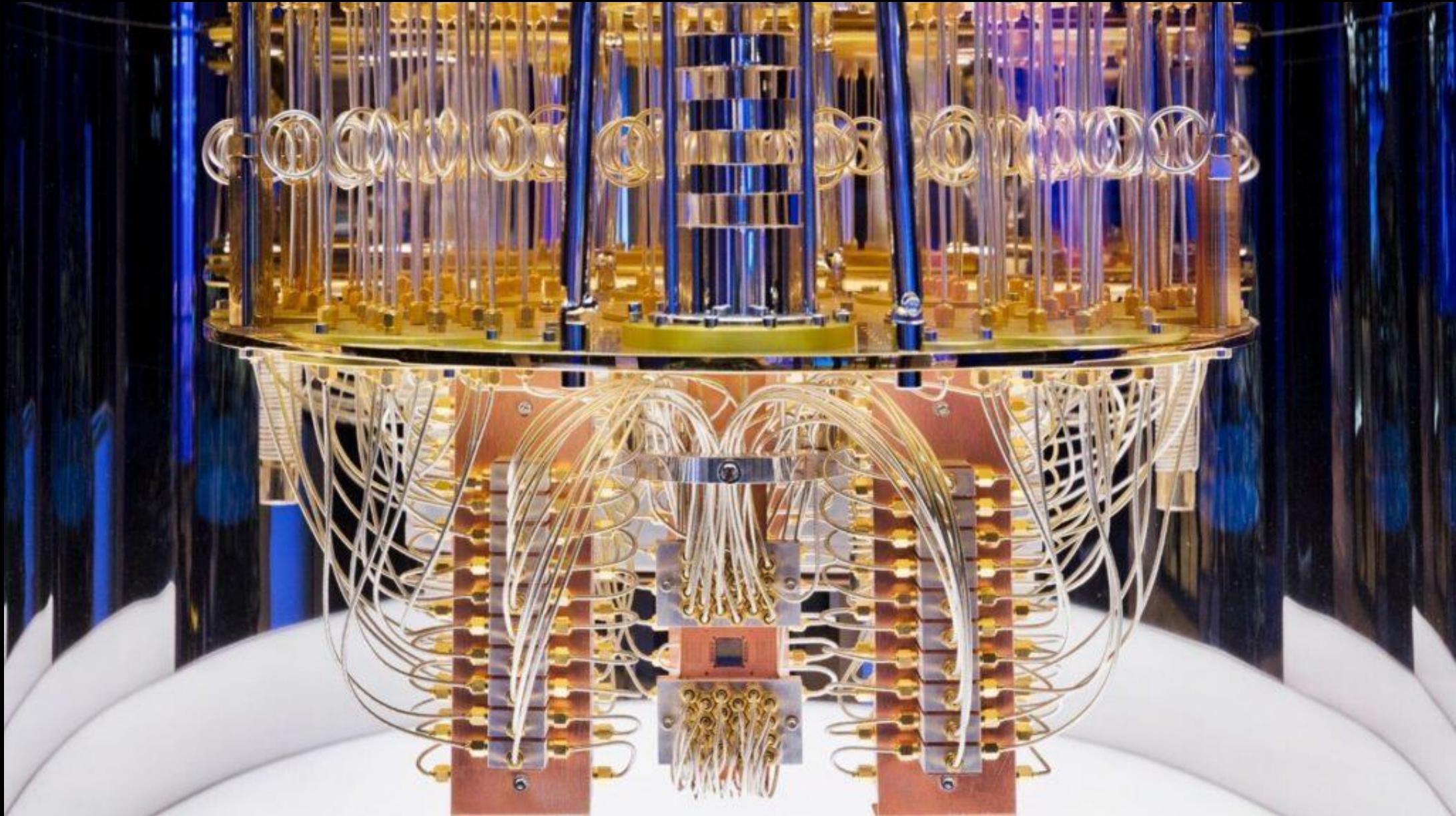


# Simulating and Visualizing Quantum Circuits

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

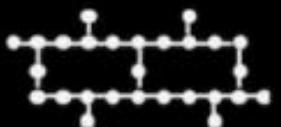


2021  
**Eagle**  
127 Qubits

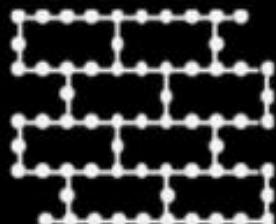


2022  
**Osprey**  
433 Qubits

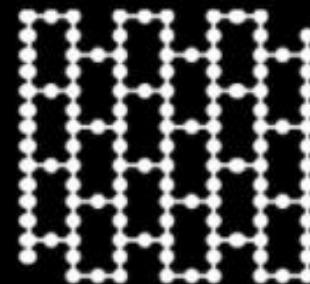
**Falcon:**  
27 qubits



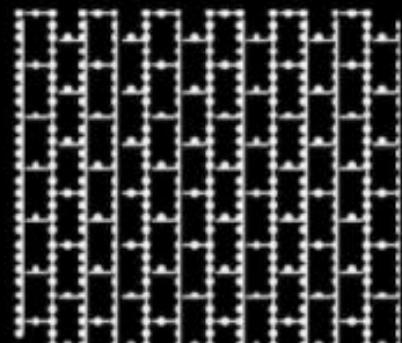
**Hummingbird:**  
65 qubits



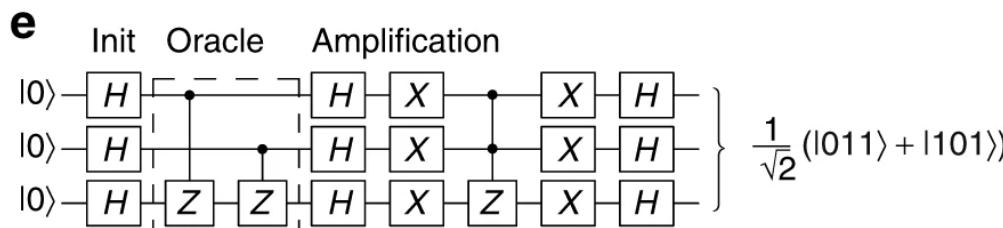
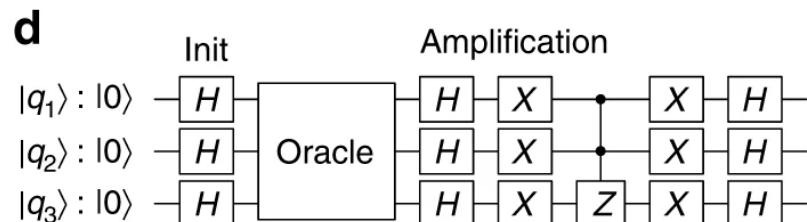
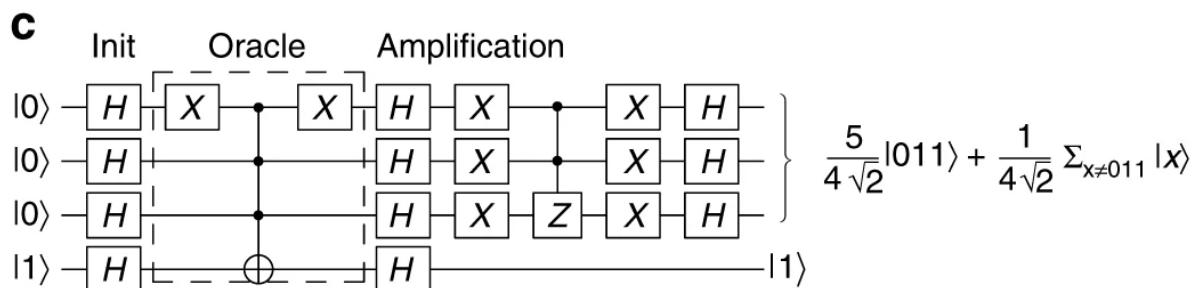
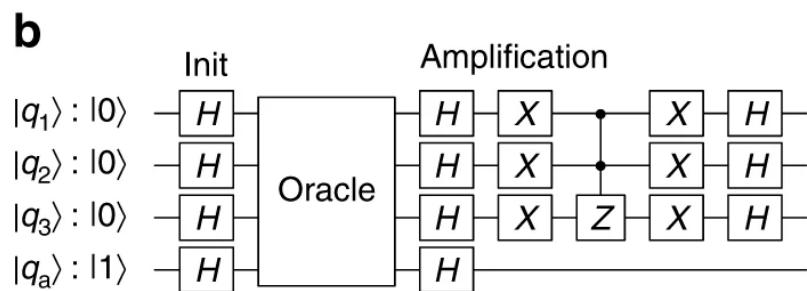
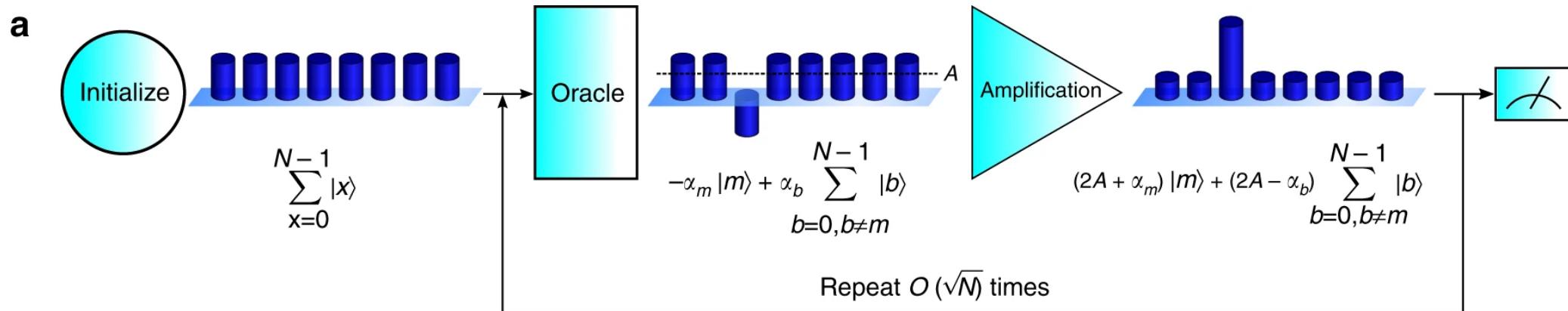
**Eagle:**  
127 qubits



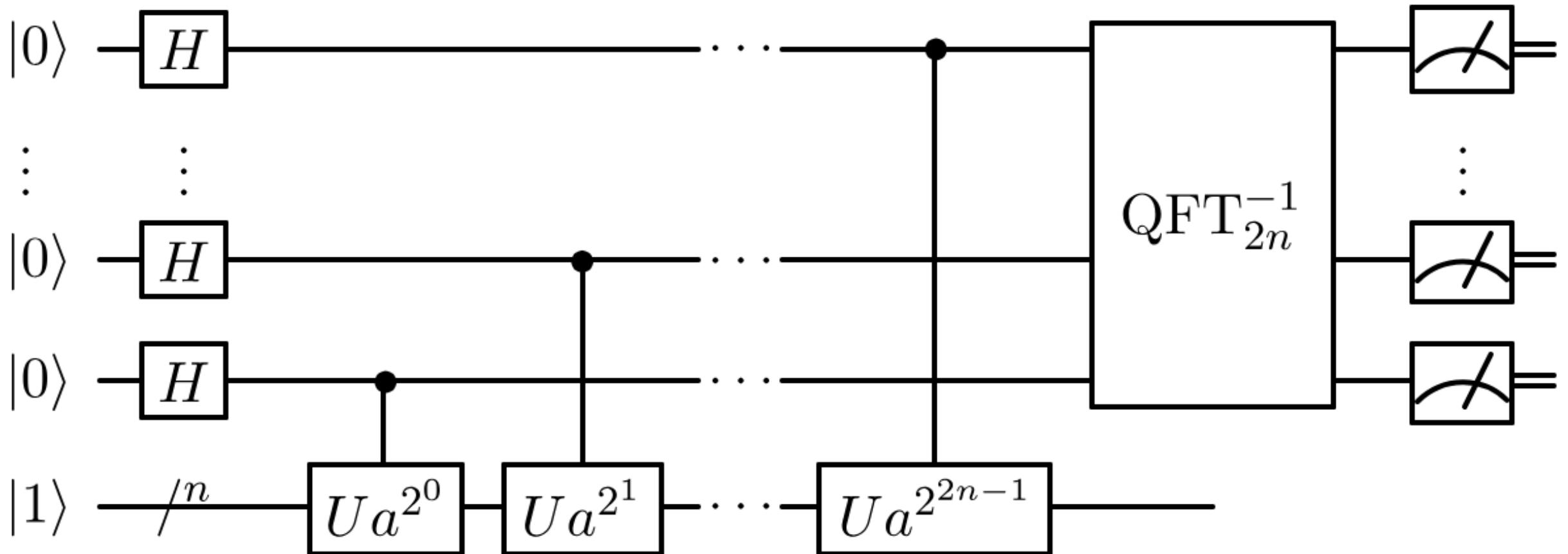
**Osprey:**  
433 qubits



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



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



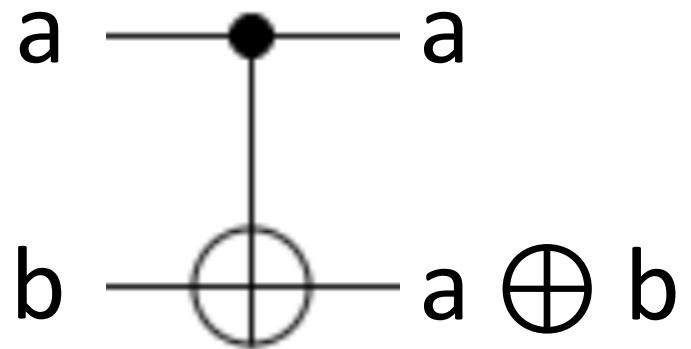
- Qubits can be implemented using photon polarization, particle spin, superconducting current loops...
- Qubits can store 0 and 1 in superposition. Measuring the state of a qubit causes it to collapse to 0 or 1. Qubits can also be entangled (i.e., there is a correlation between their measured values).
- 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

