BILINGUAL PUBLISHING CO.
Pioneer of Global Academics Since 1984

Journal of Computer Science Research
https://ojs.bilpublishing.com/index.php/jcsr

ARTICLE

Quick Quantum Circuit Simulation

Daniel Evans*

Pace University, New York, United States

ARTICLE INFO

Article history
Received: 15 August 2021
Accepted: 9 September 2021
Published Online: 18 September 2021

Keywords:
Quantum
Computing
Circuit
Simulation
Education
Software

ABSTRACT

Quick Quantum Circuit Simulation (QQCS) is a software system for computing the result of a quantum circuit using a notation that derives directly from the circuit, expressed in a single input line. Quantum circuits begin with an initial quantum state of one or more qubits, which are the quantum analog to classical bits. The initial state is modified by a sequence of quantum gates, quantum machine language instructions, to get the final state. Measurements are made of the final state and displayed as a classical binary result. Measurements are postponed to the end of the circuit because a quantum state collapses when measured and produces probabilistic results, a consequence of quantum uncertainty. A circuit may be run many times on a quantum computer to refine the probabilistic result. Mathematically, quantum states are 2^n-dimensional vectors over the complex number field, where n is the number of qubits. A gate is a $2^n \times 2^n$ unitary matrix of complex values. Matrix multiplication models the application of a gate to a quantum state. QQCS is a mathematical rendering of each step of a quantum algorithm represented as a circuit, and as such, can present a trace of the quantum state of the circuit after each gate, compute gate equivalents for each circuit step, and perform measurements at any point in the circuit without state collapse. Output displays are in vector coefficients or Dirac bra-ket notation. It is an easy-to-use educational tool for students new to quantum computing.

1. Introduction

The beginning quantum computing student immediately confronts the steep hurdle of the mathematics of complex vector spaces needed to understand the basics. While there are new languages and extensive systems available to aid quantum computations, they add to the learning curve. Quantum algorithms are presented as circuits or gate sequences, and one wants to know several things that are not immediately available in quantum programming systems. First, a trace of the quantum state after each circuit gate is convenient. A display of the gate equivalent at any point in the circuit is also desirable. Measures of the quantum probabilities, without state collapse, are helpful. All this information is available in a circuit simulation. A circuit simulation is not a quantum computer simulation. It is a mathematical rendering of each step of a quantum algorithm described by a sequence of gate operations on an initial quantum state and rendered by the software system described in this paper, Quick Quantum Circuit Simulation (QQCS). The system allows a student to quickly construct a circuit using a linear notation motivated by the circuits themselves and acquire the information to analyze an algorithm without the need for extensive computation.

As an example of the operation of QQCS, consider

**Corresponding Author:*
Daniel Evans,
Pace University, New York, United States;
Email: de36804p@pace.edu

 DOI: https://doi.org/10.30564/jcsr.v3i4.3567

the circuit in Figure 1 [1,2] which is a sequence of basic one- and two-qubits gates that together implement a more complicated gate known as a controlled Hadamard gate.

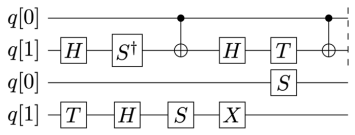

Figure 1. A Controlled Hadamard Gate Sequence

In an interactive QQCS session, the user enters the circuit in a single QQCS statement on one line, shown in Listing 1, and the result is immediately displayed after the Enter key.

```
(1)[1] :_H :_Sa :Cx :_H :_T :Cx :_T :_H :_S :_X :S_
[0.7+0.7i  0          0          0
0          0.7+0.7i   0          0
0          0          0.5+0.5i   0.5+0.5i
0          0          0.5+0.5i   -0.5-0.5i]
```

Listing 1. QQCS Linear Notation, Input and Output for the Controlled-H Gate Sequence of Figure 1

Listing 1 Explanation

(1) The user enters the full gate sequence. A gate is introduced by a colon (:) followed by a mnemonic for the common name of the gate, H for Hadamard, Sa for S-gate adjoint ($\pi/2$ phase gate inverse), Cx for controlled-X, and so forth. The underscore character (_) is used to position the gate in the circuit, and to represent a qubit line with no gate (an implied identity gate). In Figure 1, all the gates one-qubit gates except for the two-qubit Cx gates at step 3 and step 6 of the circuit.

The following output display is the final gate matrix result. Since $e^{i\pi/4} = \cos(\pi/4) + i\sin(\pi/4) = 0.7 + 0.7i$, the result is equivalent to

$$e^{i\pi/4}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

a controlled-Hadamard gate with a global phase factor of $e^{i\pi/4}$.

In this QQCS statement, there is no initial quantum state, so the output display represents the gate equivalent, the complex matrix that is the controlled-Hadamard gate, the product of eleven 4×4 matrix multiplications. The quantum computing student who is using QQCS as a study tool, can quickly see and dissect the operation of any circuit encountered in a textbook. Supporting software for quantum programming education is important [3]. QQCS provides a simple tool that does not detract from the primary learning task.

2. Related Work

For an extensive review of quantum programming, see the article "Quantum Programming Languages" [4]. The following references are selected as systems in which results close to those of QQCS could be obtained. In a few cases, they are explicitly compared to QQCS.

