Programmer un Ordinateur Quantique

Arnaud RENARD

Centre de Calcul Régional ROMEO

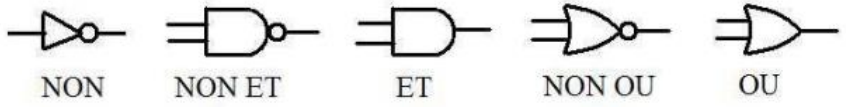
arnaud.renard@univ-reims.fr
http://romeo.univ-reims.fr

Informatique binaire

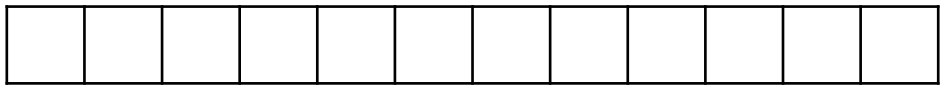
Information élémentaire : le bit

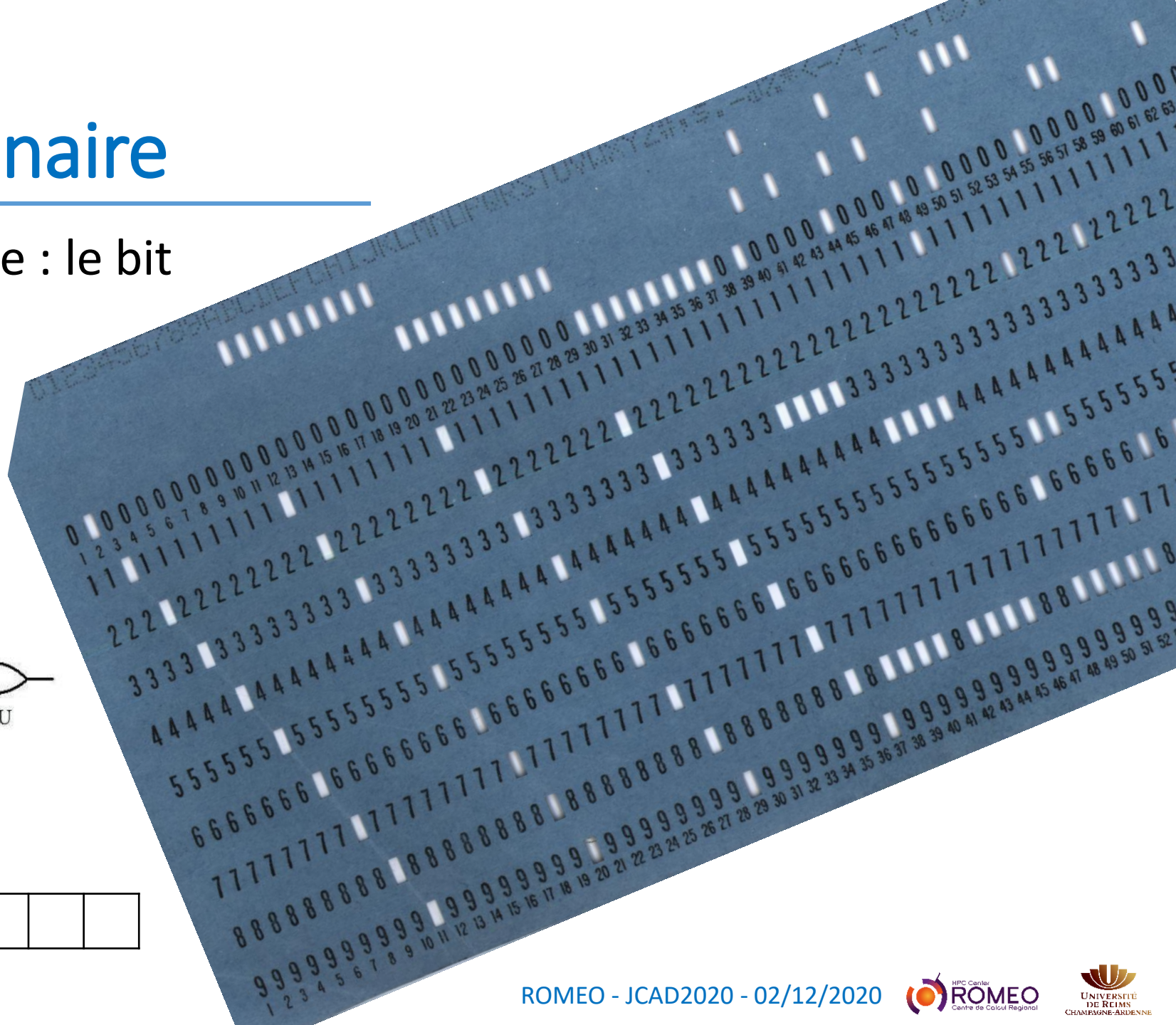
0 ou 1

- Adresses mémoire
- Opérations
