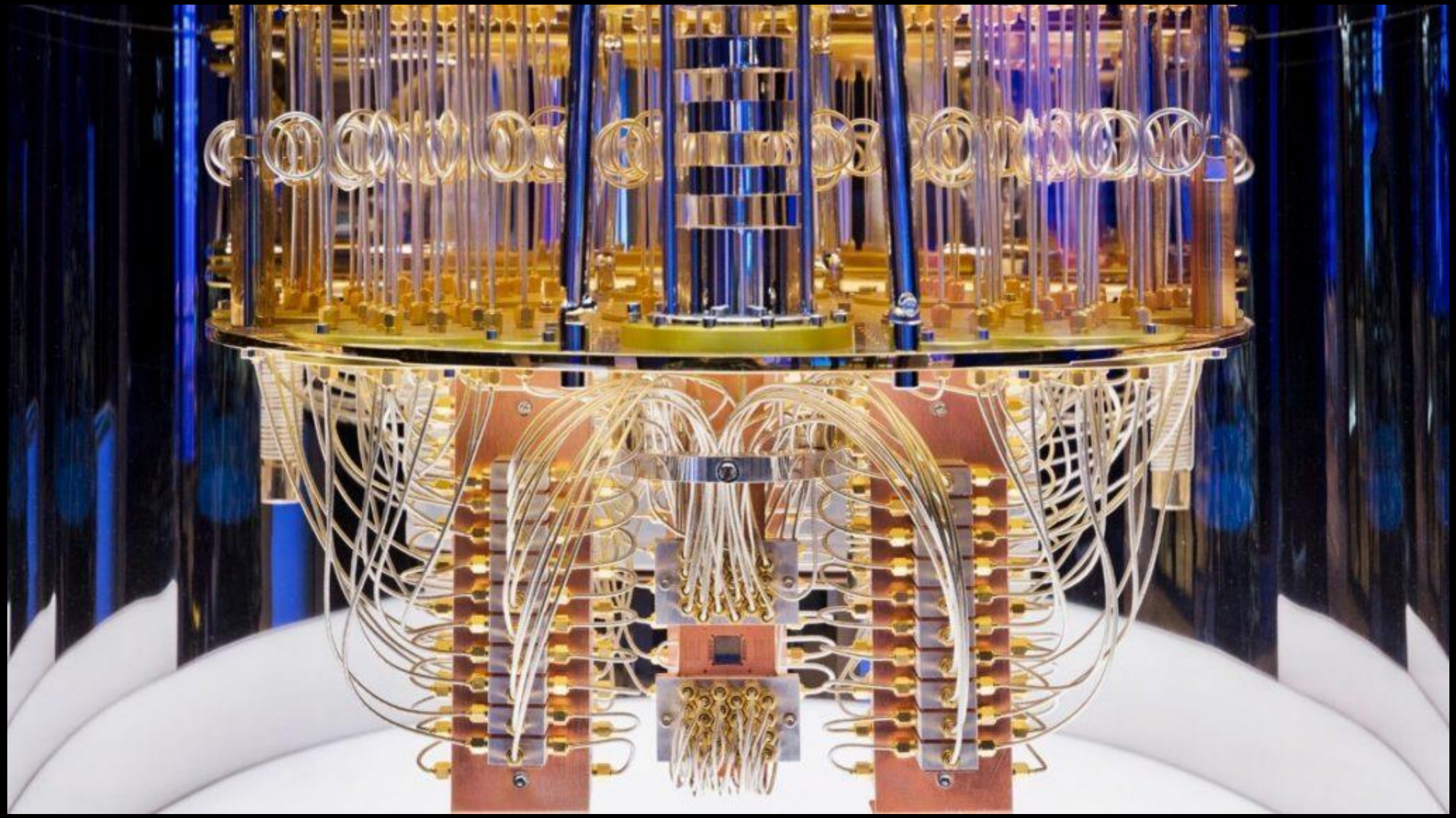
Simulating and Visualizing Quantum Circuits

Michael McGuffin, Professor
École de Technologie Supérieure (ETS),
2023-11-24

Quantum Computers: IBM



Quantum Computers: IBM

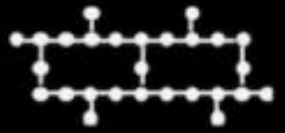
2019
Falcon
27 Qubits

2020
Hummingbird
65 Qubits

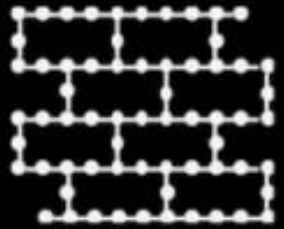
2021
Eagle
127 Qubits

2022
Osprey
433 Qubits

Falcon:
27 qubits

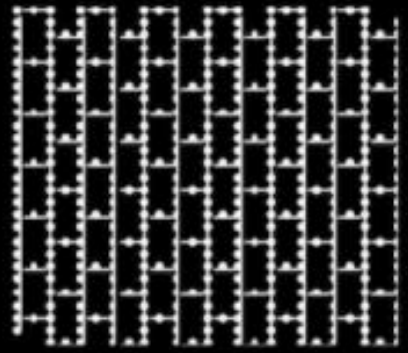


Hummingbird:
65 qubits

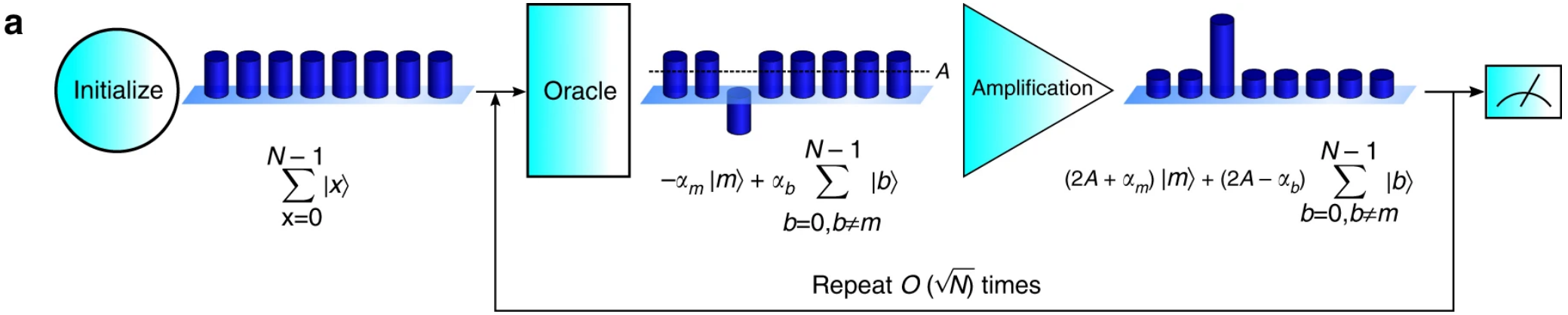


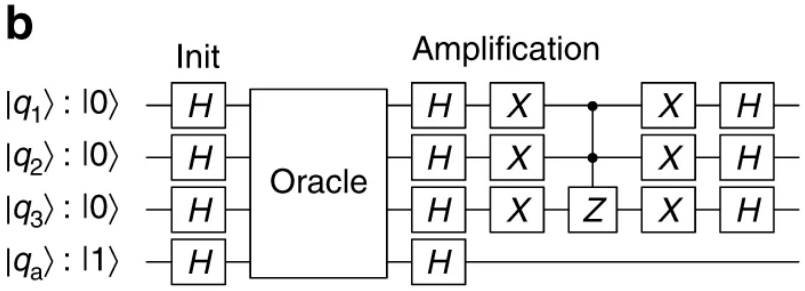
Eagle:
127 qubits

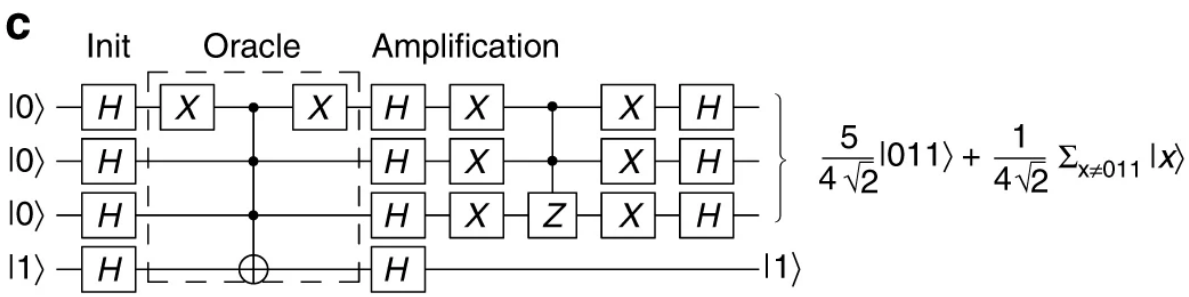
Osprey:
433 qubits

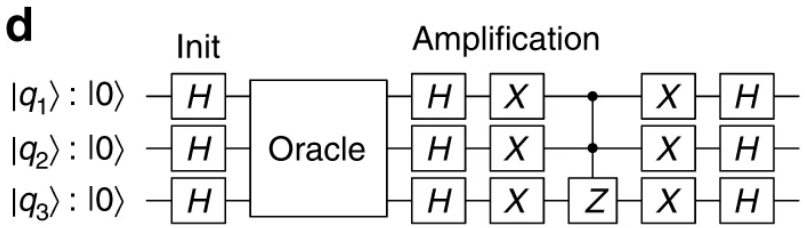


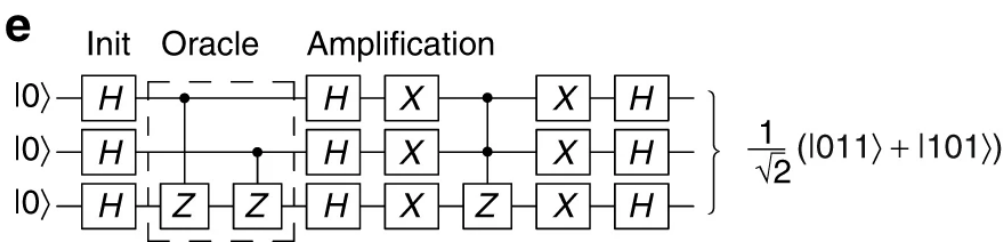
Grover's algorithm: search over N items in $O(N^{1/2})$ time instead of $O(N)$ time, yielding a quadratic speedup





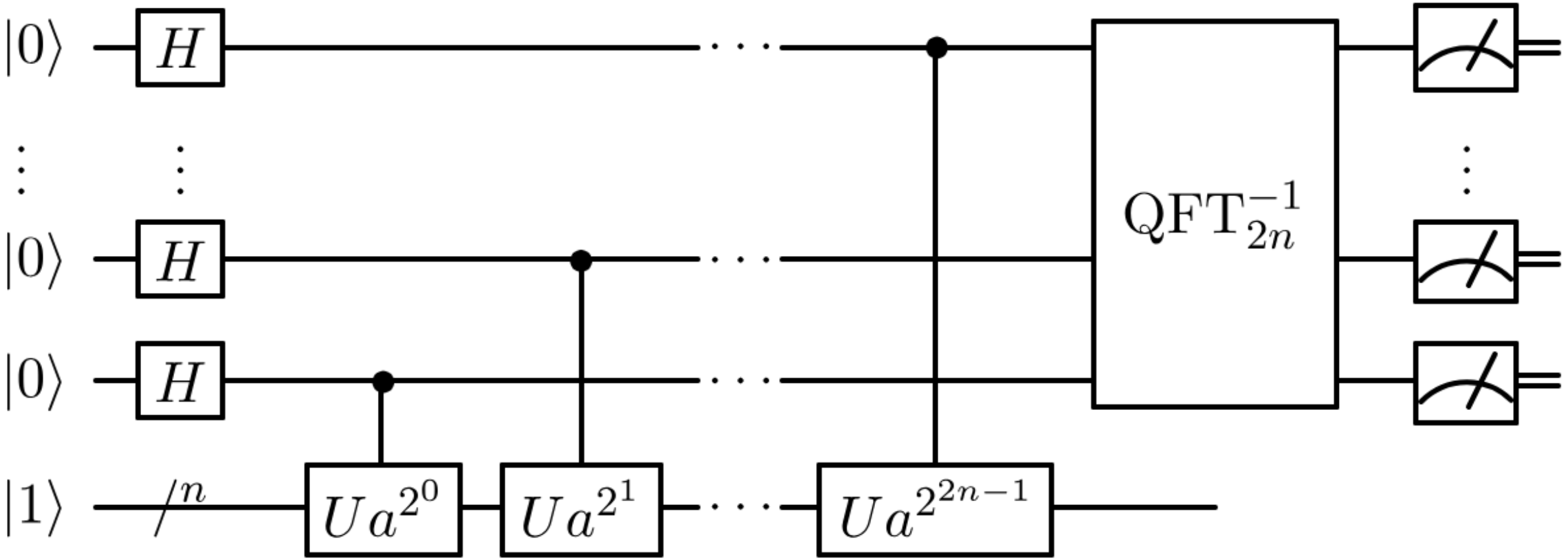






https://www.nature.com/articles/s41467-017-01904-7

Shor's algorithm: factors *N* in time that is polynomial in log *N* instead of time that is sub-exponential in log *N* , yielding a superpolynomial speedup



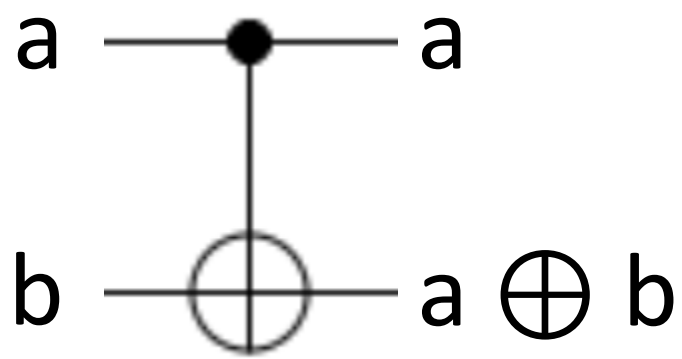
https://en.wikipedia.org/wiki/Shor%27s_algorithm

- Qubits can be implemented using photon polarization, particle spin, superconducting current loops…
- Qubits can store 0 and 1 in superposition. Measuring the state of a qubit causes it to collapse to 0 or 1. Qubits can also be entangled (i.e., there is a correlation between their measured values).
- Quantum circuits and quantum logic gates are an abstraction layer on top of physical qubits
  - We hide the details of the physical qubits
  - We typically assume no noise, no decoherence, and assume the qubits are fully interconnected, and assume all qubit logic gates are available to us. In reality, there may be a compiler or translation stage that substitutes gates, adds swaps, optimizes for noise, etc.
  - The 'wires' (horizontal lines) are metaphorical; they show the sequence of operations on each qubit, with time advancing left-to-right. They are not physical wires.
  - The circuit is typically executed many times, allowing us to measure probabilities of 0 and 1
- All quantum circuits can be simulated on a classical computer
- Simulating an N-qubit circuit with a classical computer requires exponential ($O(2^N)$) time and space, but this does not necessarily mean that a quantum algorithm gives us an exponential speedup over the best classical algorithm. The speedup from a quantum computer depends on the details of the algorithm.

NOT gate (also called X gate)

a	NOT a
0	1
1	0

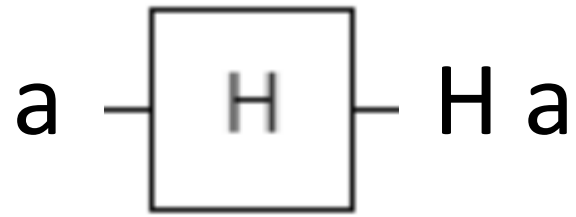
CNOT “controlled-not” gate (similar to XOR gate)



a	b	a⊕b
0	0	0
0	1	1
1	0	1
1	1	0

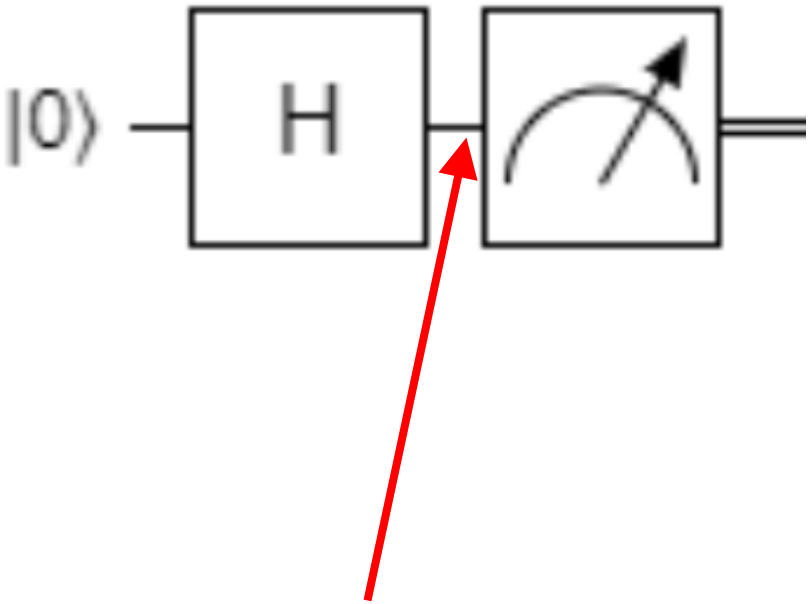
It *seems* like the second qubit has no causal effect on the first qubit, that the direction of causality is from first to second qubit. However, as we will see, with superposition and entanglement, this gate does strange things.

Hadamard gate: one of many gates that create superposition



a	**H a**
0	0 and 1
1	0 and 1

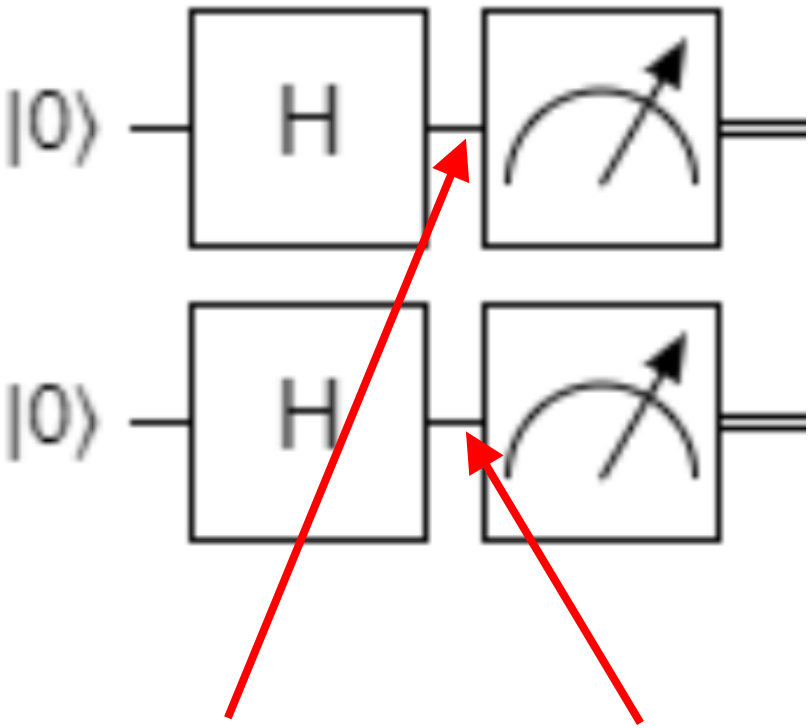
Measurement of a qubit in superposition can result in 0 or 1



Superposition of 0 and 1

Measured output is sometimes 0, sometimes 1.
If we execute the circuit and measure many times, we find the probabilities of 0 and 1 are close to 50%.

