quAPL: A Quantum Computing Library in APL

Marcos Frenkel, Santiago Núñez-Corrales, Bruno Abreu

NCSA/IQUIST, University of Illinois Urbana-Champaign

Dyalog '23 17/10/2023

National Center for Supercomputing Applications
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

IQUIST

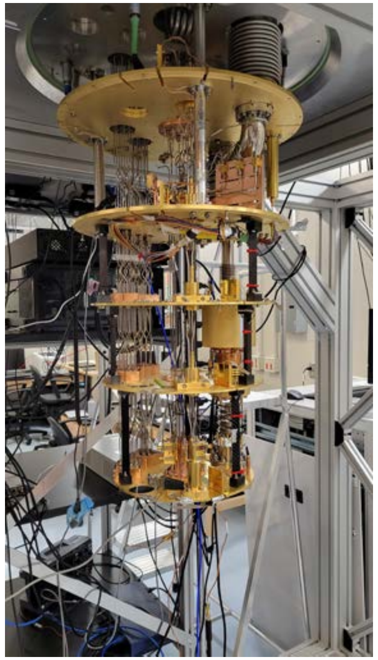
NSF #2016136

About me

- I have been working at an experimental quantum computing lab, called Pfafflab, for the past 3 years
- Helped developed all the software around measurements, including experiment control, instrument communication, data analysis, etc.
- Joined Santiago’s work on quAPL early this year

Classical and quantum computers have different sets of resources

Space, Time

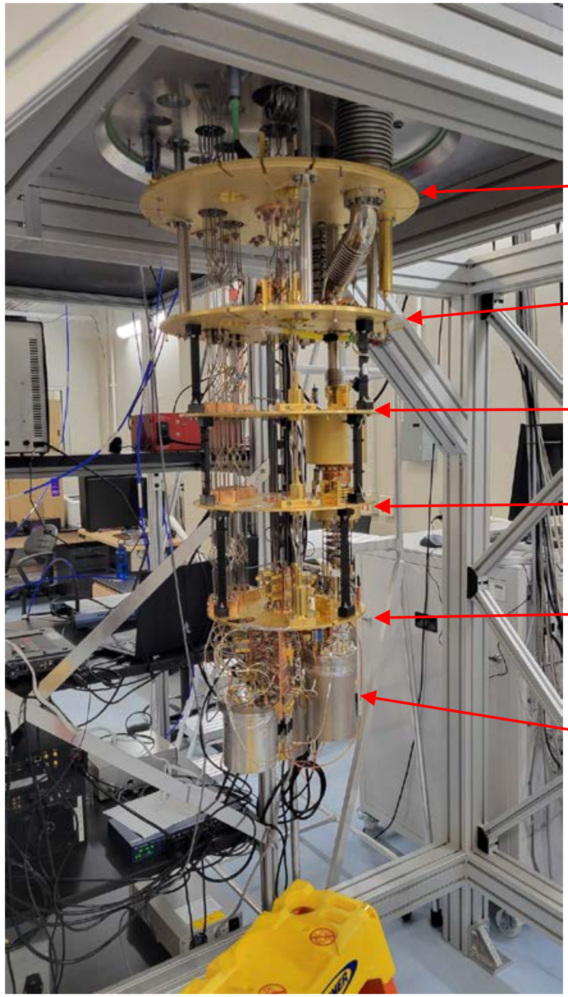


Space, Time
Superposition, Coherence, Entanglement, Interference

Credit: dilution refrigerator

Chitambar, E. and Gour, G., 2019. Quantum resource theories. *Reviews of modern physics*, *91*(2), p.025001.

