Quantum Computing Notes

Prof. Hamzeh Roumani
EECS, York University, Toronto, Ont., Canada

SINGLE QUBIT SYSTEMS

1. The Qubit

The *qubit* (short for **qu**antum **bit**) is the smallest unit of information in the quantum world. It is much richer (in terms of information content) than the bit of the classical world, which can only be in one of two states (yes/no, true/false, 0/1, …). But once a qubit is measured (information about its state leaks out of the system being studied), its state collapses, and it reduces to a bit, thereby losing all its information content except for the 0 or 1 outcome.

In our Apple gedanken, the qubit is *person* while inside rooms and corridors. It could have all kinds of thoughts about apples while in there, but once we observe from which door it exited, it collapses to either a yes or a no person, i.e. the qubit becomes a bit.

A qubit can be realized using any two-state quantum system. Examples include: a photon (with transmitted versus reflected states); a photon (with horizontal versus vertical polarization); an electron or a nucleus (with spin up versus down); and a superconducting loop with Josephson junctions (with clockwise versus counterclockwise current).

2. The State

The state of a qubit (its information content) before a measurement is a superposition of the two possible outcomes of that measurement. There are a number of mathematical structures capable of representing such a two-entity superposition (a point in a plane, a vector, a column vector, an ordered pair, etc.) and we will go with a **vector**. And as noted in the Apple Gedanken experiment, this vector must have a **norm** (length) associated with it (so we can normalize it) and must be scalable by a **complex** number (so we can represent speakers). Finally, in order to accommodate superposition, we need to be able to **add** these vectors.

The vector that represents a qubit in state s can be written as **s** (in bold), or as s with an arrow on top, or using Dirac's bra-ket notation, which we will adopt throughout. In this notation, the vector is written as a **ket** |s>.

A set of elements (vectors) that have an addition operation + (defined with the typical commutative group properties), a multiplication * by a scalar field (complex numbers), and a distributive behaviour of * over + is known as a **vector space**. And if we augment this space with a dot product that allows us to define a distance metric (and thus norm) in a complete way, we end up with what is known as a **Hilbert** space.

Basis

Since our qubit is a two-state system, our vector space is two dimensional, i.e. any vector in it can be written uniquely as a linear combination of two vectors that span the space. These two vectors are known as the **basis** of the vector space. We usually work in the so-called **standard** (or laboratory) basis in which the two vectors are denoted by **|0>** and **|1>**. The state |s> of our qubit can thus be written as:

$|s> = \alpha|0> + \beta|1>$

where α and β are complex numbers known as the **components** (or **projections**) of the state along the two basis vectors. In this basis, we can also represent |s> using the column vector:

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

If we take the adjoint of |s> (transpose and complex conjugate), we get the row vector:

$$\begin{pmatrix} \alpha^* & \beta^* \end{pmatrix}$$

which is denoted using the bra-ket notation with the **bra**: <s|.

It should be noted that our vector space has infinitely many bases. Any two unit-length vectors that are not multiples of each other can be used to form a basis.

Dot Product

Given two vectors, |s> (with components s_1 and s_2) and |t> (with components t_1 and t_2), their **inner product** (aka **dot product**) denoted by <s|t> (or s · t) is defined as the complex number:

$$s\,.\,t = < s|t > = s_1^* \times t_1 + s_2^* \times t_2$$

Using this, one can easily show that:

- <s|s> is real and non-negative
- $<s|t> = <t|s>^*$
- $<s|(a|t_1> + b|t_2>) = a<s|t_1> + b<s|t_1>$

The first property allows us to define the norm (or length) of a vector as:

$$\|s\| = \sqrt{< s|s >}$$

A vector is **normalized** if its length is 1, and two vectors are **orthogonal** if their dot product is 0. If two vectors are both normalized and orthogonal, they are **orthonormal**. Our basis vectors are always chosen to be orthonormal. Hence, for the standard basis, we have:

<0|0> = <1|1> = 1 and <0|1> = <1|0> = 0

And as noted earlier, there are infinitely many bases in the vector space and all of which are orthonormal. Let |x>, |y> be the vectors of a non-standard base, and note the followings:

