# Practica 1

March 14, 2021

## 1 EJERCICIO 1

En este ejercicio comento similitudes y diferencias encontradas entre los distintos entornos de programación proporcionados

### 1.1 Codigo para la generacion de numeros aleatorios

#### 1.1.1 Qiskit

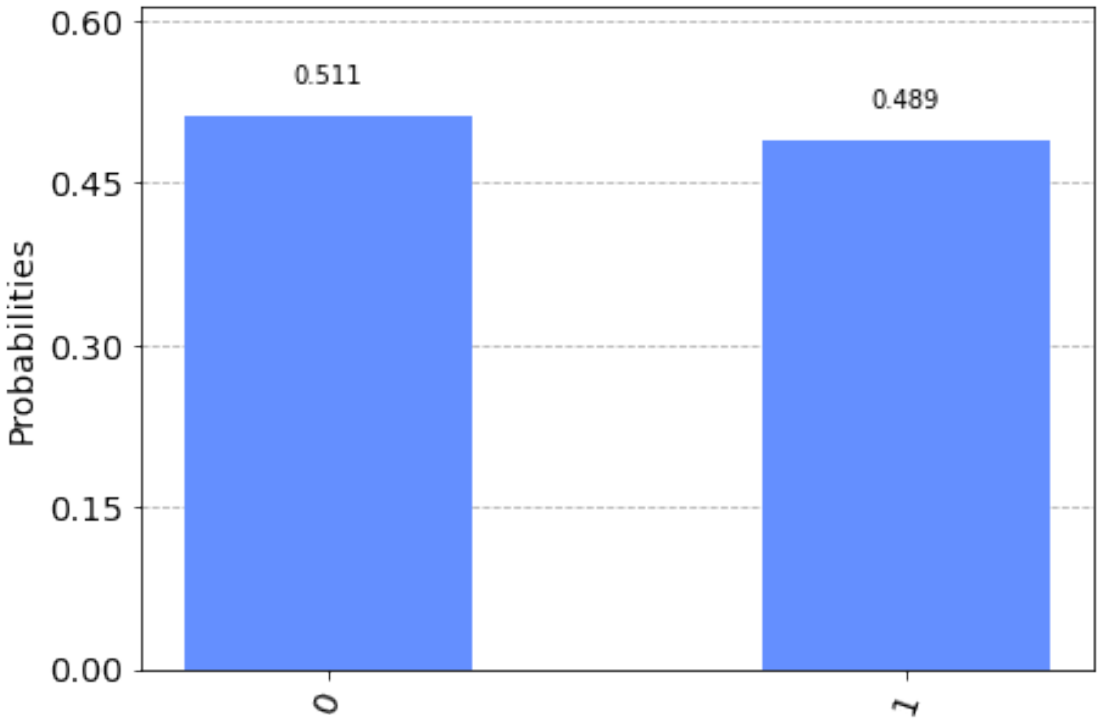
Este entorno es mucho mas visual, aprte de tener una gran similitud con el resto de entornos, existe la posibilidad de dibujar el circuito (aunque la manera en la que he creado la memoria no me permitia usar muchos de los caracteres necesarios para crear la imagen por lo que no aparecera en la misma) y de ves el histograma como barras ,cada una con su propio valor y porcentaje

[9]:
```
import numpy as np
from qiskit import(
  QuantumCircuit,
  execute,
  Aer)
from qiskit.visualization import plot_histogram

simulator = Aer.get_backend('qasm_simulator')
circuit = QuantumCircuit(1,1)
circuit.h(0)
circuit.measure([0],[0])
job = execute(circuit, simulator, shots=1000)
result = job.result()
counts = result.get_counts(circuit)
print("\nNúmero de ocurrencias para 0 y 1:", counts)
```

```
Número de ocurrencias para 0 y 1: {'0': 511, '1': 489}
```

[10]:
```
plot_histogram(counts)
```

[10]:




### 1.1.2 Forest

Las diferencias entre este entorno y el resto es la necesidad de comprovar si se puede utilizar qvm y quilic antes de ejecutar el programa y que no hace falta añadir una puerta de medicion en el circuito ademas de existir una funcion que ejecuta y mide el circuito al mismo tiempo

[22]:
```
import numpy as np
from pyquil import Program, get_qc
from pyquil.gates import *
from pyquil.api import local_forest_runtime

# construir la puerta Hadamard para el circuito
p = Program(H(0))

#check make sure of qvm and quilc availability and run the program on a QVM
with local_forest_runtime():
    qc = get_qc('9q-square-qvm')
    result = qc.run_and_measure(p, trials=1000)

unique, counts = np.unique(result[0], return_counts=True)
print(unique, counts)
```

```
[0 1] [501 499]
```

### 1.1.3 Cirq

En este entorno existe la necesidad de nombrar al qbit para luego devolver su histograma (lo que no pasa en otros entornos). Tambien existe la posibilidad de darle al circuito un Amplitude Damping Channel con una provavilidad dada gamma

[20]:
```
import cirq

q = cirq.NamedQubit('a')
circuit = cirq.Circuit(cirq.H(q), cirq.amplitude_damp(0.2)(q), cirq.measure(q))
simulator = cirq.DensityMatrixSimulator()
result = simulator.run(circuit, repetitions=1000)
print(result.histogram(key='a'))
```

```
Counter({0: 613, 1: 387})
```