One of the most well-known quantum computing resources is IBM's Quantum Experience [2] public web site. IBM supports the open-source software QISKit [5], a set of Python libraries which allow one to write quantum programs (circuits) in a language called Open Quantum Assembly Language (QASM), the language used to program the real quantum computers available through the website. QISKit can submit to the website or simulate locally the operation of QASM. In QISKit, the circuit of Figure 1 would be written as shown in Listing 2.

```
def ctl_h():
   qs = QuantumRegister(2, 'qs')
   cr = ClassicalRegister(2, 'cr')
   ckt = QuantumCircuit(qs, cr)
   # initialization to |11>
   ckt.x(qs[0])
   ckt.x(qs[1])
   # end init
   ckt.h(qs[1])
   ckt.sdg(qs[1])
   ckt.cx(qs[0], qs[1])
   ckt.h(qs[1])
   ckt.t(qs[1])
   ckt.cx(qs[0], qs[1])
   ckt.t(qs[1])
   ckt.h(qs[1])
   ckt.s(qs[1])
   ckt.x(qs[1])
   ckt.s(qs[0])
   ckt.measure(qs, cr)
   return ckt
```

Listing 2. QISKit Controlled-Hadamard

This circuit would be run using the 'qasm_backend' executor for 100 shots and would output a result that would provide counts for each probabilistic result that would show approximately half the shots producing the value 10 and half the shots producing 11. These reflect

1 Program output has been edited to accommodate the needs of publication.

DOI: https://doi.org/10.30564/jcsr.v3i4.3567

the actual mathematical result $\mathrm{CtlH}(|11\rangle) = \frac{1}{\sqrt{2}}(|10\rangle - |11\rangle)$, so one could assume that the sequence of gates was equivalent to a controlled-Hadamard gate.

Listing 3 shows the controlled-Hadamard computation expressed in QuTIP-qip [6], the Quantum Toolbox in Python component for quantum information processing. The component has a circuit simulator, and the computation looks very much like QISKit. It has facilities to display the equivalent matrix, shown at the end of the listing. Compare this to Listing 1. QuTIP is an extensive package going far beyond circuit simulation and is an excellent tool for the advanced quantum computing student, worthy of study on its own.

```
def sdg_gate2():
    # S adjoint gate
    mat = np.array([[1., 0],
                    [0, -1.j]])
    return Qobj(mat, dims=[[2], [2]])
def ctl_h():
    q = QubitCircuit(2, reverse_states=False)
    q.user_gates = {'SDG': sdg_gate2}
    q.add_gate('SNOT', targets=[1])
    q.add_gate('SDG', targets=[1])
    q.add_gate("CNOT", controls=[0], targets=[1])
    q.add_gate('SNOT', targets=[1])
    q.add_gate('T', targets=[1])
    q.add_gate("CNOT", controls=[0], targets=[1])
    q.add_gate('T', targets=[1])
    q.add_gate('SNOT', targets=[1])
    q.add_gate('S', targets=[1])
    q.add_gate('X', targets=[1])
    q.add_gate('S', targets=[0])
    return q
Quantum object: dims = [[2, 2], [2, 2]],
shape = (4, 4), type = oper, isherm = False
Qobj data =
[[ 0.7+0.7j  0. +0.j   0.  +0.j   0.  +0.j  ]
 [ 0. +0.j   0.7+0.7j  0.  +0.j   0.  +0.j  ]
 [ 0. +0.j   0. +0.j   0.5 +0.5j  0.5 +0.5j ]
 [ 0. +0.j   0. +0.j   0.5 +0.5j -0.5 -0.5j ]]
```

Listing 3. QuTIP-qip Controlled-Hadamard

The language Q# [7] was used as a teaching tool, described in "Teaching Quantum Computing through a Practical Software-driven Approach: Experience Report" [3]. A Q# implementation of the controlled-Hadamard computation would look very much like Listing 2 and Listing 3. It provides another approach but is an additional learning burden.

Although quantum computing is a relatively new computer science subfield, there are many software systems to aid in quantum computation, most of which are open source. ProjectQ [8] is a compiler framework capable of targeting various types of hardware, containing a high-performance simulator with emulation capabilities, based on a Python-embedded domain-specific language. Toqito [9] is a Python library for studying various objects in quantum information, states, channels, and measurements. QuNetSim [10] is a quantum network simulation framework. Interlin-q [11] is a simulation platform for simulating distributed quantum algorithms. Cirq [12] is a Python library for writing, manipulating, and optimizing quantum circuits and running them against quantum computers and simulators. QRAND [13] is a smart quantum random number generator for arbitrary probability distributions, which operates by providing a multiplatform NumPy adapter interface. Qrack [14] is a GPU-accelerated HPC quantum computer simulator framework. Pulser [15] is a Python library for programming neutral-atom quantum devices at the pulse level. QCOR [16] is a quantum-retargetable compiler platform providing language extensions for both C++ and Python that allows programmers to express quantum code as stand-alone kernel functions. XACC [17] is a service-oriented, system-level software infrastructure in C++ promoting an extensible API for the typical quantum-classical programming, compilation, and execution workflow. Yao [18] is a framework that aims to empower quantum information research with software tools in the Julia programming language. Quantify [19] is a Python-based data acquisition platform focused on quantum computing and solid-state physics experiments.