- **By Orthonormality**
  <x|x> = <y|y> = 1 and <x|y> = <y|x> = 0

- **Change of Basis**
  If |s> = α|0> + β|1> then we can also express it in the (x,y) basis as: |s> = α'|x> + β'|y>, where α', β' are the components (or projections) of |s> on |x> and |y>. To compute them, multiply both sides of the first expression by the bra <x| and again by <y|:

  |s> = α|0> + β|1> => <x|s> = α<x|0> + β<x|1> => α' = α<x|0> + β<x|1>
  |s> = α|0> + β|1> => <y|s> = α<y|0> + β<y|1> => β' = α<y|0> + β<y|1>

  Hence:

$$\begin{pmatrix} \alpha' \\ \beta' \end{pmatrix} = \begin{pmatrix} < x|0 > & < x|1 > \\ < y|0 > & < y|1 > \end{pmatrix} \times \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

- **Superposition is Basis Dependent**
  If |s> = α|0> + β|1>, with α and β ≠ 0 then |s> is a superposition state. But if we express the same state in the x,y basis, |s> = α'|x> + β'|y>, then it is possible that either α' or β' is 0, which means |s> is *not* a superposition in the eyes of the (x,y) basis.

Global Phase

A vector |s> = α|0> + β|1> that represents a qubit must be normalized. Hence:

$$< s|s >= \alpha^* \times \alpha + \beta^* \times \beta = |\alpha|^2 + |\beta|^2 = 1$$

We can build this constraint in by writing each of α and β in terms of modulus and argument and expressing the two moduli as the sin and cos of some angle (so the sum of their squares will be 1). And by pulling out a global phase, we end up with:

$$|s >= e^{ig} \times \left[cos(\theta/2)|0 > + sin(\theta/2)e^{i\varphi}|1 >\right]$$

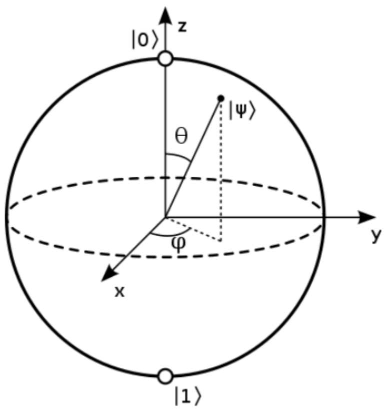
The global phase **g** has no physical meaning (it has no implication) and can be dropped. In other words, all state vectors that have the same θ and φ are considered equivalent regardless of the values of g in them. We will from now on set g to zero to represent all these vectors, and hence, the most general qubit state (in the quotient space) is represented by:

$|s> = cos(\theta/2)|0> + sin(\theta/2)e^{i\varphi}|1>$

Observe the number of parameters (and hence the entropy or the information content) needed to specify a qubit: rather than just 0 or 1 for the state of a bit, a qubit needs two real numbers: $\theta \in [0,\pi]$ and $\varphi \in [0,2\pi)$.

Bloch Sphere

Since the qubit state can expressed as a linear combination of |0> and |1>, one is tempted to visualize it as a 2D vector with |0> being the unit vector of the horizontal x-axis and |1> the unit vector on the vertical y-axis. This picture would be valid if both components are real but is very deceptive otherwise. We will therefore employ a different geometric view, the **Bloch sphere**, as shown. The sphere has a radius of 1, and its surface represents all the possible qubit states. The qubit state vector (from the centre to a point on the surface) is normalized and makes the two angles, θ and φ, as shown. Notice how |0> and |1> are not perpendicular in this view—they are along the +z and -z axis.



In addition to the standard basis, we sometime use the so-called Hadamard basis whose two vectors are:

$|+> = (|0> + |1>) / \sqrt{2}$
$|-> = (|0> - |1>) / \sqrt{2}$

Comparing these expressions to the one describing the most general qubit vector reveals that they correspond to θ=π/2 in both and $\varphi=0$ for |+> and π for |—>. The Hadamard basis vectors in the Bloch view are thus unit vectors along the +x and -x axis.

The Bloch Sphere provides a visual account of the difference between a qubit and a bit vis-à-vis information content: the qubit can be any point on the surface, but once measured in the standard basis, it reduces to a bit—a point either at the north or the south pole of the sphere.

