Week 11

Lecture 21
Python, IBM's QISKit

Lecture 22
- Further quantum algorithms

Lab 11
Implementing small algorithms on IBM's 16 qubit machine

QISKit and Python

Physics 90045
Lecture 21

Overview

In Friday's laboratory we will be programming IBM's 16 qubit quantum computer. This lecture introduces the tools you can use to access it on your computer:

- Python primer
- Jupyter notebooks
- Using QISKit

On your own machine: Installing Python

Install Python3

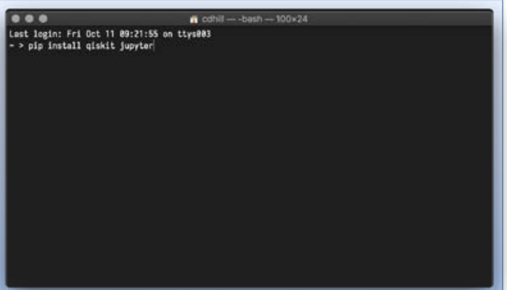
Install Anaconda with ***Python 3.6*** version, download from:
https://www.anaconda.com/download/

Reference:
https://www.qiskit.org/documentation/install.html

Quantum Computing and IBM Q #IBMQ

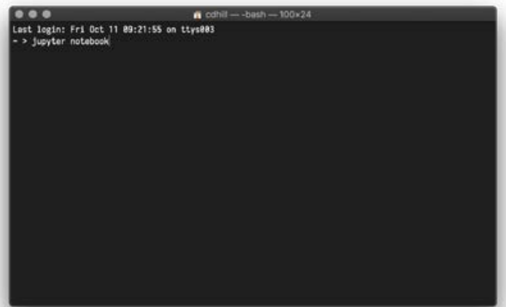
PHYC90045 Introduction to Quantum Computing

Install Qiskit

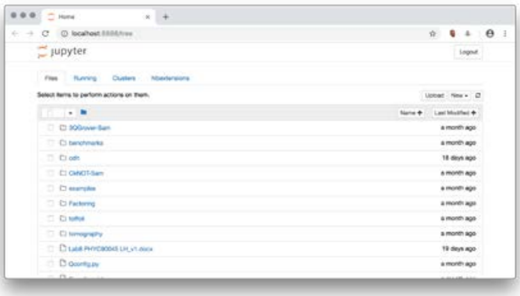


PHYC90045 Introduction to Quantum Computing

Juptyer notebook

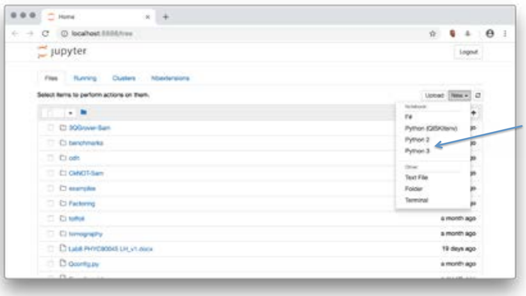


Jupyter notebook



PHYC90045 Introduction to Quantum Computing

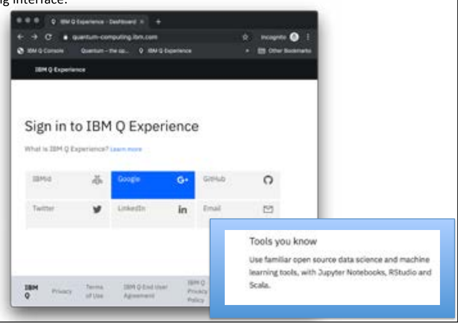
Creating a new file



PHYC90045 Introduction to Quantum Computing

IBM Quantum Computing

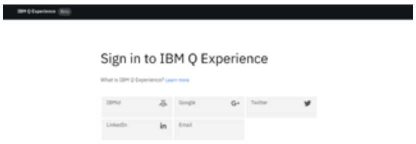
An easy way to use jupyter notebooks is online through IBM's quantum computing interface:



PHYC90045 Introduction to Quantum Computing

Using IBM Quantum Experience

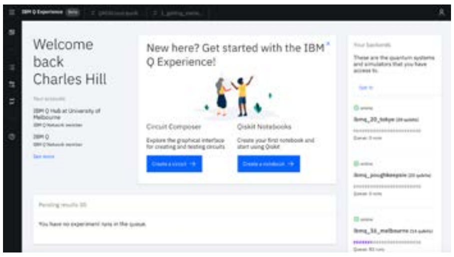
Go to IBM Quantum Experience website: **quantum-computing.ibm.com**



Sign up. Sign in. If you haven't already, get yourself added to the list of users in the Melbourne Q-Hub.