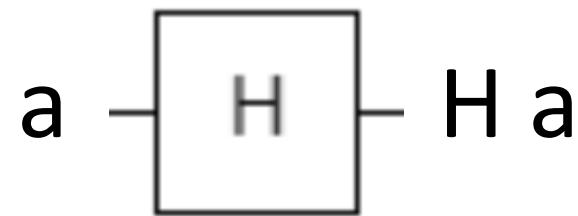
# CNOT “controlled-not” gate (similar to XOR gate)



<b>a</b>	<b>b</b>	<b><math>a \oplus b</math></b>
0	0	0
0	1	1
1	0	1
1	1	0

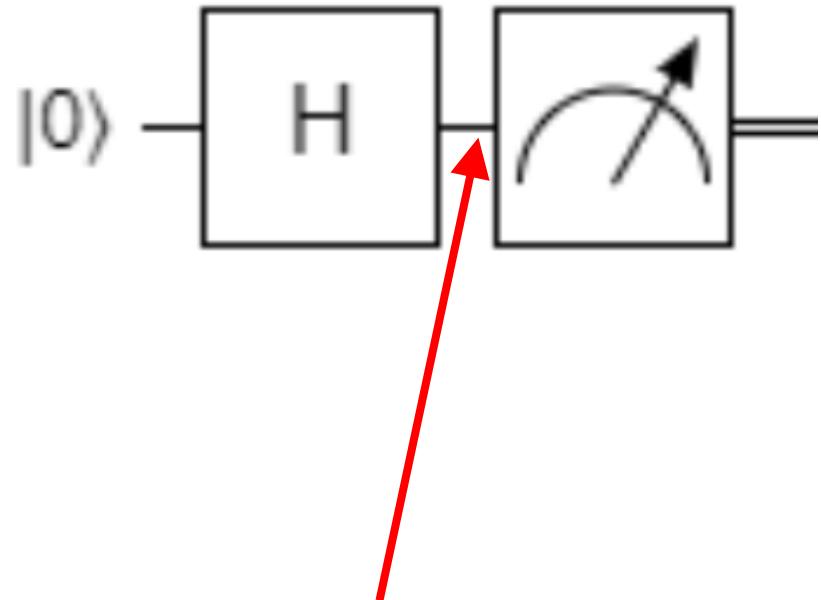
*It seems like the second qubit has no causal effect on the first qubit, that the direction of causality is from first to second qubit. However, as we will see, with superposition and entanglement, this gate does strange things.*

Hadamard gate: one of many gates that create superposition



a	Ha
0	0 and 1
1	0 and 1

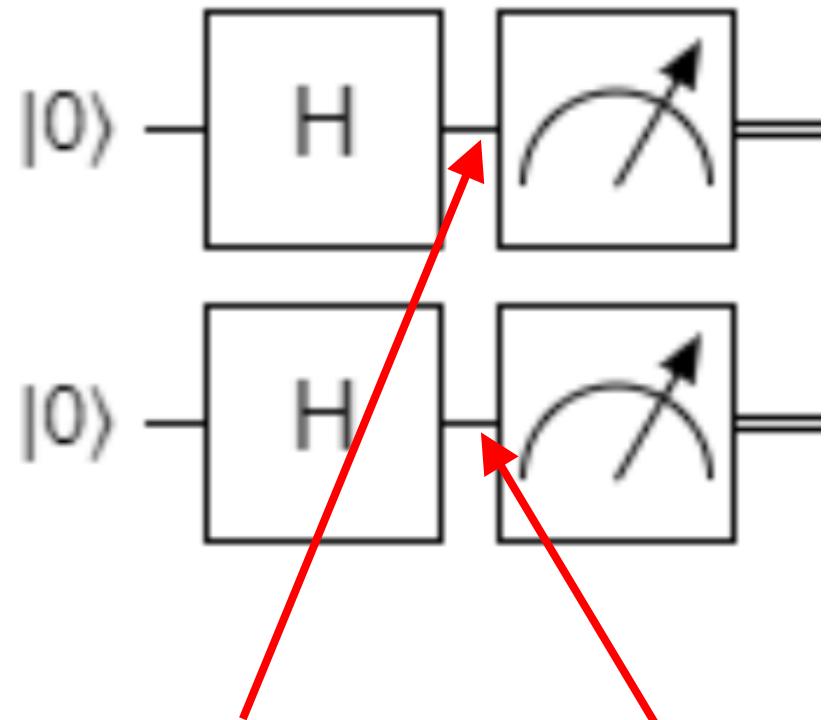
Measurement of a qubit in superposition can result in 0 or 1



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

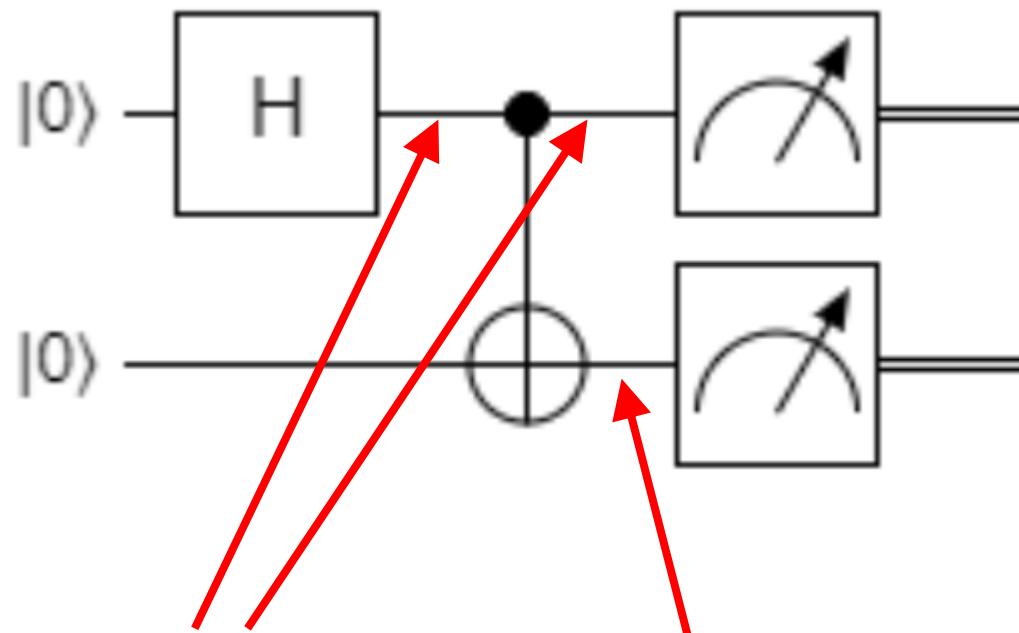


Superposition  
of 0 and 1

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

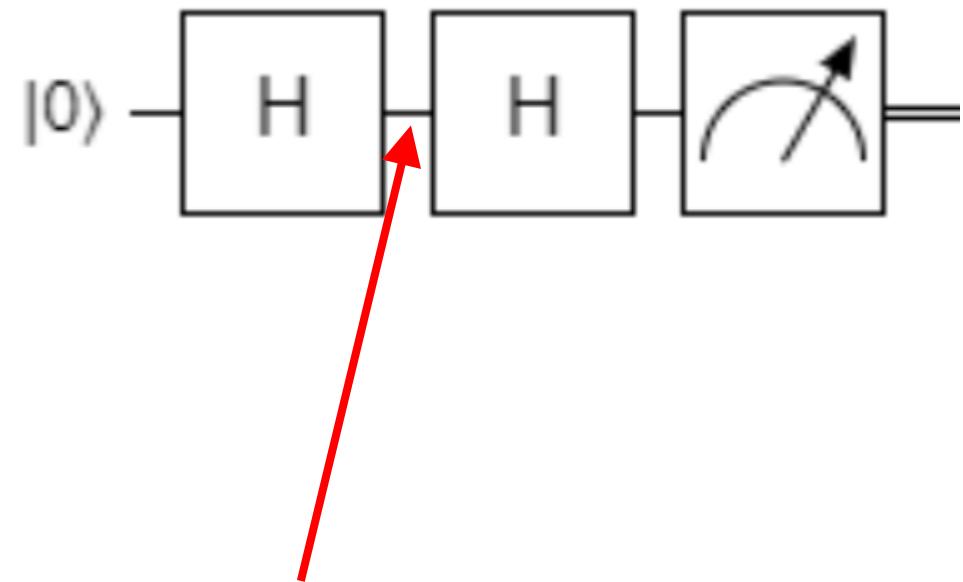


Superposition  
of 0 and 1

Superposition  
of 0 and 1

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

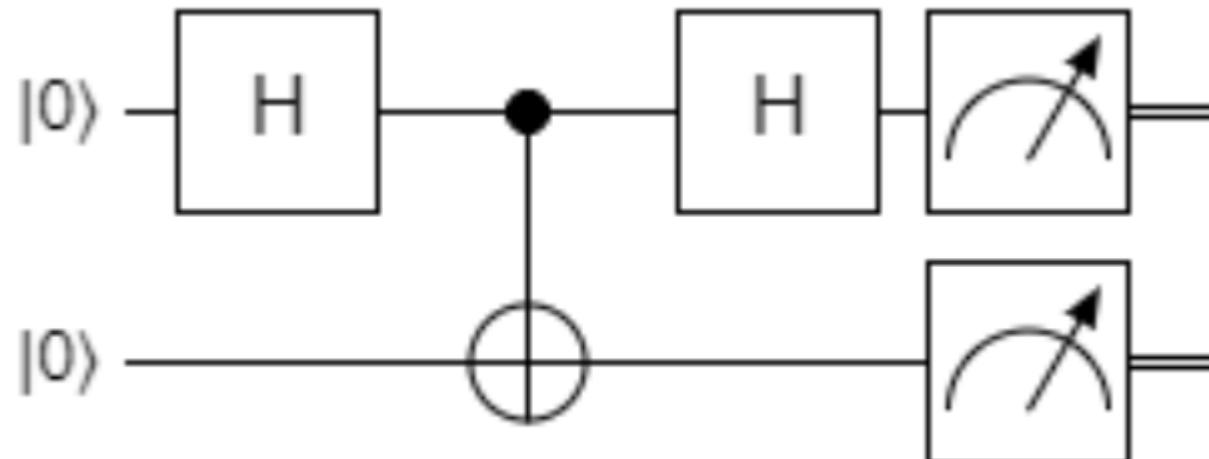
Hadamard is its own inverse.  $H^2 = I$



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.

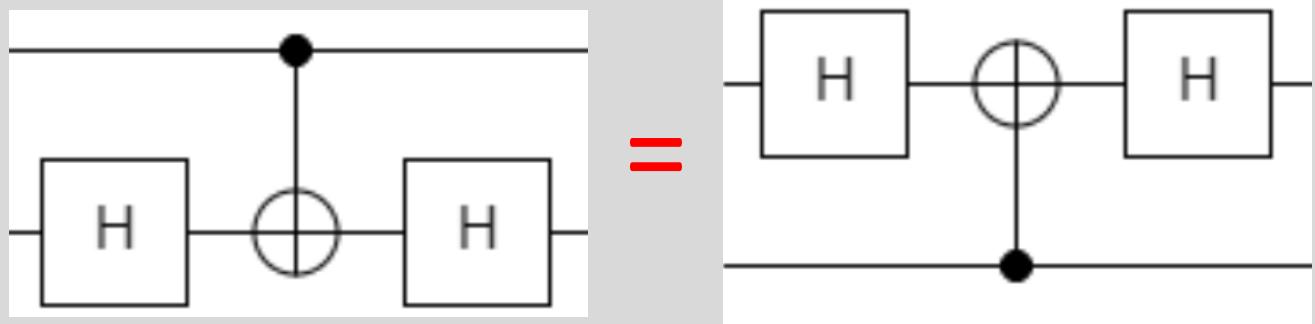
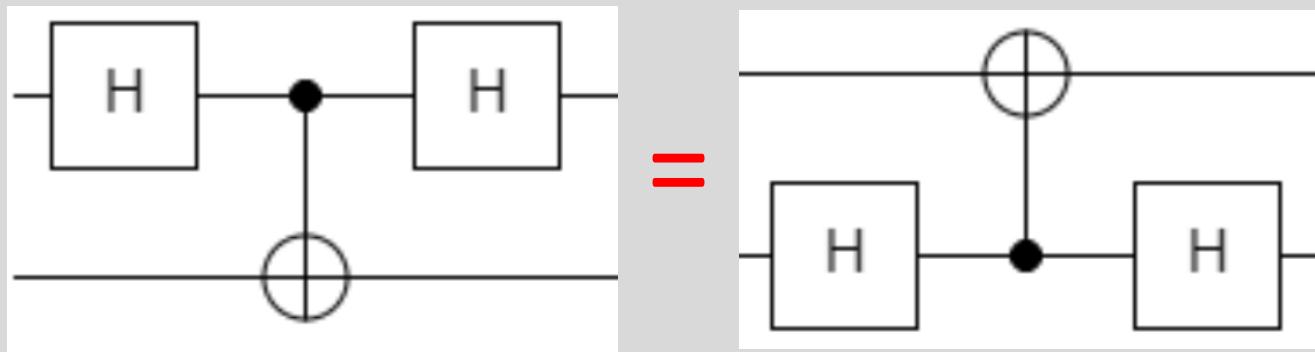
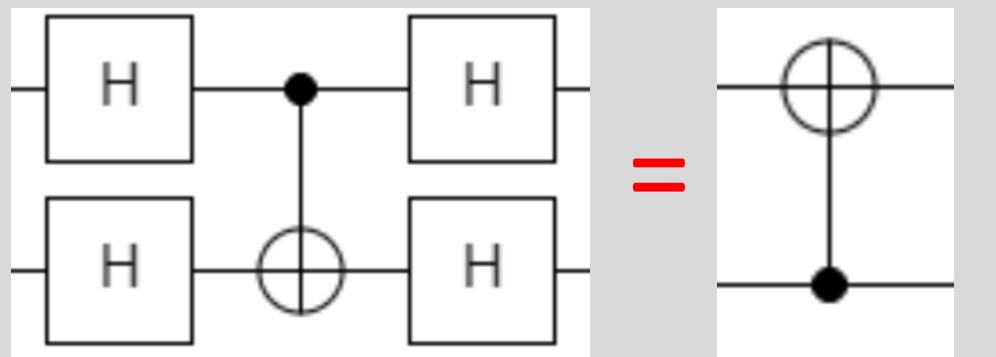
The control bit does not just “read” !!