- Prove that <s|t> = [<t|s>]*
- Prove that <s|s> is real and non-negative
- Prove that: <u | (α|s> + β|t>) = α <u|s>) + β <u|t>
- Consider the two vectors: |***i***> = (|0> + *i*|1>)/√2 and |—***i***> = (|0> — *i*|1>)/√2
  Prove that they form an orthonormal basis and locate them on the Bloch sphere.
- Prove that these two vectors represent the same state:
  |0> and *i*|0>.
- Prove that these two vectors represent the same state:
  (*i*|1> — |0>)/√2 and (*i*|0> + |1>)/√2.
- Prove that these two vectors represent the same state:
  (|+> + |—>)/√2 and |0>.

- Is this state a superposition in the standard basis?
  (|0> – i|1>)/√2
- Is this state a superposition in the standard basis?
  √3/2|+> – 1/2|–>
- Prove the change-of-basis matrix is unitary.
- Prove that any transformation can be thought of as a change of basis. Conclude from that a way to check if a matrix is unitary.
  Hint: the rows (and columns) are the vectors of two bases, and these bases are orthonormal.

3. Transformations

Devices that modify the qubit state without learning anything about its current state (no leak of information) are transformations that take one vector to another within the Hilbert space. In the Apple Gedanken experiment, the rooms are such devices. In physical realizations, beam splitters, glass plates, magnetic fields, or microwave pulse (depending on how the qubit is realized) are used to modify the state of the qubit, e.g. take it from |0> to |1> or from one superposition to another. The time evolution of a closed system is also a state transformation.

Mathematically, transformations appear as operators in the state space, and once a basis is chosen, these operators can be represented by matrices.

Linearity

All transformations in the quantum world are **linear**, i.e. given a transformation T,

T (α|0> + β|1>) = α**T**|0> + β **T**|1>

Linearity is very useful in quantum computing because it tells us how a device will act on a qubit in a superposition state if we know how it acts on classical bits (the basis vectors). Linearity also justifies using matrices to represent transformations.

Unitarity

All transformations in the quantum world preserve the normalization of the state, and this is achieved by requiring unitarity. A transformation matrix T is **unitary** if:

T Unitary ⇔ $T^\dagger T = T T^\dagger = I$

where $\mathbf{T}^\dagger$ is the adjoint (complex conjugate of the transpose) of T. And from the above equality, we see that $\mathbf{T}^\dagger$ is no other than $\mathbf{T}^{-1}$, the inverse of **T**.

Reversibility

All transformations in the quantum world are reversible. Mathematically, this is a consequence of unitarity because the inverse of a unitary matrix exists, so we can apply it to the output of the device to reproduce its input. For computing, this is quite a contrast to classical gates (the building blocks of computers), such as OR and AND, which are non-reversible.

Complex Rotation

Since all vectors in the state space are normalized, and since all transformations are unitary, the length of any transformed vector remains 1. Hence, all transformations are actually **rotations**. It should be kept in mind however, that this is a complex space, so these rotations do not look the same (geometrically) as O(3) rotations. Keep in mind that there 4 different spaces: the complex Hilbert space where linear algebra reigns; the projective space (all vectors normalized and only one vector per global phase) where the qubit states live; the parameter space (θ,φ) where the Bloch sphere allows us to visualize the states; and our "real" space where the physical qubits (photons, electrons, …) reside.

4. Measurement

Unlike a transforming device, a measuring device seeks to glean information about the state of the system, and it does so in an odd way. Specifically:

- Each measuring device has its own basis for the vector space. Let $|e_1>$ and $|e_2>$ denote the two basis vectors of that basis.
- Given a vector $|s>$ to be measured, the device computes its components α and β along the $|e_1>$ and $|e_2>$.
- The system is forced to collapse to (i.e. become) either $|e_1>$ or $|e_2>$. The choice is random but not uniformly so. It chooses $|e_1>$ with probability $|\alpha|^2$ and $|e_2>$ with probability $|\beta|^2$.

The following key points must be kept in mind vis-à-vis measurement and collapse:

A. Once a measurement is made, the collapsed state of the system **does not change** (assuming the system does not interact with anything). In other words, no matter how many times the measurement is repeated, the outcome will stay the same.

B. **Randomness is intrinsic in Nature**. Unless the state vector happened to be along one of the basis vectors of the measuring device (in which case the outcome is certain to be the state), the outcome is truly random and has a distribution derived from the components of the state vector along the vectors of the measurement basis. This is in stark contrast with our classical world in which randomness is a convenience that covers our inability to know the initial condition exactly or to cope with complexity.

