Quantum Computing Project

Release 1.0

Group 5

Mar 18, 2024

CONTENTS:

CHAPTER

ONE

NAME THE CHAPTER UP HERE

This narrative can explain the whole point of the module (the file).

You can include maths inline like this: $a^2 + b^2 = c^2$

Or, if you need to display, do it like this (be sure to leave a blank line above the .. math:: delcaration):

$$\lim_{n \to \infty} \frac{1}{n} \neq \infty$$

We can make use of referencing code objects like classes and methods like this:

In this chapter we will start by explaining the *Example* class and the required arguments for the `__init__()` method.

Note: Add notes like this.

Warning: Add warnings like this.

Insert code sample with backticks *def __init__()*. Make text **bold**, or *italics* just like markdown.

This is a list:

- With
- Bullets.

This is also a list:

1. With
2. Numbers.

class example.**Example**(*arg1: int*)

Class explanation.

This can be a high level overview of what the object is.

arg1

Describe the *self.arg1* term. This behaviour is unique to the init method.

usefulMethod(*input: int*)

Explain the utility of this method.

Params:

input:
This `int` should represent something.

Returns:
This method returns a `Vector`.

`class` example.**Example2**

This is the second class

CHAPTER

TWO

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*.

CHAPTER

THREE

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.

CHAPTER

FOUR

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.

CHAPTER

FIVE

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.

CHAPTER

SIX

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, outer products, addition, subtraction, scalar multiplication, and exponentiation. It also provides methods for applying an operator to a vector and checking equality between vectors.

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

This class represents a quantum operator as a sparse matrix.

Args:

size (int): The size of the operator matrix. elements (Union[list, coo_array]): The matrix elements as a list or a *coo_array*.

Attributes:

matrix (coo_array): The sparse matrix representation of the operator. size (int): The size of the operator matrix.

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.

Args:

elements (Union[list[int], int, coo_array]): The vector elements as a list, an integer (0 or 1),
or a *coo_array*.

Attributes:
vector (coo_array): The sparse vector representation. dimension (int): The dimension of the vector.

`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.

CHAPTER

SEVEN

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.

Note: This code is still under development and until the version is incremented to a 1.* you should not trust any of this documentation.

---

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 π 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:

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

Params:
target (int): The target state to be marked by the oracle.

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

> **Warning:** unsure of this gate description, maybe worth to check others as well

`reflection`()

Reflection gate for Grover's algorithm.

The reflection gate reflects the amplitudes of all states about the average amplitude. 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

CHAPTER

EIGHT

TENSOR MODULE TEST SUITE

This module tests the classes included in the tensor module.

class tests.test_tensor.**TestOperator**(*methodName='runTest'*)

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:

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

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_vector_construction()

This test aims to check…

PYTHON MODULE INDEX

INDEX