# QUICK START

(see accompanying helloworld.py)

Lets start with a simple example of using qclib, something like a 'hello world' --

import qclib

qc = qclib.qcsim(8)

qc.qgate(qc.H(),[0])

qc.qgate(qc.C(),[0,3])

qc.qreport()

qc.qmeasure(0)

qc.qreport()

When run, the above code generates the following output --

State

00000000 0.70710678+0.00000000j

00001001 0.70710678+0.00000000j

State

00001001 1.00000000+0.00000000j

With respect to the functionality, the above code starts with an 8-qbit system, and then sets up a bell state between qbits 3 and 0. And then measures qbit 0. This is depicted in the diagram below --

7 -------------------------

6 -------------------------

5 -------------------------

4 -------------------------

3 -----O-------------------

2 -----|-------------------

1 -----|-------------------

0 -[H]-.-(/)---------------

(Using (/) to depict the measurement operation.)

The output emitted by the above code is the dump of the state right after the setting up of the bell state, and then after measuring the qbit 0.

Lets go line by line to understand the code --

**import qclib**

This line imports the qclib. Fairly straight forward.

**qc = qclib.qcsim(8)**

The main class in the qclib library is qcsim. This line creates an instance of that class. As a convention in this document, qc is always used to represent an instance of the class qcsim. The qcsim class instance, qc, represents a quantum computer. The number 8 as the parameter to the constructor of class qclib, specifies the number of qbits in that quantum computer.

**qc.qgate(qc.H(),[0])**

qgate() is the function that applies a quantum gate on a given set of qbits in the system. In the above line of code, it applies the hadamard gate (H()), to qbit 0. See the documentation below for the gates available.

**qc.qgate(qc.C(),[0,3])**

Here a CNOT gate (C()) is applied on bits 0 and 3, with qbit 0 being the control qbit.

**qc.qreport()**

The function qreport() outputs the current superposition state. The first batch of lines is that output.

**qc.qmeasure(0)**

The qmeasure() function measures qbit 0.

**qc.qreport()**

This last qreport() function call outputs the state after the measurement of the qbit 0.

Finally, let us run the same code with trace turned on (therefore removed the qreport() function calls). Turning trace ON, it outputs the state after init and each gate and measurement step (see accompanying helloworld\_traceON.py) --

import qclib

qc = qclib.qclib(8, qtrace=True)

qc.qgate(qc.H(),[0])

qc.qgate(qc.C(),[0,3])

qc.qmeasure(0)

and, here is its output --

Initial State

00000000 1.00000000+0.00000000j

HADAMARD Qbit[0]

00000000 0.70710678+0.00000000j

00000001 0.70710678+0.00000000j

C-NOT Qbit[0, 3]

00000000 0.70710678+0.00000000j

00001001 0.70710678+0.00000000j

MEASURED Qbit[0] = 1 with probality = 0.5

00001001 1.00000000+0.00000000j

# FUNCTIONS

## qclib.qcsim(nqbits, qtrace=False, qzeros=False)

First argument specifies the number of qbits in the system. Prepares all qbits to |0>.

If qtrace=True, causes each operation (gate, measurement) to emit resulting state.

If qzeros=True, prints even those whose amplitude is 0.

## qgate(gate\_function, list\_of\_qbits, display=False)

qgate() is used to perform quantum gate operations on a set of qbits.

There are a number of gates pre-created within the qclib class, and additional gates can be defined by the users (see Section USER DEFINED GATE below). If a gate operates on more than 1 qbit (e.g., CNOT gate, or SWAP gate, etc.) then the list in the second argument (list\_of\_qbits) would contain that many qbits.

The function qgate() validates the number of bits passed in list\_of\_qbits against the number of qbits required for the gate, if not correct, throws an exception.

If display is True, the resulting state is printed out.

## comb\_op = qcombine\_seq(name, op\_list)

Combines a sequential application of gates into one equivalent gate. The parameter name is the name of the resulting gate. Parameter op\_list is a list of gates each of the structure [name,matrix].

--[A]-–[B]--[C]--[D]— 🡺 --[G]--

To combine the above –

G = qc.qcombine\_seq(“SEQ”,[A,B,C,D])

and to use it, for instance to apply it on qbit 2 --

qc.qgate(G,[2])

## comb\_op = qcombine\_par(name, op\_list)

Combines a parallel application of gates into one equivalent gate. The parameter name is the name of the resulting gate. Parameter op\_list is a list of gates each of the structure [name,matrix].

