Implementation of the Teleportation algorithm in qSOA®

Contents

1 The teleportation algorithm

1.1 The problem

Alice and Bob want to share quantum information. Suppose Alice wants to send the qubit state $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$ to Bob. In quantum mechanics, there exists a theorem that states that it is impossible to create an independent and identical copy of an arbitrary unknown quantum state, known as the no-cloning theorem. We can only copy classical states, not superpositions, therefore Alice can not copy $|\phi\rangle$ and sent it to Bob.

However, with two classical bits and an entangled qubit pair, Alice can transfer her state, $|\phi\rangle$, to Bob. It is called teleportation because, in the end, Bob will have the state $|\phi\rangle$ and Alice won't anymore.

1.2 Quantum teleportation algorithm

To transfer a quantum bit, Alice and Bob must use a third party, Telamon, to send the entangled qubit pair, as we can see in figure 1. Alice then performs some operations on her qubit and sends the results to Bob over a classical communication channel. Finally, Bob performs some operations on his end to receive Alice's qubit.

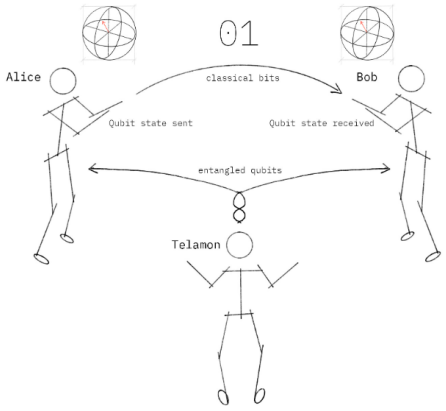


Figure 1: Teleportation diagram

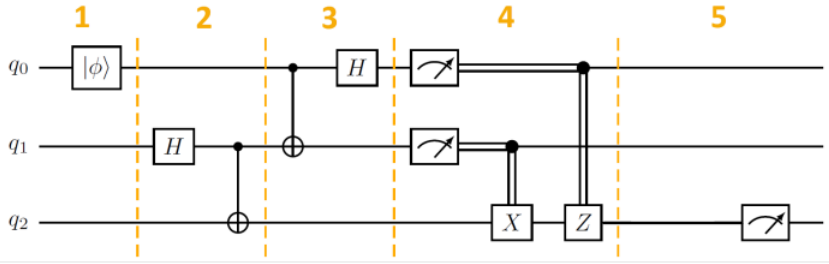


Figure 2: Teleportation circuit

In figure 2, we can see the circuit that describes quantum teleportation. The qubit q_0 is Alice's qubit, and Bob's will be q_2. The steps followed by the algorithm are:

1. Alice initializes her qubit in the state ϕ.
2. A third party, Telamon, prepares a pair of entangled qubits. For example, a Bell state:

$$|\phi_B\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

   So, in this step we have the state:

$$|\Psi\rangle = |\phi\rangle \otimes |\phi_B\rangle = \frac{1}{\sqrt{2}}(\alpha|0\rangle \otimes (|00\rangle + |11\rangle) + \beta|1\rangle \otimes (|00\rangle + |11\rangle))$$

3. Now, using the entangled pair, Alice applies a CNOT gate, state $|\Psi'\rangle$, and a Hadamard gate on her qubit, leaving the state to be $|\Psi''\rangle$. That is:

$$\begin{aligned}|\Psi'\rangle = &\frac{\alpha}{2}[|000\rangle + |011\rangle + |100\rangle + |111\rangle] \\ &+ \frac{\beta}{2}[|010\rangle + |001\rangle - |110\rangle - |101\rangle]\end{aligned}$$

$$\begin{aligned}|\Psi''\rangle = &\frac{1}{2}|00\rangle(\alpha|0\rangle + \beta|1\rangle) + \frac{1}{2}|01\rangle(\beta|0\rangle + \alpha|1\rangle) \\ &+ \frac{1}{2}|10\rangle(\alpha|0\rangle - \beta|1\rangle) + \frac{1}{2}|11\rangle(-\beta|0\rangle + \alpha|1\rangle)\end{aligned}$$

4. Alice measures the first two qubits and sends the result as two classical bits to Bob. From $|\Psi''\rangle$ we can see that Alice can obtain one of the four standard basis states, $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$, with equal probability. On her measurement, Bob's state will be projected to the following states:

$$\begin{aligned}|00\rangle &\rightarrow \alpha|0\rangle + \beta|1\rangle \\ |01\rangle &\rightarrow \alpha|1\rangle + \beta|0\rangle \\ |10\rangle &\rightarrow \alpha|0\rangle - \beta|1\rangle \\ |11\rangle &\rightarrow \alpha|1\rangle - \beta|0\rangle\end{aligned}$$

   So when Alice measures her qubits and tells Bob her results, Bob knows how he can obtain the original state, ϕ, by applying the corresponding gates.