### 1.1.4 tket

Este entorno tiene muchas similitudes con qiskit y ademas utilizo el backend utilizado en ese mismo entorno (Aer) , las diferencias notorias son la existencia de una funcion para medir todas las puertas que haya y que para poder ejecutar dicho circuito hay que compilarlo antes para satisfacer al backend

[35]:
```
from pytket import Circuit
from pytket.backends.ibm import AerBackend
c = Circuit(1,1) # definir el circuito cin 1 qubit y 1 bit
c.H(0)           # añadir la Hadamerd al qbit
c.measure_all()  # medir todos los qbits(1 en este caso)

b = AerBackend()                # conectar al backend
b.compile_circuit(c)            # compilar el circuito para satisfacer las␣
 ↪condiciones del backend
handle = b.process_circuit(c, n_shots = 1000)  # ejecutar 100 veces
counts = b.get_result(handle).get_counts()   # recuperar los resultados
print(counts)
```

```
Counter({(1,): 506, (0,): 494})
```

### 1.1.5 Projectq

Este entorno tiene una gran diferencia con el resto ya que la forma en la que se ponen las puertas es pasando el qbit por dicha puerta , en vez de colocarla en el camino que tomara el qbit y despues de pasarlas y medirlas te da el resultado del qbit

[19]:
```
from projectq import MainEngine  # Se importa el compilador principal
from projectq.ops import H, Measure  # se importan las operaciones a realizar␣
 ↪(Hadamard y Medicion)

eng = MainEngine()  # crear un compildor
```

```
qubit = eng.allocate_qubit()  # se asigna el qbit

for num in range(100):
    H | qubit  # se aplica la puerta Hadamard
    Measure | qubit  # se mide el qbit resultante
    eng.flush()  # se liberan las puertas y se mide
    print("Measured {}".format(int(qubit)))  # resultado
```

```
Measured 1, Measured 0, Measured 0, Measured 0, Measured 0, Measured 0, Measured
1, Measured 1, Measured 1, Measured 0
Measured 0, Measured 1, Measured 1, Measured 0, Measured 0, Measured 0, Measured
0, Measured 1, Measured 0, Measured 0
Measured 1, Measured 0, Measured 1, Measured 0, Measured 0, Measured 0, Measured
0, Measured 0, Measured 1, Measured 0
Measured 1, Measured 1, Measured 1, Measured 1, Measured 1, Measured 1, Measured
1, Measured 1, Measured 1, Measured 1
Measured 1, Measured 0, Measured 1, Measured 1, Measured 1, Measured 1, Measured
0, Measured 1, Measured 0, Measured 0
Measured 1, Measured 0, Measured 1, Measured 1, Measured 1, Measured 0, Measured
0, Measured 1, Measured 1, Measured 1
Measured 0, Measured 1, Measured 1, Measured 1, Measured 1, Measured 1, Measured
1, Measured 0, Measured 1, Measured 1
Measured 0, Measured 0, Measured 1, Measured 0, Measured 1, Measured 1, Measured
0, Measured 0, Measured 0, Measured 1
Measured 1, Measured 0, Measured 0, Measured 0, Measured 0, Measured 1, Measured
0, Measured 0, Measured 1, Measured 0
Measured 0, Measured 1, Measured 1, Measured 0, Measured 0, Measured 0, Measured
1, Measured 0, Measured 0, Measured 0
```

# 2 EJERCICIO 2

## 2.1 Codigo para la generacion de los estados de Bell

[ ]:
```
import numpy as np
from qiskit import(
  QuantumCircuit,
  execute,
  Aer)
from qiskit.visualization import plot_histogram, plot_state_city

# Creamos un objeto Quantum Circuit que actúa sobre el registro cuántico por␣
 ↪defecto (q)
# de un bit (primer parámetro) y que tiene un registro clásico de un bit␣
 ↪(segundo parámetro)
circuit = QuantumCircuit(2,2)
# Añadimos una puerta Hadamard con el qubit q_0 como entrada
circuit.h(0)
```

```
circuit.cnot(0, 1)
# Mapeamos la medida de los qubits (primer parámetro) sobre los bits clásicos
circuit.measure([0,1], [0,1])
# Dibujamos el circuito
circuit.draw()
```