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=Cj41yWg38Oo>

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

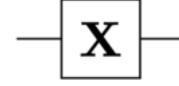
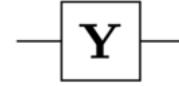
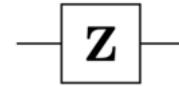
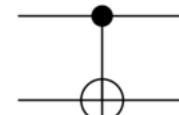
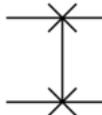
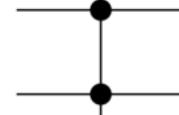
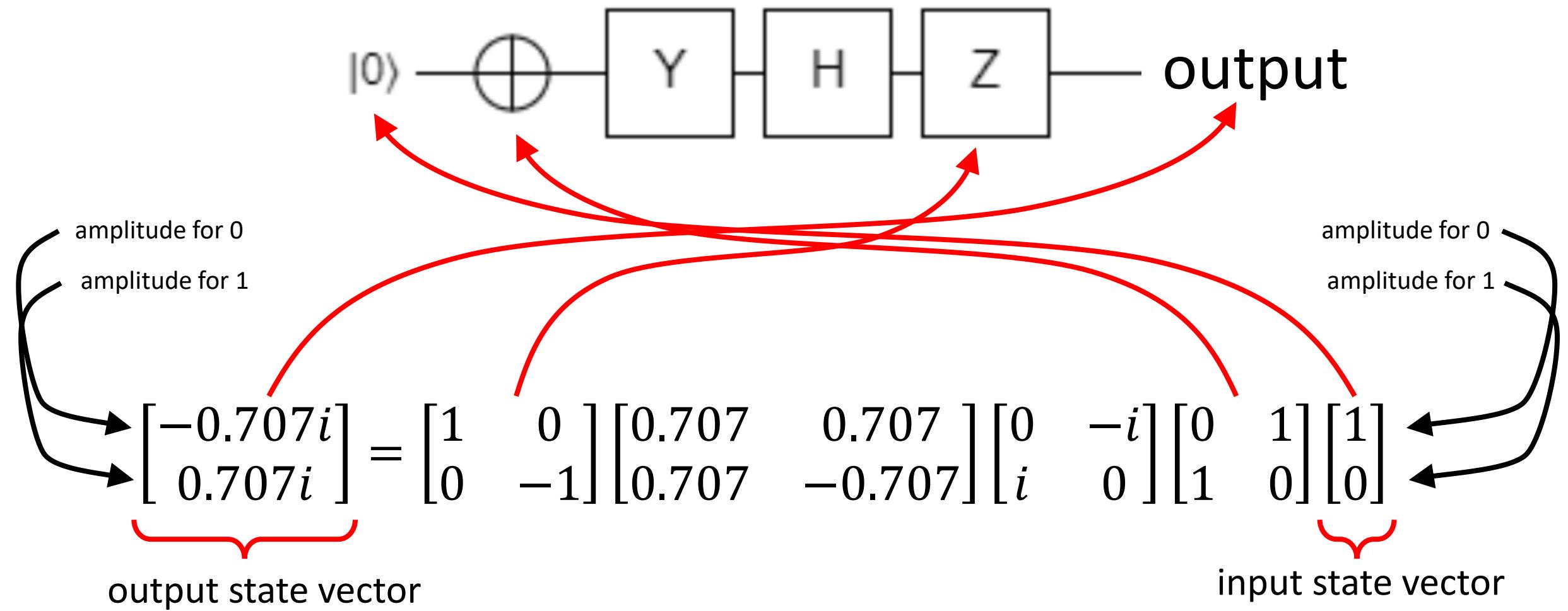
Operator	Gate(s)	Matrix
<b>Pauli-X (X)</b>	 	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
<b>Pauli-Y (Y)</b>		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
<b>Pauli-Z (Z)</b>		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
<b>Hadamard (H)</b>		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1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

Table adapted from  
[https://en.wikipedia.org/wiki/Quantum\\_logic\\_gate](https://en.wikipedia.org/wiki/Quantum_logic_gate)

The processing of qubits can be described with matrix multiplications



# Bra and ket notation

- $\langle \psi |$  is read as “bra psi”, and means the state psi as a row vector
- $|\psi\rangle$  is “ket psi”, the state psi as a column vector
- $\langle a | b \rangle$  means the dot product of the row vector a with the column vector b, for example:

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

- $|b\rangle\langle a|$  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

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

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

amplitude for 0  
amplitude for 1

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

↑  
amplitudes (complex numbers)  
↑  
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

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

state vector →      ↑ amplitudes      ↓  
 ↑ base state vectors

- 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  $\alpha$  is positive real. This makes the two degrees of freedom,  $\alpha$  and  $\varphi$ , 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

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

- Certain superposed states have special names:

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

- Example:

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

state vector →

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

↑ amplitudes

↑ base state vectors

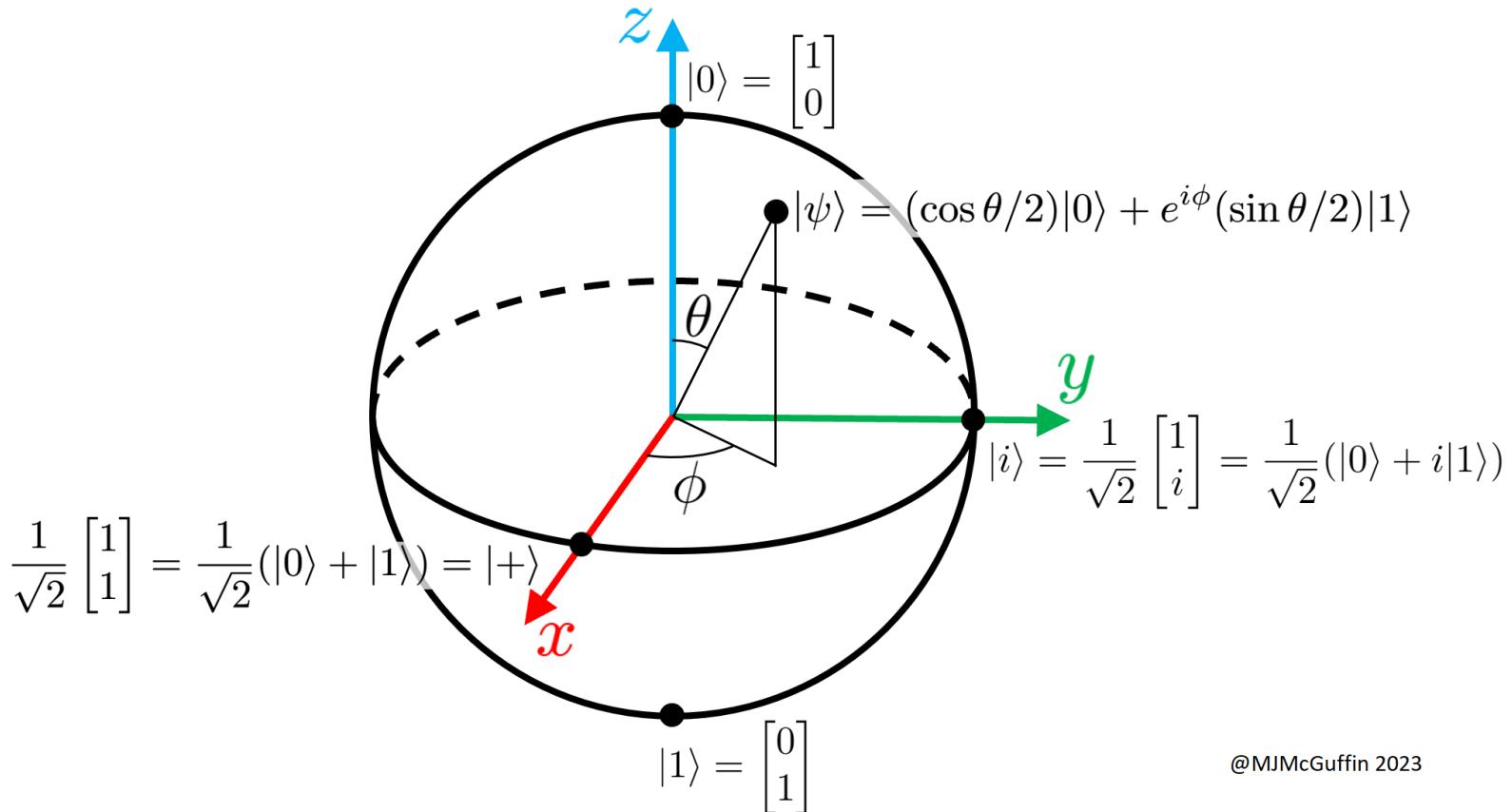
# Bloch sphere: to visualize one qubit

We apply a global phase shift to obtain the form

$$|\psi\rangle = \cos(\theta/2) |0\rangle + e^{i\phi} \sin(\theta/2) |1\rangle$$

which has two degrees of freedom,  $\theta$  and  $\phi$ , mapped to a surface.

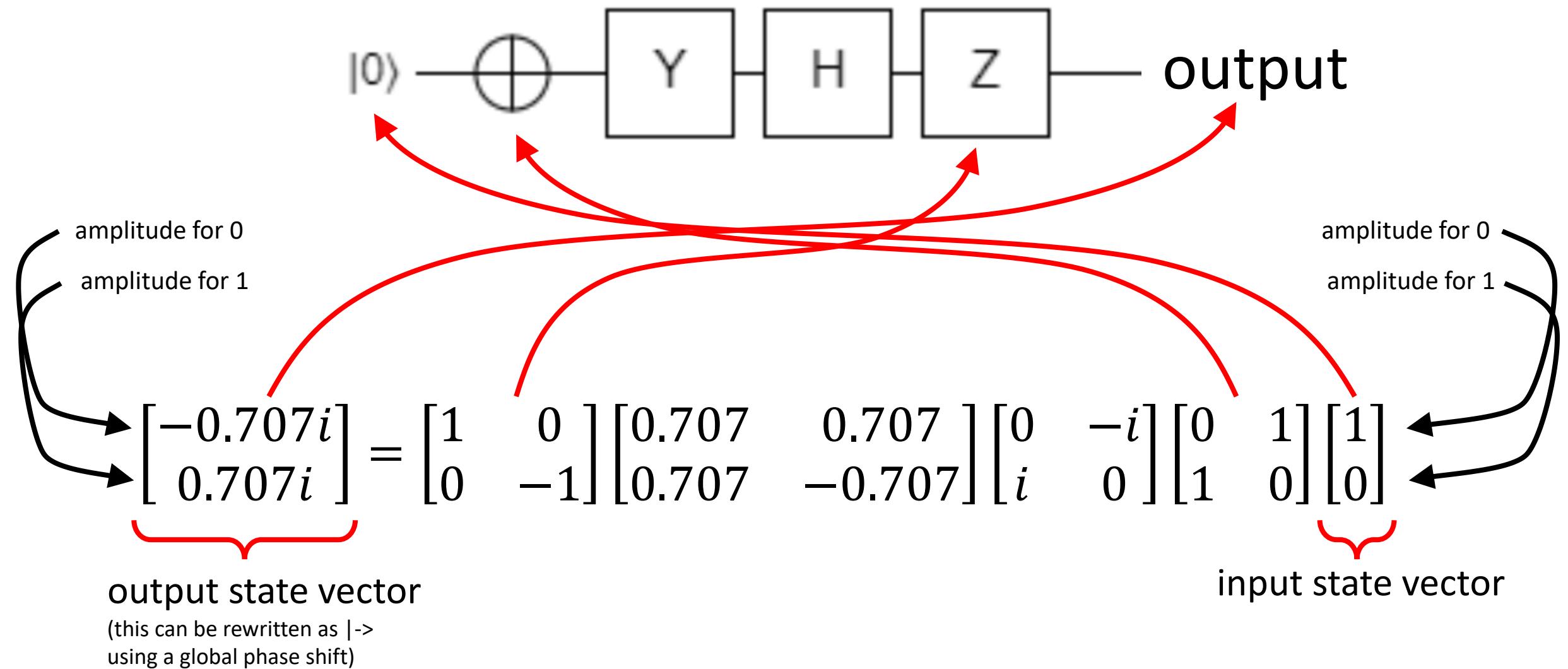
The vertical position of the point (encoded with  $\theta$ ) corresponds directly to the probability of measuring 0 or 1.  $\phi$  is the *relative phase*, and is only meaningful and measurable if there are other qubits.



