tp17_a72450_a74564

June 6, 2021

1 Trabalho Prático 2, Interação e Concorrência

1.1 Grupo 17

1.1.1 Maria Francisca Fernandes - A72450

1.1.2 Leandro Gonçalves - A74564

Consideramos, pelo enunciado, uma lista não ordenada com 8 elementos, sendo que os valores variam entre 0 e 7, já que para qualquer número natural, N,

$$Nmod8 = [0, 1, 2, 3, 4, 5, 6, 7]$$

Sendo, então, que os valores da lista variam entre 0 e 7, necessitamos de 3 qubits para os representar.

Para o presente trabalho, queremos localizar um elemento, s em específico, a partir do nosso $N = 17$, sendo que

$$s = Nmod8$$

$$s = 17mod8$$

$$s = 1$$

Pelo que estudamos anteriormente, pretendemos encontrar 001, então, pelo algoritmo de *Grover*

$$f(x) = 1, x = 001$$

$$f(y) = 0, y\ 001$$

0	1	0	0	0	0	0	0
000	001	010	011	100	101	110	111

[1]:
```
#grupo 17 (17 modulo 8 = 1)

w = 17 % 8
number_bin = bin(w)[2:].zfill(3)
number_qubits = len(number_bin)

print('representação de ', w, ' em binário: ', number_bin)
```

```
print('número de qubits: ', number_qubits)
```

```
representação de  1  em binário:  001
número de qubits:  3
```

1.2 1. Algoritmo de *Grover*

Vamos recorrer ao algoritmo de *Grover* que é constituido por 3 etapas: Inicialização, Oráculo e Amplificação. A primeira etapa, a Inicialização, consiste na aplicação da porta de *Hadamard* em todos os *qubits*. De seguida, o estado que procuramos será marcado pelo oráculo, através da negação da sua amplitude. Já na última etapa, iremos, tal como o nome da etapa indica, amplificar a amplitude do alvo que procuramos.

[2]:
```
from qiskit import *
from qiskit.tools.visualization import *

import matplotlib.pyplot as plt
%matplotlib inline
```

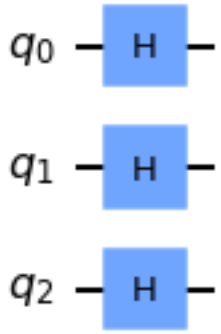
1.2.1 1.1. Inicialização

[3]:
```
# Começamos por criar um circuito quântico.

qc = QuantumCircuit(number_qubits)
```

Aplicamos as portas de *Hadamard* aos nossos qubits de forma a criar uma sobreposição quântica uniforme, isto para inicializarmos o sistema com a mesma amplitude em todos os estados possíveis.

[4]:
```
# Aplicar as portas de Hadamard aos 3 $qubits$ inicializados.
for q in range(number_qubits):
    qc.h(q)

qc.draw('mpl')

#qc.barrier()
```

[4]:



1.2.2 1.2. Oráculo

O Oráculo é o responsável por identificar a solução, no caso do nosso sistema o qubit $|001\rangle$ cuja fase faz uma rotação de π radianos, enquanto os outros estados se mantém inalterados. Com isto, a amplitude do estado $|001\rangle$ torna-se negativa, o que significa que a amplitude média foi reduzida.

No nosso caso, sendo que a nossa representação binária é 001, queremos marcar as posições 1 e 2, mas, também, para além de recorrer à porta de Hadamard, usamos, as portas Pauli-X e Controlled-X.

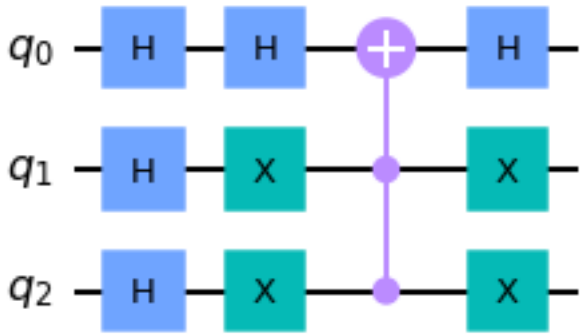
A porta *Pauli-X* atua num *qubit* singular, sendo o equivalente quântico da porta NOT. Não passa de uma simples rotação no eixo X e torna o *qubit* em $|1\rangle$ se este for $|0\rangle$ e vice-versa, por isto é também denominado de inversão do *qubit*.

A porta *Controlled-X*, ou CNOT, atua em 2 *qubits* e executa uma ação NOT no segundo *qubit*, apenas, quando o primeiro se apresenta como $|1\rangle$.

Definimos, então, o oráculo para marcar o estado $|001\rangle$, da seguinte forma:

[5]:
```
# Definir oráculo para o estado $|001\rangle$

qc.h(0)
qc.x(1)
qc.x(2)
qc.ccx(2,1,0)
qc.h(0)
qc.x(1)
qc.x(2)

qc.draw('mpl')

#qc.barrier()
```

[5]:



[6]:
```
backend_vector = Aer.get_backend("statevector_simulator")
```

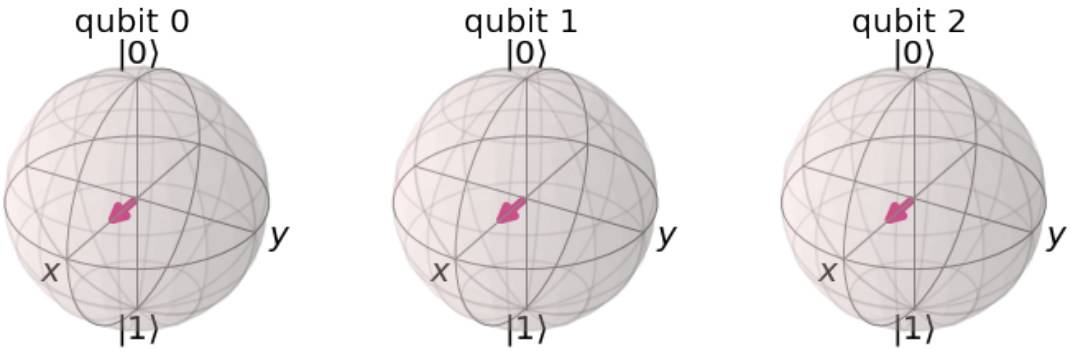
[7]:
```
result = execute(qc, backend_vector).result()
psi = result.get_statevector(qc)
```