C. **We cannot determine the state of a given qubit exactly**. In fact, w*e cannot even determine it probabilistically* because once we measure, the state collapses. The only way out would be to make many copies of that state (an ensemble) and then measure all the copies to obtain stats about the various outcomes. Alas, it is not possible to copy a state as shown by the following theorem:

The No-Cloning Theorem
It is impossible to copy a given unknown quantum state. The proof involves multi-qubit vectors, which we haven't covered yet, so we will return to this proof. Note that you can of course copy a *known* state (by constructing it) but not an unknown one.

The post-measurement collapse and the inability to copy states sounds ominous for quantum computing. For even if we benefited from the rich information content and the rich parallelism of the quantum world, the results of any computation we do there is seemingly inaccessible to us. We will see later that clever transformations in the quantum world before we measure, can put the result state in a form that is useful to us even after it collapses.

D. **We cannot always distinguish two given states exactly**. Even if we know exactly the states of two qubits, and we just want to know which is which, we cannot perform a measurement to tell them apart in all cases. If the two vectors are identical, then there is nothing to do. If they are orthogonal, then we measure using them as basis, and in that case, the outcomes would be 0 for one of them, 1 for the other. In all other cases (i.e. vectors neither parallel not perpendicular), the outcomes are probabilistic and cannot distinguish the two exactly.

E. A measuring device is designed to measure some *observable*, e.g. the spin of an electron. Mathematically, that observable is represented by a Hermitian matrix (one that is equal to its adjoint). The **eigenvectors** of that matrix define the device's basis. The act of forcing the state to become one of the basis vectors is thus by a projection operator (or a matrix). The outcome of the measurement is the eigenvalue of the chosen eigenvector. For example, to force the state to be along |0>, we use the projector |0><0| or the matrix [1,0], [0,0]].

- A qubit in state (2|0> + |1>)/√5 is measured in the standard basis. Show that the outcome of the measurement will be the bit 0 80% of the time. But if we measure in the Hadamard basis then the state will collapse to |+> 90% of the time.
- What is wrong with this argument: "A qubit in pure state |0> is measured in the +/− basis. Given the 1/√2 amplitude, the measurement outcome is certain to be ½"?

5. Gates

When transformations are performed as part of a computation (an algorithm), we call them gates and use circuit-derived terminology such as input and output. Hence, gates are always linear, unitary, and reversible, and are represented by operators or matrices.

To specify the function or purpose of a gate, we often specify how the gate acts on a classical bit and then invoke linearity to determine how it acts on a qubit. To obtain the matrix of the gate, we can do so by inspection, given the gate's function, or use projection operators (outer products). Here is an example:

Suppose we need to transform a state to its negation. In the classical bit world, negating a bit means flipping it from 0 to 1 or from 1 to 0. In the quantum world, we leverage linearity to define the negation of $|s\rangle = \alpha|0\rangle + \beta|1\rangle$ as $|s\rangle = \alpha|1\rangle + \beta|0\rangle$. How do we find the matrix of this transformation? We will do that by expressing the transformation as an operator first and then obtaining the matrix. We note that multiplying a state by $|0\rangle\langle 1|$ will project its $|1\rangle$ component on $|0\rangle$, and similarly, $|1\rangle\langle 0|$ will project its $|0\rangle$ component on $|1\rangle$. Hence, if we do both, the state will be negated. Hence, the sought operator is:

$|0\rangle\langle 1| + |1\rangle\langle 0|$

These two outer products lead to the matrix:

$$\begin{pmatrix}1\\0\end{pmatrix}\begin{pmatrix}0 & 1\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix}\begin{pmatrix}1 & 0\end{pmatrix} = \begin{pmatrix}0 & 1\\0 & 0\end{pmatrix} + \begin{pmatrix}0 & 0\\1 & 0\end{pmatrix} = \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}$$

This is the **Pauli X matrix**, and it behaves like a NOT. Here are the three Pauli matrices:

$$I = \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}, X = \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}, Y = \begin{pmatrix}0 & -i\\i & 0\end{pmatrix}, Z = \begin{pmatrix}1 & 0\\0 & -1\end{pmatrix}$$

