REPORT

Introduction

Quantum computation has been an exciting field of research especially since the past few years. Banking on the postulates of Quantum Mechanics, especially Superposition and Interference, it is thought that quantum computers might as well help in tackling the paucity caused by the gradual saturation of Moore's Law that has hampered scalability of Integrated Circuits and chips in the classical computing industry. This is precisely where fabrication, design and study of Quantum Hardware and Qubits is important, so that we understand the hardware associated challenges concerning the architecture of large scale quantum devices.
Superconducting qubits have formed the basis of study of prominent platforms for constructing multi-qubit quantum processors where information is stored in quantum degrees of freedom of nanofabricated, anharmonic oscillators constructed from superconducting circuit elements.

Size and scalability:
Superconducting qubits are macroscopic in size and are lithographically defined. They can be designed to exhibit atom-like energy spectra with certain required properties bestowed in terms of transition frequencies, anharmonicity and complexity.

Why do we need Quantum Engineering/ Fabrication ?
The motive being, the quantum states necessary for quantum computing or other experimental schemes are sensitive to stray fields and thermal noise.
Important aspects of qubit design include preserving the quantum properties of a system and coherence.
To help mitigate this problem of reducing sensitivity to fluctuations in local electric charge density , the transmon qubit was conceived. It has helped maintain the strong trend in increasing coherence times for superconducting qubit designs.
Due to the macroscopic nature of quantum phenomena in superconductivity, the properties of qubits can be engineered / designed to meet certain requirements .

A fundamental element in any superconducting qubit design is the Josephson junction, consisting of two superconducting electrodes separated by a thin insulating barrier.

In a Josephson junction, the separation between two superconducting electrodes is sufficiently small to create a weak coupling between them. The macroscopic superconducting wavefunctions of the electrodes overlap, leading to the tunnelling of Cooper pairs across the barrier
When cooled down to sufficiently low temperatures, where $k_BT \ll E_J$, E_C, Josephson junction circuits exhibit quantum properties.

The single Cooper pair box (CPB) couples a small superconducting island via a Josephson junction to a gate electrode.

Circuit diagram of a cooper pair box :

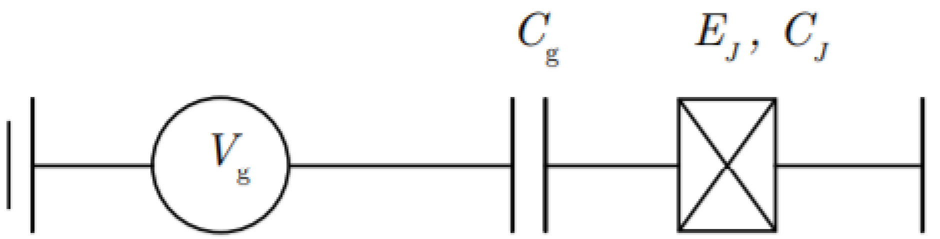


Image source: [Link]

Circuit diagram of a transmon qubit, here an extra capacitance has been added between the ground and the island.

