DD2367 HT24
Final Project Report
Rose Maresh, Naoto Uemura

Development of a Multi-Qubit Quantum Computer Simulator

Problems to Solve

We are developing a quantum computer simulator in order to understand the behavior of quantum computers. While making a quantum computer simulator using classical computing defeats the purpose of using quantum computing to begin with due to memory limitations, it plays a vital role in quantum research for when quantum computers become available at a large scale in the future.

Methods

In order to achieve the simulation of quantum computers, a few systems must be in place. We will represent the quantum state with a vector of complex numbers, and quantum gates as unitary matrices which will be applied to the aforementioned states. When the gates are applied to entire multi-qubit registers instead of single-qubit registers, then we will expand them using the Kronecker product. For details, refer to Appendix A-I. We will represent circuits as Python functions, which can be passed as a parameter into a special function we will define to facilitate the simulation of multiple “runs” of a circuit and the collection of data from these runs.

Implementation

Quantum State

In order to realize the methodology above, the following implementations were used. We created a quantum_state class that includes a field for the state (which is represented as a numpy array of complex numbers), as well as the number of qubits. For testing purposes,

we also included the option to name a quantum state. The reader may refer to Appendix B-I for the __init__() method of the quantum_state class.

Quantum Gates and Instructions

For each gate, we created the corresponding matrix to be applied to the vectors representing the quantum states. Depending on the number of qubits in the state we wish to apply the gate to, it is sometimes necessary to expand the matrix, which we implemented using the Kronecker product (Appendix A-I). For details of this implementation, see Appendix B-II.

To apply these gates, we defined methods to call in the quantum_state class. The application of a gate transforms a quantum state via a dot product, such as this application of the NOT gate:

```
def x(self):
    self.state = self.state.dot(not_matrix(self.qubits))
```

However, there are a few more considerations when applying gates to multi-qubit states. When a gate is applied to just one qubit in a multi-qubit state, it is in practice applied not to the whole state but instead to qubit pairs. In the case of controlled gates, the qubit pairs to which the gates are applied are specified based on the control qubit(s). We implemented this by generating the qubit pairs first based on the target qubit, then removing from our set those pairs which are not specified by the control qubit(s). For more details on the implementation of operating on multi-qubit states, see Appendix B-III and Appendix B-IV.

We implemented the support for many crucial gates, including HAD, NOT, CNOT, CCNOT, PHASE, CPHASE, ROTX, ROTY, and most importantly, the READ operation. The READ instruction is fundamental to a quantum computer simulator because it needs to bridge the gap between the hardware differences of classical and quantum computers. On a quantum computer, READ collapses a quantum state out of superposition because of the properties of quantum particles. In our representation, we needed to simulate the randomness of this collapse as well as ensure the state of the register was consistent with this collapse after the operation. We implemented READ with three cases, which can be understood by looking at the function header:

```
def read(self, index1=None, index2=None):
```

There are two optional parameters in this function, index1 and index2. If there is no value passed in for either of these, then the function reads the entire register. If there is one parameter passed in to a function call, then the function reads the qubit at the index that is passed in. If there are two parameters, then they are treated as the bounds for a range of indices to be read from the register.

For each of these cases, we needed to ensure that our implementations satisfied the desired properties that we described above. We simulated the randomness of a quantum mechanical system by using Python's random() function. While this is not truly random, it represents the behavior of a quantum particle. Our second desired property for the READ operation was that the final state of the register would be consistent with the collapse that would occur as an inherent side effect of reading a quantum register. For the case in which we read the entire register, we implemented this by setting the value at the index corresponding to the read value to 1 and the rest of the values to 0. We had to approach this collapse a little differently in the case that only one qubit from a multiqubit register was read. This is because the other qubits in the register may still be in superposition, even if one of the qubits is collapsed, requiring us to renormalize the remaining probabilities instead of simply setting the read index to 1. The implementation of these two cases can be seen in Appendix B-V and Appendix B-VI.

Because we renormalize the probabilities each time we read a single qubit, if we wish to read a range of indices from the register, we can read each qubit within that range individually and combine them into the final read value. We iterated over the range, concatenating each read value into a binary string, then returned the integer representation of that string. The details for this procedure are demonstrated in Appendix B-VII.

