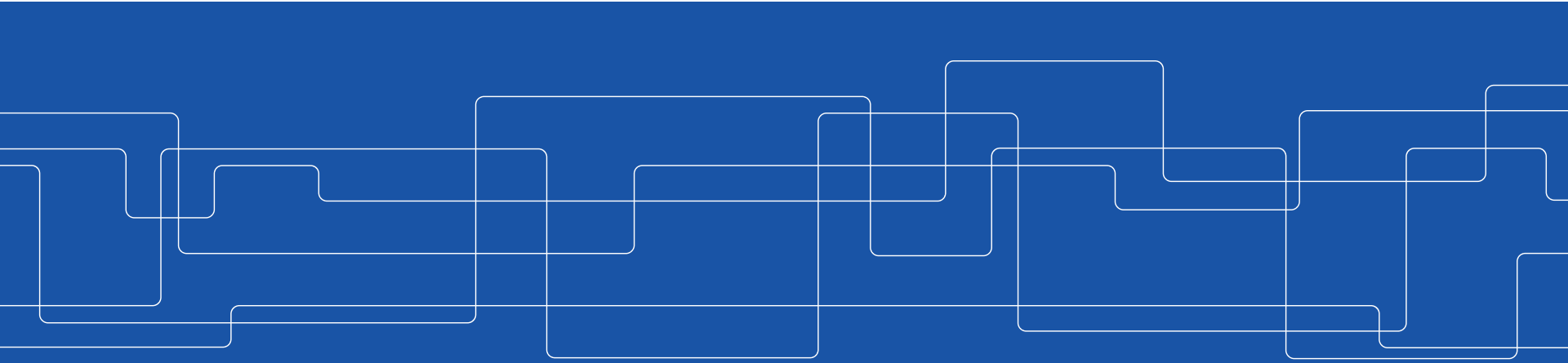
Future Programming Models

How to Program a QPU

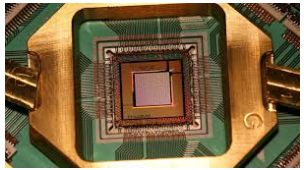
S. Markidis, I.B. Peng, S. Rivas-Gomez and E. Laure (KTH)

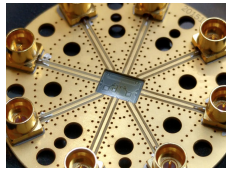


Most of the material from this lecture is courtesy of: https://medium.com/@decodoku/how-to-program-a-quantum-computer-982a9329ed02

QPU = Quantum Processing Units

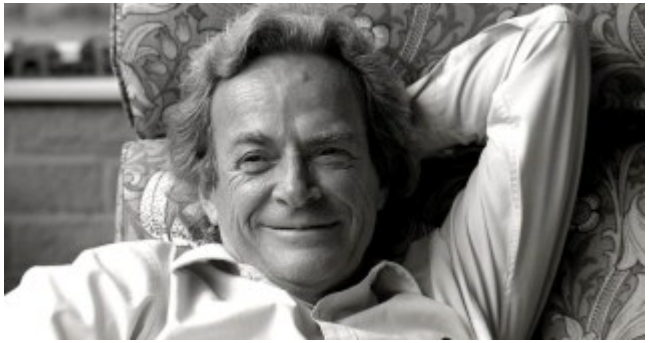
- **QPUs** are devices that implement the principles of quantum computing
  - Many different technologies demonstrated
  - Sequences of operations demonstrated
- Early stage vendors are offering QPU access
  - D-Wave, **IBM**, Google, Rigetti
  - Client-server or host/device interaction model
  - Loose integration with modern computing





Courtesy of https://www.dwavesys.com/, http://rigetti.com, https://researchweb.watson.ibm.com/ibm-q/

Why QPU?



***Nature isn't classical**, **dammit,** and if you want to make a **simulation of nature**, you'd better make it **quantum mechanical**.*

*Can a **quantum system** can be **simulated exactly** by a universal computer? **NO!***

Richard Feynman, *Simulating Physics with Computers*

Timeline of Quantum Computers

- **Early 90s** - quantum computing was codified to exploit capabilities of quantum physics:
  - Use ‘’inherent parallelism’’ of quantum systems
  - Exponential speed ups over selected classical algorithms, i.e. factorization
- **For 20 years** - quantum technologies remain proof of concept
  - R&D with significant basic research investments
  - Left with a large, diverse quantum technology base
- **In the 2010s** - research and development began to address system-level concerns
  - Microarchitecture
  - Programming
  - Macro-architecture

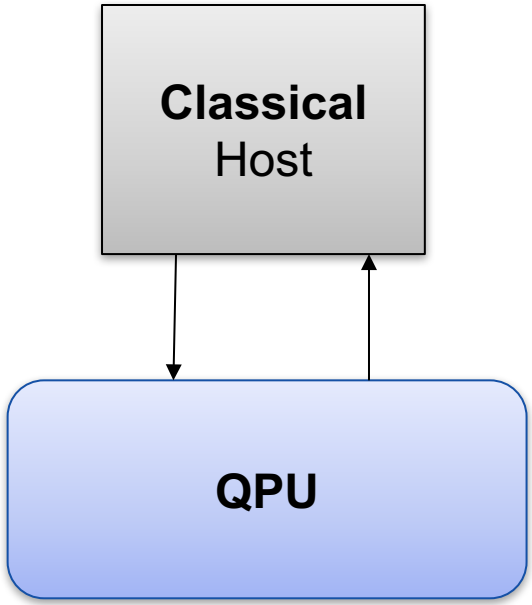
Quantum Computers are Hybrid Systems

Quantum computers will be always **hybrid** devices, partly quantum (**QPU**) and partly classical (Host).

The classical host is required to handle inputs and outputs.