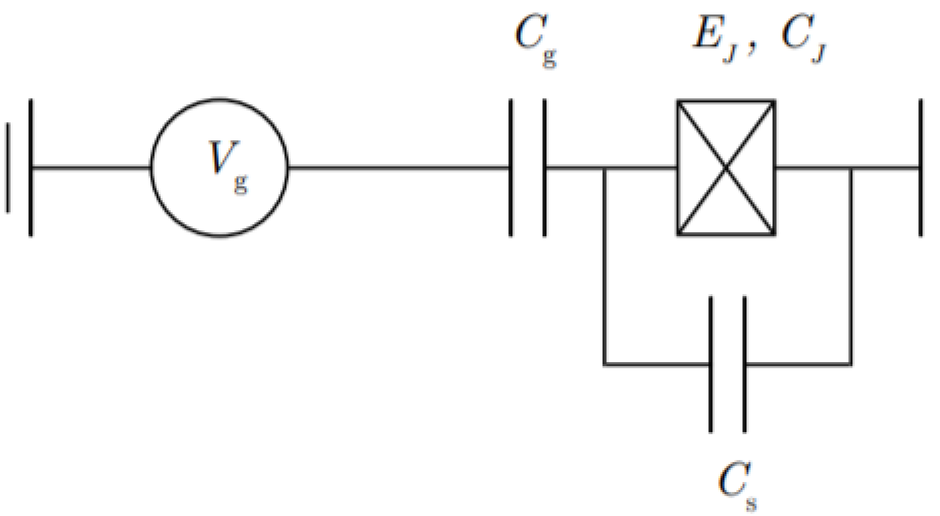


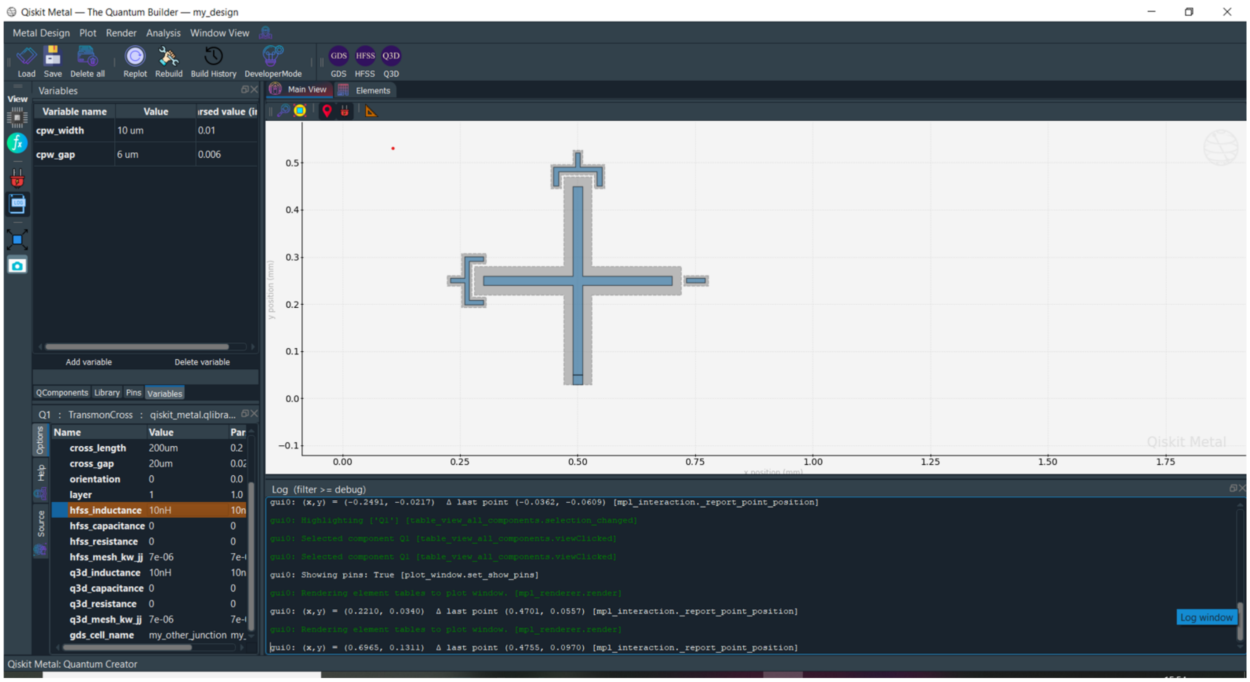
Image source: [link]

Hahn echoes, BCS Theory, Rabi Oscillations, etc that were some of the topics in the review paper were briefly studied.

To study a few aspects of Transmon Qubits, QHOs, Superconducting Qubits, I implemented a few codes which can e found in my GitHub repository : [GitHub]

QuTiP is open-source software for simulating the dynamics of open quantum systems. The program attempts to show the difference in energy levels of the QHO and the transmon by calculating them from their hamiltonian using QuTip .

Qiskit Metal:
Qiskit Metal is an open-source framework (and library) for the design of superconducting quantum chips and devices.
Using Qiskit Metal, I attempted to create the 2D design of a simple transmon cross chip. It has a junction and 3 connectors on the remaining arms.



Randomized benchmarking is a simple and efficient protocol for measuring an average error rate of a quantum information processor (QIP), and is among the most commonly used experimental methods for characterizing QIPs.
The performances of three quantum processing units namely IBMQ Belem, Santiago and Athens were compared using this procedure.

Comparing the transmon and the Quantum Harmonic Oscillator

It is known to us from theory that the Quantum Harmonic Oscillator has evenly spaced energy levels while that is not so in the case of a transmon. Some amount of anharmonicity is required for the two-level approximation of the qubit to remain valid. The transmon qubit has a design similar to the cooper pair box (But in the latter case, the qubits encoded as charge states are particularly sensitive to charge noise).

In this code, I attempt to show the difference in energy levels of the QHO and the transmon by calculating them from their hamiltonian using QuTip which is basically the standard Quantum Toolbox in Python .

Date: 22-06-2021

In this code, we predominantly make use of the Transmon Hamiltonian

$$\hat{H}_{\text{tr}} = 4E_c\hat{n}^2 - E_J\cos\hat{\phi},$$

In [1]:

```
import numpy as np
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *

# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()
```

In [5]:

```
pip install qutip
```

```
Collecting qutip
  Downloading qutip-4.6.2-cp38-cp38-manylinux2010_x86_64.whl (16.0 MB)
     |████████████████████████████████| 16.0 MB 61 kB/s s eta 0:00:01          | 2.3 MB 11.0 MB/s eta 0:00:02
Requirement already satisfied: scipy>=1.0 in /opt/conda/lib/python3.8/site-packages (from qutip) (1.6.1)
Requirement already satisfied: numpy>=1.16.6 in /opt/conda/lib/python3.8/site-packages (from qutip) (1.20.1)
Requirement already satisfied: packaging in /opt/conda/lib/python3.8/site-packages (from qutip) (20.9)
Requirement already satisfied: pyparsing>=2.0.2 in /opt/conda/lib/python3.8/site-packages (from packaging->qutip) (2.4.7)
Installing collected packages: qutip
Successfully installed qutip-4.6.2
Note: you may need to restart the kernel to use updated packages.
```

In [14]:

```
import matplotlib.pyplot as plt
#E_J denotes the Josephson energy, w denotes Omega which is (8*EcEj)^1/2 - Ec where -Ec refers to the anharmoncity or Delta

E_J = 20e9
w = 5e9
Delta = -300e6

N_phis = 101
phis = np.linspace(-np.pi,np.pi,N_phis)
mid_idx = int((N_phis+1)/2)

# PE_QHO is denotes the potential energy of the Quantum Harmonic Oscillator
#PE_transmon denotes the Potential Energy of the transmon
#We apply the standard formulae to obtain these values
PE_QHO = 0.5*E_J*phis**2
PE_QHO = PE_QHO/w
PE_transmon = (E_J-E_J*np.cos(phis))
PE_transmon = PE_transmon/w
```

In [15]:

```
#We now import Qutip
import qutip
from qutip import destroy

#Now constructing the Hamiltonian and then solving for the energies
N = 35
N_energies = 5
c = destroy(N)
H_QHO = w*c.dag()*c
E_QHO = H_QHO.eigenenergies()[0:N_energies]
H_transmon = w*c.dag()*c + (Delta/2)*(c.dag()*c)*(c.dag()*c - 1)
E_transmon = H_transmon.eigenenergies()[0:2*N_energies]
```

In [19]:

```
# We now print the energy levels
print(E_QHO)
print(E_transmon)
```

```
[0.0e+00 5.0e+09 1.0e+10 1.5e+10 2.0e+10]
[0.00e+00 1.70e+09 5.00e+09 6.60e+09 9.70e+09 1.12e+10 1.41e+10 1.55e+10
 1.82e+10 1.95e+10]
```

As can be seen from the above generated energy level results, the values in the array E_QHO are evenly spaced, i.e they differ by 0.5e+10 units of energy , while in the case of the E_transmon, there seems to be an innate anharmonicity.

