Introduction to Quantum Computing (Intro2QC) Workshop for High School STEM Students - A Lesson Plan for Teachers and Volunteers

Workshop Outline

The Intro2QC workshop is an engaging introduction to the unintuitive yet interesting nature of quantum mechanics, the emergence of quantum computing and its increasing relevance in today's tech-driven world.

The workshop is made up of three (3) parts. The first part highlights the challenges of computation for real-world problems. The second part introduces quantum mechanics and quantum computing. The third part is an online quantum programming tutorial.

Target Learners

The target audience for this workshop is **High School Grades 10-12**.

Suggested Delivery and Time Management

The workshop should take 1hr 20 minutes to 1hr 30 minutes in total. The first part is suggested to take ~20-25 minutes. The second part should take ~30-35 minutes. The last part should be done in ~30-40 minutes.

Presentation slides are available for the delivery of the first and second parts of the workshop. For the third part of the workshop, students would be allowed to navigate to the workshop website at https://intro2qc.uvic.ca and onto the programming section.

On the workshop website, only **Programming Exercises** is meant to be completed within the timeframe of the workshop. Additional coding activities are found in **Extras**. Students are encouraged to complete the extra programming activities as well as recap on Part 1 and Part 2 of the workshop by going through the other two sections of the website on their own time.

Part 1: Why Computing is hard ...

> Let's **trick** students into understanding *Mathematical Combinatorics* and *Optimization* with real-life examples and applications without using too much technical language!

▼ Teaching Objectives

1. To use an analogy on decision-making in daily life to cover basic **Combinatorics**.
2. To deduce the formula for the total number of possible outcomes given a set of *n* **variables** and their **degrees of freedom** *{d}*.
3. To use real-life examples to illustrate the difficulty of **Combinatorial Optimization.**
4. To review the concepts of "**bits**" and "**bit strings**" and introduce "**dits**" and "**dit strings**".
5. To give a **circuit** illustration of the analogy and emphasize the amount of **computational resources** required to run the problem.
6. To illustrate **combinatorial explosion** with a graph.
7. To illustrate **limitations of classical computing** in terms of computational resource requirements and state that we need a better way to do computation.

▼ Learning Materials

Students will have access to some **interactive slides** for this quantum workshop to follow along with during the workshop.

▼ Lesson

▼ Initial Analogy

In our daily lives, we make lots of decisions. From the time we set our alarms to wake up, to what we eat for breakfast, to the order in which we get ready and so on.

Imagine we have a **calculator** (*function*) that tells us how optimal our day was. We tell this **calculator** all the decisions we made today (*input*) and it calculates for us how optimally we spent our day out of 100% (*output*). As it's calculating it takes into consideration the work

we did, the fun we had, the amount of rest we got etc. We don't (or may not) know exactly how it's calculating but we somehow get an answer.

For the first decision of the day, we have 3 options. To wake up at 5:30 am, to wake up at 6:00 am or to wake up at 6:30 am.

For the second decision of the day, we choose whether we drink coffee/ tea first or eat breakfast first.

For the third decision, we decide exactly what we want to have for breakfast. [Let's give a list of 5 things here]

Learning point #1

What are the possible combinations of decisions we can make? And how many are there in total ? 🧠

One example of a decision combination is to choose to wake up at 5:30am, choose to eat breakfast first and finally choose to eat eggs.

Another decision combo is to wake up at 6:30am, choose to have coffee first and finally choose to eat eggs.

The first decision is the first variable of the problem with **3 degrees of freedom**. The second decision is the second variable of the problem with **2 degrees of freedom**. The third decision is the third variable with **5 degrees of freedom**. The total number of decision paths we can take are

$$\text{Total combination of decisions} = 3 \times 2 \times 5 = 30$$

You must multiply the degrees of freedom for each variable together.

Learning point #2

Which decision combo gives the best result?

To get the answer to this, we need to run each of the 30 combinations through our **Day Optimality Calculator** in order to see which gives us the best result. Since we don't always have a full idea of how our calculator is computing (the full functional landscape or knowledge of the function itself, i.e. an **oracle**/**blackbox**), we must use **brute-force** and check each output one-by-one to see what gives the best results.