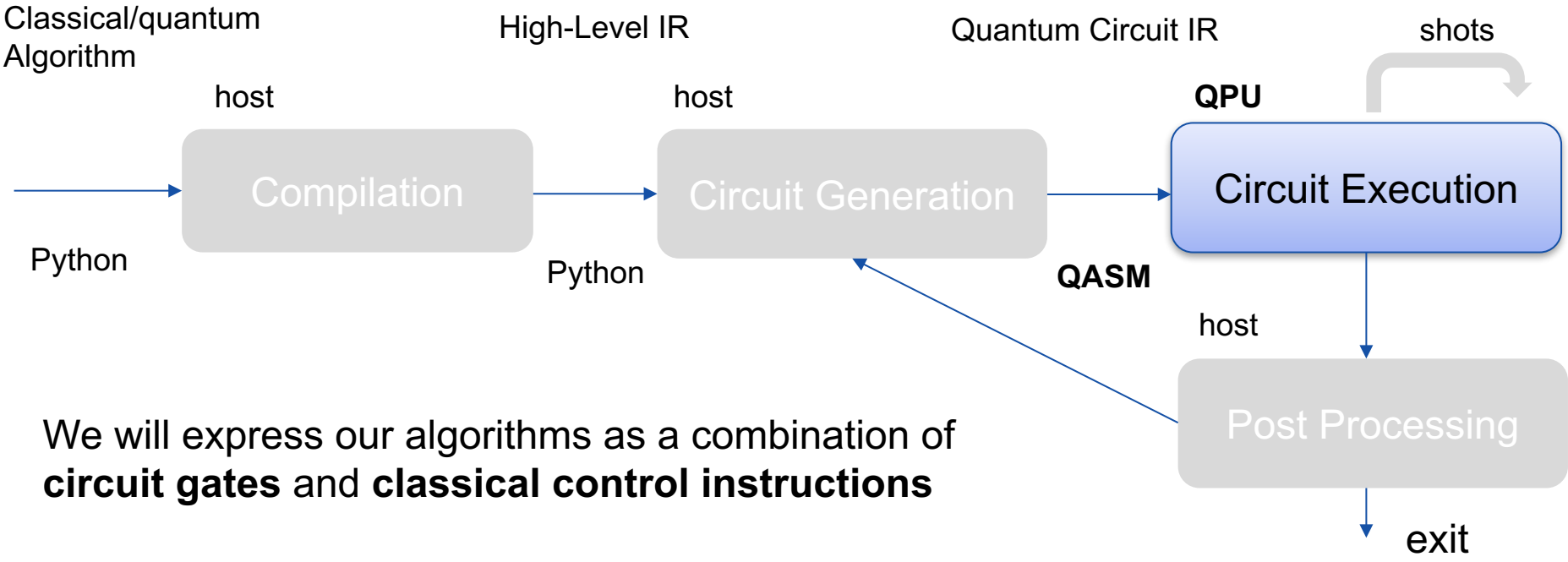
The quantum device is used to accelerate computations

Question: what does this remind you of?



Basic Programming Approach

IR = Intermediate Representation



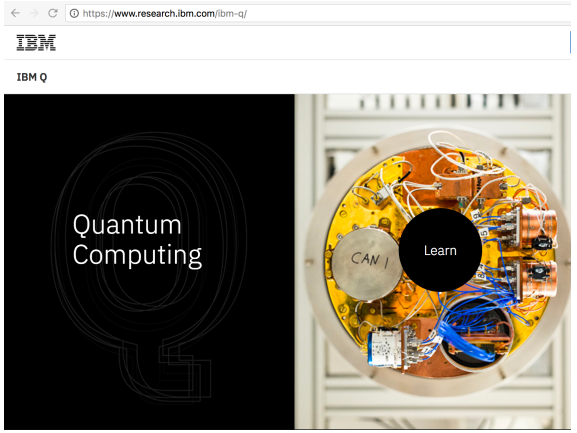
We will express our algorithms as a combination of **circuit gates** and **classical control instructions**

The IBM Quantum Information Software Kit

We will use the IBM Quantum Information Software Kit (**`qiskit`**) that is a Python development kit to create quantum programs, compile and run them on an online quantum computer called **IBM Q**.

Python code used for:

- Classical I/O
- Create quantum circuit
- Compile the code
- Execute it on a remote IBM quantum computer or on local emulator.



Requirements for running your Quantum Program

You will need:

- **Python3**
- **Jupyter** notebook (best to install Anaconda that includes it)

Get the **`qiskit`** with:

- `git clone` https://github.com/QISKit/qiskit-sdk-py

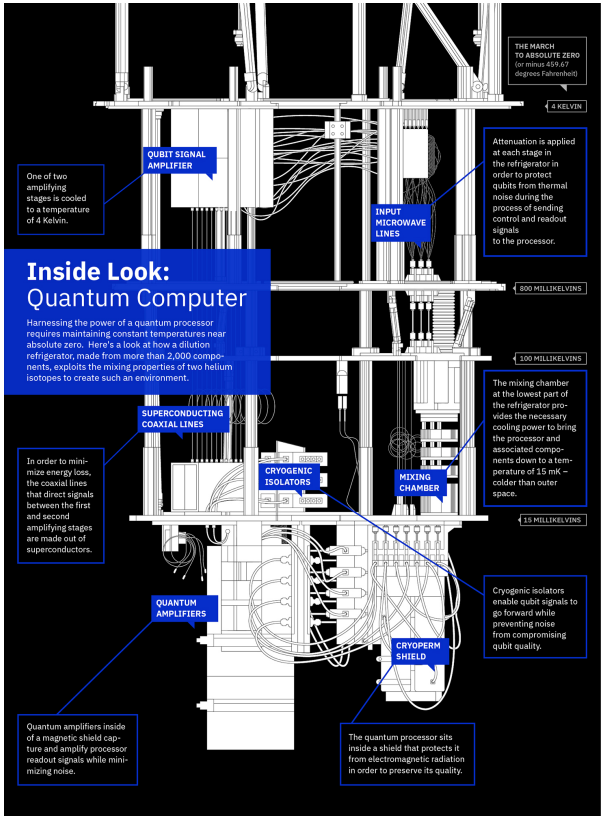
Create an IBM Quantum Experience account to connect and use the IBM Q quantum computer

- you will have 5 credits for free

For detailed info, check https://github.com/QISKit/qiskit-sdk-py

IBM Q itself

https://www.youtube.com/watch?v=CuC-9e8hfi0



Quantum Program: Battleships Game

Game between 2 human players:

- Decide where to place the ships
- Decide where to bomb

Use a Japanese variant of Battleships game:

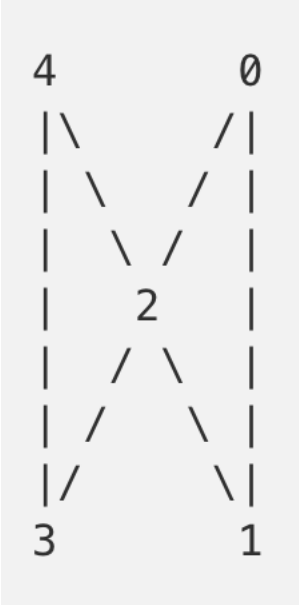
- all ships take up **only a single square**, but some take more hits to sink than others. The first ship can be sunk by a single bomb, the second needs two and the third needs three.
- The state `0` = **fully intact ship**, and `1` = ship that has been **destroyed**. Some boat might be partially damage and needs more bombs to be sunk.

