Introduction to the FRIB-TA Summer School: Quantum Computing and Nuclear Few- and Many-Body Problems with teaching plan and learning outcomes

Facility for Rare Isotope Beams, Michigan State University, USA

June 20

Introduction

Recent developments in quantum information systems and technologies offer the possibility to address some of the most challenging large-scale problems in science, whether they are represented by complicated interacting quantum mechanical systems or classical systems. The last years have seen a rapid and exciting development in algorithms and quantum hardware. The emphasis of this summer school is to highlight, through a series of lectures and hands-on exercises and practice sessions, how quantum computing algorithms can be used to study nuclear few- and many-body problems of relevance for low-energy nuclear physics. And how quantum computing algorithms can aid in studying systems with increasingly many more degrees of freedom compared with more classical few- and many-body methods. Several quantum algorithms for solving quantum-mechanical few- and many-particle problems with be discussed. The lectures will start with the basic ideas of quantum computing. Thereafter, through examples from nuclear physics, we will elucidate how different quantum algorithms can be used to study these systems. The results from various quantum computing algorithms will be compared to standard nuclear few- and many-body methods

Organizers: Alexei Bazavov (MSU), Scott Bogner (MSU), Heiko Hergert (MSU), Matthew Hirn (MSU), Morten Hjorth-Jensen (MSU), Dean Lee (MSU), Huey-Wen Lin (MSU), and Andrea Shindler (MSU)

Contact person: Morten Hjorth-Jensen, hjensen@msu.edu

Additional material in these slides. In addition to containing information about the school, we provide a reminder on **numpy** and a brief introduction to Qiskit. Please take a look at these notes before coming to the school. This material is provided at the end of the these introductory notes.

Aims and Learning Outcomes

The following topics will be covered.

1. Basic elements of quantum computing (first day) with introduction to relevant software, including
   (a) Introduction to quantum computing, qubits and systems of qubits
   (b) Measurements, Superposition and Entanglement
   (c) Gates, unitary transformations and quantum circuits
   (d) Quantum algorithms and implementation on a real quantum computer
2. Simulating quantum-mechanical few- and many-body systems
   (a) Quantum algorithms for quantum mechanical systems
   (b) Quantum simulation of the Schroedinger equation
   (c) Quantum computing and nuclear few- and many-body systems
   (d) Quantum state preparation and Quantum simulations
   (e) Quantum simulations on a real quantum computer
3. Quantum field theory and quantum computing
4. Noise, error correction and mitigation

All of the above topics will be supported by examples, hands-on exercises and project work.

Practicalities

1. Lectures Monday through Wednesday, starting at 830am, see schedule below
2. Hands-on sessions before lunch and in the afternoons till 6pm
3. For all lecture days we provide relevant jupyter-notebooks you can work on
4. Lectures are in auditorium 1200. Hands-on sessions will be in both the main lecture hall 1200 and in rooms 1221 A&B (12 noon - 6 pm) and room 1309 (8:30am-6pm).

Learning material and resources

1. Qiskit textbook, free online, see https://qiskit.org/textbook/preface.html
2. Scherer, The Mathematics of Quantum Computing, see https://link.springer.com/book/10.1007/978-3-030-12358-1
3. Chuang and Nielsen, Quantum Computation and Quantum Information, https://www.cambridge.org/highereducation/books/quantum-computation-and-quantum-informa 01E10196D0A682A6AEFFEA52D53BE9AE#overview
4. Hundt, Quantum Computing for programmers, https://www.cambridge.org/core/books/quantum-computing-for-programmers/BA1C887BE4AC0D0D5653E71FFBEF61C6

Good resources. With the hands-on programming component we strongly recommend that you install Qiskit on your computer before the school starts.

1. For Qiskit, follow the instructions at https://qiskit.org/documentation/getting_started.html
2. We strongly recommend using the Jupyter notebook environment at https://quantum-computing.ibm.com/. This environment has Qiskit already set up and is free, just requires an email to register. It has built in support for Jupyter notebooks and should be sufficient for everything needed.
3. See also Ryan Larose's (from 2019) Quantum computing bootcamp with Qiskit - https://github.com/rmlarose/qcbq.
4. See also https://www.ryanlarose.com/external-resources.html

Scientific articles of interest.