The “quantum computer” is really a big cold fridge



60 K

4 K

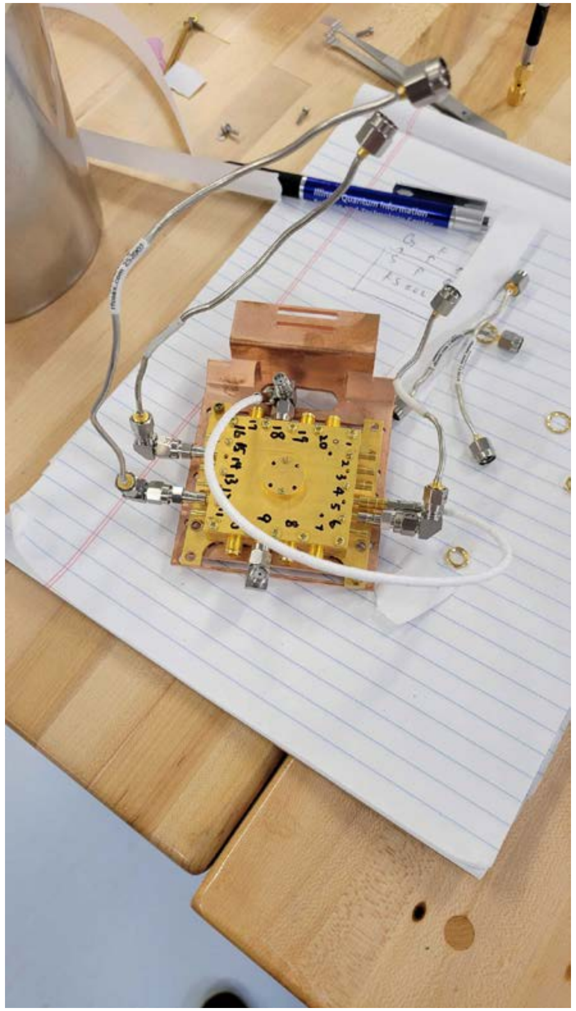
0.7 K

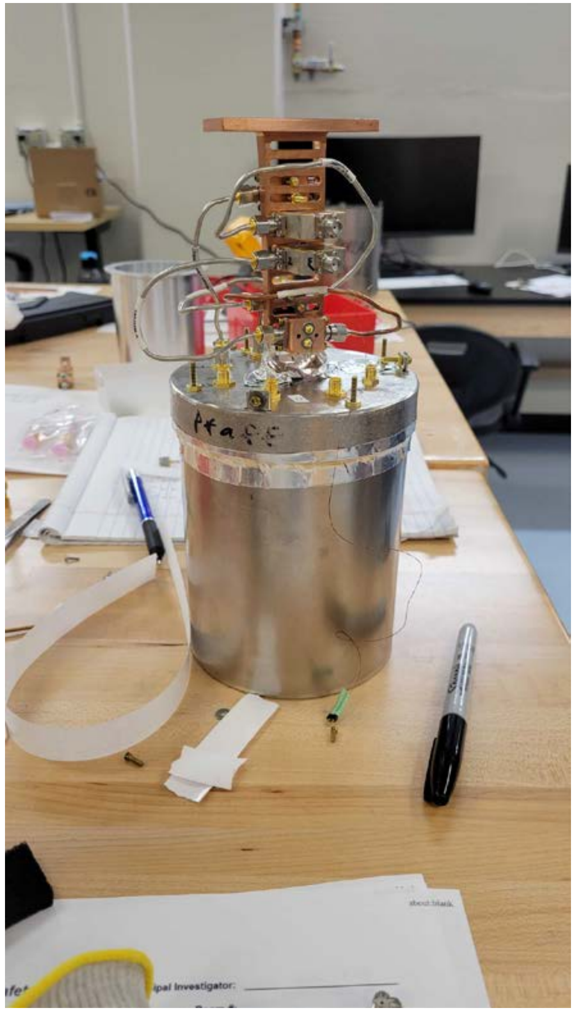
100 mK

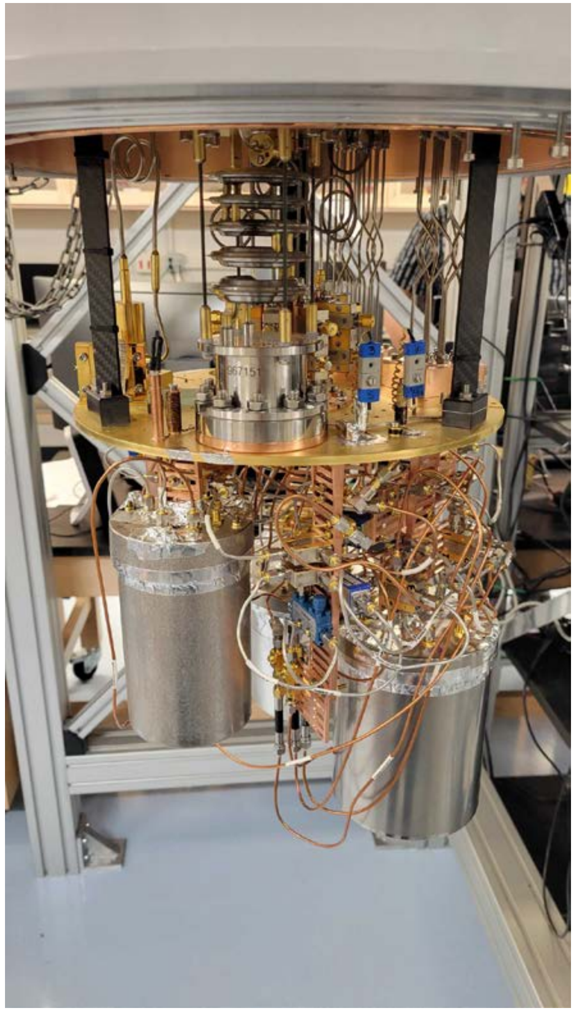
10 mK

Samples in magnetic shields

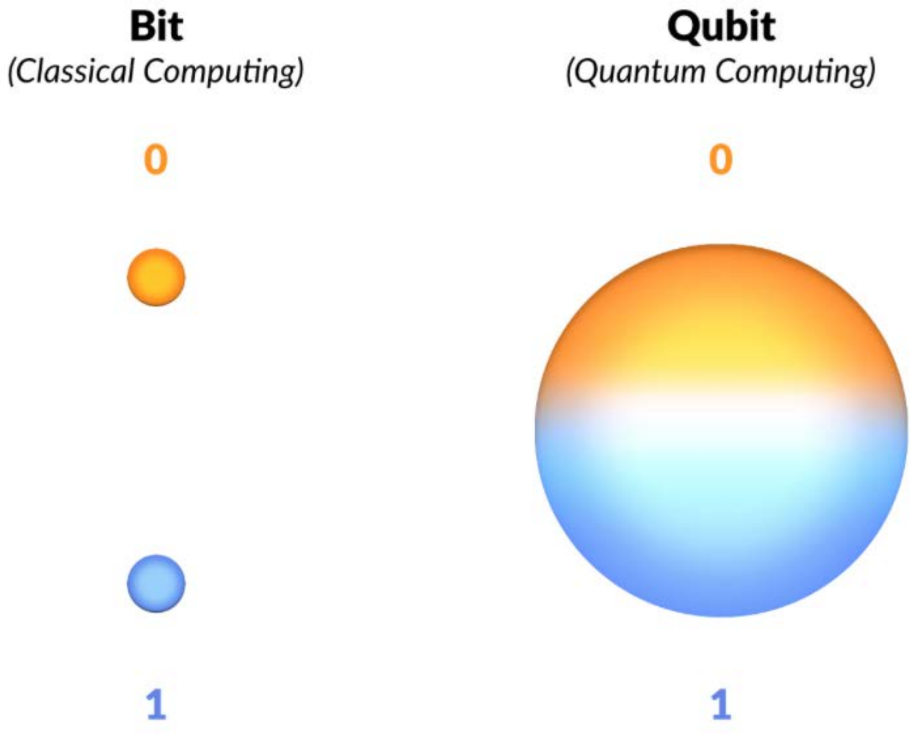
The samples can get really complicated

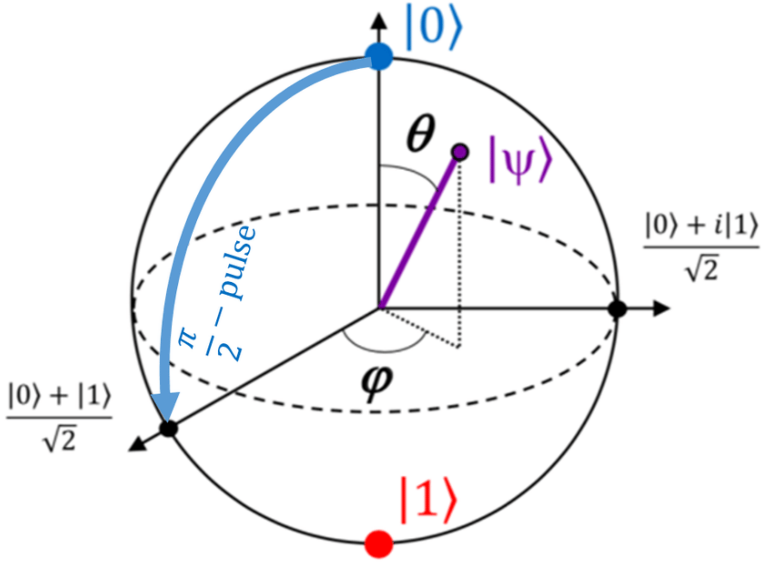






Classical bits vs quantum bits (qubits)





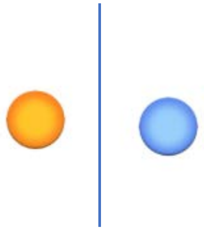
$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$$

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

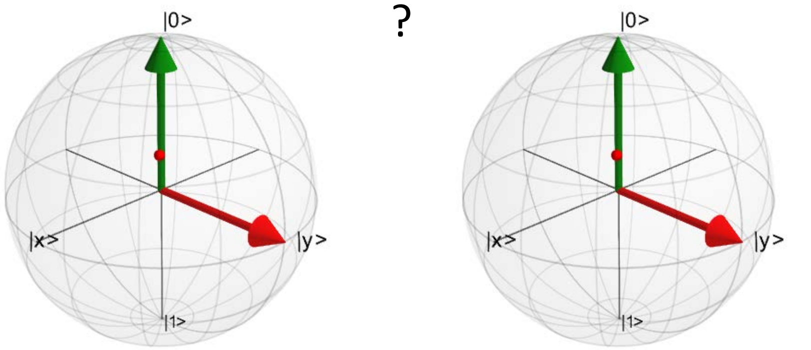
$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + e^{i\varphi}sin\frac{\theta}{2}|1\rangle$$

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

Composite finite states



01

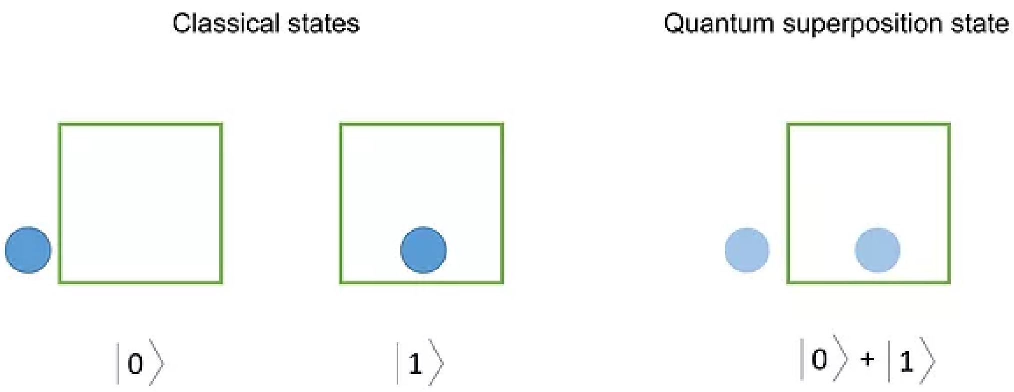


Consequence: an arbitrary state of *n* qubits, a *quantum register*, requires (at least) 2^n classical bits to describe

Why? A consequence of tensor products defined over vector spaces on complex fields.

$$\begin{bmatrix}\alpha\\ \beta\end{bmatrix} \otimes \begin{bmatrix}\gamma\\ \delta\end{bmatrix} = \begin{bmatrix}\alpha\begin{bmatrix}\gamma\\ \delta\end{bmatrix}\\ \beta\begin{bmatrix}\gamma\\ \delta\end{bmatrix}\end{bmatrix} = \begin{bmatrix}\alpha\gamma\\ \alpha\delta\\ \beta\gamma\\ \beta\delta\end{bmatrix}$$

Quantum superposition

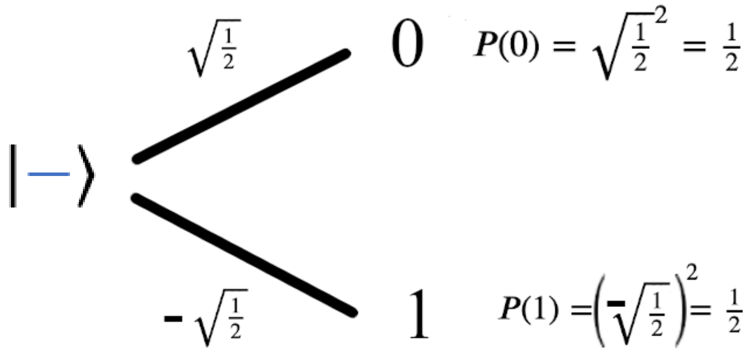


$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$$

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

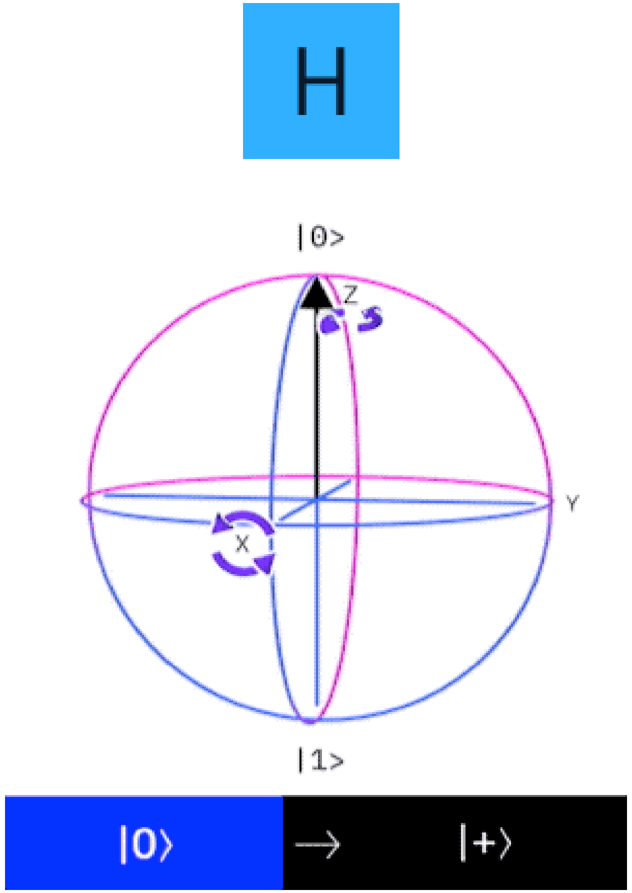
$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

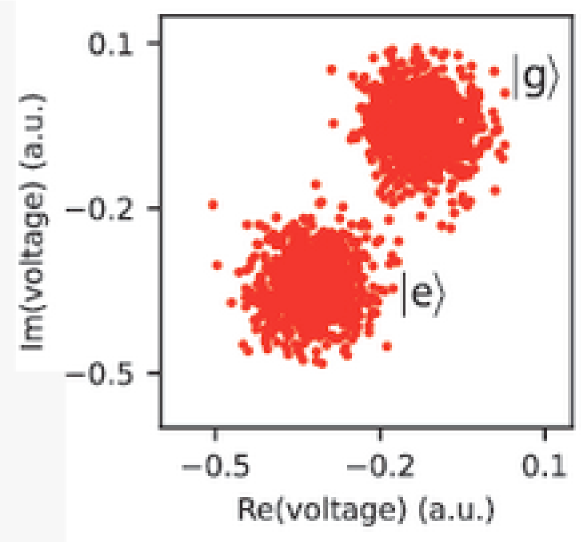
$$|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

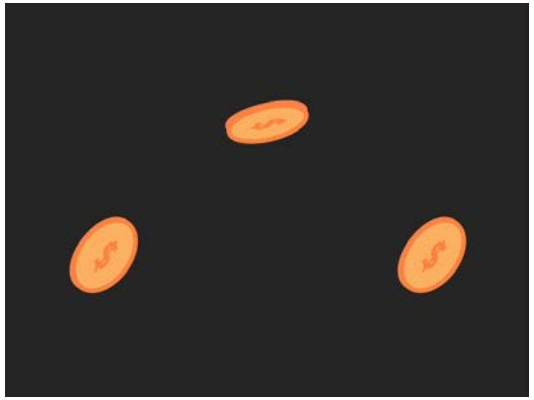


But: we can only read one result, probabilistically!