Uncorrelated qubits

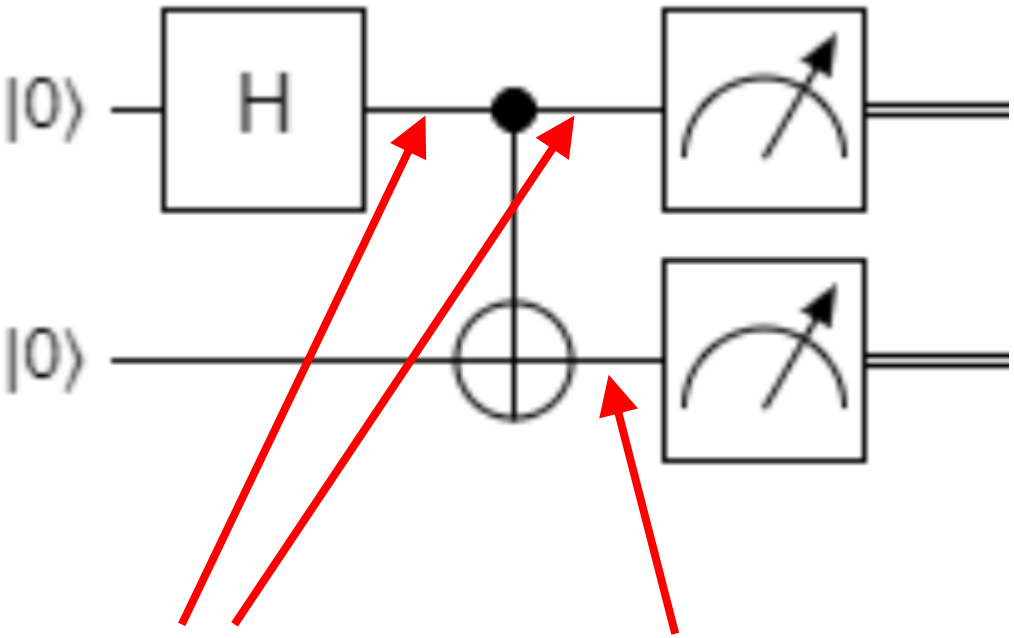


Superposition of 0 and 1

Superposition of 0 and 1

Measured output is sometimes 00, sometimes 01, sometimes 10, sometimes 11, each with 25% probability.

Circuit that entangles two qubits

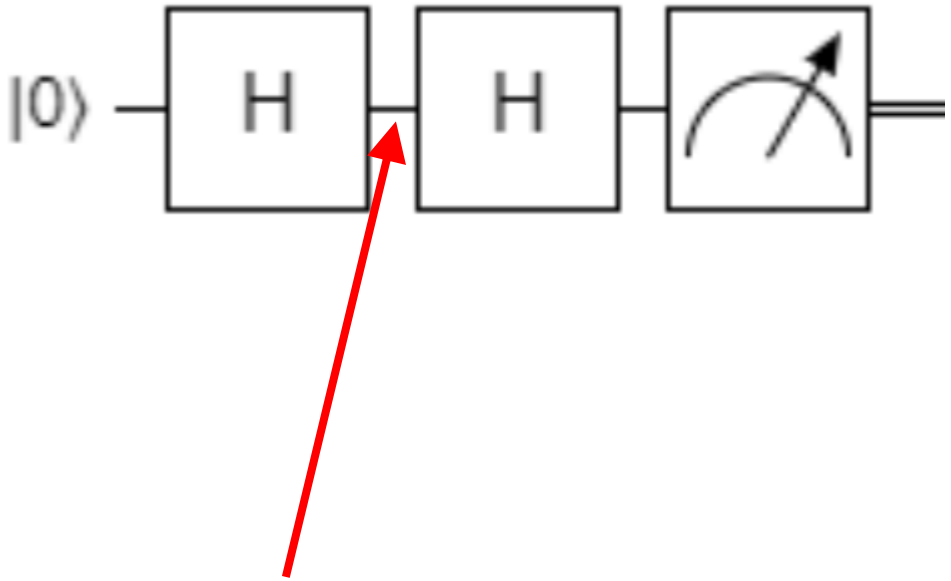


Superposition of 0 and 1

Superposition of 0 and 1

Measured output is sometimes 00, sometimes 11, never 01 nor 10. This correlation means the qubits were entangled.

Hadamard is its own inverse. $H^2 = I$

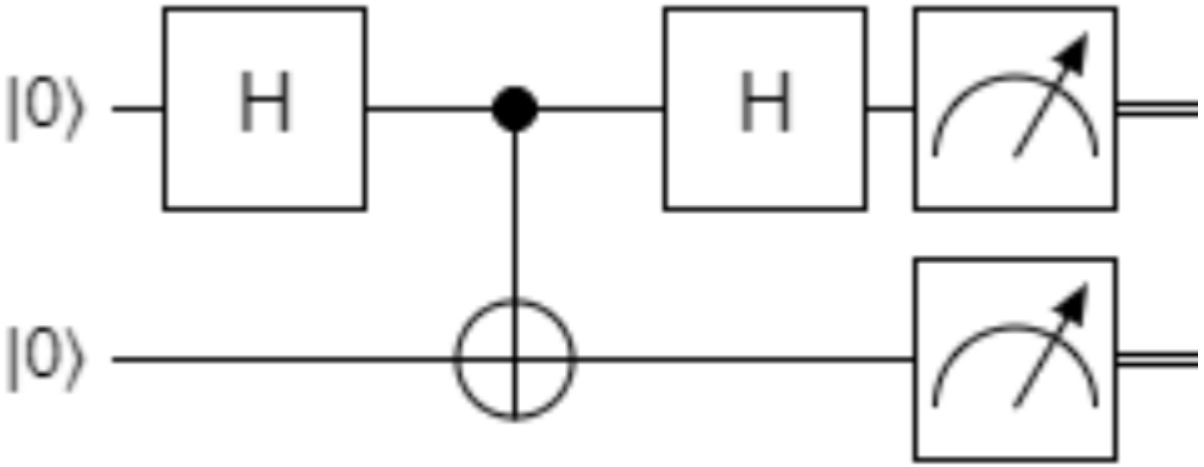


Superposition
of 0 and 1

Measured output is always 0.

We can say that the two superposed states destructively interfere in the second H gate.

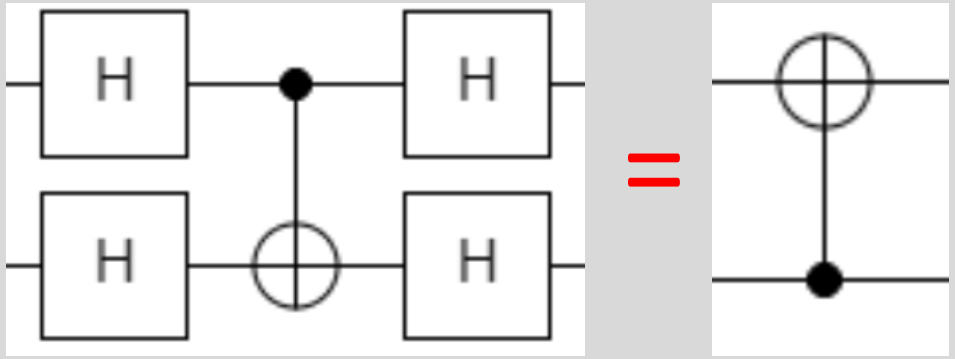
The control bit does not just “read” !!

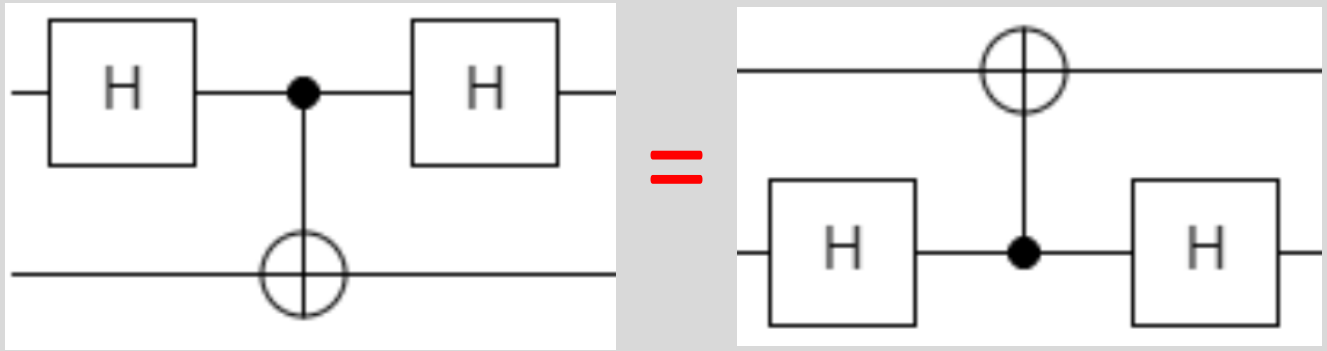


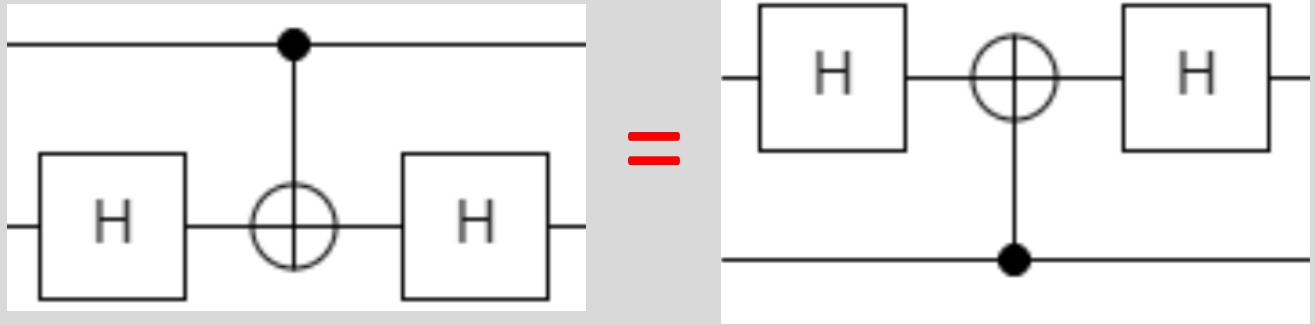
Measured output can be 00, 01, 10, or 11, each with 25% probability!
So the second H gate no longer has the same effect as previously.

Explanation by Maria Violaris:
https://www.youtube.com/watch?v=CI41yWg38Oo

There are equivalent sets of gates. The three equations below are all equivalent to each other (we can see this by adding more H gates and canceling gates when possible). These equivalences also seem to reverse the direction of causality in the CNOT gate!







Each gate can be described with a matrix

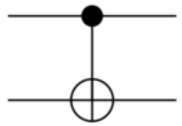
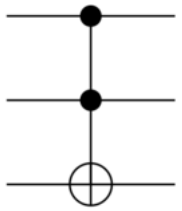
Operator	Gate(s)	Matrix
Pauli-X (X)		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)		$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
SWAP		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

Table adapted from https://en.wikipedia.org/wiki/Quantum_logic_gate

The processing of qubits can be described with matrix multiplications

amplitude for 0

amplitude for 1

amplitude for 0

amplitude for 1

$$\begin{bmatrix} -0.707i \\ 0.707i \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0.707 & 0.707 \\ 0.707 & -0.707 \end{bmatrix} \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

output state vector

input state vector

Bra and ket notation

- <ψ| is read as “bra psi”, and means the state psi as a row vector
- |ψ> is “ket psi”, the state psi as a column vector
- <a|b> means the dot product of the row vector a with the column vector b, for example:

$$\langle a|b\rangle = [a_1 \quad a_2]\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

- |b><a| means the matrix product resulting from column vector b multiplied by row vector a, for example:

$$|b\rangle\langle a| = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}[a_1 \quad a_2] = \begin{bmatrix} a_1 b_1 & a_2 b_1 \\ a_1 b_2 & a_2 b_2 \end{bmatrix}$$

- In a single qubit circuit, the state vector is a 2-element column vector containing complex numbers. Each complex number is an amplitude

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

amplitude for 0

amplitude for 1

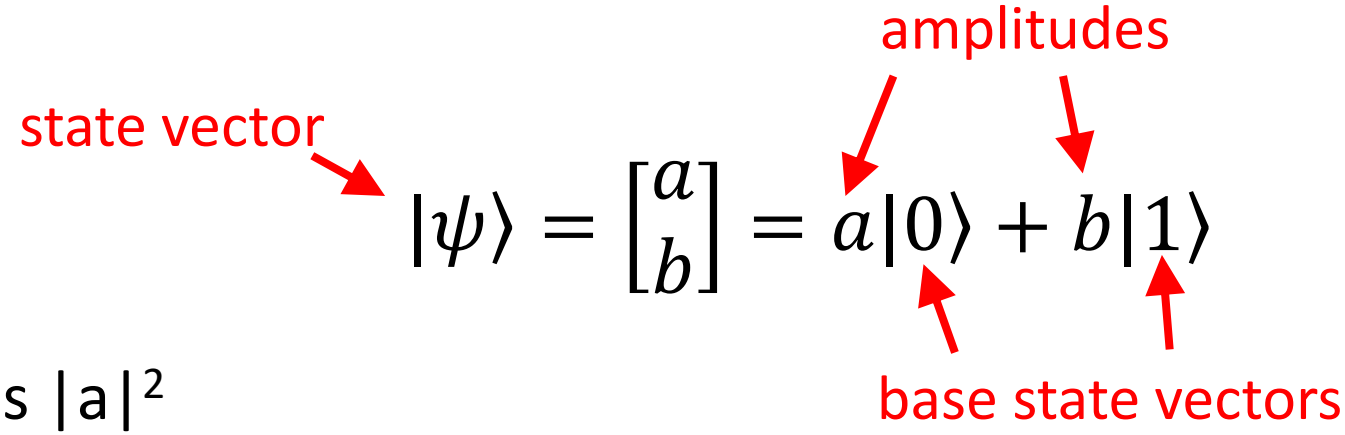
amplitudes (complex numbers)

state vector

$$|\psi\rangle = \begin{bmatrix} a \\ b \end{bmatrix} = a|0\rangle + b|1\rangle$$

base state vectors

- We might think that $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ has 4 degrees of freedom, but …
- The probability of measuring a particular base state is given by the squared magnitude of the amplitude



- Probability of measuring 0 is $|a|^2$
- Probability of measuring 1 is $|b|^2$
- Constraint: $|a|^2+|b|^2 = 1$ (this removes one degree of freedom)
- We can also multiply the amplitudes by some number e^{iG} to apply a *global phase* shift of angle G around the complex plane. This global phase shift has no physical meaning and cannot be measured. This removes another degree of freedom from the amplitudes. Sometimes we choose G to make the first amplitude a positive real number, so we can write
  $|\psi\rangle = \alpha\,|0\rangle + e^{i\varphi}(1-\alpha^2)^{1/2}\,|1\rangle$ where α is positive real. This makes the two degrees of freedom, α and φ, more apparent.

- In a single qubit circuit, the state vector is a 2-element column vector containing complex numbers. Each complex number is an amplitude

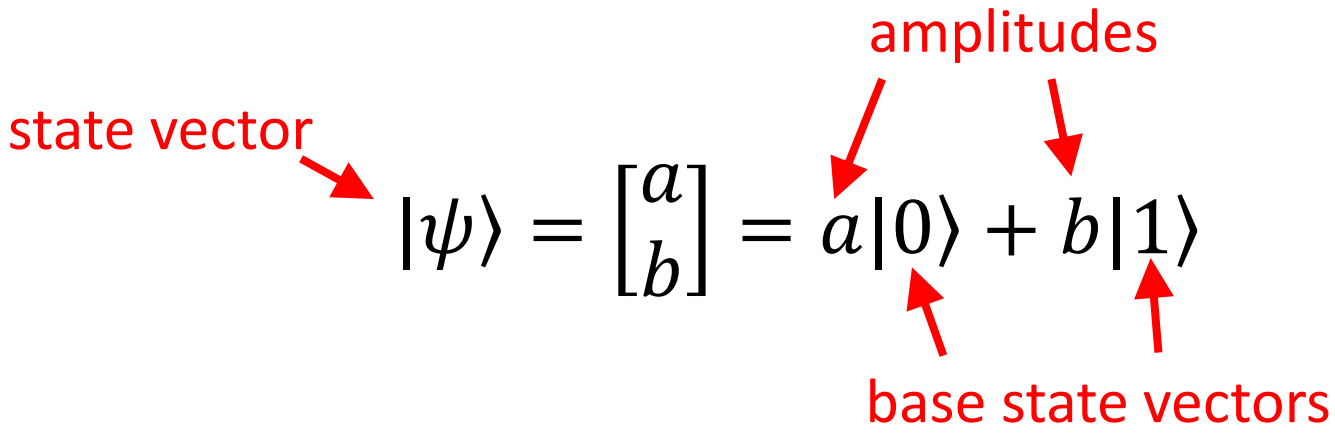
$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \qquad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} \qquad |\psi\rangle = \begin{bmatrix}a\\b\end{bmatrix} = a|0\rangle + b|1\rangle$$

- Certain superposed states have special names:

$$|+\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \qquad |-\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix} \qquad |i\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\i\end{bmatrix} \qquad |-i\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-i\end{bmatrix}$$

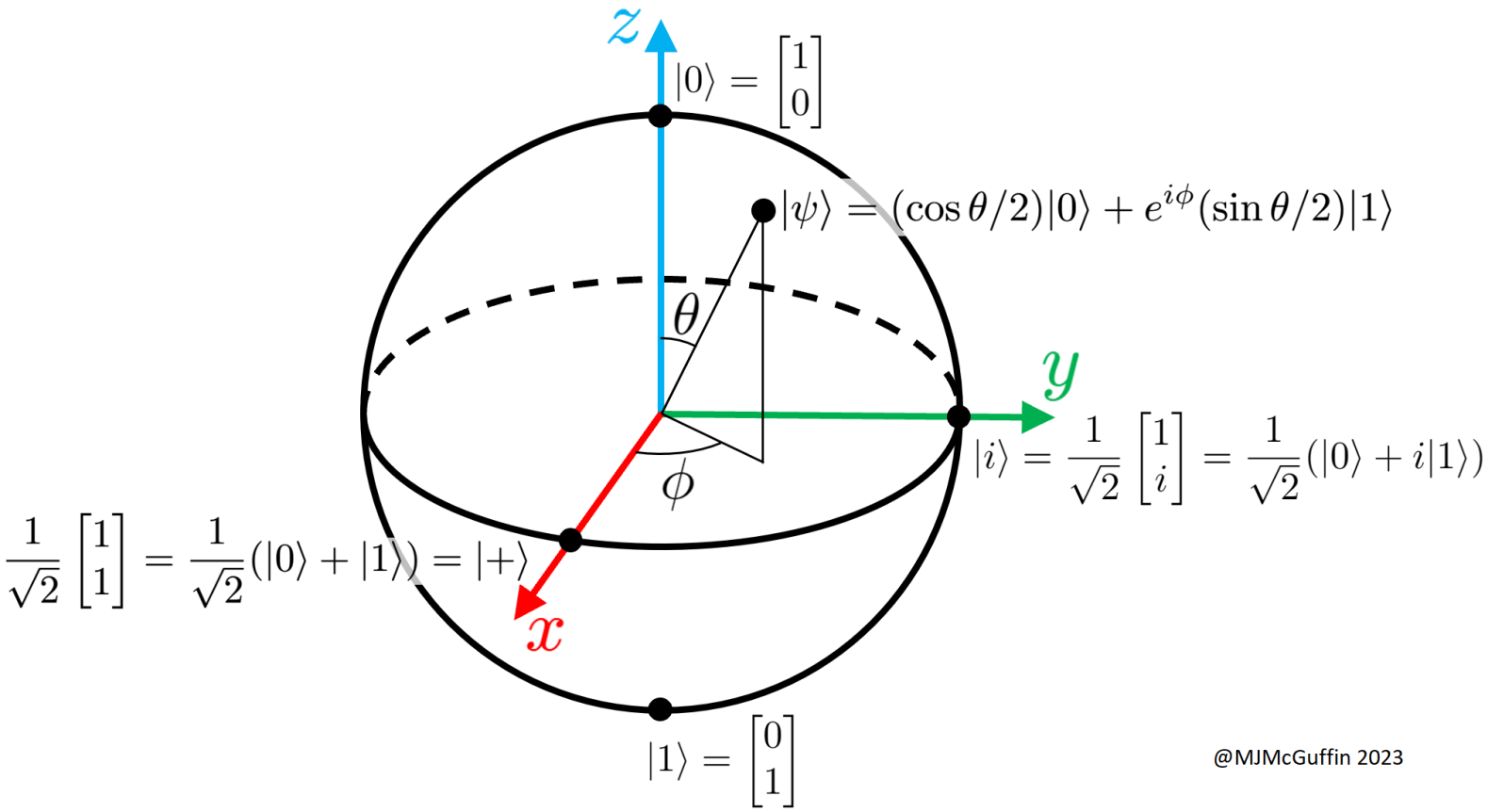
- Example:

$$H|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = |+\rangle$$



Bloch sphere: to visualize one qubit

We apply a global phase shift to obtain the form
$|\psi\rangle = \cos(\theta/2)\,|0\rangle + e^{i\varphi}\sin(\theta/2)\,|1\rangle$
which has two degrees of freedom, θ and φ, mapped to a surface.
The vertical position of the point (encoded with θ) corresponds directly to the probability of measuring 0 or 1. φ is the *relative phase*, and is only meaningful and measurable if there are other qubits.