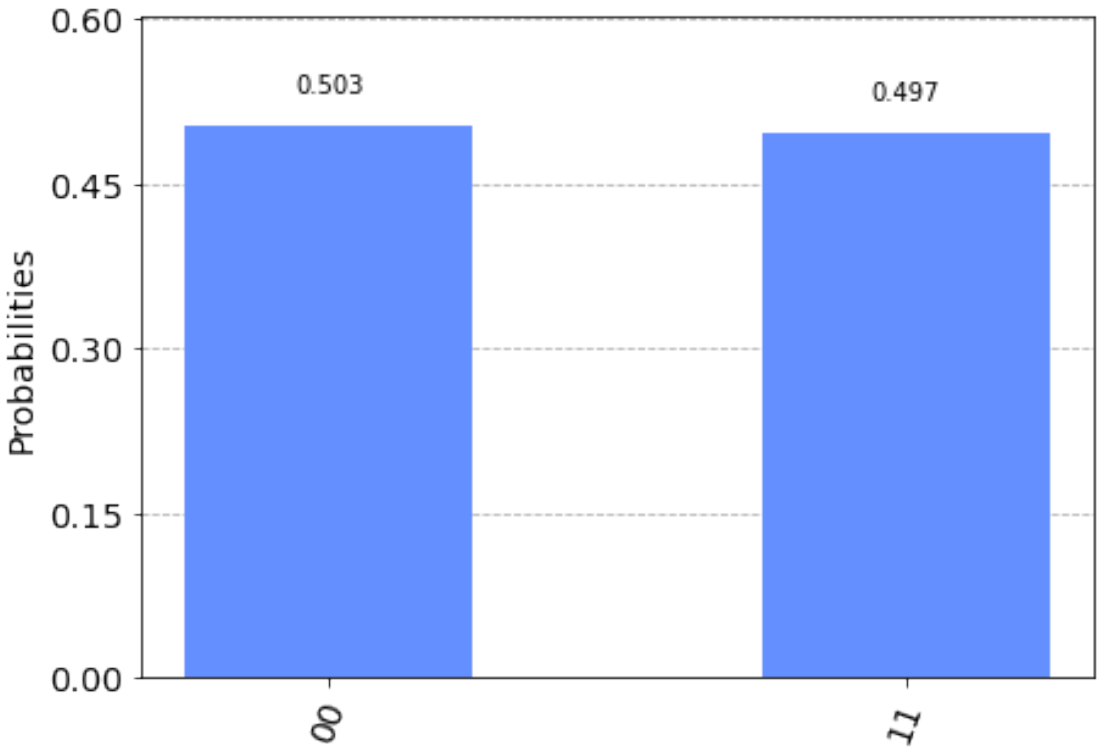
### 2.1.1 QASM_SIMULATOR

Este simulador se puede ver tanto la cantidad de resultados obtenidos como que valor tenia cada uno de ello y todos estos valores son facilmente representables en un histograma que nos permite observar mas facilmente los resultados y su relacion entre ellos para su facil comparación.

[15]:
```
# Usamos el qasm_simulator de Aer
simulator_qasm = Aer.get_backend('qasm_simulator')
# Ejecutamos el circuito sobre el simulador qasm
job_qasm = execute(circuit, simulator_qasm, shots=1000)
# Almacenamos los resultados
result_qasm = job_qasm.result()
# Capturamos las ocurrencias de salida
counts_qasm = result_qasm.get_counts(circuit)
# Escribimos el número de ocurrencias
print("\nNúmero de ocurrencias:",counts_qasm)
plot_histogram(counts_qasm)
```

```
Número de ocurrencias: {'00': 503, '11': 497}
```

[15]:

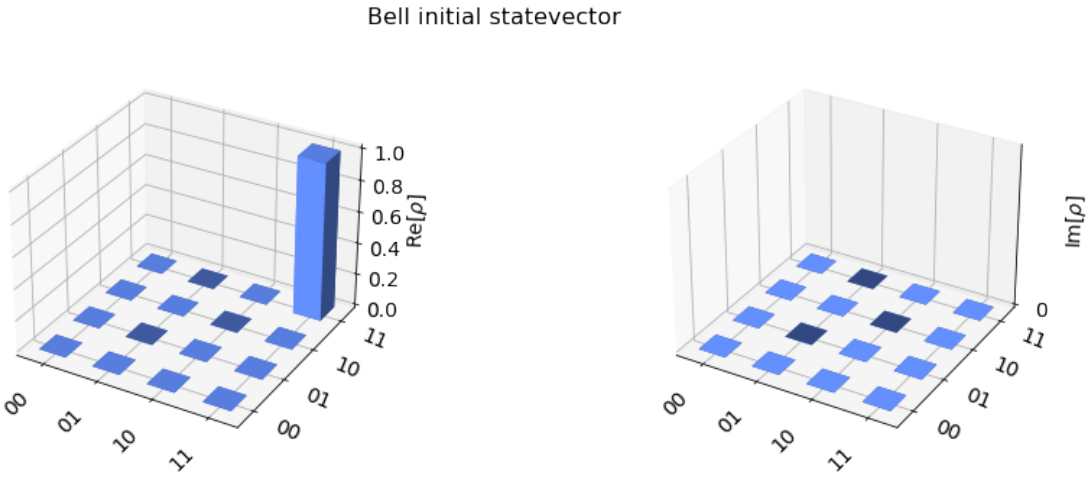



### 2.1.2 STATEVECTOR_SIMULATOR

El statevector_simulator nos permite ver el resultado de una de las soluciones a las que se ha llegado en las 1000 iteraciones sin poder ver su frecuencia en el total de resultados.Ademas este simulador nos presenta una forma tridimensional de representar la solucion con la funcion plot_state_city.

[17]:
```
simulator_statevector = Aer.get_backend('statevector_simulator')
# Ejecutamos el circuito sobre el simulador qasm
job_statevector = execute(circuit, simulator_statevector, shots=1000)
# Almacenamos los resultados
result_statevector = job_statevector.result()
# Capturamos las ocurrencias de salida
counts_statevector = result_statevector.get_counts(circuit)
# Escribimos el número de ocurrencias
print("\nNúmero de ocurrencias:",counts_statevector)
plot_state_city(result_statevector.get_statevector(circuit), title="Bell initial␣
↪statevector")
```

```
Número de ocurrencias: {'11': 1}
```
[17]:




### 2.1.3 UNITARY_SIMULATOR

El ultimo simulador a provar es el Unitary_simulator que como el anterior solo nos da un unico resultado en las mil ejecuciones y tampoco nos devuelve su frecuancia.Tambien observamos que la forma de visualizar este resultado es bastante diferente al resto, siendo con la funcion get_unitary que devuelve el la matriz unitaria final del experimento

[11]:
```
simulator_unitary = Aer.get_backend('unitary_simulator')

circ = QuantumCircuit(2,2)
circ.h(0)
circ.cx(0, 1)

result = execute(circ, simulator_unitary, shots= 1000).result()
unitary = result.get_counts(circ)
print("Circuit unitary:\n", unitary,"\n")
result_unitary = execute(circ, simulator_unitary, shots=1000).result()
unitary = result_unitary.get_unitary(circ)
print("Bell states unitary:\n", unitary)
```

```
Circuit unitary:
 {'00': 1}

Bell states unitary:
 [[ 0.70710678+0.00000000e+00j  0.70710678-8.65956056e-17j
   0.        +0.00000000e+00j  0.        +0.00000000e+00j]
 [ 0.        +0.00000000e+00j  0.        +0.00000000e+00j
   0.70710678+0.00000000e+00j -0.70710678+8.65956056e-17j]
 [ 0.        +0.00000000e+00j  0.        +0.00000000e+00j
   0.70710678+0.00000000e+00j  0.70710678-8.65956056e-17j]
```

```
 [ 0.70710678+0.00000000e+00j -0.70710678+8.65956056e-17j
   0.        +0.00000000e+00j  0.        +0.00000000e+00j]]
```

### 2.1.4 Observaciones

En defenitiva todos los simuladores utilizan las mismas operaciones para calcular los resultados obtenidos en una ejecucion, pero cada uno tiene una manera diferente de visualizar dichos resultados.

# 3 EJERCICIO 3 y 4

## 3.1 Estados GHZ

Para la generacion de los estados ghz añadimos una cnot controlada a los estados de bell con el qbit 1 como qbit de control y el 2 como qbit en el que se realizará la operación

## 3.2 Condiciones Idóneas (No Reales)

En esta parte observaremos como sin presencia de ruido(condiciones idoneas) el circuito se comportara siempre de la manera que esta prevista.

[3]:
```
import numpy as np
from qiskit import(
  QuantumCircuit,
  execute,
  Aer)
from qiskit.visualization import plot_histogram

# Usamos el qasm_simulator de Aer
simulator1 = Aer.get_backend('qasm_simulator')
# Creamos un objeto Quantum Circuit que actúa sobre el registro cuántico por␣
 ↪defecto (q)
# de un bit (primer parámetro) y que tiene un registro clásico de un bit␣
 ↪(segundo parámetro)
circuit = QuantumCircuit(3, 3)
# Añadimos una puerta Hadamard con el qubit q_0 como entrada
circuit.h(0)
circuit.cnot(0, 1)
circuit.cnot(1, 2)
# Mapeamos la medida de los qubits (primer parámetro) sobre los bits clásicos
circuit.measure([0,1,2], [0,1,2])


# Ejecutamos el circuito sobre el simulador qasm
job = execute(circuit, simulator1, shots=1000)
# Almacenamos los resultados
result = job.result()
```

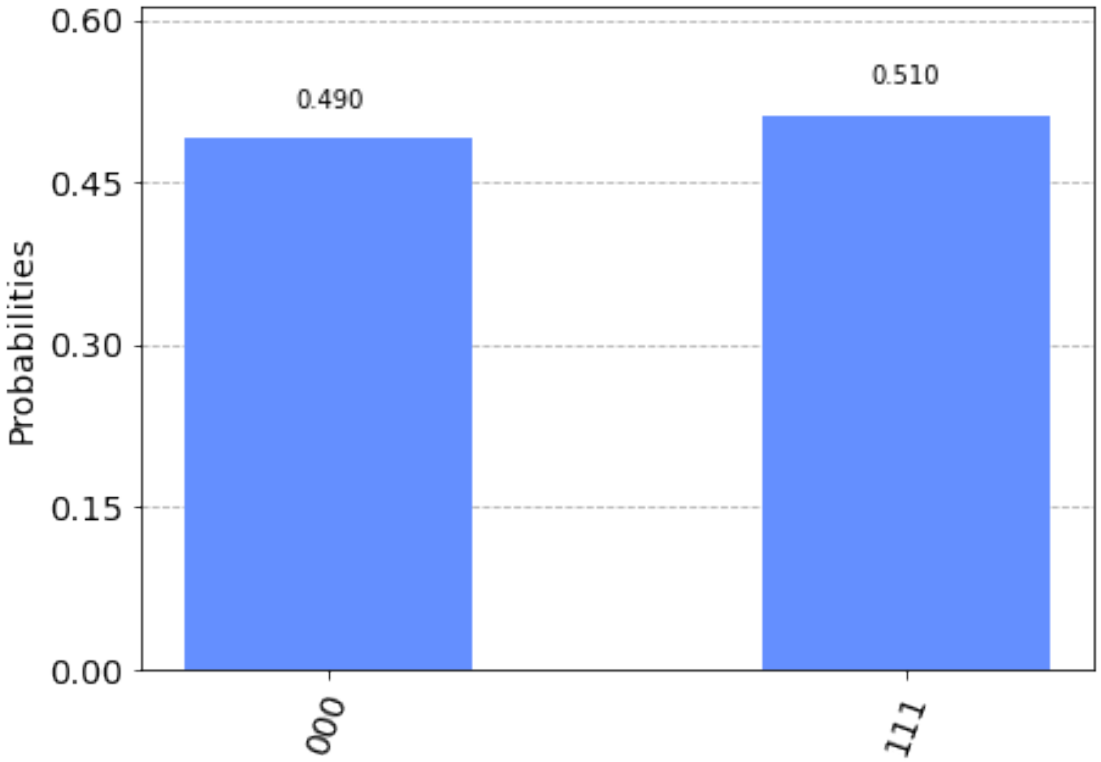
```
# Capturamos las ocurrencias de salida
counts = result.get_counts(circuit)
# Escribimos el número de ocurrencias
print("\nNúmero de ocurrencias para 0 y 1:",counts)
```

```
Número de ocurrencias para 0 y 1: {'000': 490, '111': 510}
```

Aqui se puede observar que en condiciones ideales solo existen dos salidas '111' y '000'.