3. QQCS

3.1 Quantum Circuits, Briefly

Quantum programs are often constructed and displayed as quantum circuit diagrams. As shown in Figure 1, circuit diagrams are stacked horizontal lines with various connections between them. Each horizontal line represents a qubit. A line is also called a wire, but it is only a wire conceptually. The qubit it represents may be physically realized in several different ways by a quantum computer. The lines are read from left to right corresponding to the sequential execution of the circuit and are best thought of as representing movement in time. Elements that are vertically aligned in the circuit are considered to happen simultaneously. Gates are labeled rectangles, named for the type of gate. Measurement sets a classical bit from a qubit. Measurement is indicated in a quantum circuit by a meter symbol, and usually appears at the end of the circuit. The double wire exiting a meter indicates that the line now carries a classical bit, not a qubit. There are

 DOI: https://doi.org/10.30564/jcsr.v3i4.3567

a small number of additional conventions. A controlled gate, such as a controlled NOT, has a control qubit and a target qubit, and is represented not by a rectangle but by a vertical line from the control qubit, indicated by the black dot at the line intersection, to the target qubit, indicated by the $\oplus$ at the intersection. The third and sixth gates in Figure 1 are controlled NOT gates, with controls on line 0 and targets on line 1. A Toffoli gate, a three-qubit controlled gate with two controls and one target, is represented the same way; the control intersections have black dots, and the target intersection has a $\oplus$. A Toffoli gate is shown in Figure 4c.

3.2 Gate Linear Notation

A quantum circuit is a sequence of gates, and as the number of qubits increases, the options for placing and connecting the gates increases, too. Most gate placements, however, are of only a few varieties. The simulation's available gates are one-qubit gates named with one and two letter abbreviations, which are then augmented with prefixes and suffixes describing their positions within the circuit. Additional conventions provide for multiple gates in a single time slice, and for arbitrary control and target lines anywhere within a ten-qubit circuit. The full syntax is shown in Appendix A.

The basic gate names are shown in Table 1 [20,21].

In the linear notation, a gate name is preceded by a colon (:) character.

3.3 Rotational Gates

All the rotational gates specify the angle parameters as factors of π radians, with π implicit. Thus, Rx(.5) is an X-axis rotation of $\pi/2$ radians, or 90 degrees. The parameter range for all angles is (0,4).

The U gate may have one, two, or three parameters: i) $U(\lambda) = U(0,0,\lambda)$, ii) $U(\phi,\lambda) = U(\pi/2,\phi,\lambda)$, or iii) $U(\theta,\phi,\lambda)$. The three-parameter version implements the general unitary matrix:

$$\begin{pmatrix} e^{-i(\phi+\lambda)/2}\cos(\theta/2) & -e^{-i(\phi-\lambda)/2}\sin(\theta/2) \\ e^{i(\phi-\lambda)/2}\sin(\theta/2) & e^{i(\phi+\lambda)/2}\cos(\theta/2) \end{pmatrix}$$

The Rx(θ) gate is equivalent to $U(\theta,-\pi/2,\pi/2)$.

The Ry(θ) gate is equivalent to $U(\theta,0,0)$.

The Rz(λ) gate is equivalent to $U(0,0,\lambda)$.

An alternate general unitary definition is available, invoked by the -u command line flag, or the $ualt comment flag. The alternate definition differs only by a phase factor from the default definition above, but it can simplify the elements of some rotational gates. The definition is:

$$\begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\phi+\lambda)}\cos(\theta/2) \end{pmatrix}$$

Table 1. The Basic 1-, 2-, and 3-qubit Gate Names

Names	Gate Description
1-Qubit	
H	Hadamard gate
I	Identity gate
_	ungated lines (implied Identity)
Kp(θ)	Phase gate (universal set) [21]
Rp(θ)	Rotation gate (universal set) [21]
Rx(θ)	Pauli X rotation gate
Ry(θ)	Pauli Y rotation gate
Rz(θ)	Pauli Z rotation gate
S	S gate ($\pi/2$ phase gate)
Sa	S adjoint
T	π gate ($\pi/4$ phase gate)
Ta	T adjoint
Tp(θ)	Phase rotation gate (universal set) [21]
U(θ,ϕ,λ)	Universal one-, two-, or three-parameter rotation gate
X	Pauli X gate
Y	Pauli Y gate
Z	Pauli Z gate (π phase gate)
2-Qubit	
C	general CNOT (used with a 2-digit control suffix)
Cx	CNOT with control qubit *q* and target qubit *q+1*
Cr	reverse CNOT with control qubit *q+1* and target qubit *q*
Sw	General swap (used with a 2-digit control suffix)
3-Qubit	
Tf	general Toffoli gate (used with a numerical suffix)
Fr	general Fredkin gate (used with a numerical suffix)
n-Qubit	
Im	Mean Inversion (used with 1-digit size suffix)
Qf	Quantum Fourier Transform (used with 1-digit size suffix)
Qa	QFT adjoint (used with 1-digit size suffix)

3.4 Oracles

Table 2. Oracles

Oracles	specified with 1-digit size suffix, and optional parameters
Ob	Bernstein-Vazirani
Od	Deutsch-Jozsa
Os	Simon
Og	Grover

Oracles are available for the well-known algorithms of Deutsch, Deutsch and Josza, Bernstein and Vazirani, Simon, and Grover.

The oracles are specified with the syntax :Ox(p)n.

DOI: https://doi.org/10.30564/jcsr.v3i4.3567

x is set to *d* for Deutsch and Deutsch-Josza, which are distinguished by their qubit size, *b* for Bernstein-Vazirani, *s* for Simon, and *g* for Grover. The optional parameter *p* is specific to the oracle and determines whether the oracle will implement a random function or a function determined by the parameter. The *n* suffix is the qubit size and must be specified. The qubit size includes any ancilla qubits.