State probability determines measurement outcomes

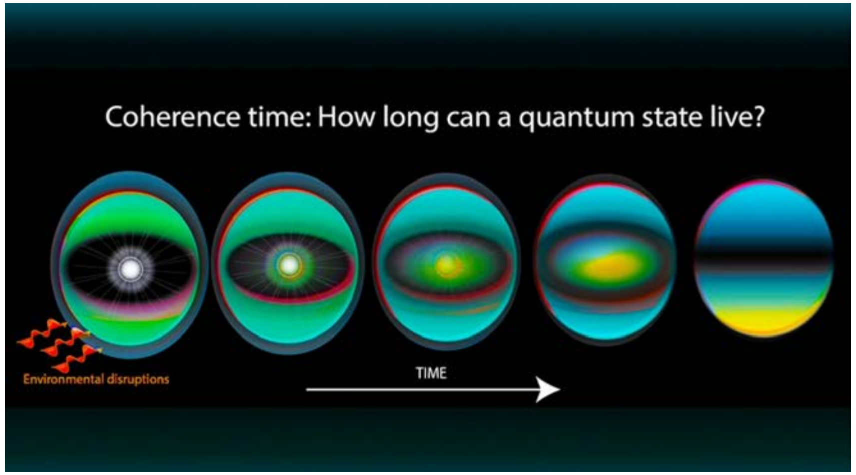






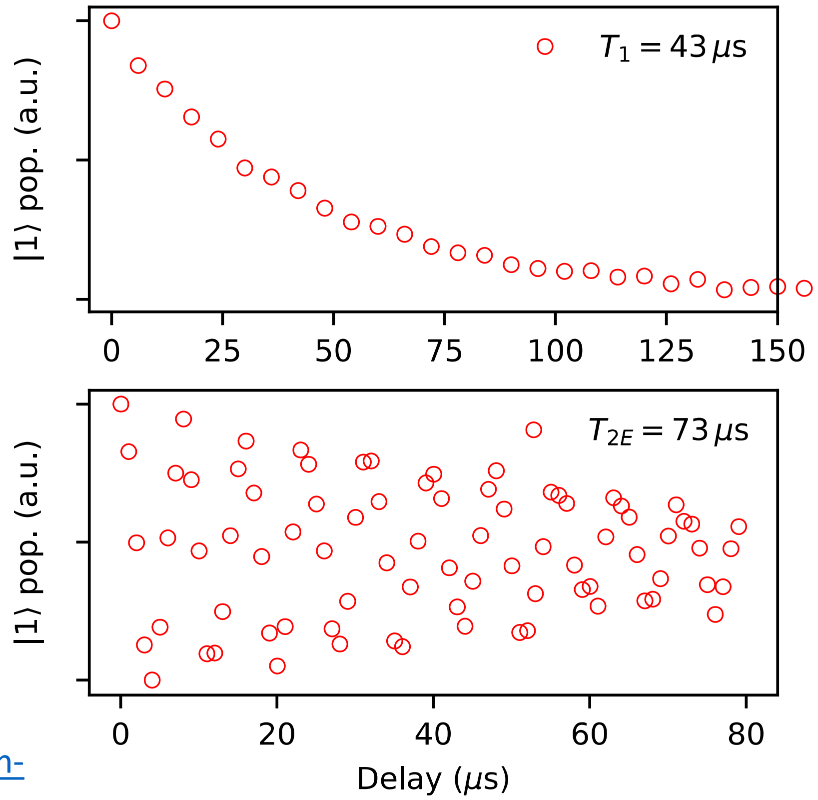
Credit:
Pfafflab

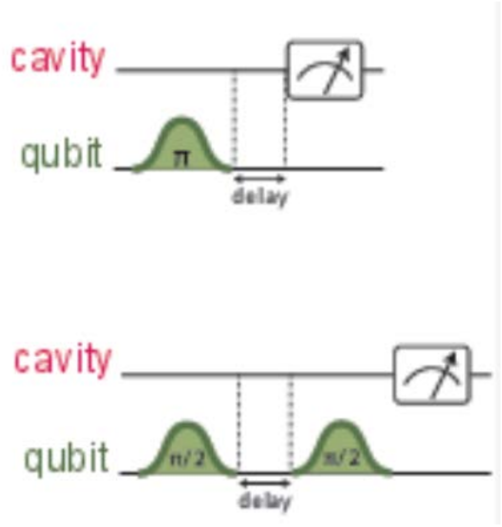
Quantum states decohere



https://www.nist.gov/topics/physics/introduction-new-quantum-revolution/strange-world-quantum-physics