Battleships game with 5 bit

We will use a 5 bit quantum computer (IBM Q):

- Each player choose where to put three ships. Five possible positions are available, corresponding to the five bits on **IBM's ibmqx2 chip**.
- The first ship from input requires 1 bomb to be sunk, second one 2 bombs and third 3 bombs.
- Where we have ship we can initialize as state `0`, and then apply a `NOT` when it gets hit. When we find it in state `1`, we'll know it has been destroyed.

We will learn how to use **quantum computer to represent the partially damaged ships.**



Question: Why do we display the grid like this?

Quantum Computer: qubit instead of bit

In order to implement our game we will need to use the quantum equivalent of bit, called **quantum bits** or **qubits**, and use quantum gates.

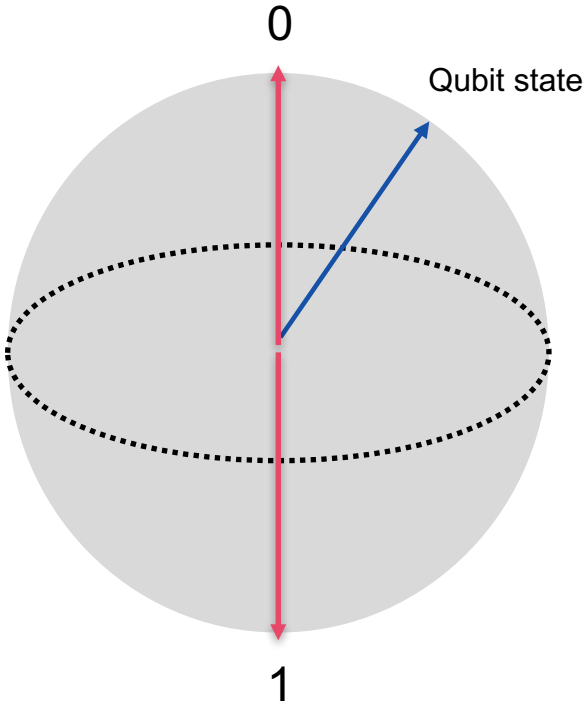
Like normal bit, qubit have also two possible values which can call `0` and `1`. But the laws of quantum mechanics also allow **other possibility**, which we call **superposition of states**.

The superposition of different states

Quantum Bit - Qubit

We can picture a qubit as a sphere of radius 1 with `0` and `1` sitting on opposite poles. The superposition states are all the other possible points on the surface.

Superposition states are values that exist part way between the extreme of `0` and `1`.



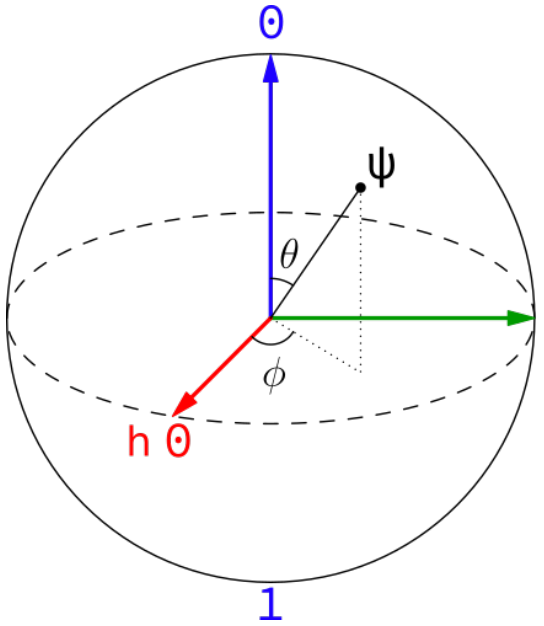
Quantum Variables: part continuous, part discrete variables

Qubits seem like a **continuous variable**: they can be any point of a sphere.

However, we can **never extract more than a binary information from a qubit**:

- we can't ask for the exact details of the superposition state, we only force it choose between `0` and `1`.

Qubit is a **quantum variable** as **it has property of continuous and discrete ones**



