Using Qiskit to Build Quantum Circuits

Nuno Marques, Physics of Classical and Quantum Information, 2022/23

Instituto Superior Técnico, Universidade de Lisboa, 10/1/2023

What is Qiskit?

- Qiskit is an open-source software development kit that allows you to simulate circuits and algorithms using Python.
- Can test your circuits on real devices.
- It is best used in combination with Jupyter Notebooks!

Initialization

- Initialize a circuit by calling the function `QuantumCircuit(n,k)` ; `n` is the number of qubits and `k` is the number of bits.
- By default each qubit is set to 0.
- Set the qubits by labelling with a string (e.g `001`) or by using a list separated by commas. The values in this list are the scalars for the basis vectors. Be mindful of $little Endian$!
- Example for a three-qubit system: $|000\rangle$ $|001\rangle$ $|010\rangle$ $|011\rangle$ $|100\rangle$ $|101\rangle$ $|110\rangle$ $|111\rangle$

Note: you can draw both the state and the circuit using the `draw()` method.

In [1]:
```
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector
import numpy as np

qc = QuantumCircuit(3)

qc.initialize([1, 0, 0, 0, 0, 0, 0, 0], qc.qubits)

st0 = Statevector(qc)
st0.draw(output='latex')
```

Out[1]:

$$|000\rangle$$

In [2]:
```
qc.initialize('000', qc.qubits)
st0 = Statevector(qc)
```

```
st0.draw(output='latex')
```

Out[2]:

$$|000\rangle$$

Adding Gates

Now that we know how to initialize our state we want to know how to add some gates.

This is actually very easy since it is just calling a `method` to the circuit `object`!

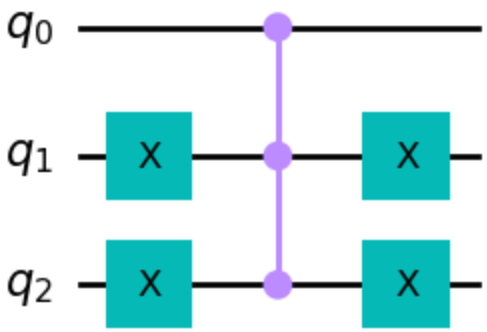
The basic syntax goes like this: `qc.GATE(control_qubit1: QubitSpecifier, ..., target_qubit: QubitSpecifier)`. It is also important to know that we should add the gates from the left to the right.

Let's try to make a very simple circuit consisting of three qubits and five gates. Two **X** gates acting on the first two qubits, followed by a **CCZ** gate controlled by these qubits and acting on the third and, again, two **X** gates acting on the first two qubits.

This is actually the $FlipOperator$ from PSET9!

In [3]:
```
qc = QuantumCircuit(3)

qc.x(1)

qc.x(2)

qc.ccz(1, 2, 0)

qc.x(1)

qc.x(2)

qc.draw('mpl')
```

Out[3]:



Evolution

We can see what the circuit does to our initial state using the `evolve` method.

In [4]:
```
state = Statevector.from_label('001')

state.evolve(qc).draw(output='latex')
```

Out[4]:

$$-|001\rangle$$

Measurement

We move on to a more complex task. In PSET8 there is a circuit that allows us to measure the overlap between some arbitrary states $|\alpha\rangle$ and $|\beta\rangle$.

To test this we start initializing these qubits as random states (`random_statevector` method) and passing them through the circuit.

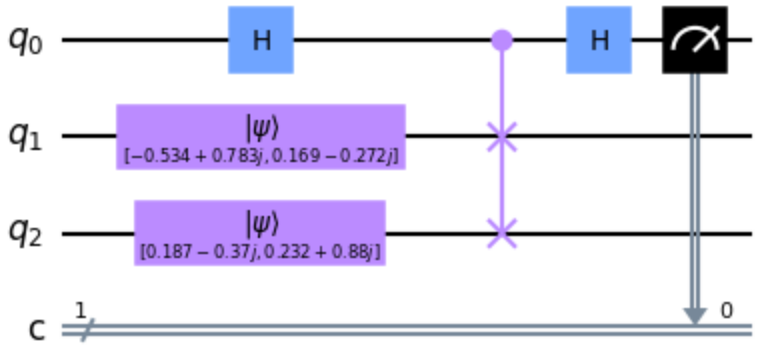
We can measure the circuit by calling the `measure(qubit: QubitSpecifier, cbit: ClbitSpecifier)` method on the circuit `object` where the second argument is the bit where we want to store our measurement. If we wanted to measure all our qubits we could just use `measure.all()` .