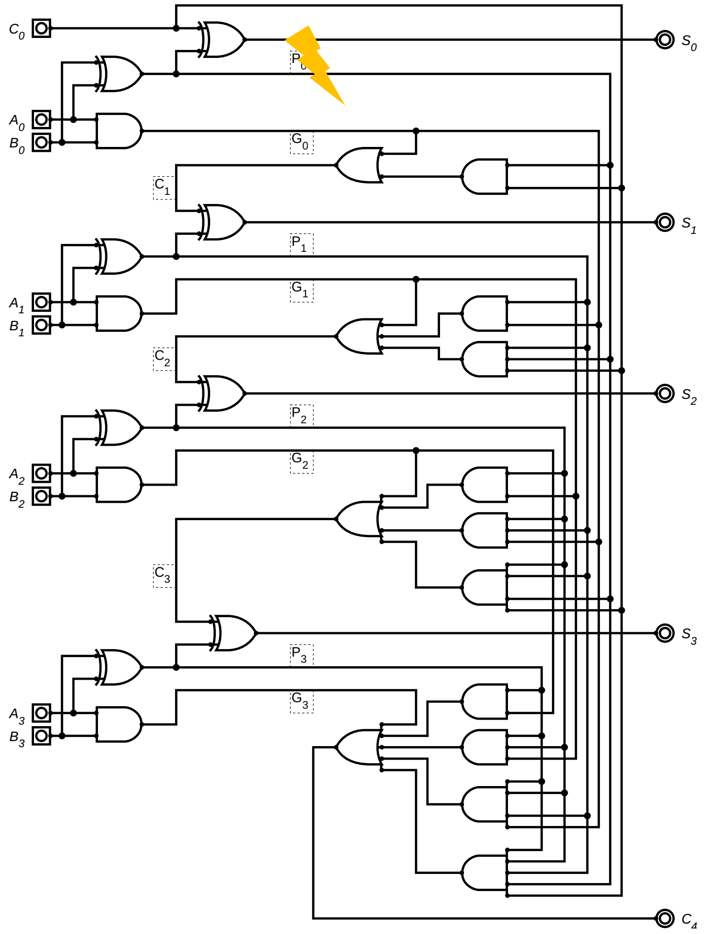
Real decoherence

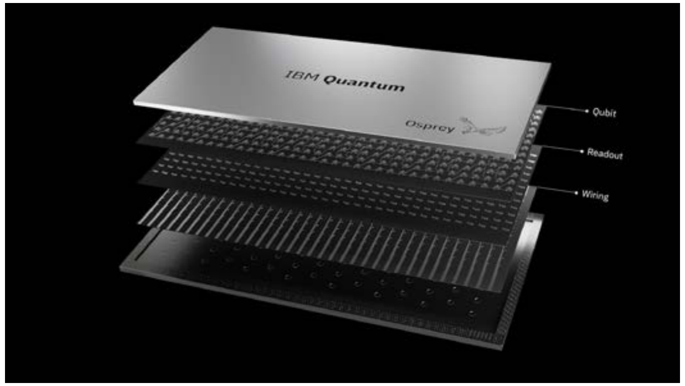


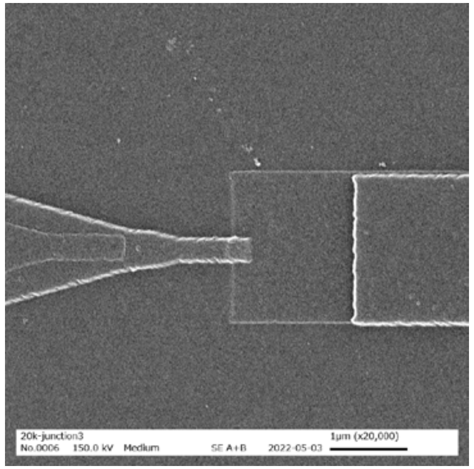


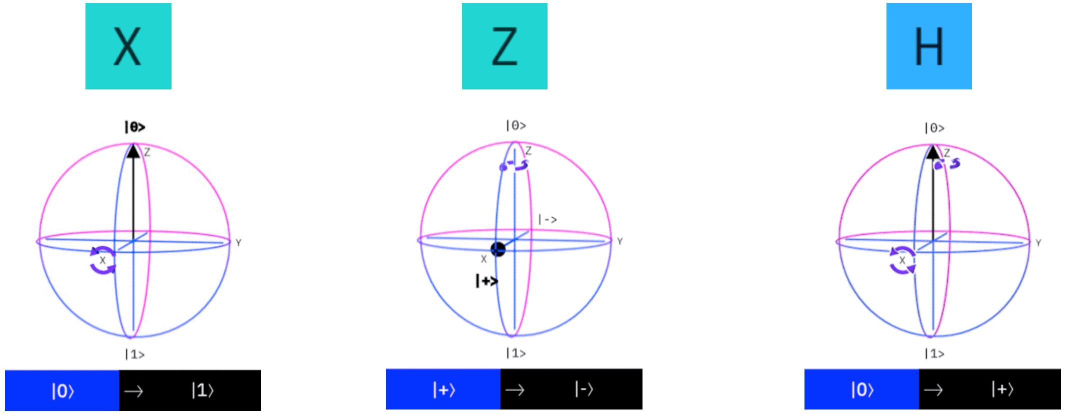
Credit: Pfafflab

Bits travel through circuits, circuits travel through qubits



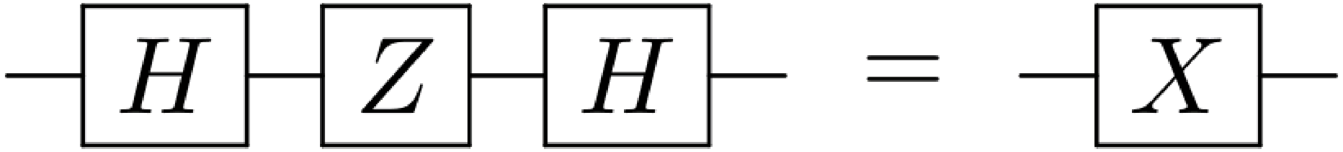


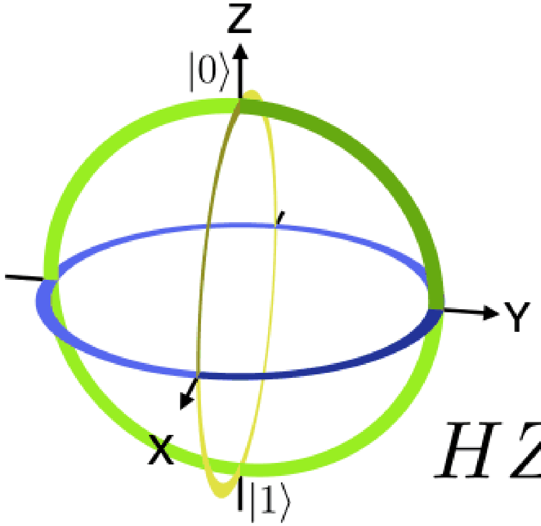




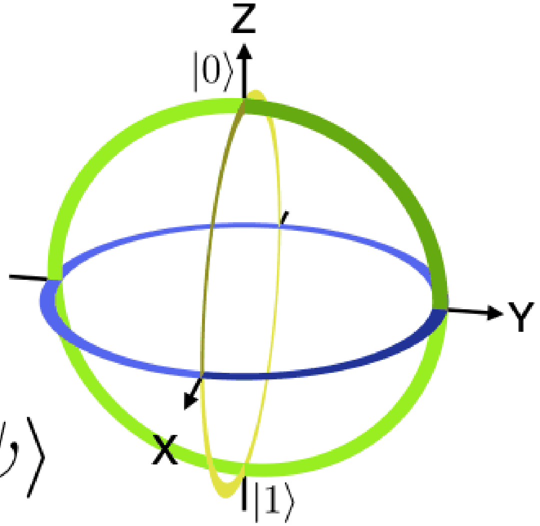
https://lewisla.gitbook.io/learning-quantum/quantum-circuits/single-qubit-gates

Quantum circuits contain equivalences





$$HZH\,|\psi\rangle = X\,|\psi\rangle$$



https://github.com/cduck/bloch_sphere

How do we program quantum computers now?

```
 * Repeat-until-success circuit for Rz(theta),
 * cos(theta-pi)=3/5, from Nielsen and Chuang, Chapter 4.
 */
OPENQASM 3;
include "stdgates.inc";

/*
 * Applies identity if out is 01, 10, or 11 and a Z-rotation by
 * theta + pi where cos(theta)=3/5 if out is 00.
 * The 00 outcome occurs with probability 5/8.
 */
def segment qubit[2]:anc, qubit:psi -> bit[2] {
  bit[2] b;
  reset anc;
  h anc;
  ccx anc[0], anc[1], psi;
  s psi;
  ccx anc[0], anc[1], psi;
  z psi;
  h anc;
  measure anc -> b;
  return b;
}

qubit input;
qubit ancilla[2];
bit flags[2] = "11";
bit output;

reset input;
h input;
```

```
!pip install cirq
import cirq

def main():
    # Pick a qubit.
    qubit = cirq.GridQubit(0, 0)
    qubit2 = cirq.GridQubit(1, 0)

    # Create a circuit
    circuit = cirq.Circuit(
        cirq.H(qubit),
        cirq.CNOT(control = qubit, target = qubit2),

        cirq.measure(qubit, key='m'),
        cirq.measure(qubit2, key='n')
    )
    print("Circuit:")
    print(circuit)

    # Simulate the circuit several times.
    simulator = cirq.Simulator()
    result = simulator.run(circuit, repetitions=20)
    print("Results:")
    print(result)

if __name__ == '__main__':
    main()
```

Cirq

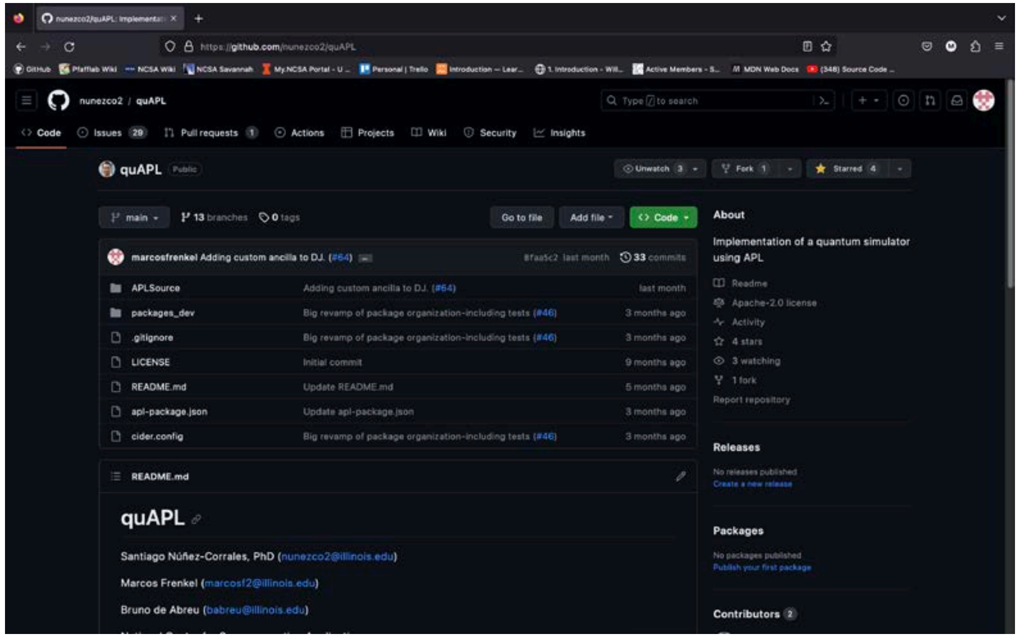
```
def solve[n:!ℕ](bits:!𝔹^n){
  // prepare superposition between 0 and 1
  x:=H(0:𝔹);
  // prepare superposition between bits and 0
  qs := if x then bits else (0:int[n]) as 𝔹^n;
  // uncompute x
  forget(x=qs[0]); // valid because `bits[0]==1`
  return qs;
}

// EXAMPLE CALL

def main(){
  // example usage for bits=1, n=2
  x := 1:!int[2];
  y := x as !𝔹^2;
  return solve(y);
}
```

SILQ

Enter quAPL

quAPL: A quantum computing library in APL

Github: https://github.com/nunezco2/quAPL



Coming to Tatin soon!

The examples shown had their namespace abbreviated to make the code more legible

Creating and manipulating qubits

```
a ← q0
b ← q1
```

Creating and manipulating qubits

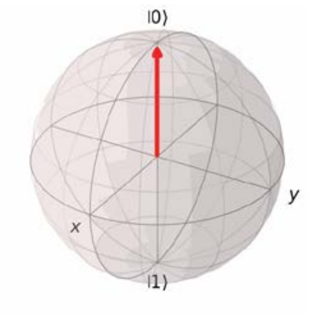
```
      a ← q0
a
┌→┐
↓1│
│0│
└~┘
```

Squared root of the probability of qubit being in ground state

```
      b ← q1
b
┌→┐
↓0│
│1│
└~┘
```

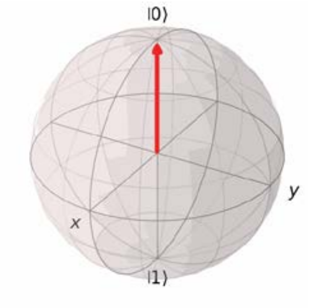
Squared root of the probability of qubit being in excited state

X




```
X
┌→──┐
↓0 1│
│1 0│
└~──┘
```

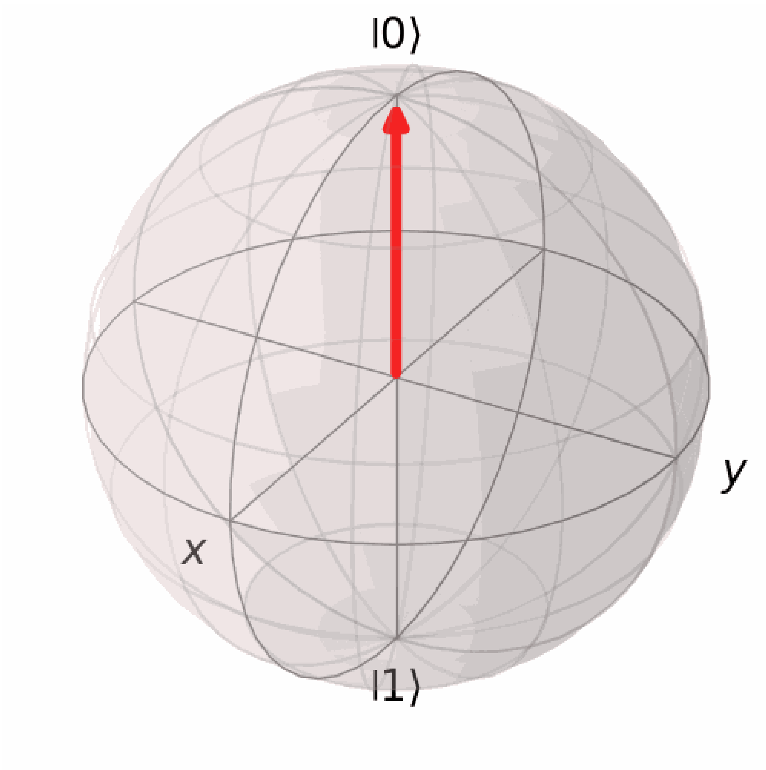
H




```
H
┌→─────────────────────────────────┐
↓0.7071067812 0.7071067812 │
│0.7071067812 ¯0.7071067812│
└~─────────────────────────────────┘
```

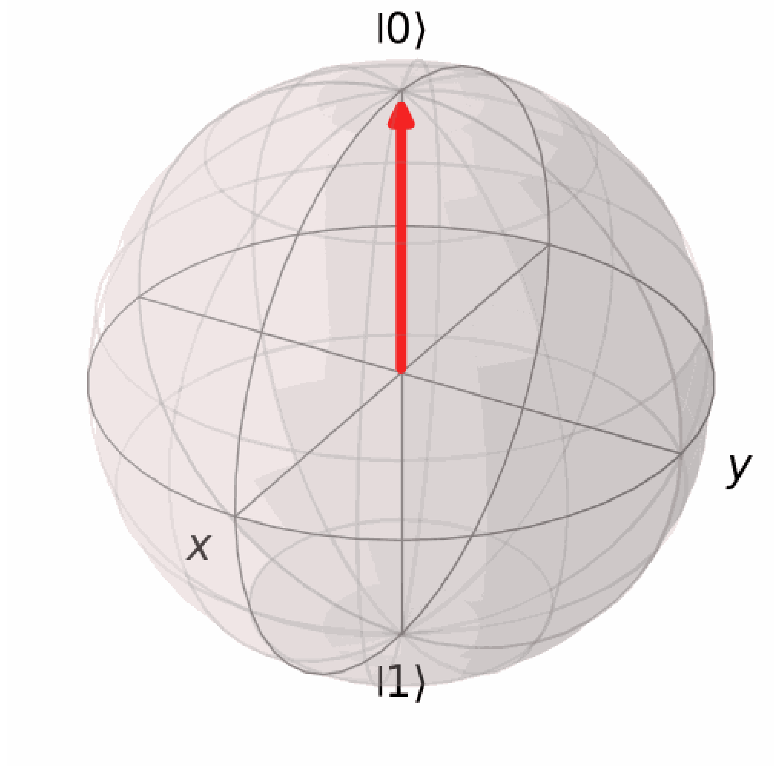
Creating and manipulating qubits

```
      X +.× a
┌→┐
↓0│
│1│
└~┘

      X +.× b
┌→┐
↓1│
│0│
└~┘
```