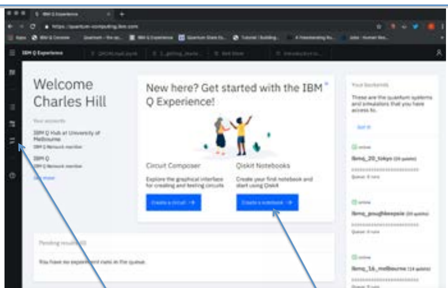
PHYC90045 Introduction to Quantum Computing

IBM Q Experience Landing Page



PHYC90045 Introduction to Quantum Computing

Starting a new Workbook

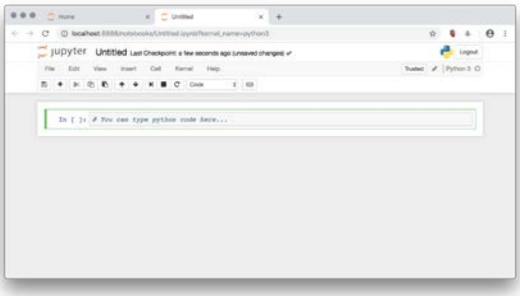


To see your existing notebooks, tutorials or start a new one. Select "QISKIT Notebooks"

Or click on the button to create a new notebook

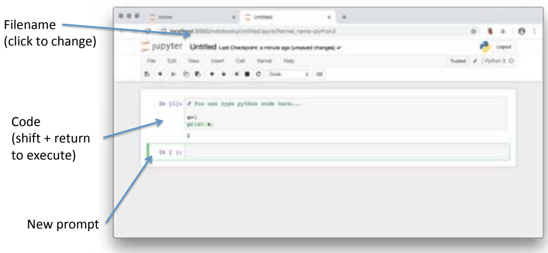
PHYC90045 Introduction to Quantum Computing

Interactive Python Environment



PHYC90045 Introduction to Quantum Computing

Using Jupyter Notebook



PHYC90045 Introduction to Quantum Computing

Some Python Basics

```
In [2]: a=6
        b=7
        life = a*b
        life
Out[2]: 42
```

Similar to many other imperative languages you may know for numerical work:
(C/C++, MATLAB, R, FORTRAN, Julia) and often used for data processing.

Defining and calling Functions

def keyword indicates a new function

No types on parameters

Colon

Comment

```
def square(x):
    # This is a comment
    return x*x
```

Whitespace is significant in python.
Indentation indicates a new block.

No semicolons.
Newline is the end of a statement

Calling a function:

```
square(4)
square(x=4)
```

Named parameters

PHYC90045 Introduction to Quantum Computing

Lists and for loops

Lists stores a sequence of values. Square brackets indicate a list:

```
["This", "is", "a", "list"]

primes = [2, 3, 5, 7, 11]
```

Eg. For loops often use lists:

```
for p in primes:
    print(p)
```

```
2
3
5
7
11
```

Accessing an individual element.
0-based!

```
primes[2]
```

```
5
```

PHYC90045 Introduction to Quantum Computing

Dictionaries

Dictionaries store key-value pairs.

Curly braces indicate a dictionary

key

value

```
me = {"name": "Charles", "height":1.79, "favourite_food": "pizza"}
me["favourite_food"]
```

```
'pizza'
```

```
me["favourite_food"] = "sweet and sour pork"
me["favourite_food"]
```

```
'sweet and sour pork'
```

Importing other libraries

```
import numpy as np

X = np.matrix([[0,1],
               [1,0]])
```

Importing a module ("as np" is optional). numpy gives similar functionality to MATLAB

Calling functions from that module.
Here creating an X matrix.

Or import individual functions and classes:

```
from qiskit import QuantumProgram
from qiskit import available_backends, execute, get_backend, compile
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, QISKitError
```

qiskit is an API for interacting with IBM's quantum computers remotely.

IBM's Qiskit

Terra, Aer, Ignis, Aqua

Tera (Earth): Access to IBM Q Devices through python interface
Aer (Air): Classical simulation of quantum algorithms/circuits
Ignis (Fire): Characterisation of errors, tomography
Aqua (Water): Large selection of quantum algorithms

QISKit Introduction

Introduction to Qiskit

Constructing a circuit in Qiskit

In this tutorial we will construct a circuit to create a Bell-state using Qiskit.

First we will load the required libraries from qiskit.