Afterwards, we construct an histogram by observing the results of the experiment many times using the `Aer` simulator.

Then we are able to actually compute the overlap!

In [5]:
```
from qiskit.quantum_info import random_statevector

qc = QuantumCircuit(3, 1)

state_a = random_statevector(2)
state_b = random_statevector(2)

actual_overlap = (np.abs(state_a.inner(state_b)))

qc.initialize(state_a, 1), qc.initialize(state_b, 2), qc.h(0), qc.cswap(0, 1

qc.measure(0, 0)

qc.draw('mpl', scale = 0.75)
```

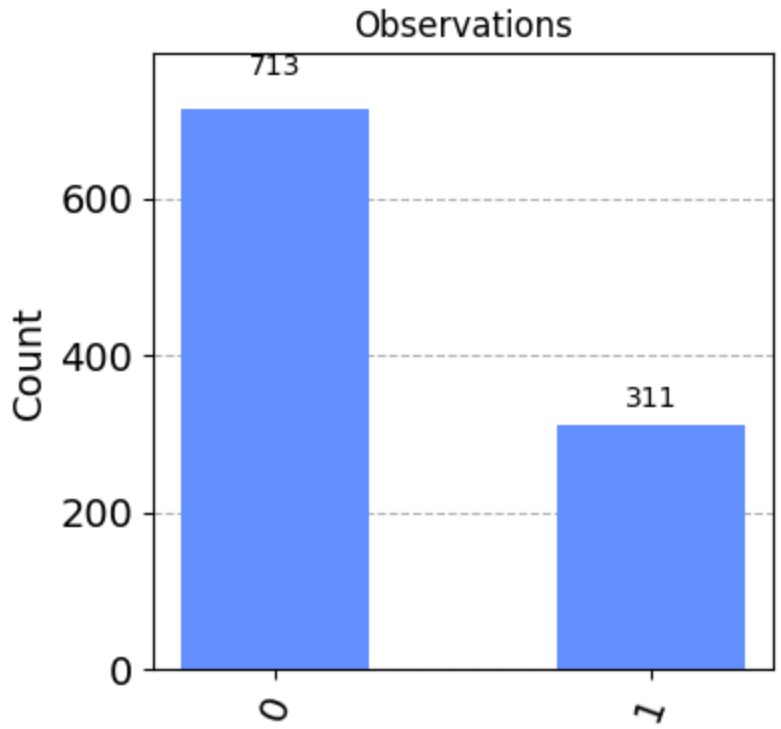
Out[5]:



In [6]:

```
from qiskit import Aer, transpile, assemble
from qiskit.visualization import plot_histogram

simulator = Aer.get_backend('aer_simulator')
circ = transpile(qc, simulator)
result = simulator.run(qc).result()
counts = result.get_counts(qc)

plot_histogram(counts, title='Observations', figsize=(4, 4))
```

Out[6]:



In [7]:

```
measured_prob = counts.get('1') / (counts.get('0') + counts.get('1'))

measured_overlap = np.abs(np.sqrt(1 - 2*measured_prob +0j))

print("Actual overlap: " + str(actual_overlap) + ", Measured overlap: " +
      str(measured_overlap) + ", Difference: " + str(np.abs(actual_overlap-m
```

```
Actual overlap: 0.6454241825568905, Measured overlap: 0.6265605517426069, D
ifference: 0.01886363081428355
```

Making Deutsch-Jozsa

The first step in implementing this algorithm is creating a function that generates random Oracles.

In order to do this we first notice that the the Oracle leaves the system in the state: $\frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2n-1}(-1)^{f(x)}|x\rangle$.

If the Oracle is constant then it can be considered to either be the identity matrix I or the diagonal matrix where all of it's elements are -1.

If the Oracle is balanced then it can be viewed as a diagonal matrix where half of its elements are chosen randomly to be -1 and the rest 1!