Circuits and Simulation

Furthermore, to enable simulatability of circuits over multiple runs and extract data, we modeled circuits as python functions that return measured register values, similar to a "classical register". As an example, here is the implementation of a quantum random number generator:

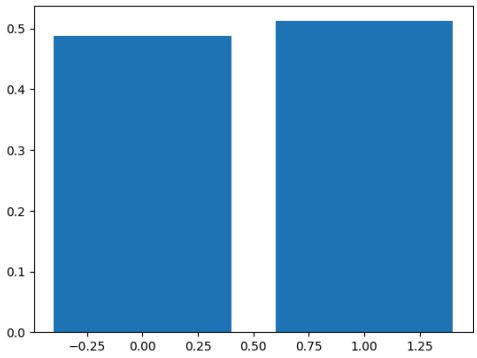
```
def rng(reg=None, **kwargs):
```

```
  if reg == None :
    reg = kwargs['registers'][0]
  reg.h()
  return reg.read()
```

Notable here is that registers are passed into the circuit as either a named parameter or as part of a variable-length parameter list. We made this implementation decision to make our circuits compatible with our simulate() function. We created this function to facilitate the simulation of multiple runs of the circuit, and obtain a distribution of the results in either frequency or decimal values. As described in Appendix B-VIII, the implementation of this function allows for variable number and sizes of registers and can collect and compile output data of variable size.

When the user wishes to simulate a circuit, all they need to do is initialize the quantum registers and pass the name of the circuit function into simulate() along with the registers, number of qubits measured, and desired properties of the simulation. Then, they may access the results of the simulation as the return value of the simulate() function. These results may be displayed or visualized however the user wishes. Here is an example of how the random number generator could be simulated, as well as the results displayed via Python's matplotlib pyplot:

```
s0 = quantum_state(1, 0, name="s0")
results = simulate(rng, 1, 500, True, s0)
plt.bar(results.keys(), results.values())
plt.show()
```

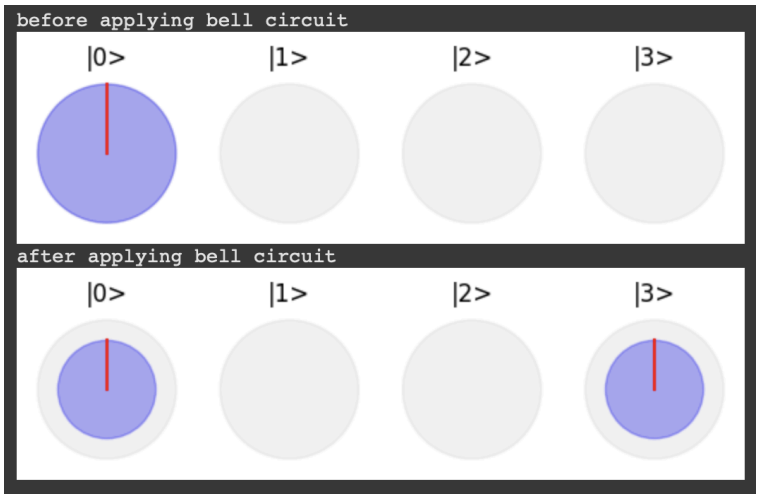


This functionality, combined with a function for visualizing quantum states in circle notation, allowed us to test and demonstrate several quantum circuits, which we will describe below. The function for visualizing the circuits was modified from the viz2() function provided in Module I of this course.

Result

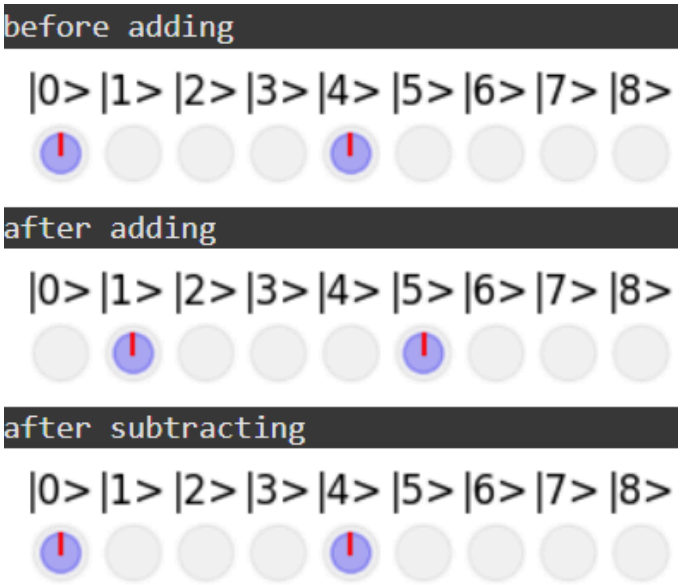
In order to test and demonstrate the functionality of our quantum simulator, in addition to the quantum random number generator described above, we have simulated several fundamental quantum circuits.