[5]:
```
plot_histogram(counts)
```

[5]:




### 3.3 Condiciones no Idóneas (Reales)

En este parte observaremos como el ruido afecta al resultado de aplicar el circuito a tres qbits en condiciones no ideales(con presencia de ruido).

[7]:
```
from qiskit import QuantumCircuit, execute
from qiskit import IBMQ, Aer
from qiskit.visualization import plot_histogram
from qiskit.providers.aer.noise import NoiseModel
from qiskit.compiler import transpile, assemble, schedule
```

```
IBMQ.save_account('[redacted]
                  ␣
 ↪'[redacted]',␣
 ↪overwrite=True)
provider = IBMQ.load_account()
```

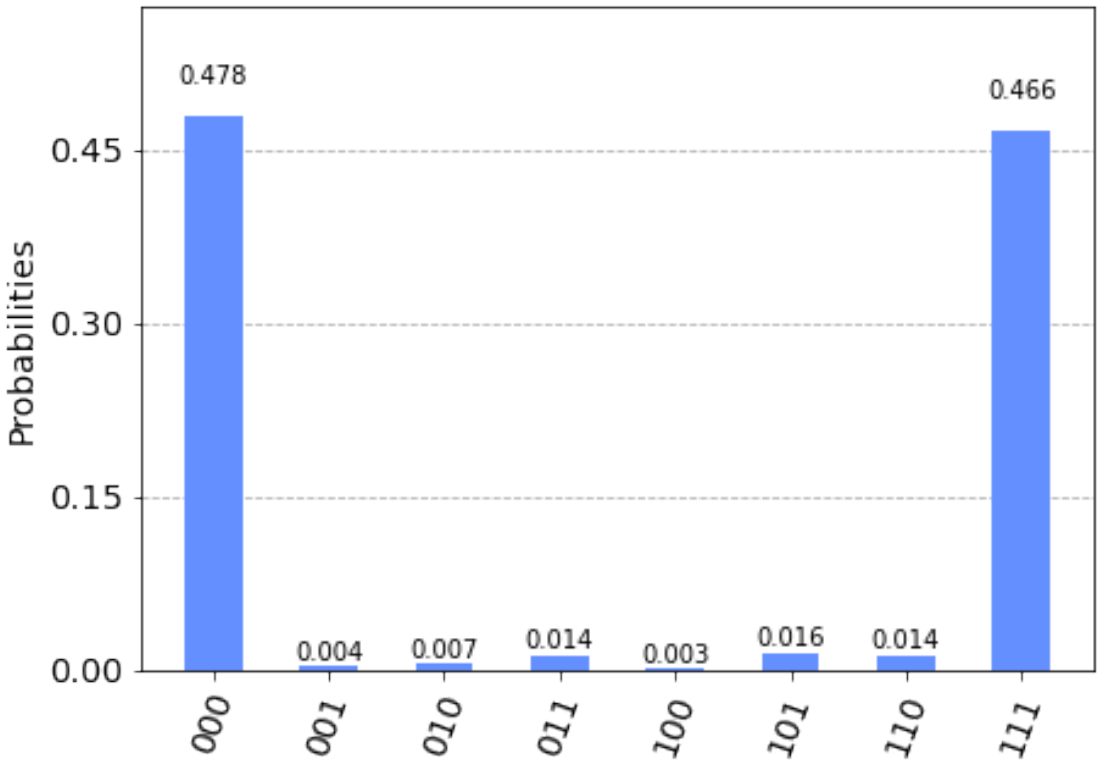
### 3.3.1 Simulacion con ruido

En este primer apartado utilizaremos modelos de ruido proporcionados de backends reales y añadiremos dicha información al simulador para que este genere las salidas que podrían ocurrir al ejecutarlos en ordenadores cuanticos reales.

[17]:
```
# Construir un modelo de ruido a partir de las características de un backend real
backend = provider.get_backend('ibmq_santiago')
noise_model = NoiseModel.from_backend(backend)

# Obtener el mapa de interconexión de los qubits
coupling_map = backend.configuration().coupling_map

# Obtener las características de las puertas básicas
basis_gates = noise_model.basis_gates

######################
# Crear circuito     #
######################

# Perform a noise simulation

result = execute(circuit, Aer.get_backend('qasm_simulator'),
                 coupling_map=coupling_map,
                 basis_gates=basis_gates,
                 noise_model=noise_model).result()

######################
# Mostrar resultados #
######################

counts = result.get_counts(circuit)
print("\nIBMQ_SANTIAGO:")
print("\nNúmero de ocurrencias:",counts)
plot_histogram(counts)
```