1. Adam Smith, M. S. Kim, Frank Pollmann, and Johannes Knolle, Simulating quantum many-body dynamics on a current digital quantum computer, NPJ Quantum Information 5, Article number: 106 (2019), see https://www.nature.com/articles/s41534-019-0217-0

Detailed lecture plan

The duration of each lecture is approximately 45-50 minutes and there is a small break of 10-15 minutes between each lecture. Longer breaks at 1030am-11am and 3pm-330pm, except for Monday where there is also the possibility fora guided FRIB tour. In-person attendance is the main teaching modus, but lectures and hands-on sessions will be broadcasted via zoom for those who cannot attend in person. The zoom link will be sent to those who have expressed that they cannot attend in person. The lectures will also be recorded.

Teachers.

- AB = Alexei Bazavov
- BH = Benjamin Hall
- DJ = Danny Jammoa (online discussions and hands-on sessions)
- DL = Dean Lee
- JW = Jacob Watkins
- JB = Joey Bonitati
- MHJ = Morten Hjorth-Jensen
- RL = Ryan Larose
- QZ - Zhenrong Qian

Monday June 20.

- 8am-830am: Welcome and registration
- 830am-930am: Introduction to quantum computing, qubits, systems of qubits, gates and quantum circuits (AB)
- 930am-1030am: Measurements, Superposition, Entanglement (AB)
- 1030am-11am: Break, coffee, tea etc
- 11am-12pm: Hands-on session with applications and introduction to software libraries (AB, JW, RL)
- 12pm-1pm: Lunch (shorter lunch, else 1h30m lunches)
- 1pm-2pm: Algorithms for quantum dynamics (DL), simple problems
- 2pm-3pm: Quantum phase estimation and adiabatic evolution (ZQ and JB), simple problems
- 3pm-4pm: Break, coffee, tea or tour for FRIB for those interested. Please let us know if you are interested in a tour of FRIB.
- 4pm-6pm: Hands-on sessions and problem solving (AB+all)

Tuesday June 21.

- 830-930am: Hamiltonian simulation: a general overview (JW)
- 930am-1030am: Introduction to VQE and simple model (BH)
- 1030am-11am: Break, coffee, tea etc
- 11am-12pm: Many-body theory and nuclear few- and many-body systems (BH and MHJ)
- 12pm-130pm: Lunch
- 130pm-230pm: Quantum algorithms (VQE) and nuclear physics with applications (BH and MHJ), part 1
- 230pm-330pm: Quantum algorithms (VQE) and nuclear physics with applications (BH and MHJ), part 2
- 330pm-4pm: Break, coffee, tea etc
- 4pm-6pm: Hands-on sessions and problem solving (BH and JW+all)

Wednesday June 22.

- 830am-930am: Noise, error correction and mitigation, part I (RL)
- 930am-1030am: Noise, error correction and mitigation, part II (RL)
- 1030am-11am: Break, coffee, tea etc
- 11am-12pm: Practicing error correction and mitigation, hands-on part (RL)
- 12pm-130pm: Lunch
- 130pm-230pm: Wrapping up and defining nuclear many-body system to study for hands-on session (All)
- 230pm-330pm: Start hands-on session (RL)
- 330pm-4pm: Break, coffee, tea etc
- 4pm-6pm: Hands-on sessions and problem solving (all)

Prerequisites

You are expected to have operating programming skills in programming languages like Python (preferred) and/or Fortran, C++, Julia or similar and knowledge of quantum mechanics at an intermediate level (senior undergraduate and/or beginning graduate). Knowledge of linear algebra is essential. Additional modules for self-teaching on Python and quantum mechanics are also provided.

Software and needed installations

We will make extensive use of Python as programming language and its myriad of available libraries. You will find Jupyter notebooks invaluable in your work.

If you have Python installed (we strongly recommend Python3) and you feel pretty familiar with installing different packages, we recommend that you install the following Python packages via **pip** as

1. pip install numpy scipy matplotlib ipython scikit-learn mglearn sympy pandas pillow

For Python3, replace **pip** with **pip3**.