[8]:
```
print(psi)
```

```
[ 0.35355339-4.32978028e-17j -0.35355339+4.32978028e-17j
  0.35355339+0.00000000e+00j  0.35355339+0.00000000e+00j
  0.35355339+0.00000000e+00j  0.35355339+0.00000000e+00j
  0.35355339+0.00000000e+00j  0.35355339+0.00000000e+00j]
```

[9]:
```
plot_bloch_multivector(psi)
```

[9]:



1.2.3 1.3. Amplificação

Durante esta etapa do algoritmo, para além de voltar a inverter o qubit $|001\rangle$, também aumenta sua amplitude.

Em relação ao algoritmo de *Grover* e, também, na documentação, esta etapa é identificada como *diffusion transform*.

[10]:
```
# Aplicar Hadamard gates aos $qubits$ inicializados.
for q in range(number_qubits):
    qc.h(q)

# Aplicar X gates aos $qubits$ inicializados.
for q in range(number_qubits):
    qc.x(q)

# Aplicar Z gates aos $qubits$ inicializados.
qc.h(0)
qc.ccx(2,1,0)
qc.h(0)

# Aplicar Hadamard & X gates aos $qubits$ inicializados.
for q in range(number_qubits):
    qc.x(q)
    qc.h(q)
```

1.2.4 1.4. Repetição

De modo a otimizar os resultados obtidos, teremos de repetir os dois passos anteriores. Aplicá-los $\sqrt{N}$ vezes, para sermos exatos, sendo que N é igual ao resultado da função a seguir apresentado.

[11]:
```
import math as m

times = round(m.sqrt(2 ** number_qubits))
print("Número de vezes a executar: ", times)
```

```
Número de vezes a executar:  3
```

Pelos resultados das diferentes execuções, percebemos que apesar de, teoricamente, devermos executar todo o processo 3 vezes, o número ótimo de execuções é de 2, ou seja, apenas uma repetição.

Sem repetição, i.e. uma execução do oráculo e amplificação, 0.775

Uma repetição, i.e. duas execuções do oráculo e amplificação, > 0.930

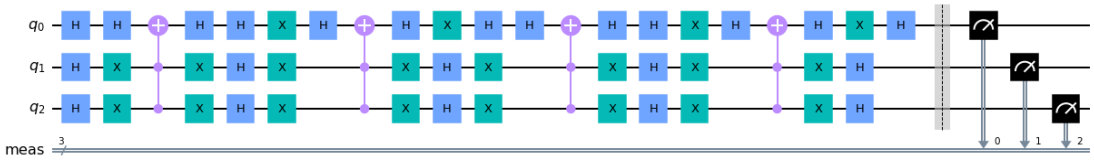
Duas repetições, i.e. três execuções do oráculo e amplificação, 0.316

Da mesma forma, podemos ver a diferença nos estados dos *qubits* a partir dos seus *statevectors*, sendo que todos os vetores estão a indicar quase que um estado $|111\rangle$.

[12]:
```
# Repetir passos anteriores (3-1=)2 vezes
qc.h(0)
```

```
qc.x(1)
qc.x(2)
qc.ccx(2,1,0)
qc.h(0)
qc.x(1)
qc.x(2)

#qc.barrier()

# Aplicar Hadamard gates aos $qubits$ inicializados.
for q in range(number_qubits):
    qc.h(q)
    qc.x(q)

# Aplicar porta Z controlada
qc.h(0)
qc.ccx(2,1,0)
qc.h(0)

# Aplicar portas X e Hadamard aos qubits.
for q in range(number_qubits):
    qc.x(q)
    qc.h(q)

# 'Medir' os qubits para aceder aos seus estados $|001\rangle$
qc.measure_all()
qc.draw(output = 'mpl')
```

[12]:



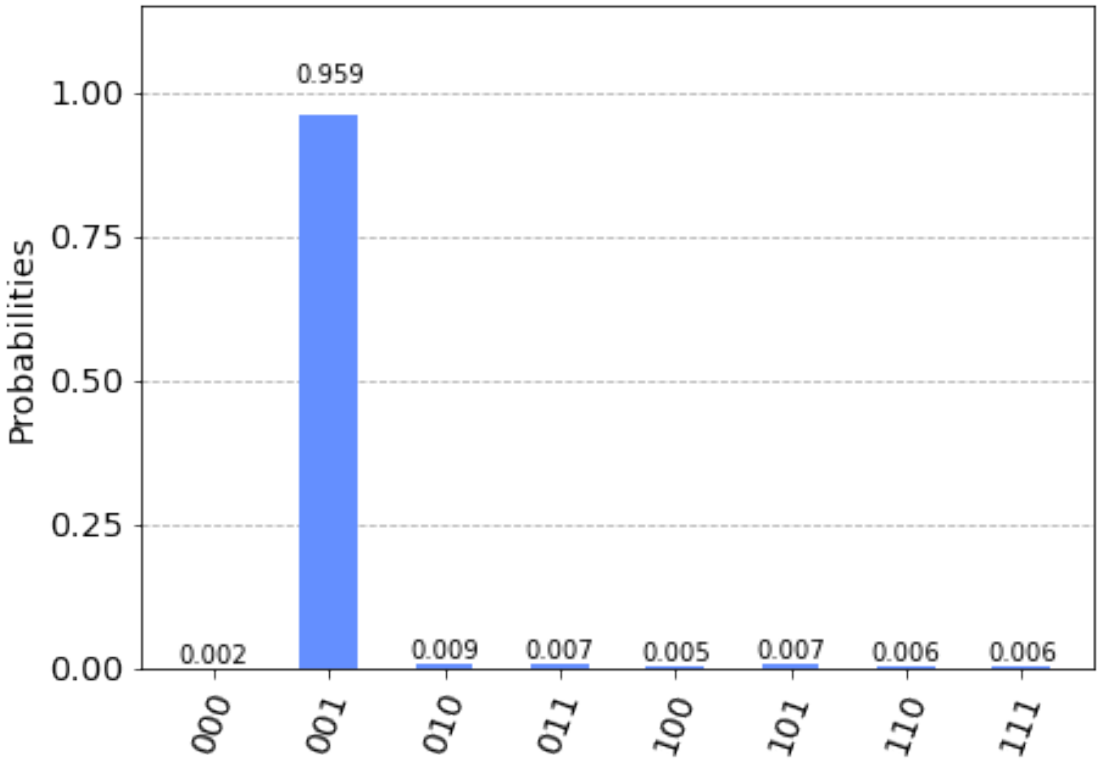
1.3 2. Simulação de ruído com Aer

Utilizamos os modelos de simulação de ruído do Aer para tentar prever qual será a melhor otimização para o nosso sistema.

[13]:
```
backend = Aer.get_backend('qasm_simulator')

shots = 1024

# Executa a lista de circuitos quânticos no backend e guarda o resultado
results = execute(qc, backend = backend, shots = shots).result()
```

```
# Obter os dados para o histograma
answer = results.get_counts(qc)

# Desenhar o histograma
plot_histogram(answer)
```

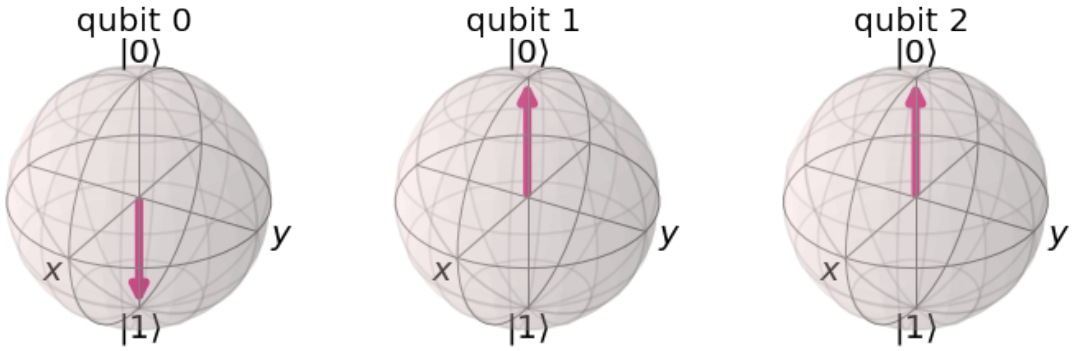
[13]:



[14]:

```
backend_vector = Aer.get_backend("statevector_simulator")

result = execute(qc, backend_vector).result()
psi = result.get_statevector(qc)

plot_bloch_multivector(psi)
```

[14]:



Com a ferramenta *statevector* é-nos possível visualizar o estado do sistema, no nosso caso, a representação da nossa solução e oráculo $|001\rangle$.

