|  |
| --- |
|  |
| qvantum – python module |
| Documentation |

|  |
| --- |
| Sohonyai, Ádám  2018.11.18. |

Contents

[1 Introduction 2](#_Toc735602)

[2 Installing 2](#_Toc735603)

[2.1 pip install 2](#_Toc735604)

[2.2 wheel install 2](#_Toc735605)

[2.3 setup file 2](#_Toc735606)

[3 Module Classes 2](#_Toc735607)

[3.1 Qubit class 3](#_Toc735608)

[3.2 Register class 5](#_Toc735609)

[3.3 Gate class 10](#_Toc735610)

[3.4 Layer class 15](#_Toc735611)

[3.5 Circuit class 15](#_Toc735612)

[3.6 Bloch class 15](#_Toc735613)

[4 Examples 15](#_Toc735614)

[4.1 Quantum teleportation 15](#_Toc735615)

[4.2 Grover’s algorithm 15](#_Toc735616)

[5 Notes 15](#_Toc735617)

[5.1 Module reading error 15](#_Toc735618)

[5.2 Deleting a qubit from register 16](#_Toc735619)

[5.3 Ѱ sign in python2 16](#_Toc735620)

# Introduction

qvantum is a python module with the goal to ensure an easy use library for understanding quantum computing better or designing new quantum algorithms. Working with this module helps you the get more familiar with the basic concepts such as qubit, register or quantum gate, meanwhile the tool has the power for deeper analysis and development.

The module is in beta release phase: tested but it might contain bugs, therefore every constructive note is highly welcomed. Also If you would like to collaborate in the developing process then do not hesitate and contact us.

# Installing

qvantum module can be easily installed using three different approach below.

## pip install

The latest version of the module can be installed online from the PyPi page using pip in command line:

pip install qvantum

or

python –m pip install qvantum

or

python -m pip install --index-url <https://test.pypi.org/simple/> qvantum

## wheel install

The latest version of the module can be downloaded from the PyPi page in .whl format which can be used for installation:

pip install qvantum-x.xx-py2.py3-none-any.whl

or

python –m pip install qvantum-x.xx-py2.py3-none-any.whl

## setup file

A setup.py file is also available on PyPi page. Download the file then run the command in the folder where the setup.py file was downloaded. Use –e if you want the module be immediately available for every user in your system:

pip install . (or pip install –e .)

or

python –m pip install . (python –m pip install –e .)

# Module Classes

In qvantum module there are some classes which represents the basic objects in quantum computing such as: a qubit, a register, a gate, a layer and a circuit. These objects (and therefore the classes which represents them) are built on each other. Due to this concept a register is built on qubits, layers are formed by gates and circuits are created out of gates.

There is a sixth class, the bloch class which is used for teh Bloch representation and visualisation of a qubit.

## Qubit class

In quantum computing a qubit or quantum bit is the basic unit of quantum information. Every qubit has two clear states such as 0 and 1 but unlike a classical bit a qubit can be in superposition which is a special mixture of these clear states.

An instance of the qubit class has these methodes below.

qvantum.Qubit.\_\_init\_\_(alpha, beta)

Method to initialize an instance of the qubit class. The squared sum of alpha and beta bust be equal to zero otherwise a ValueError will be thrown.

Parameters:

alpha: int, float or complex

beta: int, float or complex

Examples:

import math

import qvantum

q = qvantum.Qubit(1, 0)

qvantum.Qubit(1 / math.sqrt(2), 1 / math.sqrt(2))

q = qvantum.Qubit(5, 2)

q = qvantum.Qubit(1, ’red’)

qvantum.Qubit(1, 0).show()

qvantum.Qubit.get\_alpha()

Getter method of alpha.

Examples:

import qvantum

q = qvantum.Qubit(1, 0)

q.get\_apha()

qvantum.Qubit.get\_beta()

Getter method of beta.

Examples:

import qvantum

q = qvantum.Qubit(1, 0)

q.get\_beta()

qvantum.Qubit.set\_amplitudes(alpha, beta)

Setter method to replace the old coefficients to new ones. The squared sum of alpha and beta bust be equal to zero otherwise a ValueError will be thrown.