The XYZ Pauli matrices can be used to effect state rotations on the Bloch sphere. Specifically, the following operators rotate the state by an angle θ about the X, Y, and Z axis, respectively.

$$R_x = e^{-iX\theta/2} = \begin{pmatrix}\cos(\theta/2) & -isin(\theta/2)\\-isin(\theta/2) & \cos(\theta/2)\end{pmatrix}$$

$$R_y = e^{-iY\theta/2} = \begin{pmatrix}\cos(\theta/2) & -sin(\theta/2)\\sin(\theta/2) & \cos(\theta/2)\end{pmatrix}$$

$$R_z = e^{-iZ\theta/2} = \begin{pmatrix}\mathrm{e}^{-i\theta/2} & 0\\0 & \mathrm{e}^{i\theta/2}\end{pmatrix}$$

In general, $\cos(\theta/2)\mathbb{1} + i\sin(\theta/2)\,(X.n_X + Y.n_Y + Z.n_Z)$ rotates the state by an angle θ about **n**.

Two other gates, **Hadamard** and **Phase**, are used frequently in quantum algorithms. The first is denoted by H and it turns the standard 0/1 basis to the Hadamard +/− basis. Based on this, we can see that its operator and matrix are:

$$H = |+\rangle\langle 0| + |-\rangle\langle 1| \Rightarrow \; H = 1/\sqrt{2}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}$$

The phase gate applies a phase to either or both components of a qubit. We can always pull out and drop a global phase and leave only the phase on the second component. Hence:

$$Phase = \begin{pmatrix}1 & 0\\0 & e^{i\theta}\end{pmatrix}$$

If θ = π/2 (the phase is *i*) then this gate is often denoted by **S** or **P**, and if θ = π/4 then this gate is often denoted by **T**.

Once the gate's matrix is obtained, we need to come up with a physical realization of it so it can be inserted in a quantum computer. To that end, the notion of **universality** is handy. In classical digital logic, it was found that *any* gate can be built using only gates chosen from a universal set, e.g. the {AND, OR, NOT} set. The same was found for quantum gates: *any single-qubit gate can be synthesized from gates chosen from a **universal set***. Hence, once the gates in a universal set are realized, all other gates can be synthesized. There are *many* universal sets. For example, { R_y, R_z } is universal in that given any gate G, we can always find angles a, b, c such that, up to a global phase, G = R_z(a) @ R_y(b) @ R_z(c). The set { H, Phase } is also universal.

- What does the Z matrix do? Express it as the sum of two outer products.
- The Hadamard matrix turns |0> to |+> and |1> to |–>. Create its matrix via outer products.
- Compute X@Y (@ is matrix multiplication) and express it in terms of a Pauli matrix.
- Show that the Hadamard gate takes |x> to [|0> + $(-1)^x$ |1>] / √2, where x is either 0 or 1.
- Why is 1/√2 x [[exp(ipi/4), exp(–ipi/4], [exp(–ipi/4), exp(ipi/4]] known as the square root of the X gate. It takes 0 not to 1 but halfway to 45 degrees …
- Prove that X = HZH.

6. Gates by Code

Rather than doing matrix operations by hand, we can use a programming language to do it for us. Python is a light-weight programming language that has a library called **NumPy** intended for scientific computations, including linear algebra. For example, to build the X gate from the sum of two outer products |0><1| + |1><0|, we start by importing the library module and giving it an abbreviated name np:

```
import numpy as np
```

In this module, vectors and matrices are called **arrays**. So to define the ket |0>, which is a column vector with 1 on top and 0 below, we write:

```
ket0 = np.array([[1],[0]])
```

To define the corresponding bra <0|, which is the adjoint of |0>, we transpose the above ket and complex conjugate the elements to obtain the row vector:

```
bra0 = np.conjugate(ket0.T)
```

Similarly for |1> and <1|:

```
ket1 = np.array([[0],[1]])
bra1 = np.conjugate(ket1.T)
```

Finally, we compute the two outer products (aka Kronecker product) and add them:

```
x = np.kron(ket0, bra1) + np.kron(ket1, bra0)
```

You can insert print statements throughout to see what is being computed, e.g.

```
print("ket0: \n", ket0)
print("x: \n", x)
```

(The "\n" character is to insert a new line between the textual message and the value.)