The first of these circuits is the Bell circuit, which generates an entangled quantum state. The details of the operation can be found in Appendix A-II, and the implementation of the circuit can be found in Appendix C-I. Here are the results of this simulation, visualized in circle notation:



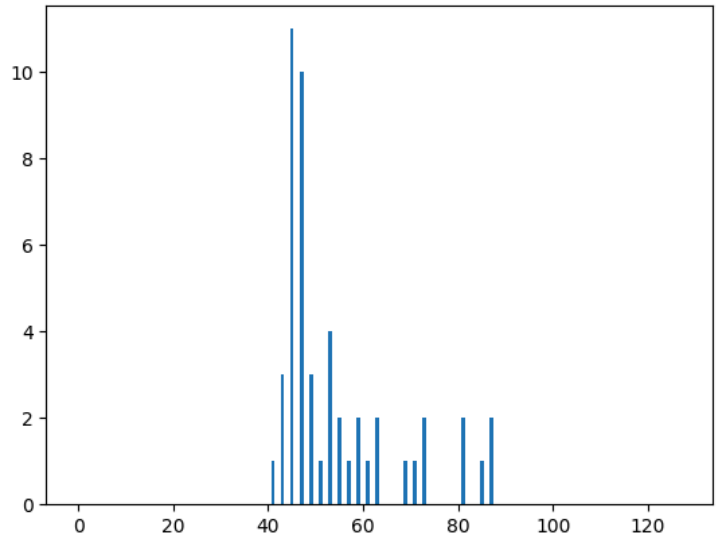
Entanglement is a fundamental property of quantum mechanics. The Bell circuit demonstrates our simulator's support for combining quantum states, as well as the functionality of h() and cnot(), two crucial gates.

Next, we simulated quantum arithmetic, which includes the important increment and decrement primitives, as demonstrated in the visualization below:



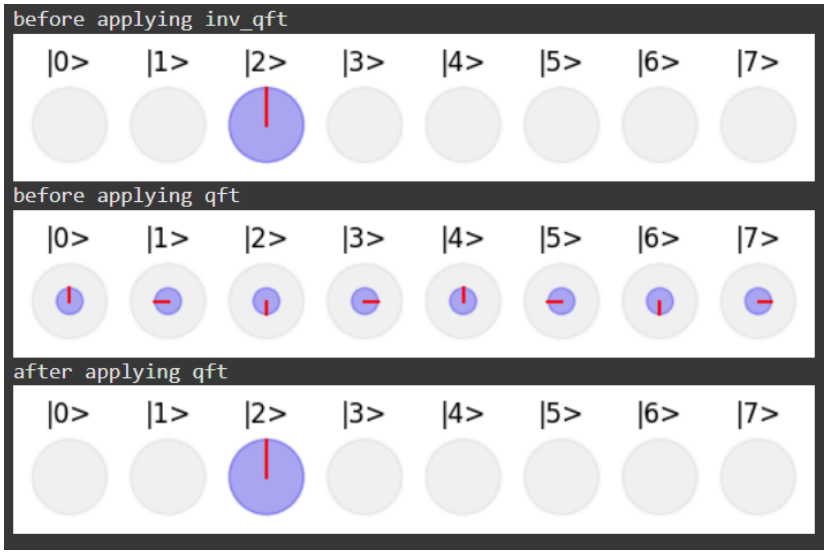
The implementation details of this circuit are included in Appendix C-II. This circuit is important because it demonstrates the ability to perform arithmetic operations on quantum states in superposition, with the fundamental limitation of the non-cloning theorem. Quantum arithmetic also demonstrates our simulator's ability to handle gates with multiple control qubits, as ccnot() is an important building block of this circuit.