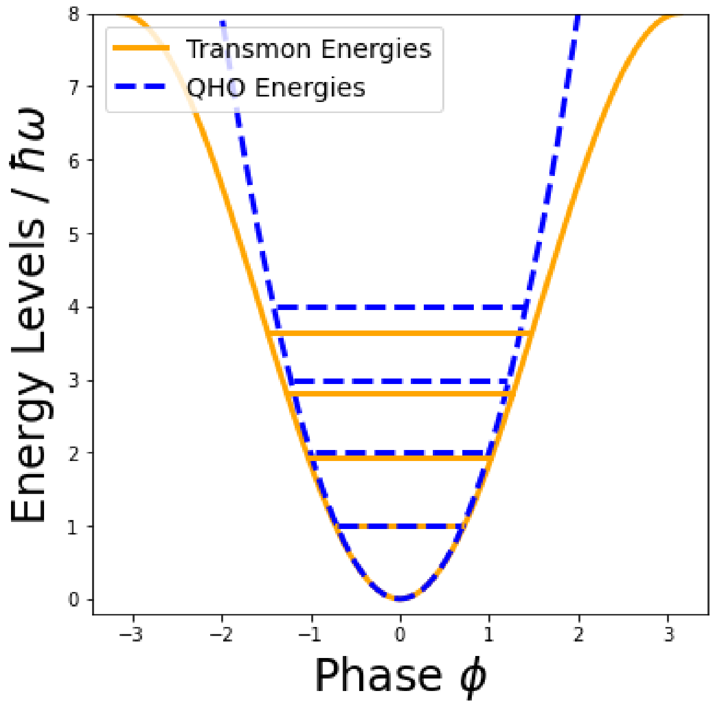
We can now plot the graph between energy levels and phase using the inbuilt plotting tools in Qubit and Matplotlib

In [18]:

```
fig, axes = plt.subplots(1, 1, figsize=(6,6))

axes.plot(phis, PE_transmon, '-', color='orange', linewidth=3.0)
axes.plot(phis, PE_QHO, '--', color='blue', linewidth=3.0)

for eidx in range(1,N_energies):
    delta_E_QHO = (E_QHO[eidx]-E_QHO[0])/w
    delta_E_transmon = (E_transmon[2*eidx]-E_transmon[0])/w
    QHO_lim_idx = min(np.where(PE_QHO[int((N_phis+1)/2):N_phis] > delta_E_QHO)[0])
    trans_lim_idx = min(np.where(PE_transmon[int((N_phis+1)/2):N_phis] > delta_E_transmon)[0])
    trans_label, = axes.plot([phis[mid_idx-trans_lim_idx-1], phis[mid_idx+trans_lim_idx-1]], \
                             [delta_E_transmon, delta_E_transmon], '-', color='orange', linewidth=3.0)
    qho_label, = axes.plot([phis[mid_idx-QHO_lim_idx-1], phis[mid_idx+QHO_lim_idx-1]], \
                           [delta_E_QHO, delta_E_QHO], '--', color='blue', linewidth=3.0)

axes.set_xlabel('Phase $\phi$', fontsize=24)
axes.set_ylabel('Energy Levels / $\hbar\omega$', fontsize=24)
axes.set_ylim(-0.2,8)

qho_label.set_label('QHO Energies')
trans_label.set_label('Transmon Energies')
axes.legend(loc=2, fontsize=14)
```

Out[18]:

```
<matplotlib.legend.Legend at 0x7fd3bc78aac0>
```



Comparing the performances of three Quantum Processing Units (namely IBMQ Athens, Santiago and Belem) by building their noise models using Randomized Benchmarking.

In [37]:

```
from qiskit import QuantumCircuit, execute
from qiskit import IBMQ, Aer
from qiskit.visualization import plot_histogram
import numpy as np

# first we load the necessary packages
```

In [38]:

```
IBMQ.save_account('07f879b8e9a85f2d318bb3ea48fe6bfc14907207745e9d584650779d3906be087c7e16
0b89f4b5c31943999cb602631d9ba46416acb9004a511d17d8c903ffe6')
```

```
configrc.store_credentials:WARNING:2021-04-29 22:04:19,572: Credentials already present.
Set overwrite=True to overwrite.
```

In [39]:

```
provider = IBMQ.load_account()
```

```
C:\Users\DRA\anaconda3\lib\site-packages\qiskit\providers\ibmq\ibmqfactory.py:192: UserWa
rning: Timestamps in IBMQ backend properties, jobs, and job results are all now in local
time instead of UTC.
  warnings.warn('Timestamps in IBMQ backend properties, jobs, and job results '
ibmqfactory.load_account:WARNING:2021-04-29 22:04:33,072: Credentials are already in use.
The existing account in the session will be replaced.
```

In [10]:

```
from qiskit.providers.aer.noise import NoiseModel
import qiskit.ignis.verification.randomized_benchmarking as rb
import matplotlib.pyplot as plt
```

We make noise Models to mimic the Circuit Based Quantum Processors that we intend to compare.