For OSX users we recommend, after having installed Xcode, to install **brew**. Brew allows for a seamless installation of additional software via for example

1. brew install python3

For Linux users, with its variety of distributions like for example the widely popular Ubuntu distribution, you can use **pip** as well and simply install Python as

1. sudo apt-get install python3 (or python for pyhton2.7)

etc etc.

Python installers

If you don't want to perform these operations separately and venture into the hassle of exploring how to set up dependencies and paths, we recommend two widely used distrubutions which set up all relevant dependencies for Python, namely

- Anaconda,

which is an open source distribution of the Python and R programming languages for large-scale data processing, predictive analytics, and scientific computing, that aims to simplify package management and deployment. Package versions are managed by the package management system **conda**.

- Enthought canopy

is a Python distribution for scientific and analytic computing distribution and analysis environment, available for free and under a commercial license.

Furthermore, Google's Colab is a free Jupyter notebook environment that requires no setup and runs entirely in the cloud. Try it out!

Useful Python libraries

Here we list several useful Python libraries we strongly recommend (if you use anaconda many of these are already there)

- NumPy is a highly popular library for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays
- The pandas library provides high-performance, easy-to-use data structures and data analysis tools
- Xarray is a Python package that makes working with labelled multi-dimensional arrays simple, efficient, and fun!
- Scipy (pronounced "Sigh Pie") is a Python-based ecosystem of open-source software for mathematics, science, and engineering.
- Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.
- Autograd can automatically differentiate native Python and Numpy code. It can handle a large subset of Python's features, including loops, ifs, recursion and closures, and it can even take derivatives of derivatives of derivatives
- SymPy is a Python library for symbolic mathematics.
- scikit-learn has simple and efficient tools for machine learning, data mining and data analysis
- TensorFlow is a Python library for fast numerical computing created and released by Google
- Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano
- And many more such as pytorch, Theano etc

All learning material and teaching schedule pertinent to the course is avaliable at this GitHub address. A simple **git clone** of the material gives you access to all lecture notes and program examples. Similarly, running a **git pull** gives you immediately the latest updates.

Numpy examples and Important Matrix and vector handling packages

There are several central software libraries for linear algebra and eigenvalue problems. Several of the more popular ones have been wrapped into ofter

software packages like those from the widely used text **Numerical Recipes**. The original source codes in many of the available packages are often taken from the widely used software package LAPACK, which follows two other popular packages developed in the 1970s, namely EISPACK and LINPACK. We describe them shortly here.

- LINPACK: package for linear equations and least square problems.
- LAPACK:package for solving symmetric, unsymmetric and generalized eigenvalue problems. From LAPACK's website http://www.netlib.org it is possible to download for free all source codes from this library. Both C/C++ and Fortran versions are available.
- BLAS (I, II and III): (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. Blas I is vector operations, II vector-matrix operations and III matrix-matrix operations. Highly parallelized and efficient codes, all available for download from http://www.netlib.org.

Numpy and arrays

Numpy provides an easy way to handle arrays in Python. The standard way to import this library is as

```
import numpy as np
```

Here follows a simple example where we set up an array of ten elements, all determined by random numbers drawn according to the normal distribution,

```
n = 10
x = np.random.normal(size=n)
print(x)
```

We defined a vector x with $n = 10$ elements with its values given by the Normal distribution $N(0,1)$. Another alternative is to declare a vector as follows

```
import numpy as np
x = np.array([1, 2, 3])
print(x)
```

Here we have defined a vector with three elements, with $x_0 = 1$, $x_1 = 2$ and $x_2 = 3$. Note that both Python and C++ start numbering array elements from 0 and on. This means that a vector with n elements has a sequence of entities $x_0, x_1, x_2, \ldots, x_{n-1}$. We could also let (recommended) Numpy to compute the logarithms of a specific array as

```
import numpy as np
x = np.log(np.array([4, 7, 8]))
print(x)
```

In the last example we used Numpy's unary function *np.log*. This function is highly tuned to compute array elements since the code is vectorized and does not require looping. We normaly recommend that you use the Numpy intrinsic functions instead of the corresponding **log** function from Python's **math** module. The looping is done explicitely by the **np.log** function. The alternative, and slower way to compute the logarithms of a vector would be to write

```
import numpy as np
from math import log
x = np.array([4, 7, 8])
for i in range(0, len(x)):
    x[i] = log(x[i])
print(x)
```