\_

3 --[A]---- 3 --| |--

2 -–[B]---- 🡺 2 --| |--

1 --[C]---- 1 --|G|--

0 --[D]---- 0 --| |--

-

To combine the above into one operation with 4 qbits as inputs –

G = qc.qcombine\_par(“PAR”,[A,B,C,D])

and to use it, for instance to apply it on qbits 7,5,3,1 --

qc.qgate(G,[7,5,3,1])

An interesting example is –

\_

1 --[H]---- 1 --| |--

|H2|

0 --[H]---- 0 --| |--

--

H2 = qc.qcombine\_par(“H2”,[H,H])

\_

3 ---.----- 1 --| |--

| | |

2 ---O----- 1 --| |--

|C2|

1 ---.----- 0 --| |--

| | |

0 ---O----- 0 --| |--

--

C2 = qc.qcombine\_par(“C2”,[C,C])

Create 2 bell states between quits 7,6 and 5,4 using these --

qc.qgate(H2,[7,5])

qc.qgate(C2,[7,6,5,4])

## v = qmeasure(qbit\_number,display=False)

Returns the measured value of the qbit, 0 or 1.

If the bit is in a superposition with some amplitude, the measurement operation will cause the state to randomly collapse to one or the other with the appropriate probability. In this simulator, the Python random.random() function is used to generate the randomness to decide which state to collapse into.

Cleary, the computations can continue after the measurement operations.

If display is True, the resulting state is printed out.

## qreport(state=None, header="State")

Prints the current state of the system. if state parameter is provided with a column numpy.matrix, it prints that state instead.

The parameter header provides the text to be printed above the state information.

## st = qstate()

Returns a Python array of the complex amplitudes of all possible superpositions of the qbits in the system.

## n = qsize()

Returns the number of qbits in the system.

## qtraceON(boolean)

Turns ON or OFF outputting of state after each qgate() and qmeasure() function call.

## qzerosON(boolean)

Turns ON or OFF of zero amplitude states in trace outputs and qreport() outputs.

# PERFORMANCE

On my laptop PC, I could go upto 11 qbits. Up until 8 qbits the simulator runs pretty fast. Beyond that the speed starts slowing down. At 11 qbits the simulator takes many seconds to apply 1 gate. At 12 qbits I get Memory Error.

I have not found any functionality bugs in this version of the simulator. Please report below if you find any.

# PRE DEFINED GATES

A number of gates are pre-defined in the qclib class. The following is the list --

**H()** Hadamard gate

**X()** Pauli\_x gate

**Y()** Pauli\_y gate

**Z()** Pauli\_z gate

**R(phi)** Phase rotation by phi

**C()** CNOT gate

**SWAP()** SWAP gate

# CREATING USER DEFINED GATES

(see accompanying user\_def\_gates.py)

Qcsim allows using user defined gates. A user defined gate would be written as a function that returns a Python array with two elements [name\_string, unitary\_matrix]. The element unitary\_matrix is the matrix that specifies the gate. It should be created using numpy.matrix([...],dtype=complex), or equivalent. The element name\_string is a string that is a user-friendly name of that gate that is used in logs and debug traces.

Qcsim does NOT verify if unitary\_matrix in the user-defined gate is indeed a unitary matrix. It is left as the responsibility of the user. <Perhaps an enhancement for later on.>

Here is an example of a simple way for a user to define a CNOT gate --

def myCNOT():

return ["MY-CNOT", numpy.matrix([[1,0,0,0],[0,1,0,0],[0,0,0,1],[0,0,1,0]],dtype=complex)]

As shown in the code above, while writing the unitary\_matrix for myCNOT we assumed as if the entire system has only 2 qbits.

Now, since in this definition of CNOT the higher order qbit (MSB) is the controlling qbit, hence when applying this gate to any two qbits, say, 4 and 7, where say, bit 4 is the controlling qbit, the qgate() function would be invoked as --

qc.qgate(myCNOT(),[4,7])

Or, if qbit 7 is to be the controlling qbit, then --

qc.qgate(myCNOT(),[7,4])

Qclib does all the trickery required to convert the gate matrix to handle all the qbits of the system, and performing the gate operation only on the speciic qbits.

So, in general, if the gate operates on n qbits, then it should be written as if the entire system consists of only n qbits, and when calling qgate(), the order of qbits should be [msb, ..., lsb].

The system looks at the size of the matrix specifying the gate to determine the number of qbits required for that gate. If the number is incorrect, it throws an exception.

Since the gate is defined in form of a function, it can take parameters. For instance, for a rotation gate, the rotation angle can be passed as a parameter --

def myR(theta):

c = numpy.cos(theta)

s = numpy.sin(theta)

return ["MY-Rotation({:0.4f})".format(theta), numpy.matrix([[1,0],[0,complex(c,s)]],dtype=complex)]

qc.qgate(myR(numpy.pi/2),[5])

Note: To see some change in the state, perform an X() or H() or some such on the qbit before the phase rotation.