- Portes logiques



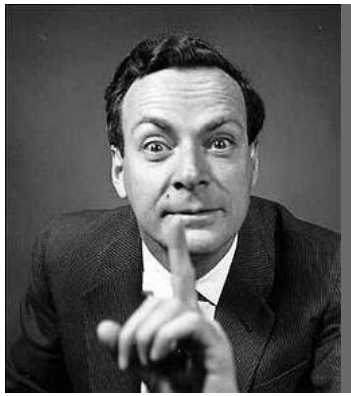
- Transistors
- Tableaux





Informatique quantique

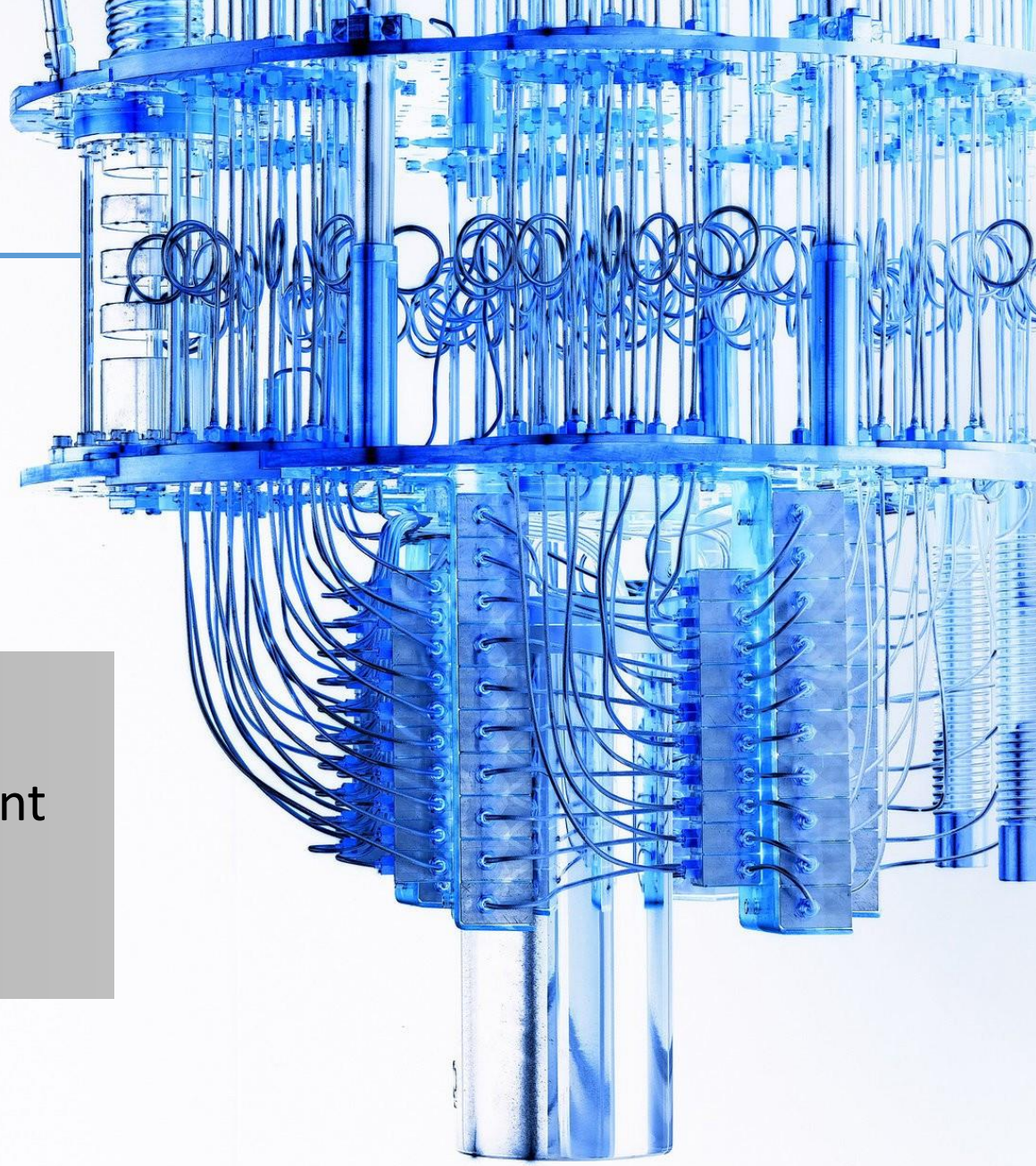
- Lois de la mécanique quantique



Disclaimer

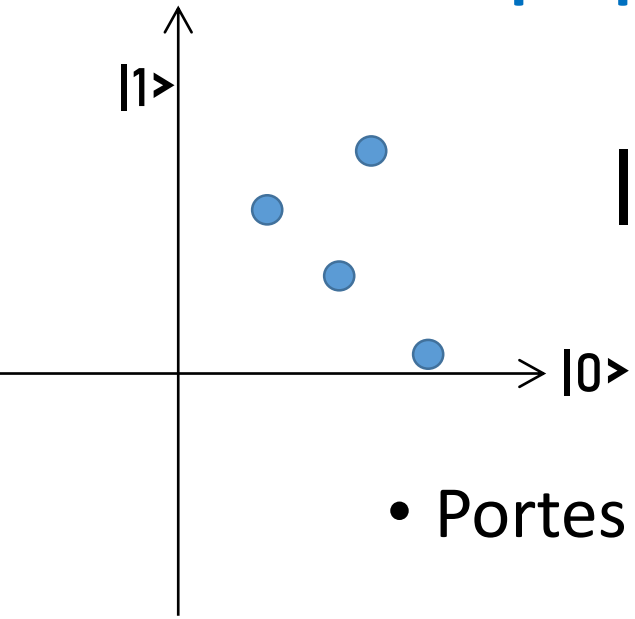
« **Personne ne comprend** vraiment la physique **quantique** »

[Richard Phillips Feynman 1918 - 1988]