The final output should be the same as the Pauli X matrix:

```
pauliX = np.array([[0,1], [1,0]])
print(pauliX)
```

Finally, let us test that our matrix does indeed negate the state. Start with:

|state> = (3|0> + 4|1>)/5 = 0.6|0> + 0.8|1>

and transform it via X (the @ operator multiplies matrices):

```
state = np.array([[0.6],[0.8]])
print(x@state)
```

The output should be a column vector with 0.8 on top and 0.6 below.

7. Circuits with Qiskit

Computing outputs programmatically rather than manually makes designing and testing gates a lot faster and less prone to errors. In this section, we move beyond gates and look at circuits made up of several gates intended to implement an algorithm. To that end, we enrich the programming approach with the Qiskit framework. Qiskit was designed specifically for quantum computing, and hence, it knows about gates and states and has ready-made functions, so we don't need to build things from scratch as we did with NumPy. Moreover, Qiskit comes with visualization features that allows us to see the circuit and the Bloch Sphere, and with compilation features that allows us to actually run the circuit either on a simulator or on a real quantum computer backend.

To build a circuit in Qiskit, *five* steps are needed:

1. Indicate the number of qubits needed for the computation. These qubits are collectively known as the **quantum register**, and you create it using a statement like this:
   ```
   q = QuantumRegister(1,'q')
   ```
   The left-hand-side variable q holds this register; the "1" parameter means only one qubit is needed, and the final 'q' is an arbitrary label that we attach to this register, and it appears in the circuit diagram. It can be any label you like.

2. Indicate the number of classical bits needed for the computation. All quantum algorithms end up in some measurement and in qiskit, the outcome of any measurement is stored in a classical bit. These bits are also collectively known as the **classical register**, and you create it using a statement like this:
   ```
   c = ClassicalRegister(1,'c')
   ```
   The left-hand-side variable c holds this register; the "1" parameter means only one qubit is needed, and the final 'c' is an arbitrary label that we attach to this register, and it appears in the circuit diagram. It can be any label you like.

3. Create the quantum circuit and give it a name. Use a statement like this:
   ```
   qc = QuantumCircuit(q,c)
   ```
   The left-hand-side variable qc holds the circuit, and the right-hand-side specifies the qubits it works on and the bits to store the measurement outcomes in.

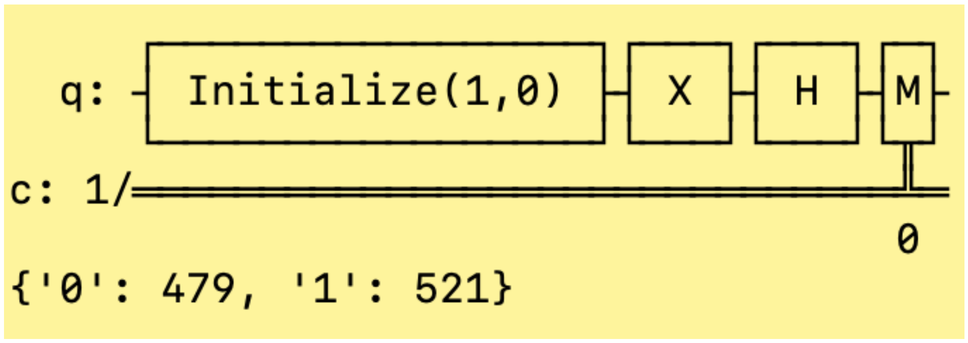
4. Initialize the input, i.e. specify the initial state of the qubits in the quantum register:
   ```
   qc.initialize([1,0], q)
   ```
   This function takes an array to indicate the initial state and the name of the quantum register. In our example, the register holds only 1 qubit so the array has only two elements.

5. Indicates the gates (x, y, z, h, p, …) that make up the circuit, e.g.
   ```
   qc.x(q[0])
   qc.h(q[0])
   ```
   Each statement starts with the circuit variable qc followed by a dot, followed by the gate. Note how Qiskit "knows" about the Hadamard gate h and the Pauli X gate x, so we don't need to build their matrices from scratch. You need to specify the quantum register and which qubit within it you like the gate to operate on. In our case, there is only one register and one qubit in it, so you can write q[0] or just q.