   The transformations that he needs to apply are:

Alice Results	Bob's state	Gate needed
00	$\alpha\|0\rangle + \beta\|1\rangle$	I
01	$\alpha\|1\rangle + \beta\|0\rangle$	X
10	$\alpha\|0\rangle - \beta\|1\rangle$	Z
11	$\alpha\|1\rangle - \beta\|0\rangle$	ZX

After this state, Bob will have successfully reconstructed Alice's state.

5. We are now ready to measure the results.

In figure 2, we see that we must measure before finishing the circuit, but the quantum computers that are available to date do not support instructions after measurements, which means that we cannot perform quantum teleportation on real hardware.

Fortunately, there is a deferred measurement principle, that can be seen in chapter 4.4 from ref.[2], which states that any measurement can be postponed until the end of the circuit. That is, we can move all measurements to the end, and we should see the same results.

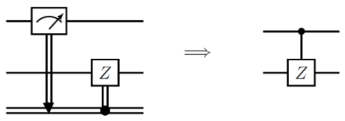


Figure 3: Equivalent circuits

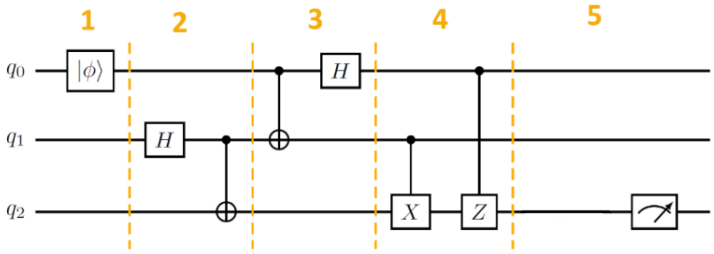


Figure 4: Equivalent teleportation circuit

In this tutorial we are going to set Alice's state to $\phi = \alpha|0\rangle + \beta|1\rangle$, where $\alpha = cos(\theta/2)$, &, $\beta = sen(\theta/2)$, with $\theta = \pi/4$. We are going to do so by applying a R_y-gate.

1.3 Usefulness of the algorithm

We have shown that we can share and send quantum information. As we have already said, there is a theorem that says that we can always move the measurements to the end of the circuit. We have also seen the equivalence between measuring and applying a gate with a controlled gate.

The only downside is that you need the same quantum communication channel. In fact, teleportation has already been carried out in real life but always through a common channel and using a single circuit. Using, for example, fibre optics and photons, which seem to be much more reliable[1][2][3].

After analyzing how the algorithm is implemented in the hardware that has been developed, it is safe to say that there is no real way to break the circuit into pieces, due to the algorithm limitations.

[1] https://journals.aps.org/prxquantum/pdf/10.1103/PRXQuantum.1.020317
[2] https://iopscience.iop.org/article/10.1088/1742-6596/1634/1/012089/pdf
[3] http://mmrc.amss.cas.cn/tlb/201702/W020170224608150244118.pdf

2 Implementation of the algorithm in qSOA®

Once we are aware of how the algorithm works, we are ready to implement it in qSOA®. This will allow us to create the circuit and execute it in different quantum computing providers, among many other things.

The process of implementing an algorithm in qSOA® is comprised of four steps:

1. Setting up qSOA® and selecting the quantum solution
2. Creating a circuit with the algorithm and assigning it to the solution
3. Introducing a circuit flow to control the number of launches of the algorithm
4. Executing the flow on different quantum devices.

As can be seen in qSOA's® manual, ref.[1], there are multiple ways to secure the connection depending on the context. Following the best practices of qSOA®, in this tutorial we are integrating the security in the code. Similarly, one can work with asynchronous or synchronous programming. Keeping in mind the purpose of this tutorial we will use the synchronous version.

2.1 Setting up qSOA®

Firstly, we import the SDK that has been previously installed, see ref.[1], and create the qSOA® workspace to work with.