In [1]:
```
%matplotlib inline

# Importing standard Qiskit libraries and configuring account
import numpy as np
from qiskit import *
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *

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

PHYC90045 Introduction to Quantum Computing

Now, we will construct a quantum circuit with two qubits.

In [3]:
```
# Create a Quantum Register with 2 qubits.
q = QuantumRegister(2, 'q')
c = ClassicalRegister(2, 'c')

# Create a Quantum Circuit acting on the q register
circ = QuantumCircuit(q, c)
```

PHYC90045 Introduction to Quantum Computing

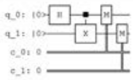
Adding gates to our circuit.

In [4]:
```
# Add a H gate on qubit 0, putting this qubit in superposition.
circ.h(q[0])
# Add a CX (CNOT) gate on control qubit 0 and target qubit 1, putting
# the qubits in a Bell state.
circ.cx(q[0], q[1])

# Measure the results
circ.measure(q, c)
```

Out[4]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x1244de690>
```

In [5]:
```
circ.draw()
```

Out[5]:



PHYC90045 Introduction to Quantum Computing

Simulating the circuit

First, let's run the circuit on a simulator.

IBM's simulators are in a package called Aer. Let's get a backend which can simulate our circuit.

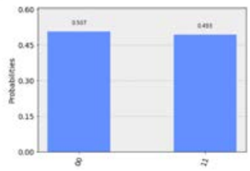
In [6]:
```
backend = Aer.get_backend('qasm_simulator')
job = execute(circ, backend)
```

PHYC90045 Introduction to Quantum Computing

We can obtain samples from this circuit (from the simulator). To do this:

In [7]:
```
result_sim = job.result()
counts_sim = result_sim.get_counts(circ)
plot_histogram([counts_sim])
```

Out[7]:



PHYC90045 Introduction to Quantum Computing

Using a real IBM Q Device

Now, let us submit the circuit to a real quantum computer, and see how it performs.

PHYC90045 Introduction to Quantum Computing

A number of different machines are available for use. Note the machines available to you might be different.

In [27]:
```
print("Available backends:")
provider.backends()
```

```
Available backends:
```

Out[27]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open', pr
oject='main')>,
 <IBMQBackend('ibmqx2') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_16_melbourne') from IBMQ(hub='ibm-q', group='open', projec
t='main')>,
 <IBMQBackend('ibmq_vigo') from IBMQ(hub='ibm-q', group='open', project='mai
n')>,
 <IBMQBackend('ibmq_ourense') from IBMQ(hub='ibm-q', group='open', project='ma
in')>]
```

PHYC90045 Introduction to Quantum Computing

Let us find the least busy device and run on that.

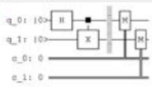
In [9]:
```
from qiskit.providers.ibmq import least_busy

large_enough_devices = provider.backends(filters=lambda x: x.configuration().n_qub
its < 10 and
                                                           not x.configuration().simul
ator)
backend = least_busy(large_enough_devices)
print("The best backend is " + backend.name())
```

```
The best backend is ibmq_vigo
```

PHYC90045 Introduction to Quantum Computing

In [10]:
```
# Create a Classical Register with 2 bits.
q = QuantumRegister(2, 'q')
c = ClassicalRegister(2, 'c')

# Create a Quantum Circuit
qc = QuantumCircuit(q,c)

# Add gates
qc.h(q[0])
qc.cx(q[0],q[1])

qc.barrier(q)

# Map the quantum measurement to the classical bits
qc.measure(q,c)

#drawing the circuit
qc.draw()
```

Out[10]:



Now, let us use the Qiskit API to submit job to IBM remotely.

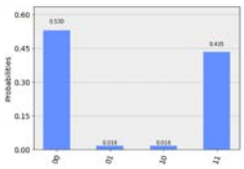
In [11]:
```
from qiskit.tools.monitor import job_monitor
shots = 1024            # Number of shots to run the program (experiment); maximum
is 8192 shots.