We note that our code is much longer already and we need to import the **log** function from the **math** module. The attentive reader will also notice that the output is $[1, 1, 2]$. Python interprets automagically our numbers as integers (like the **automatic** keyword in C++). To change this we could define our array elements to be double precision numbers as

```
import numpy as np
x = np.log(np.array([4, 7, 8], dtype = np.float64))
print(x)
```

or simply write them as double precision numbers (Python uses 64 bits as default for floating point type variables), that is

```
import numpy as np
x = np.log(np.array([4.0, 7.0, 8.0]))
print(x)
```

To check the number of bytes (remember that one byte contains eight bits for double precision variables), you can use simple use the **itemsize** functionality (the array x is actually an object which inherits the functionalities defined in Numpy) as

```
import numpy as np
x = np.log(np.array([4.0, 7.0, 8.0]))
print(x.itemsize)
```

Matrices in Python

Having defined vectors, we are now ready to try out matrices. We can define a 3×3 real matrix $\boldsymbol{A}$ as (recall that we user lowercase letters for vectors and uppercase letters for matrices)

```
import numpy as np
A = np.log(np.array([ [4.0, 7.0, 8.0], [3.0, 10.0, 11.0], [4.0, 5.0, 7.0] ]))
print(A)
```

If we use the **shape** function we would get $(3, 3)$ as output, that is verifying that our matrix is a 3×3 matrix. We can slice the matrix and print for example the first column (Python organized matrix elements in a row-major order, see below) as

```
import numpy as np
A = np.log(np.array([ [4.0, 7.0, 8.0], [3.0, 10.0, 11.0], [4.0, 5.0, 7.0] ]))
# print the first column, row-major order and elements start with 0
print(A[:,0])
```

We can continue this way by printing out other columns or rows. The example here prints out the second column

```
import numpy as np
A = np.log(np.array([ [4.0, 7.0, 8.0], [3.0, 10.0, 11.0], [4.0, 5.0, 7.0] ]))
# print the first column, row-major order and elements start with 0
print(A[1,:])
```

Numpy contains many other functionalities that allow us to slice, subdivide etc etc arrays. We strongly recommend that you look up the Numpy website for more details. Useful functions when defining a matrix are the **np.zeros** function which declares a matrix of a given dimension and sets all elements to zero

```
import numpy as np
n = 10
# define a matrix of dimension 10 x 10 and set all elements to zero
A = np.zeros( (n, n) )
print(A)
```

or initializing all elements to

```
import numpy as np
n = 10
# define a matrix of dimension 10 x 10 and set all elements to one
A = np.ones( (n, n) )
print(A)
```

or as unitarily distributed random numbers

```
import numpy as np
n = 10
# define a matrix of dimension 10 x 10 and set all elements to random numbers with x \in [0, 1]
A = np.random.rand(n, n)
print(A)
```

There are several other extremely useful functionalities in Numpy. Numpy contains many functions which are useful if we wish to perform a statistical analysis. As an example, consider the discussion of the covariance matrix. Suppose we have defined three vectors $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ with n elements each. The covariance matrix is defined as

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix},$$

where for example

$$\sigma_{xy} = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - \overline{x})(y_i - \overline{y}).$$

The Numpy function **np.cov** calculates the covariance elements using the factor $1/(n-1)$ instead of $1/n$ since it assumes we do not have the exact mean values. The following simple function uses the **np.vstack** function which takes each vector of dimension $1 \times n$ and produces a $3 \times n$ matrix $\boldsymbol{W}$

$$\boldsymbol{W} = \begin{bmatrix} x_0 & x_1 & x_2 & \dots & x_{n-2} & x_{n-1} \\ y_0 & y_1 & y_2 & \dots & y_{n-2} & y_{n-1} \\ z_0 & z_1 & z_2 & \dots & z_{n-2} & z_{n-1} \end{bmatrix},$$

