

---

# Quantum Computing Project Code Documentation

*Release 1.0*

Group 5

Mar 20, 2024



**CONTENTS:**

|          |                                     |           |
|----------|-------------------------------------|-----------|
| <b>1</b> | <b>Quantum Circuit Module</b>       | <b>1</b>  |
| <b>2</b> | <b>Tensor Module</b>                | <b>3</b>  |
| <b>3</b> | <b>Gates Module</b>                 | <b>7</b>  |
| <b>4</b> | <b>Tensor Module Test Suite</b>     | <b>9</b>  |
| <b>5</b> | <b>Matrix Interface</b>             | <b>11</b> |
| <b>6</b> | <b>Sparse Matrix Implementation</b> | <b>13</b> |
| <b>7</b> | <b>Dense Matrix Implementation</b>  | <b>15</b> |
|          | <b>Python Module Index</b>          | <b>17</b> |
|          | <b>Index</b>                        | <b>19</b> |



---

## QUANTUM CIRCUIT MODULE

This module provides the implementation of a quantum circuit simulator. It defines the *Circuit* class which represents a quantum circuit and provides methods for applying quantum gates and running Grover's algorithm.

**class** `qc.Circuit`(*register\_size: int*)

This is the quantum circuit class which represents a quantum circuit with a specified register size. It provides methods for applying quantum gates, running Grover's algorithm, and measuring the result.

**grover**(*target: int, plot=False*)

Runs Grover's algorithm on the quantum circuit to find the target state.

This method calculates the number of iterations required for Grover's algorithm based on the register size and the target state. It then constructs the Grover operator by combining the oracle, Hadamard gates, and reflection operator.

The Grover operator is applied to the quantum register for the calculated number of iterations to amplify the amplitude of the target state.

**Params:**

`target (int)`: The target state to find using Grover's algorithm.

**h()**

Applies the Hadamard gate to all qubits in the quantum register.

This method retrieves the Hadamard gate matrix from the *gates* object, raises it to the power of the register size to create a tensor product, and then applies the resulting operator to the quantum register.

The quantum register is updated with the new state after applying the Hadamard gate.

**measure**(*target: int*)

Measures the quantum circuit and prints the probability of the target state.

This method performs a measurement on the quantum circuit to determine the probability of observing the target state. It calculates the inner product between the target state vector and the current state vector of the quantum register.

**Params:**

`target (int)`: The target state to measure the probability for.

---

`utils.state_vector.makeStateVector`(*value: int, size: int = 0*)

Creates a state vector based on the given integer value.

The function takes an integer *value* and an optional *size* parameter. It converts the integer to its binary representation and creates a tensor product of individual qubit state vectors corresponding to each binary digit.

If the *size* parameter is not provided, the size of the state vector is determined by the length of the binary representation of the *value*.



## TENSOR MODULE

This module provides classes for working with operators and vectors in a quantum computing context. It includes the *Operator* class for representing quantum operators and the *Vector* class for representing quantum state vectors.

The *Operator* class supports operations tensor products, addition, subtraction, exponentiation, and matrix multiplication. It also provides methods for updating matrix elements, negating the matrix, and scaling the matrix by a value.

The *Vector* class supports operations tensor products, addition, subtraction, scalar multiplication, and exponentiation. It also provides methods for applying an operator to a vector and checking equality between vectors.

This is all achieved abstractly through the implementation of a *Matrix* property of each object. See *matrixInterface* for more details on available methods.

**class** `utils.tensor.Operator`(*size: int, elements: list | matrixInterface*)

This class represents a quantum operator as a sparse matrix.

It may take a list or *matrixInterface* in the constructor.

**equal**(*target*)

Checks if the operator is equal to another operator.

**Args:**

target (Operator): The operator to compare with.

**Returns:**

bool: True if the operators are equal, False otherwise.

**negate**()

Negates the operator matrix.

**Returns:**

Operator: The negated operator.

**scale**(*value: float*)

Scales the operator matrix by a value.

**Args:**

value (float): The scaling factor.

**Returns:**

Operator: The scaled operator.

---

**tensor**(*target: Operator*)

Computes the tensor product of the operator with another operator.

**Args:**

target (Operator): The operator to compute the tensor product with.

**Returns:**

Operator: The tensor product operator.

**update**(*row*, *column*, *value*)

Updates a specific element of the operator matrix.

**Args:**

*row* (int): The row index of the element to update. *column* (int): The column index of the element to update. *value* (float): The new value of the element.

**Returns:**

Operator: The updated operator.