💡 This is only one example of an analogy.

A more fun example can be selecting **ice cream** at an **ice cream bar.** For example, you have to select the **flavour of ice cream,** the **toppings** and the **sauces.** The **blackbox function** can be a kid who tastes to see how they like selection that you made and gives it a grade out of 10.

▼ **Formula**

If you have a system with *n* variables called "x" with specified degrees of freedom for each variable "d", then the total number of combinations "T" is simply.

$$\text{Set of Variables} = \{x_1, x_2, x_3, ..., x_n\}$$

$$\text{Set of DOFs} = \{d_1, d_2, d_3, ..., d_n\}$$

$$\text{Total combinations (T)} = d_1 \times d_2 \times d_3 \times ... \times d_n$$

Learning point #3

What happens when we increase the number of variables or the degrees of freedom?

If we increase the number of variables, we have more numbers to multiply together.

If we increase the degrees of freedom, then the numbers we have to multiply together are larger.

▼ **Real-life Examples of Optimization Problems**

- **Business and Finance**: Which stocks to invest in and how much to minimize risk and optimize profit.
- **Manufacturing**: What materials to use for building considering cost, quality and human safety.
- **Shipping and Logistics**: What sets of items/orders to pack into each transport vehicle considering volume, load, cost, driver's route, delivery speed etc.
- **Science**: Given all the elements in the periodic table, which combination can help us design the ideal drug to treat a specific disease.

Learning point #4

Why Computing the best answer is hard ...

Many of these problems are still extremely difficult to answer/ compute. Once the number of variables in the problem and/or the number of degrees of freedom for each variable becomes big, then the total number of possible combinations becomes incredibly large. This is **combinatorial explosion**!

▼ Bits, Dits, Bitstrings and Ditstrings

Learning point #5

What is a bit? What is a bit string? 🧠

- A **bit (**or **binary digit)** is a unit of computational information. It can either take the value of '0' or '1'. Since there are only two possible options (i.e. **2 degrees of freedom)**, it is called **binary.**
- A **bit string** is a sequence of **bits**.

$$\text{Bitstring} = x_1x_2x_3...x_n \text{ with } x_i \in \{0, 1\}$$

- A **dit** is a unit of information with **d** possible options. A **dit string** is a sequence of **dits**.

$$\text{Ditstring} = y_1y_2y_3...y_n \text{ with } y_i \in \{0, 1, .., d-1\}$$

Learning point #6

For a string of length *n=4,* give the formula for the total possible # of (i) bit strings and (ii) dit strings. 🧠

Each bit represents a variable and each variable has only 2 options, i.e. '0' or '1'.

For a bit string of length *n=4 :*

$$\text{Total combinations of bitstrings } (T_{bit}) = 2 \times 2 \times 2 \times 2 = 2^4$$

For a dit string of length *n=4 :*

$$\text{Total combinations of ditstrings } (T_{dit}) = d \times d \times d \times d = d^4$$

For any length *n*, the total possible # of bitstrings:

$$\text{Total combinations of bits } (T_{bit}) = 2^n$$

For any length *n*, the total possible # of ditstrings:

$$\text{Total combinations of dits } (T_{dit}) = d^n$$

$$q_4$$

▼ Algorithmic Circuit

> 💡 Here, we are introducing how quantum circuits are represented. This will be useful later in Parts 2 and 3 of the workshop.

Use **horizontal lines** to represent a variable for the problem.

If you have a problem with 5 variables, we draw a stack of 5 horizontal lines.

Draw a **box** midway over the **lines** to represent a **blackbox function,** and label the box according to the analogy used.

On the left side of the diagram, we can label each line '0' or '1' to indicate the chosen input for the function.

Learning point #7

We illustrate here that if we want to test out the function to get the best answer, we'd have to re-label each line for each possible input, and run the circuit that many times, i.e. T_{bit} times.