which in turn is converted into into the 3×3 covariance matrix $\boldsymbol{\Sigma}$ via the Numpy function **np.cov()**. We note that we can also calculate the mean value of each set of samples $\boldsymbol{x}$ etc using the Numpy function **np.mean(x)**. We can also extract the eigenvalues of the covariance matrix through the **np.linalg.eig()** function.

```
# Importing various packages
import numpy as np

n = 100
x = np.random.normal(size=n)
print(np.mean(x))
y = 4+3*x+np.random.normal(size=n)
print(np.mean(y))
z = x**3+np.random.normal(size=n)
print(np.mean(z))
W = np.vstack((x, y, z))
Sigma = np.cov(W)
print(Sigma)
Eigvals, Eigvecs = np.linalg.eig(Sigma)
print(Eigvals)

import numpy as np
import matplotlib.pyplot as plt
from scipy import sparse
eye = np.eye(4)
print(eye)
sparse_mtx = sparse.csr_matrix(eye)
print(sparse_mtx)
x = np.linspace(-10,10,100)
y = np.sin(x)
plt.plot(x,y,marker='x')
plt.show()
```

Meet the Pandas

Another useful Python package is pandas, which is an open source library providing high-performance, easy-to-use data structures and data analysis tools for Python. **pandas** stands for panel data, a term borrowed from econometrics and is an efficient library for data analysis with an emphasis on tabular data. **pandas** has two major classes, the **DataFrame** class with two-dimensional data objects and tabular data organized in columns and the class **Series** with a focus on one-dimensional data objects. Both classes allow you to index data easily as we will see in the examples below. **pandas** allows you also to perform

mathematical operations on the data, spanning from simple reshapings of vectors and matrices to statistical operations.

The following simple example shows how we can, in an easy way make tables of our data. Here we define a data set which includes names, place of birth and date of birth, and displays the data in an easy to read way.

```
import pandas as pd
from IPython.display import display
data = {'First Name': ["Frodo", "Bilbo", "Aragorn II", "Samwise"],
        'Last Name': ["Baggins", "Baggins","Elessar","Gamgee"],
        'Place of birth': ["Shire", "Shire", "Eriador", "Shire"],
        'Date of Birth T.A.': [2968, 2890, 2931, 2980]
        }
data_pandas = pd.DataFrame(data)
display(data_pandas)
```

In the above we have imported **pandas** with the shorthand **pd**, the latter has become the standard way we import **pandas**. We make then a list of various variables and reorganize the aboves lists into a **DataFrame** and then print out a neat table with specific column labels as *Name*, *place of birth* and *date of birth*. Displaying these results, we see that the indices are given by the default numbers from zero to three. **pandas** is extremely flexible and we can easily change the above indices by defining a new type of indexing as

```
data_pandas = pd.DataFrame(data,index=['Frodo','Bilbo','Aragorn','Sam'])
display(data_pandas)
```

Thereafter we display the content of the row which begins with the index **Aragorn**

```
display(data_pandas.loc['Aragorn'])
```

We can easily append data to this, for example

```
new_hobbit = {'First Name': ["Peregrin"],
              'Last Name': ["Took"],
              'Place of birth': ["Shire"],
              'Date of Birth T.A.': [2990]
              }
data_pandas=data_pandas.append(pd.DataFrame(new_hobbit, index=['Pippin']))
display(data_pandas)
```

Here are other examples where we use the **DataFrame** functionality to handle arrays, now with more interesting features for us, namely numbers. We set up a matrix of dimensionality 10×5 and compute the mean value and standard deviation of each column. Similarly, we can perform mathematial operations like squaring the matrix elements and many other operations.

```
import numpy as np
import pandas as pd
from IPython.display import display
np.random.seed(100)
# setting up a 10 x 5 matrix
rows = 10
```

```
cols = 5
a = np.random.randn(rows,cols)
df = pd.DataFrame(a)
display(df)
print(df.mean())
print(df.std())
display(df**2)
```

Thereafter we can select specific columns only and plot final results

```
df.columns = ['First', 'Second', 'Third', 'Fourth', 'Fifth']
df.index = np.arange(10)

display(df)
print(df['Second'].mean() )
print(df.info())
print(df.describe())

from pylab import plt, mpl
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'

df.cumsum().plot(lw=2.0, figsize=(10,6))
plt.show()


df.plot.bar(figsize=(10,6), rot=15)
plt.show()
```

We can produce a 4×4 matrix

```
b = np.arange(16).reshape((4,4))
print(b)
df1 = pd.DataFrame(b)
print(df1)
```

and many other operations.