In [8]:
```python
import random as rand

def dj_oracle(n):
    '''
    Input: the registry size n
    Output: Random unitary matrix U and a string

    The function is chosen randomly to be constant or balanced.
    If it's balanced it can either be f(x)=0 or f(x)=1.
    If it's balanced we choose N / 2 elements without replacement

    When f(x)=1 we multiply U by -1
    When its balanced we multiply the chosen indexes by -1
    '''

    N = 2 ** n

    indexes = list(np.arange(N))

    U = np.identity(N)

    type = rand.choice(["constant", "balanced"])

    if type == "constant":

        type = rand.choice(["f(x)=0", "f(x)=1"])

        if type == "f(x)=1":

            U = U * -1

    if type == "balanced":

        for i in range(int(N/2)):
            ind = rand.choice(indexes)
```

```
            U[ind][ind] = -1
            indexes.remove(ind)

    return U, type
```

Then we can actually implement the algorithm.

To do this we simply copy the steps from the original algorithm.

Note: Since our return type is a `np.array` we use the `unitary` method to add the oracle to the circuit.

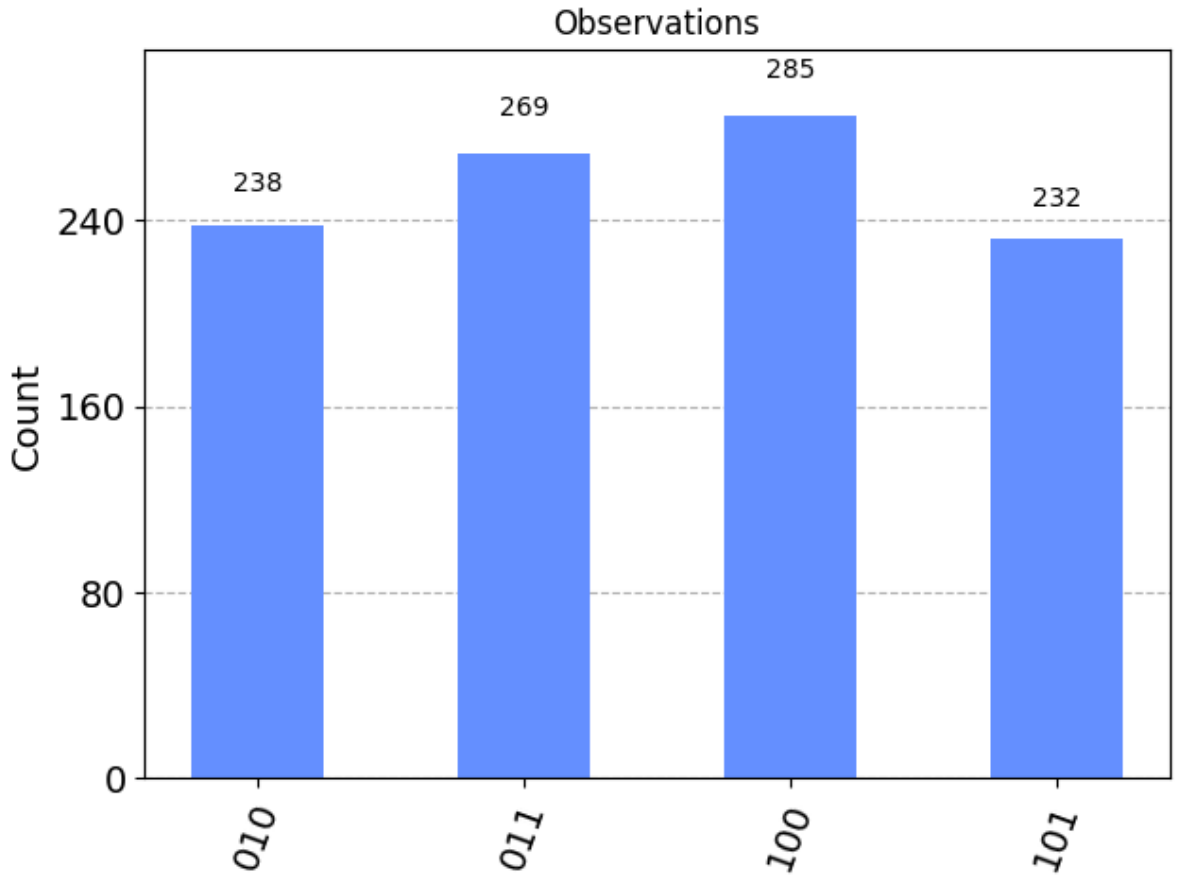
In [9]:
```
def dj_algorithm(oracle):
    '''
    Input: The Oracle
    Output: O circuito quântico dj

    '''

    n = int(np.log2(np.size(oracle[0])))

    dj = QuantumCircuit(n + 1, n)

    # Hadamard nos bits input
    for i in range(n):
        dj.h(i)

    # |0> -> |1> -> |->
    dj.x(n)
    dj.h(n)

    # Oráculo
    dj.unitary(oracle, range(n))

    for i in range(n):
        dj.h(i)

    for i in range(n):
        dj.measure(i, i)

    return dj
```