Alternatively, if we wanted to run all the bit strings simultaneously, we'd need to increase the number of horizontal times by T_{bit} and also have access to T_{bit} blackbox functions. This is analagous to **parallel computing.**

In the first example, we require **computational time** that is proportional to T_{bit}. In the second example, we require **computational memory** that is proportional to T_{bit}.

▼ Combinatorial Explosion

Learning point #8

Here, we'll demonstrate graphically the exponential nature of these problems with plots for 2^n, 3^n,4^n etc.

With increasing *n,* the total # of combinations rises rapidly for the function 2^n and even more so with the functions for the larger degrees of freedom.

▼ The Limits of Classical Computing

Learning point #9

There is a **limit** to what we can compute classically.

Consider that in a small hospital with about 100 nurses, we want to schedule 50 nurses to work on a particular day. The total number of ways we can select those numbers are given by the formula for the **binomial coefficient** in the **binomial theorem**.

$$\text{Total nurse combinations} = \binom{100}{50} = \frac{100!}{50!50!} \approx 10^{29}$$

That's a lot! Say now that we want to select the best combination of nurses, in consideration of their level of experience, the number of days they've been working in a row, their risk for burnout, the hospital demand for that day, etc. To simultaneously store all possible combinations of nurses on a classical computer, we'd need

$$\begin{aligned}\text{Storage } (S) &= 50 \times 10^{29} \sim 10^{30} \\ &\sim 2^{102} \text{ bits}\end{aligned}$$

This is more than 5×10^5 YB, which is more than the **current global data storage capacity** (~ 149 ZB in 2024) .

> 💡 A diagram can be shown here to illustrate the levels of the different units.

Now we may not need to simultaneously store all this data, but it gives us an example of what we're working with. Thinking now of much larger and complex problems, like being able to **model the global weather, predict new materials**, or **find the best drug to treat a particular disease**, the amount of computational storage and power required explodes!

Is there a more powerful way to compute?

▼ Notes

Even though the terms "**combinatorics**", "**combinatorial optimization**" or "**blackbox function**" etc., are used here to highlight the topics being taught, we may wish to avoid using these terms altogether.

However, it would be important to use terms such as "**variables**", "**degrees of freedom**" and perhaps mention "**combinatorial explosion**".

Part 2: What's "Quantum" anyway and what's so special about it?

> Teaching quantum to high-schoolers is *easy* ... unless it's not. We're introducing students to the unintuitive behaviours of quantum objects and how they can be used for computing. Let them generate their own thoughts on how to interpret quantum mechanics and encourage as much open-ended discussions as time allows!

▼ Teaching Objectives

1. To explain how **computation** *physically works* and what are **bits** actually.
2. To introduce the concept of the **qubit** or **quantum bit** and state that the laws of quantum mechanics makes it *more powerful* for computation.
3. To demonstrate **quantum superposition** and **wavefunction collapse** using the **Schrödinger cat demo kit**.
4. To demonstrate **quantum entanglement** using the **Schrödinger cat demo kit**.
5. To explain **quantum computation, quantum gates (operations)** and **quantum circuits.**
6. To show what a **quantum computer** looks like and highlight parts of its functioning.

▼ Learning Materials

Students will have access to some **interactive slides** for this quantum workshop to follow along with during the workshop.

The Schrödinger cat demo kit contains two cat plushies. One side of each plushie features a cat that is alive, and the other side a cat that doesn't look so.

▼ Lesson

▼ The Essence of Computation

Learning point #10

Let's ask some questions to see how the students respond 🧠

> **What is computer, actually? What is and what isn't a computer / What can and can't be a computer? What about a computer allows you to compute stuff? Can a molecule be a computer? What's a simulation? We've talked about bits, but what are bits physically?**

💡 The idea here is to get them thinking in a bigger way what it means **to compute/simulate** something. Now, the answers may even tend towards the philosophical.

For example, if they respond by "*a device that allows us to calculate stuff*", ask them what they mean by 'calculate', or how does it calculate the stuff, or what's doing the calculating?

Another example, if they respond along of the lines of " input-processing-output ", ask them then if a molecule or biological cell can be a computer, as it uses environmental signals, processes, and adjusts its state accordingly.