[15]:
```
qc.depth()
```

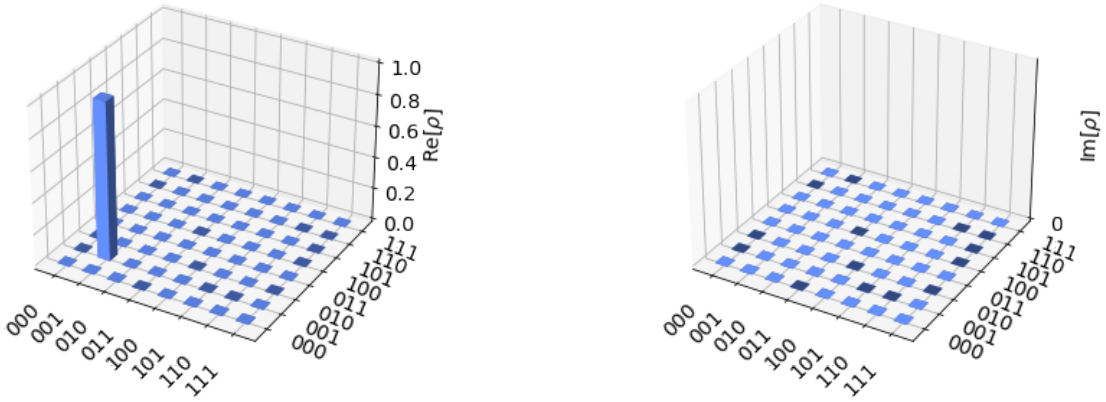
[15]:
```
22
```

[16]:
```
psi.real
```

[16]:
```
array([ 0.,  1.,  0., -0.,  0., -0.,  0.,  0.])
```

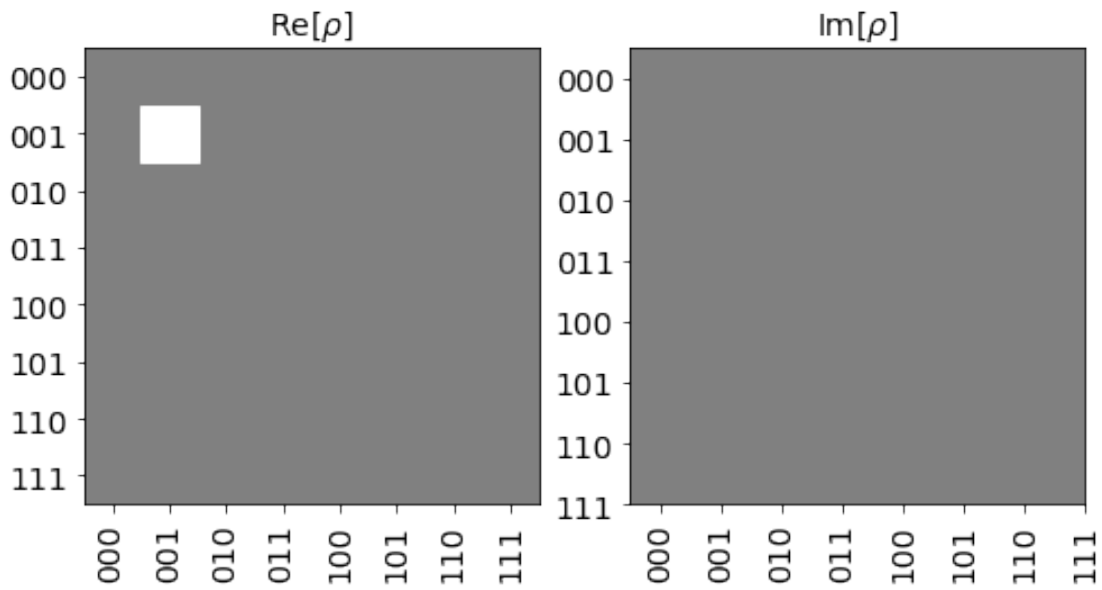
[17]:
```
plot_state_city(psi)
```

[17]:



[18]:
```
plot_state_hinton(psi)
```

[18]:



1.4 3. IBM

[19]:
```
provider = IBMQ.load_account()
provider.backends()
```

[19]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmqx2') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_16_melbourne') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_armonk') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_athens') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_santiago') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_extended_stabilizer') from IBMQ(hub='ibm-q',
```

```
group='open', project='main')>,
 <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open',
project='main')>]
```

[20]:
```
# Backend overview
import qiskit.tools.jupyter

%qiskit_backend_overview
```

```
VBox(children=(HTML(value="<h2 style ='color:#ffffff; background-color:#000000;
 ↪padding-top: 1%; padding-bottom…
```

[21]:
```
from qiskit.tools.monitor import backend_overview, backend_monitor

backend_overview()
```

```
ibmq_manila
-----------
Num. Qubits:  5
Pending Jobs: 6
Least busy:   False
Operational:  True
Avg. T1:      151.0
Avg. T2:      67.0

ibmq_quito
----------
Num. Qubits:  5
Pending Jobs: 10
Least busy:   False
Operational:  True
Avg. T1:      75.2
Avg. T2:      73.2

ibmq_belem
----------
Num. Qubits:  5
Pending Jobs: 0
Least busy:   True
Operational:  True
Avg. T1:      79.3
Avg. T2:      91.6

ibmq_lima
---------
Num. Qubits:  5
Pending Jobs: 6
Least busy:   False
Operational:  True
Avg. T1:      69.2
Avg. T2:      64.9

ibmq_santiago
-------------
Num. Qubits:  5
Pending Jobs: 6
Least busy:   False
Operational:  True
Avg. T1:      136.2
Avg. T2:      136.4

ibmq_athens
-----------
Num. Qubits:  5
Pending Jobs: 4
Least busy:   False
Operational:  True
Avg. T1:      95.9
Avg. T2:      120.6

ibmq_armonk
-----------
Num. Qubits:  1
Pending Jobs: 25
Least busy:   False
Operational:  True
Avg. T1:      124.6
Avg. T2:      217.3

ibmq_16_melbourne
-----------------
Num. Qubits:  15
Pending Jobs: 1
Least busy:   False
Operational:  True
Avg. T1:      57.5
Avg. T2:      56.2

ibmqx2
------
Num. Qubits:  5
Pending Jobs: 6
Least busy:   False
Operational:  True
Avg. T1:      54.1
Avg. T2:      40.5
```

[22]:
```
backend_device = provider.get_backend('ibmqx2')
print("Running on: ", backend_device)
```

```
Running on:  ibmqx2
```

[23]:
```
backend_monitor(backend_device)
```

```
ibmqx2
======
Configuration
-------------
    n_qubits: 5
    operational: True
    status_msg: active
    pending_jobs: 4
    backend_version: 2.3.6
    basis_gates: ['id', 'rz', 'sx', 'x', 'cx', 'reset']
    local: False
    simulator: False
    description: 5 qubit device Yorktown
    allow_q_object: True
    sample_name: family: Canary, revision: 1
    max_shots: 8192
    dtm: 0.2222222222222222
    u_channel_lo: [[{'q': 1, 'scale': (1+0j)}], [{'q': 2, 'scale': (1+0j)}],
[{'q': 0, 'scale': (1+0j)}], [{'q': 2, 'scale': (1+0j)}], [{'q': 0, 'scale':
(1+0j)}], [{'q': 1, 'scale': (1+0j)}], [{'q': 3, 'scale': (1+0j)}], [{'q': 4,
'scale': (1+0j)}], [{'q': 2, 'scale': (1+0j)}], [{'q': 4, 'scale': (1+0j)}],
[{'q': 2, 'scale': (1+0j)}], [{'q': 3, 'scale': (1+0j)}]]
    qubit_channel_mapping: [['d0', 'u1', 'u4', 'u2', 'u0', 'm0'], ['d1', 'm1',
'u3', 'u2', 'u0', 'u5'], ['u10', 'u4', 'u1', 'u6', 'u3', 'u7', 'u8', 'u5', 'd2',
'm2'], ['m3', 'd3', 'u6', 'u8', 'u9', 'u11'], ['u7', 'u9', 'm4', 'u11', 'u10',
'd4']]
    coupling_map: [[0, 1], [0, 2], [1, 0], [1, 2], [2, 0], [2, 1], [2, 3], [2,
4], [3, 2], [3, 4], [4, 2], [4, 3]]
    qubit_lo_range: [[4.782332600983115e+18, 5.782332600983115e+18],
[4.74750723639363e+18, 5.74750723639363e+18], [4.5333809552439096e+18,
5.53338095524391e+18], [4.791966759946752e+18, 5.791966759946752e+18],
[4.5784444616408463e+18, 5.578444461640846e+18]]
    discriminators: ['linear_discriminator', 'hw_centroid',
'quadratic_discriminator']
    acquisition_latency: []
    dt: 0.2222222222222222
    online_date: 2017-01-24 05:00:00+00:00
```