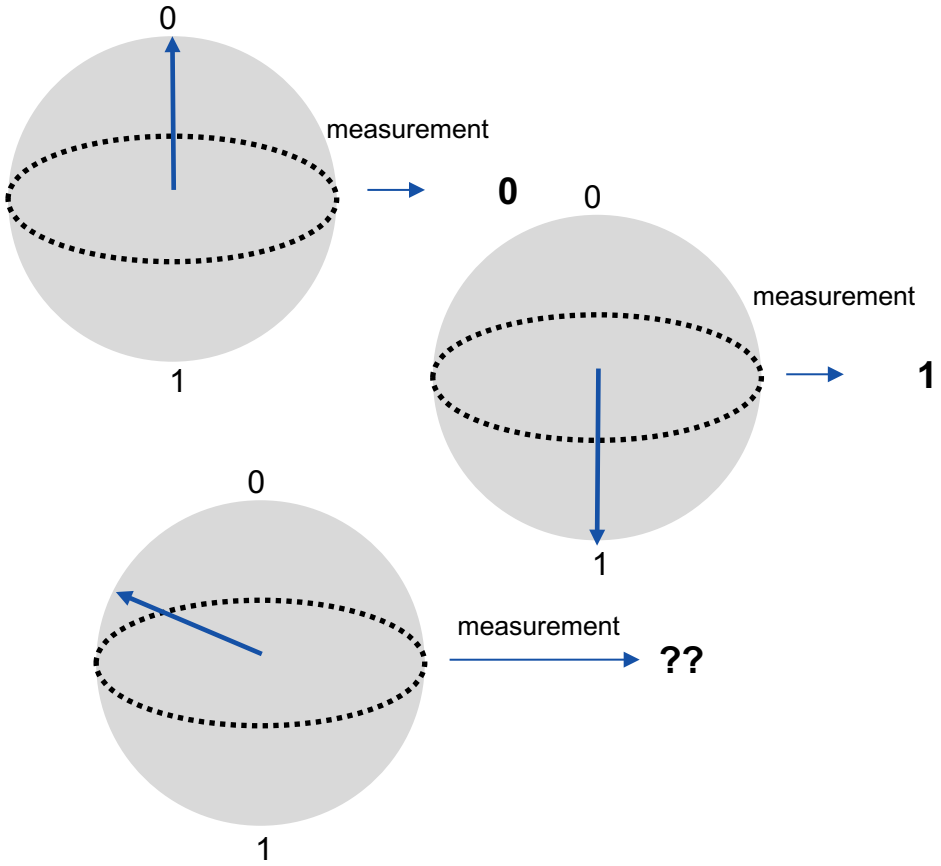
Measuring a Qubit

Any measurement is simply us asking a qubit to choose between two opposite `0` and `1`.

If the qubit is in the state `0`, it will go for `0` in the measurements. A qubit in state `1`, similarly will give the result `1`.

For any other state, **the outcome will be random** with the closest option being the most likely.

Question: how do I get to know the unknown state?

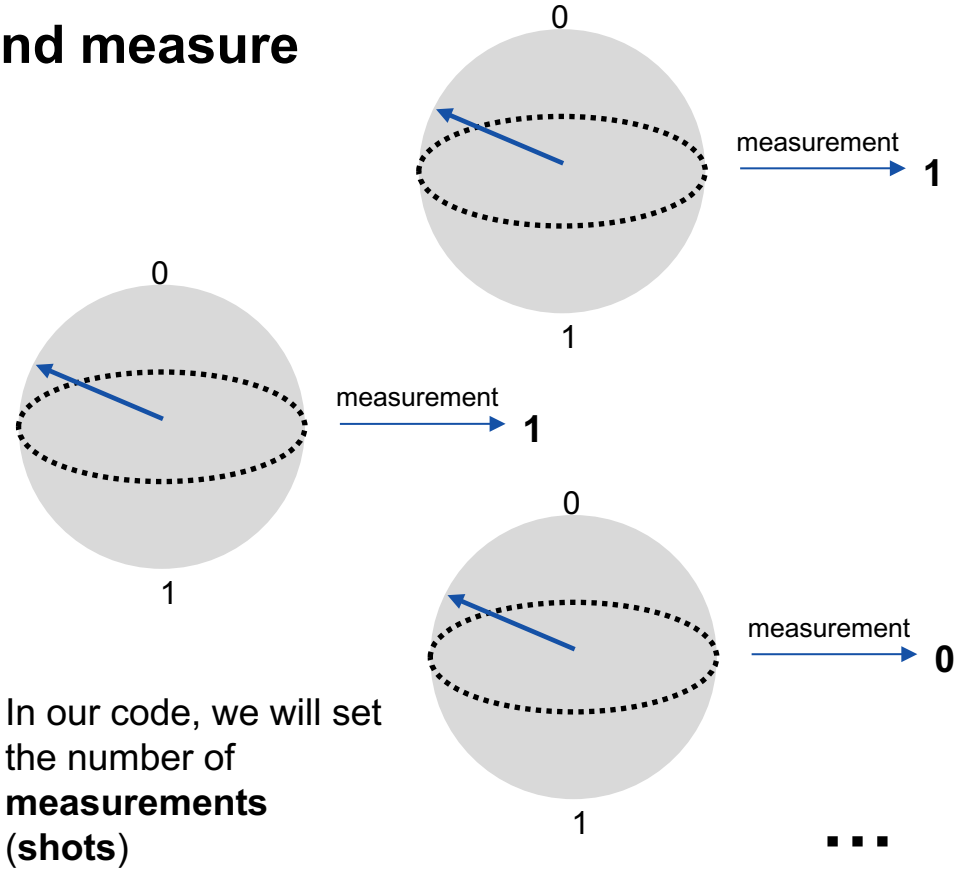


Make several experiments and measure

it is possible to determine an unknown state with arbitrary precision if an ensemble of **N identically prepared copies of the unknown quantum state** is available.

If we perform a measurement on each of the N independent copies, we obtain a sequence of N independent measurement results giving us probability distributions.

For large N, the relative frequency approaches the expectation value



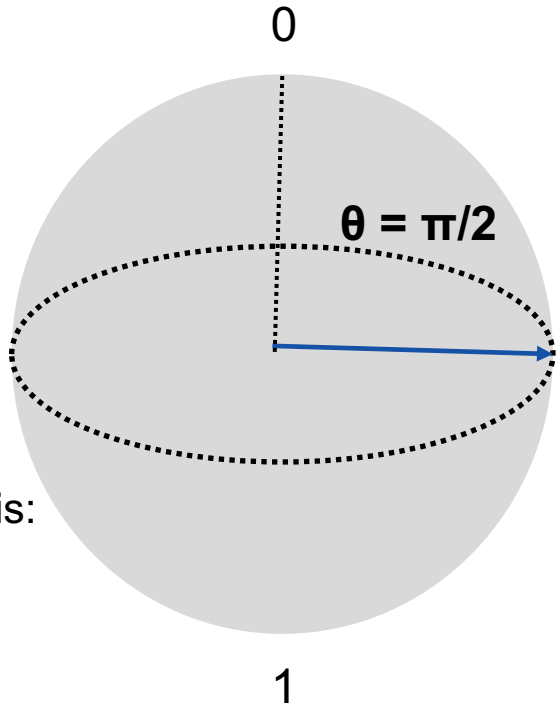
In our code, we will set the number of **measurements** (**shots**)

Question: what will be the measurement of this state?

On the equator, it is a **50/50 chance** either way.