[1]:
```
from QuantumPathQSOAPySDK import QSOAPlatform  # Import SDK
```

[2]:
```
# Create qSOA workspace, login manually
qsoa = QSOAPlatform()

username = 'username'
password = 'password' # password encrypted in SHA-256

authenticated = qsoa.authenticateEx(username, password)

print('Authentication completed:', authenticated)
```

```
Authentication completed: True
```

Then we review the existing solutions and select the one we are interested in.

[3]:
```
# Get catalogs
solutionList = qsoa.getQuantumSolutionList()
print("   ",solutionList)
idSolution = int(input("Select idSolution: "))
```

```
    {'12345': 'QS_GateTutorials'}
Select idSolution: 12345
```

2.1.1 Securing the connection

As has been said, qSOA® allows multiple business development contexts. Therefore, the user can secure the connection through a configuration file, *.qpath*, useful at a personal level, or in parameterized way, as it is done here.

2.2 Assigning a circuit to the solution

Once the solution has been selected, we must link it to the circuit that we are interested in implementing. Therefore, we first need to define the circuit and create it.

2.2.1 Defining the circuit

As we are working with a gates circuit, we can either enter the circuit in *visual language*, VL, or *intermediate language*, IL. In this case we are going to work with VL and, therefore, we can either enter the circuit as a *String* or as a *CircuitGates* object, which is what we are going to go for.

In order to define the circuit shown in figure 4, we just need to write the gates we want to use, in the correspondent qubits.

[4]:
```
## Defining circuit w/ Circuit Gates
circuitG = qsoa.CircuitGates()
circuitG.ry(0,'pi/4')
circuitG.h(1)
circuitG.cx(1,2)
circuitG.barrier()
circuitG.cx(0,1)
circuitG.h(0)
circuitG.barrier()
circuitG.cx(1,2)
circuitG.mcg(0,circuitG.z(2,False))
circuitG.measure()

print(circuitG.getCircuitBody())
```

```
[[{'id': 'RY', 'arg': 'pi/4'}, 'H'], [1, 'CTRL', 'X'], ['SPACER', 'SPACER',
'SPACER'], ['CTRL', 'X'], ['H'], ['SPACER', 'SPACER', 'SPACER'], [1, 'CTRL',
'X'], ['CTRL', 1, 'Z'], ['Measure'], [1, 'Measure'], [1, 1, 'Measure']]
```

Note: Every gate can be applied to a single qubit, a list of them, introduced as a list, or to every qubit in the circuit, using ().

2.2.2 Creating the circuit

In order to create the circuit we are going to use the *createAssetSync* function. This function receives the following fields as inputs:

- idSolution: to associate the circuit with the solution we have selected before.

- assetName: to set the name of the circuit.
- assetNamespace: to associate the circuit with a class of circuits that share something in common. In this case, we associate the circuit with a set of basic circuits.
- assetDescription: to write a brief description of the algorithm.
- assetBody: to select the circuit we have previously defined.
- assetType: to select if we are working with a gates circuit or an annealing one.
- assetLevel: to select either visual language or intermediate language, according to the definition of the circuit.

[5]:
```
## Circuit creation
assetName = 'QC_qSOA_Teleportation'
assetNamespace = 'Manual.Gates.Teleportation'
assetDescription = 'Creating the teleportation circuit from qSOA'

assetBody = circuitG
assetType = 'GATES'
assetLevel = 'VL'

CircuitManagementResult = qsoa.createAssetSync(idSolution, assetName,
assetNamespace, assetDescription, assetBody, assetType, assetLevel)
```

Note: Note that the synchronous version of this function is being used. This is because the *createAsset* function creates, compiles and transpiles the asset, and the synchronous function waits for all of it to be done before moving on. For a better understanding of how this functions works, we recommend using the asynchronous version, *createAsset*, and the *getAssetManagementResult* function.

2.3 Assigning a circuit flow to the circuit

The quantum flow is a box diagram that will allow us to control the number of runs of our algorithm. This is very useful, as the results obtained in quantum computing have a certain probability associated with them. That means that the more times we run the algorithm, the more robust the results will be.

2.3.1 Defining the flow

As happens with the circuit, the flow can be written in VL or IL. If we choose to do it with VL then it can be entered as a *String* or a *CircuitFlow* object. Otherwise, it can only be entered as a *String*. For this example we are choosing VL and *CircuitFlow*.