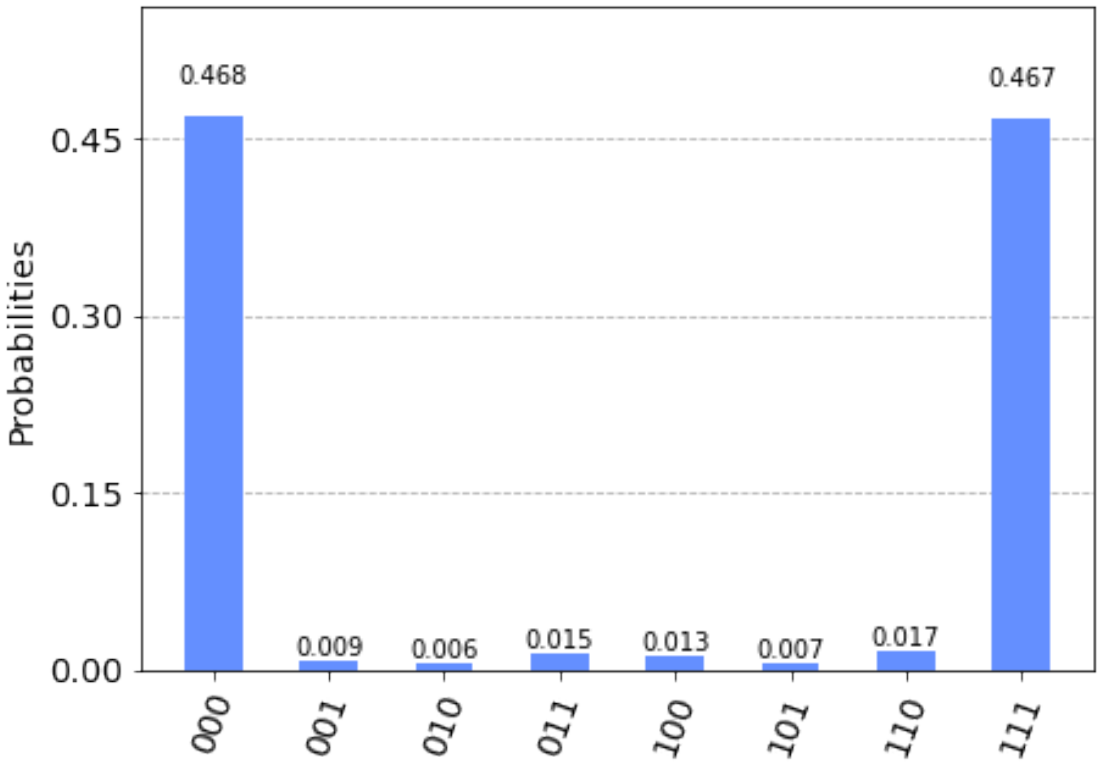
```
IBMQ_SANTIAGO:
Número de ocurrencias: {'000': 489, '001': 4, '010': 7, '011': 14, '100': 3,
'101': 16, '110': 14, '111': 477}
```

[17]:



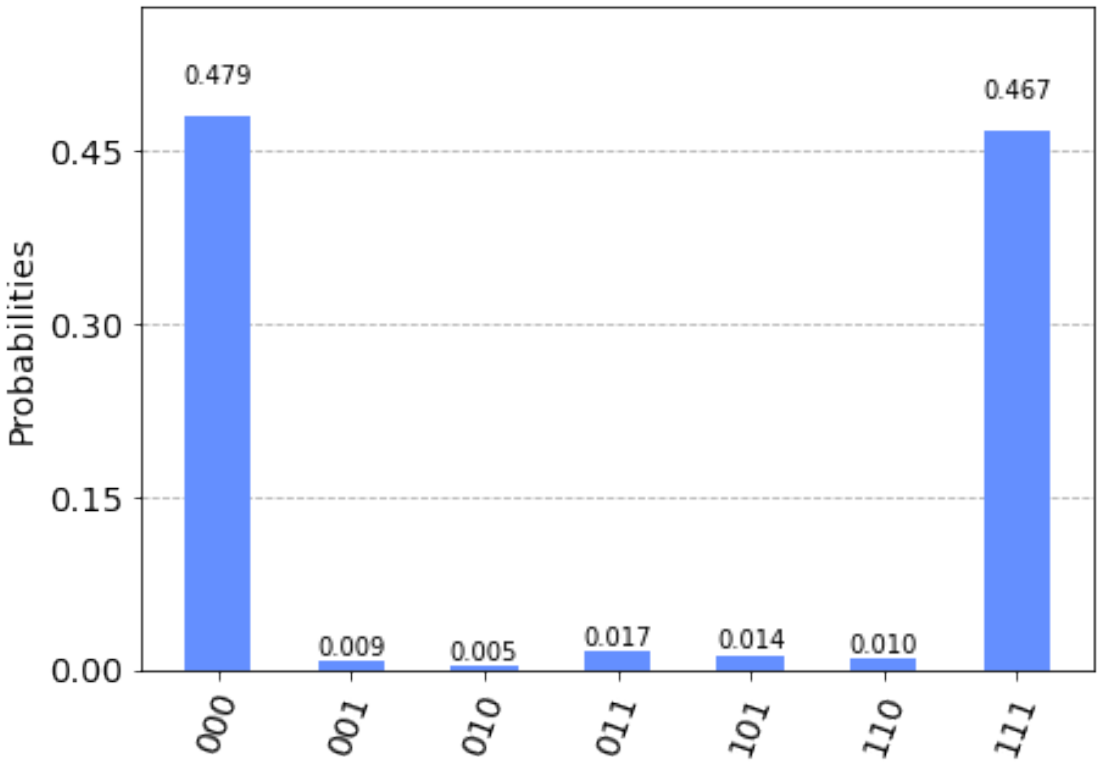

```
------------------- Datos generardos por otros modelos de ruido -------------------

IBMQ_ATHENS:
Número de ocurrencias: {'000': 479, '001': 9, '010': 6, '011': 15, '100': 13,
'101': 7, '110': 17, '111': 478}
```