**class** `utils.tensor.Vector`(*elements: list[int] | int | matrixInterface*)

This class represents a quantum state vector.

It may take the integer 1 or 0 to create a single qubit in the given state, a list or *matrixInterface* in the constructor.

**apply**(*operator: Operator*)

Applies an operator to the vector.

**Args:**

*operator* (Operator): The operator to apply.

**Returns:**

Vector: The resulting vector after applying the operator.

**equal**(*target: Vector*)

Checks if the vector is equal to another vector.

**Args:**

*target* (Vector): The vector to compare with.

**Returns:**

bool: True if the vectors are equal, False otherwise.

**measure**(*basis: Vector*)

Measures the vector with respect to a basis vector.

**Args:**

*basis* (Vector): The basis vector to measure against.

**Returns:**

float: The measurement result.

---

**scale**(*scalar: float*)

Scales the vector by a value.

**Args:**

*value* (float): The scaling factor.

**Returns:**

Vector: The scaled vector.

**tensor**(*target: Vector*)

Computes the tensor product of the vector with another vector.

**Args:**

*target* (Vector): The vector to compute the tensor product with.



**Returns:**

Vector: The tensor product vector.



## GATES MODULE

This module provides the implementation of quantum gates used in Grover's algorithm. The *Gate* class contains methods for creating the Hadamard gate, oracle gate, reflection gate, and identity gate.

**class** gates.**Gate**(*dimension: int*)

This is the gate class which contains the gates we need to implement Grover's algorithm.

When we initialise, we declare the dimension of the quantum register so that the gates (except the Hadamard) can be provided in the correct dimension.

The Hadamard gate is scaled (using tensor products) at the point of implementation to make the application more obvious.

**h()**

Single-qubit Hadamard gate.

This gate is a  $\pi$  rotation about the X+Z axis, and has the effect of changing computation basis from  $|0\rangle, |1\rangle$  to  $|+\rangle, |-\rangle$  and vice-versa.

**Matrix Representation:**

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

**i()**

Identity gate.

The identity gate performs no operation on the quantum state. It is represented by the identity matrix of size determined by the dimension of the quantum register.

**Matrix Representation:**

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

**Returns:**

Operator: The identity gate as an *Operator* object.

---

**oracle**(*target: int*)

Oracle gate for Grover's algorithm.

The oracle gate marks the target state by flipping its phase. It is a diagonal matrix with all elements on the diagonal equal to 1, except for the element corresponding to the target state, which is -1.

**Matrix Representation when targeting the second state:**

$$O = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

**Params:**

target (int): The target state to be marked by the oracle in little endian convention.

**Returns:**

Operator: The oracle gate as an *Operator* object.

**reflection()**

Reflection gate for Grover's algorithm.

The reflection gate reflects the quantum register in the direction of the initial state. It is a diagonal matrix with the first element on the diagonal equal to 1 and all other elements equal to -1.

**Matrix Representation:**

$$R = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

**Returns:**

Operator: The reflection gate as an *Operator* object

## TENSOR MODULE TEST SUITE

This module tests the classes included in the tensor module.

**class** tests.test\_tensor.TestOperator(*methodName='runTest'*)

This class aims to provide end to end testing of the Operator class within the tensor module.

The purpose of each test and the method by which it is validated is outlined below.

**test\_operator\_applied\_to\_vector()**

This test confirms that *Operator's are correctly applied to Vector's when the apply() method is used.*

**test\_operator\_tensor\_product()**

This test checks that the tensor product of two operators is computed correctly.

**test\_operator\_tensor\_product\_vs\_notes()**

Tensor Product of Hadamard, Identity and Hadamard, as presented in the slides -  $H \otimes \mathbb{I} \otimes H$ .

This manually creates the hadamard gates and performs the tensor product using the `tensor()` method.

The result is compared against the result in the slides:

$$H \otimes \mathbb{I} \otimes H = \frac{1}{2} \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 \\ 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 \\ 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & -1 & -1 \\ 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 \end{pmatrix}$$

**test\_product\_of\_H\_gates()**

This test confirms that raising a Hadamard gate to a power correctly computes the tensor product and provides the correct dimension of *Operator*.

**test\_update()**

This test confirms that when the update method is used on an *Operator* the correct element is updated to the correct value.

**class** tests.test\_tensor.TestVector(*methodName='runTest'*)

This class aims to provide end to end testing of the Vector class within the tensor module.

The purpose of each test and the method by which it is validated is outlined below.

**test\_scalar\_multiplication()**

This test checks the scalar multiplication method behaves as expected.