@MJMcGuffin 2023

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

The processing of qubits can be described with matrix multiplications



```

// 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](https://github.com/MJMcGuffin/muqcs.js)

```

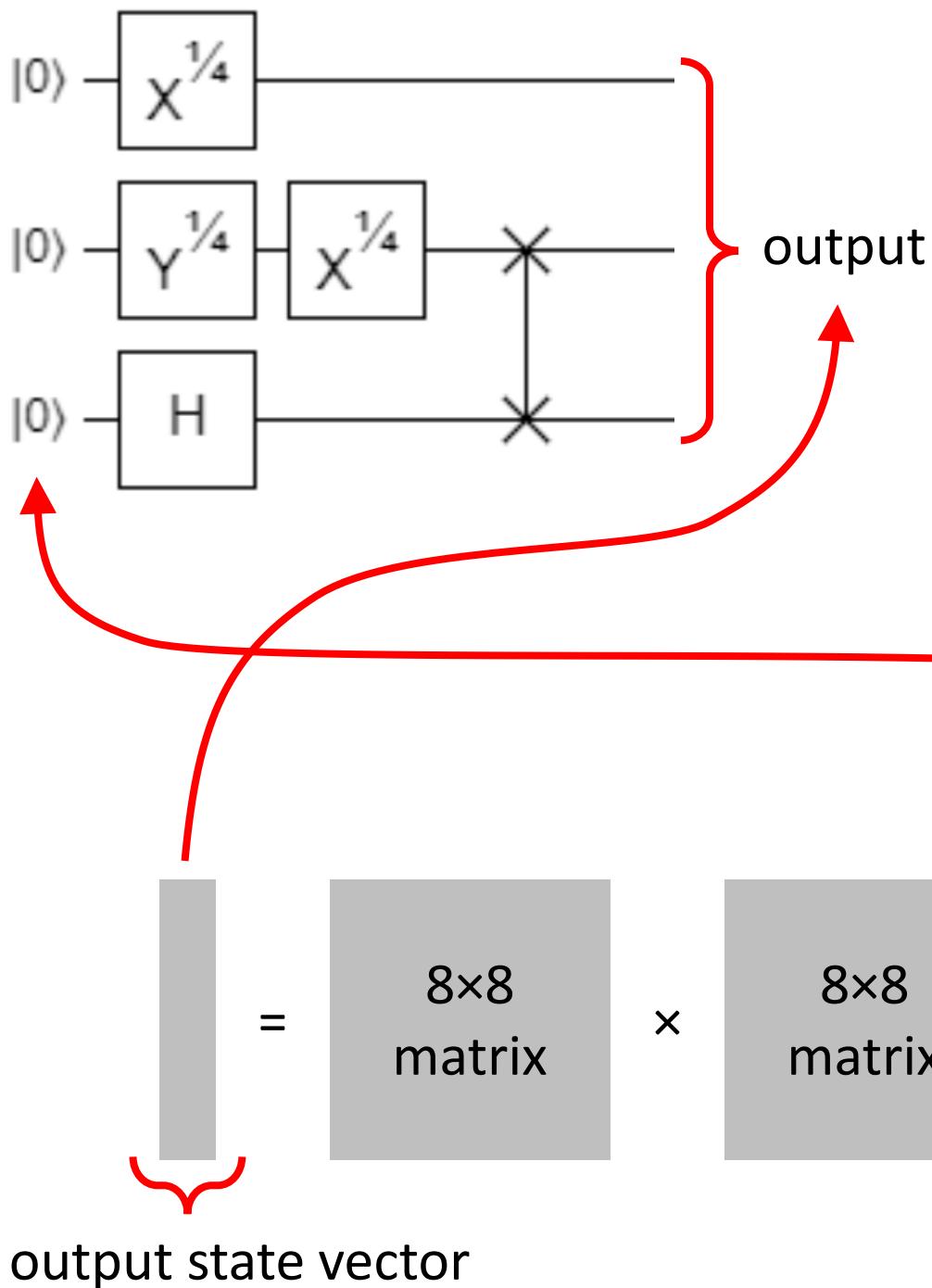
[1,0 ] * [0.707,0.707 ] * [0 , -1i] * [_ , 1] * [1] = |0>[-0.707i]
[0,-1] [0.707,-0.707] [1i,0 ] [1,_ ] [_ ] |1>[0.707i ]

```

My code's output

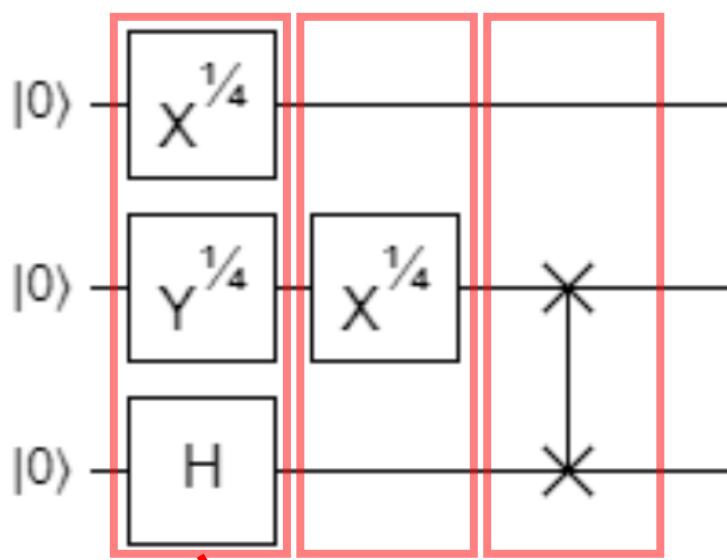
Simulation  
with Quirk



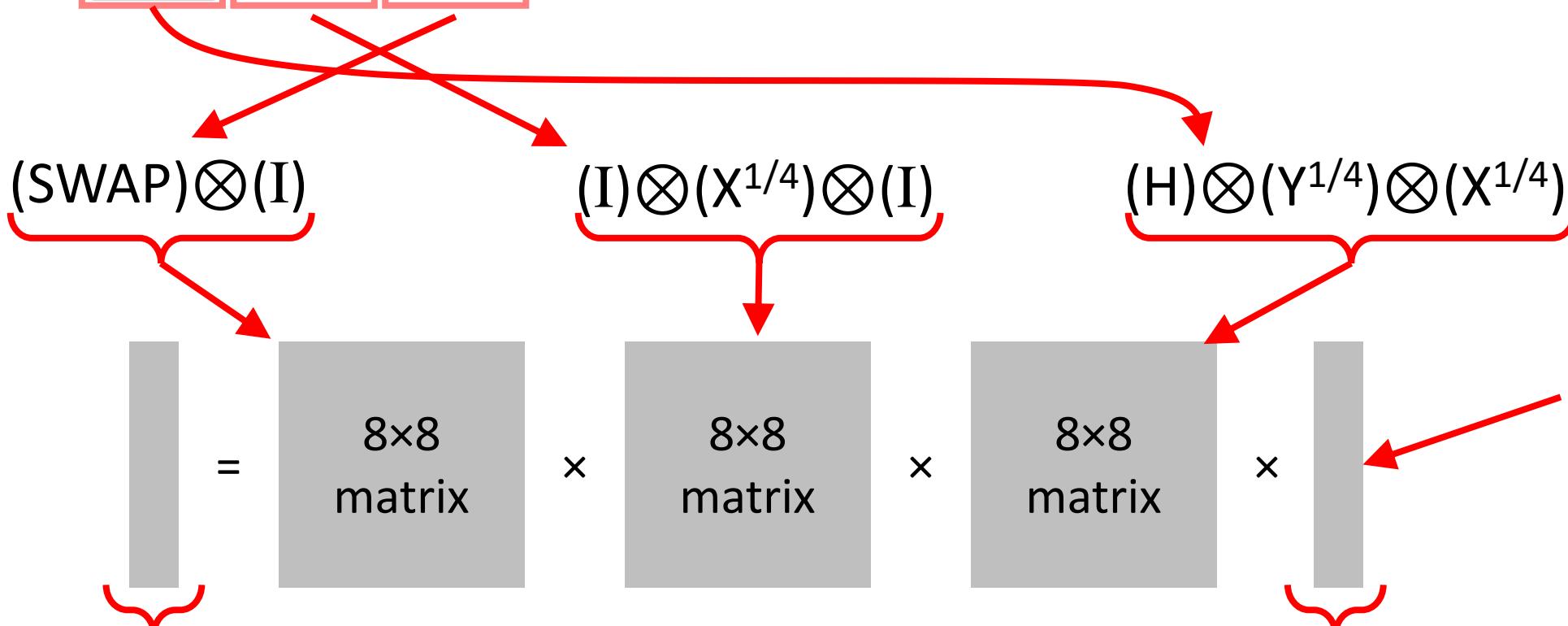


The processing of qubits can be described with matrix multiplications

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



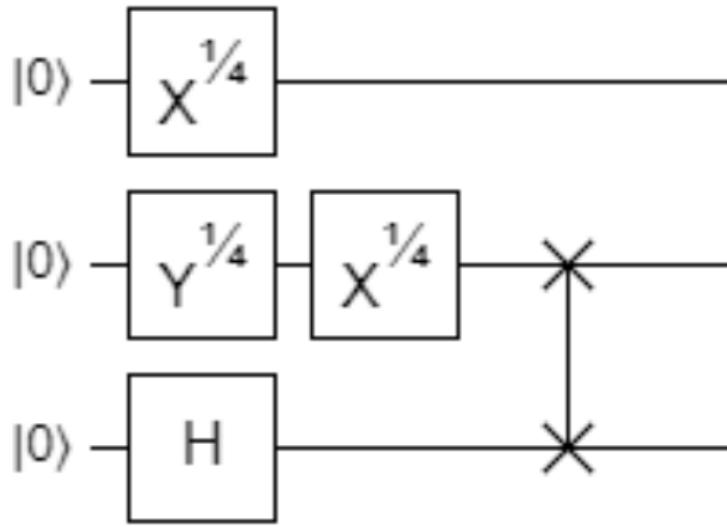
Each  $8 \times 8$  matrix is a tensor product (or kronecker product) of smaller matrices



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

$8 \times 1$  output state vector

$8 \times 1$  input state vector

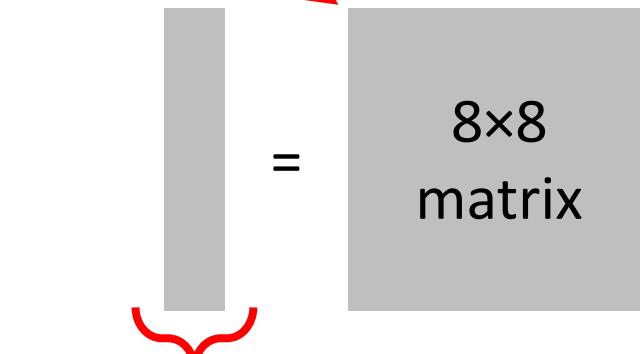


Each  $8 \times 8$  matrix is a tensor product (or kronecker product) of smaller matrices

$$\underbrace{(SWAP)}_{4 \times 4} \otimes \underbrace{(I)}_{2 \times 2}$$

$$(I) \otimes \underbrace{(X^{1/4})}_{2 \times 2} \otimes (I)$$

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



$$= \begin{matrix} 8 \times 1 \\ \text{output state vector} \end{matrix} \times \begin{matrix} 8 \times 8 \\ \text{matrix} \end{matrix} \times \begin{matrix} 8 \times 8 \\ \text{matrix} \end{matrix} \times \begin{matrix} 8 \times 1 \\ \text{input state vector} \end{matrix}$$

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

# Tensor product or kronecker product

$$\begin{array}{|c|c|} \hline a & b \\ \hline c & d \\ \hline \end{array} \otimes \begin{array}{|c|c|} \hline w & x \\ \hline y & z \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline a w & a x & b w & b x \\ \hline a y & a z & b y & b z \\ \hline c w & c x & d w & d x \\ \hline c y & c z & d y & d z \\ \hline \end{array}$$

- 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

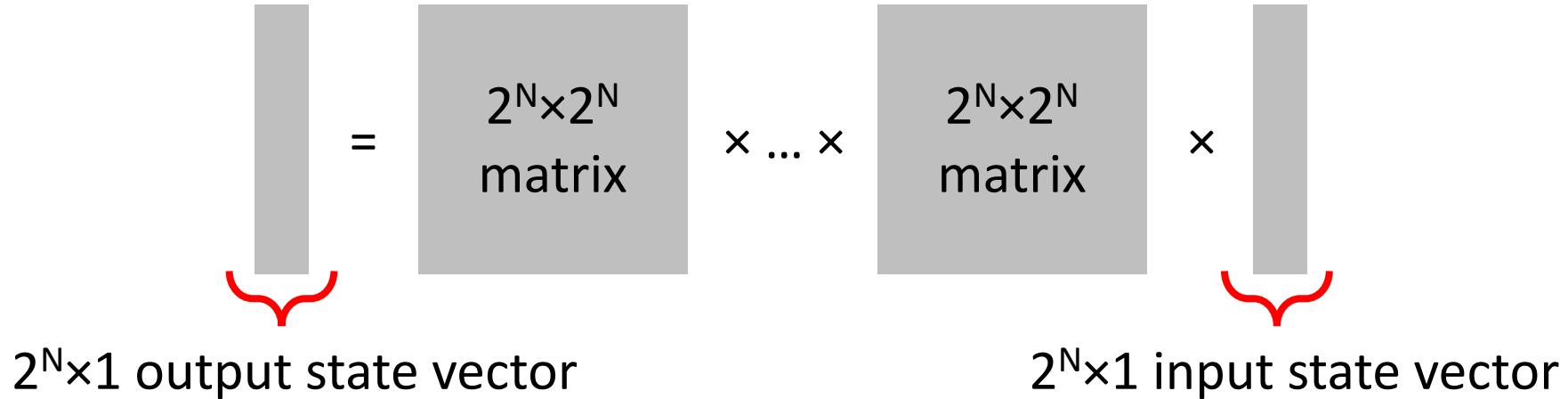
Example: the  $8 \times 1$  input state vector for this circuit is a tensor product

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

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

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})
));

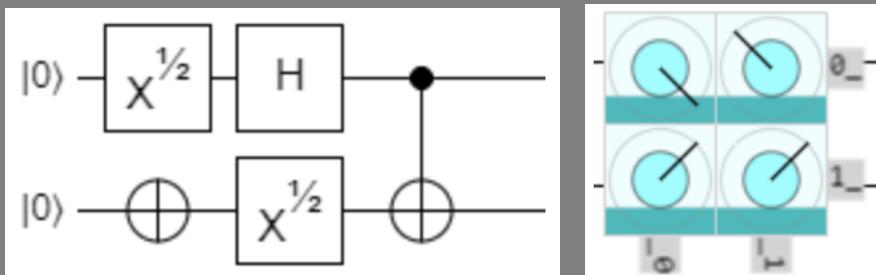
```