We don't need to ask all the questions or give them any answers, this is just to generate some thought and/or allow them to recall what they may have learned previously in a Computer Science class.

Let's go back to a more conventional definition for now.

> A **computer** is an electronic device that processes information (data) to perform specific tasks. It takes input, processes it using instructions (programs), and produces output. Computers are designed to follow precise instructions step by step, making them powerful tools for solving problems, storing data, and automating tasks. - *ChatGPT*

💡 A diagram of **Von Neummann architecture** is shown as support.

Learning point #11

About information and data

From the ChatGPT definition, we notice that the words 'information', 'data' as well as 'tasks' and 'instructions' are emphasized. We also want to go back to the definition of a **bit** as a "*unit of computational information*". Some might say it's the "most basic unit of **data**".

But what is **information** and what is **data**? When you think about it, information is **everywhere**, whether from texts that we read, to audio that we listen to, the photos on our phone. **Data** is a way to record the **state** of something, and **information** is the interpretation of that data.

Let's say we have an instrument that measures the intensity of light hitting it. One second, it reads '100', and the other second it reads '150'. The **data** are *the assigned numbers*, but the **information** is the *interpretation* of what those numbers represent/mean.

Learning point #12

About bits

So we've talked about **bits** as these (almost) abstract units of **data** and **information**. **But bits are also physical. Digital bits** represent the **information** on the **state** of some **physical** thing.

Examples:

- We can read out the voltage of tiny **integrated circuits (ICs)** or the states of **transistors**. If the voltage level is beyond a particular threshold, it is '**high'** or '**1',** and if it is below that threshold, it is '**low'** or '**0'.** If a **transistor** is on, it is in **'1'** state and if it is off, it is in the **'0'** state.
- **Hard drive technology** uses magnetic material and reads out the magnetic orientation of defined regions of the material (called **magnetic domains**).
- **CD technology** uses **pits** and **lands** on a **polycarbonate material** to encode information. In the read out, a **laser** is focussed on the material as there is a difference in the reflection off the pits and the lands. The state **'1'** is read when there is a change from *pit-to-land* or *land-to-pit*, and a state **'0'** if there is no change.

Learning point #13

About computers

The fact that *the knowledge of the state of some physical system* represents **information** that may be used for **computation** really puts things into perspective as to what things can or can't be used as a **computer.** Technically, a **computer** is anything that is able to do **computation**, i.e. receive input, process the input according to some instructions, and give an output.

Outside of our conventional understanding of a computer, here are *other examples.*

- **Embedded systems**, e.g. cars, medical devices, wearable technology, home appliances (refrigerators, washing machines, thermostats)
- **Analog computers** can use *mechanics* or *hydraulic* properties to compute.
- **The brain** because, of course.
- **DNA computers** use strands of DNA with the sequence of nucleotides [adenine (A), thymine (T), cytosine (C), guanine (G)] to compute via biochemical reactions. A and T

are ‘0’ and C and G are ‘1’.

So yes, **molecules, chemical** and **biological systems** might be used for computing.

There is a key point here though. Whilst they may be able to process input information and give **output**, they should also be **programmable.**

1. **Input encoding**
2. **Output decoding**
3. **Information processing**
4. **Programmability**

▼ **The Quantum Bit**

Thankfully, yes! Let’s define the **qubit** or **quantum bit** and compare it to the **classical bit.**

Learning point #14

The classical bit versus the quantum bit

The best way to think about it is that a **classical bit** can only be in one of the discrete states, i.e. ‘0’ or ‘1’.

On the other hand, the **quantum bit** or **qubit** can simultaneously be in the state ‘0’ and the state ‘1’.

Why? It’s because of **Quantum Mechanics.**

In fact, if we think of a bit in state ‘0’ as **an arrow pointing to the north pole of a sphere** and a bit in state ‘1’ as **an arrow pointing to the south pole of a sphere**, a quantum bit can point **to any point on the sphere.**

This is the **Bloch Sphere** representation for the state of a quantum bit. The **state of a qubit** is represented as an arrow pointing to a given point on the sphere.