Creating and manipulating qubits

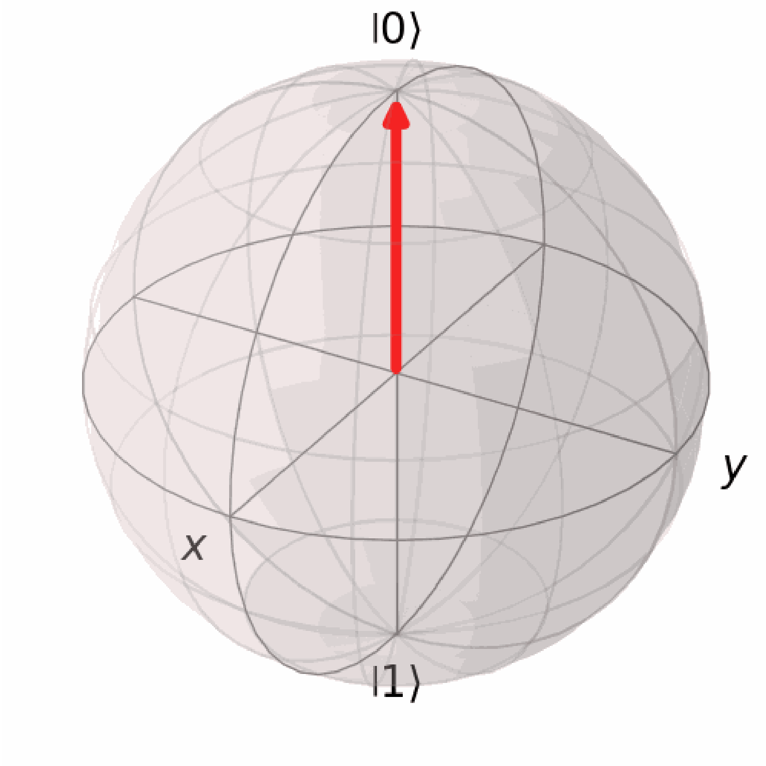
```
      X +.× (X +.× a)
┌→┐
↓1│
│0│
└~┘

      X +.× (X +.× b)
┌→┐
↓0│
│1│
└~┘
```



All quantum logic gates are reversible

Creating and manipulating qubits

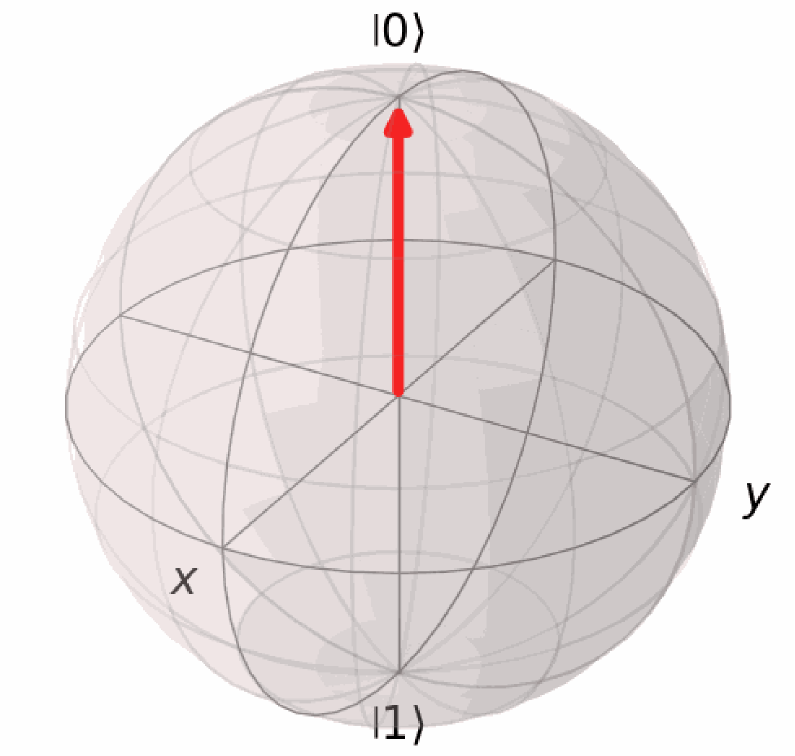
```
      H +.× a
┌→───────────┐
↓0.7071067812│
│0.7071067812│
└~───────────┘

      H +.× b
┌→────────────┐
↓ 0.7071067812│
│¯0.7071067812│
└~────────────┘
```



Creating and manipulating qubits

```
      H +.× a
┌→───────────┐
↓0.7071067812│
│0.7071067812│
└~───────────┘

      H +.× b
┌→────────────┐
↓ 0.7071067812│
│¯0.7071067812│
└~────────────┘
```

Notice the difference in sign. This indicates a difference in phase of the qubit not in the magnitude



Creating and manipulating qubits

```
      H +.× a
┌→───────────┐
↓0.7071067812│
│0.7071067812│
└~───────────┘
```

```
      H +.× b
┌→────────────┐
↓ 0.7071067812│
│¯0.7071067812│
└~────────────┘
```

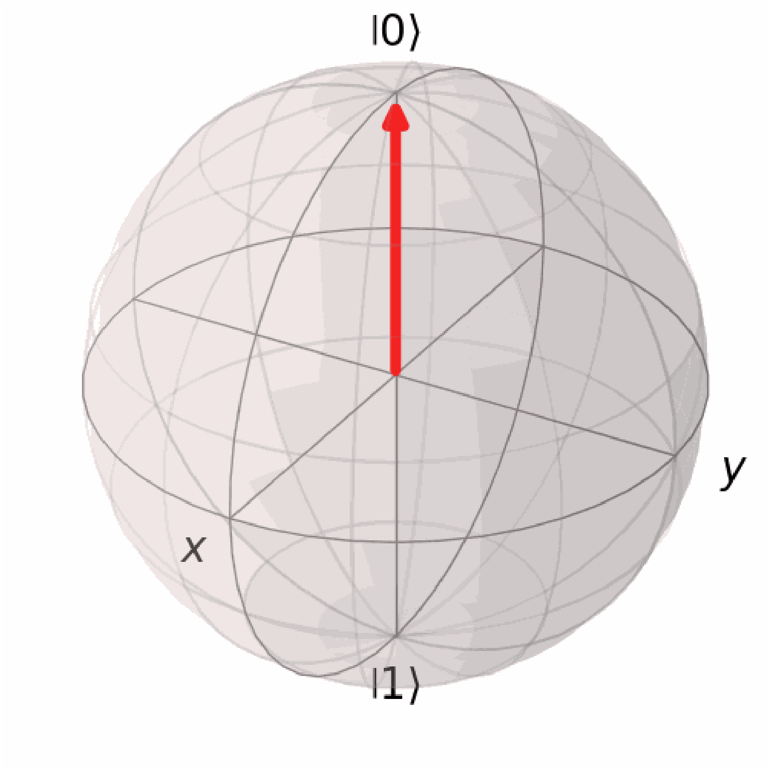
And because of this, the probabilities to measure |0> or |1> are the same

```
      (H +.× a) * 2
┌→──┐
↓0.5│
│0.5│
└~──┘
```

```
      (H +.× b) * 2
┌→──┐
↓0.5│
│0.5│
└~──┘
```

Creating and manipulating qubits

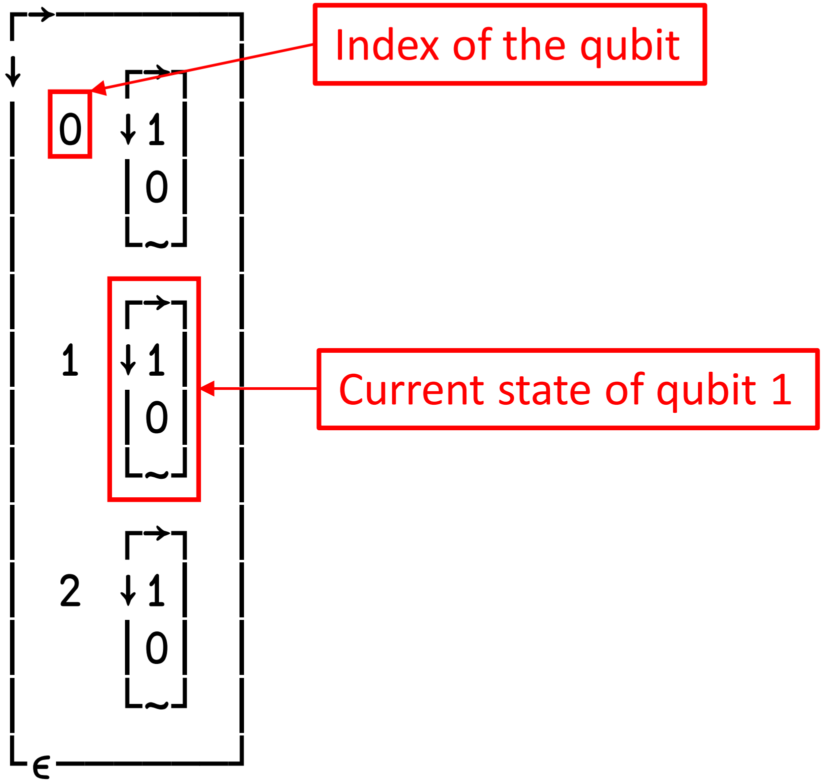
```
      H +.× (H +.× a)
┌→┐
↓1│
│0│
└~┘

      H +.× (H +.× b)
┌→┐
↓0│
│1│
└~┘
```



Multiple qubits and quantum registers

Things get a lot more interesting when you have more than 1 qubit

```
reg_3 ← reg 3
reg_3
```



Multiple qubits and quantum registers

Things get a lot more interesting when you have more than 1 qubit

```
      thread_reg reg_3
┌→┐
↓1│
│0│
│0│
│0│
│0│
│0│
│0│
│0│
└~┘
```

Threading the register into a vector state

Multiple qubits and quantum registers

Things get a lot more interesting when you have more than 1 qubit

```
(tnsidx 3) (thread_reg reg_a)
```

```
┌→──────────────────────┐
│ ┌→────────┐ ┌→┐       │
│ ↓0  0  0  │ ↓1│       │
│ │0  0  1  │ │0│       │
│ │0  1  0  │ │0│       │
│ │0  1  1  │ │0│       │
│ │1  0  0  │ │0│       │
│ │1  0  1  │ │0│       │
│ │1  1  0  │ │0│       │
│ │1  1  1  │ │0│       │
│ └~────────┘ └~┘       │
└∊──────────────────────┘
 0  1  2
```

Each number represents the square root of the probability of the entire state (meaning each individual qubit) of being in 0 or 1 respective of the values on the left.