The **Series** class is another important class included in **pandas**. You can view it as a specialization of **DataFrame** but where we have just a single column of data. It shares many of the same features as **DataFrame**. As with **DataFrame**, most operations are vectorized, achieving thereby a high performance when dealing with computations of arrays, in particular labeled arrays.

For multidimensional arrays, we recommend strongly xarray. **xarray** has much of the same flexibility as **pandas**, but allows for the extension to higher dimensions than two. We will see examples later of the usage of both **pandas** and **xarray**.

Basic features of Qiskit

Introduction. This notebook constitutes some introductory information with relavant examples on getting started with Qiskit, an open-source software for quantum computation. A more complete overview of the available features can be found from the Qiskit documentation as well as the Qiskit textbook.

Step 0: Download the software and import packages. Here is a hands-on tutorial to install Qiskit. The first step is to create an IBM ID account. Then you will be able to use Qiskit on the cloud through the IBM Quantum Lab on your dashboard. To get started locally with Qiskit, we will use the Anaconda distribution of Python. The tutorial explains how to download Qiskit and store the account information locally.

```
# Import the relevant packages.
from qiskit import *
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
```

Step 1: Construct your quantum circuit. A quantum circuit is a computational routine which incorporates classical computations into coherent quantum operations on quantum data. The Qiskit circuit library shows us the syntax to program a quantum circuit and add quantum operations to the qubits of interest. To get started, we will introduce the class QuantumCircuit, in which we will define the circuit and explore the available built-in methods.

Define the quantum circuit.

```
from qiskit import QuantumCircuit  # You can ignore this if 'from qiskit import *' has been execu
```

Here there are two ways we can define a quantum circuit, with three qubits and three classical bits:

```
# Method 1:
qc_1p1a = QuantumCircuit(3,3)
qc_1p1a.draw('mpl')

# Method 2:
qr = QuantumRegister(3)
cr = ClassicalRegister(3)
ar = AncillaRegister(1)  # You can add ancilla qubit if it's needed, otherwise no need to include
qc_1p1b = QuantumCircuit(qr, ar, cr)
qc_1p1b.draw('mpl')
```

Combine two (or multiple) quantum circuits.

- **Method 1:** If two circuits $circ_1$ and $circ_2$ have the same number of qubits and classical bits,

then their combination could just be implemented like adding two numbers $circ = circ_1 + circ_2$. A more flexible method QuantumCircuit.compose is developed, which we can combine any two circuits and specify the qubits to compose onto. This video shows an example to implement these features. Here's another example:

```
circ_1p2a = QuantumCircuit(3)
circ_1p2a.x([0,1,2])
circ_1p2a.draw('mpl')
```

```
circ_1p2b = QuantumCircuit(2)
circ_1p2b.h(0)
circ_1p2b.cx(0,1)
circ_1p2b.draw('mpl')

circa = circ_1p2a.compose(circ_1p2b, qubits=[2,1])
circa.draw('mpl')
```

- **Method 2:** First convert a circuit into a quantum gate using to_gate method, then

append the gate to another circuit. An example is provided here. Note that the circuits with classical bits cannot be converted to gate. We can alternatively generate the *circa* as:

```
gate_x3 = circ_1p2a.to_gate()
gate_hcx = circ_1p2b.to_gate()
circb = QuantumCircuit(3)
circb.append(gate_x3, [0,1,2])
circb.append(gate_hcx, [2,1])
circb.measure_all()
circb.draw('mpl')

# Check circb and circa have the same construction.
circb.decompose().draw('mpl')
```

Generating parametrized circuits. Parametrized quantum circuits is useful in solving variational problems. To construct parameterized circuits and assign values to circuit parameters in Qiskit, we will use Qiskit's Parameter and ParameterVector (construct multiple parameters at once) class.

```
from qiskit.circuit import Parameter, ParameterVector
import numpy as np
```