:Od2 is considered the Deutsch oracle, and any larger qubit size is the Deutsch-Josza oracle. Both algorithms use a single ancilla qubit, and the random function is either a constant or balanced binary function of domain size *n-1*. If the optional parameter is specified, a value of 0 generates a constant function whose values are all 0. A value of 1 generates a constant function whose values are all 1. Any other value generates a balanced function.

:Obn is the Bernstein-Vazirani oracle. The algorithm uses a single ancilla qubit, and the function implements a hidden binary string of size *n-1*. If the optional parameter is specified, it determines the hidden string and should be a value between 0 and $2^n - 1$.

:Osn is the Simon oracle. The algorithm uses *n/2* ancilla qubits, and the function implements a binary string of size *n/2* representing the "period" [22] of the function, which is discovered by the Simon algorithm. If the optional parameter is specified, it determines the "period" and should be a value between 0 and $2^{n/2} - 1$.

:Ogn is the Grover oracle. The oracle randomly selects one basis vector from its n-qubit input and changes its phase to the opposite sign. If the optional parameter is specified, it determines the basis vector to be changed and should be a value between *0* and $2^n - 1$.

3.5 Permutation Gates

Permutation gates are matrices with a single 1 in each row and column and 0's in all other elements. The CNOT gate is a typical permutation gate. When applied to a quantum state, permutation gates shift the amplitudes from one basis vector to another. A permutation gate is specified with the syntax :P(pair, ...)n. Each pair is syntactically real number, but it is interpreted as a pair of integers separated by a period. The *n* suffix is the qubit size of the gate. The integers in a pair must be in the domain 0 to $2^n - 1$. For example, the number 2.6 is taken as the pair 2→6, referencing the basis vectors $|010\rangle$ and $|110\rangle$. The gate :P(2.6,6.4,4.2)4 will cycle the amplitudes of three basis vectors in a four-qubit circuit. A two-qubit CNOT gate is equivalent to the permutation specification :P(2.3,3.2)2. The QQCS specification of a large permutation gate is tedious. The simplest way to use one is to specify it once and assign it to a custom gate, then reuse the custom gate as needed.

3.6 Positioning and Replicating Gates

When a gate is positioned in a circuit, it may have qubit lines above and/or below on which there are no gates. The Identity gate is implied when no gate is specified. To indicate this, QQCS uses an underscore (_), repeated once for each ungated qubit line. If the gate is replicated on several circuit lines, the gate name can be repeated, or the replication can be abbreviated with a digit. :H_ is a Hadamard gate on qubit 0, with no gate on qubit 1. To place the Hadamard gate on qubit line 1, use :_H. See Figure 2. The _ can be repeated as many times as needed. :_____H is a one-qubit Hadamard gate on line 5 of a six-qubit circuit. :____H_ moves the Hadamard gate up to line 4.

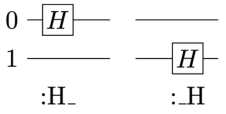

Figure 2. Gate Positioning

To place a gate across multiple qubit lines, follow its name with a digit replicator suffix. To transform a $|0000\rangle$ initial value in a four-qubit circuit to a balanced superposition, use :HHHH or :H4 as a four-qubit Hadamard gate on lines 0 through 3, shown in Figure 3a. The replicator suffix is applicable only to one-qubit gates.

In instances where several gates appear on non-adjacent qubit lines, and are therefore executed simultaneously, the gates can be listed in sequence. If there are implied identity gates between some gates, use one or more underscores. To put an X-gate on lines 1 and 3 of a four-qubit circuit, use :_X_X, as shown in Figure 3b.

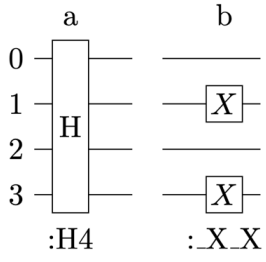

Figure 3. Positioning in a 4-Qubit Circuit

3.7 Controlled Gate Names

Qubit line numbers on a controlled gate can be relative to the span of the gate, or absolute.

For relative line numbers, the span of a gate is the difference between the minimum and maximum control/target lines plus one. Reading left to right, controls occur

 DOI: https://doi.org/10.30564/jcsr.v3i4.3567

first, then the target, each as a single digit. As an example, the control suffix 02 has a span of 3 *(2-0+1=3)* lines and indicates a control on relative line 0 and a target on relative line 2.

Ungated prefixes and suffixes are used, as in all other gates, to position the gate vertically in the circuit. Lines within the span that are not control or target lines are ungated by implication.

To place controlled gates in a circuit, start as if the gate were placed at line 0, then identify the controls and the target, in that order. A controlled NOT gate with a three-qubit span with the control on line 2 and the target on line 0 is :C20. To reverse the control and target, use :C02. See Figure 4. If the gate needs to be positioned within a larger qubit circuit, use leading underscores to shift it. The control and target numbers are with relative to the span of the gate, not the number of qubit lines in the circuit. This means that if the gate spans 4 qubits, the lines within the span are referenced from 0 to 3, regardless of the position. The controlled NOT gate :_C02 is shown in Figure 4b. The common names :Cx (equivalent to :C01) and :Cr (equivalent to :C10) are also available for CNOT and reverse CNOT gates with a span of 2.