This means that the **dimensionality of the state of a qubit** is higher than **the dimensionality of the state of a bit**, which means that it can **do more** in terms of computation.

▼ **Properties of Quantum Mechanics**

Because qubits are **quantum**, let’s learn a couple **quantum properties.**

Learning point #15

Quantum Superposition and State Collapse

Superposition is exactly what was described before. It's the ability of a qubit to be in both the '0' and '1' state at the same time.

But here's a *strange detail...* A qubit can only be in *superposition* **once we're not watching it!**

What does that even mean?

> 💡 **Schrödinger Cat Demo #1**. We have a cat plushie with one side that is alive and the other side that is dead (Schrodinger's cat). We place the plushie in a box [with sides missing for visual and practical purposes] and we use our hands to move the cat around so that it switches between being alive and dead many times, whilst we aren't looking. Once we decide to look, we stop moving the cat and select a side. We repeat this, showing possibly different outcomes each time. **Ask the students to describe what happens once we look** 🧠 They would most likely say that the cat chooses one state once we look. **Then, ask them how could we describe the state of the cat before we look (according to what we see when we look)** 🧠 The answer is that we can **count** and **take statistics** on the number of times we see the cat alive versus the opposite outcome.

The truth is that the higher dimensional state of the qubit **collapses** into a **classical state** once we observe it. In order to describe the state of the qubit before collapse, we must have multiple qubits prepared in the same way, make multiple observations, and take statistics of those observations in order to recover the qubit state prior to collapse.

$$\text{Qubit state} = c_0 \times \text{state '0'} + c_1 \times \text{state '1'}$$

Here, the numbers c_0 and c_1 are *related* to the probabilities for observing that state.

And why does observing a qubit cause it to collapse?

The act of **observing** is actually a **measurement** of the system. **Measurements** disturb **quantum systems** causing them to *"lose their quantumness".*

> 💡 In actuality, whilst **most measurements** permanently cause **state collapse**, there are some types of **measurements** that can preserve the quantum state of a qubit. However, in very many cases we are talking about **strong measurements**, rather than **weak** or **indirect measurements** (which usually only gives partial information).

Learning point #16

How could the state of a qudit or quantum dit be represented 🧠

A **qubit** can only collapse onto 1 of **2 possible states**. A **qudit** can collapse onto 1 of **d possible states.**

$$\text{Qudit state} = c_0 \times \text{state '0'} + c_1 \times \text{state '1'} + c_2 \times \text{state '2'} + ... + c_{d-1} \times \text{state 'd-1'}$$

Learning point #17

Quantum Entanglement

When qubits interact, something *even stranger* happens...

> 💡 **Schrödinger Cat Demo #2**. We have two cat plushies and a box that separates into two. We have the plushies together in the one (separable) box and pull the plushies apart into each there own box to opposite ends of the space/table/room. They are still "vibing" (switching between alive and dead). However, once each person looks (at the same time), the plushies' states collapse. Make sure to coordinate so that the state collapsed for both plushies is the same for each repeat (both dead or both alive). **Ask the students to describe what happens once we look 🧠**

The outcomes of each plushie becomes **correlated** because of the interaction. This is called **entanglement.**

Why is this strange?

It's strange because the final measured outcome was not decided until the moment of observation. The qubit states collapse probabilistically into their measured state. For the measurements to be correlated in this way, one might think that they had to "communicate" at the exact moment of measurement what state they were about to collapse to. However, physics forbids this explanation as this would indicate information travelling faster than the speed of light between the qubits.

The current theory, which is increasingly being demonstrated as true, is what we call **quantum non-locality.** It is the idea that no matter how far we separate the qubits after the interaction, their states can no longer be described independently of the other. It's like they behave as "one" system, no matter the space-time separation. **What in the... telepathy?!**

This theory might sound a bit crazy, but experimental demonstrations of quantum non-locality was the subject of the 2022 Nobel Prize in Physics.

Learning point #18

How might the entangled state in the cat demo be written 🧠

The state must be described in terms of the two qubits, in such a way that only the same result is obtained for both.

$$\text{Entangled state} = c_{00} \times \text{state '00'} + c_{11} \times \text{state '11'}$$

In this example, notice that the states '01' and '10' are not possible because of the entanglement.