**test\_vector\_construction()**

This test checks that the *Vector* is correctly constructed when the *init* method is called.

**test\_vector\_tensor\_product()**

This test checks that the tensor product of two *Vector* is computed correctly.

## MATRIX INTERFACE

The matrix interface relies on the pipeline pattern whereby, with few exceptions, the methods should all return an object of the `matrixInterface`. This allows for methods to be chained for more readable code.

The parent class and available methods are described.

**class** `utils.matrixInterface.matrixInterface`(*size, elements*)

This class serves as the Abstract Base Class (ABC or interface) for the concrete implementations of the matrix object.

**abstract** `add`(*other: matrixInterface*) → *matrixInterface*

Performs elementwise addition of two matrices of the same size.

**abstract** `dimension`() → *matrixInterface*

Returns the number of rows in a column matrix or the number of rows and columns for a square matrix.

**abstract** `equal`(*other: matrixInterface*) → *matrixInterface*

Compares two matrices to check for equality, this will be a costly operation so should be reserved for testing.

**abstract** `flat`() → `_SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]`

Returns a flattened version of the matrix, this is equivalent to the transpose for column matrices so can be used during measurement.

**abstract** `property matrix`: `coo_array | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]`

The matrix property stores the array like object which handles the state of the object.

**abstract** `multiply`(*other: matrixInterface*) → *matrixInterface*

Performs matrix multiplication of two matrices. The order should be provided as it would be written.

**abstract** `negate`() → *matrixInterface*

Returns the original matrix with each element set to its own negative.

**abstract** `power`(*exponent: int*) → *matrixInterface*

Iteratively performs a tensor product as many times required. Useful for example when raising a gate to the power of the size of the quantum register to fully entangle the register.

**abstract** `reshape`(*rows: int, cols: int*) → *matrixInterface*

Casts the matrix object into a specific shape e.g. into a column for vectors or a square for operators.

**abstract** `scale`(*factor: float*) → *matrixInterface*

Performs elementwise scalar multiplication by the supplied factor.

**abstract property size: int**

The size property is used when the parent object is an operator and casts the array into a size x size matrix.

**abstract subtract**(*other: matrixInterface*) → *matrixInterface*

Performs elementwise subtraction of two matrices of the same size.

**abstract tensor**(*other: matrixInterface*) → *matrixInterface*

Computes the tensor product of the matrix with another matrix.

**abstract toVector**() → *matrixInterface*

Convenience method to cast a matrix into a column i.e. 1 column and as many rows as elements.

**abstract update**(*row: int, col: int, value: float*) → *matrixInterface*

Updates a specific index of the matrix to the provided value.



## SPARSE MATRIX IMPLEMENTATION

**class** `utils.sparseMatrix.sparseMatrix`(*size: int, elements: list | matrixInterface, vector=False*)

**add**(*other: matrixInterface*) → *sparseMatrix*

Performs elementwise addition of two matrices of the same size.

**dimension**() → *int*

Returns the number of rows in a column matrix or the number of rows and columns for a square matrix.

**equal**(*other: matrixInterface*) → *bool*

Compares two matrices to check for equality, this will be a costly operation so should be reserved for testing.

**flat**() → *\_SupportsArray[dtype[Any]] | \_NestedSequence[\_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | \_NestedSequence[bool | int | float | complex | str | bytes]*

Returns a flattened version of the matrix, this is equivalent to the transpose for column matrices so can be used during measurement.

**property matrix:** *coo\_array*

The matrix property stores the array like object which handles the state of the object.

**multiply**(*other: matrixInterface*) → *sparseMatrix*

Performs matrix multiplication of two matrices. The order should be provided as it would be written.

**negate**() → *sparseMatrix*

Returns the original matrix with each element set to its own negative.

**power**(*exponent: int*) → *sparseMatrix*

Iteratively performs a tensor product as many times required. Useful for example when raising a gate to the power of the size of the quantum register to fully entangle the register.

**reshape**(*rows: int, cols: int*) → *sparseMatrix*

Casts the matrix object into a specific shape e.g. into a column for vectors or a square for operators.

**scale**(*factor: float*) → *sparseMatrix*

Performs elementwise scalar multiplication by the supplied factor.

**property size**

The size property is used when the parent object is an operator and casts the array into a size x size matrix.

**subtract**(*other: matrixInterface*) → *sparseMatrix*

Performs elementwise subtraction of two matrices of the same size.

**tensor**(*other: matrixInterface*) → *sparseMatrix*

Computes the tensor product of the matrix with another matrix.