In [28]:

```
# We basically build the noise model using the backend properties (that have been drawn f
rom my IBM QE account in the previous lines of code)

backend_santiago = provider.get_backend('ibmq_santiago')
noise_model_santiago = NoiseModel.from_backend(backend_santiago)

backend_athens = provider.get_backend('ibmq_athens')
noise_model_athens = NoiseModel.from_backend(backend_athens)

backend_belem = provider.get_backend('ibmq_belem')
noise_model_belem = NoiseModel.from_backend(backend_belem)

# We wish to compare IBMQ Athens , IBQ Santiago and IBMQ Belem, all of which are supercon
ducting QPUs
```

Random circuit generation from a collection of gates

In [40]:

```
#glenghts here is an array to store the gate lengths. We consider them to vary from 1 to
300.There are 9 lengths in total.
```

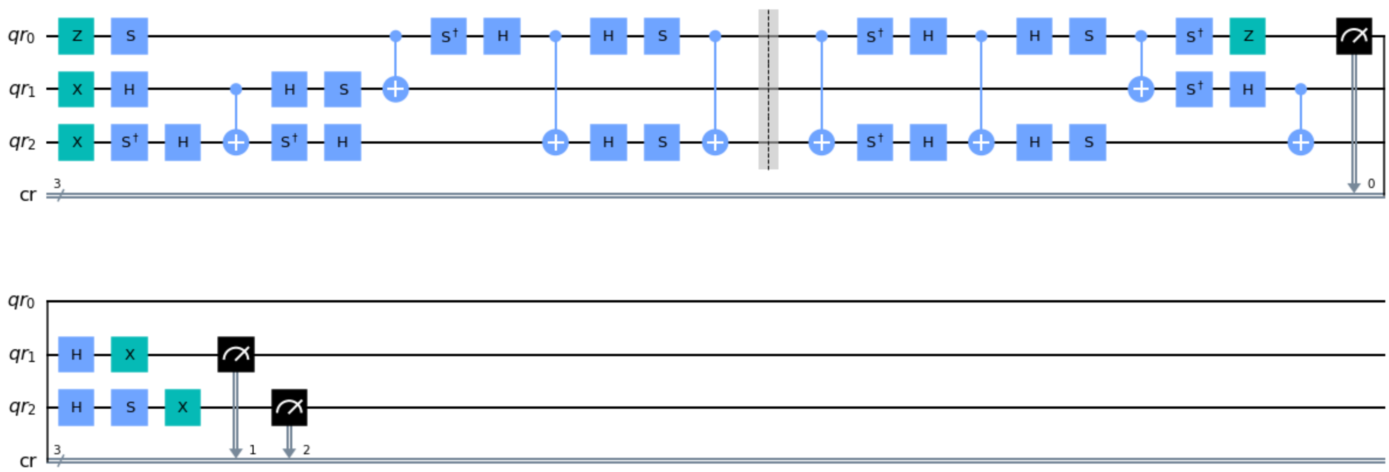
```
#npq stores the number and pattern of the qubits, since we are considering 2 qubit circui
ts, the values that npq can take are 0, 1 and 2

glengths = [1, 10, 20, 75, 100, 125, 150, 175, 300]

npq=[[0,1,2]]

rb_circs, _ = rb.randomized_benchmarking_seq(length_vector = glengths, rb_pattern = npq)

#rb_circs here refers to the 2-D vector created by the function rb.randomized_benchmarkin
g_seq
```

We now draw the circuit with the given glengths

In [30]:

```
rb_circs[0][0].draw('mpl')
```

Out[30]:



Running the randomized circuits using noise models

In [32]:

```
# We now run the 9 circuits generated one at a time using both the three noise models. Fo
r loop has been used to scan through the array.
countOf_000_athens = np.zeros(9) # To store the counts of 000 for athens
countOf_000_santiago = np.zeros(9)  # To store the counts of 000 for santiago
countOf_000_belem = np.zeros(9)  # To store the counts of 000 for belem
backend = Aer.get_backend('qasm_simulator')
sh = 1000*256   # the number of shots, I have chosen an arbitrarily large number
for i in range(9):
    # We first obain the results for the athens noise model
    job_athens = execute(rb_circs[0][i], backend, noise_model=noise_model_athens, shots
= sh)
    results_athens = job_athens.result()
    counts_athens = results_athens.get_counts()
    countOf_000_athens[i] = counts_athens["000"] #we're extracting the 000 counts from th
e athens results

    # Now obtaining the results for the santiago noise model using same number of shots
    job_santiago = execute(rb_circs[0][i], backend, noise_model=noise_model_santiago, sh
ots = sh)
    results_santiago = job_santiago.result()
    counts_santiago = results_santiago.get_counts()
    countOf_000_santiago[i] = counts_santiago["000"] #we're extracting the 000 counts fr
om the santiago results

    #Results for the belem noise model
    job_belem = execute(rb_circs[0][i], backend, noise_model=noise_model_belem, shots =
sh)
    results_belem = job_belem.result()
```

```
    counts_belem = results_belem.get_counts()
    countOf_000_belem[i] = counts_belem["000"]



    print("Finished circuit run ",str(i+1),"out of 9 on using the three noise models.") # Just to check if the code runs
```

```
Finished circuit run  1 out of 9 on using the three noise models.
Finished circuit run  2 out of 9 on using the three noise models.
Finished circuit run  3 out of 9 on using the three noise models.
Finished circuit run  4 out of 9 on using the three noise models.
Finished circuit run  5 out of 9 on using the three noise models.
Finished circuit run  6 out of 9 on using the three noise models.
Finished circuit run  7 out of 9 on using the three noise models.
Finished circuit run  8 out of 9 on using the three noise models.
Finished circuit run  9 out of 9 on using the three noise models.
```

Plotting and comparing results

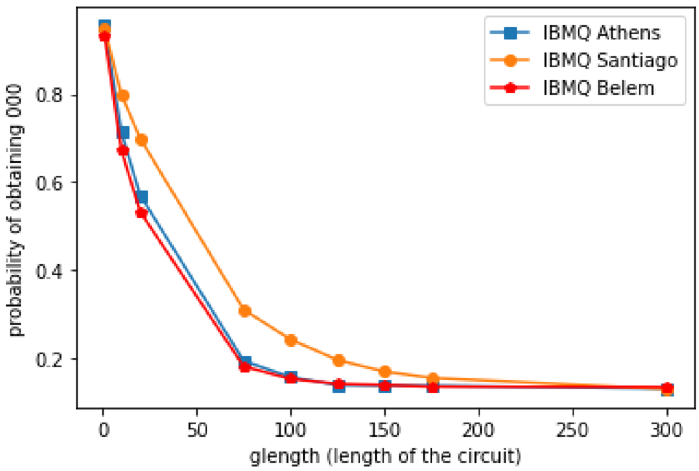
We plot the counts of 000 obtain for each of the QPUs against the glength parameter, i.e the length of the circuit for each count. The plot so obtained will show us the probability of getting the right result 000 as the length of the circuit increases.