> 💡 Note, however that, qubits can also be entangled to always give opposite states. In that example, the states '01' and '10' are the only possible outcomes and the states '00' and '11' are forbidden. [Whist there are different *degrees of entanglement,* we will only apply the definition of the *maximally-entangled states* here.]

▼ Quantum Computation

Learning point #19

Quantum Computing: How & Why

Here we'll explain how quantum systems can be used for computing by e**ncoding information** about a problem we'd like to solve onto them, **manipulating their quantum states** and **measuring their states**.

Quantum programs are developed for carrying out instructions of a compuational task on a quantum computer.

A **quantum algorithm** is a *recipe* to do a particular task on a quantum computer.

Quantum algorithms can be expressed as a **quantum circuit.** A **quantum circuit** is a sequence of **quantum gates** that tells us exactly how the quantum computer should manipulate the individual and collective states of the qubits during the computation, much like how **logic gates** operate on classical bits.

A **quantum algorithm** that is able to solve problems more efficiently than a classical computer is said to demonstrate a **quantum advantage**.

The **goal of quantum computing** is therefore to develop quantum algorithms that fully take advantage of quantum properties (superposition, entanglement and interference) to

achieve a quantum advantage for solving particularly difficult problems.

Learning point #20

Physical Qubits

Quantum bits are quantum objects that we can relatively easily initiate, manipulate and measure their states. Such quantum objects include atoms, ions, photons, and superconducting circuits (artificial atoms).

Learning point #21

The Advantage of Superposition?

The *beauty of quantum computing* is in the fact that we can place each qubit into **superposition.**

We demonstrate a quantum algorithmic circuit using 4 horizontal lines as qubits which are the inputs to a quantum blackbox.

What is happening if we place each qubit into superposition of 0 and 1 🧠

All possible combinations of binary variables are possible if each qubit is in superposition:

{'0000', '0001', '0010', '0011', '0100', '0101', '0110', '0111', '1000', '1001', '1010', '1011', '1100', '1101', '1110', '1111'}.

Therefore one major way quantum computing can bring about an advantage is by the ability to compute on multiple bit states at once.

There are $2^4 = 16$ combinations. **That means that the state of the qubits are simultaneously in these 16 combinations!** In classical computing, we'd need to either run the 16 different combinations using 4 bits one at a time or use $16 \times 4 = 64$ bits and 4 separate calculators to run all combinations at once **Here, we need only 4 qubits to compute on all 16 binary combinations at the same time!**

▼ Notes

The essential teaching points in this section include:

- The definition of a quantum bit and its comparison to the classical bit
- How the state of a quantum bit can be written
- Quantum superposition
- Wavefunction/State collapse
- Quantum entanglement and an example for writing a two-qubit entangled state
- What quantum computing is about

All other points can be omitted and is subject to available time.

Part 3: Quantum Programming 101

> You don't need to know how to program to start quantum programming! Let's cover the basics here.

▼ Teaching Objectives

1. To explain the concept of a **quantum circuit.**
2. To explain what **Qiskit** is.
3. To show students how to launch the programming interface on the workshop website.
4. To show how to import **Qiskit** and other modules into a Python code editor for building quantum circuits.
5. To show how to check the version of Qiskit we have access to.
6. To show how to construct a **single-qubit circuit** using Qiskit and how to display the circuit diagrams.
7. To show how to add **quantum gates** to circuits.
8. To show how to **apply measurement** to a quantum circuit.
9. To show how to define **backends** and run **jobs** on the backend.
10. To show how to fetch results from circuit jobs.
11. To show two ways to display results from jobs: **counts** and **histograms**.
12. To show ways to display the **quantum state** for a circuit on a simulator backend: **statevector** and **Bloch sphere**.
13. To demonstrate the actions of the **X-gate** and **H-gate** on a qubit on the Bloch sphere.
14. To demonstrate **multi-qubit circuits** and the concept of **equal superposition of states**.
15. To demonstrate the action of the **CX-gate.**
16. To demonstrate **quantum entanglement**.