- **Method 1:** use the **Parameter** class.

```
# Define your parameters.
a, b, c = Parameter('a'), Parameter('b'), Parameter('c')
# Define the quantum circuit.
circ_1p3 = QuantumCircuit(3)
circ_1p3.rx(a, 0)          # RX(a) on qubit 0.
circ_1p3.ry(b, 1)          # RY(b) on qubit 1.
circ_1p3.h(2)              # A regular gate no need for parametrization.
circ_1p3.crz(c, 0, 2)      # CRZ(c) controlled on qubit 0, acting on qubit 2.
circ_1p3.draw('mpl')

# Assign (bind) the values
circ1p3_bind = circ_1p3.bind_parameters({a: np.pi, b: np.pi/2, c: np.pi/2})
circ1p3_bind.draw('mpl')
```

- **Method 2:** use the **ParameterVector** class, where you assign all the parameters within a single vector. Therefore, we can generate the same circuit above by:

```
# Define your parameters.
p = ParameterVector('p', 3)
# Define the quantum circuit.
circ_1p3 = QuantumCircuit(3)
circ_1p3.rx(p[0], 0)          # RX(a) on qubit 0.
circ_1p3.ry(p[1], 1)          # RY(b) on qubit 1.
circ_1p3.h(2)                 # A regular gate no need for parametrization.
circ_1p3.crz(p[2], 0, 2)      # CRZ(c) controlled on qubit 0, acting on qubit 2.
circ_1p3.measure_all()        # A side note: measurement will add classical registers in your circ
circ_1p3.draw('mpl')

circ1p3_bind = circ_1p3.bind_parameters({p: [np.pi, np.pi/2, np.pi/2]})
circ1p3_bind.draw('mpl')
```

Step 2: Run your quantum simulation & Data collection. Having the quantum circuit(s) ready, we will run our quantum simulation by creating and submitting jobs to the available device. While the jobs are running, we can monitor their status. We are also able to view the jobs we have submitted and are on the waitlist through the dashboard of IBM Quantum account.

Choosing the backend to run your circuit. A useful package for simulating quantum circuits is called Qiskit Aer, which provides multiple backends for running a quantum simulation. The main simulator backend of the Aer provider is the **AerSimulator** backend, who mimics the execution of an actual quantum computer by default.

```
from qiskit import Aer, transpile  # You can ignore this if 'from qiskit import *' has been execu
from qiskit.tools.visualization import plot_histogram, plot_state_city

# List the available backends.
Aer.backends()

Here's a sample code for simulating the quantum circuit above, `circ1p3_bind`, with the `aer_simul

# Let's see the results!
# Transpile for simulator
simulator = Aer.get_backend('aer_simulator')
circ = transpile(circ1p3_bind, simulator)

# Run and get counts
result = simulator.run(circ).result()
counts = result.get_counts(circ)
print(counts)
plot_histogram(counts, title='Result for circ1p3_bind')
```

Monitor the status of your experiment and check your submitted jobs. Using Qiskit, we can also send jobs to IBM Quantum computers, and monitor their status. An overview of how to use your IBM Quantum account to access the systems and simulators available in IBM Quantum is available here.

```
from qiskit.tools import job_monitor
```

```
# Submit your job to a quantum computer 'ibmq_belem'.
# Select provider and backend.
provider = IBMQ.get_provider(hub='ibm-q')
backend = provider.get_backend('ibmq_belem')
# Run the circuit 'circ1p3_bind' and execute the job.
job = execute(circ1p3_bind, backend)
# Monitor the job.
job_monitor(job)
result = job.result()
counts = result.get_counts()
plot_histogram(counts)
```

Once the job have been submitted to a quantum computer, you will be able to check and see the it appears on the pending list on your IBM Quantum dashboard.

Submit multiple quantum circuits to a backend. If there are multiple circuits to be submitted, you can bundle the circuits in a single job to reduce the queue times. For example, if we want to the cirucit circ1p3$_b$*indandcircbtogetheronasimulator*, *wecando* :

```
simulator = Aer.get_backend('qasm_simulator')
qc_list = [circ1p3_bind, circb]       # Include all the circuits in a single list.
job = execute(qc_list, simulator)
job.result().get_counts()
```

Some visualization tools.

- If you want to visualize a quantum circuit in a LaTex document, Qiskit offers the LaTex drawer which geynerates the code you can copy and paste into a LaTex document. Another useful package for graphing quantum circuit in Latex is called Quantikz.
- Kaleidoscope provides an option to visualize the quantum states on on Bloch sphere. You can generate the plot in a Jupyter notebook.