job_exp = execute(qc, backend=backend, shots=shots)
job_monitor(job_exp)
```
```
Job Status: job has successfully run
```


Once the job has run (we may need to wait), we can examine the results:

In [12]:
```
result_exp = job_exp.result()
result_exp
```
Out[12]:
```
Result(backend_name='ibmq_vigo', backend_version='1.0.1', date=datetime.dateti
me(2019, 10, 2, 12, 34, 26, tzinfo=tzutc()), execution_id='f4831262-e510-11e9-
9f14-ac1f6b47c318', header=Obj(backend_name='ibmq_vigo', backend_version='1.0.
1'), job_id='5d94994158870600187d441f', qobj_id='6a21ae0c-df84-4b5c-9b75-a91a4
4afb365', results=[ExperimentResult(data=ExperimentResultData(counts=Obj(0x0=5
43, 0x1=18, 0x2=18, 0x3=445)), header=Obj(clbit_labels=[['c', 0], ['c', 1]], c
reg_sizes=[['c', 2]], memory_slots=2, n_qubits=5, name='circuit2', qreg_sizes=
[['q', 5]], qubit_labels=[['q', 0], ['q', 1], ['q', 2], ['q', 3], ['q', 4]]),
meas_level=2, memory=False, shots=1024, success=True)], status='Successful com
pletion', success=True, time_taken=8.320900440216064)
```


Plotting the results:

In [13]:
```
counts_exp = result_exp.get_counts(qc)
plot_histogram([counts_exp])
```
Out[13]:



Now with more qubits

Let's go over that one more time with the five qubit GHZ state:

$$\frac{|00000\rangle + |11111\rangle}{\sqrt{2}}$$

```
In [14]: %matplotlib inline

         # Importing standard Qiskit libraries and configuring account
         import numpy as np
         from qiskit import *

In [17]: # Loading your IBM Q account(s)
         provider = IBMQ.load_account()
```



```
In [18]: # Create a Quantum Register with 5 qubits.
         q = QuantumRegister(5, 'q')
         c = ClassicalRegister(5, 'c')

         # Create a Quantum Circuit acting on the q register
         qc = QuantumCircuit(q, c)
```



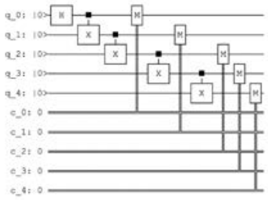
```
qc.h(q[0])
qc.cx(q[0], q[1])
qc.cx(q[1], q[2])
qc.cx(q[2], q[3])
qc.cx(q[3], q[4])

qc.measure(q, c)
```

PHYC90045 Introduction to Quantum Computing

Draw the circuit, and check everything is correct!

In [21]:
```
qc.draw()
```

Out[21]:

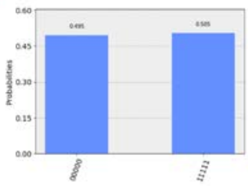


PHYC90045 Introduction to Quantum Computing

Simulate the circuit and ensure you get correct results.

In [23]:
```
backend = Aer.get_backend('qasm_simulator')
job = execute(qc, backend)
result_sim = job.result()
counts_sim = result_sim.get_counts(qc)
plot_histogram([counts_sim])
```

Out[23]:



PHYC90045 Introduction to Quantum Computing

When everything is correct, submit the job to IBM Q Device to run:

In [24]:
```
from qiskit.providers.ibmq import least_busy

large_enough_devices = provider.backends(filters=lambda x: x.configuration().n_qub
its < 10 and
                                                        not x.configuration().simul
ator)
backend = least_busy(large_enough_devices)
print("The best backend is " + backend.name())
```

```
The best backend is ibmq_vigo
```

In [25]:
```
from qiskit.tools.monitor import job_monitor
shots = 1024            # Number of shots to run the program (experiment); maximum is 8192 shots.
max_credits = 3         # Maximum number of credits to spend on executions.

job_exp = execute(qc, backend=backend, shots=shots, max_credits=max_credits)
job_monitor(job_exp)
```

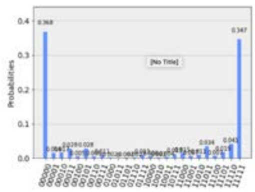
```
Job Status: job has successfully run
```

PHYC90045 Introduction to Quantum Computing

Plot the results

In [26]:
```
result_exp = job_exp.result()
counts_exp = result_exp.get_counts(qc)
plot_histogram([counts_exp])
```

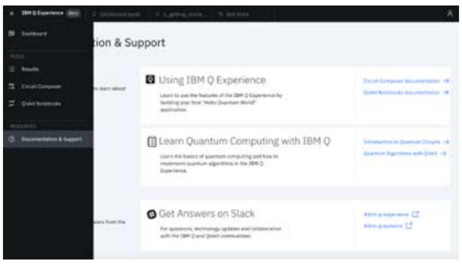
Out[26]:



PHYC90045 Introduction to Quantum Computing

Qiskit Documentation

Make use of "Documentation and Support" in the left hand menu!



Week 11

Lecture 21
Python, IBM's QISKit

Lecture 22
- Further quantum algorithms

Lab 11
Implementing small algorithms on IBM's 16 qubit machine