```
    supported_instructions: ['u1', 'reset', 'rz', 'sx', 'cx', 'u3', 'u2', 'x',
'play', 'shiftf', 'measure', 'id', 'delay', 'acquire', 'setf']
    conditional: False
    multi_meas_enabled: True
    allow_object_storage: True
    pulse_num_qubits: 3
    hamiltonian: {'description': 'Qubits are modeled as Duffing oscillators. In
this case, the system includes higher energy states, i.e. not just |0> and |1>.
The Pauli operators are generalized via the following set of
transformations:\n\n$(\\mathbb{I}-\\sigma_{i}^z)/2 \\rightarrow O_i \\equiv
b^\\dagger_{i} b_{i}$,\n\n$\\sigma_{+} \\rightarrow b^\\dagger$,\n\n$\\sigma_{-}
\\rightarrow b$,\n\n$\\sigma_{i}^X \\rightarrow b^\\dagger_{i} +
b_{i}$.\n\nQubits are coupled through resonator buses. The provided Hamiltonian
has been projected into the zero excitation subspace of the resonator buses
leading to an effective qubit-qubit flip-flop interaction. The qubit resonance
frequencies in the Hamiltonian are the cavity dressed frequencies and not
exactly what is returned by the backend defaults, which also includes the
dressing due to the qubit-qubit interactions.\n\nQuantities are returned in
angular frequencies, with units 2*pi*GHz.\n\nWARNING: Currently not all system
Hamiltonian information is available to the public, missing values have been
replaced with 0.\n', 'h_latex': '\\begin{align} \\mathcal{H}/\\hbar = & \\sum_{i
=0}^{4}\\left(\\frac{\\omega_{q,i}}{2}(\\mathbb{I}-\\sigma_i^{z})+\\frac{\\Delta
_{i}}{2}(O_i^2-O_i)+\\Omega_{d,i}D_i(t)\\sigma_i^{X}\\right) \\\\ & +
J_{0,1}(\\sigma_{0}^{+}\\sigma_{1}^{-}+\\sigma_{0}^{-}\\sigma_{1}^{+}) +
J_{1,2}(\\sigma_{1}^{+}\\sigma_{2}^{-}+\\sigma_{1}^{-}\\sigma_{2}^{+}) +
J_{2,3}(\\sigma_{2}^{+}\\sigma_{3}^{-}+\\sigma_{2}^{-}\\sigma_{3}^{+}) +
J_{2,4}(\\sigma_{2}^{+}\\sigma_{4}^{-}+\\sigma_{2}^{-}\\sigma_{4}^{+}) \\\\ & +
J_{3,4}(\\sigma_{3}^{+}\\sigma_{4}^{-}+\\sigma_{3}^{-}\\sigma_{4}^{+}) +
J_{0,2}(\\sigma_{0}^{+}\\sigma_{2}^{-}+\\sigma_{0}^{-}\\sigma_{2}^{+}) \\\\ & +
\\Omega_{d,0}(U_{0}^{(0,1)}(t)+U_{1}^{(0,2)}(t))\\sigma_{0}^{X} +
\\Omega_{d,1}(U_{2}^{(1,0)}(t)+U_{3}^{(1,2)}(t))\\sigma_{1}^{X} \\\\ & + \\Omega
_{d,2}(U_{6}^{(2,3)}(t)+U_{5}^{(2,1)}(t)+U_{7}^{(2,4)}(t)+U_{4}^{(2,0)}(t))\\sig
ma_{2}^{X} + \\Omega_{d,3}(U_{8}^{(3,2)}(t)+U_{9}^{(3,4)}(t))\\sigma_{3}^{X}
\\\\ & + \\Omega_{d,4}(U_{11}^{(4,3)}(t)+U_{10}^{(4,2)}(t))\\sigma_{4}^{X} \\\\
\\end{align}', 'h_str': ['_SUM[i,0,4,wq{i}/2*(I{i}-Z{i})]',
'_SUM[i,0,4,delta{i}/2*O{i}*O{i}]', '_SUM[i,0,4,-delta{i}/2*O{i}]',
'_SUM[i,0,4,omegad{i}*X{i}||D{i}]', 'jq0q1*Sp0*Sm1', 'jq0q1*Sm0*Sp1',
'jq1q2*Sp1*Sm2', 'jq1q2*Sm1*Sp2', 'jq2q3*Sp2*Sm3', 'jq2q3*Sm2*Sp3',
'jq2q4*Sp2*Sm4', 'jq2q4*Sm2*Sp4', 'jq3q4*Sp3*Sm4', 'jq3q4*Sm3*Sp4',
'jq0q2*Sp0*Sm2', 'jq0q2*Sm0*Sp2', 'omegad1*X0||U0', 'omegad2*X0||U1',
'omegad0*X1||U2', 'omegad2*X1||U3', 'omegad3*X2||U6', 'omegad1*X2||U5',
'omegad4*X2||U7', 'omegad0*X2||U4', 'omegad2*X3||U8', 'omegad4*X3||U9',
'omegad3*X4||U11', 'omegad2*X4||U10'], 'osc': {}, 'qub': {'0': 3, '1': 3, '2':
3, '3': 3, '4': 3}, 'vars': {'delta0': -2.078515989791283, 'delta1':
-2.076140198460304, 'delta2': -2.3632544243955147, 'delta3': -2.071793501418874,
'delta4': -2.092928090491978, 'jq0q1': 0.011968734726718661, 'jq0q2':
0.01143731530238952, 'jq1q2': 0.0077637124867075335, 'jq2q3':
0.011434086611531646, 'jq2q4': 0.011382837107272241, 'jq3q4':
```

0.012622615872488169, 'omegad0': 0.40095597097840147, 'omegad1':
0.3610866164540819, 'omegad2': 0.3199705799902724, 'omegad3':
0.2924994191502375, 'omegad4': 0.4133318994824383, 'wq0': 33.18987458613284,
'wq1': 32.971060367027015, 'wq2': 31.62566526342609, 'wq3': 33.25040779218019,
'wq4': 31.90880762470931}}
    meas_map: [[0, 1, 2, 3, 4]]
    meas_levels: [1, 2]
    parametric_pulses: ['gaussian', 'gaussian_square', 'drag', 'constant']
    uchannels_enabled: True
    max_experiments: 75
    meas_lo_range: [[6.030433052e+18, 7.030433052e+18], [5.981651108e+18,
6.981651108e+18], [5.93654928e+18, 6.93654928e+18], [6.078886966e+18,
7.078886966e+18], [6.030066921e+18, 7.030066921e+18]]
    url: None
    rep_times: [0.001]
    input_allowed: ['job']
    processor_type: {'family': 'Canary', 'revision': 1}
    default_rep_delay: 250.0
    quantum_volume: 8
    n_uchannels: 12
    rep_delay_range: [0.0, 500.0]
    memory: True
    channels: {'acquire0': {'operates': {'qubits': [0]}, 'purpose': 'acquire',
'type': 'acquire'}, 'acquire1': {'operates': {'qubits': [1]}, 'purpose':
'acquire', 'type': 'acquire'}, 'acquire2': {'operates': {'qubits': [2]},
'purpose': 'acquire', 'type': 'acquire'}, 'acquire3': {'operates': {'qubits':
[3]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire4': {'operates':
{'qubits': [4]}, 'purpose': 'acquire', 'type': 'acquire'}, 'd0': {'operates':
{'qubits': [0]}, 'purpose': 'drive', 'type': 'drive'}, 'd1': {'operates':
{'qubits': [1]}, 'purpose': 'drive', 'type': 'drive'}, 'd2': {'operates':
{'qubits': [2]}, 'purpose': 'drive', 'type': 'drive'}, 'd3': {'operates':
{'qubits': [3]}, 'purpose': 'drive', 'type': 'drive'}, 'd4': {'operates':
{'qubits': [4]}, 'purpose': 'drive', 'type': 'drive'}, 'm0': {'operates':
{'qubits': [0]}, 'purpose': 'measure', 'type': 'measure'}, 'm1': {'operates':
{'qubits': [1]}, 'purpose': 'measure', 'type': 'measure'}, 'm2': {'operates':
{'qubits': [2]}, 'purpose': 'measure', 'type': 'measure'}, 'm3': {'operates':
{'qubits': [3]}, 'purpose': 'measure', 'type': 'measure'}, 'm4': {'operates':
{'qubits': [4]}, 'purpose': 'measure', 'type': 'measure'}, 'u0': {'operates':
{'qubits': [0, 1]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u1':
{'operates': {'qubits': [0, 2]}, 'purpose': 'cross-resonance', 'type':
'control'}, 'u10': {'operates': {'qubits': [4, 2]}, 'purpose': 'cross-
resonance', 'type': 'control'}, 'u11': {'operates': {'qubits': [4, 3]},
'purpose': 'cross-resonance', 'type': 'control'}, 'u2': {'operates': {'qubits':
[1, 0]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u3': {'operates':
{'qubits': [1, 2]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u4':
{'operates': {'qubits': [2, 0]}, 'purpose': 'cross-resonance', 'type':
'control'}, 'u5': {'operates': {'qubits': [2, 1]}, 'purpose': 'cross-resonance',
'type': 'control'}, 'u6': {'operates': {'qubits': [2, 3]}, 'purpose': 'cross-