Columns representing the state of the numbered qubit

Circuit and circuit stages

By circuit we mean a sequence of quantum gates that get applied to a register/vector state

```
      vector_state ← thread_reg reg_a
      (tnsidx 3) (((2) (⊂H)) stage vector_state)
┌→─────────────────────────────────┐
│ ┌→──────┐  ┌→───────────────┐    │
│ ↓0  0  0│  ↓0.7071067812│    │
│ │0  0  1│  │0.7071067812│    │
│ │0  1  0│  │0           │    │
│ │0  1  1│  │0           │    │
│ │1  0  0│  │0           │    │
│ │1  0  1│  │0           │    │
│ │1  1  0│  │0           │    │
│ │1  1  1│  │0           │    │
│ └~──────┘  └~───────────────┘    │
└∊─────────────────────────────────┘
```

Circuit and circuit stages

By circuit we mean a sequence of quantum gates that get applied to a register/vector state

```
vector_state ← thread_reg reg_a
(tnsidx 3) (((2) (⊂H)) stage vector_state) * 2
```

```
┌→────────────────────┐
│ ┌→────┐  ┌→──┐      │
│ ↓0 0 0│  ↓0.5│      │
│ │0 0 1│  │0.5│      │
│ │0 1 0│  │0  │      │
│ │0 1 1│  │0  │      │
│ │1 0 0│  │0  │      │
│ │1 0 1│  │0  │      │
│ │1 1 0│  │0  │      │
│ │1 1 1│  │0  │      │
│ └~────┘  └~──┘      │
└∊────────────────────┘
```

The whole vector state has equal chance of being in any of these 2 stages

Circuit and circuit stages

By circuit we mean a sequence of quantum gates that get applied to a register/vector state

```
vector_state ← thread_reg reg_a
(tnsidx 3) (((0 2) (H H)) stage vector_state)
```

```
┌→──────────────────┐
│                   │
│ ┌→─────┐  ┌→───┐  │
│ ↓0  0  0│  ↓0.5│  │
│ │0  0  1│  │0.5│  │
│ │0  1  0│  │0  │  │
│ │0  1  1│  │0  │  │
│ │1  0  0│  │0.5│  │
│ │1  0  1│  │0.5│  │
│ │1  1  0│  │0  │  │
│ │1  1  1│  │0  │  │
│ └~─────┘  └~───┘  │
│                   │
└∊──────────────────┘
```

Circuit and circuit stages

By circuit we mean a sequence of quantum gates that get applied to a register/vector state

```
vector_state ← thread_reg reg_a
(tnsidx 3) (((0 2) (H H)) stage vector_state) * 2
```

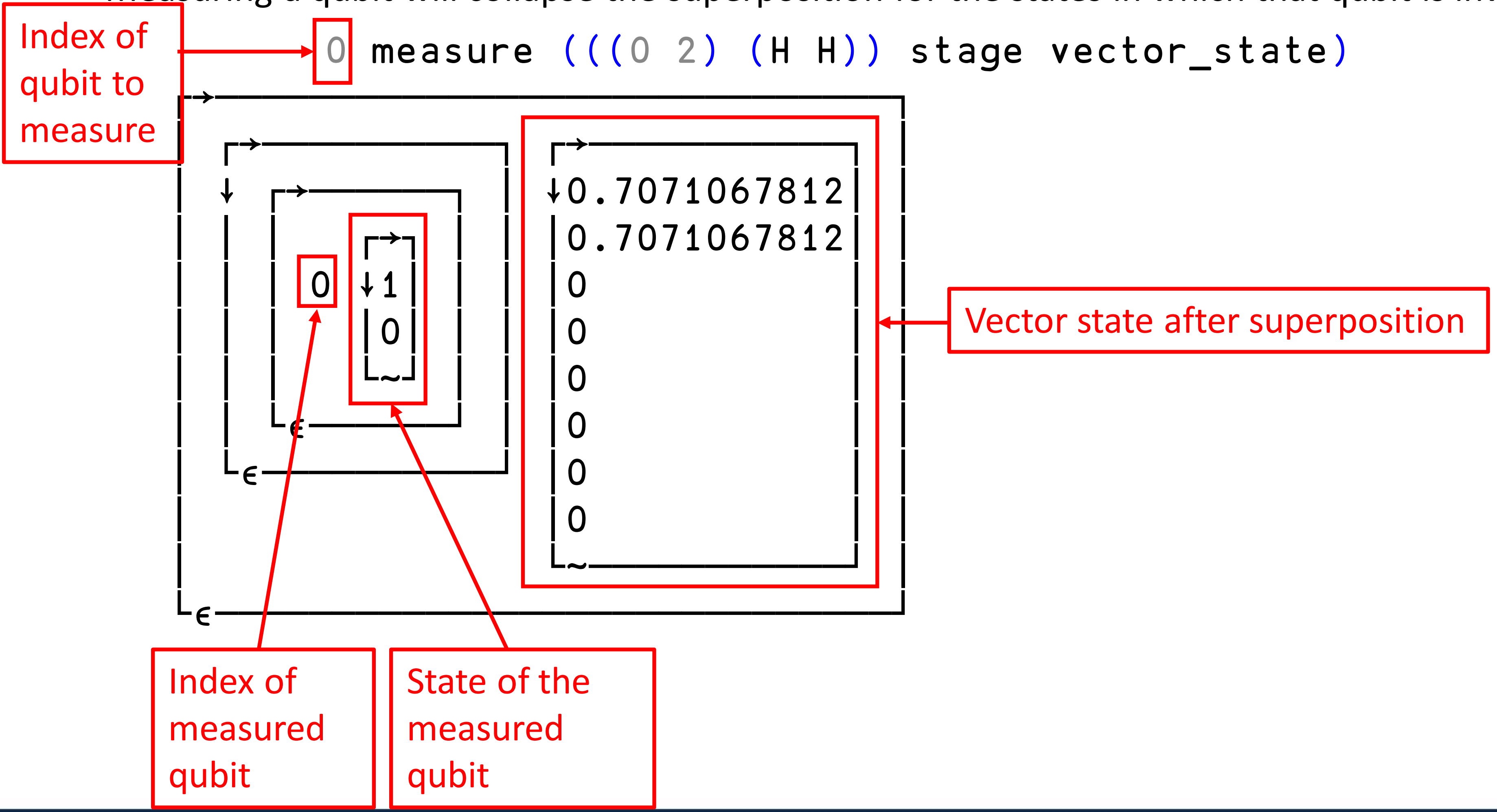
```
┌→────────────────────────┐
│ ┌→────┐  ┌→───┐         │
│ ↓0 0 0│  ↓0.25│         │
│ │0 0 1│  │0.25│         │
│ │0 1 0│  │0   │         │
│ │0 1 1│  │0   │         │
│ │1 0 0│  │0.25│         │
│ │1 0 1│  │0.25│         │
│ │1 1 0│  │0   │         │
│ │1 1 1│  │0   │         │
│ └~────┘  └~───┘         │
└∊────────────────────────┘
```

Once the vector state is squared we can see a 25% chance of measuring any of these states

Measuring collapses the superposition

Measuring a qubit will collapse the superposition for the states in which that qubit is involved

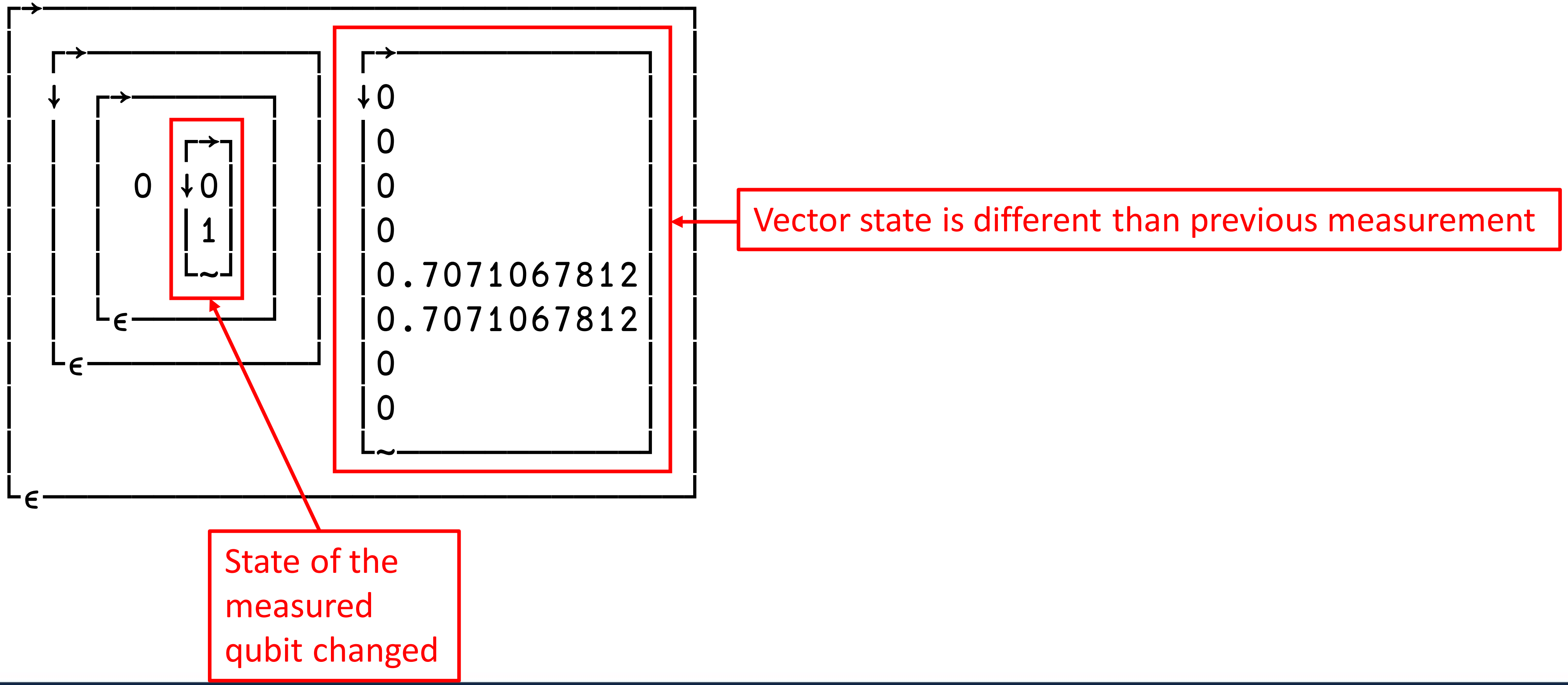
```
0 measure (((0 2) (H H)) stage vector_state)
```