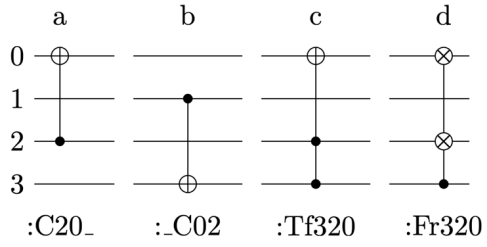

Figure 4. Controlled Gate Linear Notation

Gates that involve two or more control lines and one target, or one control and two targets, will have three (or more) digits following the gate name, in the order (control1, control2, target), or (control, target1, target2), as shown in Figure 4c and 4d. Again, the control and target line numbers are relative to the span.

Any of the built-in one-qubit gates can be supplied with a 2-digit control suffix to add a control line to the gate. See Figure 5 showing a three-qubit quantum Fourier equivalent circuit [20,22], using several different controlled gate forms. The final gate is a swap between lines 0 and 2.

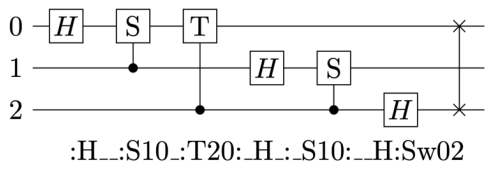

Figure 5. 3-qubit Quantum Fourier Transform

With an absolute suffix, the control and target digits indicate the actual lines of the circuit. No leading _'s are needed for positioning. Trailing _'s may still be needed to indicate the full qubit size of the gate. Either absolute or relative notation may be used.

A Toffoli gate may have more than two control qubits.

3.8 Display

A circuit simulation display starts with the initial value if it is a quantum state and ends with the resulting quantum state at the end of the circuit. By default, both displays are row vectors. Internally, quantum states are column vectors, but they are transposed for linear display. To see the balanced superposition result of a Hadamard gate across three qubits, enter the circuit sequence in Listing 4 and press *Enter*. The result is shown following the entry[2].

```
(1)   |000>:H3
[1  0   0  0  0  0  0  0]       H3
[ 0.354   0.354  0.354  0.354
0.354  0.354  0.354  0.354 ]
```

Listing 4. Balanced Superposition

Listing 4 Explanation

(1) The user enters a three-qubit initial value, and the three-qubit Hadamard gate. The following output display is the initial value as a transposed vector, the gate sequence, and the final quantum state, which shows the balanced superposition.

If the gate sequence does not start with an initial value, the display is an empty initial value and the ending gate matrix. The ending matrix is the matrix product of all the gates in the circuit. In Listing 5, there is only a single two-qubit Hadamard gate.

```
(1)  :H2
[ ]  H2  [
0.5     0.5     0.5     0.5
0.5    -0.5     0.5    -0.5
0.5     0.5    -0.5    -0.5
0.5       -0.5    -0.5     0.5]
```

Listing 5. 2-Qubit Hadamard Gate

Listing 5 Explanation

(1) The user enters a two-qubit Hadamard gate. The following output display is the gate itself.

2 In Listings, line wraps and indentation are artificial for format purposes

DOI: https://doi.org/10.30564/jcsr.v3i4.3567

3.9 Comments

An input comment is anything following a '#' character to the end of the line. A comment is also searched for switch specifications beginning with the character '$'. Keywords following the '$' set or unset internal options, most of which correspond to command line flags.

3.10 Measurement

If simple quantum state display is not sufficient, the results can also be measured and displayed. Measuring produces a probability display for each non-zero basis vector component of the result. Measurement is specified by the M pseudo-gate. M is not an actual gate, but it can be placed anywhere in the circuit gate sequence just as if it were a gate. Unlike measurement in an actual quantum computer, measurement in the simulation does not collapse the quantum state. It is designed for trace purposes. It can occur in a circuit any number of times. If the measurement is only to be applied to some subset of the qubits, specify it exactly as if M were a gate, with underscore prefixes, suffixes, infixes. The controlled-H sequence of Figure 1 is shown with three additional measuring points in Listing 6.

```
(1) |10>:_H:M_:_Sa:Cx:_H:M_:_T
:Cx:_T:_H:_S:_X:S_:M2
M1={1:1}
M2={1:1}
M3={10:0.5,  11:0.5}
[0  0  1  0] _H  _Sa  Cx  _H  _T  Cx  _T  _H
_S  _X  S_ [0  0  0.5+0.5i  0.5+0.5i]
```

Listing 6. Measurement

Listing 6 Explanation

(1) The gate sequence has internal two measurements for qubit line 0, and a full measurement of both lines as the final gate.

[M1] The output of the first measurement, measuring only the first qubit, shows a probability of 1 for the qubit value 1.

[M2] The output of the second measurement also shows a probability of 1 for the qubit value 1.

[M3] The final measurement shows a probability of .5 for each of $|10\rangle$ and $|11\rangle$.

[last] The output is the initial state, the gate sequence, and the final quantum state.

Note that the first two measurements only measure the qubit on line 0. The last measurement, at the end, is for both qubits. The measurement outputs are sequentially numbered so they can be distinguished, and the results are enclosed in braces indicating that it is not a quantum state. The measurement output is a list of measurement outcomes and the probability of each. Even when measuring fewer qubits than are in the circuit, the probabilities will always add to one. The final measurement in Listing 6 shows the probabilities, but the final quantum state shows that the probabilities arise from interesting basis coefficients.