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



```

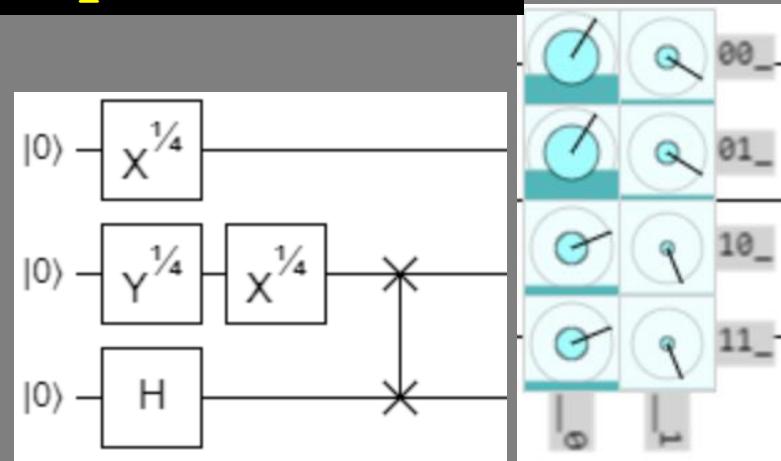
// qubit q0 |0>----(x^0.25)-----
//
// qubit q1 |0>----(y^0.25)-----(x^0.25)----X-----
//                                |
// qubit q2 |0>-----H-----X-----
//
input = CMatrix.naryTensor( [ CMatrix.ketZero /*q2*/, CMatrix.ketZero /*q1*/, CMatrix.ketZero /*q0*/ ] );
step1 = CMatrix.naryTensor( [ CMatrix.gate2x2hadamard /*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 ,0 ]	[1]  000>[0.302+0.479i]
[_,1,_,_,_,_,_,_]	[0 ,0.9+0.4i,0 ,0.1-0.4i,0 ,0 ,0 ,0 ,0 ]	[_]  001>[0.198-0.125i]
[_,_,_,_,1,_,_,_]	[0.1-0.4i,0 ,0.9+0.4i,0 ,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 ,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]

$(m) \otimes (I) \otimes (I)$

$(I) \otimes (m) \otimes (I)$

$(I) \otimes (I) \otimes (m)$

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 = CMMatrix.gate2x2identity;
m = CMMatrix.create([[1,2],[3,4]]);
m0 = CMMatrix.naryTensor([m,identity,identity,identity]);
m1 = CMMatrix.naryTensor([identity,m,identity,identity]);
m2 = CMMatrix.naryTensor([identity,identity,m,identity]);
m3 = CMMatrix.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,_,_2,_,_,_]	[_,_,_,1,2,_,_,_]	[_,_,_1,2,_,_,_]
[_,_,_,_,_,1,_2,_]	[_,3,_4,_,_2,_]	[_,_,_,_,1,2,_]	[_,_,_3,4,_]
[_,_,_,_,_,_,1,2,_]	[_,_,3,_4,_,_2]	[_,_,_,_,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]	[_,_,_,_,1,2]
[_,_,3,_4,_]	[_,_,_,1,2]	[_,_,_,3,4]	[_,_,_,1,2]
[_,_,_,3,4]	[_,_,_,_,1,2]	[_,_,_,3,4]	[_,_,_,_,3,4]

$m \otimes I \otimes I \otimes I$

$I \otimes m \otimes I \otimes I$

$I \otimes I \otimes m \otimes I$

$I \otimes I \otimes I \otimes m$

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 CMMatrix.

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

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

$m \otimes I \otimes I \otimes I \otimes I$

$I \otimes m \otimes I \otimes I \otimes I$

$I \otimes I \otimes m \otimes I \otimes I$

$I \otimes I \otimes I \otimes m \otimes I$

$I \otimes I \otimes I \otimes I \otimes m$

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 CMMatrix.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})
));

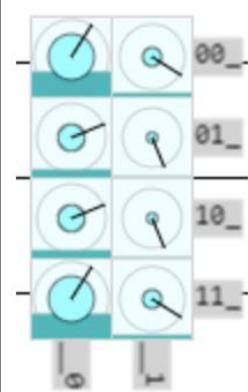
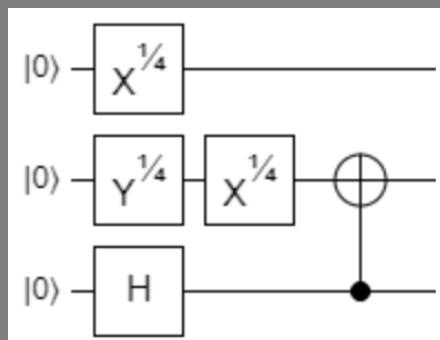
```

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 ,0.9+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]]);

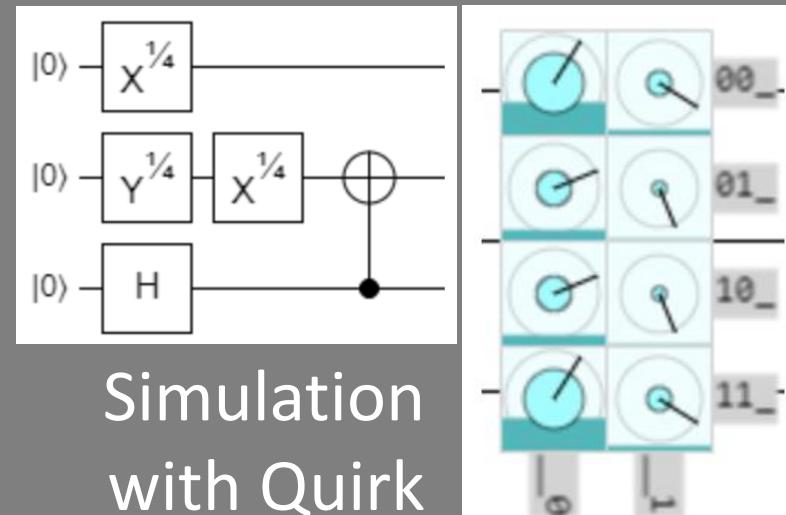
console.log(StringUtil.concatenateMultilineStrings(
    input.toString(),
    " -> ", step1.toString(),
    " -> ", step2.toString(),
    " -> ", output.toString({binaryPrefixes:true})
));

```

[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

My code, with the  
second optimization



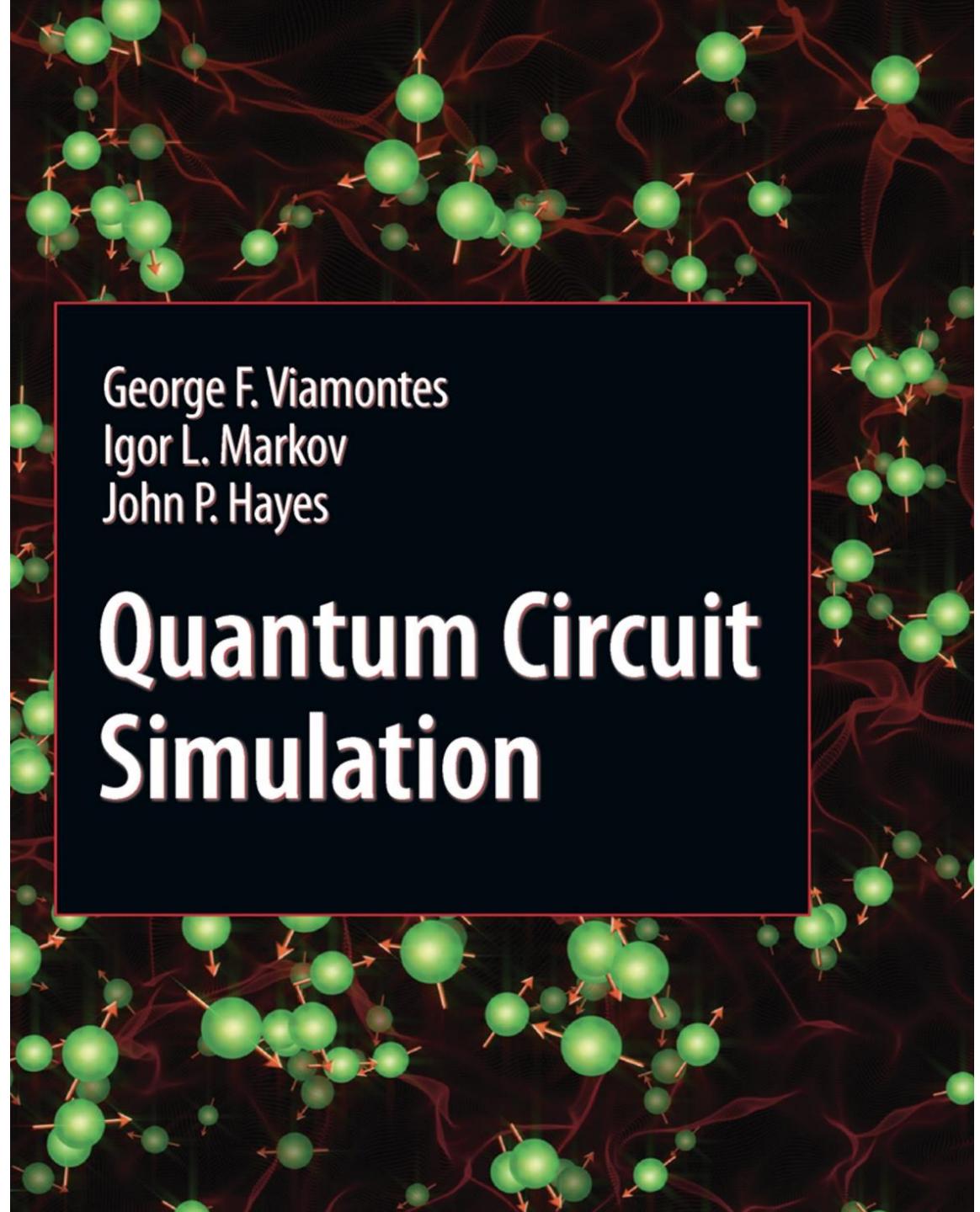
- Without the second optimization, we store explicit  $2^N \times 2^N$  matrices, and at  $N=13$  qubits, each matrix takes 0.5 gigabytes (assuming 32 bit precision)
- With the second optimization, by not storing explicit matrices, we can scale up to 20+ qubits

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

quantum-computing.ibm.com/composer/files/new

IBM Quantum Composer

Composer files

File Edit Inspect View Share Setup and run

Untitled circuit

Visualizations seed 5352

OpenQASM 2.0

No data available

Quantum gates and operations

Gates and operations manipulate qubits. They are the building blocks of quantum circuits. You will drag and drop operation blocks onto the wires below to construct your quantum circuit.

2 of 8 Back Next

Probability (%)

Computational basis states

Phase

St

The screenshot shows the IBM Quantum Composer web application. At the top, there's a navigation bar with back, forward, and search icons, followed by the URL 'quantum-computing.ibm.com/composer/files/new'. Below the URL is the 'IBM Quantum Composer' logo and a sidebar titled 'Composer files' which says '0 files' and has a 'New file +' button. The main workspace is titled 'Untitled circuit' and contains a grid of quantum gate icons: H, CNOT, Toffoli, Invert, Swap, T, S, Z, T†, S†, P, RZ, RXX, RZZ, and various rotation gates (RX, RY, U). A tooltip for 'Quantum gates and operations' explains that they manipulate qubits and are used to build circuits. At the bottom, there are plots for 'Probability (%)' and 'Computational basis states', along with a phase diagram and a visualization seed of 5352.

# IBM Quantum Lab / IBM Quantum Composer

The screenshot shows the IBM Quantum Composer web application interface. At the top, the URL is quantum-computing.ibm.com/composer/files/new. The main area is titled "Untitled circuit". On the left, there's a sidebar for "Composer files" showing 0 files and a "New file +" button. The central workspace displays a quantum circuit with three qubits (q\_0, q\_1, q\_2) and one classical register (c3). A gate palette at the top provides access to various quantum gates like H, RZ, T, S, Z, T†, S†, P, RXX, and RZZ. To the right of the circuit, the OpenQASM 2.0 code is visible:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg c[3];

```

A modal window titled "Let's build your first circuit" is open, providing instructions to construct a Bell state by dragging a Hadamard gate onto the first qubit wire. The modal also indicates "3 of 8" steps completed. At the bottom, there's a "Probability (%)" scale and a "Computational basis states" visualization.