All quantum gates acting on 1 qubit are rotations in this space!

The processing of qubits can be described with matrix multiplications

output

amplitude for 0

amplitude for 1

amplitude for 0

amplitude for 1

$$\begin{bmatrix} -0.707i \\ 0.707i \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0.707 & 0.707 \\ 0.707 & -0.707 \end{bmatrix} \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

output state vector

(this can be rewritten as |-> using a global phase shift)

input state vector

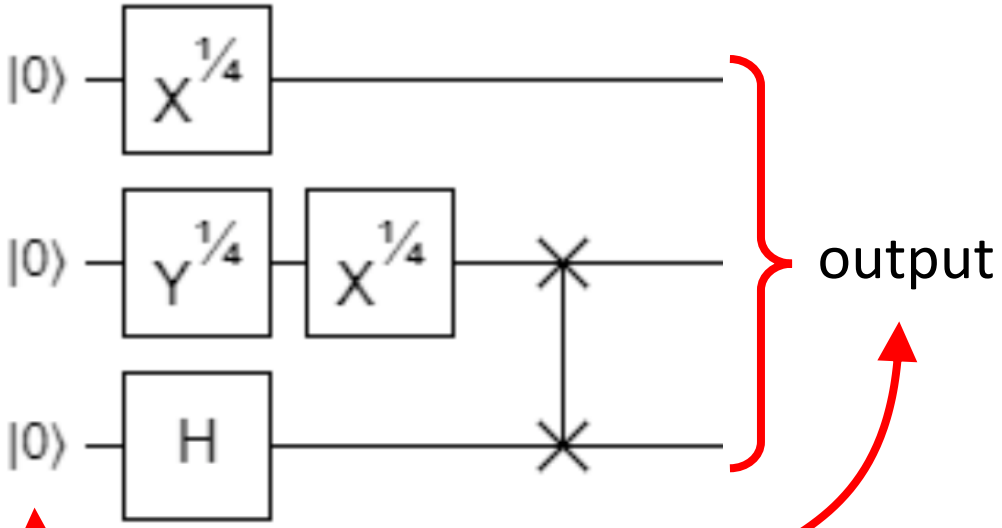
```
// qubit q0 |0>---(+)---(y)---hadamard---(z)---
//
input = CMatrix.ketZero /*q0*/;
step1 = CMatrix.gate2x2not;
step2 = CMatrix.gate2x2y;
step3 = CMatrix.gate2x2hadamard;
step4 = CMatrix.gate2x2z;
output = CMatrix.naryMult([ step4, step3, step2, step1, input ]);
console.log(StringUtil.concatenateMultilineStrings(
    step4.toString(),
    " * ", step3.toString(),
    " * ", step2.toString(),
    " * ", step1.toString(),
    " * ", input.toString(),
    " = ", output.toString({binaryPrefixes:true})
));
```

Code using classes defined at https://github.com/MJMcGuffin/muqcs.js

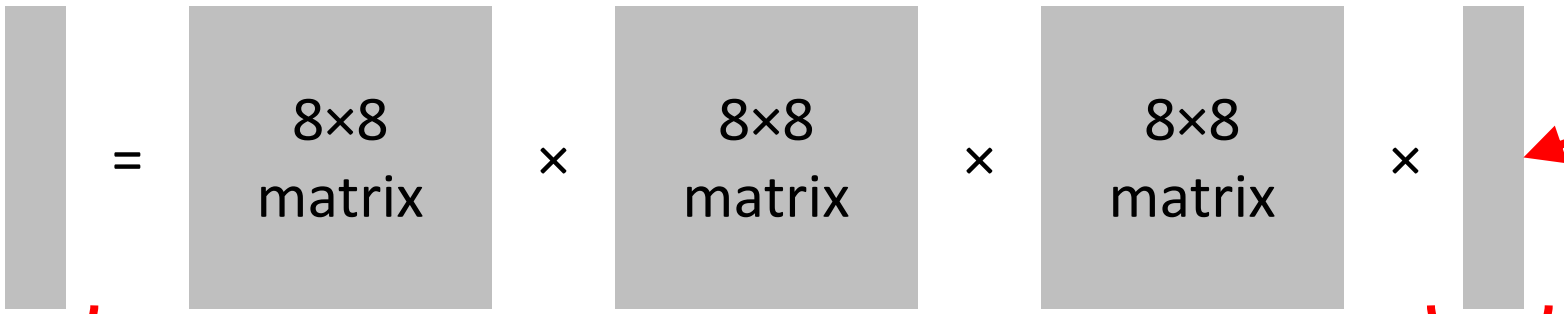
```
[1,0 ] * [0.707,0.707 ] * [0 ,-1i] * [_,1] * [1] = |0>[-0.707i]
[0,-1]   [0.707,-0.707]   [1i,0  ]   [1,_]   [_]   |1>[0.707i ]
```

My code's output

Simulation with Quirk



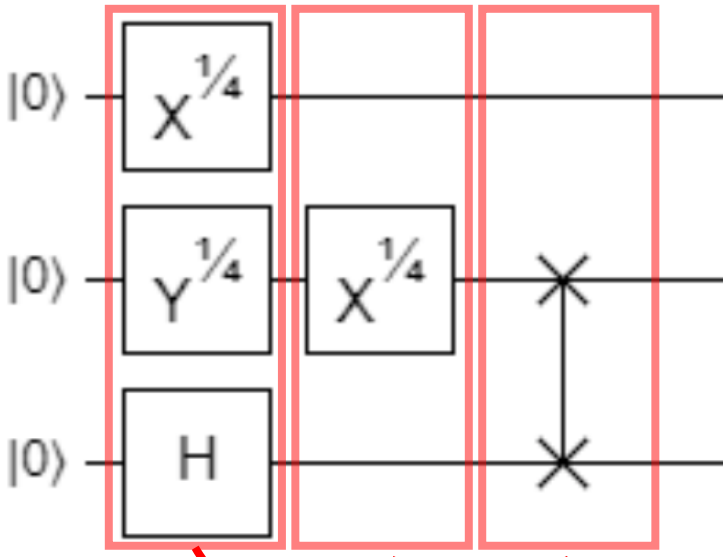
The processing of qubits can be described with matrix multiplications



8 amplitudes for states 000, 001, 010, 011, …, 111

8×1 output state vector

8×1 input state vector



Each 8×8 matrix is a tensor product (or kronecker product) of smaller matrices

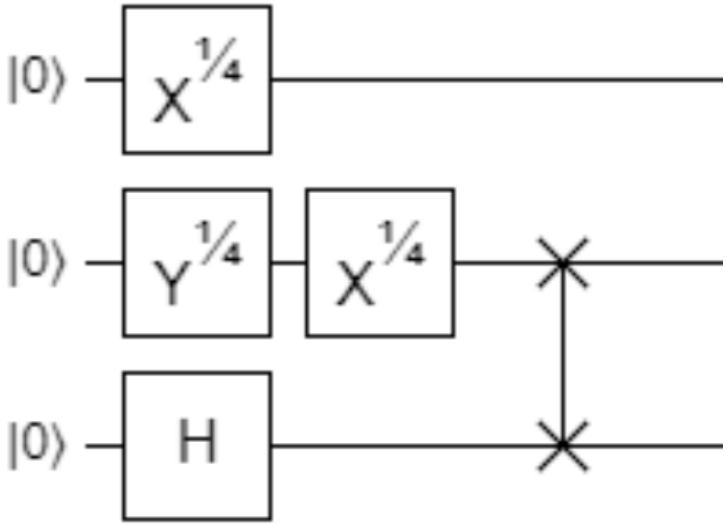
$(SWAP)\otimes(I)$

$(I)\otimes(X^{1/4})\otimes(I)$

$(H)\otimes(Y^{1/4})\otimes(X^{1/4})$

8×1 output state vector = 8×8 matrix × 8×8 matrix × 8×8 matrix × 8×1 input state vector

8 amplitudes for states 000, 001, 010, 011, …, 111



Each 8×8 matrix is a tensor product (or kronecker product) of smaller matrices

4×4 2×2

$(SWAP)\otimes(I)$

2×2 2×2 2×2

$(I)\otimes(X^{1/4})\otimes(I)$

2×2 2×2 2×2

$(H)\otimes(Y^{1/4})\otimes(X^{1/4})$

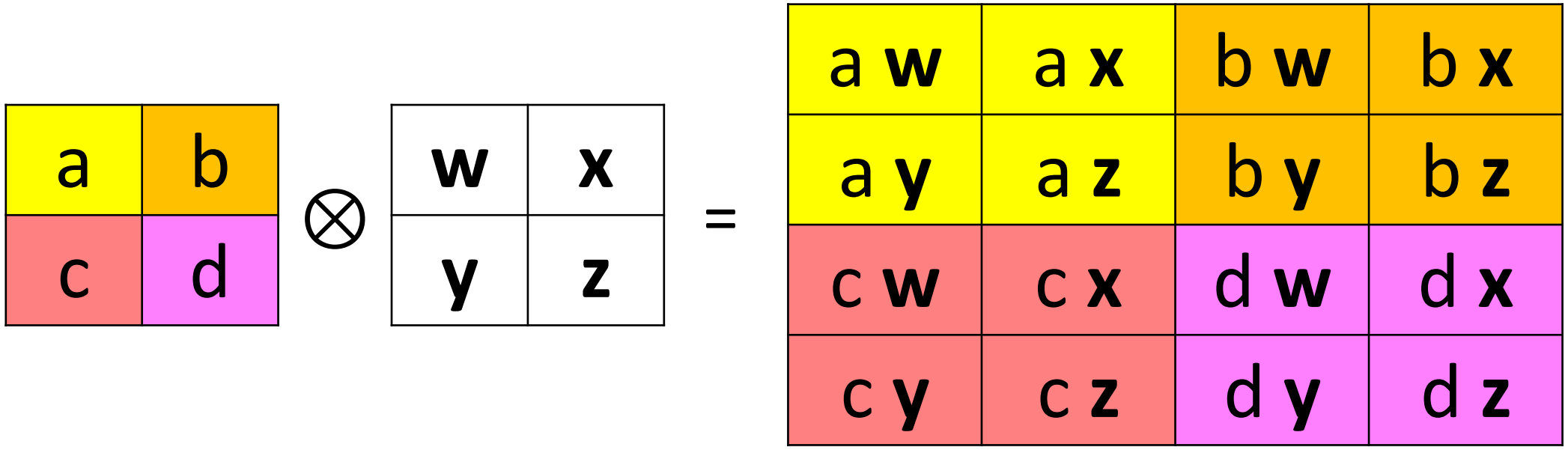
= 8×8 matrix × 8×8 matrix × 8×8 matrix ×

8 amplitudes for states 000, 001, 010, 011, …, 111

8×1 output state vector

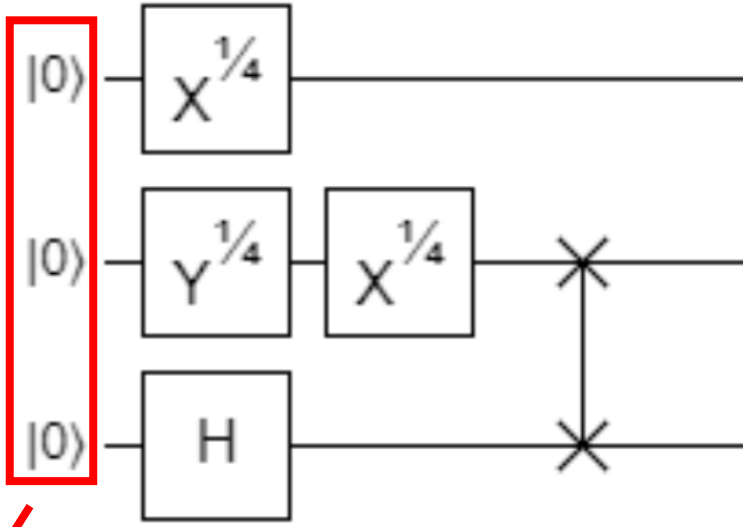
8×1 input state vector

Tensor product or kronecker product



- We stretch the first matrix and make many copies of the second matrix, and multiply pairs of elements
- The tensor product is associative but not commutative

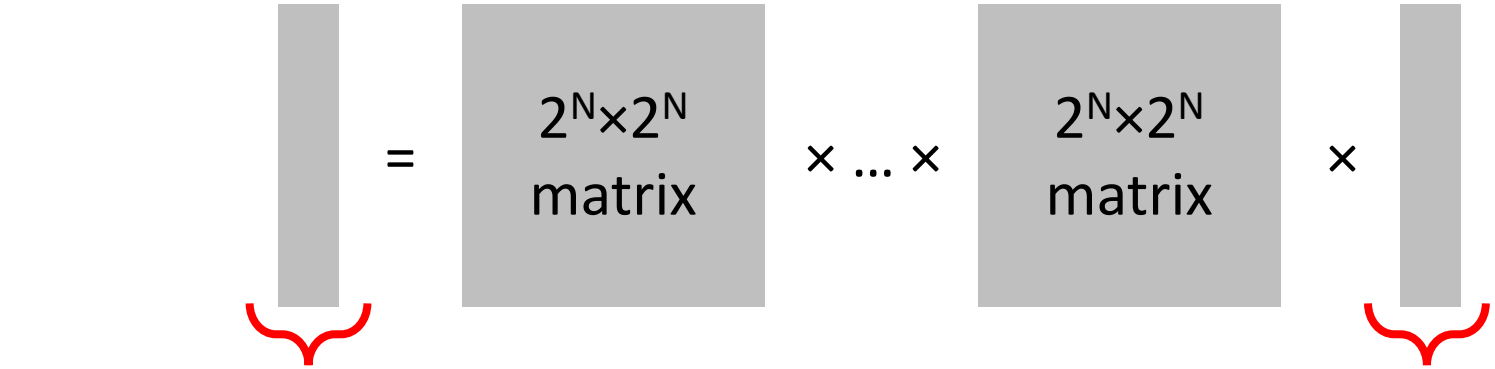
Example: the 8×1 input state vector for this circuit is a tensor product



$$|0\rangle\otimes|0\rangle\otimes|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}\otimes\begin{bmatrix}1\\0\end{bmatrix}\otimes\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}\otimes\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}$$

8 amplitudes for states 000, 001, 010, 011, …, 111

The size of the matrices grows exponentially in the number N of qubits



- Generating and storing each matrix costs $O((2^N)^2)=O(2^{2N})$ space.
- Multiplying two $2^N \times 2^N$ matrices using a naïve algorithm costs $O((2^N)^3)$ time.
- First optimization: we don't actually need to multiply these large matrices together, because we can just multiply from right to left, updating the $2^N \times 1$ state vector with each step. (Implemented in my code in CMatrix.naryMult().) So each step will only take $O(2^{2N})$ time instead of $O(2^{3N})$.

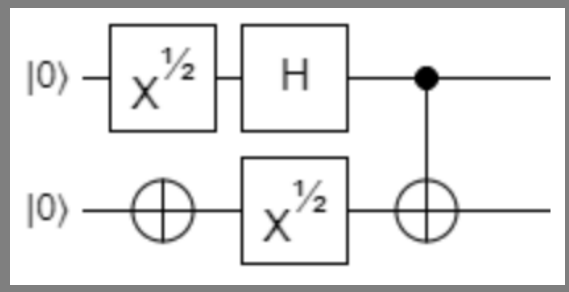
```
// qubit q0 |0>---(x^0.5)--hadamard----o-----
//                                     |
// qubit q1 |0>-----(+)-----(x^0.5)---(+)----
//
input = CMatrix.tensor( CMatrix.ketZero /*q1*/, CMatrix.ketZero /*q0*/ );
step1 = CMatrix.tensor( CMatrix.gate2x2not /*q1*/, CMatrix.gate2x2rootx /*q0*/ );
step2 = CMatrix.tensor( CMatrix.gate2x2rootx /*q1*/, CMatrix.gate2x2hadamard /*q0*/ );
step3 = CMatrix.gate4x4cnot;
output = CMatrix.naryMult([ step3, step2, step1, input ]);
console.log(StringUtil.concatenateMultilineStrings(
    step3.toString(),
    " * ", step2.toString({decimalPrecision:2}),
    " * ", step1.toString(),
    " * ", input.toString(),
    " = ", output.toString({binaryPrefixes:true})
));
```

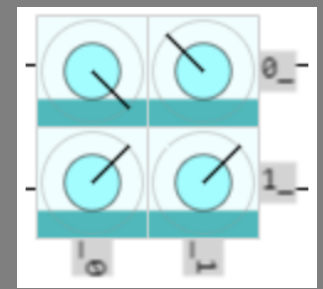
First optimization implemented in this, which multiplies smaller matrices first before bigger ones.

```
[1,_,_,_]   [0.35+0.35i,0.35+0.35i ,0.35-0.35i,0.35-0.35i ]   [0       ,0       ,0.5+0.5i,0.5-0.5i]   [1]   |00>[0.354-0.354i ]
[_,_,_,1] * [0.35+0.35i,-0.35-0.35i,0.35-0.35i,-0.35+0.35i] * [0       ,0       ,0.5-0.5i,0.5+0.5i] * [_] = |01>[-0.354+0.354i]
[_,_,1,_]   [0.35-0.35i,0.35-0.35i ,0.35+0.35i,0.35+0.35i ]   [0.5+0.5i,0.5-0.5i,0       ,0       ]   [_]   |10>[0.354+0.354i ]
[_,1,_,_]   [0.35-0.35i,-0.35+0.35i,0.35+0.35i,-0.35-0.35i]   [0.5-0.5i,0.5+0.5i,0       ,0       ]   [_]   |11>[0.354+0.354i ]
```