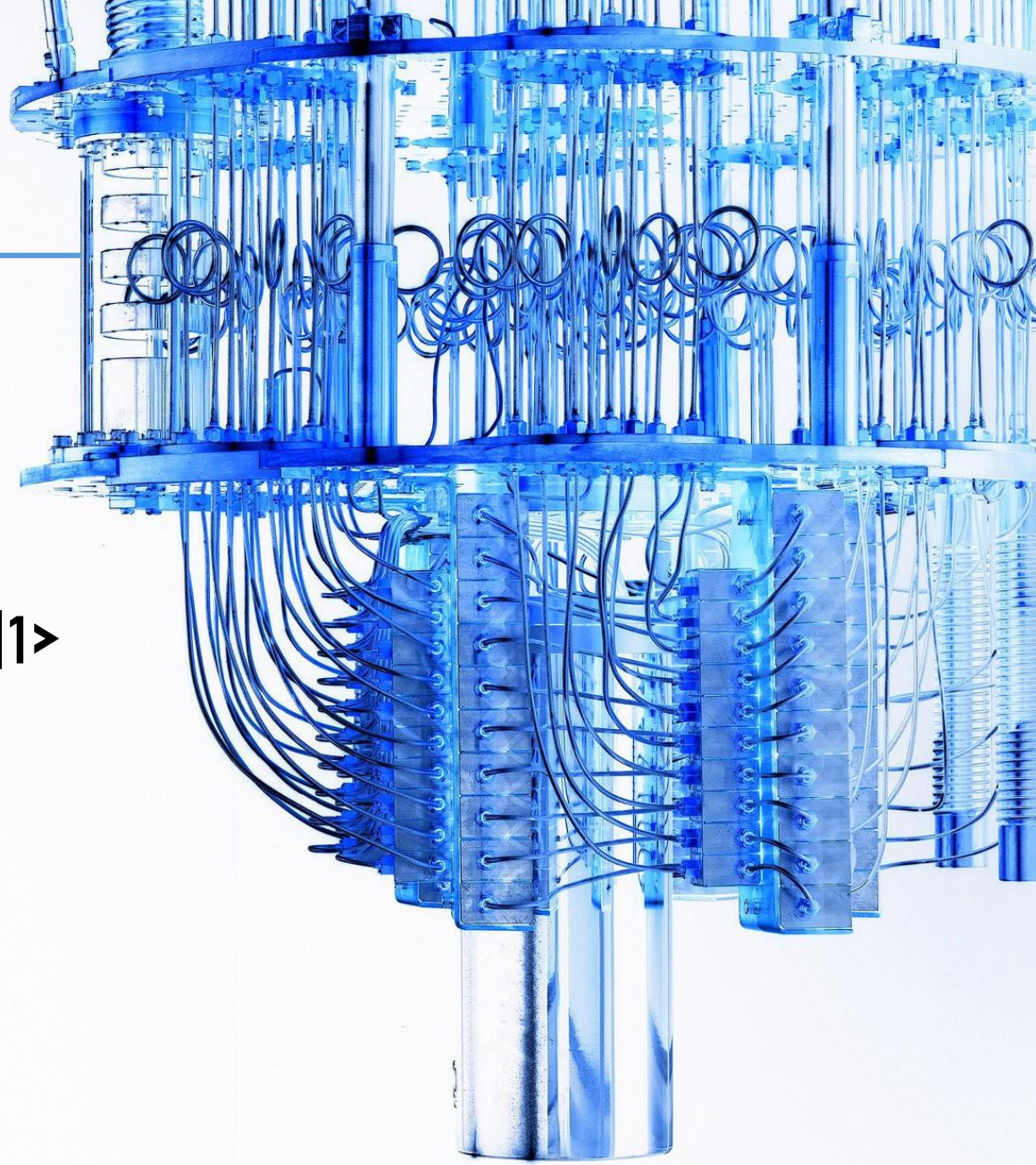
Informatique quantique

- Lois de la mécanique quantique
- Principe de superposition
- Information élémentaire : *qubit*
- **Superposition** des états |0> et |1>