In [36]:

```
# To obtain the probability, we divide the number of counts by the number of shots.
plt.plot(glengths, countOf_000_athens/sh, marker = 's')
plt.plot(glengths, countOf_000_santiago/sh, marker = 'o')
plt.plot(glengths, countOf_000_belem/sh, marker = 'p', color='red')
plt.legend(['IBMQ Athens','IBMQ Santiago' , 'IBMQ Belem'])
plt.xlabel('glength (length of the circuit)')
plt.ylabel('probability of obtaining 000')
```

Out[36]:

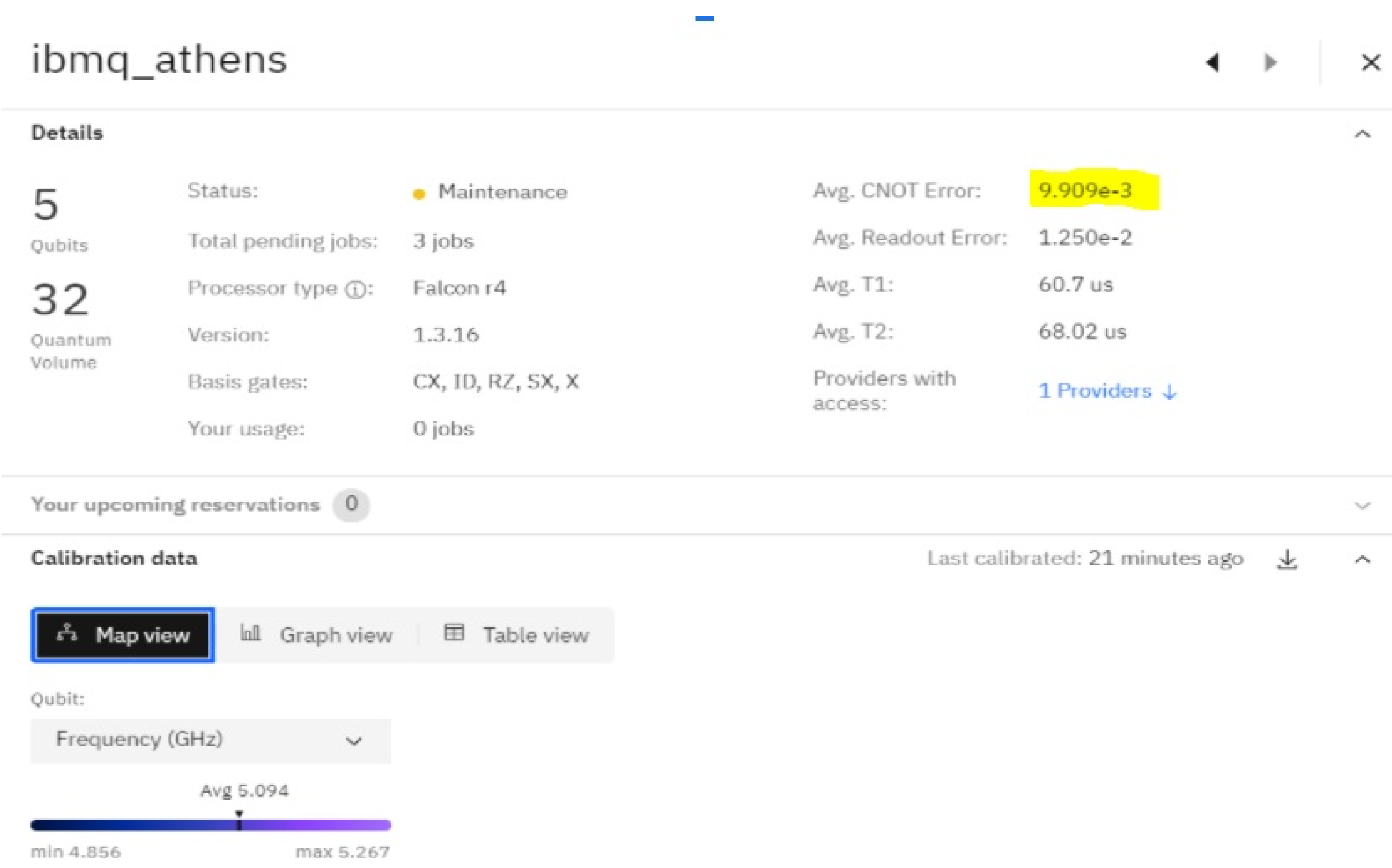
```
Text(0, 0.5, 'probability of obtaining 000')
```