```
resonance', 'type': 'control'}, 'u7': {'operates': {'qubits': [2, 4]},
'purpose': 'cross-resonance', 'type': 'control'}, 'u8': {'operates': {'qubits':
[3, 2]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u9': {'operates':
{'qubits': [3, 4]}, 'purpose': 'cross-resonance', 'type': 'control'}}
    dynamic_reprate_enabled: True
    n_registers: 1
    conditional_latency: []
    open_pulse: False
    meas_kernels: ['hw_boxcar']
    pulse_num_channels: 9
    backend_name: ibmqx2
    credits_required: True

Qubits [Name / Freq / T1 / T2 / RZ err / SX err / X err / Readout err]
--------------------------------------------------------------------
    Q0 / 5.28233 GHz / 52.63547 us / 24.26255 us / 0.00000 / 0.00102 / 0.00102 /
0.10580
    Q1 / 5.24751 GHz / 43.40835 us / 26.96789 us / 0.00000 / 0.00155 / 0.00155 /
0.02960
    Q2 / 5.03338 GHz / 59.29261 us / 70.52557 us / 0.00000 / 0.00077 / 0.00077 /
0.09710
    Q3 / 5.29197 GHz / 54.64515 us / 28.86973 us / 0.00000 / 0.00069 / 0.00069 /
0.03860
    Q4 / 5.07844 GHz / 60.42556 us / 51.65079 us / 0.00000 / 0.00048 / 0.00048 /
0.05180