Now we can finally simulate the algorithm using the techniques learned previously!

In [10]:
```
U, type = dj_oracle(3)
print(type)
dj = dj_algorithm(U)

circ = transpile(dj, simulator)
result = simulator.run(circ, shots=1024).result()
counts = result.get_counts(circ)

plot_histogram(counts, title='Observations')
```

```
balanced
```

Out[10]:



Making Grover's

As was previously done, we need to start by constructing a random Oracle.

In the original problem we can know a priori the number of solutions. If we don't know then we choose a random number between 1 and N.

Also if the solution is not known it is useful to consider a list of size $2N$ instead of N, since the solutions might make up most of our list. We apply the same strategy if the solution is known and takes up half or more of the list.

Then we generate M random indexes ranging from 0 to $N-1$. These indexes will be the indexes of our solutions.

We then take the identity matrix I and multiply the elements corresponding to these indexes by -1, since the operation of the Oracle acting on our system is $|x\rangle \rightarrow (-1)^{f(x)}|x\rangle$. ($(-1)^{f(x)} = -1$ if x is solution)!

In [11]:
```
def g_oracle(N, M='r'):
    '''
    Input: Size of the list and number of solutions if known.
    Output: The Oracle and the ideal number of iterations.

    '''
```

```
    solutions = []

    if M == 'r':
        M = rand.randint(1, N)
        N = 2 * N

    elif M >= N / 2:
        N = 2 * N

    n = int(np.ceil(np.log2(N)))

    indexes = list(range(N))

    # M random solutions
    for i in range(M):
        ind = rand.choice(indexes)
        solutions.append(ind)
        indexes.remove(ind)

    bin_sol = []
    for sol in solutions:
        bin_sol.append('{0:b}'.format(sol))
    print(np.sort(bin_sol))

    U = np.identity(2 ** n)
    for ind in solutions:
        U[ind][ind] = U[ind][ind] * -1

    iter = int(np.floor((np.pi / 4) * np.sqrt(N/M)))

    return U, iter
```

The next logical step is to make the grover's iterator G. This is very simply done!.

In fact, we only need to look at the expression $G = (2|s\rangle\langle s| - I)O$, where s is the uniform superposition of states and O is our oracle.

We also introduce another useful method, the `to_gate` method, to cast this `np.array object` to `gate object`

In [12]:
```
def G(n, oracle):

    qc = QuantumCircuit(n)

    dim = 2 ** n

    uni_sup = np.ones((1, dim)) * (1 / (np.sqrt(dim)))

    G = np.matmul((2 * np.kron(np.transpose(uni_sup),
                  uni_sup) - np.identity(dim)), oracle)

    qc.unitary(G, range(n))

    G_gate = qc.to_gate()
```

```
    G_gate.name = "G"

    return G_gate
```