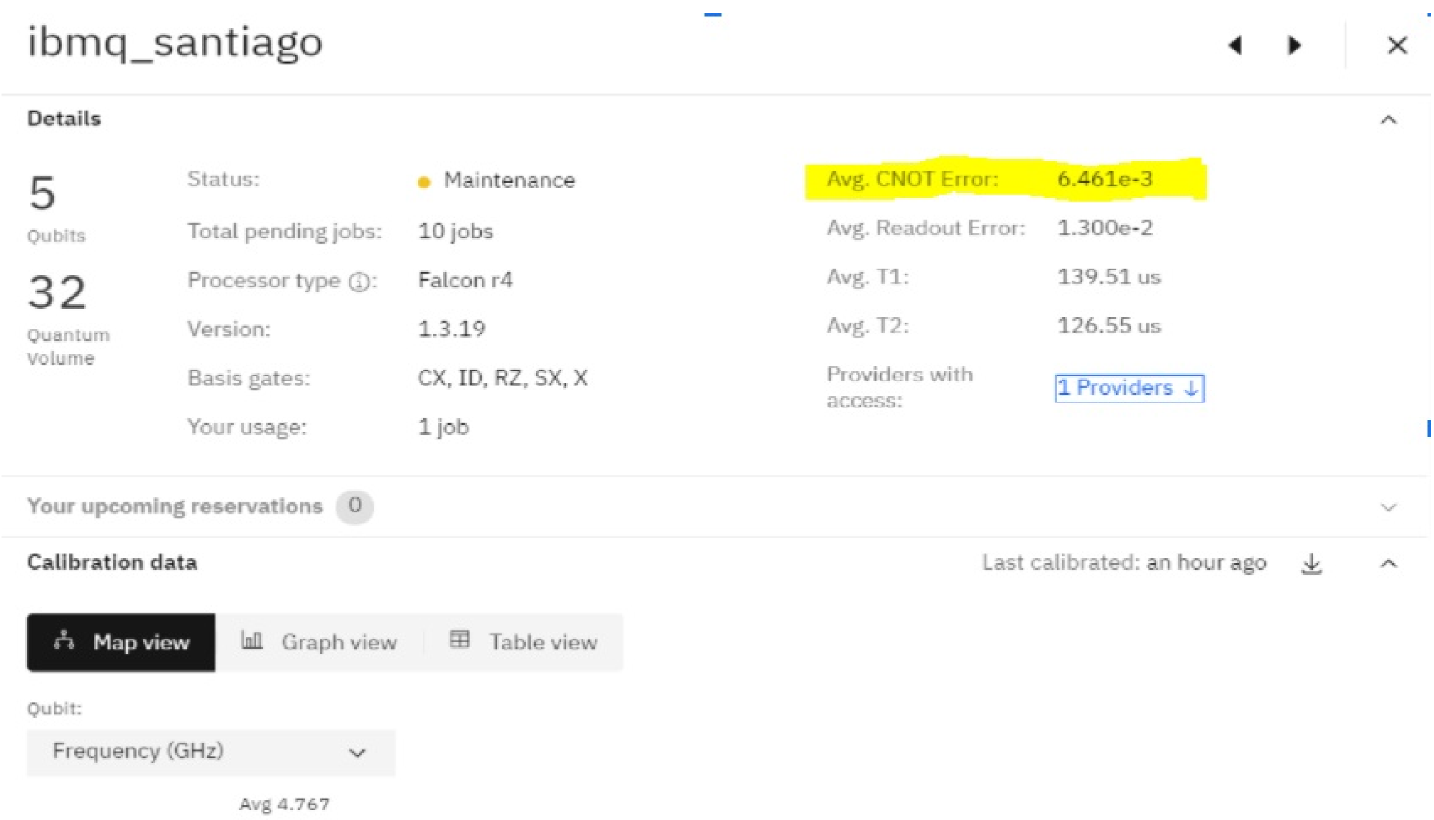


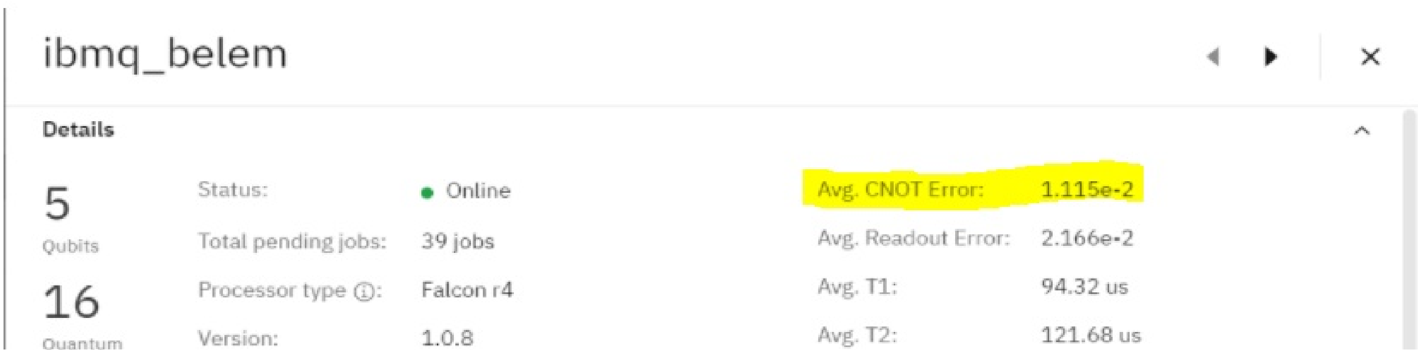
Observations and Inference

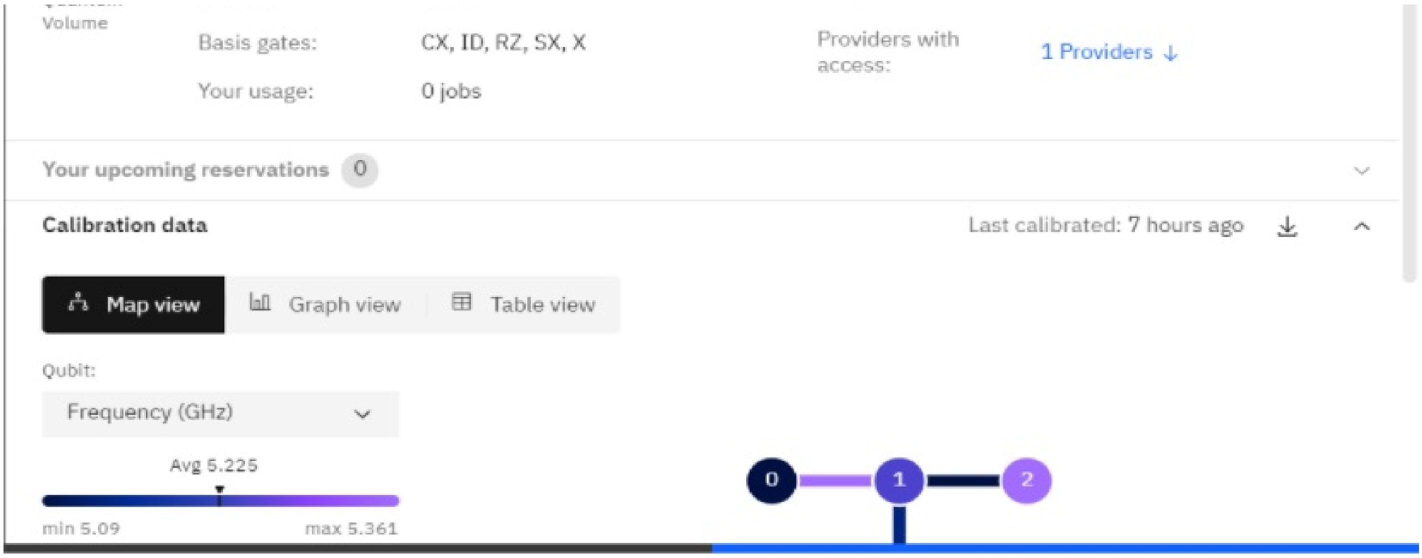
According to the plot obtained above, we observe that as the length of the circuit / gate length increases, the probabilty of obtaining our desired result / an accurate result decreases for all the three QPU s . But looking closely, it is seen that the decrease in accuracy for the IBMQ Santiago is slightly lesser than the other two . This is followed by IBM Q Athens whose decrease in accuracy is slightly lesser than IBMQ Belem until the gate length of around 120 beyond which both Athens and Belem happen to have similar gate fidelity. After the gate length of around 250, all the three models show similar behaviour. Hence on the basis of avergage CNOT error, or gate error, we can rank the three models in the following order IBMQ Santiago < IBMQ Athens < IBMQ Belem