3.11 Initial Values

Quantum circuits are generally assumed to start with an initial value of $|0\rangle_n$ where n is the number of qubits. QQCS uses the presence or absence of an initial value to distinguish between displays. If an initial value is present at the beginning of a circuit, the ending display will be the ending quantum state. If there is no initial value, the ending display will be the equivalent gate matrix. An initial value syntactically is quantum state, a sum of basis kets with complex coefficients. Listing 7 shows four interactions with QQCS in which a Hadamard gate operates on initial values of $|0\rangle$, $|1\rangle$, $0.707|0\rangle + 0.707|1\rangle$, and $0.707|0\rangle - 0.707|1\rangle$. It is an illustration of measurement in the Hadamard basis.

```
(1)  |0>:H
[1  0]  H  [0.707  0.707]
(2)  |1>:H
[0  1]  H  [0.707  -0.707]
(3)  0.707|0>+0.707|1>:H
[0.707  0.707]  H  [1  0]
(4)  0.707|0>-0.707|1>:H
[0.707  -0.707]  H  [0  1]
```

Listing 7. Measurement in the Hadamard Basis

Listing 7 Explanation

(1) User enters $|0\rangle$ followed by a Hadamard gate. The output is the initial state $|0\rangle$ (as a transposed column vector), followed by the gate sequence, followed by the final state.

(2) The same sequence with an initial value of $|1\rangle$.

(3) The initial value is $1/\sqrt{2}(|0\rangle + |1\rangle)$, which is $|0\rangle$ in the Hadamard basis, as the following H transformation shows.

(4) Complete the example by showing $|1\rangle$ in the Hadamard basis.

3.12 Tensor Products

An initial value can be constructed from a tensor product. If more than one quantum state is entered as an

 DOI: https://doi.org/10.30564/jcsr.v3i4.3567

initial value and the states are parenthesized, a tensor product is implied. This is shown in Listing 8. As in all other QQCS interactions, the first value displayed is that of the first operand. The last value displayed is the result of the computation.

```
(1)  (|0>) (|1>)
[1  0]   [0  1  0  0]
(2)  (0.707|0>+0.707|1>)
(0.707|0>-0.707|1>)
[0.707   0.707]  [0.5  -0.5  0.5  -0.5]
(3)  (0.707|0>+0.707|1>)
(0.5|00>-0.5|01>+0.5|10>-0.5|11>)
[0.707   0.707]
[0.354    -0.353    0.354    -0.353
0.354     -0.353    0.354    -0.353]
```

Listing 8. Tensor Products

Listing 8 Explanation

(1) The tensor product $|0\rangle \otimes |1\rangle$

(2) The tensor product $1/\sqrt{2}(|0\rangle + |1\rangle) \otimes 1/\sqrt{2}(|0\rangle - |1\rangle)$

(3) The tensor product $1/\sqrt{2}(|0\rangle + |1\rangle) \otimes 1/2(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$

3.13 Factoring

The Quantum Fourier Transform circuit in Figure 5 will display the matrix shown in Listing 9.

```
0.354   0.354          0.354       0.354          ...
0.354   0.25+0.25i     0.354i      -0.25+0.25i    ...
0.354   0.354i         -0.354      -0.354i        ...
0.354   -0.25+0.25i    -0.354i     0.25+0.25i     ...
0.354   -0.354         0.354       -0.354         ...
0.354   -0.25-0.25i    0.354i      0.25-0.25i     ...
0.354   -0.354i        -0.354      0.354i         ...
0.354   0.25-0.25i     -0.354i     -0.25-0.25i    ...
..
```

Listing 9. QFT With No Factoring

In texts [22], the n-qubit QFT is the matrix

$$\frac{1}{\sqrt{N}}\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{1} & \omega^{2} & \dots & \omega^{N-1} \\ 1 & \omega^{2} & \omega^{4} & \dots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{pmatrix}$$

where $N = 2^n$ and ω is a primitive N'th root of unity.

By factoring out $\frac{1}{\sqrt{8}}$ (0.35355) using the suffix operator (/) at the end of the circuit, it is easier to see that the result of the circuit is the three-qubit QFT, as in Listing 10.

```
[2]  :H__:S10_:T20:_H_:_S10:__H
:C02:C20:C02/0.35355
1   1                  1          1                  ...
1    0.707+0.707i      1i         -0.707+0.707i      ...
1    1i                -1         -1i                ...
1   -0.707+0.707i      -1i        0.707+0.707i       ...
1   -1                 1          -1                 ...
1   -0.707-0.707i      1i         0.707-0.707i       ...
1   -1i                -1         1i                 ...
1    0.707-0.707i      -1i        -0.707-0.707i      ...
..
```

Listing 10. QFT Factored

4. Examples

4.1 Oracles for the Grover Search

The two-qubit Grover Search tries to determine the phase encoding of the input quantum state. The algorithm uses a black box circuit, called an Oracle, to initially change the phase of one of the basis kets in a balanced superposition two-qubit quantum state. It then uses inversion to the mean to amplify the phase difference before a final measurement. Listing 11 shows four oracle gate sequences to change the phase of the input in each of the possible ways, that are alternatives to the built-in QQCS :Og gate.

```
(1)  |00>:H2:_H:C01:_H
[1  0  0  0]  H2 _H  C01  _H
[0.5  0.5  0.5  -0.5]
(2)  |00>:H2:S_:_H:C01:_H:S_
[1  0  0  0]  H2  S_  _H  C01 _H  S_
[0.5  0.5  -0.5  0.5]
(3)  |00>:H2:_S:_H:C01:_H:_S
[1  0  0  0]  H2  _S  _H  C01  _H  _S
[0.5  -0.5  0.5  0.5]
(4)  |00>:H2:Y2:_H:C01:_H:Y2
[1  0  0  0]  H2  Y2  _H  C01  _H  Y2
[-0.5  0.5  0.5  0.5]
```

Listing 11. Four Oracles For Grover Search

Listing 11 Explanation

(1) Change the phase of $|11\rangle$.