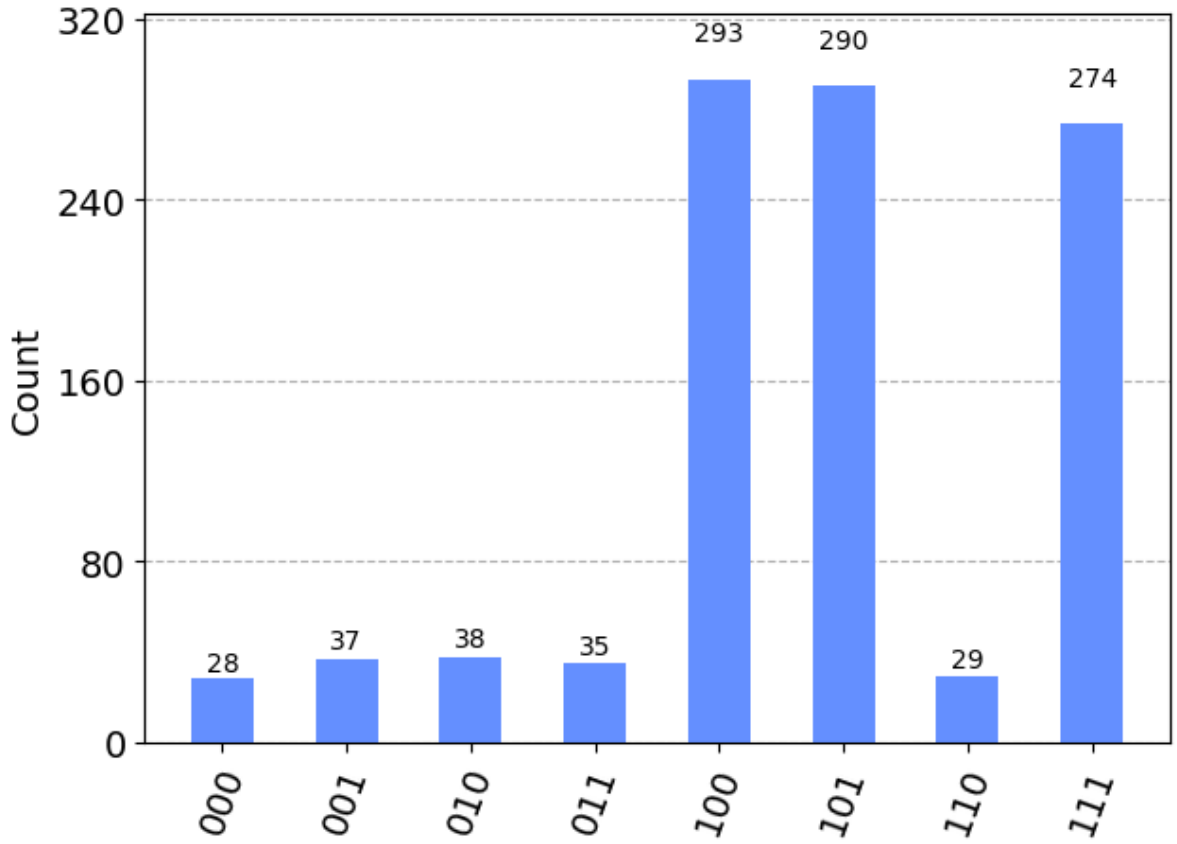
Lastly, we implement the algorithm by following the steps of the original and run a couple of simulations!

In [13]:
```
def grover(oracle):

    n = int(np.log2(np.size(oracle[0])))

    qc = QuantumCircuit(n)

    for qubit in range(n):
        qc.h(qubit)

    for i in range(ITER):
        qc.append(G(n, oracle), range(n))

    qc.measure_all()
    return qc
```

In [14]:
```
oracle, ITER = g_oracle(4, 3)
grover_qc = grover(oracle)

transpiled_grover_circuit = transpile(grover_qc, simulator)
results = simulator.run(transpiled_grover_circuit, shots=1024).result()
counts = results.get_counts()
plot_histogram(counts)
```

```
['100' '101' '111']
```

Out[14]:



Using Real Devices

We can run our circuit in real devices. For that we need to get a `provider`, a `device` and a `backend`. Some parameters might need tuning according to the circuit we are trying to simulate.

If you want to simulate it yourself you will need to create an IBM account.

This will not be done live sadly because it takes time!