In order to define a flow we need:

1. Starting node
2. Initializing node: usually set to 0

3. Circuit node: where we write the circuit we want lo launch
4. Repeat node: where the number of repetitions can be establish
5. End node
6. Links between each node we have created

[6]:
```
# Defining flow w/ CircuitFlow
flow = qsoa.CircuitFlow()
startNode = flow.startNode()
initNode = flow.initNode(0)
circuitNode = flow.circuitNode('Manual.Gates.Teleportation.
 ↪QC_qSOA_Teleportation')
# Namespace + CircuitName
repeatNode = flow.repeatNode(1000)
endNode = flow.endNode()

flow.linkNodes(startNode, initNode)
flow.linkNodes(initNode, circuitNode)
flow.linkNodes(circuitNode, repeatNode)
flow.linkNodes(repeatNode, endNode)
print(flow.getFlowBody())
```

[6]:
```
{'class': 'go.GraphLinksModel', 'nodeDataArray': [{'category': 'Start', 'text':
'Start', 'key': -1, 'loc': ''}, {'category': 'Init', 'text': '0', 'key': -2,
'loc': ''},{'category': 'Circuit', 'text': 'Manual.Gates.Teleportation.
 ↪QC_qSOA_Teleportation', 'key': -3, 'loc': ''}, {'category': 'Repeat', 'text':
'1000', 'key': -4, 'loc': ''}, {'category': 'End', 'text': 'End', 'key': -5,
'loc': ''}], 'linkDataArray': [{'from': -1, 'to': -2, 'points': []}, {'from':
-2, 'to': -3, 'points': []}, {'from': -3, 'to': -4, 'points': []}, {'from': -4,
'to': -5, 'points': []}]}
```

2.3.2 Creating the flow

In order to create the flow we are using *createAssetFlowSync*, although the *createAssetSync* function would also work. The inputs that this function requires are:

- idSolution: to associate the flow with the solution we have selected before.
- assetName: to set the name of the flow.
- assetNamespace: to associate the flow with a class of flows that share something in common. In this case, we associate the flow with a set of basic flows.
- assetDescription: to write a brief description of the algorithm.
- assetBody: to select the flow we have previously defined.
- assetLevel: to select either VL or IL, according to the definition of the flow.

- publish: to select if we want to publish the flow on qSOA® or not.

[7]:
```
## Flow creation
assetName = 'QF_qSOA_Teleportation'
assetNamespace = 'Manual.Gates.Teleportation'
assetDescription = 'Creating the teleportation flow from qSOA'
assetPublication = True

assetBody = flow
assetType = 'FLOW'
assetLevel = 'VL'

FlowManagementResult = qsoa.createAssetFlowSync(idSolution, assetName,
assetNamespace, assetDescription, assetBody, assetLevel,assetPublication)
```

Note: Note that the synchronous version of this function is being used. This is because the *createAssetFlow* function creates, compiles and transpiles the flow, and the synchronous function waits for all of it to be done before moving on. For a better understanding of how this functions works, we recommend using the asynchronous version, *createAssetFlow*, and the *getAssetManagementResult* function.

2.4 Implementation of the algorithm

2.4.1 Executing the algorithm

We are now ready to execute the algorithm, so lets see on what platforms we can do so, and select the ones we are interested in.

[8]:
```
deviceList = qsoa.getQuantumDeviceList(idSolution)
print('Device List:', deviceList)
DeviceID = input("Select a device to run the flow in: ")
```

```
Device List: {'14': 'AMAZON BRAKET 25qbits Local Simulator', '2': 'QISKIT Local
Simulator', '1': 'Q# Local Simulator Framework'}
Select a device to run the flow in: 14
```

Now, we proceed to run the quantum algorithm with the *runQuantumApplicationSync* function.

[9]:
```
exe_application = qsoa.runQuantumApplicationSync('NameTheTask', idSolution,␣
↪FlowID, DeviceID)
```

Note: Note that the synchronous version of this function is being used. This is because *runQuantumApplication* launches an execution, and the synchronous function waits for the execution to be done before moving on. If we do not know how long the execution is going to take (due to the execution itself or to the queue), or if it takes too long, the asynchronous version presents a

clear advantage. However, for a better understanding of how this functions work, we recommend using the asynchronous version, *runQuantumApplication*.