My code's output

Simulation with Quirk



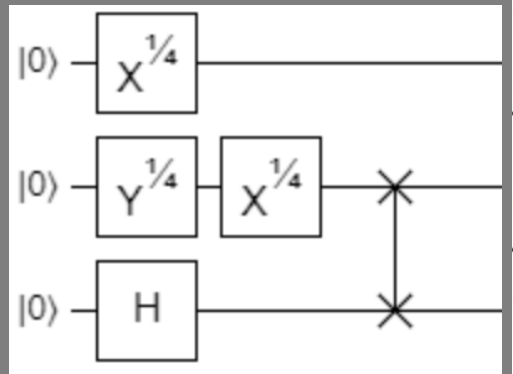


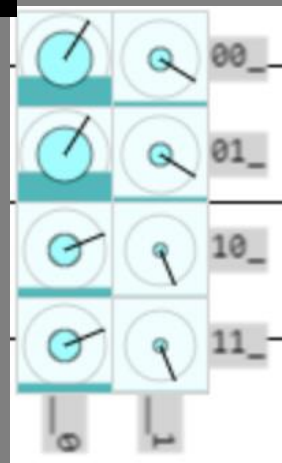
```
// qubit q0 |0>----(x^0.25)----------------------
//
// qubit q1 |0>----(y^0.25)-----(x^0.25)----X-----
//                                         |
// qubit q2 |0>-------H---------------------X-----
//
input = CMatrix.naryTensor( [ CMatrix.ketZero /*q2*/, CMatrix.ketZero /*q1*/, CMatrix.ketZero /*q0*/ ] );
step1 = CMatrix.naryTensor( [ CMatrix.gate2x2hadamard /*q2*/, CMatrix.gate2x2fourthrooty /*q1*/, CMatrix.gate2x2fourthrootx /*q0*/ ] );
step2 = CMatrix.naryTensor( [ CMatrix.gate2x2identity /*q2*/, CMatrix.gate2x2fourthrootx /*q1*/, CMatrix.gate2x2identity /*q0*/ ] );
step3 = CMatrix.wireSwap(1,2,3);
output = CMatrix.naryMult([ step3, step2, step1, input ]);
console.log(StringUtil.concatenateMultilineStrings(
    step3.toString(),
    " * ", step2.toString({decimalPrecision:1}),
    " * ", "...", // step1.toString({decimalPrecision:1}),
    " * ", input.toString(),
    " = ", output.toString({binaryPrefixes:true})
));
```

```
[1,_,_,_,_,_,_,_]   [0.9+0.4i,0       ,0.1-0.4i,0       ,0       ,0       ,0       ,0       ]         [1]   |000>[0.302+0.479i]
[_,1,_,_,_,_,_,_]   [0       ,0.9+0.4i,0       ,0.1-0.4i,0       ,0       ,0       ,0       ]         [_]   |001>[0.198-0.125i]
[_,_,_,_,1,_,_,_]   [0.1-0.4i,0       ,0.9+0.4i,0       ,0       ,0       ,0       ,0       ]         [_]   |010>[0.302+0.479i]
[_,_,_,_,_,1,_,_] * [0       ,0.1-0.4i,0       ,0.9+0.4i,0       ,0       ,0       ,0       ] * ... * [_] = |011>[0.198-0.125i]
[_,_,1,_,_,_,_,_]   [0       ,0       ,0       ,0       ,0.9+0.4i,0       ,0.1-0.4i,0       ]         [_]   |100>[0.302+0.125i]
[_,_,_,1,_,_,_,_]   [0       ,0       ,0       ,0       ,0       ,0.9+0.4i,0       ,0.1-0.4i]         [_]   |101>[0.052-0.125i]
[_,_,_,_,_,_,1,_]   [0       ,0       ,0       ,0       ,0.1-0.4i,0       ,0.9+0.4i,0       ]         [_]   |110>[0.302+0.125i]
[_,_,_,_,_,_,_,1]   [0       ,0       ,0       ,0       ,0       ,0.1-0.4i,0       ,0.9+0.4i]         [_]   |111>[0.052-0.125i]
```

My code's output

Simulation with Quirk





The tensor products are sparse (mostly contain zeros)

```
> identity = CMatrix.gate2x2identity;
  m = CMatrix.create([[1,2],[3,4]]);
  m0 = CMatrix.naryTensor([m,identity,identity]);
  m1 = CMatrix.naryTensor([identity,m,identity]);
  m2 = CMatrix.naryTensor([identity,identity,m]);
  console.log(StringUtil.concatenateMultilineStrings(m0.toString()," , ",
  m1.toString()," , ", m2.toString()));
```

```
[1,_,_,_,2,_,_,_]   [1,_,2,_,_,_,_,_]   [1,2,_,_,_,_,_,_]
[_,1,_,_,_,2,_,_]   [_,1,_,2,_,_,_,_]   [3,4,_,_,_,_,_,_]
[_,_,1,_,_,_,2,_]   [3,_,4,_,_,_,_,_]   [_,_,1,2,_,_,_,_]
[_,_,_,1,_,_,_,2] , [_,3,_,4,_,_,_,_] , [_,_,3,4,_,_,_,_]
[3,_,_,_,4,_,_,_]   [_,_,_,_,1,_,2,_]   [_,_,_,_,1,2,_,_]
[_,3,_,_,_,4,_,_]   [_,_,_,_,_,1,_,2]   [_,_,_,_,3,4,_,_]
[_,_,3,_,_,_,4,_]   [_,_,_,_,3,_,4,_]   [_,_,_,_,_,_,1,2]
[_,_,_,3,_,_,_,4]   [_,_,_,_,_,3,_,4]   [_,_,_,_,_,_,3,4]
```

(m)⊗(I)⊗(I) (I)⊗(m)⊗(I) (I)⊗(I)⊗(m)

Second optimization: we don't even need to store the matrices! There is a simple algorithm that iterates over all the non-zero entries, without storing a large matrix explicitly. In my code, see CMatrix.transformStateVectorWith2x2(), which is based on Quirk's source code https://github.com/Strilanc/Quirk/ , src/math/Matrix.js , applyToStateVectorAtQubitWithControls()

The tensor products are sparse (mostly contain zeros)

```
identity = CMatrix.gate2x2identity;
m = CMatrix.create([[1,2],[3,4]]);
m0 = CMatrix.naryTensor([m,identity,identity,identity]);
m1 = CMatrix.naryTensor([identity,m,identity,identity]);
m2 = CMatrix.naryTensor([identity,identity,m,identity]);
m3 = CMatrix.naryTensor([identity,identity,identity,m]);
console.log(StringUtil.concatenateMultilineStrings(m0.toString()," , ", m1.toString()," , ", m2.toString()," , ", m3.toString()));
```

```
[1,_,_,_,_,_,_,_,2,_,_,_,_,_,_,_]   [1,_,_,_,2,_,_,_,_,_,_,_,_,_,_,_]   [1,_,2,_,_,_,_,_,_,_,_,_,_,_,_,_]   [1,2,_,_,_,_,_,_,_,_,_,_,_,_,_,_]
[_,1,_,_,_,_,_,_,_,2,_,_,_,_,_,_]   [_,1,_,_,_,2,_,_,_,_,_,_,_,_,_,_]   [_,1,_,2,_,_,_,_,_,_,_,_,_,_,_,_]   [3,4,_,_,_,_,_,_,_,_,_,_,_,_,_,_]
[_,_,1,_,_,_,_,_,_,_,2,_,_,_,_,_]   [_,_,1,_,_,_,2,_,_,_,_,_,_,_,_,_]   [3,_,4,_,_,_,_,_,_,_,_,_,_,_,_,_]   [_,_,1,2,_,_,_,_,_,_,_,_,_,_,_,_]
[_,_,_,1,_,_,_,_,_,_,_,2,_,_,_,_]   [_,_,_,1,_,_,_,2,_,_,_,_,_,_,_,_]   [_,3,_,4,_,_,_,_,_,_,_,_,_,_,_,_]   [_,_,3,4,_,_,_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,1,_,_,_,_,_,_,_,2,_,_,_]   [3,_,_,_,4,_,_,_,_,_,_,_,_,_,_,_]   [_,_,_,_,1,_,2,_,_,_,_,_,_,_,_,_]   [_,_,_,_,1,2,_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,1,_,_,_,_,_,_,_,2,_,_]   [_,3,_,_,_,4,_,_,_,_,_,_,_,_,_,_]   [_,_,_,_,_,1,_,2,_,_,_,_,_,_,_,_]   [_,_,_,_,3,4,_,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,1,_,_,_,_,_,_,_,2,_]   [_,_,3,_,_,_,4,_,_,_,_,_,_,_,_,_]   [_,_,_,_,3,_,4,_,_,_,_,_,_,_,_,_]   [_,_,_,_,_,1,2,_,_,_,_,_,_,_,_,_]
[_,_,_,_,_,_,_,1,_,_,_,_,_,_,_,2]   [_,_,_,3,_,_,_,4,_,_,_,_,_,_,_,_]   [_,_,_,_,_,3,_,4,_,_,_,_,_,_,_,_]   [_,_,_,_,_,_,3,4,_,_,_,_,_,_,_,_]
[3,_,_,_,_,_,_,_,4,_,_,_,_,_,_,_]   [_,_,_,_,_,_,_,_,1,_,_,_,2,_,_,_]   [_,_,_,_,_,_,_,_,1,_,2,_,_,_,_,_]   [_,_,_,_,_,_,_,_,_,1,2,_,_,_,_,_]
[_,3,_,_,_,_,_,_,_,4,_,_,_,_,_,_]   [_,_,_,_,_,_,_,_,_,1,_,_,_,2,_,_]   [_,_,_,_,_,_,_,_,_,1,_,2,_,_,_,_]   [_,_,_,_,_,_,_,_,_,3,4,_,_,_,_,_]
[_,_,3,_,_,_,_,_,_,_,4,_,_,_,_,_]   [_,_,_,_,_,_,_,_,_,_,1,_,_,_,2,_]   [_,_,_,_,_,_,_,_,_,3,_,4,_,_,_,_]   [_,_,_,_,_,_,_,_,_,_,1,2,_,_,_,_]
[_,_,_,3,_,_,_,_,_,_,_,4,_,_,_,_]   [_,_,_,_,_,_,_,_,_,_,_,1,_,_,_,2]   [_,_,_,_,_,_,_,_,_,3,_,4,_,_,_,_]   [_,_,_,_,_,_,_,_,_,_,3,4,_,_,_,_]
[_,_,_,_,3,_,_,_,_,_,_,_,4,_,_,_]   [_,_,_,_,_,_,_,_,3,_,_,_,4,_,_,_]   [_,_,_,_,_,_,_,_,_,_,_,_,1,_,2,_]   [_,_,_,_,_,_,_,_,_,_,_,_,1,2,_,_]
[_,_,_,_,_,_,_,3,_,_,_,_,_,_,_,4]   [_,_,_,_,_,_,_,_,_,3,_,_,_,4,_,_]   [_,_,_,_,_,_,_,_,_,_,_,_,3,_,4,_]   [_,_,_,_,_,_,_,_,_,_,_,_,_,3,4,_]
[_,_,_,_,_,_,_,_,3,_,_,_,_,_,_,4]   [_,_,_,_,_,_,_,_,_,_,3,_,_,_,4,_]   [_,_,_,_,_,_,_,_,_,_,_,_,_,_,3,4]   [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,4]
```

m⊗I⊗I⊗I I⊗m⊗I⊗I I⊗I⊗m⊗I I⊗I⊗I⊗m

Second optimization: we don't even need to store the matrices! There is a simple algorithm that iterates over all the non-zero entries, without storing a large matrix explicitly. In my code, see CMatrix.transformStateVectorWith2x2(), which is based on Quirk's source code https://github.com/Strilanc/Quirk/ , src/math/Matrix.js , applyToStateVectorAtQubitWithControls()

The tensor products are sparse (mostly contain zeros)

```
identity = CMatrix.gate2x2identity;
m = CMatrix.create([[1,2],[3,4]]);
m0 = CMatrix.naryTensor([m,identity,identity,identity,identity]);
m1 = CMatrix.naryTensor([identity,m,identity,identity,identity]);
m2 = CMatrix.naryTensor([identity,identity,m,identity,identity]);
m3 = CMatrix.naryTensor([identity,identity,identity,m,identity]);
m4 = CMatrix.naryTensor([identity,identity,identity,identity,m]);
console.log(StringUtil.concatenateMultilineStrings(m0.toString()," , ", m1.toString()," , ", m2.toString()," , ", m3.toString() ," , ", m4.toString()));
```

m⊗I⊗I⊗I⊗I I⊗m⊗I⊗I⊗I I⊗I⊗m⊗I⊗I I⊗I⊗I⊗m⊗I I⊗I⊗ I⊗I⊗m

Second optimization: we don't even need to store the matrices! There is a simple algorithm that iterates over all the non-zero entries, without storing a large matrix explicitly. In my code, see CMatrix.

transformStateVectorWith2x2(), which is based on Quirk's source code https://github.com/Strilanc/Quirk/ , src/math/Matrix.js , applyToStateVectorAtQubitWithControls()

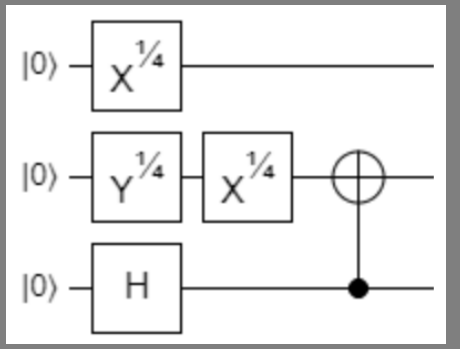
```
// qubit q0 |0>----(x^0.25)------------------------
//
// qubit q1 |0>----(y^0.25)-----(x^0.25)----(+)-----
//                                          |
// qubit q2 |0>-------H---------------------o------
//
input = CMatrix.naryTensor( [ CMatrix.ketZero /*q2*/, CMatrix.ketZero /*q1*/, CMatrix.ketZero /*q0*/ ] );
step1 = CMatrix.naryTensor( [ CMatrix.gate2x2hadamard /*q2*/, CMatrix.gate2x2fourthrooty /*q1*/, CMatrix.gate2x2fourthrootx /*q0*/ ] );
step2 = CMatrix.naryTensor( [ CMatrix.gate2x2identity /*q2*/, CMatrix.gate2x2fourthrootx /*q1*/, CMatrix.gate2x2identity /*q0*/ ] );
step3 = CMatrix.expand4x4ForNWires( CMatrix.gate4x4cnot, 2, 1, 3 );
output = CMatrix.naryMult([ step3, step2, step1, input ]);
console.log(StringUtil.concatenateMultilineStrings(
    step3.toString(),
    " * ", step2.toString({decimalPrecision:1}),
    " * ", "...", // step1.toString({decimalPrecision:1}),
    " * ", input.toString(),
    " = ", output.toString({binaryPrefixes:true})
));
```

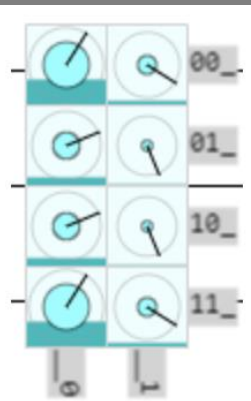
My code, without the second optimization

```
[1,_,_,_,_,_,_,_]   [0.9+0.4i,0       ,0.1-0.4i,0       ,0       ,0       ,0       ,0       ]         [1]   |000>[0.302+0.479i]
[_,1,_,_,_,_,_,_]   [0       ,0.9+0.4i,0       ,0.1-0.4i,0       ,0       ,0       ,0       ]         [_]   |001>[0.198-0.125i]
[_,_,1,_,_,_,_,_]   [0.1-0.4i,0       ,0.9+0.4i,0       ,0       ,0       ,0       ,0       ]         [_]   |010>[0.302+0.125i]
[_,_,_,1,_,_,_,_] * [0       ,0.1-0.4i,0       ,0.9+0.4i,0       ,0       ,0       ,0       ] * ... * [_] = |011>[0.052-0.125i]
[_,_,_,_,_,_,_,1,_]   [0       ,0       ,0       ,0       ,0.9+0.4i,0       ,0.1-0.4i,0       ]         [_]   |100>[0.302+0.125i]
[_,_,_,_,_,_,_,_,1]   [0       ,0       ,0       ,0       ,0       ,0.9+0.4i,0       ,0.1-0.4i]         [_]   |101>[0.052-0.125i]
[_,_,_,_,1,_,_,_]   [0       ,0       ,0       ,0       ,0.1-0.4i,0       ,0.9+0.4i,0       ]         [_]   |110>[0.302+0.479i]
[_,_,_,_,_,1,_,_]   [0       ,0       ,0       ,0       ,0       ,0.1-0.4i,0       ,0.9+0.4i]         [_]   |111>[0.198-0.125i]
```

My code's output

Simulation with Quirk



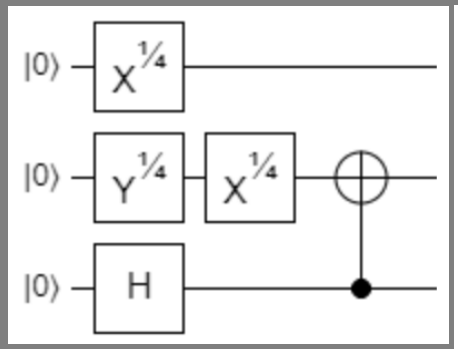


```
// qubit q0 |0>----(x^0.25)-------------------------
//
// qubit q1 |0>----(y^0.25)-----(x^0.25)----(+)-----
//                                          |
// qubit q2 |0>-------H----------------------o------
//
input = CMatrix.naryTensor( [ CMatrix.ketZero /*q2*/, CMatrix.ketZero /*q1*/, CMatrix.ketZero /*q0*/ ] );
step1 = CMatrix.transformStateVectorWith2x2(CMatrix.gate2x2hadamard,2,3,input,[]);
step1 = CMatrix.transformStateVectorWith2x2(CMatrix.gate2x2fourthrooty,1,3,step1,[]);
step1 = CMatrix.transformStateVectorWith2x2(CMatrix.gate2x2fourthrootx,0,3,step1,[]);
step2 = CMatrix.transformStateVectorWith2x2(CMatrix.gate2x2fourthrootx,1,3,step1,[]);
output = CMatrix.transformStateVectorWith2x2(CMatrix.gate2x2not,1,3,step2,[[2,true]]);
console.log(StringUtil.concatenateMultilineStrings(
    input.toString(),
    " -> ", step1.toString(),
    " -> ", step2.toString(),
    " -> ", output.toString({binaryPrefixes:true})
));
```

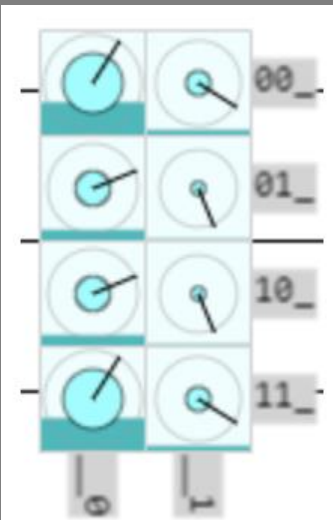
My code, with the second optimization

```
[1]     [0.427+0.427i]     [0.302+0.479i]     |000>[0.302+0.479i]
[_]     [0.177-0.177i]     [0.198-0.125i]     |001>[0.198-0.125i]
[_]     [0.177+0.177i]     [0.302+0.125i]     |010>[0.302+0.125i]
[_] ->  [0.073-0.073i] ->  [0.052-0.125i] ->  |011>[0.052-0.125i]
[_]     [0.427+0.427i]     [0.302+0.479i]     |100>[0.302+0.125i]
[_]     [0.177-0.177i]     [0.198-0.125i]     |101>[0.052-0.125i]
[_]     [0.177+0.177i]     [0.302+0.125i]     |110>[0.302+0.479i]
[_]     [0.073-0.073i]     [0.052-0.125i]     |111>[0.198-0.125i]
```