# IBM Quantum Lab / IBM Quantum Composer

IBM Quantum Composer

quantum-computing.ibm.com/composer/files/796fac8fe8ea7fc9bc918d0cef33f941

File Edit Inspect View Share Setup and run

Untitled circuit Saved Visualizations seed 5352

RZ OpenQASM 2.0

RXX

What just happened?

Visualizations use a simulator to show what's happening on a quantum computer before you measure your circuit. They are updated each time you add an operation to a wire.

Currently, the visualizations show the effect of a Hadamard gate. The qubit is now in superposition.

Back Next

Probabilities

Computational basis states	Probability (%)
$ 000\rangle$	50
$ 001\rangle$	50
$ 010\rangle$	0
$ 011\rangle$	0
$ 100\rangle$	0
$ 101\rangle$	0
$ 110\rangle$	0
$ 111\rangle$	0

Q-sphere

Phase 0

St

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg c[3];
h q[0];
```

# IBM Quantum Lab / IBM Quantum Composer

Untitled circuit Saved

Visualizations seed 5352

OpenQASM 2.0

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg c[3];
h q[0];
cx q[0],q[1];
```

Probabilities

Entangle two qubits

Add the CNOT gate ( ) to the right of the H gate ( ) on wire q0. This operation acts on two qubits: q0 and q1.

Notice that the visualizations update when you add a new gate. The qubit is now entangled.

5 of 8

Computational basis states

3π/2

St

RZ

RXX

What just happened?

Visualizations use a simulator to show what's happening on a quantum computer before you measure your circuit. They are updated each time you add an operation to a wire.

Currently, the visualizations show the effect of a Hadamard gate. The qubit is now in superposition.

4 of 8

Back Next

Probabilities (%)

Computational basis states

Phase 0 π/2 π 3π/2

Q-sphere

# IBM Quantum Lab / IBM Quantum Composer

Untitled circuit Saved

Visualizations seed 5352

OpenQASM 2.0 ▾

Open in Quantum Lab

```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3 
4 qreg q[3];
5 creg c[3];
6 
7 h q[0];
8 cx q[0],q[1];
9 measure q[0] -> c[0];
10 measure q[1] -> c[1];
```

Code editor

The code editor updates as you add operations to your circuit. If you update the openQASM code, changes will be reflected in the graphical editor.

You can see the code in openQASM or read-only Qiskit (Python). You can export the code for use in other applications.

Probability (%)

7 of 8

Back Next

Probability (%)

0 20 40 60 80

000 001 010 011 100 101 110 111

Computational basis states

Phase

$\pi/2$  0  $3\pi/2$

$|011\rangle$

St

# IBM Quantum Lab / IBM Quantum Composer

The screenshot shows the IBM Quantum Composer interface. At the top, there's a navigation bar with icons for Home, Help, and Logout, followed by the title "IBM Quantum Composer". Below the title is a dark sidebar containing links to "Composer docs & tutorials", "Getting started" (with "Create your first circuit walkthrough" and "Explore the latest updates"), "Quantum Composer user guide", "Learn quantum computing: a field guide", "Try out some circuit examples", "IBM Quantum System services", "IBM Quantum simulators", and "Glossaries". The main workspace is titled "Untitled circuit" and shows a grid of quantum gate icons. A modal window titled "Keep learning" is open in the center, containing text about finding resources and instructions for running circuits. At the bottom right of the modal are "Back" and "Finish" buttons.

IBM Quantum Composer

File Edit Inspect View Share

Untitled circuit Saved

H  $\oplus$   $\oplus$   $\oplus$   $\oplus$  I T S Z  $T^\dagger$   $S^\dagger$  P  $i$  :

RZ  $|0\rangle$   $|0\rangle$   $\text{if}$   $\sqrt{X}$   $\sqrt{X}^\dagger$  Y RX RY U

RXX RZZ + Add

Keep learning

Not sure what to do next? You will find all the information you need, including instructions to run a circuit, background information about quantum computing, and more, on our Resources panel.

8 of 8

Back Finish

# IBM Quantum Lab / IBM Quantum Composer

IBM Quantum Composer

File Edit Inspect View Share Setup and run

Untitled circuit Saved Visualizations seed 5352

OpenQASM 2.0

Open in Quantum Lab

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg c[3];
h q[0];
cx q[0],q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
```

Probabilities

Q-sphere

St

The screenshot shows the IBM Quantum Composer web application. On the left, there's a sidebar with links like 'Getting started', 'Quantum Composer user guide', and 'IBM Quantum System services'. The main area displays a quantum circuit with four qubits and two classical bits. The circuit starts with a Hadamard gate on qubit 0, followed by a CNOT gate between qubits 0 and 1, and another CNOT gate between qubits 1 and 2. The circuit ends with measurement operations on all qubits. Below the circuit, there's a bar chart showing probabilities for different states and a 3D Q-sphere visualization.

# IBM Quantum Lab / IBM Quantum Composer

Untitled circuit Saved

Visualizations seed 5352

Qiskit Read only

Open in Quantum Lab