(2) Change the phase of $|10\rangle$

(3) Change the phase of $|01\rangle$.

(4) Change the phase of $|00\rangle$.

DOI: https://doi.org/10.30564/jcsr.v3i4.3567

4.2 Modular Arithmetic Subroutine

The implementation of Shor's algorithm[23] needs a subroutine circuit to compute $g(x) = (2x) \bmod 15$, from 1 to the modulus minus 1. Instead of a straightforward, and inefficient, implementation, the book presents a simple qubit swapping circuit based on the bit patterns of the modulo computation. Listing 12 shows the gates of the swap circuit assigned to name, ss, which then can be used like any other gate. The named gate is applied to the four-qubit values from $|0001\rangle$ to $|1110\rangle$. The result shows the resulting quantum state of each of the 16 inputs, which in each case is a single basis ket whose binary value equals that of the $g(x)$ calculation.

```
(1) #$rzeroes
(2)  ss:C03:C30:C03:C01__
:C10__:C01__:_C01_:_C10_:_C01_
[]  C03 C30 C03 C01__
C10__ C01__ _C01_ _C10_ _C01_=
1 . . . . . . . . . . . . . . .
. . . . . . . . 1 . . . . . . .
. 1 . . . . . . . . . . . . . .
. . . . . . . . . 1 . . . . . .
. . 1 . . . . . . . . . . . . .
. . . . . . . . . . 1 . . . . .
. . . 1 . . . . . . . . . . . .
. . . . . . . . . . . 1 . . . .
. . . . 1 . . . . . . . . . . .
. . . . . . . . . . . . 1 . . .
. . . . . 1 . . . . . . . . . .
. . . . . . . . . . . . . 1 . .
. . . . . . 1 . . . . . . . . .
. . . . . . . . . . . . . . 1 .
. . . . . . . 1 . . . . . . . .
. . . . . . . . . . . . . . . 1
(3)  |0001>:ss
[|0001>]
ss [|0010>]
(4) |0010>:ss
[|0010>]
ss [|0100>]
. . .
(9)  |0111>:ss
[|0111>]
ss [|1110>]
(10) |1000>:ss
[|1000>]
ss [|0001>]
(11)  |1001>:ss
[|1001>]
ss [|0011>]
. . .
(15)  |1101>:ss
[|1101>]
ss [|1011>]
(16)  |1110>:ss
[|1110>]
ss [|1101>]
```

Listing 12. Shor's Algorithm Subroutine

Listing 12 Explanation

(1) set a switch to display zeros as periods

(2) Assign the swap circuit to the custom gate name **ss.** The following display is the circuit's equivalent gate matrix.

(3) Invoke the :**ss** gate circuit on the initial value $|0001\rangle$. The output that follows displays the quantum state initial value in ket format, then the gate name, then the quantum state resulting from the execution of the gate, $|0010\rangle$ again in ket format.

(4) Shows that $|0010\rangle \mapsto |0100\rangle$.

(…) …

(16) Shows that $|1110\rangle \mapsto |1101\rangle$.

4.3 The Bernstein-Vazirani Oracle

The Bernstein-Vazirani algorithm [24] can be written in QQCS as shown in Listing 13.

```
(1)  | 0 0 0 1>:H4:Ob4:H3_:M3_
M1={010:1}
[ | 0 0 01>]  H4  Ob4  H3_
0.707|0100>-0.707|0101>
(2) |0001>:H4:Ob(3)4:H3_:M3_
M1={011:1}
[ | 0 0 0 1>]  H4  Ob(3) 4  H3_
0.707|0110>-0.707|0111>
```

Listing 13. The Bernstein-Vazirani Algorithm

Listing 13 Explanation

(1) the Bernstein-Vazirani circuit; the oracle generates a random hidden string, which the measure shows as 010.

(2) a version of the Bernstein-Vazirani circuit in which the oracle's hidden string is set by the parameter to 3; the measure shows 011.

5. Conclusions

QQCS is a simple linear notation for the simulation of quantum circuits. It is an educational tool that can be easily used by students new to Quantum Computing.

 DOI: https://doi.org/10.30564/jcsr.v3i4.3567

It provides automatic mathematical analysis of circuits by incorporating the matrix mathematics necessary to provide insight into circuit operation, and by displaying the details at each execution step, something not available from quantum computer execution.

Installation

QQCS is installed with the Node Package Manager. First, install NodeJS. Then, at the command line, enter:

npm install qqcs

To run, go to the node_modules directory, and enter:

node qqcs -or- node qqcs/qdesk.js

Use the command line switch -h to get help.

Acknowledgement

The circuit diagrams in this paper were constructed using the QPIC software package [25].

References

[1] A. Cross, L. Bishop, J. Smolin and J. and Gambetta, "Open Quantum Assembly Language," 2017. [Online]. Available: https://arxiv.org/pdf/1707.03429.pdf.

[2] IBM, "IBM Quantum Experience," 2019. [Online]. Available: http://quantumexperience.ng.bluemix.net/.