Multi-Qubit Gates [Name / Type / Gate Error]
--------------------------------------------
    cx4_2 / cx / 0.01610
    cx2_4 / cx / 0.01610
    cx3_4 / cx / 0.01745
    cx4_3 / cx / 0.01745
    cx3_2 / cx / 0.02120
    cx2_3 / cx / 0.02120
    cx1_2 / cx / 0.02573
    cx2_1 / cx / 0.02573
    cx0_2 / cx / 0.02753
    cx2_0 / cx / 0.02753
    cx0_1 / cx / 0.01793
    cx1_0 / cx / 0.01793
```

[24]:
```
%qiskit_job_watcher
```

```
Accordion(children=(VBox(layout=Layout(max_width='710px', min_width='710px')),),␣
↪layout=Layout(max_height='500…
```

```
<IPython.core.display.Javascript object>
```

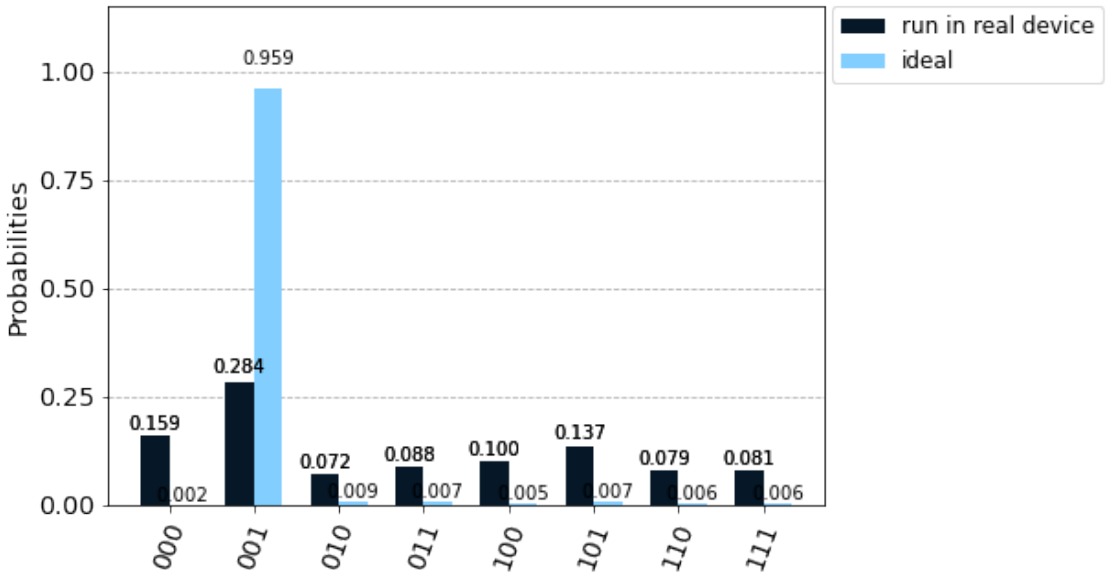
[25]:
```
job_r = execute(qc, backend_device, shots = shots)

jobID_r = job_r.job_id()

print('JOB ID: {}'.format(jobID_r))
```

```
JOB ID: 60bba7f336b2be1fc324fdc5
```

[26]:
```
# ibmqx2 runs the oracle once
job_get = backend_device.retrieve_job(jobID_r)

job_get.error_message()

result_r = job_get.result()
counts_run = result_r.get_counts(qc)
```

[27]:
```
plot_histogram([counts_run, answer], legend=[ 'run in real device', 'ideal'],␣
↪color=['#061727','#82cfff'])
```

[27]:



1.5 4. IGNIS

Iremos recorrer ao módulo Ignis para o tratamento de erros. A sua *framework* permite a caracterização de medição utilizando parâmetros de ruido, interação do dispositivo e erros nas gates de controlo. E ainda que seja feita a mitigação de erros de medição.

Começamos por fazer uso da calibração de medição para mitigar erros de medição

[28]:
```
# Import measurement calibration functions
from qiskit.ignis.mitigation.measurement import (complete_meas_cal,␣
 ↪tensored_meas_cal,
                                                  CompleteMeasFitter,␣
 ↪TensoredMeasFitter)
```

Matrizes de Calibração Começamos por usar uma matriz de calibração de modo a que seja possível obter uma boa indicação do estado do qubit. Para isso, queremos gerar uma matriz de calibração para os 3 qubits existentes.

Como temos 3 qubits, são necessários $2^3 = 8$ circuitos de calibração.

Para obtermos a lista de *QuantumCircuit* que contem os circuitos de calibração, usamos a função *complete_means_cal* que usa os parametros qubit_list, qr (registo quântico), cr(registo classico) e ciclabel (string adicionada ao inicio do nome em circuitos para a sua identificação).

[29]:
```
# Geração dos circuitos de calibração
qr = QuantumRegister(number_qubits)