My code's output



Simulation with Quirk



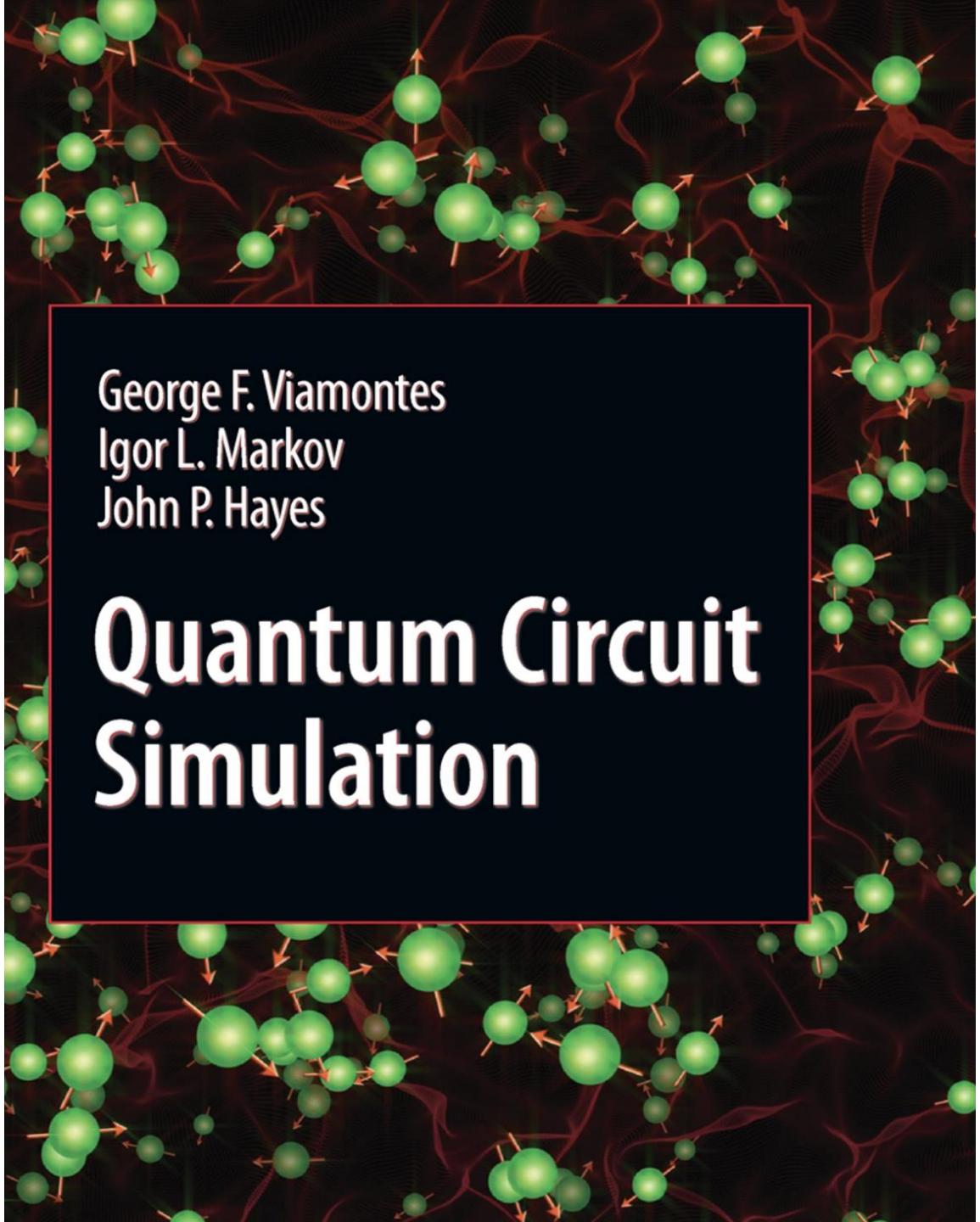
- Without the second optimization, we store explicit $2^N \times 2^N$ matrices, and at N=13 qubits, each matrix takes 0.5 gigabytes (assuming 32 bit precision)
- With the second optimization, by not storing explicit matrices, we can scale up to 20+ qubits

Runtimes on a 2018 laptop, running JavaScript code inside Chrome, without using the GPU for anything:

42 gates on 14 qubits in 249 milliseconds
43 gates on 15 qubits in 320 milliseconds
48 gates on 16 qubits in 702 milliseconds
49 gates on 17 qubits in 1296 milliseconds
54 gates on 18 qubits in 2141 milliseconds
55 gates on 19 qubits in 3974 milliseconds
60 gates on 20 qubits in 7054 milliseconds
61 gates on 21 qubits in 14207 milliseconds

For more advanced simulation algorithms, check:

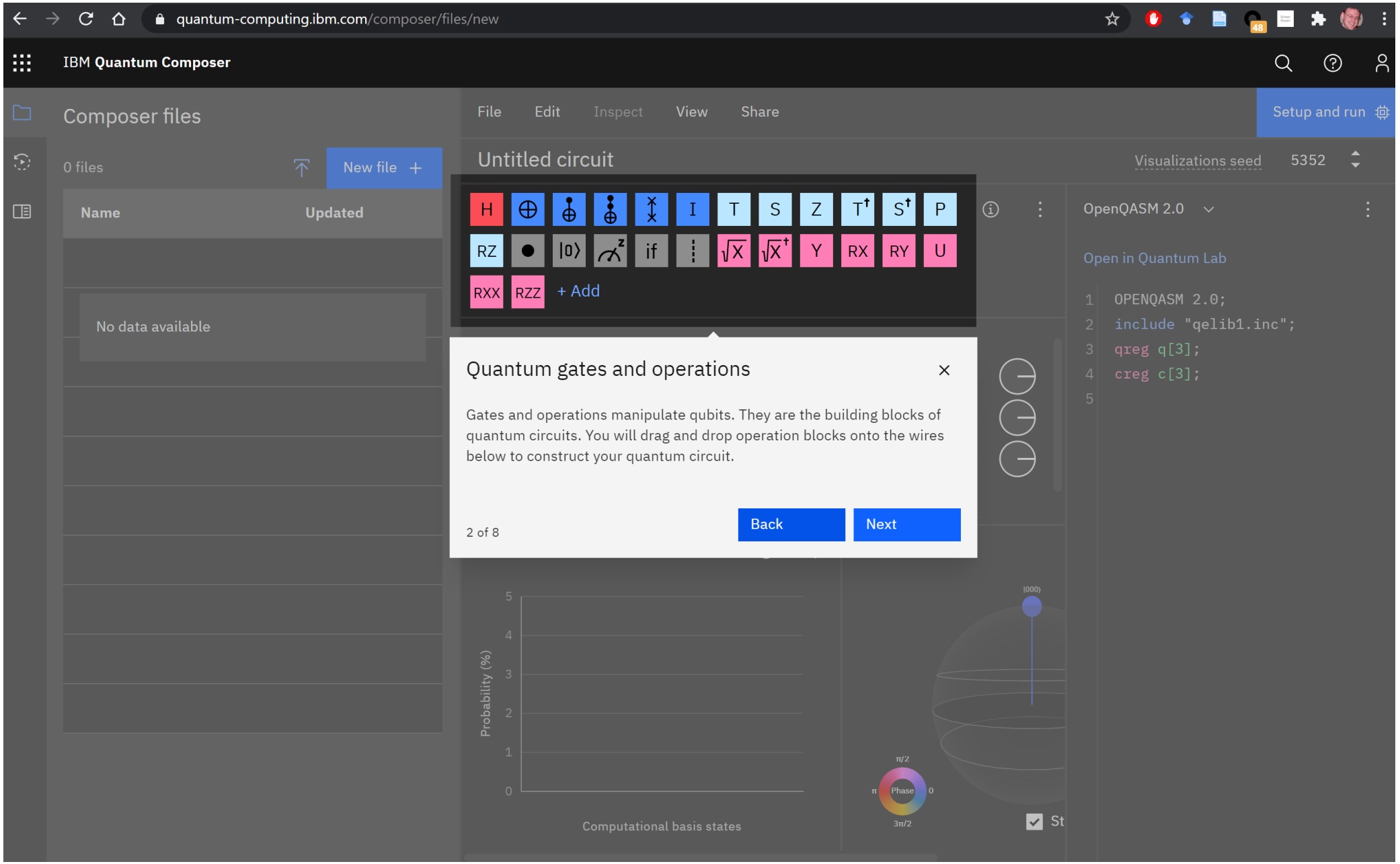
Viamontes, Markov, Hayes (2009) "Quantum Circuit Simulation"