If we do a large number of measurements:

- 50% will be `0`
- 50% will be `1`



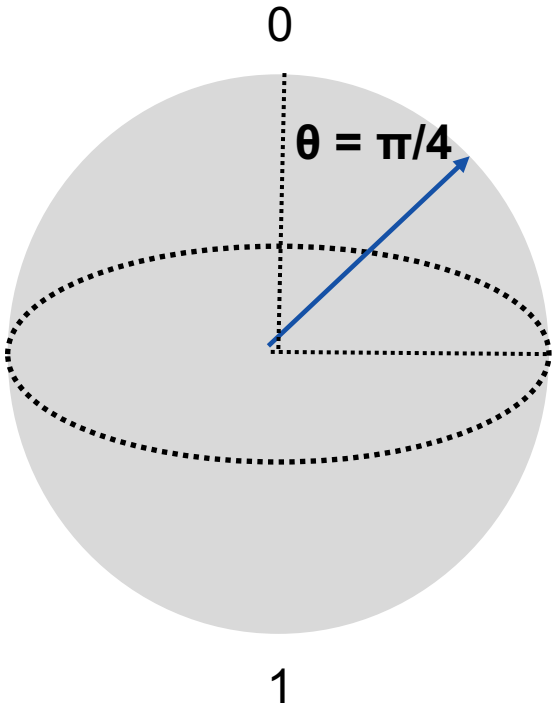
A simple formula to calculate the probability of the two states is:

- Prob(0) = $\cos^2(\theta/2) = \cos^2(\pi/4) = 0.5$
- Prob(1) = $\sin^2(\theta/2) = \sin^2(\pi/4) = 0.5$

Question: what will be the measurement of this state?

If we do a large number of measurements:

- $\cos^2(\theta/2) = \cos^2(\pi/8) =$ 0.853 will be 0
- 1 − 0.853 = .147 will be 1

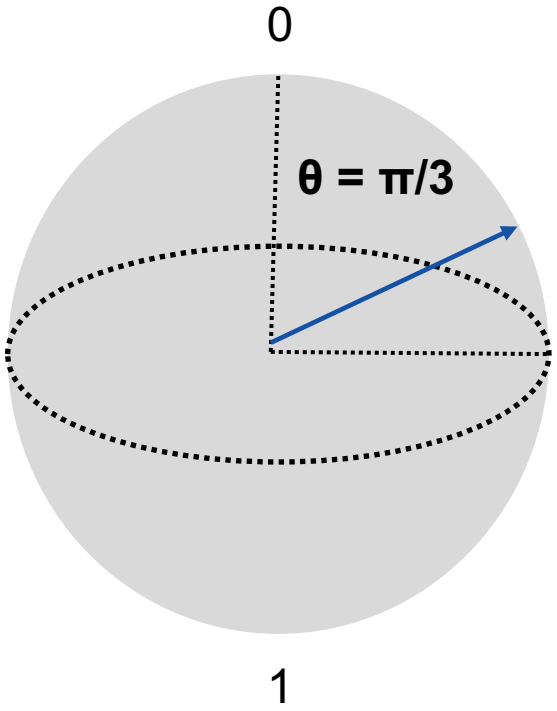


Question: what will be the measurement of this state?

So if we do a large number of measurements:

- $\cos^2(\pi/6) = 0.75$ will `0`
- `1 - 0.75 =` 0.25 will be `1`

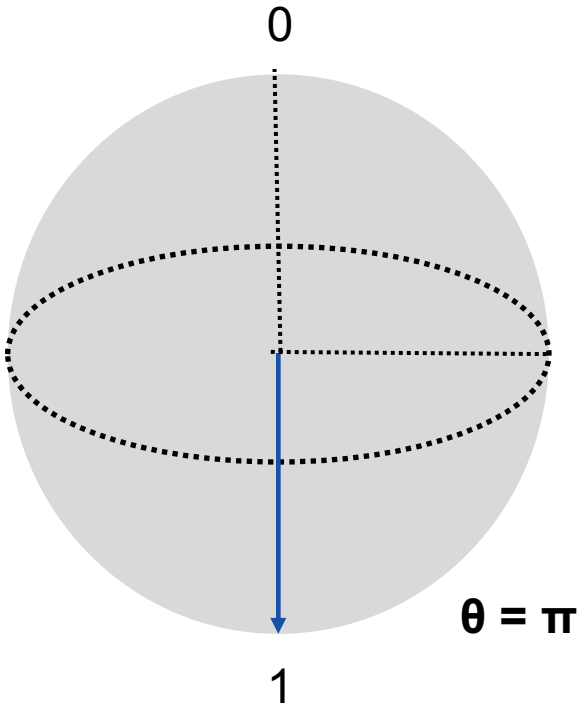
Remember this result: we will use it later!



Question: what will be the measurement of this state?

So if we do a large number of measurements:

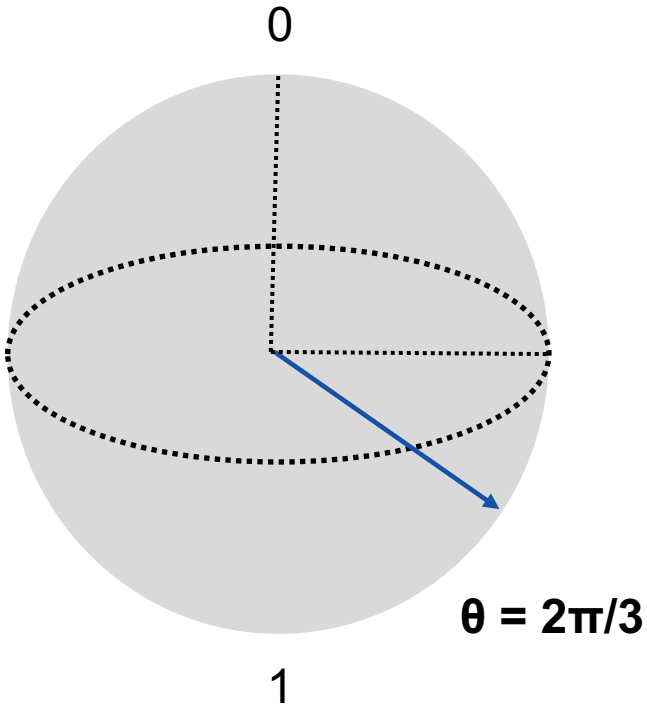
- **X** % will be `0`
- **X** % will be `1`



Question: what will be the measurement of this state?

So if we do a large number of measurements:

- 25 % will be `0`
- 75 % will be `1`



Initialize qubit for a ship

To simulate this on a quantum computer, we can use a **qubit for storing the information of a ship that is being bombed.**

With QASM, we can define a single qubit in a register called `q`. We refer to this qubit in code as `q[0]`.