Measuring collapses the superposition

Measuring a qubit will collapse the superposition for the states in which that qubit is involved

```
0 measure (((0 2) (H H)) stage vector_state)
```



Measuring collapses the superposition

Measuring a qubit will collapse the superposition for the states in which that qubit is involved

```
0 2 measure (((0 2) (H H)) stage vector_state)
┌→──────────────────────────┐
│ ┌→───────────────┐ ┌→┐ │
│ ↓ ┌→─────────┐ │ ↓0│ │
│ │ │    ┌→┐ │ │ │0│ │
│ │ │ 0 ↓0│ │ │ │0│ │
│ │ │    │1│ │ │ │0│ │
│ │ │    └~┘ │ │ │1│ │
│ │ └∊─────────┘ │ │0│ │
│ │ ┌→─────────┐ │ │0│ │
│ │ │    ┌→┐ │ │ │0│ │
│ │ │ 2 ↓1│ │ │ └~┘ │
│ │ │    │0│ │ │     │
│ │ │    └~┘ │ │     │
│ │ └∊─────────┘ │     │
│ └∊───────────────┘     │
└∊──────────────────────────┘
```

Measuring collapses the superposition

Measuring a qubit will collapse the superposition for the states in which that qubit is involved

```
0 2 measure (((0 2) (H H)) stage vector_state)
```

```
┌→───────────────────┐
│ ┌→───────────┐ ┌→┐ │
│ ↓ ┌→───────┐ │ ↓1│ │
│ │ │   ┌→┐  │ │ │0│ │
│ │ │ 0 ↓1│  │ │ │0│ │
│ │ │   │0│  │ │ │0│ │
│ │ │   └~┘  │ │ │0│ │
│ │ └∊───────┘ │ │0│ │
│ │ ┌→───────┐ │ │0│ │
│ │ │   ┌→┐  │ │ │0│ │
│ │ │ 2 ↓1│  │ │ └~┘ │
│ │ │   │0│  │ │     │
│ │ │   └~┘  │ │     │
│ │ └∊───────┘ │     │
│ └∊───────────┘     │
└∊───────────────────┘
```

Measuring collapses the superposition

Measuring a qubit will collapse the superposition for the states in which that qubit is involved

```
0 2 measure (((0 2) (H H)) stage vector_state)
┌→──────────────────┐
│ ┌→──────────┐ ┌→┐ │
│ ↓ ┌→──────┐ │ ↓0│ │
│ │ │   ┌→┐ │ │ │1│ │
│ │ │ 0 ↓1│ │ │ │0│ │
│ │ │   │0│ │ │ │0│ │
│ │ │   └~┘ │ │ │0│ │
│ │ └ϵ──────┘ │ │0│ │
│ │ ┌→──────┐ │ │0│ │
│ │ │   ┌→┐ │ │ │0│ │
│ │ │ 2 ↓0│ │ │ └~┘ │
│ │ │   │1│ │ │     │
│ │ │   └~┘ │ │     │
│ │ └ϵ──────┘ │     │
│ └ϵ──────────┘     │
└ϵ──────────────────┘
```

Deutsch-Jozsa algorithm

The problem:

I have a black box function f where:

$$f:\{0,1\}^n \rightarrow \{0,1\}$$

I am promised that f is either *constant* or *balanced*

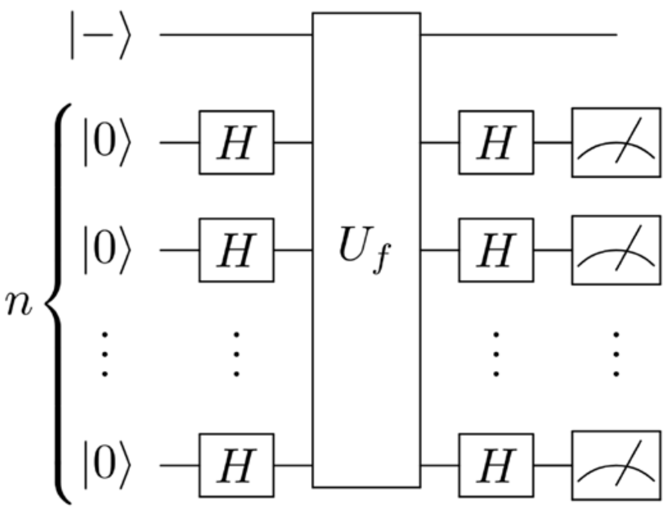
If the function is *constant*, the output of f will always be 0 or 1

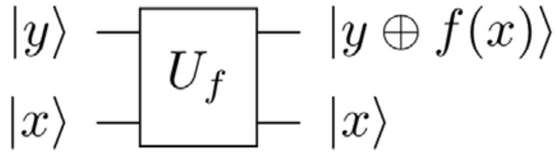
If the function is *balanced*, **exactly half** of the output of f will be 0 while the other **half** will be 1

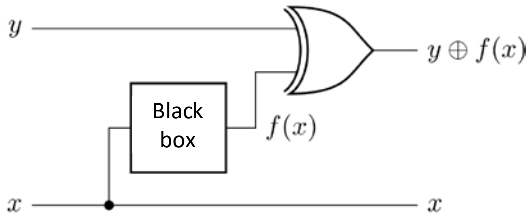
Classical solution:

Try at least half + 1 inputs in the worst of cases

Quantum solution:





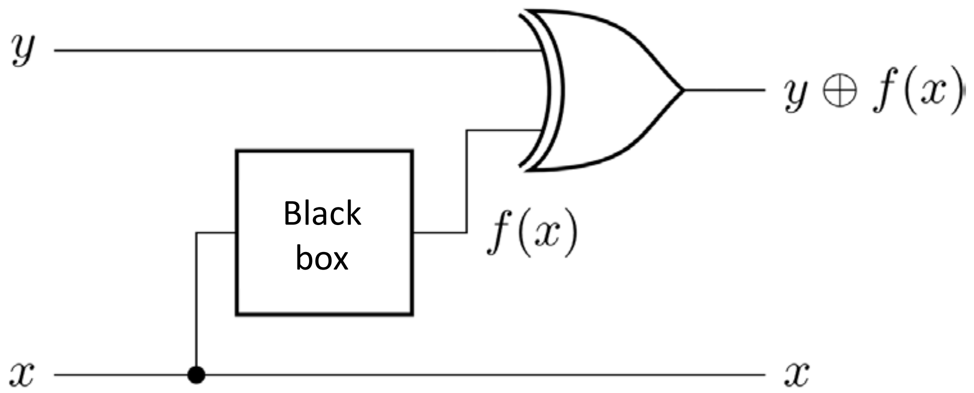


https://www.thomaswong.net/introduction-to-classical-and-quantum-computing-1e3p.pdf

How do we program the black box function

Constant function:

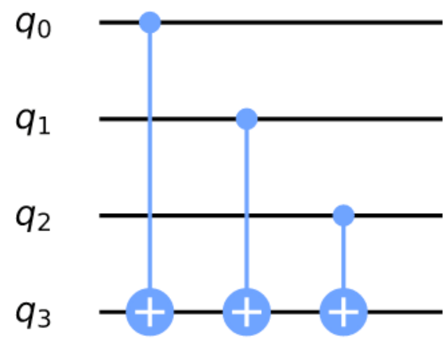
```
zero ← {
        ω
}
```

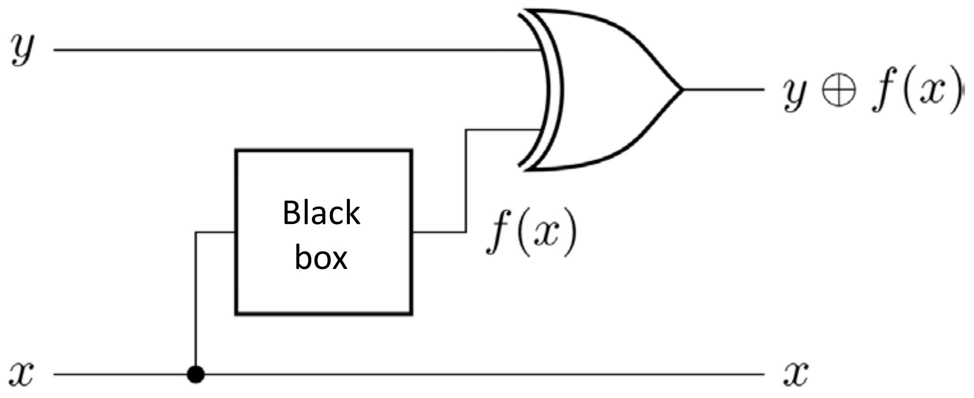


How do we program the black box function

Balanced function:

```
XOR ← {
  n_qubits ← (2⍟1⌷⍴⍵) - 1
  apply ← {
    ⍝ Recursive function that applies an XOR gate
    ⍝ from all qubits into the ancilla.
    ⍝ Ancilla should be in the index 0.
    x ← 1↑⍺
    a ← 1↓⍺
    ret ← (((⊃x)0)(⊂CNOT))stage ⍵
    (⍴a)=0:ret
    a ∇ ret
  }
  (⍳n_qubits)apply ⍵
}
```





Deutsch-Jozsa algorithm

```
_DJ_ ← {
  ⍝ Preps the state according the ancilla qubit.
  ini ← ⍺ prep ⍵
  n_qubits ← (2⍟1⌷⍴ini)

  stg_ctrl ← (((⍳n_qubits)-1)({H}¨⍳n_qubits))

  ⍝ Create the superposition for the oracle
  mid_state ← stg_ctrl stage ini

  ⍝ pass to the oracle
  oracle_state ← ⍺⍺ mid_state
  final_state ← stg_ctrl stage oracle_state

  ⍝ Unprep the state
  ⍺ prep final_state
}
```