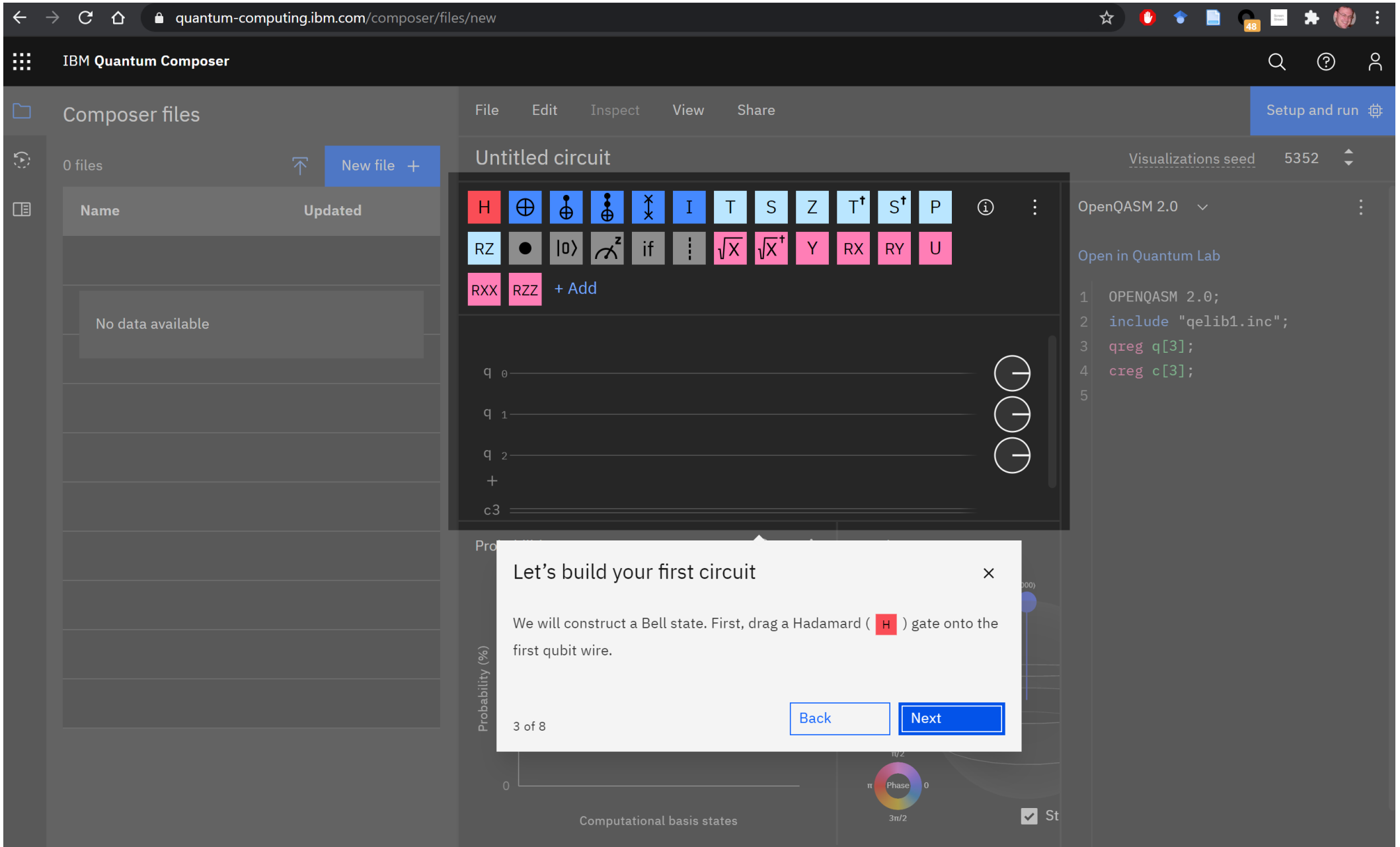
Example software

- IBM Quantum Composer
  - https://quantum-computing.ibm.com/composer

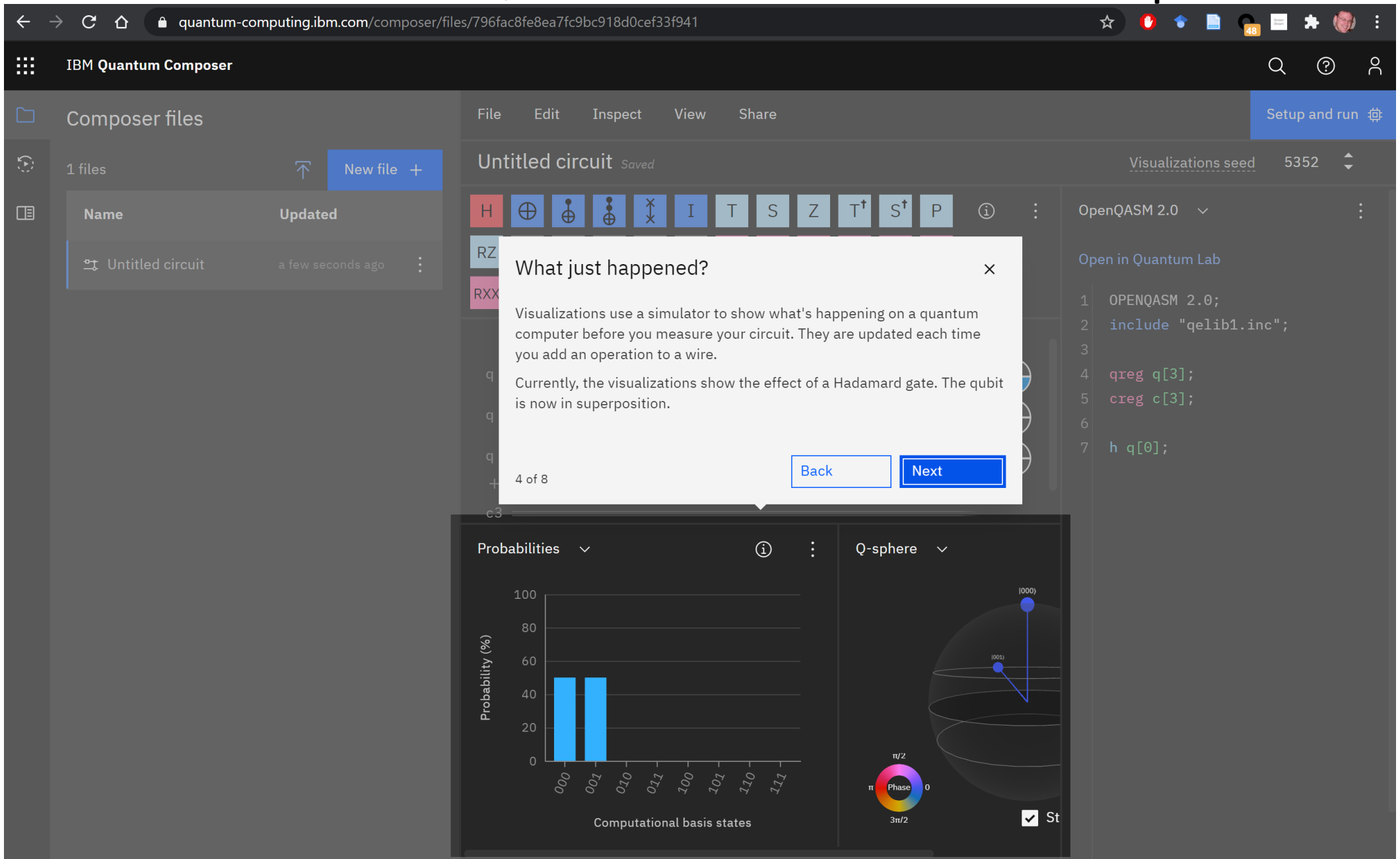
IBM Quantum Lab / IBM Quantum Composer



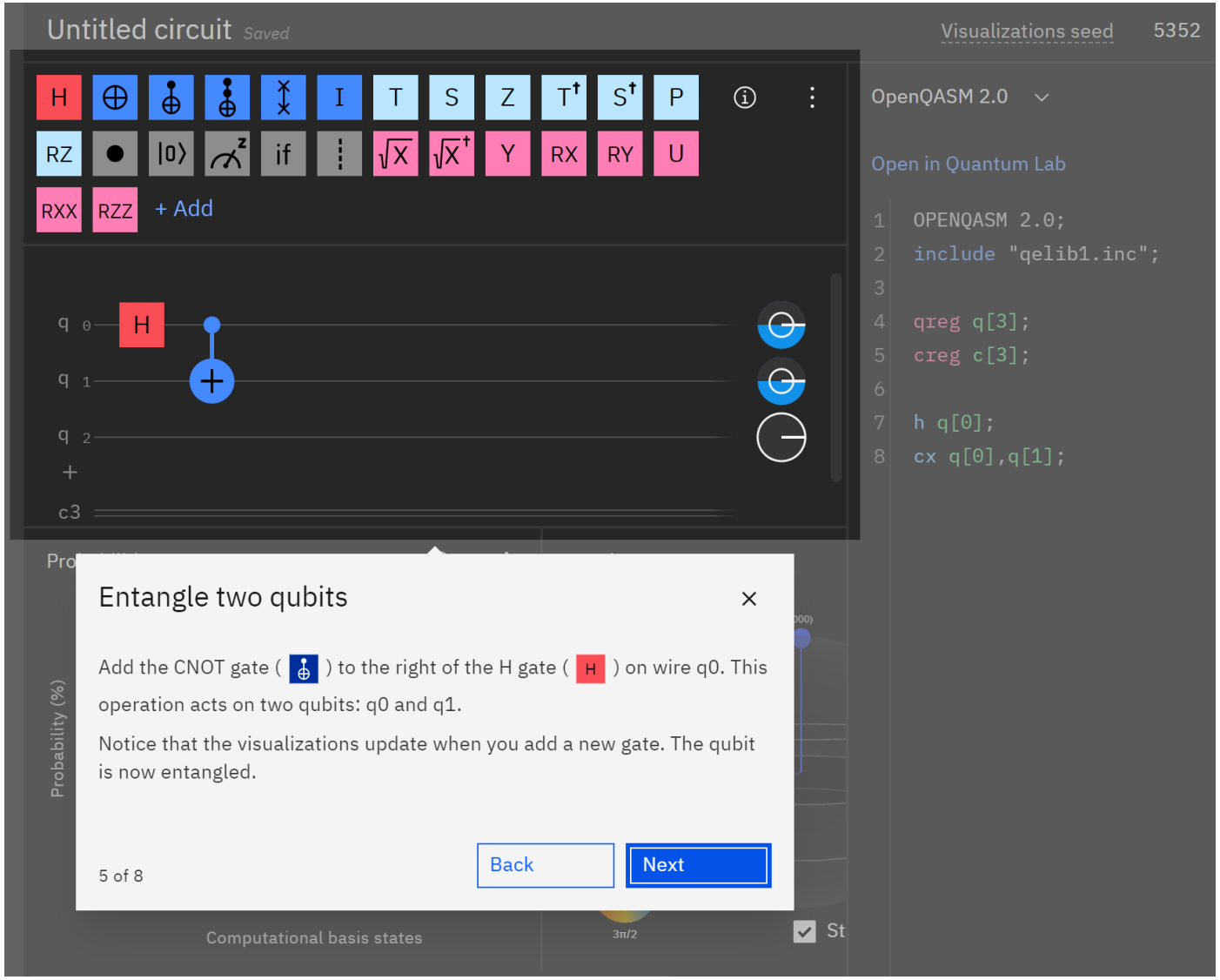
IBM Quantum Lab / IBM Quantum Composer

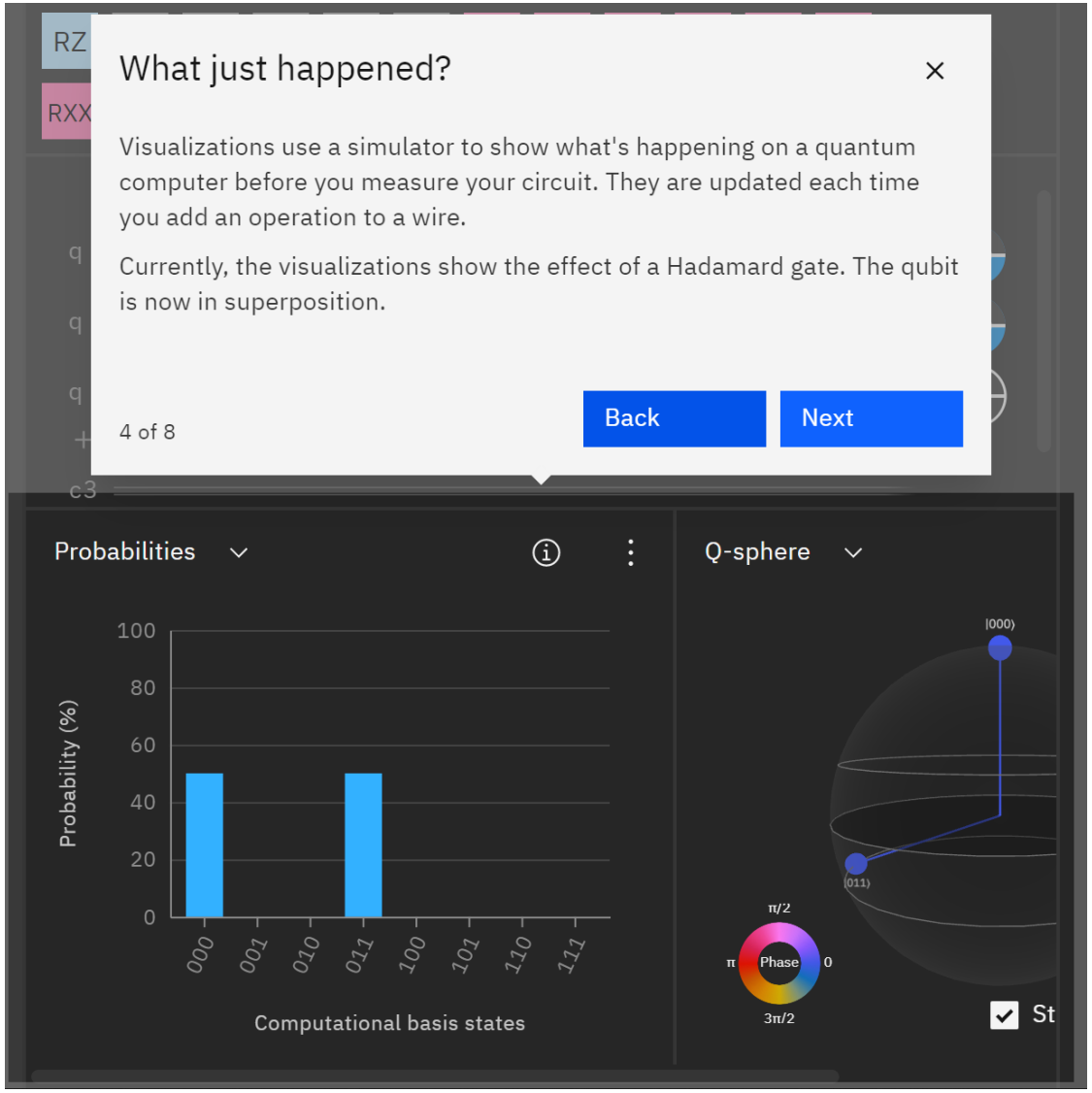


IBM Quantum Lab / IBM Quantum Composer

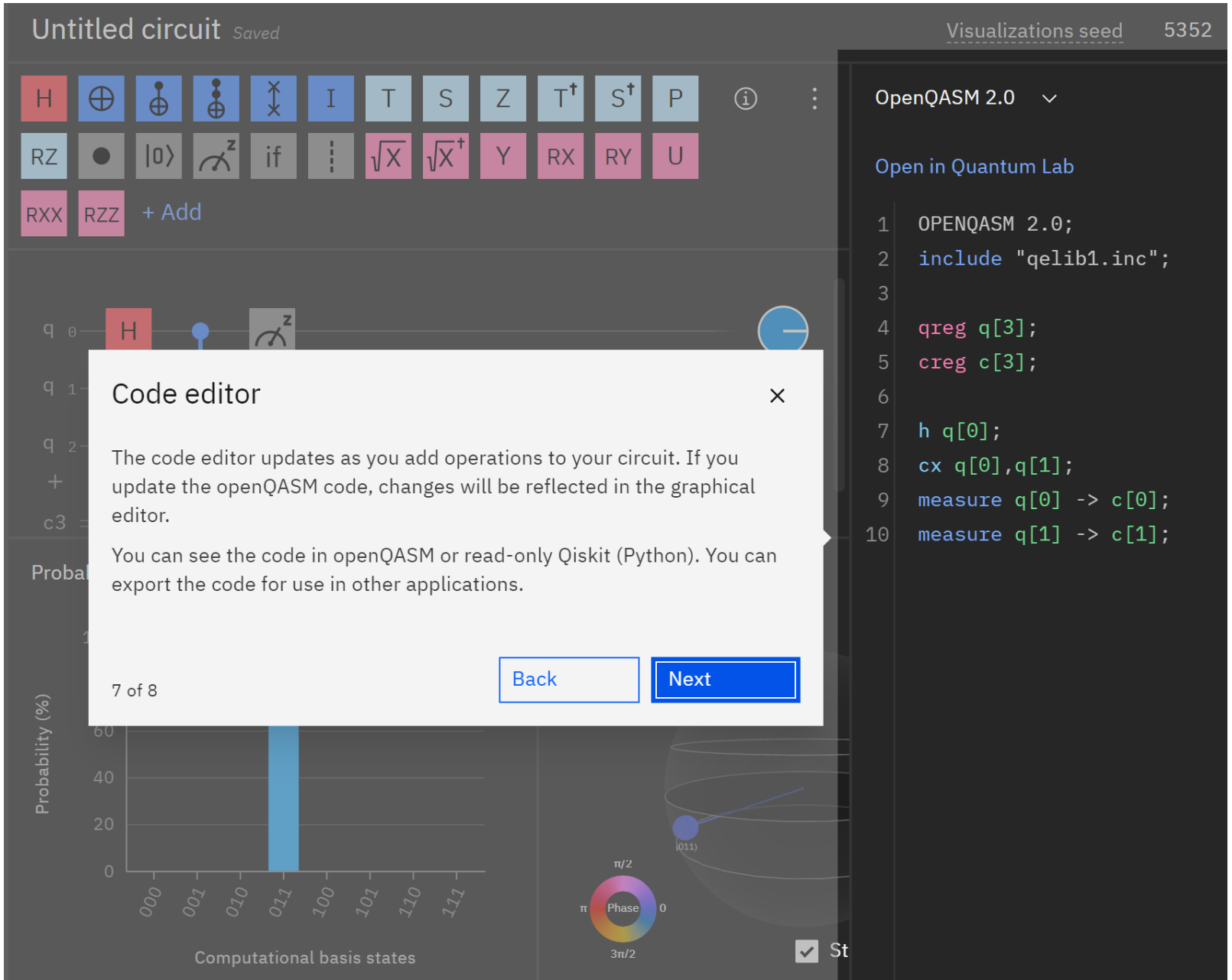


IBM Quantum Lab / IBM Quantum Composer

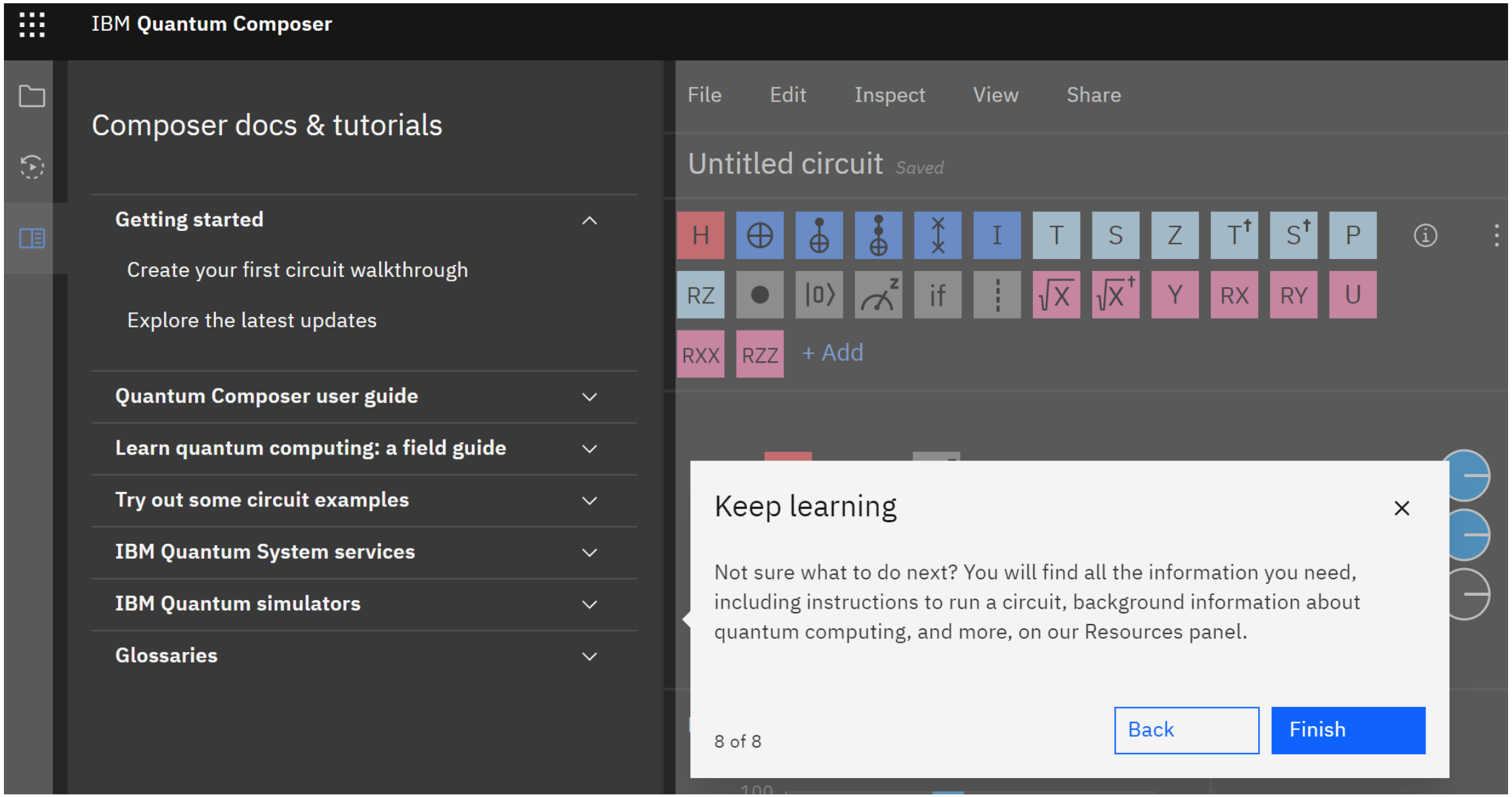




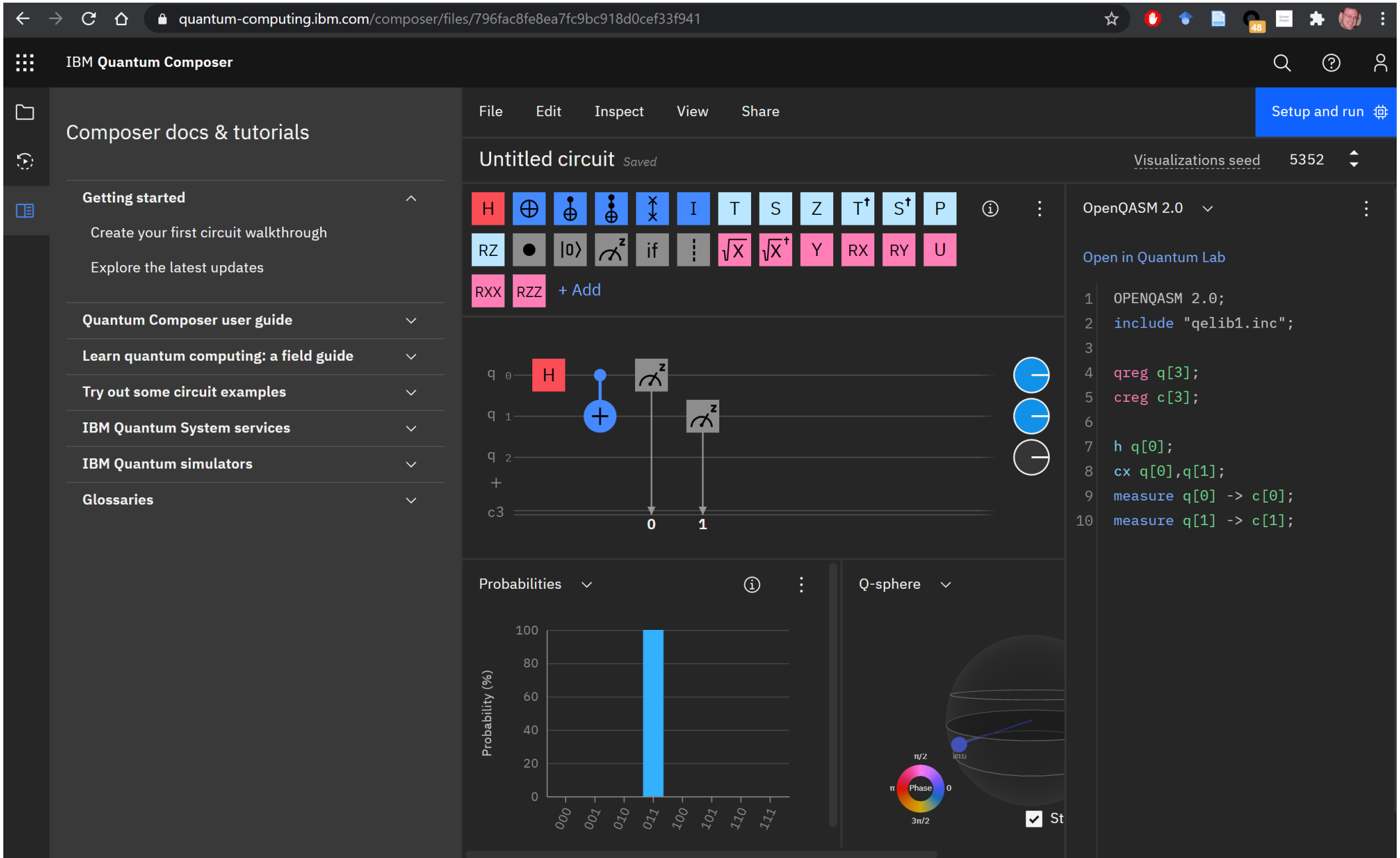
IBM Quantum Lab / IBM Quantum Composer



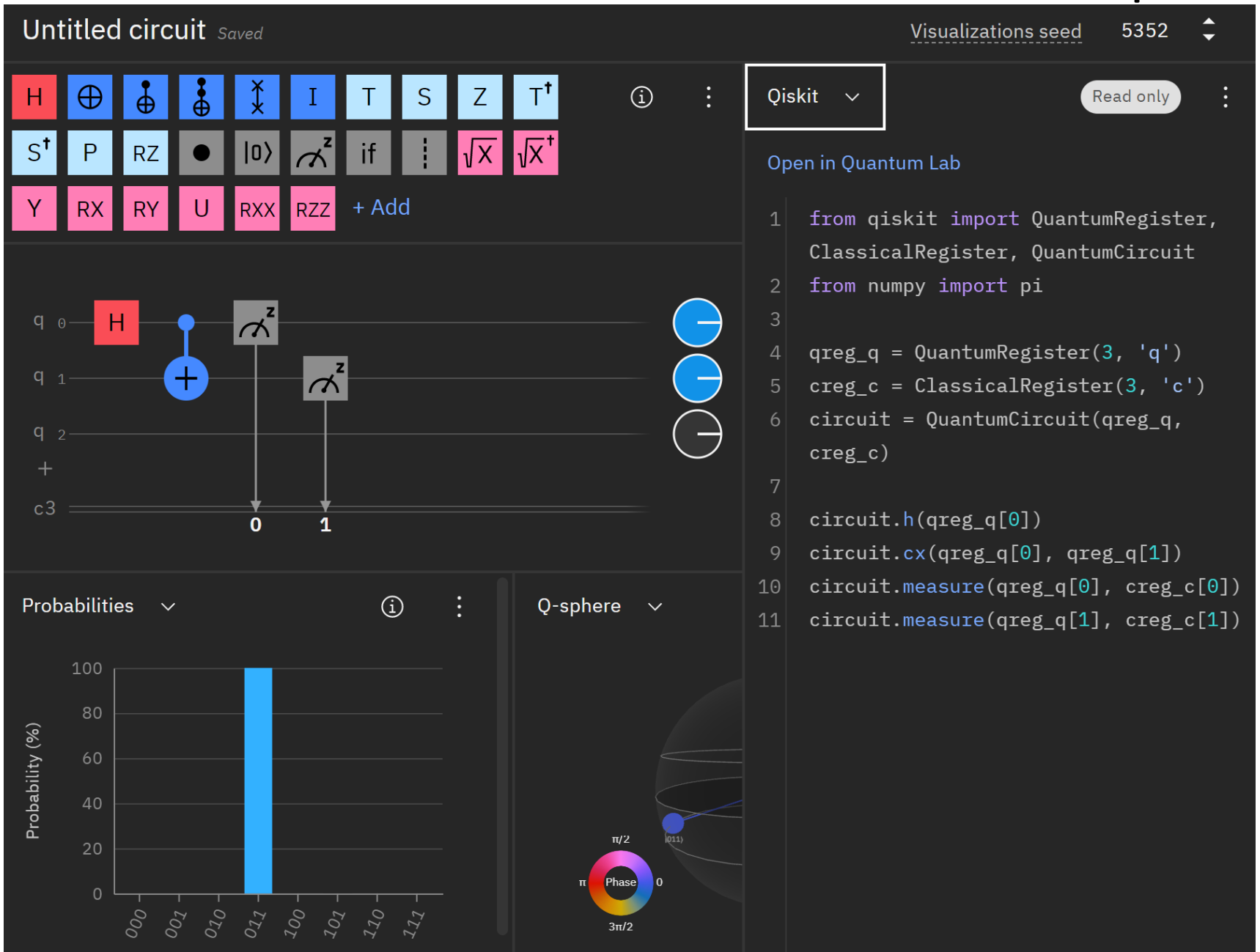
IBM Quantum Lab / IBM Quantum Composer



IBM Quantum Lab / IBM Quantum Composer



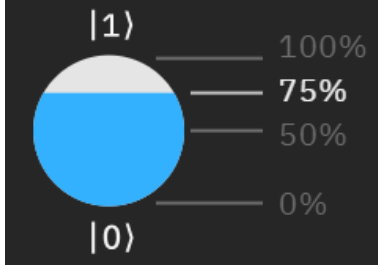
IBM Quantum Lab / IBM Quantum Composer



Phase disk

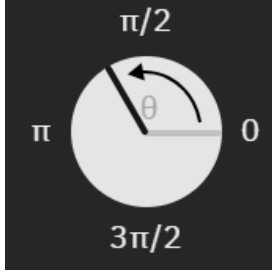
Probability the qubit is in the $|1\rangle$ state

The probability that the qubit is in the state is represented by the blue disk fill.



Quantum Phase

The quantum phase of the qubit state is given by the line that extends from the center of the diagram to the edge of the gray disk (which rotates counterclockwise around the center point).



Reduced purity of the qubit state

The radius of the black ring represents the reduced purity of the qubit state, which for qubit j in an N -qubit state $|\psi\rangle$ is given by $\mathrm{Tr}[\mathrm{Tr}_{i\neq j}[|\psi\rangle\langle\psi|]^2]$

The reduced purity for a single qubit is in the range $[0.5, 1]$; a value of one indicates that the qubit is not entangled with any other party. In contrast, a reduced purity of 0.5 shows that the qubit is left in the completely mixed state, and has some level of entanglement over the remaining $N-1$ qubits, and possibly even the environment.