# meas_calibs:
# lista de dos objetos do circuito quântico que contêm os circuitos de␣
 ↪calibração
# state_labels:
# estados de calibração
meas_calibs, state_labels = complete_meas_cal(qubit_list=[0,1,2], qr=qr,␣
 ↪circlabel='mcal')
```

[30]:
```
state_labels
```

[30]:
```
['000', '001', '010', '011', '100', '101', '110', '111']
```

Computação da matriz de calibração Caso não seja aplicado qualquer ruído, a matriz de calibração expectável será a matriz identidade 8x8. Como a matriz foi computada com recurso a um dispositivo quântico, existirá ruido no circuito.

Executando o circuito de calibração.

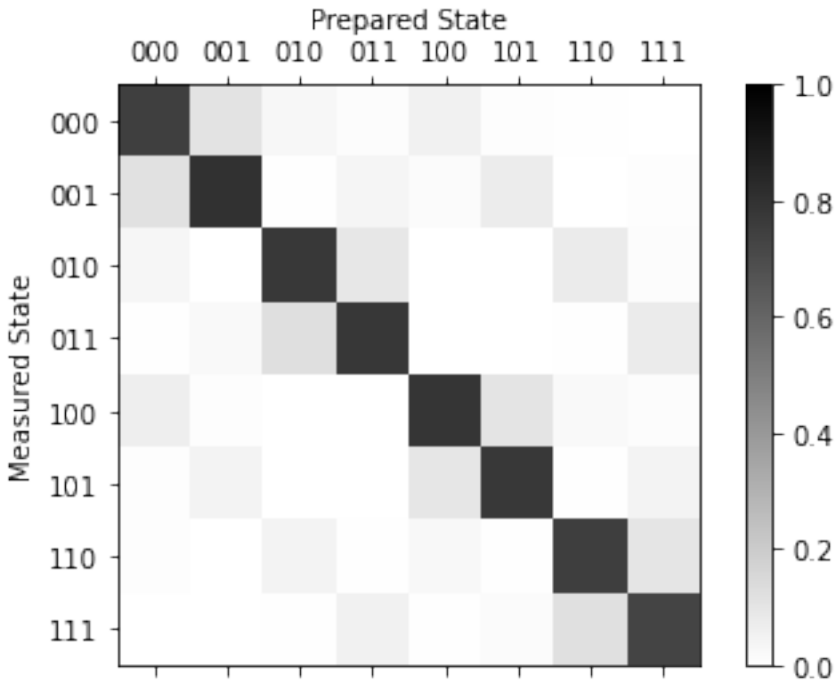
[31]:
```
job_ignis = execute(meas_calibs, backend=backend_device, shots=shots)

jobID_run_ignis = job_ignis.job_id()

print('JOB ID: {}'.format(jobID_run_ignis))
```

```
JOB ID: 60bba8295f4eaadc00daecb9
```

[32]:
```
job_get=backend_device.retrieve_job(jobID_run_ignis)
cal_results = job_get.result()
```

Após a execução, é calculada a matriz de calibração

[33]:
```
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel='mcal')

# Gráfico da matriz de calibração
meas_fitter.plot_calibration()
```



Análise de Resultados Para ser possível analisar os resultados, é calculada a fidelidade da medição efetuada anteriormente. São os elementos da diagonal da matriz que devolvem as probabilidade de medir um estado dada a sua preparação. Assim o traço obtido é a fidelidade média.

[34]:
```
#Qual é a fidelidade da medição?
print("Average Measurement Fidelity: %f" % meas_fitter.readout_fidelity())
```

```
Average Measurement Fidelity: 0.771118
```

Aplicação da calibração Para calcular o resultados com e sem a mitigação de erros, vamos aplicar a matriz de calibração.

Os nossos dados brutos serão **result_r**. Pode ainda ser aplicado um filtro baseado na matriz de calibração de modo auq se obtenham contagens mitigadas.

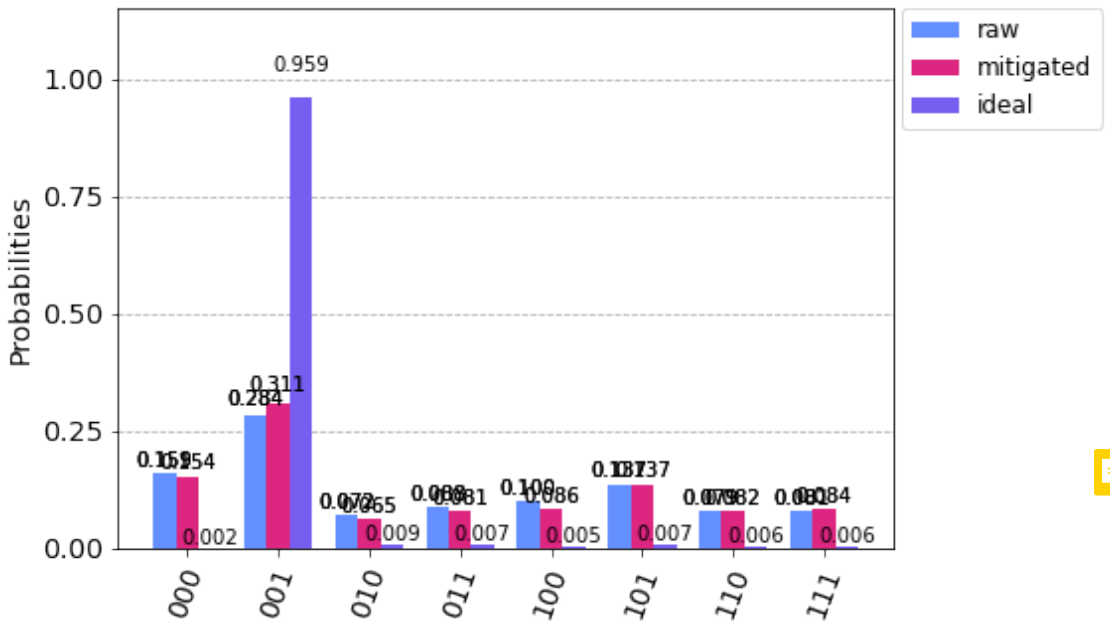
[35]:
```
# Objeto do filtro
meas_filter = meas_fitter.filter
```

```
# Resultados com mitigação
mitigated_results = meas_filter.apply(result_r)
mitigated_counts = mitigated_results.get_counts()
```

Gráfico com resultados com e sem mitigação:

[36]:
```
plot_histogram([counts_run, mitigated_counts, answer], legend=['raw',␣
↪'mitigated', 'ideal'])
```

[36]:



Por fim, acrescentamos apenas a informação relativamente ao software utilizado.

[37]:
```
%qiskit_version_table
```

```
<IPython.core.display.HTML object>
```

[ ]:
```

```