[3] M. Mykhailova and K. M. Svore, "Teaching Quantum Computing through a Practical Software-driven Approach: Experience Report," in SIGCSE '20: Proceedings of the 51st ACM Technical Symposium on Computer Science Education, 2020.

[4] B. Heim, M. Soeken, S. Marshall, C. Granade, M. Roetteler, A. Geller, M. Troyer and K. Svore, "Quantum Programming Languages," Nat Rev Phys 2, pp. 709-722, 2020.

[5] QISKit, "The QISKit SDK for quantum software development," 2019. [Online]. Available: https://github.com/QISKit.

[6] QuTIP, "Quantum Toolbox In Python," 2019. [Online]. Available: http://qutip.org/.

[7] Microsoft, "The Q# User Guide," 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/quantum/user-guide/.

[8] D. S. Steiger, T. Häner and M. Troyer, "ProjectQ: An Open Source Software Framework for Quantum Computing," 2018. [Online]. Available: https://arxiv.org/abs/1612.08091.

[9] Toqito, "toqito - a Python library for studying various objects in quantum information: states, channels, and measurements," 2021. [Online]. Available: https://github.com/vprusso/toqito.

[10] QuNetSim, "QuNetSim -a quantum network simulation framework," 2021. [Online]. Available: https://github.com/tqsd/QuNetSim.

[11] Interlin-q, "Interlin-q - a simulation platform for distributed quantum algorithms," 2021. [Online]. Available: https://github.com/Interlin-q/Interlin-q.

[12] Cirq, "Cirq - a Python library for writing, manipulating, and optimizing quantum circuits and running them against quantum computers and simulators," 2021. [Online]. Available: https://github.com/quantumlib/cirq.

[13] QRAND, "QRAND - a smart quantum random number generator for arbitrary probability distributions," 2021. [Online]. Available: https://github.com/pedror-rivero/qrand.

[14] Qrack, "Qrack - a GPU-accelerated HPC quantum computer simulator framework," 2021. [Online]. Available: https://github.com/vm6502q/qrack.

[15] Pulser, "Pulser - a Python library for programming neutral-atom quantum devices at the pulse level," 2021. [Online]. Available: https://github.com/pasqal-io/Pulser.

[16] QCOR, "QCOR - a quantum-retargetable compiler platform providing language extensions for both C++ and Python that allows programmers to express quantum code as stand-alone kernel functions," 2021. [Online]. Available: https://github.com/ornl-qci/qcor.

[17] XACC, "XACC - a service-oriented, system-level software infrastructure in C++ promoting an extensible API for the typical quantum-classical programming, compilation, and execution workflow," 2021. [Online]. Available: https://github.com/eclipse/xacc.

[18] Yao, "Yao - a framework that aims to empower quantum information research with software tools in the Julia programming language," 2021. [Online]. Available: https://github.com/QuantumBFS/Yao.jl.

[19] Quantify, "Quantify - a Python based data acquisition platform focused on Quantum Computing and solid-state physics experiments," 2021. [Online]. Available: https://gitlab.com/quantify-os.

[20] M. A. Nielsen and I. L. and Chuang, Quantum Computation and Quantum Information 10th Anniversary Ed, New York: Cambridge University Press, 2010.

[21] E. Rieffel and W. Polak, Quantum Computing, A Gentle Introduction, Cambridge: MIT Press, 2011.

[22] R. S. Sutor, Dancing With Qubits, Birmingham: Packt Publishing, 2019.

[23] C. C. Moran, Mastering Quantum Computing with IBM QX, Birmingham: Packt Publishing, 2019.

[24] J. Abhijith and e. al, "Quantum Algorithm Implementations for Beginners," 2020. [Online]. Available: https://arxiv.org/pdf/1804.03719.pdf.

DOI: https://doi.org/10.30564/jcsr.v3i4.3567

[25] QPIC, "QPIC (2018) Creating quantum circuit diagrams in TikZ," 2018. [Online]. Available: https://github.com/qpic/qpic.

Appendix A

Linear Notation Syntax

Meta-symbols

::= is defined as
| alternative
e empty
'x' x is a grammar symbol, not a meta-symbol

Grammar

```
          Pgm   ::=  stmt  stmt-list  eof
    stmt-list   ::=  eol  stmt  stmt-list | e
         stmt   ::=  ident  gate-sequence |
                     initial-value  gate-sequence
initial-value   ::=  q-state | q-state-list | e
gate-sequence   ::=  g-seq-tail  g-factor
   g-seq-tail   ::=  : gates  g-seq-tail | e
     g-factor   ::=  / unop  Complex | e
  q-state-list  ::=  ( q-state )  q-state-list | e
      q-state   ::=  unop v-comp  p-state-tail
 p-state-tail   ::=  addop v-comp  p-state-tail | e
        gates   ::=  full-gate  gates | e
    full-gate   ::=  gate  gate-suffix | ident
  gate-suffix   ::=  gate-angle  gate-repl
  gate-angle    ::=  ( unop Real reals ) | e
   gate-repl    ::=  integer | e
        reals   ::=  , unop Real reals | e
      v-comp    ::=  coeff ket
       coeff    ::=  Complex | e
         ket    ::=  '|' integer >
     Complex    ::=  complex | Real
        Real    ::=  real | integer
       addop    ::=  + | -
        unop    ::=  - | e
```

 DOI: https://doi.org/10.30564/jcsr.v3i4.3567