[17]:



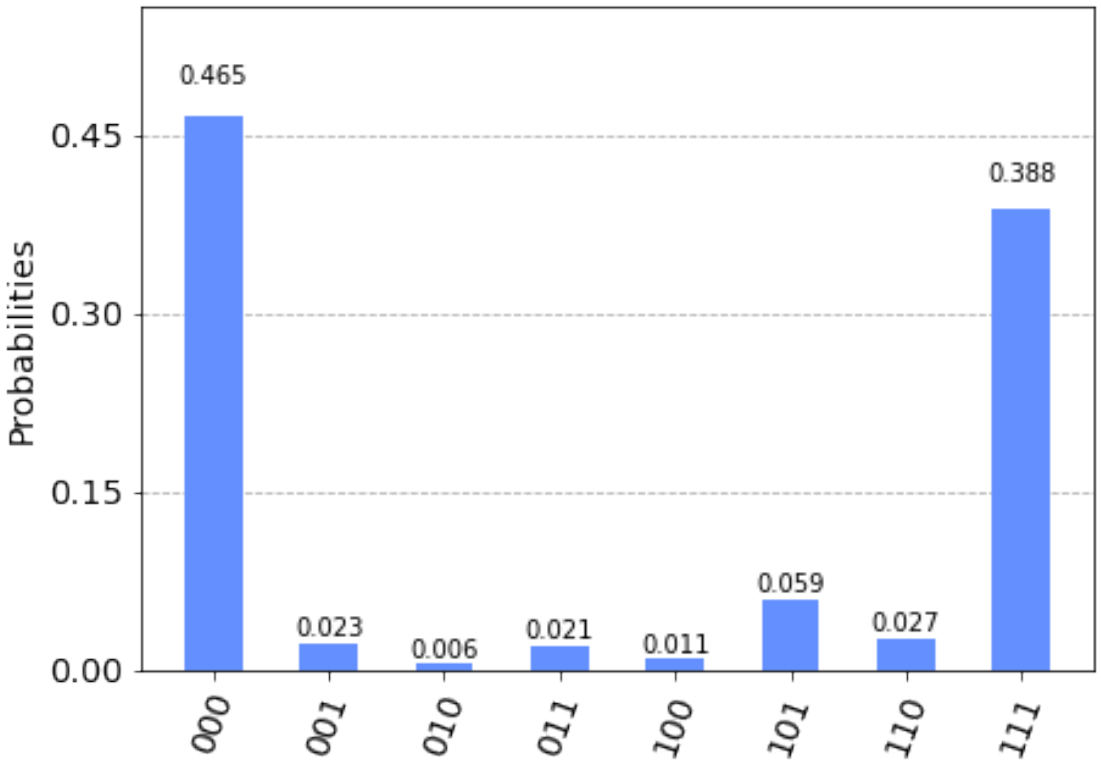

```
IBMQ_LIMA:
Número de ocurrencias: {'000': 491, '001': 9, '010': 5, '011': 17, '101': 14,
'110': 10, '111': 478}
```

[17]:




```
IBMQ_BELEM:
Número de ocurrencias: {'000': 476, '001': 24, '010': 6, '011': 22, '100': 11,
'101': 60, '110': 28, '111': 397}
```

[17]:




### 3.3.2 Ejecucion en ordenadores cuanticos reales

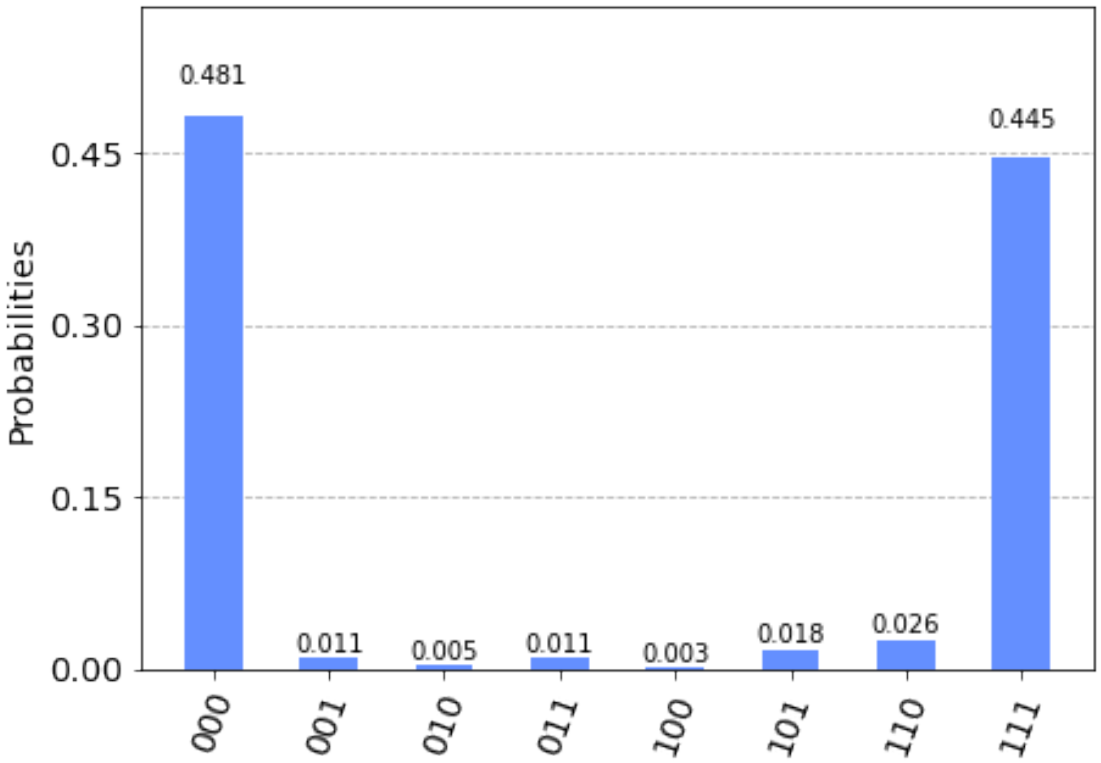
En este apartado enviaremos nuestros circuitos a backends reales como ibmq_santiago y ibmq_athens proporcionados por ibm atraves de ibm quantum experience. Estos backends nos devolveran las salidas reales que nuestros circuitos generaran y podremos observar la presencia de ruido en los mismos ya que no siempre saldra la solucion deseada, lo que nos da a entender que aun queda hasta que estos ordenadores puedan ser utilizados en todo su potencial.

