Quantum Circuit Simulator

Submit to Autograder by 11:55 pm ~~Fri 2/9~~ EDIT: Tue 2/13

[Direct Autograder Link](https://autograder.io/web/project/2432)

[Starter Code](https://drive.google.com/drive/u/1/folders/1BsYK1Rtx3cdNBbw1Lr6uHDhnPH2u6lzu)

# Learning Objectives

After successfully completing this project, students will have a thorough understanding of how quantum states and circuits are represented, and how computation is done at an abstract level.

# Introduction

In this project, you will write a simulator for a simple quantum computer, the **Little Quantum Computer 3000 (LQ3K)**. This simulator supports constructing a quantum circuit using a small set of gates, calculating the unitary representation of that circuit, and simulating the result of running the circuit with a given state vector by either returning the resulting state vector (deterministic) or performing a measurement on all bits and returning the measured value (probabilistic). This is effectively a (very) scaled down version of the Qiskit library, which you will use in the remainder of the course.

# Project Details

The simulator is implemented as a Python class. We have provided method stubs for you to complete. You are welcome to add additional methods or member variables as you wish, but you must not change the method declarations provided.

These are the following gates supported by the LQ3K simulator:

* X
* Y
* Z
* T
* T dag (complex conjugate of T)
* Hadamard (H)
* SWAP
* Controlled Not (CX)
* Toffoli (CCX)

The LQ3K does **NOT** support arbitrary measurements. The only way to measure a circuit is to run "simulate\_run()" on an object after composing the circuit. This will automatically measure every qubit in the circuit at the end of computation and return the results.

When storing a state vector as an array, the simulator follows the same ordering conventions as presented in lecture, i.e. index N holds the probability amplitude of |N>. For example, in a 2-qubit system the state |q1q0> = a|00> + b|01> + c|10> + d|11> is represented by the array [a, b, c, d].

Additionally, multi-bit unitary matrices are ordered using the larger index bit as more significant. E.g. for a circuit with a controlled-NOT gate with qubit 0 controlling qubit 1:

qc = LQ3K(2)

qc.cx(0, 1)

qc.get\_unitary()

Column 0 corresponds to the output when the input is {qubit1,qubit0} = {0,0}, column 1 corresponds to input {0,1}, column 2 corresponds to input {1,0}, and column 3 corresponds to input {1,1}, e.g this gives the unitary matrix:

| 1 | 0 | 0 | 0 |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |

# Testing

You must provide a set of test functions written in tests\_p1~~p1\_tests~~.py to the autograder. That provides unit tests for each of the methods provided. You must use the [Unittest model](https://docs.python.org/3/library/unittest.html) discussed in lab. Your tests will be graded on whether they cause assertion failures when run on buggy solutions, but do not cause assertion failures on correct implementations.

# Error Handling

Your implementations for simulate\_run() and evolve() must check that the provided state vector is valid, i.e. that the norm is 1. If any invalid input is provided, your code must raise an "LQ3K.InvalidStateVector" exception.

# Restrictions

Complete all function methods in p1.py and all method tests in tests\_p1~~p1\_tests~~.py. The only packages you may import are numpy, math, random, and unittest.

# Submission and Grading

Submit p1.py and tests\_p1~~p1\_tests~~.py to the autograder using the direct link at the top of this page. Your final grade is out of ~~35~~33 points.

We will grade your code on functional correctness. As a reminder, you may not share any part of your code with others in a way that can be copied. You are encouraged, however, to discuss the project with each other and help each other with problems you may encounter. You will get feedback on your total score, but you will not have access to what the private test cases are checking for.

Efficiency is not graded, but your code must complete in a reasonable amount of time. Note that for general quantum circuits, simulation takes an exponential amount of time. We will only grade your code on circuits containing up to 6 qubits.

Due to rounding, your unitary matrices and state vector calculations may slightly deviate from the correct answer. We will check that every value calculated is within .001.

The output of "simulate\_run()" is generally non-deterministic but should follow a precise distribution for a given circuit. We will grade this function by running it several times and verifying that it matches the theoretical distribution within 10%.

# Appendix

You are free to implement this simulator however you see fit. However, we recommend modeling each circuit as a unitary matrix using numpy.array objects. Each addition of a gate should update this matrix accordingly, and passing a given state vector through the circuit can be implemented by multiplying the matrix representation with the state vector. See the [course reference](https://colab.research.google.com/drive/1d2aoZ8dtNXiBz1Crt-Oc8WLzTryhSm2I?authuser=1#scrollTo=WjZmcRWCJt4B) for a brief overview on how to deal with matrices and vectors in Python using NumPy. Below is a brief description on how to approach each section of the project using this strategy.

## Comparing Values

Anytime you perform floating point calculations, there are likely to be small rounding errors. So for example, when calculating the norm of a vector, although it should be 1, you may find that the value is 0.99999987, or something like that. This could cause problems when writing your own tests, or writing functionality for your LQ3K class (e.g. ). We recommend using the [allclose](https://numpy.org/doc/stable/reference/generated/numpy.allclose.html) method to deal with this.

## Single qubit gates

Applying a single qubit gate to a multi-qubit circuit is equivalent to applying the Identity operation to every remaining qubit. The overall operation can be described as an appropriate Kronecker product between several instances of the Identity matrix and the 2x2 matrix representing the single qubit gate.

## SWAP

Swap is easier to implement if the qubits being swapped are adjacent. In that case, the full operation can be described by a Kronecker product between Identity matrices and the 4x4 matrix representing the SWAP operation discussed in lecture.

Swapping non-adjacent qubits can be implemented through multiple swaps of adjacent qubits. E.g. swapping [0,2] can be done by swapping [0,1], then [1,2], then [0,1] again.

|  | = |  |
| --- | --- | --- |

## CNOT

As with SWAP, CNOT is easiest when the bits are adjacent (allowing for a simple Kronecker product between Identity matrices and the 4x4 CNOT matrix described in lecture). For non-adjacent qubits, SWAP can be used to move one of the qubits to be adjacent, and then swapped back after the CNOT is applied.

|  | = |  |
| --- | --- | --- |

## Applying multiple gates

Applying multiple gates in sequence is equivalent to performing a matrix multiplication between the matrices representing those gates.

## Simulating Circuit

When simulating measurement of your circuit, you must randomly choose one of the possible outcomes, waited by a set of calculated probabilities. You may find the [random.choices](https://www.w3schools.com/python/ref_random_choices.asp) function helpful here. If outcomes is a list holding the possible outcomes, and probabilities is a list such that probabilities[i] is the probability of measuring outcomes[i], then the following statement will select one of the outcomes with the appropriate probability (the function returns a list, which is why you must access [0] at the end to read the actual outcome):

random.choices(outcomes, probabilities)[0]