Comparing the results with the standard Avg. CNOT values of the gates obtained from IBM QE









As can be seen from the standard set of Avg CNOT Error values, the gate error for IBMQ Belem is most followed by IBMQ Santiago , and is least for IBMQ Athens.

Therefore, these are in tandem with the graphical results.

Implications / Impact

The degree to which a qubit is affected by noise is dependent on the amount of noise impinging on the qubit and it's susceptibility to that noise. Both these points are intrinsically realted to the fabrication and design of qubits. The noise response of a qubit depends on how it couples with noise (either through longitudinal or transverse coupling with respect to the qubit's quantization axis).

```
In [ ]:
```

Creating a single transmon

In [1]:

```
%load_ext autoreload
%autoreload 2
#We're automoatically reloading modules in case they change
```

In [2]:

```
import qiskit_metal as metal
from qiskit_metal import designs, draw
from qiskit_metal import MetalGUI, Dict, open_docs
```

In [4]:

```
design = designs.DesignPlanar()
    #Instantiating a new circuit design class, QDesign
```

In [5]:

```
#Launching Qiskit Metal GUI
gui = MetalGUI(design)
```

In [6]:

```
from qiskit_metal.qlibrary.qubits.transmon_cross import TransmonCross
TransmonCross.get_template_options(design)
```

Out[6]:

```
{'pos_x': '0um',
 'pos_y': '0um',
 'connection_pads': {},
 '_default_connection_pads': {'connector_type': '0',
  'claw_length': '30um',
  'ground_spacing': '5um',
  'claw_width': '10um',
  'claw_gap': '6um',
  'connector_location': '0'},
 'cross_width': '20um',
 'cross_length': '200um',
 'cross_gap': '20um',
 'orientation': '0',
 'layer': '1',
 'hfss_inductance': '10nH',
 'hfss_capacitance': 0,
 'hfss_resistance': 0,
 'hfss_mesh_kw_jj': 7e-06,
 'q3d_inductance': '10nH',
 'q3d_capacitance': 0,
 'q3d_resistance': 0,
 'q3d_mesh_kw_jj': 7e-06,
 'gds_cell_name': 'my_other_junction'}
```

In [7]:

```
xmon_options = dict(
    connection_pads=dict(
        a = dict( connector_location = '0', connector_type = '0'),
        b = dict(connector_location = '90', connector_type = '0'),
        c = dict(connector_location = '180', connector_type = '1'),
    ),
)

# Creating a new Transmon Cross object with name 'Q1'
q1 = TransmonCross(design, 'Q1', options=xmon_options)
```

```
gui.rebuild()   # rebuild the design and plot
gui.autoscale() #resize GUI to see QComponent
gui.zoom_on_components(['Q1'])
```

In [8]:

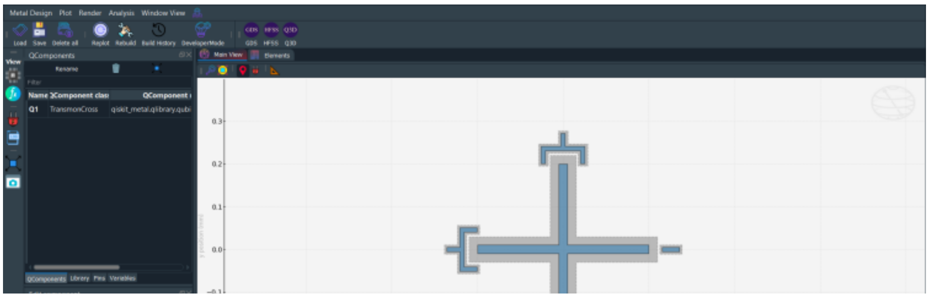
```
q1
```

Out[8]:

```
name:    Q1
class:   TransmonCross
options:
  'pos_x'                 : '0um',
  'pos_y'                 : '0um',
  'connection_pads'       : {
       'a'                     : {
            'connector_type'     : '0',
            'claw_length'        : '30um',
            'ground_spacing'     : '5um',
            'claw_width'         : '10um',
            'claw_gap'           : '6um',
            'connector_location': '0',
                                 },
       'b'                     : {
            'connector_type'     : '0',
            'claw_length'        : '30um',
            'ground_spacing'     : '5um',
            'claw_width'         : '10um',
            'claw_gap'           : '6um',
            'connector_location': '90',
                                 },
       'c'                     : {
            'connector_type'     : '1',
            'claw_length'        : '30um',
            'ground_spacing'     : '5um',
            'claw_width'         : '10um',
            'claw_gap'           : '6um',
            'connector_location': '180',
                                 },
                            },
  'cross_width'           : '20um',
  'cross_length'          : '200um',
  'cross_gap'             : '20um',
  'orientation'           : '0',
  'layer'                 : '1',
  'hfss_inductance'       : '10nH',
  'hfss_capacitance'      : 0,
  'hfss_resistance'       : 0,
  'hfss_mesh_kw_jj'       : 7e-06,
  'q3d_inductance'        : '10nH',
  'q3d_capacitance'       : 0,
  'q3d_resistance'        : 0,
  'q3d_mesh_kw_jj'        : 7e-06,
  'gds_cell_name'         : 'my_other_junction',
module:  qiskit_metal.qlibrary.qubits.transmon_cross
id:      1
```