```
1 from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
2 from numpy import pi
3
4 qreg_q = QuantumRegister(3, 'q')
5 creg_c = ClassicalRegister(3, 'c')
6 circuit = QuantumCircuit(qreg_q,
7     creg_c)
8
9 circuit.h(qreg_q[0])
10 circuit.cx(qreg_q[0], qreg_q[1])
11 circuit.measure(qreg_q[0], creg_c[0])
12 circuit.measure(qreg_q[1], creg_c[1])
```

Probabilities

Q-sphere

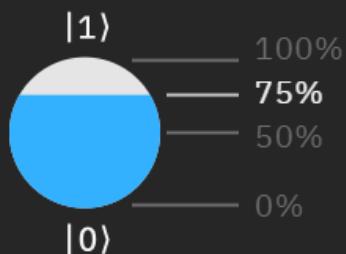
Probability (%)

State	Probability (%)
000	0
001	0
010	0
011	100
100	0
101	0
110	0
111	0

# Phase disk

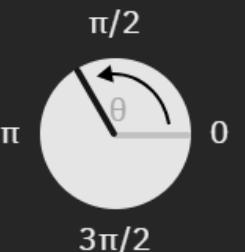
## Probability the qubit is in the $|1\rangle$ state

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).



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  $\text{Tr}[\text{Tr}_{i \neq j}[|\psi\rangle\langle\psi|]^2]$

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
$ 0\rangle\langle 0 $ $ 1\rangle\langle 1 $	Density Bloch	$Z$ Swap	$Y$	$S$ $S^{-1}$	$T$ $T^{-1}$	$Z^t$ $Z^{-t}$	$Z^{f(t)}$ $Rz(f(t))$	$Z^{A/2^n}$ $Z^{-A/2^n}$	$[Z]_{par}$
$\textcirclearrowleft$ $\textcirclearrowright$	Chance Amps	$Y$	$Y^{\frac{1}{2}}$ $Y^{-\frac{1}{2}}$	$X^{\frac{1}{2}}$ $X^{-\frac{1}{2}}$	$Y^{\frac{1}{4}}$ $Y^{-\frac{1}{4}}$	$Y^t$ $Y^{-t}$	$Y^{f(t)}$ $Ry(f(t))$	$Y^{A/2^n}$ $Y^{-A/2^n}$	$[Y]_{par}$
$\textcirclearrowleft \textcirclearrowright$		$H$	$X^{\frac{1}{4}}$ $X^{-\frac{1}{4}}$	$X^t$ $X^{-t}$	$X^t$ $X^{-t}$	$X^{f(t)}$ $Rx(f(t))$	$X^{A/2^n}$ $X^{-A/2^n}$	$X^{\oplus  0\rangle}$	$[X]_{par}$

use controls

drag gates onto circuit

outputs change

Local wire states (Chance/Bloch)

Final amplitudes

**Toolbox<sub>2</sub>**

$\ominus$ $\oplus$	$+[t]$ $-[t]$	QFT $QFT^\dagger$	input A input B input R	+1 -1 +A -A +AB -AB $\times A$ $\times A^{-1}$	$\oplus A < B$ $\oplus A > B$ $\oplus A \leq B$ $\oplus A \geq B$ $\oplus A = B$ $\oplus A \neq B$	+1 mod R -1 mod R +A mod R -A mod R $\times A$ mod R $\times A^{-1}$ mod R $\times B$ mod R $\times B^{-1}$ mod R	... 0 - i -i $\sqrt{i}$ $\sqrt{-i}$	X/Y Probes Order Frequency Inputs Arithmetic Compare Modular Scalar Custom Gates
$\emptyset$ $\otimes$			A=# default B=# default R=# default					
$ +\rangle\langle + $ $ -\rangle\langle - $	Reverse	$\text{Grad}^{\frac{1}{2}}$ $\text{Grad}^{-\frac{1}{2}}$						
$ i\rangle\langle i $ $ i\rangle\langle -i $		$\text{Grad}^t$ $\text{Grad}^{-t}$						

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:HULINSTREPOS:37368560>  
<https://dash.harvard.edu/handle/1/37368560>

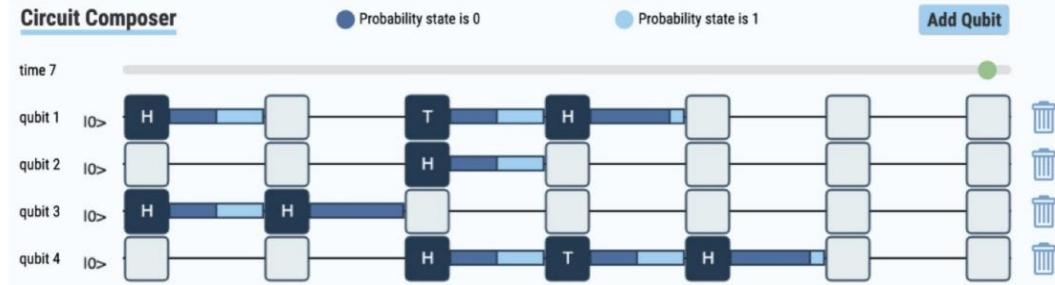


Figure 3.3: The Circuit Composer center with a four qubit circuit, showing how the stacked bar chart changes over time.

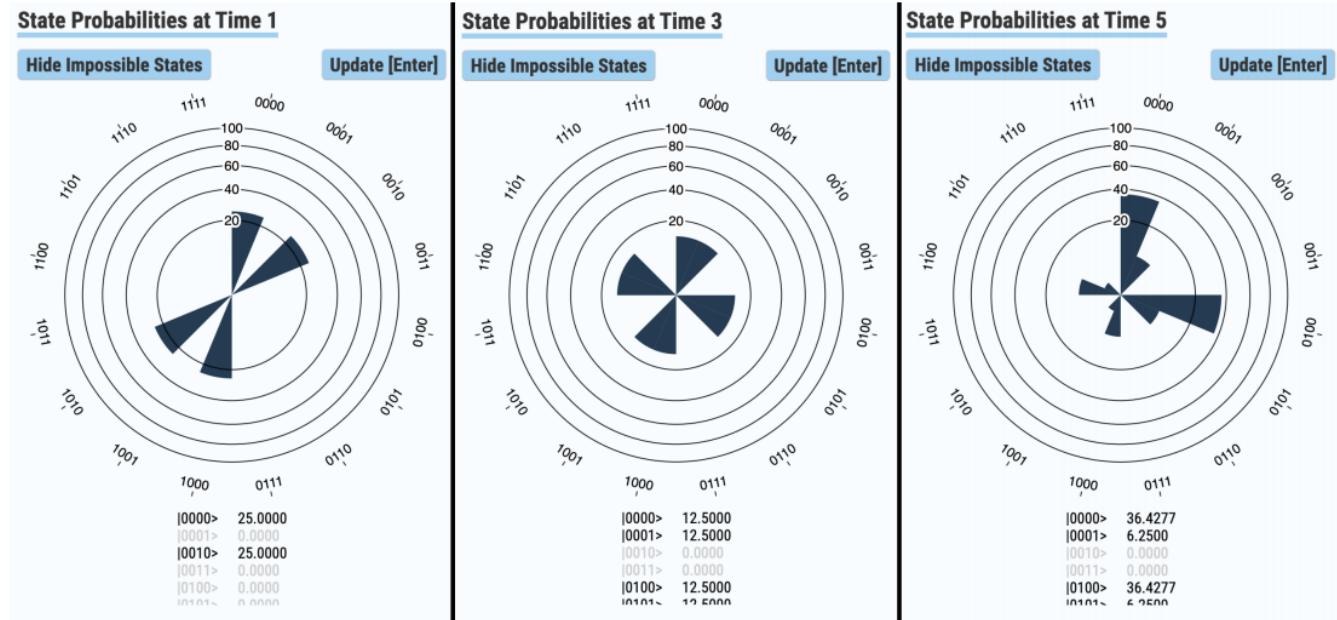
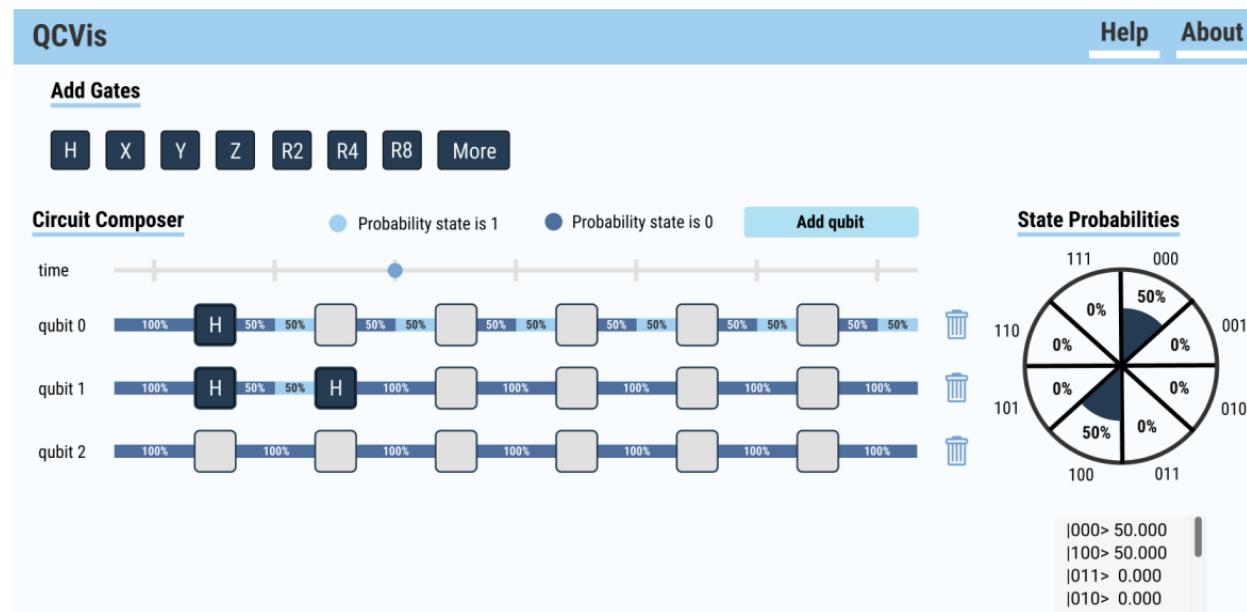