$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle$$

- Portes quantiques



DEMO : ATOS QLM

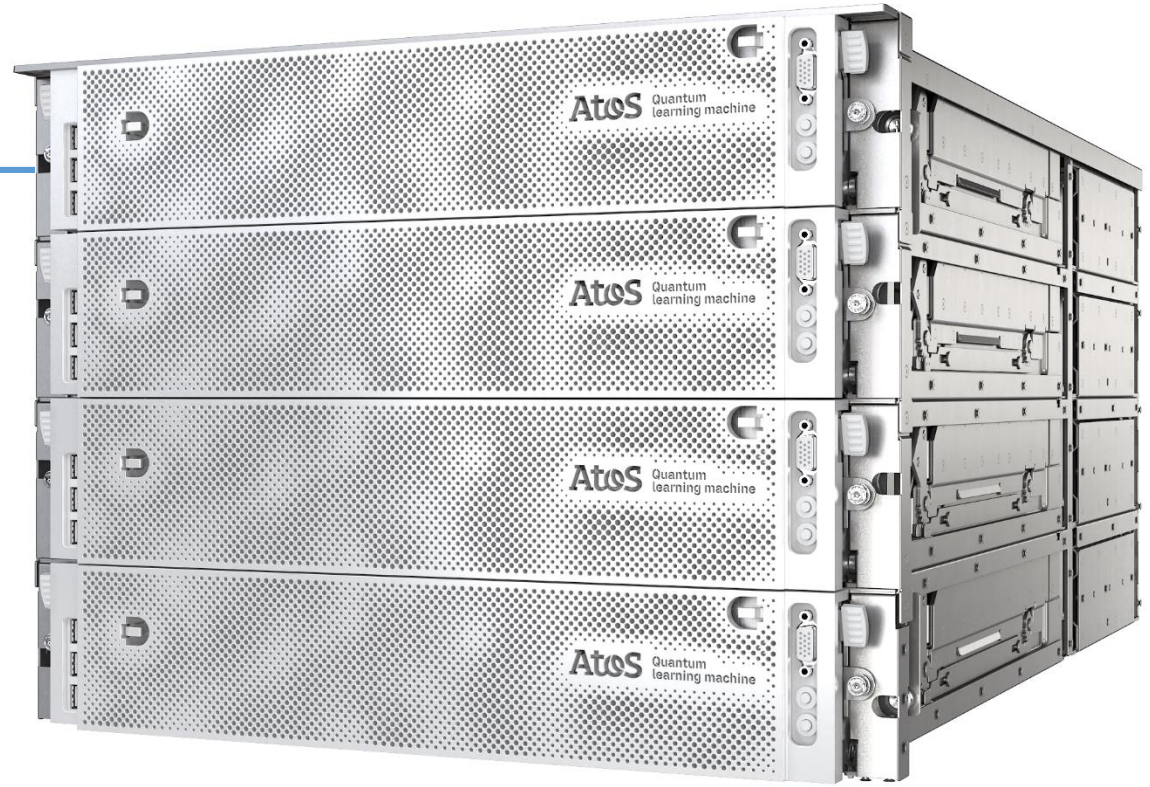
Quantum Learning Machine

ROMEO : 30 qubits

Max : 41 qubits

Environnement de programmation

Compilation et exécution



myQLM (gratuit) myQLM

QLaaS

Service / Formation

Qlul>
QLM User Club

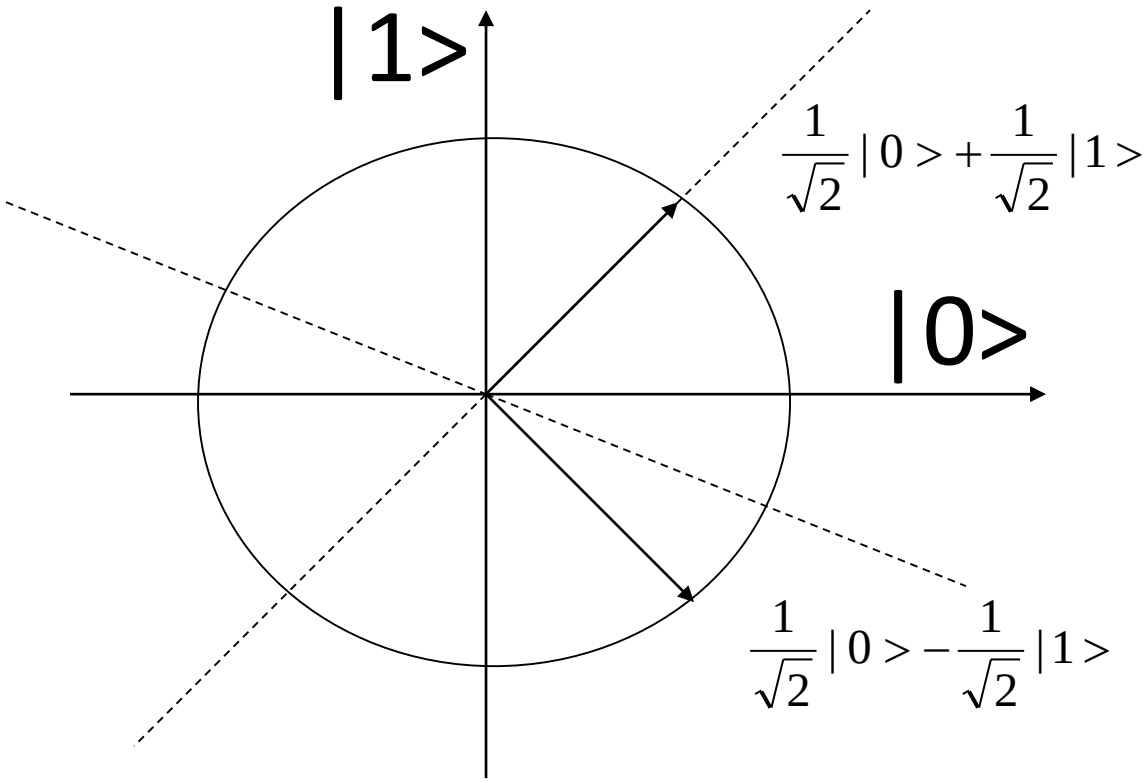
QLM User Club

- Intelligent Wave Inc. (Japon)
- CSC – IT (Finlande)
- C-DAC (Inde)
- Total (France)
- STFC Hartree Centre (Royaume-Uni)
- Argonne National Laboratory (USA)
- FH Upper Austria (Autriche)
- CEA – Commissariat à l'Energie Atomique et aux Energies Alternatives (France)
- Oak Ridge National Laboratory (USA)
- Université de Reims Champagne-Ardenne (France)
- *Leiden University (Neitherland)*
- *Irish Centre for High-End Computing – Xofia (Tx, USA)*

DEMO on ROMEO QLM

- Transformation Hadamar

$$\begin{cases} |0> \rightarrow \frac{1}{\sqrt{2}}|0> + \frac{1}{\sqrt{2}}|1> \\ |1> \rightarrow \frac{1}{\sqrt{2}}|0> - \frac{1}{\sqrt{2}}|1> \end{cases}$$



One qubits

Let us start with a simple circuit: the creation of a qubit and using some gates.

First, we need to import relevant objects from the python AQASM module:

In [1]:
```
from qat.lang.AQASM import Program, H, RX
import random, numpy
```

Creation of the quantum program

Then, we can declare a new object `Program`. Let us give it an explicit name:

In [2]:
```
oneq_prog = Program()
angles = [random.random() * 2. * numpy.pi for _ in range(2)]
```

Creation of our first qubit. Qbits are manipulated through qbit registers only (to keep things structured). Registers are allocated as follows:

In [3]:
```
qbits = oneq_prog.qalloc(1)
```

Now, we can access our qbits using the register "qbits".

Registers behave like python list/arrays.

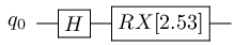
Here our qbits will be refered to using qbits[0].

We simply implement the appropriate 2-qbit rotation using a Hadamard gate (H) on the qbit:

In [13]:
```
#oneq_prog.apply(H, qbits[0])
oneq_prog.apply(RX(angles[0]), qbits[0])
```

The corresponding circuit object can be extracted directly from the Program object as follows:

In [14]:
```
circuit = oneq_prog.to_circ()
%qatdisplay circuit
```



Simulation of the execution of the circuit

Now that we have a proper circuit, we can try and simulate it:

In [15]:
```
#Let us import some qpu connected to a classical linear algebra simulator
from qat.qpus import LinAlg
qpu = LinAlg()

job = circuit.to_job()

result = qpu.submit(job)
for sample in result.raw_data:
```

```
angles = [random.random() * 2
```

Markdown

Creation of our first qubit. Qbits are manipulated through qbit registers only (to keep things structured). Registers are allocated as follows:

In [3]:
```
qbits = oneq_prog.qalloc(1)
```

Now, we can access our qbits using the register "qbits".

Registers behave like python list/arrays.

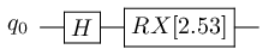
Here our qbits will be refered to using qbits[0].

We simply implement the appropriate 2-qbit rotation using a Hadamard gate (H) on the qbit:

In [13]:
```
#oneq_prog.apply(H, qbits[0])
oneq_prog.apply(RX(angles[0]), qbits[0])
```

The corresponding circuit object can be extracted directly from the Program object as follows:

In [14]:
```
circuit = oneq_prog.to_circ()
%qatdisplay circuit
```

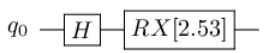


Simulation of the execution of the circuit

Now that we have a proper circuit, we can try and simulate it:

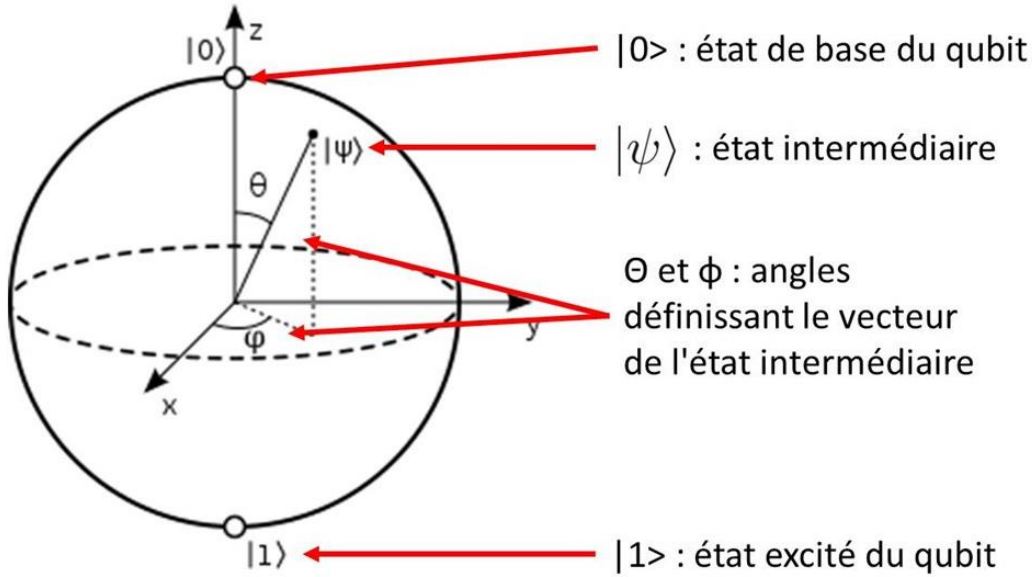
In [15]:
```
#Let us import some qpu connected to a classical linear algebra simulator
from qat.qpus import LinAlg
qpu = LinAlg()

job = circuit.to_job()

result = qpu.submit(job)
for sample in result.raw_data:
    print("State", sample.state, "with amplitude " , sample.amplitude)

circuit = oneq_prog.to_circ()
%qatdisplay circuit
```

```
State |1> with amplitude  (0.21233910667607528-0.6744717219987849j)
State |0> with amplitude  (0.21233910667607528-0.6744717219987849j)
```



Un qubit

- $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$
  - a et b $\epsilon\ \mathbb{C}$
  - $a^2 + b^2 = 1$
  - Sphère de Bloch