[32]:
```
######################
#   Crear circuito   #
######################

backend = provider.backends.ibmq_belem
qobj = assemble(transpile(circuit, backend=backend), backend=backend)
job = backend.run(qobj)
retrieved_job = backend.retrieve_job(job.job_id())

######################
# Mostrar resultados #
######################
```

```
result = job.result()
counts = result.get_counts(circuit)
print("\nIBMQ_BELEM:")
print("\nNúmero de ocurrencias para 0 y 1:",counts)
plot_histogram(counts)
```

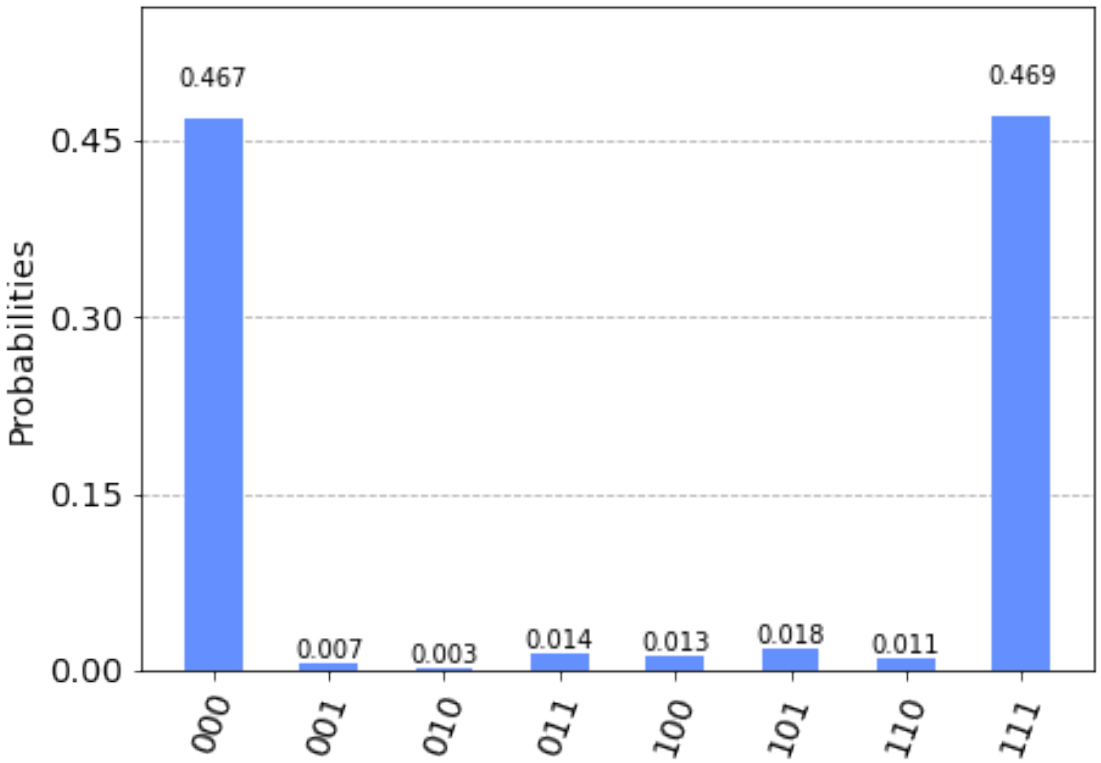
```
IBMQ_BELEM:
Número de ocurrencias para 0 y 1: {'000': 449, '001': 18, '010': 4, '011': 17,
'100': 10, '101': 20, '110': 37, '111': 469}
```

[32]:




```
------------------- Datos generardos por otros ordenadores -------------------

IBMQ_ATHENS:
Número de ocurrencias para 0 y 1: {'000': 478, '001': 7, '010': 3, '011': 14,
'100': 13, '101': 18, '110': 11, '111': 480}
```

[32]:




### 3.3.3 Observaciones

Se observa q a mayor volumen cuantico menor tasa de fallos presenta el resultado en cuanto a los estados que deberian salir en condiciones idoneas.

Tambien podemos ver que aunque son diferentes los resultados dados por un computador real y su simulacion con ruido , los resultados no varian mucho ya que el modelo de ruido esta cojido de dichos ordenadores y daran resultados muy parecidos.