Parameters:

alpha: int, float or complex

beta: int, float or complex

Examples:

import math

import qvantum

q = qvantum.Qubit(1, 0)

q.show()

q.set\_amplitudes(0, 1)

q.show()

qvantum.Qubit.show()

This method shows the state function of the qubit object.

Examples:

import qvantum

q = qvantum.Qubit(1, 0)

q.show()

qvantum.Qubit.measure()

This method performs a measurement on the qubit and returns with one clear state by the distribtion according to the coefficients.

Examples:

import qvantum

q = qvantum.Random\_Qubit()

q.show()

q.measure()

q.show()

qvantum.Qubit.ket()

This method returns with the ket vector representation of the qubit.

Examples:

import qvantum

q = qvantum.Random\_Qubit()

q.ket()

qvantum.Qubit.bra()

This method returns with the bra vector representation of the qubit.

Examples:

import qvantum

q = qvantum.Random\_Qubit()

q.bra()

qvantum.Random\_Qubit.\_\_init\_\_()

This is an inhereted class from the Qubit class. They share the same methods but when an instance of the Random\_Qubit class is created the coefficients are randomly choosen.

Examples:

import qvantum

rq = qvantum.Random\_Qubit()

rq.show()

qvantum.Random\_Qubit().show()

qvantum.check\_qubit.py

This is the decorator file and is used to check the arguments when a method is called with parameters.

## Register class

A system containing more than one qubit is called a register. The state function of a register is the superposition of the allowed states of the qubits. The number of possible states is increasing exponentially with every new qubit is added to the system.

An instance of the qubit class has these methodes below.

*qvantum.Register.\_\_init\_\_(qubit\_list)*

Method to initialize an instance of the register class. The input is a list of elements in the Qubit or Random\_Qubit class. Also this list must contain at least 2 elements.

Parameters:

qubit\_list: list of element in Qubit or Random\_Qubit class

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

r1 = qvantum.Register([q1])

r1 = qvantum.Register([q1, ’shoe’])

r1 = qvantum.Register({q1, q2})

r2 = qvantum.Register([q1, q2])

r2.show()

qvantum.Register.get\_coeff\_list()

This method returns the coefficients of the qubits in the regsiter.

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q1.show()

q2 = qvantum.Random\_Qubit()

q2.show()

r = qvantum.Register([q1, q2])

r.get\_coeff\_list()

qvantum.Register.get\_state\_number()

This method returns the number of the possible clear states for the register.

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2])

r.show()

r.get\_state\_number()

qvantum.Register.get\_qubit\_number()

This method returns the number of the qubits in the register.

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2])

r.show()

r.get\_qubit\_number()

qvantum.Register.get\_states(nth=None)

This method returns with the n-th possible state for the regsiter if the parameter is definit. If it isn’t then the return value is the list of all possible states.

Parameters:

nth: int, None

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2])

r.show()

r.get\_states(6)

r.get\_states(’red’)

r.get\_states(2)

qvantum.Register.get\_amplitudes(nth=None)

This method returns with the coefficient of the n-th possible state for the regsiter if the parameter is definit. If it isn’t then the return value is the list of the coefficients of all possible states.

Parameters:

nth: int, None

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2])

r.show()

r.get\_amplitudes(6)

r.get\_amplitudes(’red’)

r.get\_amplitudes(2)

qvantum.Register.set\_amplitudes(amp\_list)

This method sets new coefficients for the possible states of the register. The input parameter is a list of real or complex number and their squared sum must be equal to 1. Number of elements in the least must be equal with the number of possible states.

Parameters:

amp\_list: list of int, float or complex

Example:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

q3 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2, q3])

r.show()

r.set\_amplitudes([0, 0, 1, 0, 0, 0, 0, 0])

r.show()

*qvantum.Register.show()*

This method shows the state function of the register object.