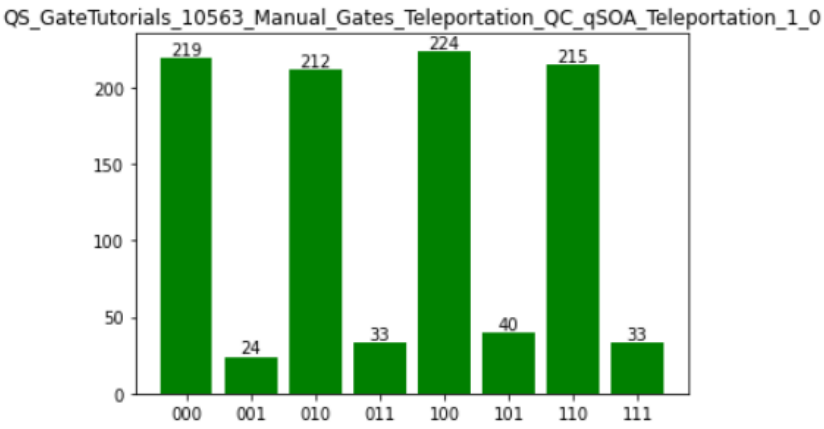
The function *runQuantumApplicationSync* gives an application object as output. For us to manage the results we need the *getQuantumExecutionResponse* function that returns the results as an execution object.

[10]:
```
restok_execution = qsoa.getQuantumExecutionResponse(exe_application.
 ↪getExecutionToken(), idSolution, FlowID)
restok_histogram = restok_execution.getHistogram()
print(restok_histogram)
```

```
{'QS_GateTutorials_12345_Manual_Gates_Teleportation_QC_qSOA_Teleportation_1_0':
{'101': 40, '000': 219, '110': 215, '010': 212, '111': 33, '011': 33, '100': 224,
'001': 24}}
```

Now that we have the results we asked for, we can proceed to represent them with the function *representResults*.

[11]:
```
# Circuit gate representation
gates_representation = qsoa.representResults(restok_execution)
```



2.4.2 Multiple devices

If we want to run the algorithm in multiple devices at the same time, we can do so by creating an array with the information needed and proceeding the same way as before.

[12]:
```
deviceList = qsoa.getQuantumDeviceList(idSolution)
print('Device List:', deviceList)
```

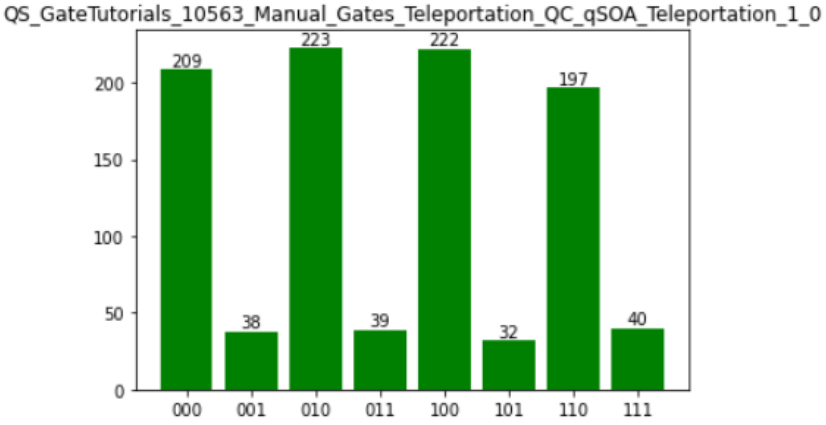
```
Device List: {'14': 'AMAZON BRAKET 25qbits Local Simulator', '2': 'QISKIT Local
Simulator', '1': 'Q# Local Simulator Framework'}
```

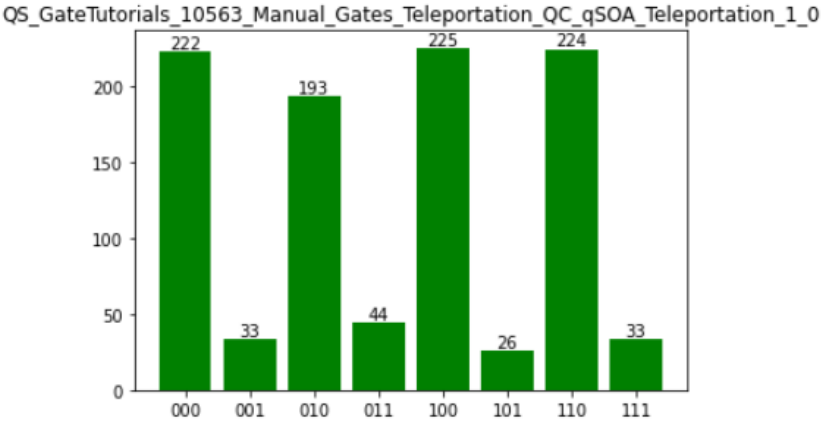
[13]:
```
## Run Quantum Gates Application
exe_ApplicationNames = ['Task_Amazonsim','Task_Qiskitsim','Task_QSharpsim']
exe_IdDevices = [14,2,1]

exe_Applications = [0] * len(exe_ApplicationNames)

for i in range(len(exe_ApplicationNames)):
    exe_Applications[i] = qsoa.runQuantumApplicationSync(exe_ApplicationNames[i],
      idSolution, FlowID, exe_IdDevices[i])
```