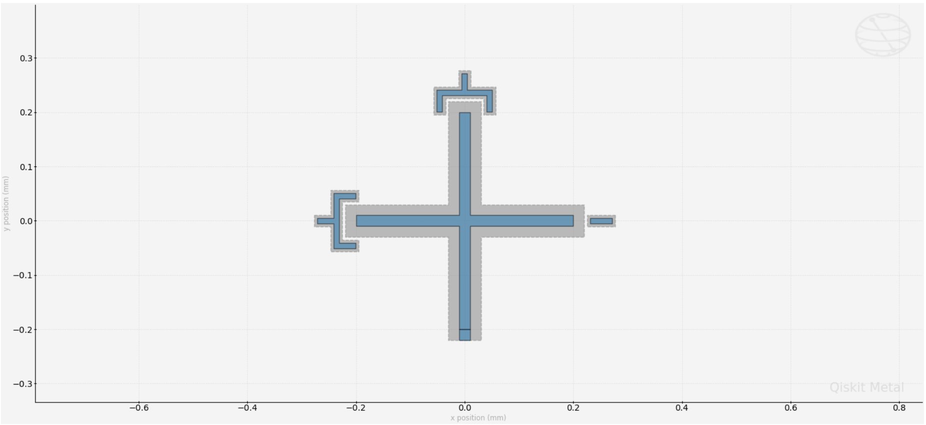
In [9]:

```
gui.screenshot()
```



In [10]:

```
gui.figure.savefig('shot.png')

from IPython.display import Image, display
_disp_ops = dict(width=500)
display(Image('shot.png', **_disp_ops))
```




```
12:19PM 19s CRITICAL [_qt_message_handler]: line: 0, func: None(), file: None  WARNING: Q
WindowsWindow::setMouseGrabEnabled: Not setting mouse grab for invisible window QWidgetWi
ndow/'menuAnalysisWindow'

12:19PM 27s CRITICAL [_qt_message_handler]: line: 0, func: None(), file: None  WARNING: Q
WindowsWindow::setMouseGrabEnabled: Not setting mouse grab for invisible window QWidgetWi
ndow/'menuRenderWindow'

12:19PM 28s CRITICAL [_qt_message_handler]: line: 0, func: None(), file: None  WARNING: Q
WindowsWindow::setMouseGrabEnabled: Not setting mouse grab for invisible window QWidgetWi
ndow/'menuAnalysisWindow'

12:19PM 28s CRITICAL [_qt_message_handler]: line: 0, func: None(), file: None  WARNING: Q
WindowsWindow::setMouseGrabEnabled: Not setting mouse grab for invisible window QWidgetWi
ndow/'menuRenderWindow'
```

In [11]:

```
gui.rebuild()

all_component_names = design.components.keys()
gui.zoom_on_components(all_component_names)
```

In [12]:

```
design.chips.main.size.size_x = '11mm'
design.chips.main.size.size_y = '9mm'
#Altering the dimensions
```

In [13]:

```
q1.options.pos_x = '0.5 mm'
q1.options.pos_y = '0.25 mm'
q1.options.pad_height = '225 um'
q1.options.pad_width  = '250 um'
q1.options.pad_gap    = '50 um'
```

In [14]:

```
gui.rebuild()
```

```
03:55PM 27s CRITICAL [_qt_message_handler]: line: 0, func: None(), file: None  WARNING: Q
WindowsNativeFileDialogBase::selectNameFilter: Invalid parameter '*.metal' not found in '
All Files (*)'.

03:55PM 28s CRITICAL [_qt_message_handler]: line: 0, func: None(), file: None  WARNING: Q
WindowsNativeFileDialogBase::selectNameFilter: Invalid parameter '*.metal' not found in '
All Files (*)'.

03:56PM 31s WARNING [save_design]: Saving is a beta feature.
03:56PM 31s INFO [save_design]: Saving design to C:/Users/Rita/Desktop/transmon 1
03:56PM 32s WARNING [find_id]: In Components.find_id(), the name=__getstate__ is not used
in design._components
03:56PM 33s ERROR [log_error_easy]:

Traceback (most recent call last):

  File "c:\users\rita\github\qiskit-metal\qiskit_metal\toolbox_metal\import_export.py", l
ine 51, in save_metal
    pickle.dump(self, open(filename, "wb"))

TypeError: 'NoneType' object is not callable

ERROR WHILE SAVING: 'NoneType' object is not callable
03:56PM 33s ERROR [save_design]: Saving failed.
```

In [15]:

```
gui.main_window.close()
```

```
04:02PM 05s WARNING [save_design]: Saving is a beta feature.
04:02PM 05s INFO [save_design]: Saving design to C:/Users/Rita/Desktop/transmon 1
04:02PM 05s WARNING [find_id]: In Components.find_id(), the name=__getstate__ is not used
in design._components
04:02PM 05s ERROR [log_error_easy]:

Traceback (most recent call last):

  File "c:\users\rita\github\qiskit-metal\qiskit_metal\toolbox_metal\import_export.py", l
ine 51, in save_metal
    pickle.dump(self, open(filename, "wb"))

TypeError: 'NoneType' object is not callable

ERROR WHILE SAVING: 'NoneType' object is not callable
04:02PM 05s ERROR [save_design]: Saving failed.
```

Out[15]:

```
True
```

In [ ]: