Module 14 Final Exam: Quantum Computation

JHU EP 606.206 - Introduction to Programming Using Python

Introduction

The dynamics of quantum bits, called qubits, are governed by how particles interact at a subatomic level. In some ways, qubits can behave like classical bits and can be treated as a 0 or a 1; these are called the "basis states" of a qubit. Unlike classical bits, however, qubits can exist in a superposition of states that can be expressed as a linear combination of the aforementioned basis states. In this Final Exam Assignment you will use a basic linear algebraic formulation of qubits to understand more about how qubits behave and how they can be used to perform computation.

Skills: conditional statements, control structures, file I/O, functions, classes (OOP), exceptions, NumPy arrays

Qubits as Vectors

It is perhaps simplest to think of a qubit as nothing more than a 2-dimension vector that lies within the complex vector plane. That is, a qubit in either of the basis states can be represented as a 2x1 vector.

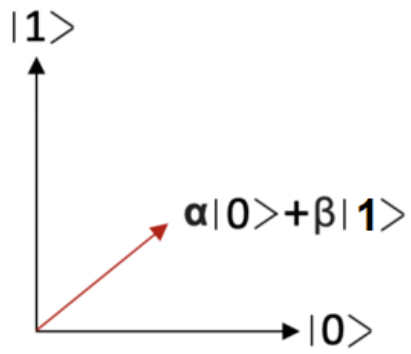
Here's an example of a qubit in the "0" basis state. The qubit, which appears on the left, is called a "ket". As you can see, a qubit in this state (which can conceptually be thought of as a classical bit with a value of 0), is represented by the 2-dimension column vector shown (with an x-component of 1 and a y-component of 0):

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

And here's an example of a qubit in the "1" basis state (which can conceptually be thought of as a classical bit with a value of 1). It has an x-component of 0 and a y-component of 1:

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

If we plot these on a 2-D plane, we get the result below where the length of each of the 2 basis vectors is 1:



So far our qubits are nothing more than 2-D vectors of unit length and don't provide any advantages over classical bits in the 0 or 1 state. But, notice the red line in the graph above. This red line represents a ***superposition*** of the two basis states and this is one of the advantage quantum bits have over classical bits. Instead of the qubit being only a 0 or 1, the qubit is in a superposition of both 0 and 1 at the same time! We mentioned in the Introduction that in quantum physics we can have qubits in their basis states (the "0" and "1" basis states above) or a superposition of the basis states. This superposition is just a linear combination of the two basis states. So, it might help to think of the superposition to mean the qubit is currently alpha amount in the "0" basis state and beta amount in the "1" basis state. The values alpha and beta, though, are probability amplitudes, so what we're really saying here is that, if we were to measure the qubit that is in a superposition, there is an alpha^2 chance we measure a "0" and a beta^2 chance we measure a "1". A very important thing to keep in mind is that the sum of the squares alpha and beta must equal 1. Here's an example:

$$\alpha|0\rangle + \beta|1\rangle, \alpha = 0.8, \beta = 0.6$$
$$\alpha^2 = 0.64, \beta^2 = 0.36$$
$$\alpha^2 + \beta^2 = 0.64 + 0.36 = 1$$

Therefore, there is a 64% chance of measuring a "0" and a 36% chance of measuring a "1". This makes sense because qubits are probabilistic in nature. If we had a qubit in this state 100 times and performed 100 measurements, we'd expect to receive 64 0's and 36 1's from our measurements. For completeness, we can write any arbitrary superposition as the 2-D column vector containing alpha and beta by plugging in the vector representation of |0> and |1>:

$$\alpha|0\rangle + \beta|1\rangle = \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Matrices as Logic Gates (Operators)

Pauli X Gate

To use qubits for computation, we must develop a set of ***operators*** (logic gates) that will act upon a (in our case, single) qubit in a similar way to how logic gates act on classical bits. For example, we've learned about the not operator in Python which negates a Boolean value (it "flips" the value so True becomes False and False becomes True). The quantum analog to the classical NOT gates is the 2x2 matrix called the Pauli X-gate or the "quantum NOT" operator:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Similar to how flipping a classical bit works when applying a NOT gate to a 0, we can apply the X-gate to a qubit (by multiplying the operator (matrix) by the qubit (vector)) in the 0 basis state and we will get a qubit in the 1 basis state:

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

Similarly, applying an X-gate to a qubit in the 1 basis state will give us a qubit in the 0 basis state:

$$X|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

Hadamard Gate

The Hadamard operator (gate) can be applied to a single qubit to place it into an equally-weighted superposition of the 0 and 1 basis states. This notion does not apply to classical bits which take on only discrete values: 0 or 1. A qubit in a superposition is in both the |0> and the |1> state at the same time! A Hadamard gate can be represented with the following 2x2 matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Here is the Hadamard operator (gate) applied to a qubit in the |0> state. We say the resulting superposition is "equally weighted" because, after squaring the amplitudes, there is a ½ chance measuring the qubit results in a |0> and a ½ chance of measuring a |1>:

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Final Exam Programming Assignment

1. Create a file called **main.py** that runs a function called **main()**. Your **main()** function should:
    a. Read in a file called **qubits.txt** that contains the state of the qubits and sequences of operators that will be applied to the qubit. Each line in the input file will contain a floating point value for alpha, a floating point value for beta, and a number of operators that will be applied to the qubit (only the X and H gates are valid operators)
        i. Ex: `0.8 0.6 H X` (alpha = 0.8, beta = 0.6, apply 1 H gate then 1 X gate)
    b. Contains a loop that processes each qubit in the input file (applies the list of operators) using the **Qubit** and **SingleQubitOperator** classes.
    c. Calls the **experiment** method (Section 2.c below) in the **Qubit** class times for each qubit in the input file and outputs the results as a percentage of 0's and 1's.
    d. Prints the initial state of the qubit at the beginning of the process and the final state of the qubit once all operators have been applied.
2. Next, create a ***Qubit*** class inside of a module named ***qubit.py***. Your class should:
    a. Keep track of the state of the qubit which will be the 0 basis state |0>, the 1 basis state|1>, or an arbitrarily weighted superposition of the two basis states. The constructor should also call **validate_amplitudes** (defined in 'c' below) to ensure the coefficients alpha and beta are valid (the sum of their squares = 1).
    b. Contains a method named ***prob_amplitudes*** that returns a tuple containing the probability of receiving a |0> and the probability of receiving a |1> from a measurement. That is, the square of alpha and the square of beta.
    c. Contains a method called ***validate_amplitudes*** that raises an ***InvalidProbabilityAmplitude*** exception that prints "Invalid probability amplitude(s)." if illegal values for alpha and/or beta (the sum of their squares is not equal to one.
    d. Contains a method called ***experiment*** that uses a properly weighted binomial distribution to demonstrate that a measurement of a qubit yields a 0 (|0>) with probability alpha^2 and yields a 1 (|1>) with probability beta^2. ***experiment*** should run 100 experiments each consisting of a single trial.
    e. Contains a ***__str__*** method that provides a string representation of a qubit object:
        i. "X|0> + Y|1>", where X and Y are the coefficients of the basis vectors
3. Next, build a ***SingleQubitOperator*** class inside of a module named ***qoperators.py*** that:
    a. Will be used to derive 2 specific single-qubit operator subclasses
    b. Contains a constructor that:
        i. accepts an operator matrix and creates an instance variable from it
        ii. raises an **InvalidOperator** exception that prints "Invalid operator." if an operator other than X or H are provided in the input file
4. Finally, build 2 classes: ***PauliX*** in a module named ***paulix.py*** and ***Hadamard*** in a module named ***hadamard.py***, both of which are child classes of ***SingleQubitOperator***. Each class should:
    a. Be a child class of the **SingleQubitOperator** class created in Step #2
    b. Call the parent class' constructor and pass in the appropriate matrix (operator)
    c. Implement a method called **operate** that accepts a qubit, applies the class' operator to the qubit, and returns a new Qubit object containing the updated state

Sample Output

```
Initial state: 1.0|0> + 0.0|1>
Final state: 1.0|0> + 0.0|1>
Percentage of 0's measured: 1.0
Percentage of 1's measured: 0.0

Initial state: 0.0|0> + 1.0|1>
Invalid operator.

Initial state: 1.0|0> + 0.0|1>
Final state: 0.7071067811865475|0> + 0.7071067811865475|1>
Percentage of 0's measured: 0.51
Percentage of 1's measured: 0.49

Initial state: 0.0|0> + 1.0|1>
Final state: 0.0|0> + -0.9999999999999998|1>
Percentage of 0's measured: 0.0
Percentage of 1's measured: 1.0

Initial state: 0.6|0> + 0.8|1>
Final state: 0.7999999999999998|0> + 0.5999999999999999|1>
Percentage of 0's measured: 0.7
Percentage of 1's measured: 0.3

Initial state: 0.7071067811865475|0> + 0.7071067811865475|1>
Final state: 0.0|0> + 0.9999999999999998|1>
Percentage of 0's measured: 0.0
Percentage of 1's measured: 1.0

Invalid probability amplitude(s).
```

Deliverables

readme.txt

So-called "read me" files are a common way for developers to leave high-level notes about their applications. Here's an example of a README file for the Apache Spark project. They usually contain details about required software versions, installation instructions, contact information, etc. For our purposes, your readme.txt file will be a way for you to describe the approach you took to complete the assignment so that, in the event you may not quite get your solution working correctly, we can still award credit based on what you were trying to do. Think of it as the verbalization of what your code does (or is supposed to do). Your readme.txt file should contain the following:

1. **Name**: Your name and JHED ID
2. **Module Info**: The Module name/number along with the title of the assignment and its due date
3. **Approach**: a detailed description of the approach you implemented to solving the assignment. Be as specific as possible. If you are sorting a list of 2D points in a plane, describe the class you used to represent a point, the data structures you used to store them, and the algorithm you used to sort them, for example. The more descriptive you are, the more credit we can award in the event your solution doesn't fully work.
4. **Known Bugs**: describe the areas, if any, where your code has any known bugs. If you're asked to write a function to do a computation but you know your function returns an incorrect result, this should be noted here. Please also state how you would go about fixing the bug. If your code produces results correctly you do not have to include this section.

Please submit your **main.py**, **qubit.py**, **operator.py, paulix.py, and hadamard.py** source code files along with a PDF file containing screenshots of your output in a PDF called JHEDID_mod14.pdf (ex: jkovba1_mod14.pdf). Please do not ZIP your files together.

Recap:

1. readme.txt
2. **main.py**
3. **qubit.py**
4. **qoperators.py**
5. **paulix.py**
6. **hadamard.py**
7. A screenshot(s) of your outputs from **main.py**
    a. Note: the exact values resulting from your experiments may differ slightly. This is because qubits are non-deterministic. If, for example, there is a 60% chance of measuring a 0 and a 40% chance of measuring a 1, you may get 57 0's and 43 1's or 61 0's and 39 1's. This is normal behavior and is the very essence of quantum computing.
    b. To help guide you towards a solution:
        i. Our **main.py** is ~55 lines
        ii. Our **qubit.py** is ~40 lines of code
        iii. Our q**operators.py** is ~35 lines of code

Please let us know if you have any questions via Teams or email!