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

q3 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2, q3])

r.show()

*qvantum.Register.measure\_register()*

This method performs a measurement on the whole register and returns the final state of the register after the process. This final state is randomized regarding to the amplitudes of the register.

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

q3 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2, q3])

r.show()

r.measure\_register()

r.show()

*qvantum.Register.measure\_nth\_qubit(nth)*

This method performs a measurement on the n-th qubit in the register and returns the final state of the register after the process. This final state is randomized regarding to the amplitudes of the register. The input parameter must be an integer corresponding to the number of qubits in the register.

Parameters:

nth: int

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

q3 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2, q3])

r.show()

r.measure\_nth\_qubit(3)

r.measure\_nth\_qubit(’red’)

r.measure\_nth\_qubit(2)

r.show()

qvantum.Register.ket()

This method returns with the ket vector representation of the register.

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2])

r.show()

r.ket()

qvantum.Register.bra()

This method returns with the bra vector representation of the register.

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2])

r.show()

r.bra()

qvantum.Regsiter.delete\_qubit(nth)

This method deletes the n-th qubit from the regsiter. This method has some drawback, for more information see section 5.2 The input parameter must be an integer corresponding to the number of qubits in the register.

Parameters:

nth: int

Examples:

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

q3 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2, q3])

r.show()

r.delete\_qubit(3)

r.delete\_qubit()

r.delete\_qubit(1)

r.show()

qvantum.Register.insert\_qubit(q, nth)

This method inserts a given qubit into a register. The input parameter must be an integer corresponding to the number of qubits in the register.

Parameters:

q: Qubit or Random\_Qubit

nth: int

Examples:

import qvantum

q1 = qvantum.Random\_Qubit()

q2 = qvantum.Random\_Qubit()

r = qvantum.Register([q1, q2])

r.show()

q3 = qvantum.Random\_Qubit()

r.insert\_qubit(q3, 3)

r.insert\_qubit(, 2)

r.insert\_qubit(’red’, 2)

r.insert\_qubit(q3, 2)

r.show()

r.insert\_qubit(q3, 2)

r.show()

qvantum.check\_register.py

This is the decorator file and is used to check the arguments when a method is called with parameters.

## Gate class

Quantum logic gates are the basic building blocks of quantum circuits. They are operating on a few number of qubits. Quantum gates have the same function as classical logical gates in digital circuits. Every quantum gate is represented as an unitary matrix moreover every unitary matrix can be used as a quantum logic gate.

*qvantum.Gate.\_\_init\_\_()*

This method initialize a 2x2 sized identity matrix. Every identity matrix is a unitary matrix as well.

Examples:

import qvantum

G = qvantum.Gate()

G.get\_name()

G.get\_matrix()

G.get\_size()

*qvantum.Gate.\_\_call\_\_(qr)*

This method makes possible to call a gate on a qubit or a register. The only restriction is the size of the gate and the size of the qubit or regsiter must be equal to each other.

Parameters:

qr: Qubit, Random\_Qubit or Register

Examples:

import qvantum

q = qvantum.Random\_Qubit()

q.show()

qvantum.Hadamard(q)

q.show()

*qvantum.Gate.get\_name()*

This method retuns the name of the gate.

Examples:

import qvantum

H = qvantum.Hadamard()

H.get\_name()

*qvantum.Gate.get\_matrix()*

This method retuns the unitary matrix of the gate.

Examples:

import qvantum

C = qvantum.CNOT()

C.get\_matrix()

*qvantum.Gate.get\_size()*

This method retuns the size of the unitary matrix of the gate.

Examples:

import qvantum

T = qvantum.Toffoli()

T.get\_size()

*qvantum.Gate.set\_name(name)*

This method sets a new name for the gate.

Parameters:

name: string

Examples:

import qvantum

H = qvantum.Hadamard()

H.get\_name()

H.set\_name(’New Hadamard’)

H.get\_name()

*qvantum.Gate.set\_matrix(matrix)*

This method sets a new unitary matrix for the gate. If matrix is not unitary then an error is raised.

Parameters:

matrix: numpy.ndarray

Examples:

import numpy

import qvantum

G = qvantum.Gate()

G.get\_matrix()

G.set\_matrix(numpy.matrix([

[1 / numpy.sqrt(2), 1 / numpy.sqrt(2)],

[1 / numpy.sqrt(2), -1 / numpy.sqrt(2)]

]))

G.get\_matrix()

*qvantum.Gate.power(power)*

This method raises the unitary matrix of the gate to the given power and overwrites the original matrix of the gate with the results matrix.

Parameters:

power: int

Examples:

import qvantum

T = qvantum.Toffoli()

T.get\_matrix()

T.power(2.1)

T.power(3)

T.get\_matrix()

*qvantum.Hadamard()*

This class is an inherited class from the Gate class. It’s the implementation of the Hadamard gate. Its unitary matrix looks like this:

*qvantum.SquareNot()*

This class is an inherited class from the Gate class. It’s the implementation of the Square-Not gate. Its unitary matrix looks like this:

*qvantum.PauliX()*

This class is an inherited class from the Gate class. It’s the implementation of the Pauli-X gate. Its unitary matrix looks like this:

*qvantum.PauliY()*

This class is an inherited class from the Gate class. It’s the implementation of the Pauli-Y gate. Its unitary matrix looks like this:

*qvantum.PauliZ()*

This class is an inherited class from the Gate class. It’s the implementation of the Pauli-Z gate. Its unitary matrix looks like this:

*qvantum.Phase()*

This class is an inherited class from the Gate class. It’s the implementation of the Phase gate. Its unitary matrix looks like this:

*qvantum.Pi8()*

This class is an inherited class from the Gate class. It’s the implementation of the Pi8 gate. Its unitary matrix looks like this:

*qvantum.Swap()*

This class is an inherited class from the Gate class. It’s the implementation of the Swap gate. Its unitary matrix looks like this:

*qvantum.SquareSwap()*

This class is an inherited class from the Gate class. It’s the implementation of the Square-Swap gate. Its unitary matrix looks like this:

*qvantum.CNOT(control\_qubit, target\_qubit)*

This class is an inherited class from the Gate class. It’s the implementation of the Controlled-Not gate. It’s called on 2 qubits. The parameters determine which one is the control and the target – (0, 1) or (1, 0)

Parameters:

control\_qubit: 0, 1

target\_qubit: 0, 1

Its unitary matrix looks like this – (0, 1) or (1, 0):

or

*qvantum.ControlledZ()*

This class is an inherited class from the Gate class. It’s the implementation of the Controlled-Z gate. Its unitary matrix looks like this:

*qvantum.ControlledPhase()*

This class is an inherited class from the Gate class. It’s the implementation of the Controlled-Phase gate. Its unitary matrix looks like this:

*qvantum.Ising(phi)*

This class is an inherited class from the Gate class. It’s the implementation of the Ising gate.

Parameters:

phi: int, float

Its unitary matrix looks like this:

*qvantum.Toffoli(target\_qubit)*

This class is an inherited class from the Gate class. It’s the implementation of the Toffoli gate. It’s called on 3 qubits. The parameters determine which one is the target qubit – 0, 1 or 2.

Parameters:

target\_qubit: 0, 1, 2

Its unitary matrix looks like this (target qubit – 0, 1 or 2):

or or

*qvantum.Fredkin(control\_qubit)*

This class is an inherited class from the Gate class. It’s the implementation of the Fredkin gate. It’s called on 3 qubits. The parameters determine which one is the control qubit – 0, 1 or 2.

Parameters:

control\_qubit: 0, 1, 2

Its unitary matrix looks like this (control qubit – 0, 1 or 2):

or or

qvantum.check\_gate.py

This is the decorator file and is used to check the arguments when a method is called with parameters.

## Layer class

xxx

## Circuit class

xxx

## Bloch class

xxx

# Examples

xxx

## Quantum teleportation

xxx

## Grover’s algorithm

xxx

# Notes

xxx

## Module reading error

xxx

## Deleting a qubit from register

xxx

## Ѱ sign in python2

xxx