|0> : état de base du qubit

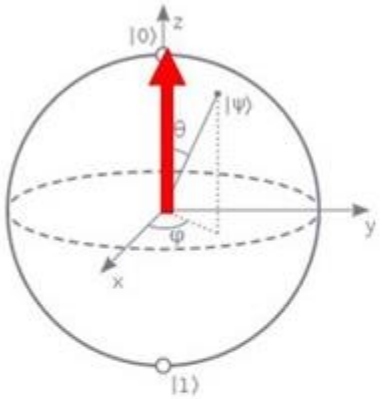
$|\psi\rangle$: état intermédiaire

Θ et ϕ : angles définissant le vecteur de l'état intermédiaire

|1> : état excité du qubit

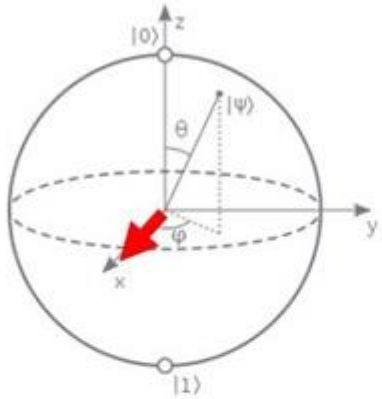
Cycle de vie d'un qubit

initialisation à 0



$|0>$

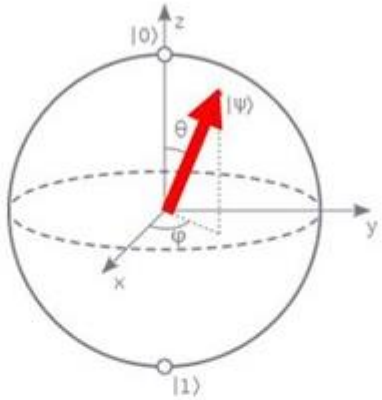
porte de Hadamard



$$\frac{|0> + |1>}{\sqrt{2}}$$

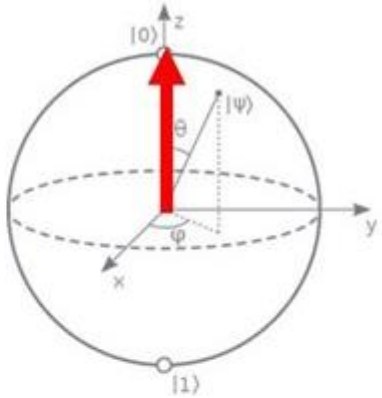
superposition de 0 et 1

autres portes

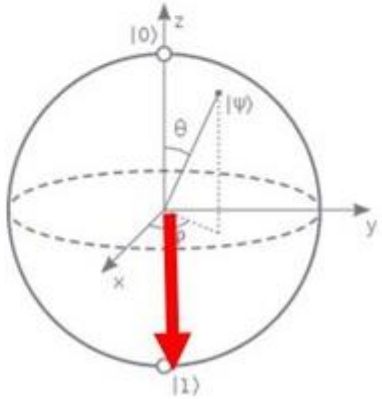


elles vont continuer à faire tourner le vecteur dans la sphère de Bloch

mesure

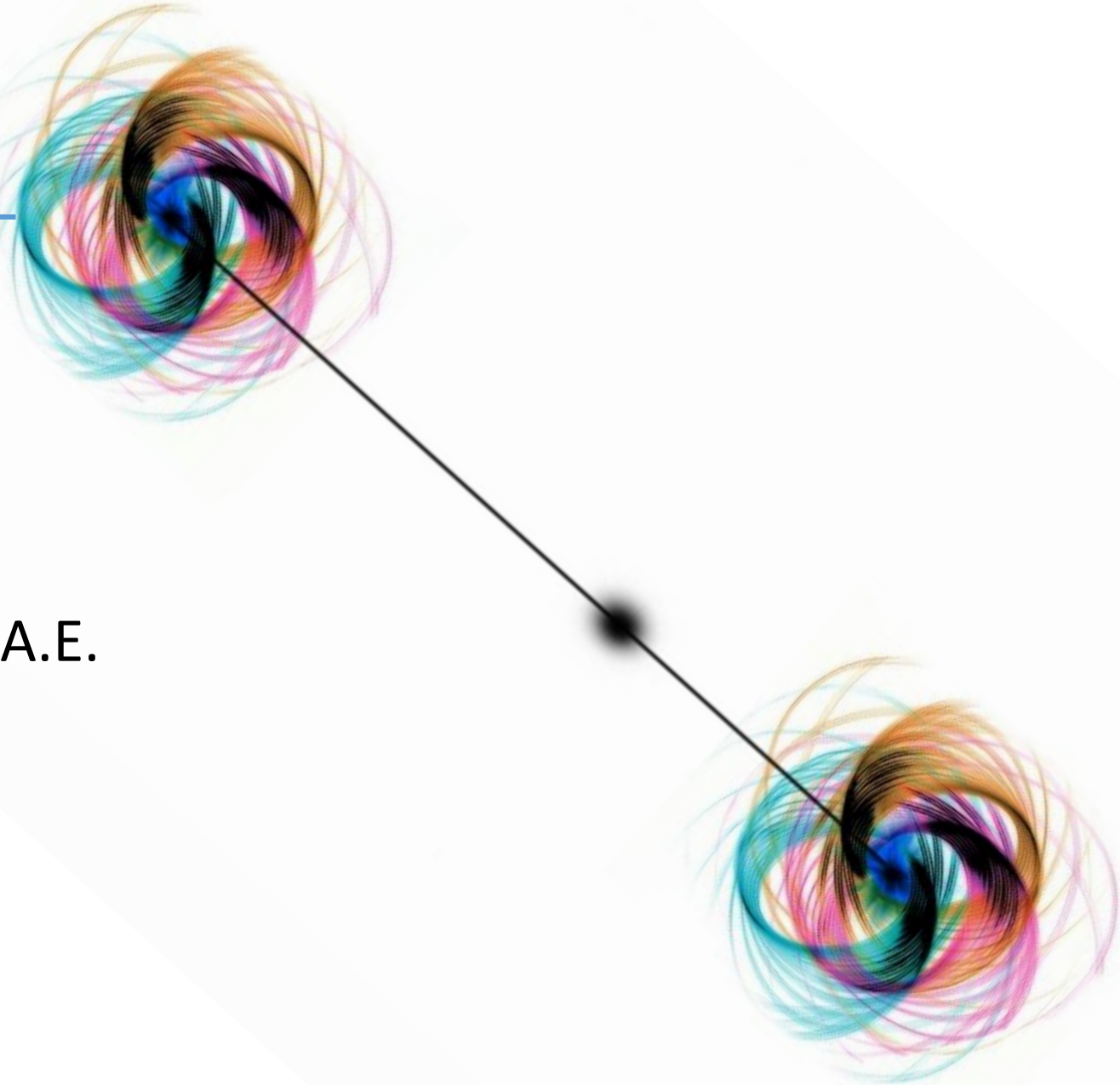


la mesure va retourner un |0> avec une probabilité alpha au carré dépendant de l'état évalué



la mesure va retourner un |1> avec une probabilité beta au carré dépendant de l'état évalué

Intrication quantique



Lois de la mécanique quantique

« d’action fantôme à distance » A.E.

Quantum entanglement

DEMO EPR

- **Einstein-Podolsky-Rosen**
- deux particules sont émises
- une relation de conservation = intrication quantique
- La mesure de l’une nous informe de l’état de l’autre

Creation of an EPR pair using two qubits

Let us start with a simple circuit: the creation of an EPR pair using two qubits.

First, we need to import relevant objects from the python AQASM module:

In [1]:
```
from qat.lang.AQASM import Program, H, CNOT
```

Creation of the quantum program

Then, we can declare a new object `Program`. Let us give it an explicit name:

In [2]:
```
epr_prog = Program()
```

To create our EPR pair, we need to manipulate two qbits. Qbits are manipulated through qbit registers only (to keep things structured). Registers are allocated as follows:

In [3]:
```
qbits = epr_prog.qalloc(2)
```

Now, we can access our qbits using the register "qbits".

Registers behave like python list/arrays.

Here our qbits will be refered to using qbits[0] and qbits[1].

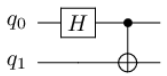
To create our EPR pair, we simply implement the appropriate 2-qbit rotation using a Hadamard gate (H) on the first qbit, followed by a controlled NOT gate (CNOT) on both qbits:

In [4]:
```
epr_prog.apply(H, qbits[0])
epr_prog.apply(CNOT, qbits)
```

Notice that since the CNOT is applied on both qbits (it is a 2-qbit gate), we can pass the whole register as argument to the `.apply` method.

The corresponding circuit object can be extracted directly from the Program object as follows:

In [5]:
```
circuit = epr_prog.to_circ()
%qatdisplay circuit
```



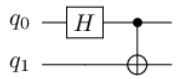
Simulation of the execution of the circuit

Now that we have a proper circuit, we can try and simulate it:

In [6]:
```
#Let us import some qpu connected to a classical linear algebra simulator
from qat.qpus import LinAlg
qpu = LinAlg()
```

%qatdisplay circuit

Code



Simulation of the execution of the circuit

Now that we have a proper circuit, we can try and simulate it:

In [6]:
```
#Let us import some qpu connected to a classical linear algebra simulator
from qat.qpus import LinAlg
qpu = LinAlg()

job = circuit.to_job()

result = qpu.submit(job)
for sample in result.raw_data:
    print(f"State {sample.state} with probability " , (sample.amplitude) )
```

```
State |00> with probability  (0.7071067811865475+0j)
State |11> with probability  (0.7071067811865475+0j)
```

Export to Atos Quantum Assembly Language (AQASM) format

We can also export our circuit in the AQASM format:

In [7]:
```
epr_prog.export("helloworld.aqasm")
```

The generated file *helloworld.aqasm* should look like this:

In [8]:
```
!cat helloworld.aqasm
```

```
BEGIN
qubits 2
cbits 2

H q[0]
CNOT q[0],q[1]
END
```

and can be compiled to circ format as follows:

In [9]:
```
!aqasm2circ helloworld.aqasm
```

In [ ]:

More and more qubits

1 qubit = superposition de 2 états |0> et |1>

1 opération = 

2 qubit : 4 états |00>, |01>, |10> et |11>

4 qubit : 16 états 

10 qubits : 1024 états (2**10)

20 qubits

43 qubits

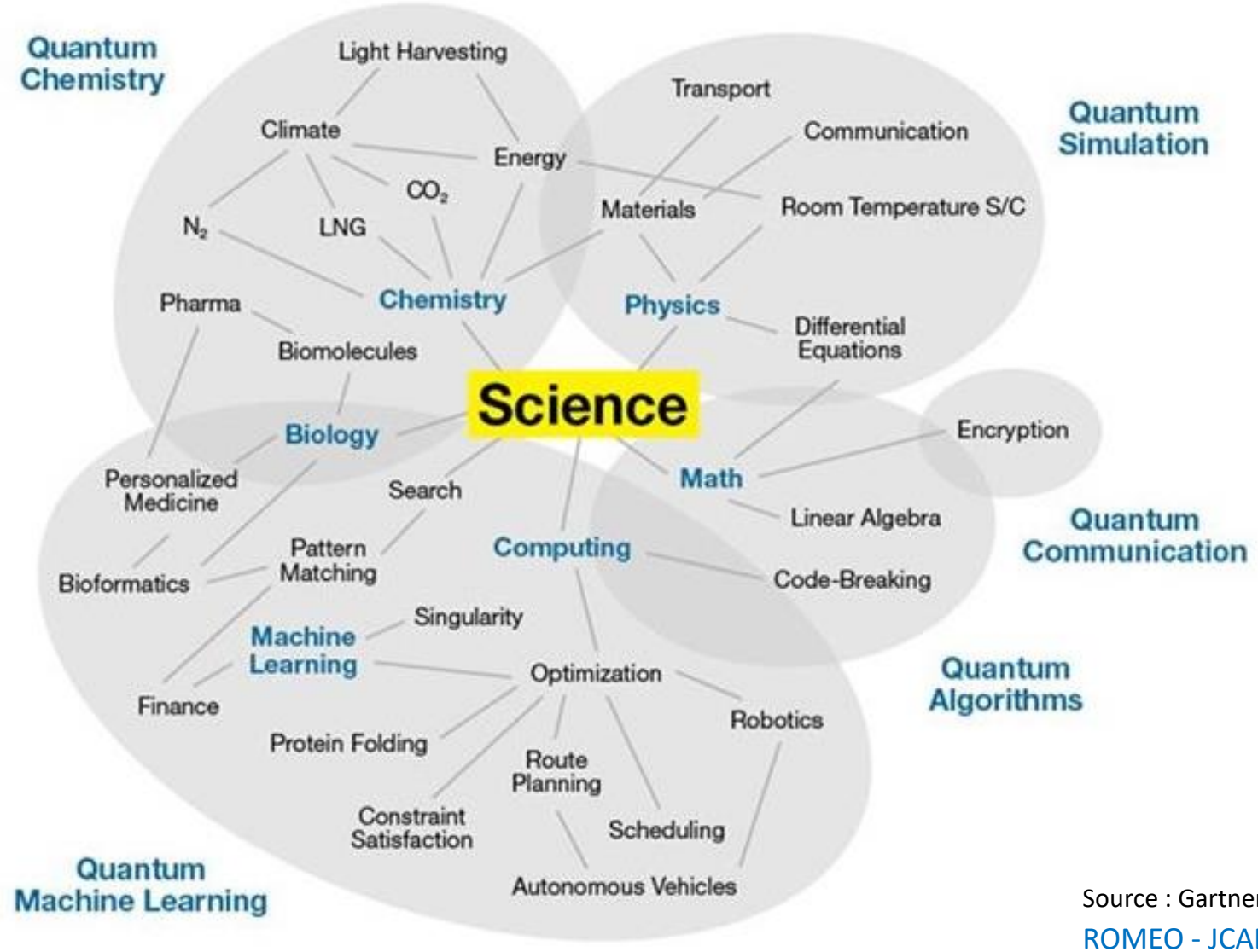
50 qubits

PASQAL

100 qubits

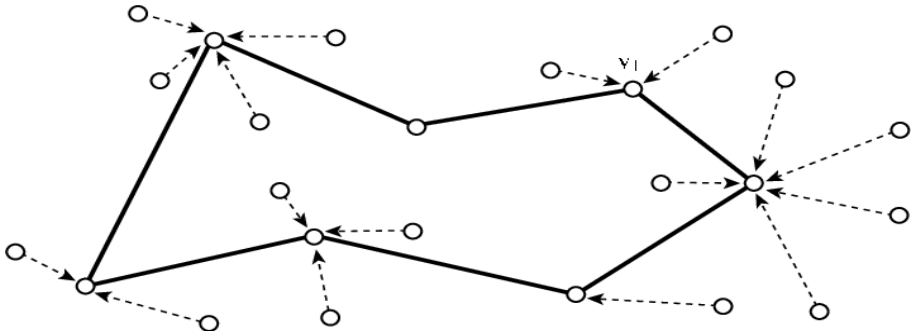
suprématie quantique

Quantum Computing – Use Cases

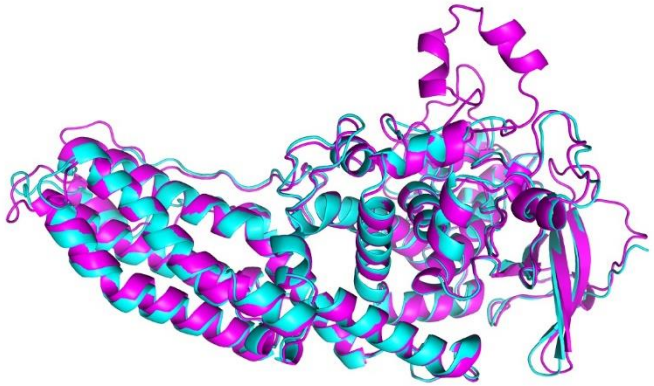


Source : Gartner.com/SmarterWithGartner

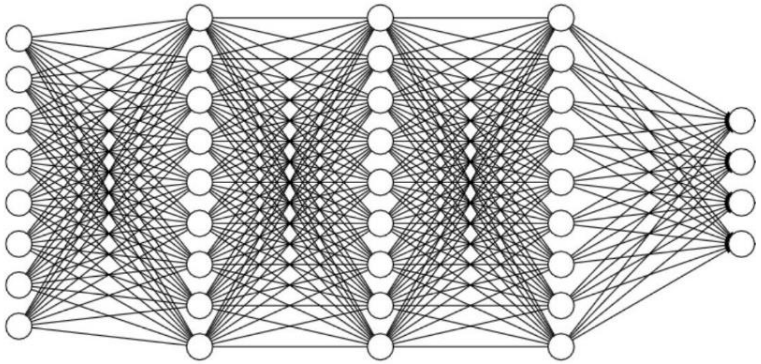
Problèmes complexes exponentiels



Optimisations combinatoires
trajets, placements, cartes



Simulations moléculaires
matériaux et biologie



Entrainement
de réseaux de neurones

```
3164702693302559231434537239493375160541061884752646
4414030417673281124749306936869204318512161183785672
6816539985465097356123432645179673853590577238179357
9008764261039437823764945917429345884971175871469169
7298476115906087325093946208557574075457709862055801
1779529884042198287643319330465064455234988142139565
7854474740235463537585373248018381203876008684165254