Deutsch-Jozsa algorithm

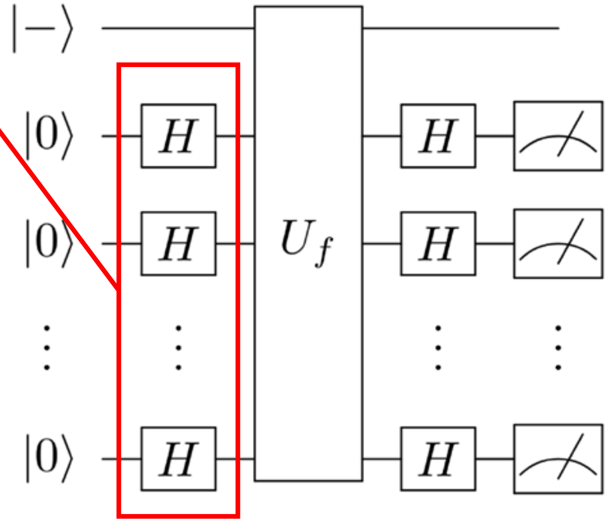
```
_DJ_ ← {
  ⍝ Preps the state according the ancilla qubit.
  ini ← ⍺ prep ⍵
  n_qubits ← (2⍟1⌷⍴ini)

  stg_ctrl ← (((⍳n_qubits)-1)({H}¨⍳n_qubits))

  ⍝ Create the superposition for the oracle
  mid_state ← stg_ctrl stage ini

  ⍝ pass to the oracle
  oracle_state ← ⍺⍺ mid_state
  final_state ← stg_ctrl stage oracle_state

  ⍝ Unprep the state
  ⍺ prep final_state
}
```

```
prep←{
  ⍝ ⍵: Vector state to apply X and SWAP to the
       ancilla qubit
  ⍝ ⍺: Index of the ancilla qubit
  mid_state←((⍺)(⊂X))stage ⍵
  ⍺{⍵:(((0 ⍺)(⊂SWAP))stage mid_state) ⋄
   mid_state}(⍺≠0)
}
```

Deutsch-Jozsa algorithm

```
_DJ_ ← {
  ⍝ Preps the state according the ancilla qubit.
  ini ← ⍺ prep ⍵
  n_qubits ← (2⍟1⌷⍴ini)

  stg_ctrl ← (((⍳n_qubits)-1)({H}¨⍳n_qubits))

  ⍝ Create the superposition for the oracle
  mid_state ← stg_ctrl stage ini

  ⍝ pass to the oracle
  oracle_state ← ⍺⍺ mid_state
  final_state ← stg_ctrl stage oracle_state

  ⍝ Unprep the state
  ⍺ prep final_state
}
```

```
prep←{
  ⍝ ⍵: Vector state to apply X and SWAP to the
       ancilla qubit
  ⍝ ⍺: Index of the ancilla qubit
  mid_state←((⍺)(⊂X))stage ⍵
  ⍺{⍵:(((0 ⍺)(⊂SWAP))stage mid_state) ⋄
   mid_state}(⍺≠0)
}
```



Deutsch-Jozsa algorithm

```
_DJ_ ← {
  ⍝ Preps the state according the ancilla qubit.
  ini ← ⍺ prep ⍵
  n_qubits ← (2⍟1⌷⍴ini)

  stg_ctrl ← (((⍳n_qubits)-1)({H}¨⍳n_qubits))

  ⍝ Create the superposition for the oracle
  mid_state ← stg_ctrl stage ini

  ⍝ pass to the oracle
  oracle_state ← ⍺⍺ mid_state
  final_state ← stg_ctrl stage oracle_state

  ⍝ Unprep the state
  ⍺ prep final_state
}
```

```
prep←{
  ⍝ ⍵: Vector state to apply X and SWAP to the
       ancilla qubit
  ⍝ ⍺: Index of the ancilla qubit
  mid_state←((⍺)(⊂X))stage ⍵
  ⍺{⍵:(((0 ⍺)(⊂SWAP))stage mid_state) ⋄
   mid_state}(⍺≠0)
}
```

Notice how no matter the black function we have, we always need to run it a single time

Running the Deutsch-Jozsa algorithm

As a reminder, our vector state is:

```
zero _DJ_ vector_state
1
0
0
0
0
0
0
0
```

All qubits are in the 0 state, indicating a constant function

```
vector_state
1
0
0
0
0
0
0
0
```

Running the Deutsch-Jozsa algorithm

As a reminder, our vector state is:

```
        XOR _DJ_ vector_state
0
0
0
1
0
0
0
0
```

```
        vector_state
1
0
0
0
0
0
0
0
```

Running the Deutsch-Jozsa algorithm

```
(tnsidx 3) (XOR _DJ_ vector_state)
 0 0 0|0
 0 0 1|0
 0 1 0|0
 0 1 1|1
 1 0 0|0
 1 0 1|0
 1 1 0|0
 1 1 1|0
```

Any non-0 qubit indicate that the function is balanced

As a reminder, our vector state is:

```
vector_state
1
0
0
0
0
0
0
0
```

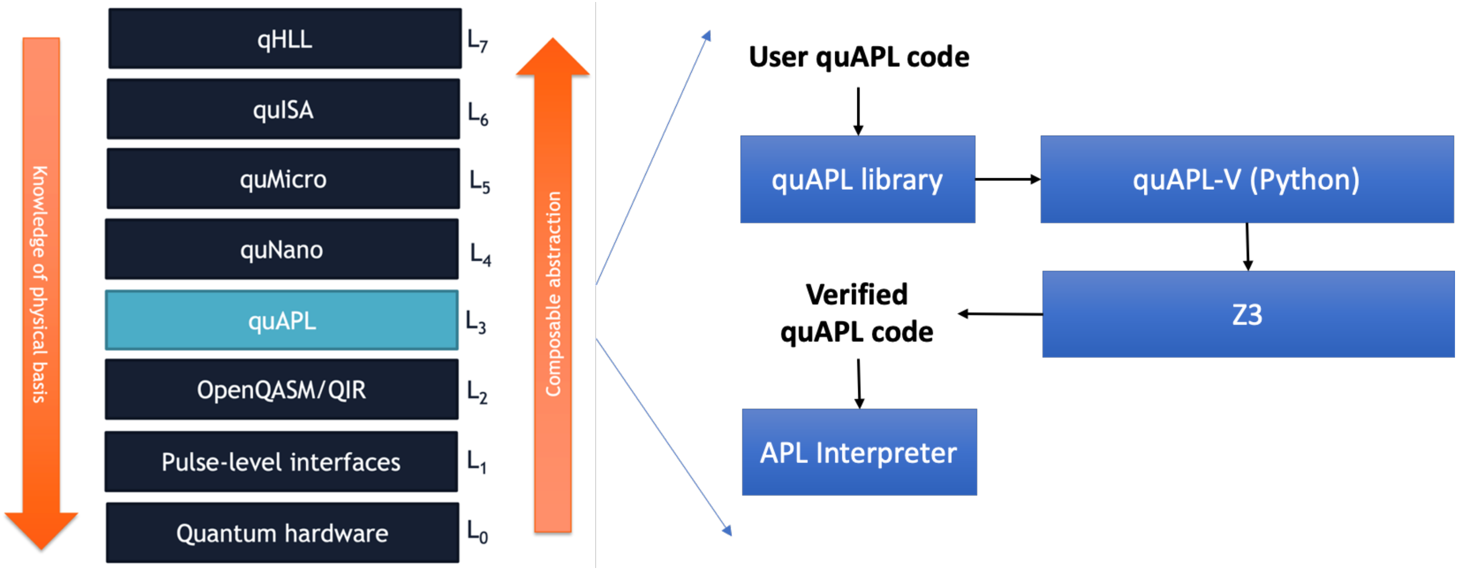
Ongoing work

Core language library
- Library of base quantum algorithms (QAs)
- Extraction of quantum motifs
- Reimplementation of QAs using motifs
- Noisy simulation

Code generation
- OpenQASM, QIR
- Pulse-level modeling
- Hardware control

APL language
- New glyphs for common quantum motifs
- Implementation of suggested extensions

Software reliability
- Formal verification with Z3 (Phuong Cao, NCSA)



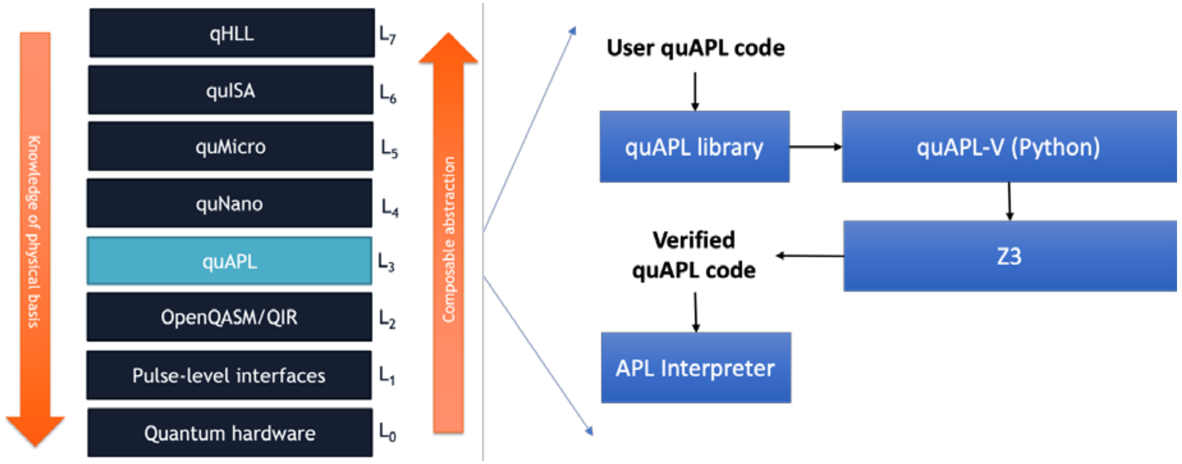
Ongoing work

Core language library
- Library of base quantum algorithms (QAs)
- Extraction of quantum motifs
- Reimplementation of QAs using motifs
- Noisy simulation

Code generation
- OpenQASM, QIR
- Pulse-level modeling
- Hardware control

APL language
- New glyphs for common quantum motifs
- Implementation of suggested extensions

Software reliability
- Formal verification with Z3 (Phuong Cao, NCSA)



Collaborators needed! Help wanted!

nunezco2@illinois.edu marcosf2@Illinois.edu