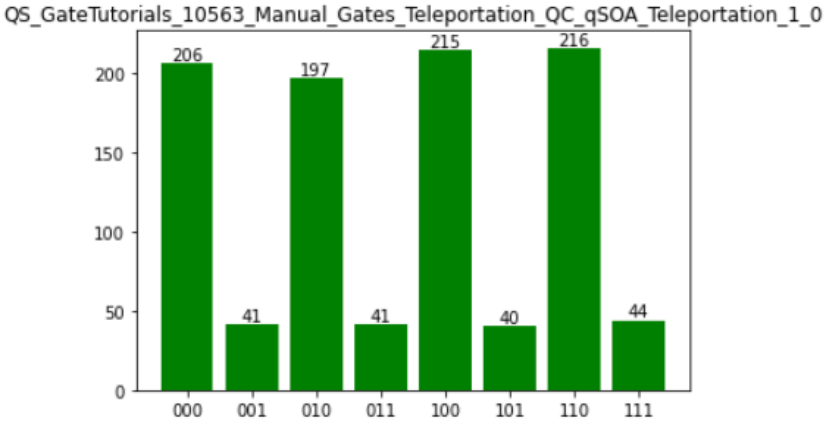
[14]:
```
# Get quantum execution response with execution token
restok_Executions = [0] * len(exe_ApplicationNames)
restok_Histograms = [0] * len(exe_ApplicationNames)

for i in range(len(exe_ApplicationNames)):
    restok_Executions[i] = qsoa.getQuantumExecutionResponse(exe_Applications[i].
 ↪getExecutionToken(), idSolution, FlowID)
    restok_Histograms[i] = restok_Executions[i].getHistogram()

print(restok_Histograms)
```

```
[{'QS_GateTutorials_12345_Manual_Gates_Teleportation_QC_qSOA_Teleportation_1_0':
{'010': 223, '100': 222, '101': 32, '001': 38, '000': 209, '110': 197, '111': 40,
'011': 39}},␣
 ↪{'QS_GateTutorials_12345_Manual_Gates_Teleportation_QC_qSOA_Teleportation_1_0':
{'110': 224, '010': 193, '001': 33, '000': 222, '100': 225, '101': 26, '011': 44,␣
 ↪'111': 33}},␣
 ↪{'QS_GateTutorials_12345_Manual_Gates_Teleportation_QC_qSOA_Teleportation_1_0':
{'111': 44, '000': 206, '101': 40, '110': 216, '100': 215, '010': 197, '001': 41,
'011': 41}}]
```

[15]:
```
# Circuit gate representation
for i in range(len(exe_ApplicationNames)):
    Gates_Representations = qsoa.representResults(restok_Executions[i])
```







2.5 Analyzing the results

We now proceed to analyze the results obtained in the simulation with just one device. It can be done analogously with the rest and the results should be equivalent.

The results shown in the histogram are:

Amazon Braket 25qbits Local Simulator			
Results	Alice	Bob	Outcome
000	00	0	219
001	00	1	24
010	01	0	212
011	01	1	33
100	10	0	224
101	10	1	40
110	11	0	215
111	11	1	33

That means that Bob receives a 0 output - with a probability of 0.87 -, and 1 - with probability 0.13.

If we recall that the state sent by Alice is:

$$\phi = \alpha\,|0\rangle + \beta\,|1\rangle \qquad Where\ \alpha = cos(\theta/2)\ \ \&\ \ \beta = sen(\theta/2)$$

and that we choose $\theta = \pi/4$, then, the probability of obtaining each state would be:

- $P(|0\rangle) = |cos(\pi/8)|^2 = 0.854$
- $P(|1\rangle) = |sin(\pi/8)|^2 = 0.146$

Therefore, as shown, comparing both the theoretical and simulated results we can confirm that the algorithm works.

References

[1] aQuantum, *QPath® Python SDK User Guide*. Available on QPath®.

[2] M. A. N. & I. L. Chuang, *Quantum Computation and Quantum Information,* Cambridge, 2010.