0079038128588825668708585545623157752793930592081176
6585308670132129155221804381548625787943020694528015
```

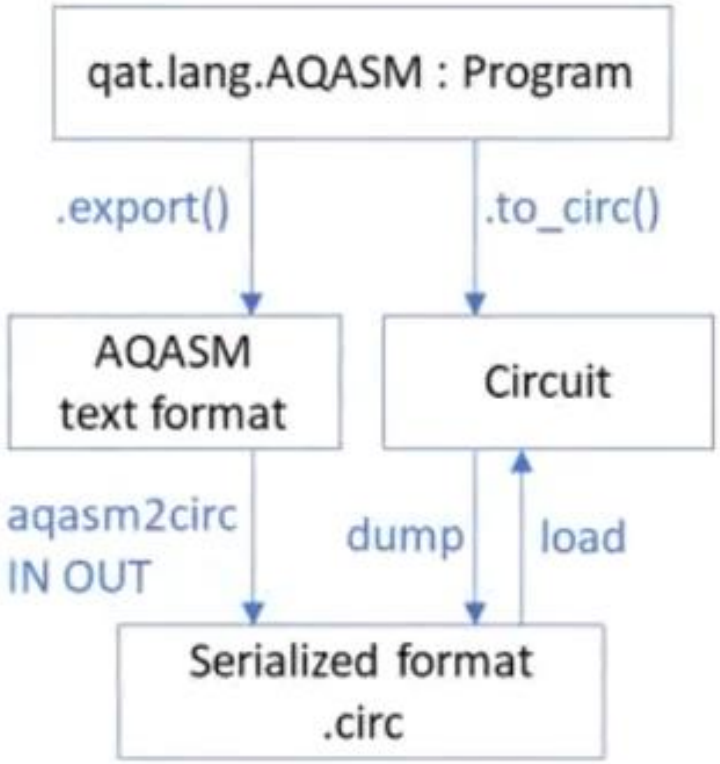
factorisation
de très grands nombres entiers

AQASM / pyAQASM

- Programme python (hybride)
- AQASM : a Quantum Assembleur
- Format circ

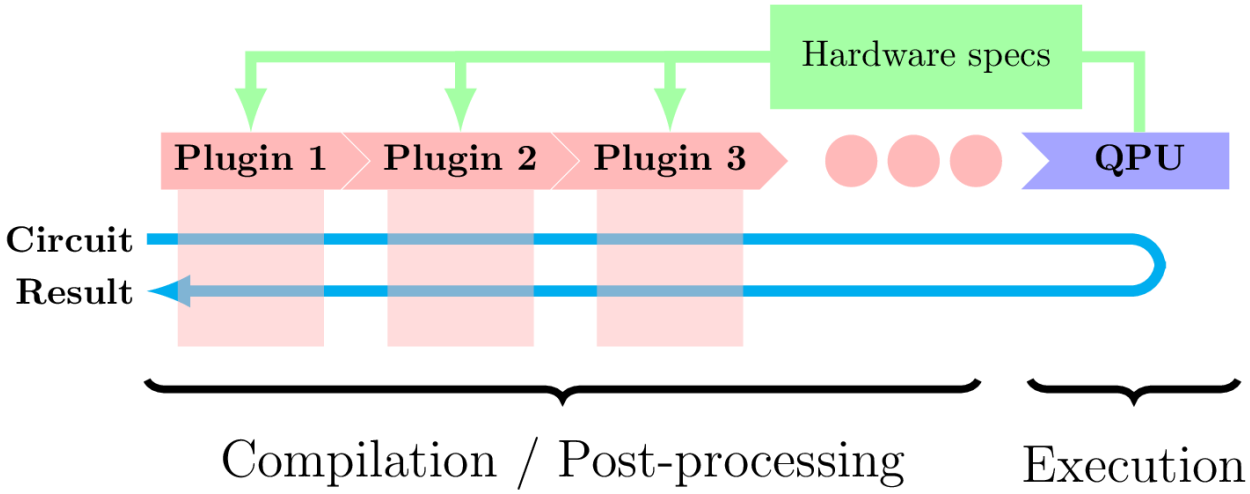
```
epr_prog = Program()
qbits = epr_prog.qalloc(2)

epr_prog.apply(H, qbits[0])
epr_prog.apply(CNOT, qbits)
```

```
BEGIN
qubits 2
cbits 2

H q[0]
CNOT q[0],q[1]
END
```



Simulation d'exécution d'un circuit

Circuit
|- Job
    |- **execution stack**
        Plugin1() | Plugin2() | Plugin3() | PyLinalg()



Un QPU peut être
- Un simulateur idéal
- Un simulateur bruité
- Ou l'interface vers un processeur quantique

```
from qat.core.qpu import DistantQPU

# Define PORT and IP
PORT = 1765
IP = "129.42.38.10"

# Define a client
qpu = DistantQPU(PORT,IP)
```

Remerciements / ressources

Xavier Geoffret,HPC & Quantum Global Presales, ATOS

Comprendre l’informatique quantique
édition 2020

Olivier Ezratty

https://quantumalgorithmzoo.org/