▼ Learning Materials

Students will now log onto https://intro2qc.uvic.ca and navigate to the **Quantum Programming** section to complete this part of the workshop.

▼ Lesson

▼ **Quantum circuits**

Learning point #22

Quantum Circuit

A **quantum circuit** is essentially *set of instructions* sent to a quantum computing platform which tells what to do on the quantum computer. We define the number of quantum bits we would like to use, the types of quantum gates/operations we'd like to apply and what qubits we are measuring.

▼ **Quantum Programming with Qiskit on Intro2QC website**

Learning point #23

Qiskit

Qiskit is a quantum programming software development kit (SDK) in Python programming language, which allows us to construct quantum circuits, simulate them or send them to an actual quantum computer, and retrieve the results.

It is the most popular quantum programming tool.

Learning point #24

Navigating to the programming tutorial

On the **intro2qc** website, either:

- click on the **Teach me quantum programming** box in the *Workshop outline* section of the landing page,
- or on **Getting Started** on the left navigation panel under *Quantum Programming.*

Ensure that all students are on this page before proceeding.

Learning point #25

Launching the programming interface

On the **Programming Exercises** page, launch the live programming interface as per instructions on the **Getting Started** page.

Once the kernel status says **ready**, the cells may be run.

Learning point #26

Importing modules and checking the version of Qiskit

Click **run** on the first cell to import the necessary modules and functions.

The output of the first cell is the version of Qiskit being used.

The command for this in the first cell is:

```
qiskit.__version__
```

Learning point #27

Creating, naming and drawing a circuit

The function in Qiskit for creating a circuit is `QuantumCircuit()` . In the brackets, we specify the number of qubits that we want our circuit to have.

We also have to name our circuit.

> 💡 Ask the students to alter the name of the quantum circuit from *qcworkshop* to whatever they want. Wherever *qcworkshop* is written in the tutorial, they must now replace with their chosen circuit name. Shorter names are easier to remember and type.

Click **run** on the second cell to create the circuit.

```
bob = QuantumCircuit(1)
```

To draw the circuit, we type the name of our renamed circuit and add `.draw(output='mpl')` to it and click **run**.

```
bob.draw(output='mpl')
```

> 💡 The ‘mpl’ output is much nicer format than the plain format. Students can test this by leaving the brackets empty and rerunning the cell.

Learning point #28

Applying a single-qubit gate

In the tutorial, we would like to apply a basic single-qubit gate which is analogous to the NOT logic gate in classical computing.

The **X-gate** flips the state of the qubit. We apply the X-gate by typing the name of our circuit and adding `.x(0)` which says that we are applying the X-gate to the first qubit. In Python, we start counting from 0, hence why `.x(0)` and not `.x(1)`.

```
bob.x(0)
```

Learning point #29

Applying a measurement gate

To measure all the qubits in our circuit, we simply tack on `.measure_all()` to the name of our circuit.

```
bob.measure_all()
```

Learning point #30

Defining a backend

We have to say what we'd want to run our circuit on. If we had direct access to a quantum computer, we would define it here. For now, we are just using a simulator, which is a way to mimic what would happen on an ideal quantum computer.

We will use `AerSimulator()`.

```
my_backend = AerSimulator()
```

Learning point #31

Running the circuit

To run a circuit, we have to create a job instance.

For the job, we need to take the backend name, and use the function `run()` to run the circuit for a number of times. The number of times we need to run the circuit is called **shots**.

```
my_job = my_backend.run(bob, shots=2000)
```

Learning point #32

Fetching results as counts

The results of the computation is now stored in bob_job, so we need to retrieve it.

We apply `.result()` to the job name and then `.get_counts()` .

```
bob_results = my_job.result()
counts = bob_results.get_counts()
```

Alternatively, if we are only interested in the counts and not any other results, we can simplify this to a one-line code.

```
counts = my_job.result().get_counts()
```

Learning point #33

Plotting a histogram of counts

With `counts` now stored, we can plot a histogram of results with `plot_histogram(counts)` . To specify a colour for the histogram:

```
plot_histogram(counts, color="hotpink")
```

💡 Let students experiment with different colour names to see if they are available.