Since outputs have to be in nice, human-readable normal information, we also define a single normal bit in a register called `c`.

QASM code for initialization

```
OPENQASM 2.0;
include "qelib1.inc";
\\ quantum register is initialized to 0 pure state
\\ Initialize a register with a single qubit
qreg q[1];
\\ Initialize a register with a normal bit
creg c[1];


measure q[0] -> c[0]; \\ measure the qubit
```

Measuring qubit

The last line of our QASM file is

`measure q[0] -> c[0];`

In this we measure the qubit. We tell `q[0]` that it has to decide what to be: `0` or `1`. The value of `c[0]` is then the output of this computation.

The qubit `q[0]` is automatically initialized in the state `0`.

QASM code for initialization

```
OPENQASM 2.0;
include "qelib1.inc";
\\ quantum register is initialized to 0 pure state
\\ Initialize a register with a single qubit
qreg q[1];
\\ Initialize a register with a normal bit
creg c[1];


measure q[0] -> c[0]; \\ measure the qubit
```

Question: what is it the result of the measure?

Question: does the measure give always the same measure?

NOT to destroy a battleship

A battle ship destroyed by a single hit will be pretty easy to simulate.
We can initialize it in state `0`, and then apply a NOT when it gets hit.
When we find it in state `1`, we'll know it has been destroyed.

So for our game, we want to take `q[0] = 0`, a fully intact ship, and perform a NOT, a fully destroyed ship.

We will implement a NOT using a QASM `u3` gate

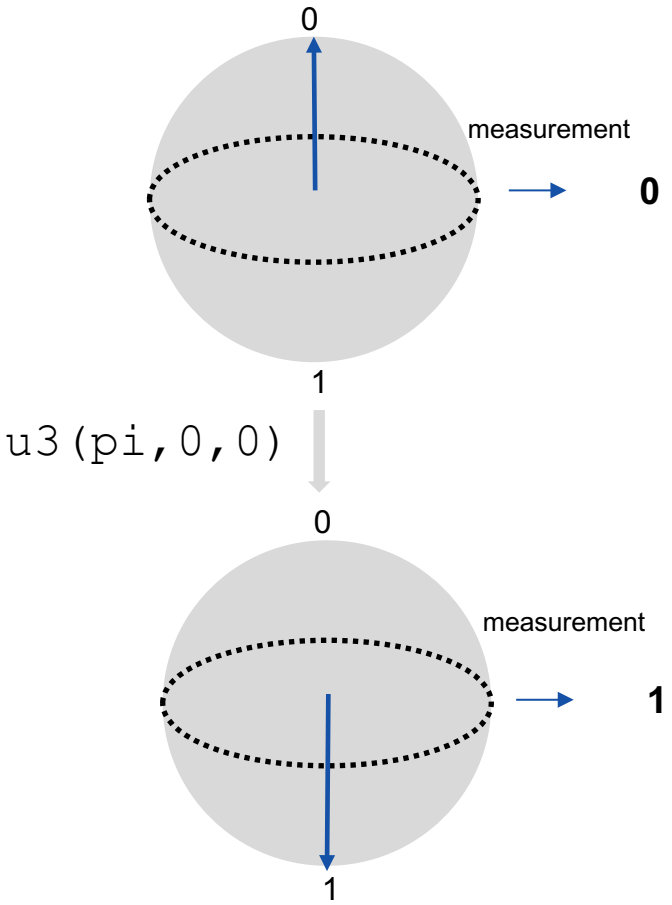
```
u3(pi,0,0) q[0];
```

u3 gate

But `u3` is an operation with three arguments that are the angles expressed in radians.

The first argument is the angle by which we are going to turn the sphere of our qubit around.

- The angle `pi` corresponds to 180°, and so means we turn the sphere completely upside down. `0` moves to `1` and `1` moves to `0`, which is why this operation acts as a **NOT**.



Question: how can we do ½ NOT?

To do half a NOT we could simply use half this angle:

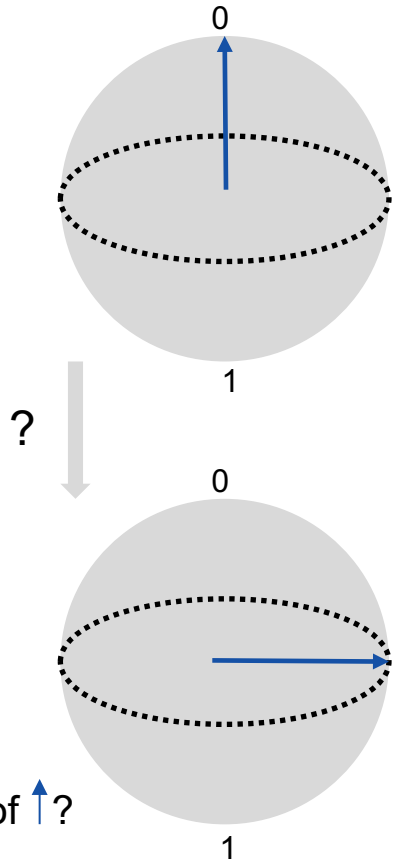
```
u3(0.5*pi,0,0)
```

Also, we have yet another way to perform a NOT on our qubit: We could do half a NOT twice:

```
u3(0.5*pi,0,0) q[0];
u3(0.5*pi,0,0) q[0];
```

We could also do a third of a not thrice, or a quarter of a NOT …

Question: what will be the result of a single measurement of **1/2 NOT** of ↑?