Next, we utilized the quantum arithmetic primitives to implement a Discrete Quantum Walk. The DQW is similar to a classical random walk but is affected by quantum properties such as superposition and interference. From the results, the effects of these properties are clear:



Unlike a classical random walk, the distribution of results from many simulations of the DQW is skewed. This is due to interference in the repeated application of the Hadamard gate to the coin qubit. There are also some positions in the walk that are never measured, which is due to interference as well. The implementation of DQW is described in more detail in Appendix C-III. This test also demonstrates our simulator's ability to collect data over many runs of a circuit.

Finally, we implemented the Quantum Fourier Transform, which extracts encoded information from the phases of an input register. To prepare a state with periodically varying frequencies, we first apply the inverse QFT to a register with a value of the desired frequency. Then, we can apply the QFT to extract the frequency from the prepared register, as shown:



The implementation of both QFT and invQFT are described in Appendix C-IV and C-V, and demonstrate the simulator's ability to manage relative phases within a register, as well as the important swap primitive.

Critical Discussion

The development of multi-qubit quantum computer simulator provided us with valuable insights into the challenges as well as the potential of quantum computers. Throughout the project, we built a framework that successfully simulates quantum computing concepts we learned in class, such as bell states, Quantum Fourier Transform (QFT), and discrete quantum walks.

One of the primary limitations with simulation of quantum computers in classical computers is the performance limitation with increased qubit count. Implementing the quantum_state class ourselves, it was obvious from the line of code below

```
self.state = np.zeros(2**size, dtype=np.complex128)
```

that each additional qubit doubles our state space, requiring more memory as well as processing time when computations are done on it.

Contributions:

Rose: States/demo implementation, presentation/report write-up

Naoto: Gates implementation/test suites, presentation/report write-up.

Expected Final Course Grade: **A**

Appendix A

A-I: Kronecker Product

Tensor product over two matrices, crucial in quantum computing and is considered a fundamental building operation. It is often used to combine two or more qubits and gates.
Official definition: $(A \otimes B)[j, k] = A[j/n, k'm] \times B[j\%n, k\%m]$

A-II: Bell States

The entangled state created by a CNOT operation where the control bit is in superposition. It creates an interdependence between qubits, where if we read out the two qubits, they will read out both 0s or both 1s with equal probability.

A-III: Quantum Fourier Transform (QFT)

A quantum circuit technique to reveal some frequency associated with phases in the qubits. It is most notably used in Shor's algorithm which finds the prime factors of an integer.

Appendix B

B-I: Implementation of the quantum_state class

```
def __init__(self, size=1, in_state=0, name=""):
    self.qubits = size
    self.state = np.zeros(2**size, dtype=np.complex128)
    self.state[in_state] = 1
    self.name = name
```

The `__init__` method demonstrates the fields that we included in our quantum_state class. We included the number of qubits, the state, and a name field. The name field is optional and is used for testing purposes. The state is represented by a numpy array of complex numbers. The value at each index is the square root of the probability of finding the register in the state with that index's value. Therefore, the length of the state array is 2^n, where n is the number of qubits in the register. Therefore, the number of qubits in the register can be determined from the length of the state array, but is included as a separate field for user convenience.

B-II: Expansion of a matrix using Kronecker product

```
def hadamard_matrix(q=1):
  sq = 1/math.sqrt(2)
  x = np.array([[sq, sq],[sq, -sq]], dtype=np.float64)
  y = x
  for a in range (q-1):
    y = np.kron(x, y)
  return y
```

This code snippet demonstrates how we expand gate matrices with the Kronecker product. In order to apply a matrix transformation to a vector (in this case, the quantum state vector, represented by a Numpy array), the matrix width must be consistent with the vector height. Since gate matrices are always square, the matrix dimensions must match the length of the quantum state array. The Kronecker product allows us to apply gate transformations to a quantum state of any size by adapting the matrix to the size of the state. For simplicity, we defined our matrices separately from the gate operations themselves. The function that applies the gate to the quantum state can call the matrix function to access it. This allows us to consistently implement gate operations, whether or not they are controlled, and regardless of if they require additional manipulation beyond simply taking the dot product of the quantum state array and the gate matrix.

B-III: Operating on one qubit in a multi-qubit state

```
def x(self, index):
  k = 0
  for i in range(int((2**self.qubits)/(2**(index+1)))):
    for j in range(2**index):
      vec = np.zeros(2, dtype=np.complex128)
      vec[0] = self.state[k+j]
      vec[1] = self.state[k + j+2**index]
      vh = vec.dot(not_matrix(1))
      self.state[k+j] = vh[0]
      self.state[k+j + 2**index] = vh[1]
    k += 2**index + j + 1
```

This code demonstrates how we are able to perform gate operations on a single qubit which is part of a multi-qubit register. To do this, we must apply the matrix transformation to qubit pairs. The number of pairs depends on the size of the multi-qubit register, and the formation of the pairs depends on which qubit is operated on. The former is accounted for with the outer loop, and the latter with the inner loop and the last line in the function. The first and second states in the pair are considered as if they make up an isolated quantum state and are operated on with the gate matrix. Then, the original quantum state is updated with the transformed values.

B-IV: Controlled operations

```
def ccnot(self, index, control):
  k = 0
  pairs = []
  for i in range(int((2**self.qubits)/(2**(index+1)))):
    for j in range(2**index):
      pairs.append((k+j, k+j+2**index))
    k += 2**index + 1 + j
  for l in control:
    for m in range (len(pairs)-1, -1, -1):
      p = pairs[m]
      b0 = bin(p[0])[2:].zfill(4)
      b1 = bin(p[1])[2:].zfill(4)
      if b0[len(b0)-1-l] == '0' or b0[len(b1)-1-l] == '0' :
        pairs.pop(m)
  for n in pairs:
    v = np.zeros(2, dtype=np.complex128)
    v[0] = self.state[n[0]]
    v[1] = self.state[n[1]]
    v = v.dot(not_matrix())
    self.state[n[0]] = v[0]
    self.state[n[1]] = v[1]
```

When we apply controlled operations to multi-qubit registers, it is similar to applying an operation to one qubit from a multi-qubit register (Appendix B-III). The difference is that not every qubit pair formed will be operated on. The control qubit designates which of the pairs should be transformed. Only pairs for which both members are at an index where the control qubit is 1 should be transformed. The way we managed this is by creating our qubit pairs as we did before, but instead of operating on them right away, we removed from our list pairs which did not meet the criteria. Then we operated on the remaining pairs and updated the original state accordingly.

B-V: Applying READ to an entire register

```
if index1==None and index2 == None:
      prob = square_all(self.state)
      rp = random.random()
      n = 0
      while rp > 0:
        rp = rp - prob[n]
        n = n + 1
      self.state = np.zeros(len(self.state), dtype=np.complex128)
      self.state[n-1] = 1
      return n-1
```

READ is one of the most important quantum operations. For our implementation, we generate a probability array that holds at each index the probability of a quantum register being read at that state. We generate this by taking the square of the value at each index of the quantum state array. Then, to simulate the randomness of a real quantum computer, we randomly generate a number between 0 and 1. Then, we find where in the probability register that number is found, by taking the cumulative sum of each consecutive index until we reach a sum greater than or equal to our randomly generated number. Then, because quantum particles collapse out of superposition into a single state when read, we ensured that the final state of our register at the end of the read operation was consistent with a single quantum state (that is, the probability of reading the register in the returned state should be 100%). We did this by setting the value at the read index to 1 and all others to 0.

B-VI: Applying READ to one qubit of a multi-qubit register

```
elif index2 == None: # reading value at one index
      prob = square_all(self.state)
      rp = random.random()
      k = 0
      pairs = []
      for i in range(int((2**self.qubits)/(2**(index1+1)))):
```

```
        for j in range(2**index1):
          pairs.append((k+j, k+j+2**index1))
        k += 2**index1 + 1 + j
      p0 = 0
      p1 = 0
      numpairs = len(pairs)
      for p in pairs:
        p0 += prob[p[0]]
        p1 += prob[p[1]]
      if(rp < p0): # read value is 0
        for l in range (numpairs):
          prob0 = prob[pairs[l][0]]
          self.state[pairs[l][0]] = np.sqrt(prob0 / p0)
          self.state[pairs[l][1]] = 0
        return 0
      else: # read value is 1
        for l in range (numpairs):
          prob1 = prob[pairs[l][1]]
          self.state[pairs[l][1]] = np.sqrt(prob1 / p1)
          self.state[pairs[l][0]] = 0
        return 1
```