Learning point #34

Displaying the statevector

We can use the function `Statevector()` to get the statevector of a circuit.

It is **important** to know that in order to display the state in Qiskit, we must not apply a measurement gate to our circuit.

For this reason, we recreate a new circuit at this point of the tutorial. Students can simply click **run** on the next cells. They do not need to rename the quantum circuit, unless they wish to.

```
circuit = QuantumCircuit(1)
circuit.x(0)
circuit.draw('mpl')
```

```
circuit_state = Statevector(circuit)
circuit_state.draw("latex")
```

Learning point #35

Displaying the Bloch sphere for a single qubit

The Bloch sphere depiction of the quantum state can be shown using `plot_bloch_multivector()` on the saved state.

Note that this option is not available for multi-qubit states, and representing quantum states in this way becomes more difficult with increased dimensionality.

```
plot_bloch_multivector(circuit_state)
```

Learning point #36

Visualizing state transitions for single-qubit gates

A cool function that allows us to see what happens to the qubit's state when we apply a gate is `visualize_transition()` .

> 💡 Students can now proceed to **Coding Activity 1** where they must create a new quantum circuit and use this function to see what happens to the state of a qubit when two X-gates are applied consecutively.

An example of the circuit that students must create to first display the state is:

```
two_x = QuantumCircuit(1)
two_x.x(0)   # Apply first X-gate
two_x.x(0)   # Apply second X-gate
two_x_state = Statevector(two_x)   # Get statevector
plot_bloch_multivector(two_x_state) # Show Bloch sphere
```

To show the transition:

```
visualize_transition(two_x,trace=True, fpg=30, spg=2)
```

Learning point #37

Creating superposition with the H-gate

The **H-gate** allows us to create superposition. To apply it, we simply apply `.h(0)` to a single-qubit quantum circuit.

```
h_circuit = QuantumCircuit(1)
h_circuit.h(0)
```

💡 Allow students to explore the action of the H-gate by running the next cells in the tutorial and ask them to test what happens when the number of shots are increased and decreased. For example, they can try `shots=100` or `shots=100000` when running the circuit.

Learning point #38

Creating an equal superposition of states with a multi-qubit circuit

We are now creating a circuit with 3 qubits and placing each qubit into superposition. We can add the same gate to a set of qubits by indicating them in square brackets [] when applying the gate

```
big_circuit = QuantumCircuit(3)
big_circuit.h([0,1,2])          # adding an h-gate to qubits 0,1 and 2.
big_circuit.measure_all()
big_circuit.draw("mpl")
```

An equal superposition of states is created in this way, as all possible bit combinations are obtained in the results.

Learning point #39

Generating a two-qubit entangled state with the CX-gate and the H-gate

The **CX-gate** is analogous to the CNOT gate from classical computing.

💡 Students can now proceed to **Coding Activity 2** where they must create a new quantum circuit to discover how the CX-gate works.

Such a circuit looks is created with the following code:

```
circ = QuantumCircuit(2)
circ.x(0)
circ.cx(0,1)
```

```
circ_state = Statevector(circ)
circ_state.draw("latex")
```

Quantum entanglement can be created with the gates that have now been introduced.

> 💡 Students can now proceed to **Coding Activity 3** where they create a two-qubit entangled state using only an H-gate and CX-gate.

The entangled state is generated as follows.

```
entangled = QuantumCircuit(2)
entangled.h(0)
entangled.cx(0,1)

entangled_state = Statevector(entangled)
entangled_state.draw("latex")
```

```
entangled.measure_all()
entangled_job = my_backend.run(entangled, shots=2000)

entangled_results = entangled_job.result()
entangled_counts = entangled_results.get_counts()

plot_histogram(entangled_counts, color="slateblue")
```

▼ Notes

Whilst all the code is provided for the tutorial, students should change name of the circuit used in *Sections 1-6* in **Programming Exercises** to be able to follow exactly what is happening at each step of the code.