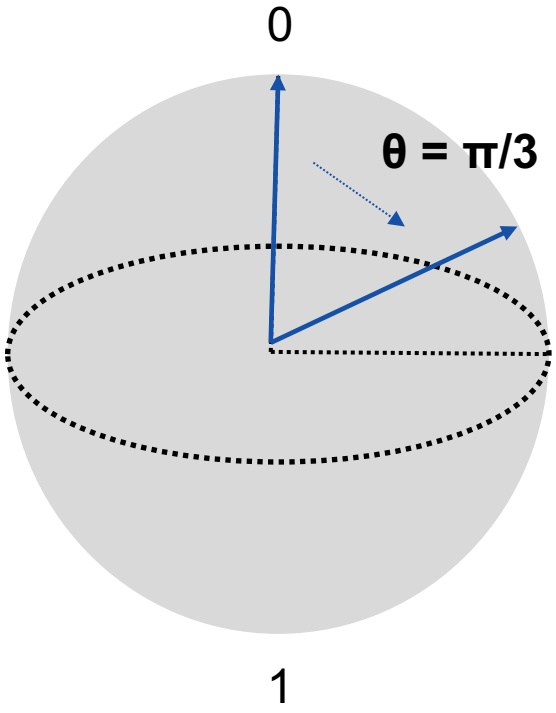


Our first Quantum Program

Implement and run our first quantum program to:

- Initialize classical and quantum registers
- Create a Quantum Circuit to perform 1/3 NOT operation
- Measure and store result in classical register

We will then compile it and execute it.



0. Loading modules for Quantum Program

```
# Checking the version of PYTHON; we only support 3 at the moment
import sys
if sys.version_info < (3,0):
        raise Exception('Please use Python version 3 or greater.')
# Import the QuantumProgram and our configuration
sys.path.append('../../qiskit-sdk-py/')
import math
from pprint import pprint
from qiskit import QuantumProgram
import Qconfig
```

1. Create Quantum Circuit

```
# instantiate Quantum program
qp = QuantumProgram()

# quantum register for the first circuit
q1 = qp.create_quantum_register('q1', 1)
c1 = qp.create_classical_register('c1', 1)
qc1 = qp.create_circuit('OneThirdNOT', [q1], [c1])
qc1.u3(3.1415/3,0,0,q1[0])
qc1.measure(q1[0], c1[0])
```

2. Check the generated QASM

```
print(qp.get_qasm('OneThirdNOT'))
```

Output:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q1[1];
creg c1[1];
u3(1.047166666666667,0.000000000000000,0.000000000000000) q1[0];
measure q1[0] -> c1[0];
```

This is somehow similar to `-s` flag to produce assembly code during classical compilation

3. Compile the Quantum Code

```
qobj = qp.compile(['OneThirdNOT'], backend='local_qasm_simulator')
```

There are different backend we can use:

- `ibxm2`, real quantum computer with 5 qubits
- `ibxm3`, real quantum computer with 16 qubits
- `local_qasm_simulator`, qskit simulator coming when you install software on your computer

4. Execute the Code

```
result = qp.execute(['OneThirdNOT'],
backend='local_qasm_simulator', shots=1024)
```

When we execute the code, we need to set-up the number of measurements (=`shots`). Higher number of measurements leads to higher measurement precision.

5. Retrieve the result

```
result.get_counts('OneThirdNOT')
```

```
Output:
{'0': 787, '1': 237}
```


Question 1: What does this mean?

We performed 1024 experiments:

- 787 were `0` → 787/1024 = 0.7685
- 237 were `1` → 237/1024 = 0.2314

Question 2: Is what we expected?

Question3: How can we improve the solutions?

6. Improve estimate increasing the number of measurements

```
result = qp.execute(…  , shots=1000)
result.get_counts('OneThirdNOT')
{'0': 770, '1': 230}

result = qp.execute(…  , shots=10000)
result.get_counts('OneThirdNOT')
{'0': 7480, '1': 2520}

result = qp.execute(…  , shots=100000)
result.get_counts('OneThirdNOT')
{'0': 75014, '1': 24986}