**toVector()** → *sparseMatrix*

Convenience method to cast a matrix into a column i.e. 1 column and as many rows as elements.

**update**(*row: int, col: int, value: float*) → *sparseMatrix*

Updates a specific index of the matrix to the provided value.

## DENSE MATRIX IMPLEMENTATION

**class** `utils.denseMatrix.denseMatrix`(*size: int, elements: list | matrixInterface, vector=False*)

**add**(*other: matrixInterface*) → *denseMatrix*

Performs elementwise addition of two matrices of the same size.

**dimension**() → int

Returns the number of rows in a column matrix or the number of rows and columns for a square matrix.

**equal**(*other: denseMatrix*) → bool

Compares two matrices to check for equality, this will be a costly operation so should be reserved for testing.

**flat**() → `_SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]]` | bool | int | float | complex | str | bytes | `_NestedSequence[bool | int | float | complex | str | bytes]`

Returns a flattened version of the matrix, this is equivalent to the transpose for column matrices so can be used during measurement.

**property matrix:** `ndarray[Any, dtype[_ScalarType_co]]`

The matrix property stores the array like object which handles the state of the object.

**multiply**(*other: denseMatrix*) → *denseMatrix*

Performs matrix multiplication of two matrices. The order should be provided as it would be written.

**negate**() → *denseMatrix*

Returns the original matrix with each element set to its own negative.

**power**(*exponent: int*) → *denseMatrix*

Iteratively performs a tensor product as many times required. Useful for example when raising a gate to the power of the size of the quantum register to fully entangle the register.

**reshape**(*rows: int, cols: int*) → *denseMatrix*

Casts the matrix object into a specific shape e.g. into a column for vectors or a square for operators.

**scale**(*factor: float*) → *denseMatrix*

Performs elementwise scalar multiplication by the supplied factor.

**property size**

The size property is used when the parent object is an operator and casts the array into a size x size matrix.

**subtract**(*other: matrixInterface*) → *denseMatrix*

Performs elementwise subtraction of two matrices of the same size.

**tensor**(*other: denseMatrix*) → *denseMatrix*

Computes the tensor product of the matrix with another matrix.

**toVector()** → *denseMatrix*

Convenience method to cast a matrix into a column i.e. 1 column and as many rows as elements.

**update**(*row: int, col: int, value: float*) → *denseMatrix*

Updates a specific index of the matrix to the provided value.

## PYTHON MODULE INDEX

### g

`gates`, [7](#)

### q

`qc`, [1](#)

### t

`tests.test_tensor`, [9](#)

### u

`utils.denseMatrix`, [15](#)

`utils.matrixInterface`, [11](#)

`utils.sparseMatrix`, [13](#)

`utils.state_vector`, [1](#)

`utils.tensor`, [3](#)



## A

add() (*utils.denseMatrix.denseMatrix method*), 15  
 add() (*utils.matrixInterface.matrixInterface method*), 11  
 add() (*utils.sparseMatrix.sparseMatrix method*), 13  
 apply() (*utils.tensor.Vector method*), 4

## C

Circuit (*class in qc*), 1

## D

denseMatrix (*class in utils.denseMatrix*), 15  
 dimension() (*utils.denseMatrix.denseMatrix method*), 15  
 dimension() (*utils.matrixInterface.matrixInterface method*), 11  
 dimension() (*utils.sparseMatrix.sparseMatrix method*), 13

## E

equal() (*utils.denseMatrix.denseMatrix method*), 15  
 equal() (*utils.matrixInterface.matrixInterface method*), 11  
 equal() (*utils.sparseMatrix.sparseMatrix method*), 13  
 equal() (*utils.tensor.Operator method*), 3  
 equal() (*utils.tensor.Vector method*), 4

## F

flat() (*utils.denseMatrix.denseMatrix method*), 15  
 flat() (*utils.matrixInterface.matrixInterface method*), 11  
 flat() (*utils.sparseMatrix.sparseMatrix method*), 13

## G

Gate (*class in gates*), 7  
 gates  
   module, 5  
 grover() (*qc.Circuit method*), 1