In [21]:

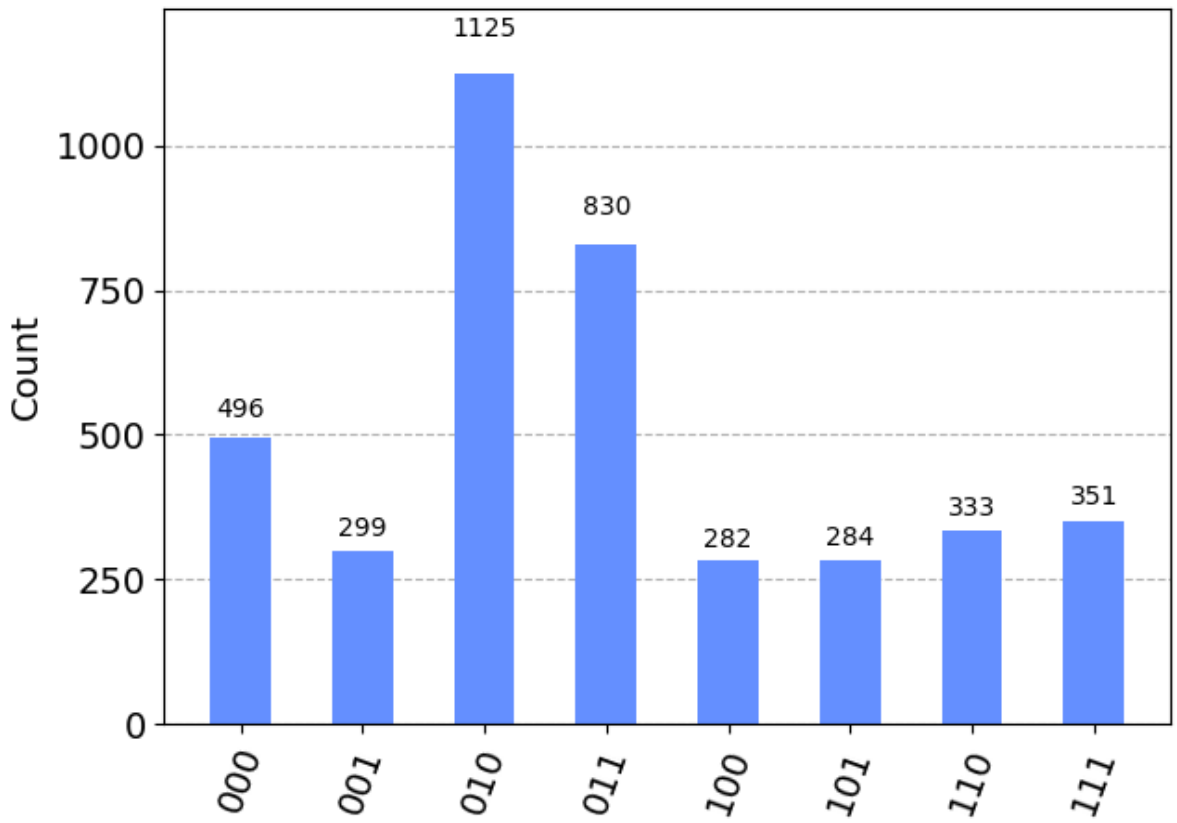
```
from qiskit import IBMQ
from qiskit.providers.ibmq import least_busy

N = 3

oracle, ITER = g_oracle(3)
grover_qc = grover(oracle)

size = np.ceil(np.log2(N))

provider = IBMQ.get_provider("ibm-q")
device = least_busy(provider.backends
(filters=lambda x: int(x.configuration().n_qubits) >= size and
not x.configuration().simulator and x.status().operational==True))

backend = least_busy(provider.backends
```

```
(filters=lambda x: int(x.configuration().n_qubits) >= size and
not x.configuration().simulator and x.status().operational==True))
```

```
['10' '11']
```

In [23]:
```
# Run our circuit on the least busy backend. Monitor the execution of the jo
from qiskit.tools.monitor import job_monitor
transpiled_grover_circuit = transpile(grover_qc,
device, optimization_level=0)

job = device.run(transpiled_grover_circuit)

# Get the results from the computation
results = job.result()
answer = results.get_counts(grover_qc)
plot_histogram(answer)
```

Out[23]:



Conclusion

We have seen that Qiskit is a powerful tool for simulating our quantum circuits, allowing us to confirm our theoretical knowledge about both the Deutsch-Jozsa Algorithm and Grover's Algorithm.

Using this kind of software is imperative in bridging the worlds of mathematics, computer science, and quantum physics, hopefully, someday, leading to the discovery of new, more complex ground-breaking algorithms.

Bibliography

[1] Nielsen, M., & Chuang, I. (2010). Quantum Computation and Quantum Information. Cambridge University Press

[2] Library Documentation https://qiskit.org/documentation

Thank you for listening!