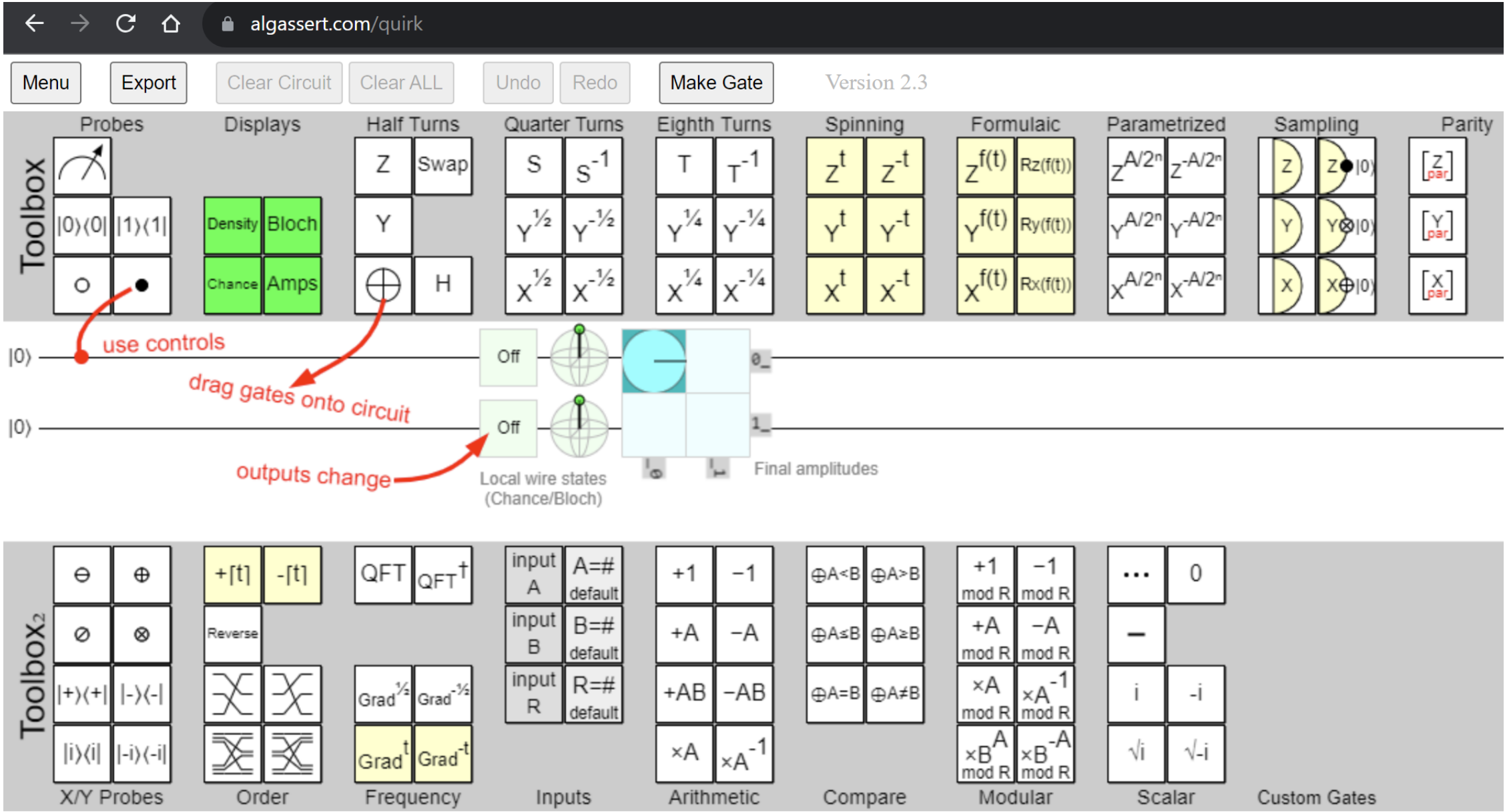
No entanglement Partially mixed

Maximally entangled

Example software

- IBM Quantum Composer
  - https://quantum-computing.ibm.com/composer
- Quirk
  - https://algassert.com/quirk

Quirk


algassert.com/quirk
Menu
Export
Clear Circuit
Clear ALL
Undo
Redo
Make Gate
Version 2.3
Toolbox
Probes
Displays
Half Turns
Quarter Turns
Eighth Turns
Spinning
Formulaic
Parametrized
Sampling
Parity
use controls
drag gates onto circuit
outputs change
Off
Local wire states
(Chance/Bloch)
Final amplitudes
Toolbox₂
X/Y Probes
Order
Frequency
Inputs
Arithmetic
Compare
Modular
Scalar
Custom Gates

Examples of proposed visualizations

Williams, Milan Marie
2021
QCVis: A Quantum Circuit Visualization and Education Platform for Novices
Bachelor's thesis, Harvard College
https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37368560
https://dash.harvard.edu/handle/1/37368560

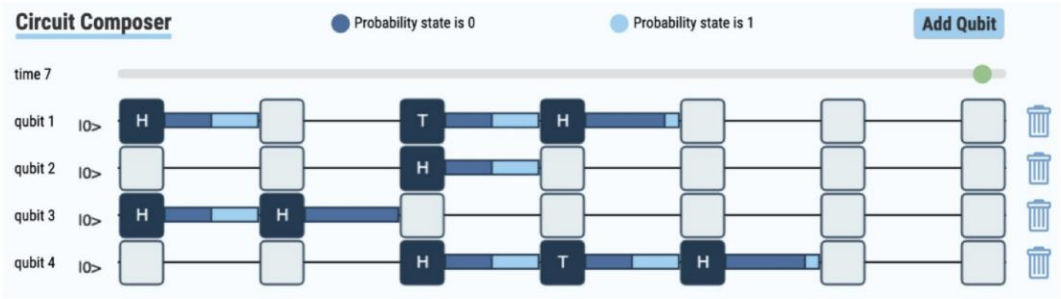


Figure 3.3: The Circuit Composer center with a four qubit circuit, showing how the stacked bar chart changes over time.

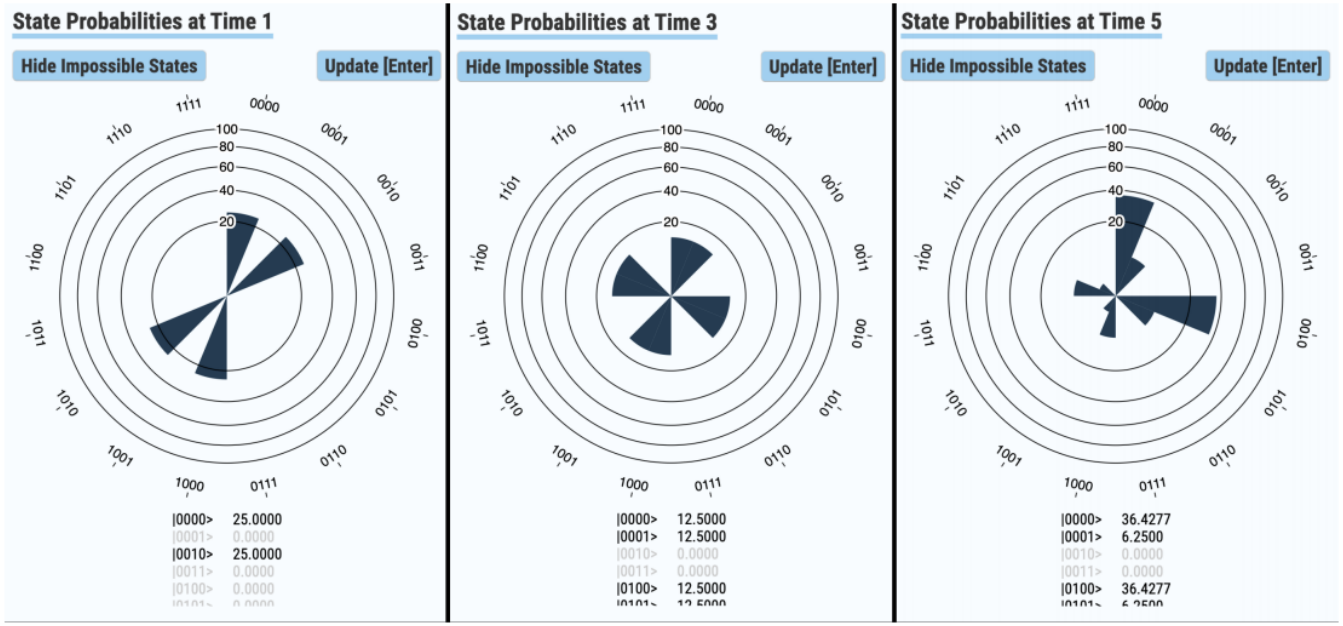
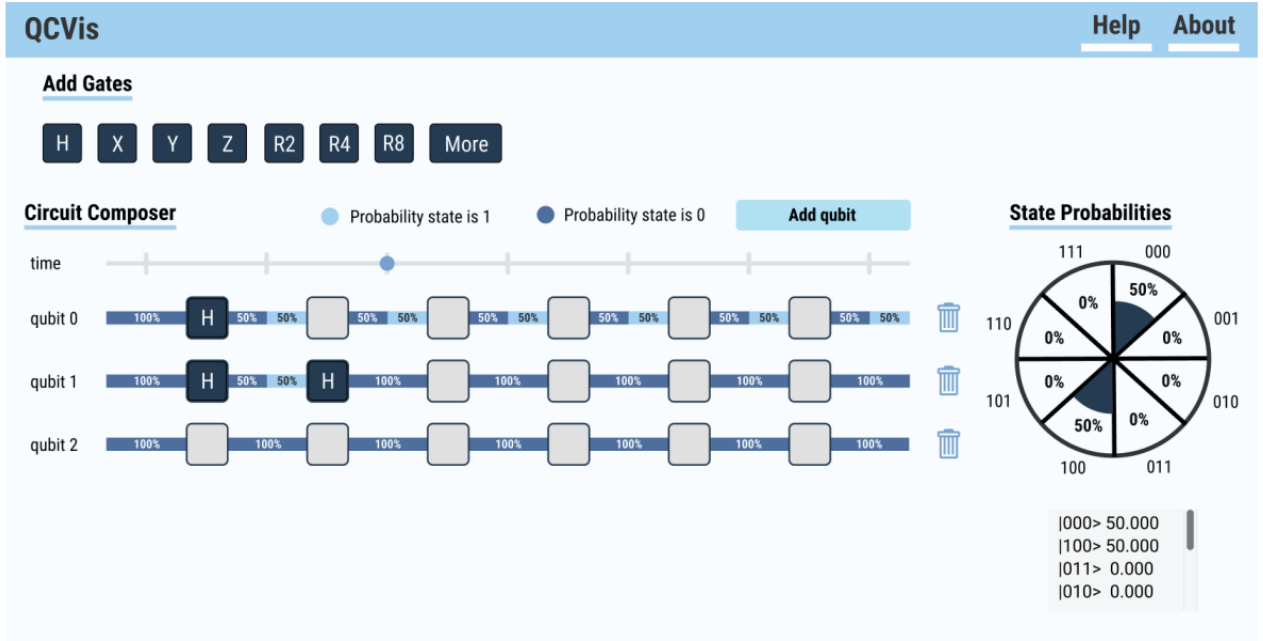


Figure 3.6: State Probability panel changes over time depending on the time slider.



% height is probability, color shows phase; it also makes it clearer when things are entangled (although this only works if the entanglement is limited to adjacent qubits)
% "entangled states are represented by boxes that span across several columns, indicating which qubits are entangled. On the contrary, non-entangled (i.e. separable) states are represented by 1-column boxes."
Jean-Baptiste Lamy
Dynamic software visualization of quantum algorithms with rainbow boxes
IVAPP 2019
2019
https://hal.archives-ouvertes.fr/hal-02264243/document

This representation requires to distinguish separable states from entangled ones, in order to determine the reference phases appropriately (*i.e.* one reference per separable factor). We achieved this by tracing entanglement through the program: at the beginning, all qubits are separated. Single-qubit gates do not create entanglement. On the contrary, multiple-qubit gates, like CNOT or CZ, entangle the two qubits they are applied on. Finally, measurements destroy the superposition with regards to the measured qubit, and thus unentangle it.

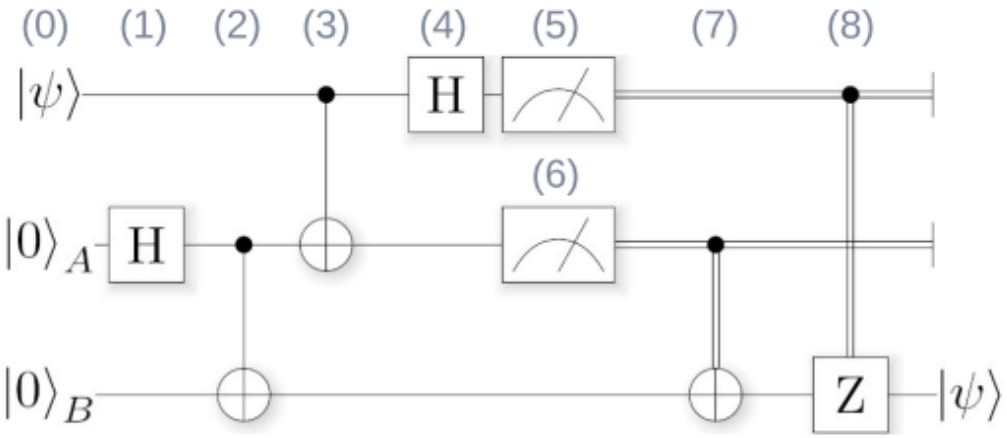


Figure 1: Block diagram of a quantum circuit performing quantum teleportation. Each horizontal line represent a qbit.

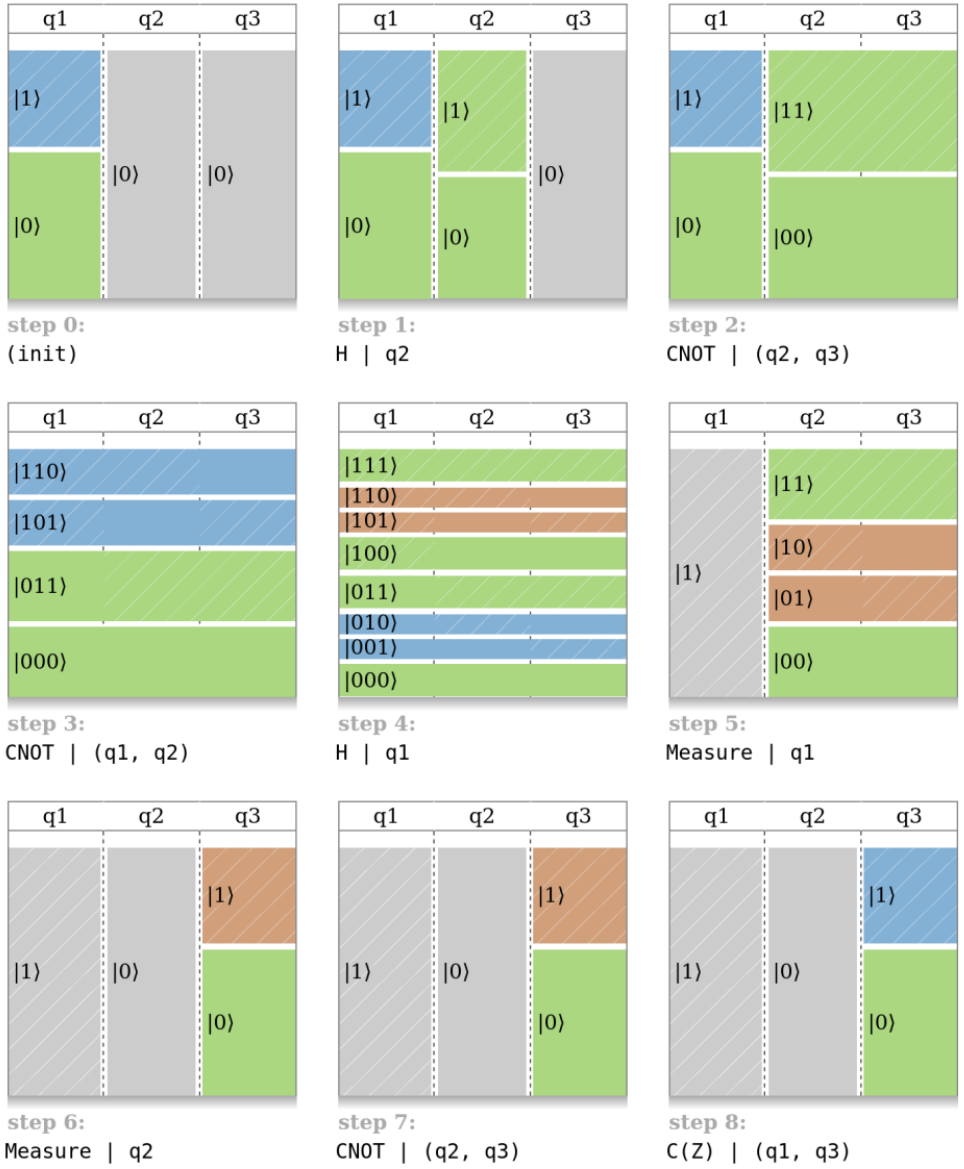


Figure 6: Visualization of the state of the system during the execution of quantum teleportation.

Lin, Siyuan, Jiang Hao, and Lingyun Sun. "Quflow: Visualizing parameter flow in quantum circuits for understanding quantum computation." In 2018 IEEE Scientific Visualization Conference (SciVis), pp. 37-41. IEEE, 2018.