## H

h() (*gates.Gate method*), 7  
 h() (*qc.Circuit method*), 1

## I

i() (*gates.Gate method*), 7

## M

makeStateVector() (*in module utils.state\_vector*), 1  
 matrix (*utils.denseMatrix.denseMatrix property*), 15  
 matrix (*utils.matrixInterface.matrixInterface property*), 11  
 matrix (*utils.sparseMatrix.sparseMatrix property*), 13  
 matrixInterface (*class in utils.matrixInterface*), 11  
 measure() (*qc.Circuit method*), 1  
 measure() (*utils.tensor.Vector method*), 4  
 module  
   gates, 5  
   qc, 1  
   tests.test\_tensor, 8  
   utils.denseMatrix, 14  
   utils.matrixInterface, 10  
   utils.sparseMatrix, 12  
   utils.state\_vector, 1  
   utils.tensor, 1  
 multiply() (*utils.denseMatrix.denseMatrix method*), 15  
 multiply() (*utils.matrixInterface.matrixInterface method*), 11  
 multiply() (*utils.sparseMatrix.sparseMatrix method*), 13

## N

negate() (*utils.denseMatrix.denseMatrix method*), 15  
 negate() (*utils.matrixInterface.matrixInterface method*), 11  
 negate() (*utils.sparseMatrix.sparseMatrix method*), 13  
 negate() (*utils.tensor.Operator method*), 3

## O

Operator (*class in utils.tensor*), 3  
 oracle() (*gates.Gate method*), 7

## P

power() (*utils.denseMatrix.denseMatrix method*), 15  
 power() (*utils.matrixInterface.matrixInterface method*), 11

`power()` (*utils.sparseMatrix.sparseMatrix method*), 13

## Q

`qc`  
module, 1

## R

`reflection()` (*gates.Gate method*), 8  
`reshape()` (*utils.denseMatrix.denseMatrix method*), 15  
`reshape()` (*utils.matrixInterface.matrixInterface method*), 11  
`reshape()` (*utils.sparseMatrix.sparseMatrix method*), 13

## S

`scale()` (*utils.denseMatrix.denseMatrix method*), 15  
`scale()` (*utils.matrixInterface.matrixInterface method*), 11  
`scale()` (*utils.sparseMatrix.sparseMatrix method*), 13  
`scale()` (*utils.tensor.Operator method*), 3  
`scale()` (*utils.tensor.Vector method*), 4  
`size` (*utils.denseMatrix.denseMatrix property*), 15  
`size` (*utils.matrixInterface.matrixInterface property*), 11  
`size` (*utils.sparseMatrix.sparseMatrix property*), 13  
`sparseMatrix` (*class in utils.sparseMatrix*), 13  
`subtract()` (*utils.denseMatrix.denseMatrix method*), 15  
`subtract()` (*utils.matrixInterface.matrixInterface method*), 12  
`subtract()` (*utils.sparseMatrix.sparseMatrix method*), 13

## T

`tensor()` (*utils.denseMatrix.denseMatrix method*), 15  
`tensor()` (*utils.matrixInterface.matrixInterface method*), 12  
`tensor()` (*utils.sparseMatrix.sparseMatrix method*), 13  
`tensor()` (*utils.tensor.Operator method*), 3  
`tensor()` (*utils.tensor.Vector method*), 4  
`test_operator_applied_to_vector()`  
(*tests.test\_tensor.TestOperator method*), 9  
`test_operator_tensor_product()`  
(*tests.test\_tensor.TestOperator method*), 9  
`test_operator_tensor_product_vs_notes()`  
(*tests.test\_tensor.TestOperator method*), 9  
`test_product_of_H_gates()`  
(*tests.test\_tensor.TestOperator method*), 9  
`test_scalar_multiplication()`  
(*tests.test\_tensor.TestVector method*), 9  
`test_update()` (*tests.test\_tensor.TestOperator method*), 9  
`test_vector_construction()`  
(*tests.test\_tensor.TestVector method*), 9  
`test_vector_tensor_product()`  
(*tests.test\_tensor.TestVector method*), 10

`TestOperator` (*class in tests.test\_tensor*), 9

`tests.test_tensor`  
module, 8

`TestVector` (*class in tests.test\_tensor*), 9

`toVector()` (*utils.denseMatrix.denseMatrix method*), 15  
`toVector()` (*utils.matrixInterface.matrixInterface method*), 12  
`toVector()` (*utils.sparseMatrix.sparseMatrix method*), 13

## U

`update()` (*utils.denseMatrix.denseMatrix method*), 16  
`update()` (*utils.matrixInterface.matrixInterface method*), 12  
`update()` (*utils.sparseMatrix.sparseMatrix method*), 14  
`update()` (*utils.tensor.Operator method*), 4  
`utils.denseMatrix`  
module, 14  
`utils.matrixInterface`  
module, 10  
`utils.sparseMatrix`  
module, 12  
`utils.state_vector`  
module, 1  
`utils.tensor`  
module, 1

## V

`Vector` (*class in utils.tensor*), 4