Once these 5 steps are taken, you can draw the circuit and run it and see the outcome of measuring the output. Here is the complete program.

```
import numpy as np
from qiskit import *
backend = Aer.get_backend('qasm_simulator')
q = QuantumRegister(1,'q')
c = ClassicalRegister(1,'c')
qc = QuantumCircuit(q,c)
qc.initialize([1,0], q)
qc.x(q[0])
qc.h(q[0])
qc.measure(q,c)
print(qc.draw(output='text'))
job = execute(qc, backend, shots=1000)
result = job.result()
counts = result.get_counts()
print(counts)
```

Running this program yields the output below.



```
   q: ┤ Initialize(1,0) ├┤ X ├┤ H ├┤M├
                                   └╥┘
c: 1/═══════════════════════════════╩═
                                    0
{'0': 479, '1': 521}
```

The measurement yielded stats of about 50%-50% for 0 and 1, so the final state before taking the measurement must have been an equal superposition: 1/√2(|0> ∓ |1>). We can't tell the relative phase between the two basis vectors, but we know their two amplitudes are equal in magnitude. Verify this manually (multiply the input states by the X and then the H matrix).

Finally, we show three Qiskit features that I find very instructive: the ability to track the state vector ket along the circuit; the ability to see the state vector on the Bloch sphere; and the ability to use a custom gate:

- ***Capturing the state ket***

  To do this, we need is to save the state vector after each stage and then display it at the end. The needed additions to the above program are shown in purple below:

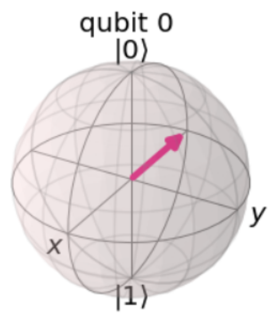
```
qc.initialize([1,0], q)
qc.save_statevector(label='v0')
qc.x(q[0])
qc.save_statevector(label='v1')
qc.h(q[0])
qc.save_statevector(label='v2')
qc.measure(q,c)

for i in range(0,3):
    print("At barrier "+str(i)+":")
    print(np.asarray(result.data(0)['v'+str(i)]))
```

Running this program adds the following to the above output:

```
At barrier 0:
[1.+0.j 0.+0.j]
At barrier 1:
[0.+0.j 1.+0.j]
At barrier 2:
[ 0.70710678-8.65956056e-17j -0.70710678+8.65956056e-17j]
```

- ***Bloch visualization of the state***
  To do this, we add two statements to your code, an import and a plot:

```
from qiskit.visualization import plot_bloch_multivector
...
state = result.data(0)['v2']
plot_bloch_multivector(state)
```



- ***Using custom gates***
  To add a custom gate (an arbitrary unitary matrix), we can either decompose it into a product of built-in, universal gates, or enter its matrix directly. For the latter, we add two statements: an import and an append, plus the matrix:

```
from qiskit.quantum_info.operators import Operator
...
custom = np.matrix( … )
…
qc.append(Operator(custom), [0]);
```

Remarks

1. **Postulate #1 of QM (Statics)**
   The state is a vector in a complex Hilbert space.
2. **Postulate #2 of QM (Dynamics)**
   The system evolves through a unitary transformation
3. **Postulate #3 of QM (Measurement)**
   Each measuring device defines an orthonormal basis. Upon measurement, the outcome and the post-measurement state will be one of those basis vectors, with probability equal to the modulus squared of the state projection on that basis vector.
4. **Postulate #4 of QM (Composite)**
   The state space of a composite system is the tensor product of the component spaces.
5. It may sound negative that unknown states cannot be cloned (because no cloning precludes being able to gather statistics), but we can turn this to our advantage: digital currency cannot be copied. Isn't the inability to forge money the key property of any currency?
6. The programs we wrote for gates and circuits are not really programs in the typical sense; they don't compute. They merely *describe* behaviour, much like Verilog programs describe hardware. In particular, they cannot have selection (*if* statements) as this requires testing a condition, which amounts to measuring in the middle of computing, which would collapse the state—we only measure at the end of a quantum algorithm.
7. Note that a circuit with a number of serial gates is just a model, or a visualization, of a number of transformations acting, one after the other, on the input. In terms of matrices, this model represents the product of the matrices of the gates. Viewed this way, the circuit seeks to perform a "big" unitary transformation, and it does that by expressing its matrix as a product of matrices (gates) belonging to some universal set.