result = qp.execute(…  , shots=1000000)
result.get_counts('OneThirdNOT')
{'0': 749507, '1': 250493}
```

Question: which numerical technique looks like this quantum computation?
Hint: we implemented it in the lab!

Question 2: how does measurement precision scale? $1/N^{1/2}$

General strategy for implementing Battleships Game

All I/O handled by host:

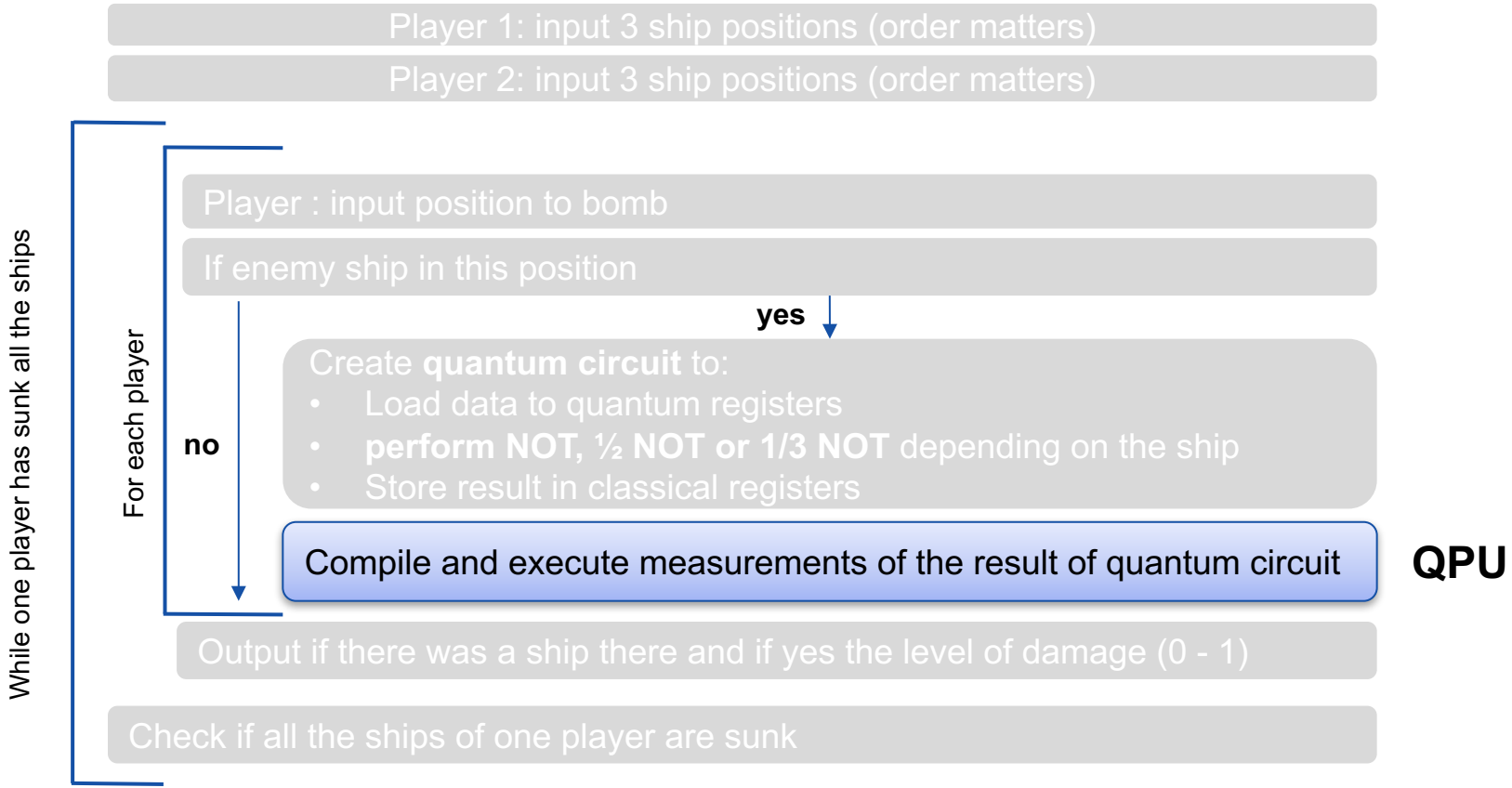
- Input ship position
- Print the results

All control of execution is done by host:

- Check if a position that is bombed by player has battleship and how much is damaged:
  - If boat #1 then create quantum circuit for NOT
  - If boat #2 then create quantum circuit for ½ NOT
  - If boat #3 then create quantum circuit for 1/3 NOT
- Check if all the ships of one player are sunk are sunk and finish the game

Host offloads computation to QPU, depending on we need NOT, 1/2 NOT or 1/3 NOT:

- Load to input to quantum register, compute `u3(pi,0,0)`, store result to classical register
- Load to input to quantum register, compute `u3(pi/2,0,0)`, store result to classical register
- Load to input to quantum register, compute `u3(pi/3,0,0)`, store result to classical register


Player 1: input 3 ship positions (order matters)
Player 2: input 3 ship positions (order matters)
While one player has sunk all the ships
For each player
Player : input position to bomb
If enemy ship in this position
yes
no
Create quantum circuit to:
• Load data to quantum registers
• perform NOT, ½ NOT or 1/3 NOT depending on the ship
• Store result in classical registers
Compile and execute measurements of the result of quantum circuit
QPU
Output if there was a ship there and if yes the level of damage (0 - 1)
Check if all the ships of one player are sunk

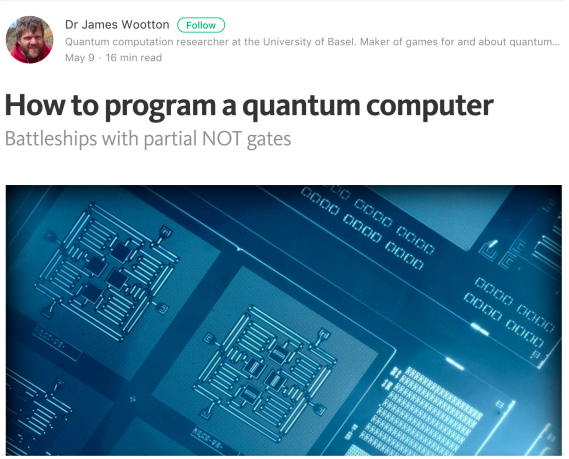
Code description and availability

A great description of the code by the developer of the code, Dr. James Wooton, is available at:

https://medium.com/@decodoku/how-to-program-a-quantum-computer-982a9329ed02

The code is also available:

https://github.com/decodoku/qiskit-sdk-py/tree/master/tutorial/sections



Demo: Battleships game on the IBM Q quantum computer

Player 1:
0, 1, 2

Player 2:
3, 4, 0

jupyter Battleships_with_partial_NOT_gates Last Checkpoint: Last Monday at 9:33 PM (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3

Markdown

In [1]:
```
import sys
sys.path.append("../../qiskit-sdk-py/")
from qiskit import QuantumProgram
import Qconfig
```

Next we import a few standard things.

In [2]:
```
import getpass, random, numpy, math
```

Now it's time for a title screen.

In [3]:
```
print("\n\n\n\n\n\n\n\n\n")
print("                                                                                ")
print("                                                                                ")
print("                                                                                ")
print("                                                                                ")
print("                                                                                ")
print("                                                                                ")
print("")
print("                                                                                ")
print("                                                                                ")
print("                                                                                ")
print("                                                                                ")
print("                                                                                ")
print("                                                                                ")
print("")
print("                    ___                ___                    _        _       ")
print("                   | _ )  _  _   |  \  __ __  ___ __| |  ___ | |__  _ _ ")
print("                   | _ \| || |  | |) |/ -_)/ _|/ _ \/ _` |/ _ \ / /| || |")
print("                   |___/ \_, |  |___/ \___|\__|\___/\__,_|\___/|_\_\ \_,_|")
print("                         |__/                                                   ")
print("                                 James Wootton, University of Basel")
print("")
```

Challenges for programming model developers

Lack of Quantum Computer Applications:

- IBM Quantum provides an initial good spectrum of applications to derive programming models requirements
- QPU ≅ GPU in some aspects, can we learn something from programming models for GPU?

A this early, difficult to understand **implication of intrinsic randomness of QPU**:

- Also classical bits have noise because they are just electrical signal …
- Not clear if randomness should emerge to programming system level and how it should be codified (different data types with different precision??)

Limitations

This was an introductory lecture on QPU.

We focus on understanding the concept of qubit and superposition and use it to implement very simple applications.

There are many important topics, like *entanglement*, that we didn't have time to cover but there are more example on how to program QPU to use advanced features of QPUs.

Conclusions

Small QPUs are accessible to everybody, but software stack and programming models are still in their infancy.

Lot of current HPC programming approaches match QPU programming and will necessarily impact it:

- CUDA/OpenCL for offloading to QPU
- Reconfigurable hardware (FPGA) programming
- …

We are just at the beginning of the programming models for QPU, **any work we done on this will drive their evolution!**

Thank you!

Hmmm no, we don’t have a DD2361 ***Applied QPU programming*** course at KTH

… at least not yet ☺