        4: [non_zero_elements[2],
            4,
            tools.binary_range(0, dim - 1 - nx, n, True, True)]
    }

    return data_item


def test_two_dimensional_discrete_laplacian(data_item, dim):
    """
    Tests the construction of block encoding the two-dimensional discrete Laplacian.

    Args:
        data_item (dict): A dictionary representing the data item to be encoded.
            It should contain the coefficients and their corresponding binary ranges.
        dim (int): The dimension of the Laplacian matrix, which must be a power of two.

    Returns:
        tuple: A tuple containing the constructed quantum circuit and the encoded
           Laplacian matrix.
            - circuit: The quantum circuit of block encoding.
            - encoded_matrix: The encoded Laplacian matrix.
    """
    # The number of qubits of register idx
    num_idx_qubits = tools.num_qubits(len(data_item))

    # The number of working qubits
    num_working_qubits = tools.num_qubits(dim)

    # The number of qubits of circuit
    num_qubits = num_idx_qubits + 1 + num_working_qubits

    # Sparse block encoding
    circuit = blockencoding.qcircuit(data_item=data_item,
                                     num_working_qubits=num_working_qubits)

    # Get the encoded matrix
    encoded_matrix = blockencoding.get_encoded_matrix(circuit, num_qubits,
        num_working_qubits)

    return circuit, encoded_matrix


if __name__ == '__main__':
    delta_x = 1
    delta_y = 2
    nx = 4
    ny = 4
    parameters = {'delta_x': delta_x, 'delta_y': delta_y}

    # Dimension of matrix
    dim = nx * ny
```

```
    # Get all distinct non-zero elements
    non_zero_elements = get_non_zero_elements(parameters)

    # Get data items
    data_items = get_data_item(non_zero_elements, nx, ny)

    # Get the two-dimensional discrete Laplacian to be encoded
    matrix = two_dimensional_discrete_laplacian(non_zero_elements, nx, ny)

    # Get the circuit and the encoded two-dimensional discrete Laplacian
    circuit, encoded_matrix = test_two_dimensional_discrete_laplacian(data_items, dim)

    # Compute the subnormalization
    subnormalization = abs(non_zero_elements[0]) + 2 * (abs(non_zero_elements[1]) + abs(
        non_zero_elements[2]))

    print(circuit)
    matrix_pd = pd.DataFrame(matrix)
    matrix_pd.to_excel('Laplacian_nx_' + str(nx) + '_ny_' + str(ny) + '.xlsx')
    encoded_matrix_pd = pd.DataFrame(encoded_matrix)
    encoded_matrix_pd.to_excel('Laplacian_nx_' + str(nx) + '_ny_' + str(ny) + '_encoded.
        xlsx')
    print('Actual subnormalization:')
    print(np.linalg.norm(matrix) / np.linalg.norm(encoded_matrix))
    print('The subnormalization:')
    print(subnormalization)
    error = np.linalg.norm(matrix - subnormalization * encoded_matrix)
    print('The error of block encoding:')
    print(error)
```

Output:

```
Actual subnormalization:
5.000000000000002
The subnormalization:
5.0
The error of block encoding:
5.2408703594204154e-15
```

Besides, the code outputs the block-encoding circuit and generates two `.xlsx` files, including the matrix to be block encoded and the block-encoded matrix.

References

[1] Chunlin Yang, Zexian Li, Hongmei Yao, Zhaobing Fan, Guofeng Zhang, and Jianshe Liu. Dictionary-based sparse block encoding with low subnormalization and circuit depth, 2025. URL https://arxiv.org/abs/2405.18007.

[2] Mikko Möttönen, Juha J Vartiainen, Ville Bergholm, and Martti M Salomaa. Quantum circuits for general multiqubit gates. *Physical review letters*, 93(13):130502, 2004. doi: 10.1103/PhysRevLett.93.130502.

[3] Daan Camps and Roel Van Beeumen. Fable: Fast approximate quantum circuits for block-encodings. *2022 IEEE International Conference on Quantum Computing and Engineering*, pages 104–113, 2022. doi: 10.1109/QCE53715.2022.00029.

[4] Zexian Li, Xiao-Ming Zhang, Chunlin Yang, and Guofeng Zhang. Binary tree block encoding of classical matrix, 2025. URL https://arxiv.org/abs/2504.05624.

[5] Daan Camps, Lin Lin, Roel Van Beeumen, and Chao Yang. Explicit quantum circuits for block encodings of certain sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 45(1):801–827, 2024. doi: 10.1137/22M1484298.