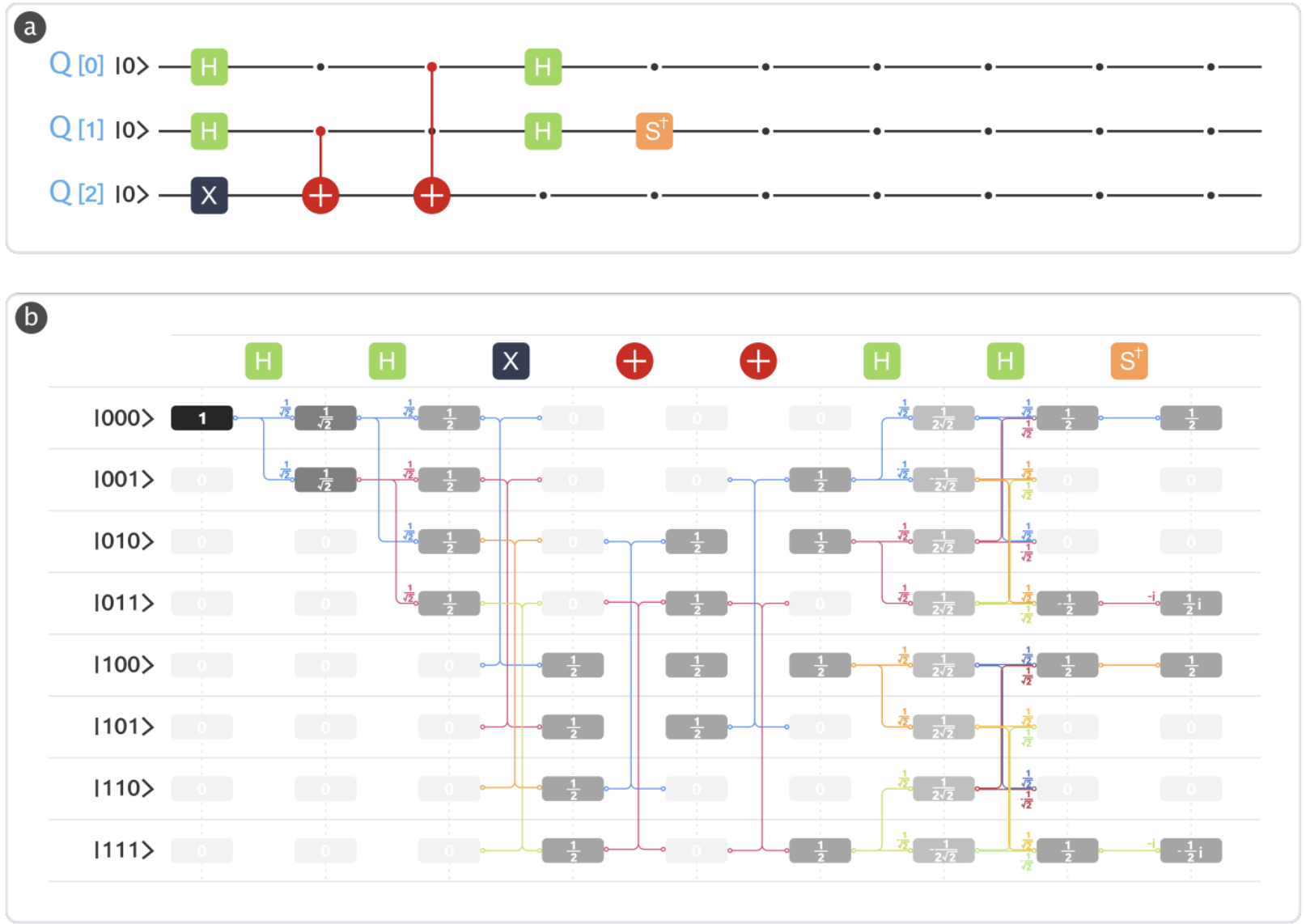
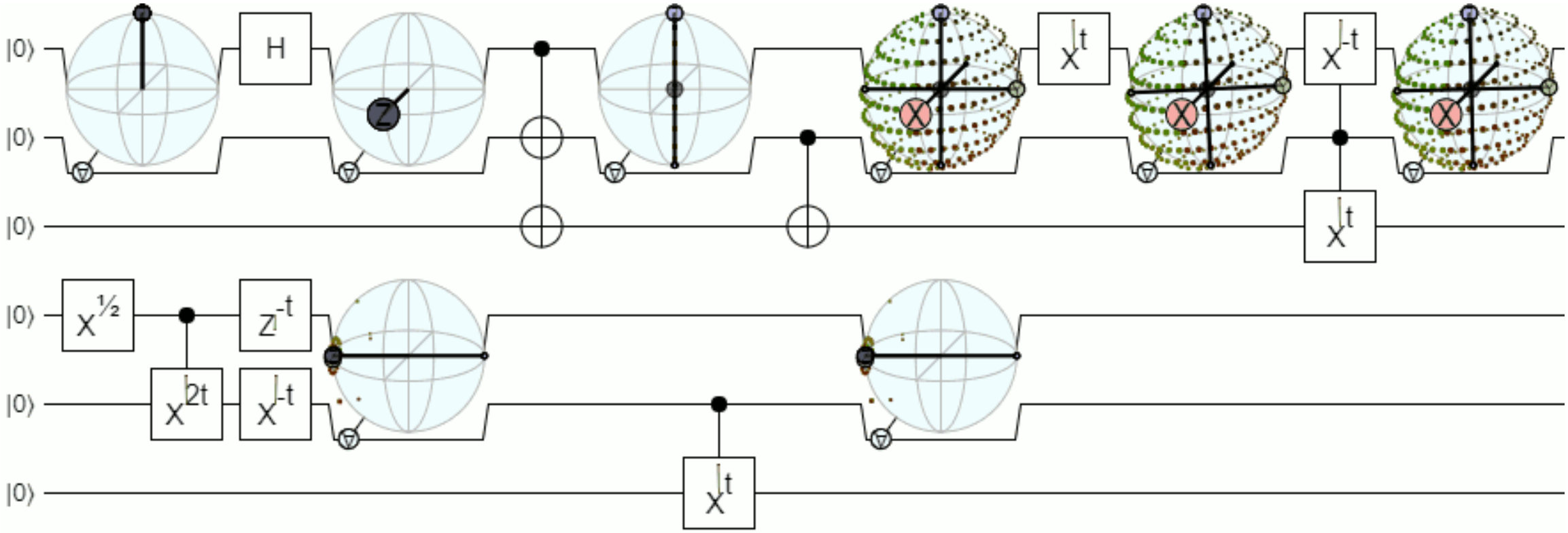


Figure 1: A parameter flow visualization of a three-qubit circuit's running process by QuFlow: a) a three-qubit circuit with 8 quantum gates including single-qubit gates (five H gates, one X gate and one S^+ gate) and a two-qubit gate (CNOT gate) ; b) the visualization result of circuit in a showing how the parameters of 8 possible outputs changes along the 8 gates' acting.

Craig Gidney, 2017

https://algassert.com/post/1716



Venus: A geometrical representation for quantum state visualization

Ruan, Shaolun and Yuan, Ribo and Guan, Qiang and Lin, Yanna and Mao, Ying and Jiang, Weiwen and Wang, Zhepeng and Xu, Wei and Wang, Yong

Computer Graphics Forum, 2023
arXiv preprint arXiv:2303.08366 (2023).

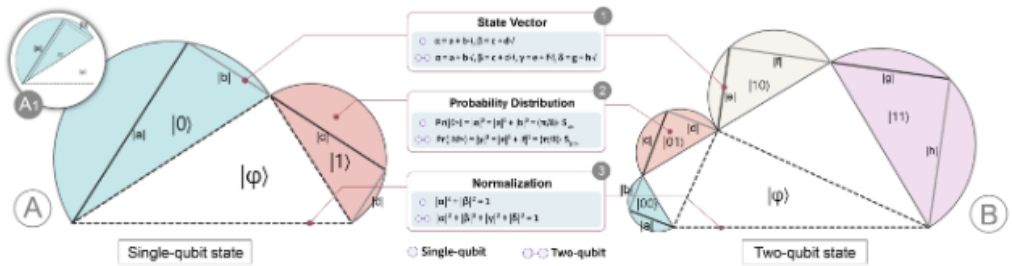


Figure 1: *The visual design of VENUS which supports single-qubit (A) and two-qubit (B) state representation based on the same visualization form.* ***Line segments*** *visualize the state vector, where the black line denotes the real part, and the grey line denotes the imaginary part based on Equation 1.* ***Semicircles****'s area indicates the probability of measuring the corresponding state based on Equation 2.* ***Triangle base****'s length consistently equals to 1, because it encodes based on the constraint of normalization (e.g., Equation 3).*

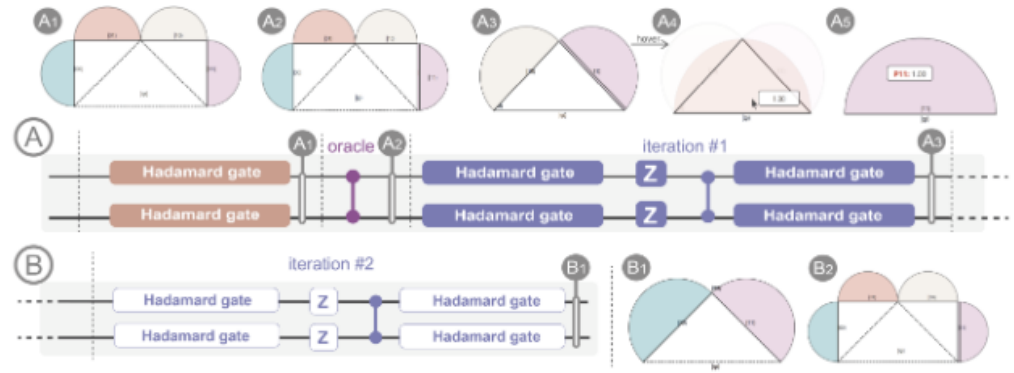


Figure 4: *The case for the two-qubit quantum algorithm, i.e., Grover's Algorithm. (A) The calculation process of Grover's Algorithm with one iteration, along with four consecutive quantum states and an interaction shown by VENUS. (B) The Grover's Algorithm with one more iteration appended after the original circuit, along with two quantum states representation.*

VACSEN: A Visualization Approach for Noise Awareness in Quantum Computing

Ruan, Shaolun, Yong Wang, Weiwen Jiang, Ying Mao, and Qiang Guan.

IEEE Transactions on Visualization and Computer Graphics, vol. 29, no. 1, pp. 462-472, Jan. 2023

https://arxiv.org/pdf/2207.14135.pdf

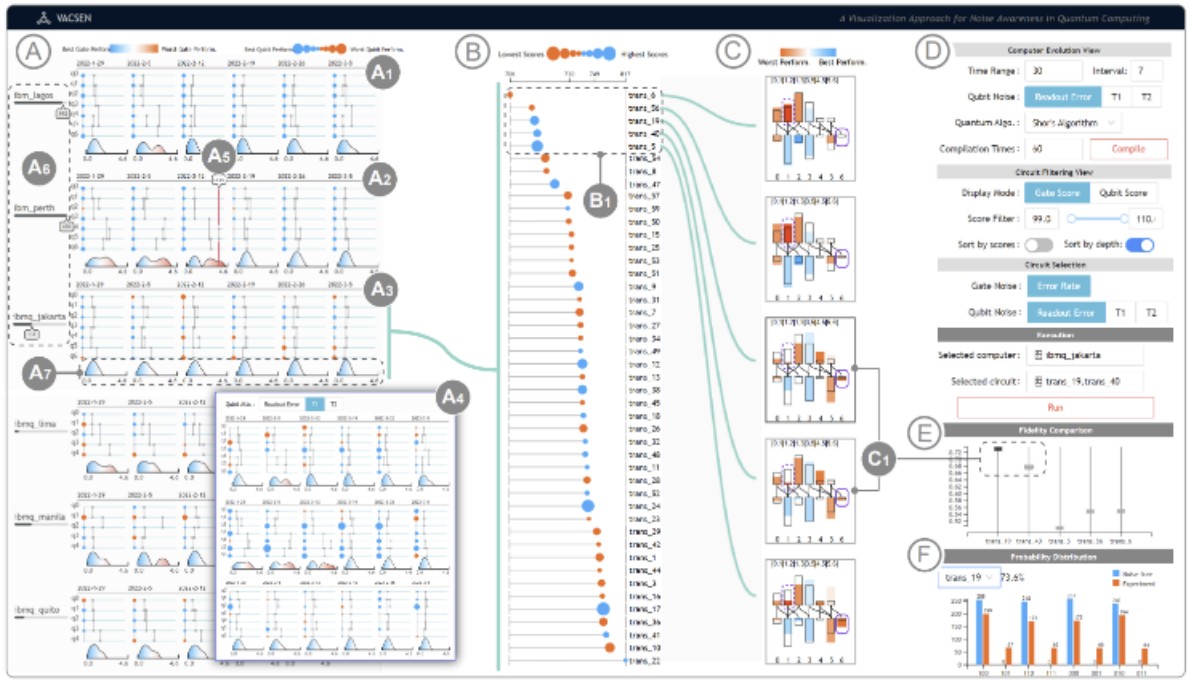


Fig. 1: The interface of *VACSEN* makes users aware of the quantum noise via three linked views (A-C). Computer Evolution View (A) allows the assessment for all quantum computers based on a temporal analysis for multiple performance metrics. Circuit Filtering View (B) supports the filtering for the potential optimal compiled circuits. Circuit Comparison View (C) supports the in-depth comparison regarding the performance of qubits or quantum gates and corresponding usages. The control panel (D) allows users to interactively configure the settings of *VACSEN*. Fidelity Comparison View (E) shows the fidelity distribution of each compiled circuit. Probability Distribution View (F) visualizes the results of state distribution of a quantum circuit execution.

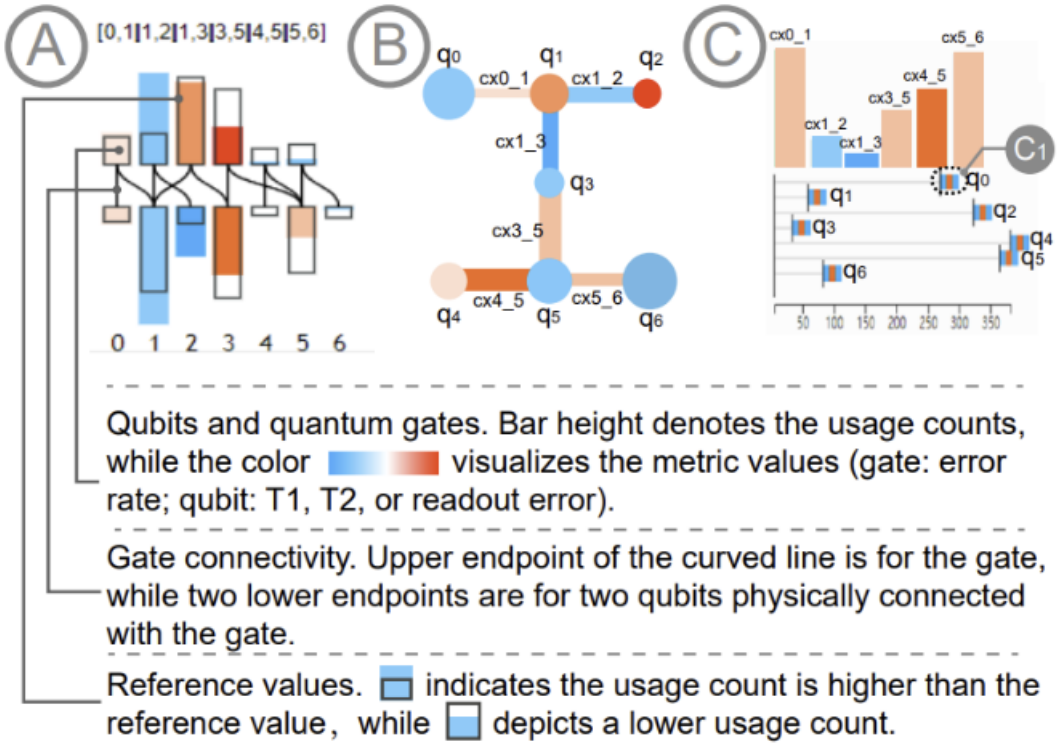


Fig. 5: Circuit Comparison View. (A) The Circuit Comparison View supports the in-depth comparison of multiple compiled circuits. (B)(C) Design alternatives of Circuit Comparison View. Both alternative designs cannot support an effective comparison of the usage times of qubits and quantum gates.

Quantivine: A Visualization Approach for Large-scale Quantum Circuit Representation and Analysis

Zhen Wen, Yihan Liu, Siwei Tan, Jieyi Chen, Minfeng Zhu, Dongming Han, Jianwei Yin, Mingliang Xu, and Wei Chen
IEEE VIS 2023

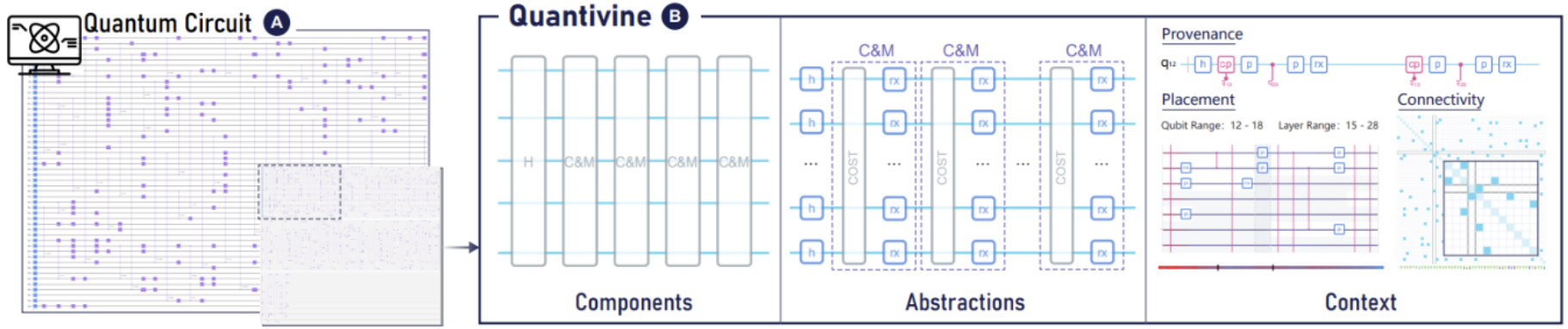


Fig. 1: Quantum circuit visualizations for a 50-qubit quantum circuit. (A) Typical quantum circuit diagram generated by Qiskit [67]. (B) The resulting visualizations of *Quantivine*, including structured components, pattern abstractions, and context enhancement.

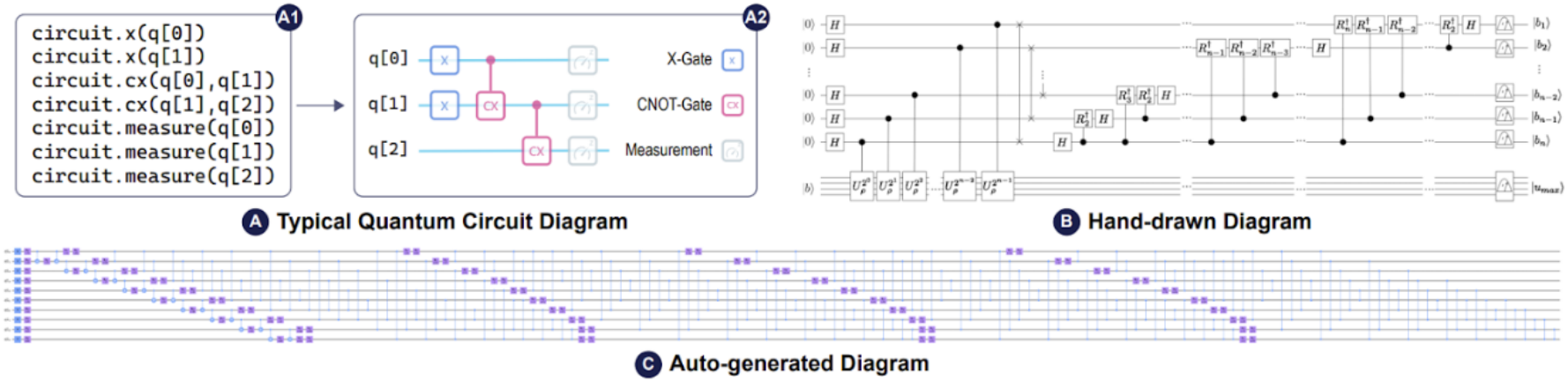


Fig. 2: The illustration of quantum circuit diagrams. (A) A typical quantum programming code (A1) and its corresponding quantum circuit diagram (A2). (B) A hand-drawn quantum circuit diagram depicts the implementation of quantum PCA algorithm [49]. (C) An auto-generated diagram (using Qiskit library [67]) visualizes a quantum circuit having 10 qubits and 306 quantum gates.

Resources

- Book:
  - Michael Nielsen, Isaac Chuang, Quantum Computation and Quantum Information, 2010
- Free books:
  - Thomas G. Wong, Introduction to Classical and Quantum Computing, 2022, http://www.thomaswong.net/
  - Venkateswaran Kasirajan, Fundamentals of Quantum Computing: Theory and Practice, 2021, https://link.springer.com/book/10.1007/978-3-030-63689-0?&utm_medium=display&utm_source=criteo&utm_campaign=3_fjp8312_product_us_springerlink&utm_content=us_banner_29012020
  - Ciaran Hughes, Joshua Isaacson, Anastasia Perry, Ranbel F. Sun, Jessica Turner, Quantum Computing for the Quantum Curious, 2021, https://library.oapen.org/bitstream/handle/20.500.12657/48236/9783030616014.pdf
- Non-technical explanation of Shor's algorithm:
  - https://scottaaronson.blog/?p=208
- Playlist:
  - https://www.youtube.com/playlist?list=PLWDaTqG3UM0Ow0QWaxeeYTmr0hyF1duRl