Figure 3.6: State Probability panel changes over time depending on the time slider.



% 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.

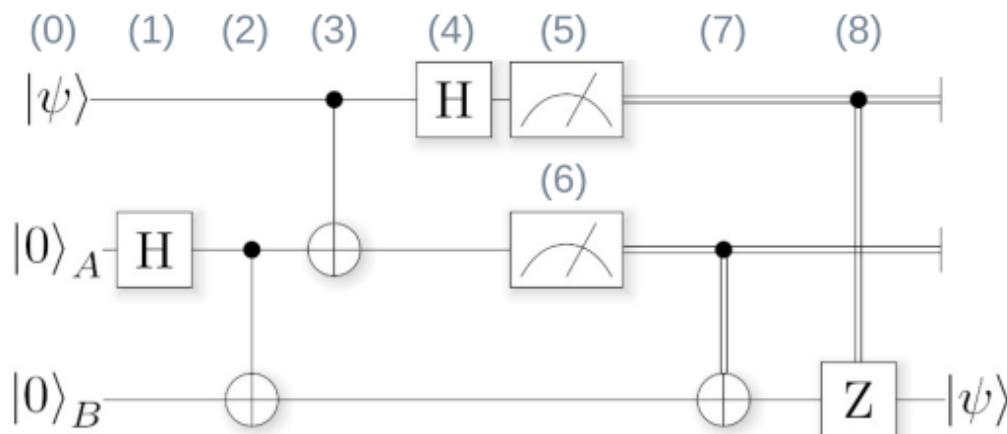


Figure 1: Block diagram of a quantum circuit performing quantum teleportation. Each horizontal line represent a qbit.

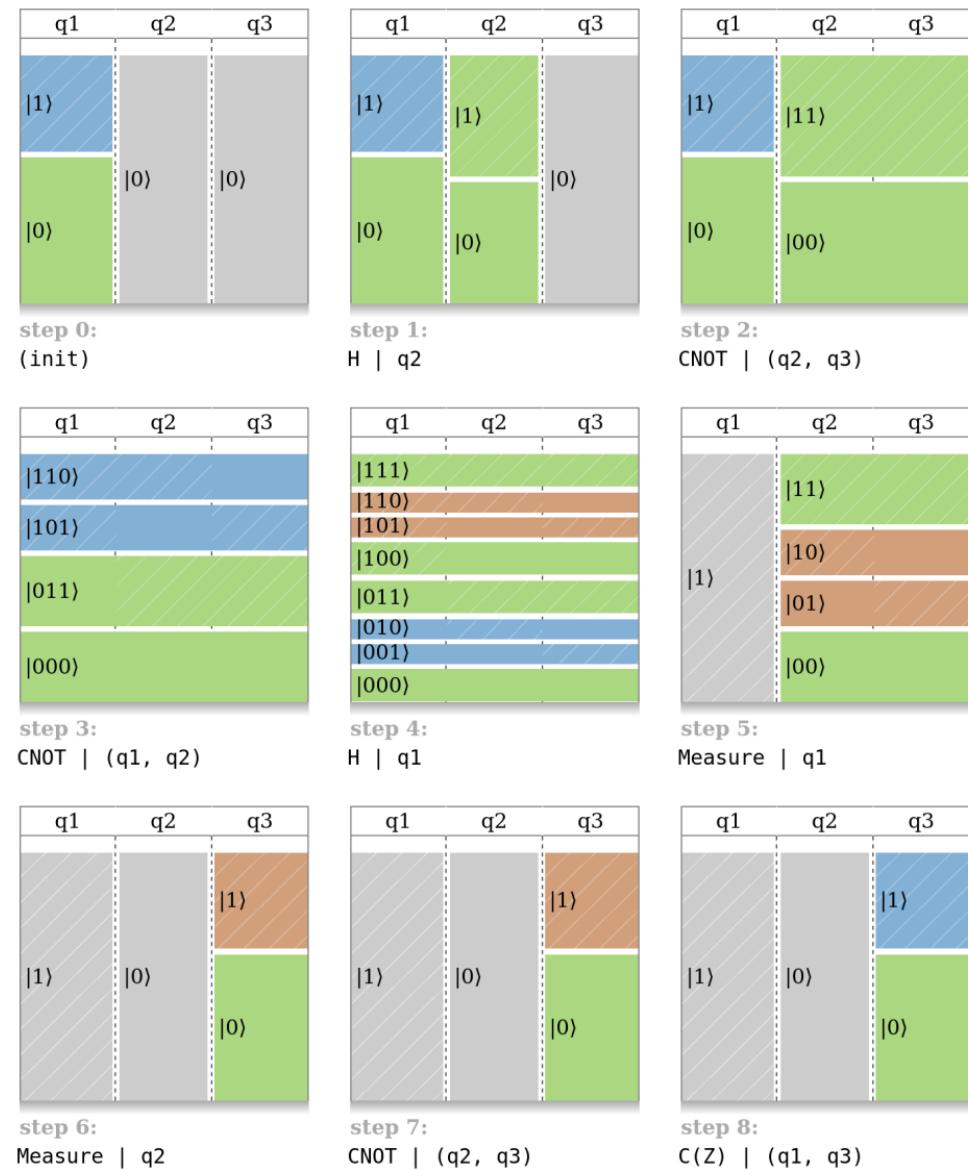


Figure 6: Visualization of the state of the system during the execution of quantum teleportation.

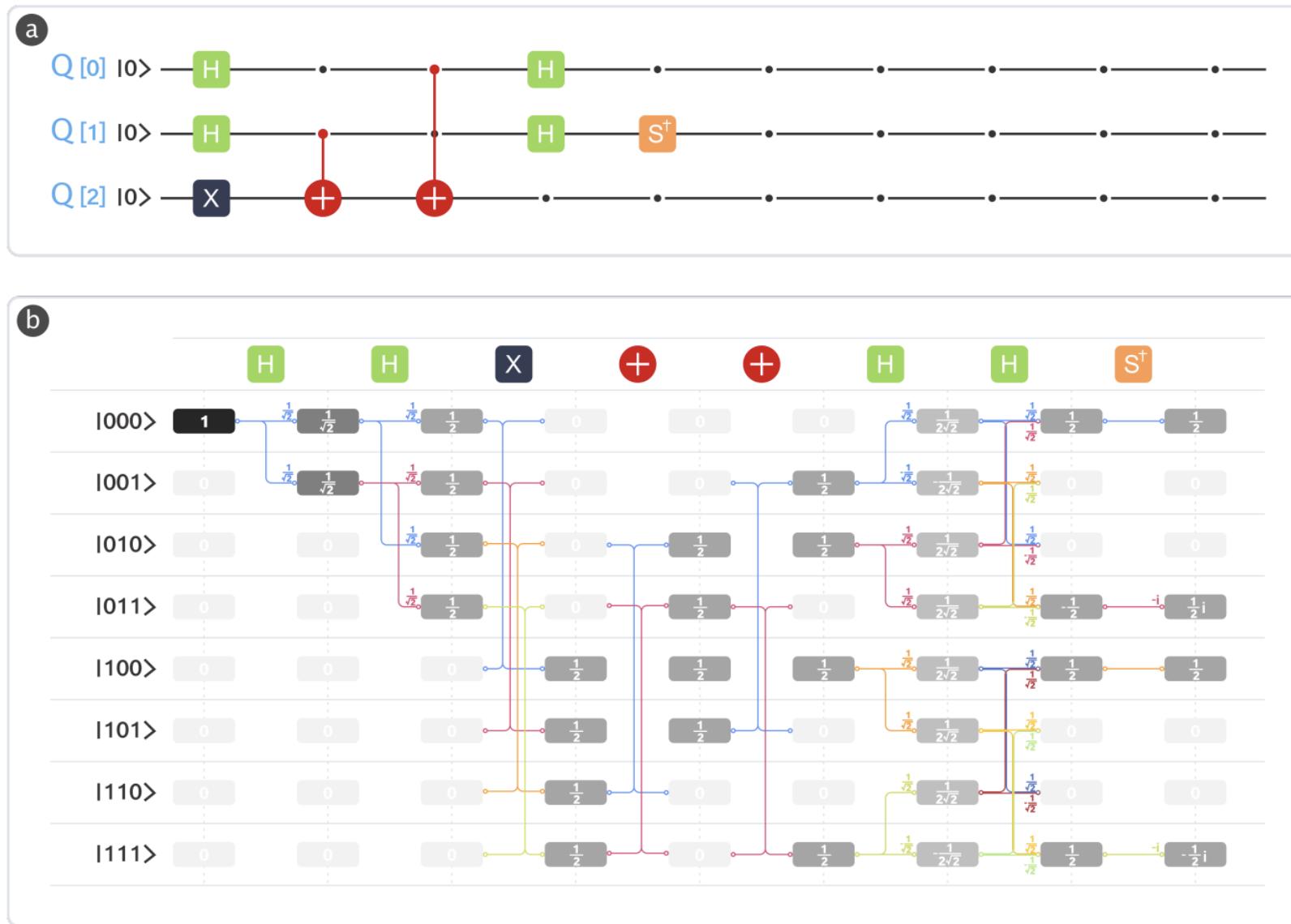
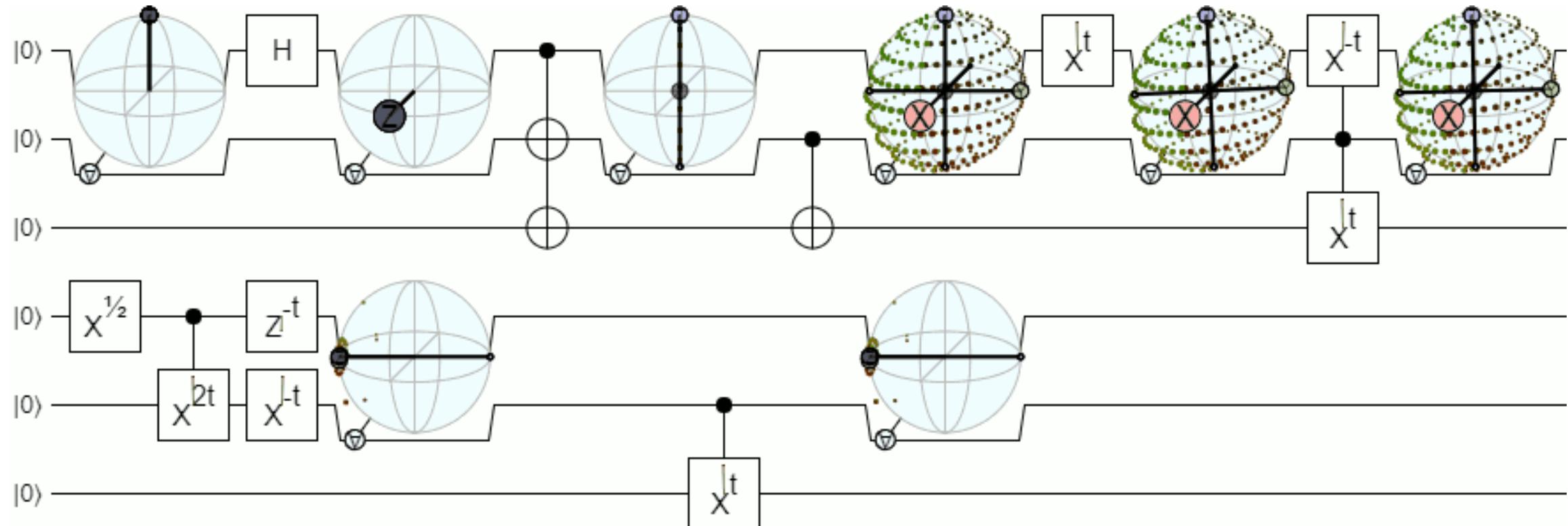


Figure 1: A parameter flow visualization of a three-qubit circuit's running process by QuFlow: a) a three-qubit circuit with 8 quantum gates including single-qubit gates (five H gates, one X gate and one  $S^\dagger$  gate) and a two-qubit gate (CNOT gate); b) the visualization result of circuit in a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

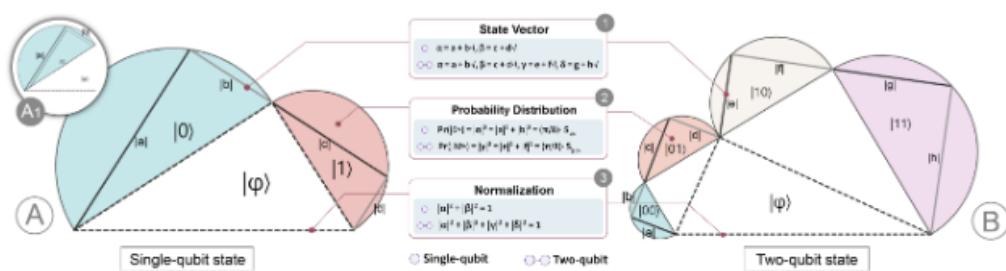
Craig Gidney, 2017

<https://algassert.com/post/1716>

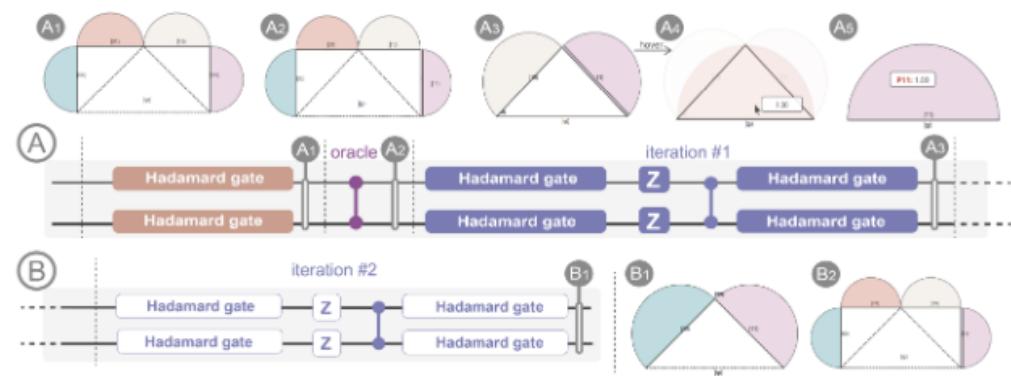


# Venus: A geometrical representation for quantum state visualization

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>

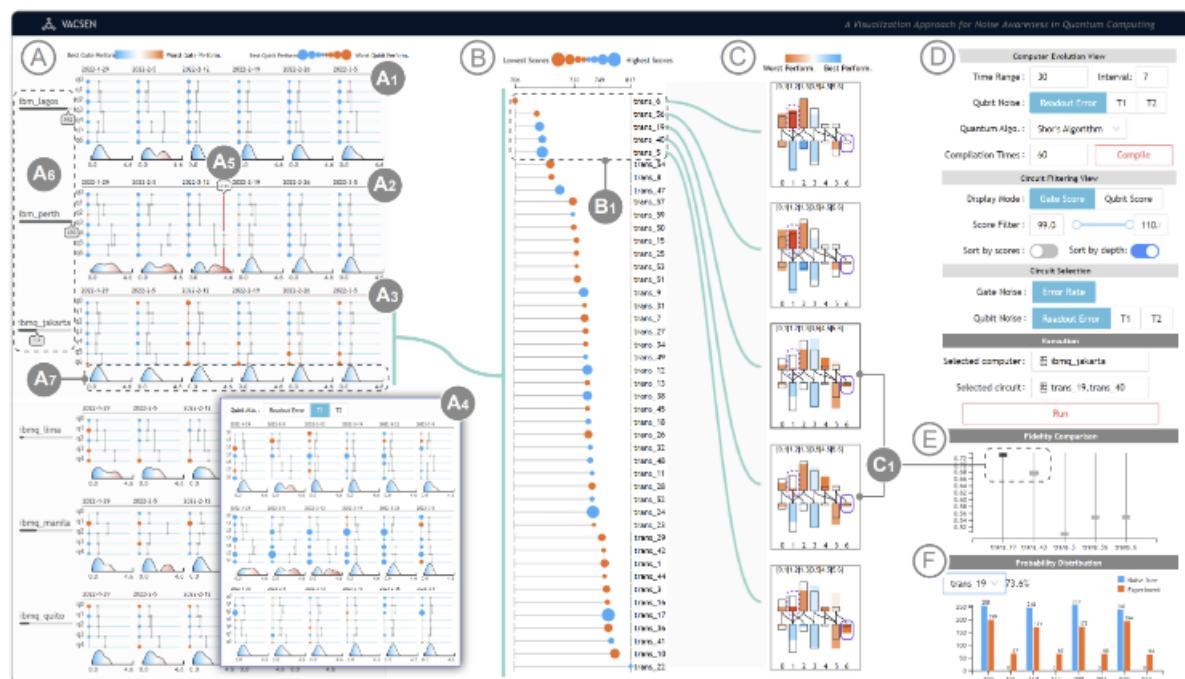


Fig. 1: The interface of VACSEN makes users aware of the quantum noise via three linked views (A-C). Computer Evolution View (A) allows the assessment for all quantum computers based on a temporal analysis for multiple performance metrics. Circuit Filtering View (B) supports the filtering for the potential 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.

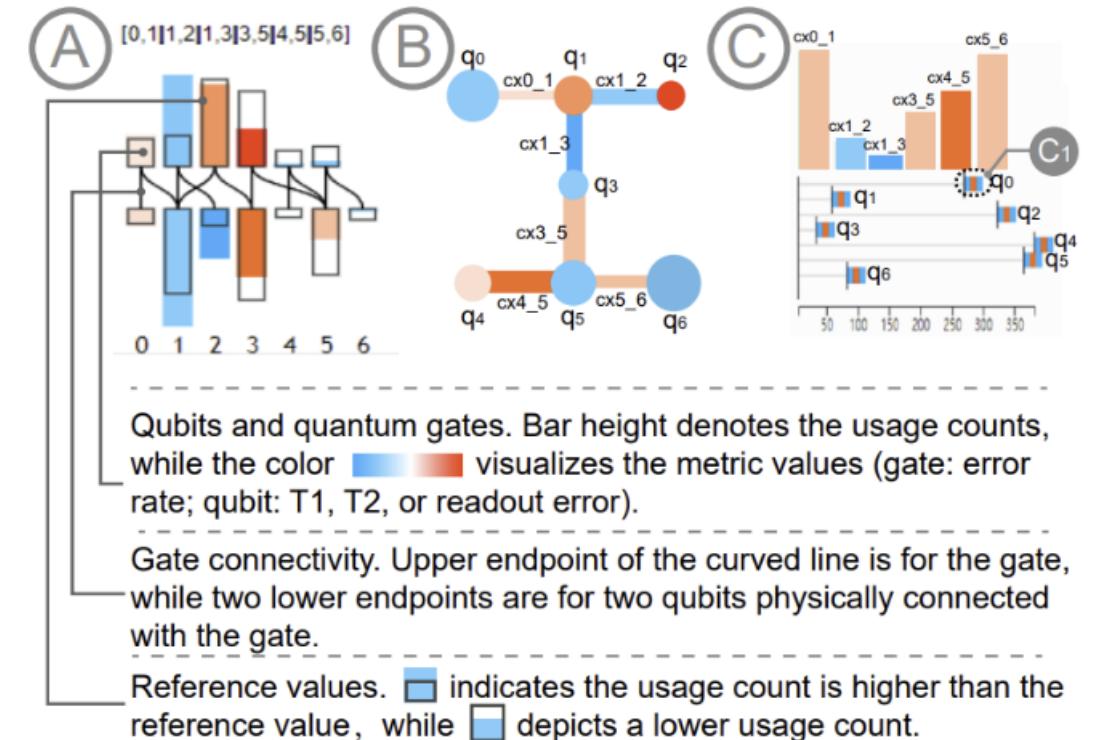


Fig. 5: Circuit Comparison View. (A) The Circuit Comparison View supports the in-depth comparison of multiple compiled circuits. (B)(C) Design alternatives of Circuit Comparison View. Both alternative designs cannot support an effective comparison of the usage times of qubits and quantum gates.

# 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

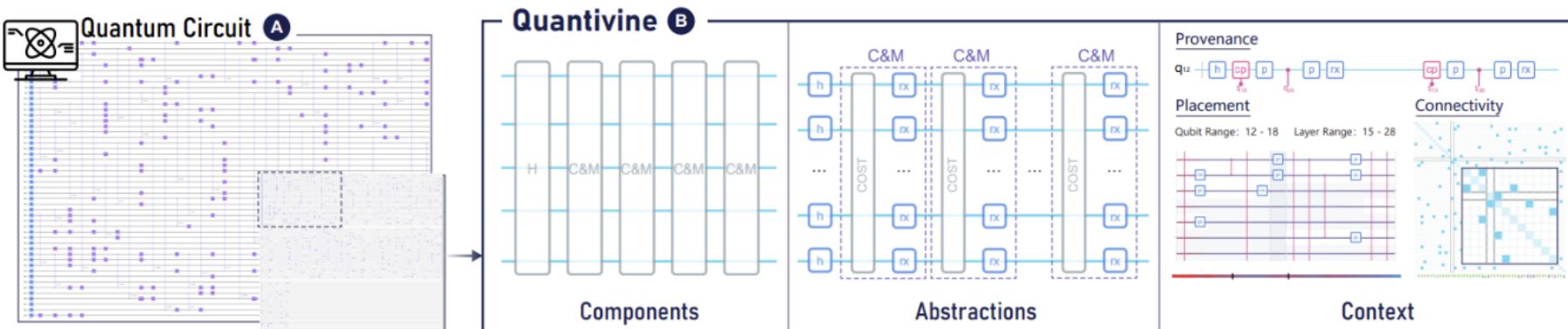


Fig. 1: Quantum circuit visualizations for a 50-qubit quantum circuit. (A) Typical quantum circuit diagram generated by Qiskit [67]. (B) The resulting visualizations of *Quantivine*, including structured components, pattern abstractions, and context enhancement.

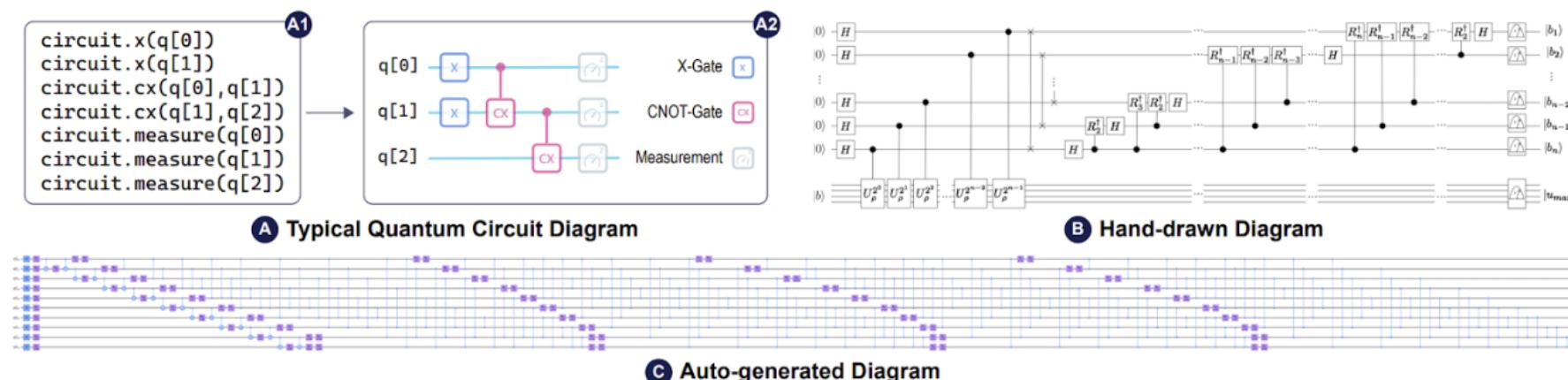


Fig. 2: The illustration of quantum circuit diagrams. (A) A typical quantum programming code (A1) and its corresponding quantum circuit diagram (A2). (B) A hand-drawn quantum circuit diagram depicts the implementation of quantum PCA algorithm [49]. (C) An auto-generated diagram (using Qiskit library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