To read just one qubit of a multi-qubit register, we combined techniques from operating on one qubit in a multi-qubit state (Appendix B-3) and from reading an entire quantum register (Appendix B-V). We formed the pairs in the same way as for any other operation, but then needed to determine the probability of reading the particular read qubit as a 0 or 1. We summed the probabilities of the first value in each pair to find the total probability of reading a 0, and did the same with the second value in each pair to find the probability of reading a 1. Just like reading an entire register, we generated a random number from 0 to 1, and compared it to the probabilities of reading a 0 or 1 to determine what the READ would yield. In either case, we needed to collapse the probabilities of the states including the value not read to 0, and renormalize the remaining probabilities. For example, if we read 0 in that particular index, then the value at the index stored in the second part of each pair must be set to 0, since the second half of each pair represents the variant of the first half of the pair with a 1 in the specified index instead of 0. Then, since the sum of probabilities (the squares of the values) in the quantum state must equal 1, the remaining probabilities must be renormalized then square-rooted.

B-VII: Reading over a range from a multi-qubit register

```
else: # reading value over range of indices
      binstr = ""
      for i in range(index2, index1 - 1, -1):
        digit = self.read(i)
        binstr = binstr + str(digit)
```

```
        return(int(binstr, 2))
```

Sometimes, it is useful to read over a part of a multi-qubit register. For example, when we perform a Discrete Quantum Walk, we store the coin qubit as the last qubit in the register, and to read the position, we need to read all qubits except the coin qubit. We utilize our ability to read a single qubit from a multi-qubit register and iterate over the desired range, appending the results of each READ into a bitstring that represents the combination of all read values.

B-VIII: Support for simulating multiple runs of a quantum circuit

```
def simulate(circuit, out_size=1, runs=500, as_decimal=False,
*args):
  a = []
  for b in args:
    a.append(b)
  nums = dict()
  for i in range(2**out_size):
    nums[i] = 0
  for j in range(runs):
    c = circuit(registers=a)
    if (c == None):
      raise RuntimeError("circuit has no measurements taken")
      return -1
    nums[c] += 1
  # If as_decimal is True, convert frequencies to decimals
  if as_decimal:
      for key in nums:
          nums[key] = nums[key] / runs
  return nums
```

One important infrastructural capability of a quantum computer simulator is the facilitation of multiple runs of a circuit. We implemented `simulate` to take in the desired circuit as a function parameter, as well as parameters to specify the size of the return register, the number of runs, and a flag to specify the desired format of results. The user can also pass in an unspecified number of unnamed parameters as the `*args` parameter. Any registers used in the circuit should be passed in as *args. This allows the simulation of circuits with any number of registers, which may be of any size.

Each circuit is implemented as a function that can return a measured or read value within a fixed range, specified by the size of the 'return register'. This is represented by the `out_size` parameter. This is used to construct a dictionary of returned values over all the runs. Depending on the `as_decimal` flag, the dictionary may be returned to the user with either the frequency of each return value or the percentage of runs in which that value was obtained. With this dictionary, the

user can construct histograms or plots as desired, to analyze the behavior of their simulated circuits.

Appendix C

C-I: Implementation of the Bell circuit

```
def bell_state(s0=None, s1=None, **kwargs):
  if(s0 == None):
    s0 = kwargs['registers'][0]
  if(s1 == None):
    s1 = kwargs['registers'][1]
  reg = quantum_state.combine_state(s0, s1)
  print("before applying bell circuit")
  viz2(reg.state, 4)
  reg.h(0)
  reg.cnot()
  print("after applying bell circuit")
  viz2(reg.state, 4)
```

The Bell circuit is a classic quantum circuit that generates quantum entanglement. Two single-qubit registers, `s0` and `s1`, are passed in and combined into one register. The Hadamard gate is applied to the first qubit to place it into superposition, and then CNOT is applied to the whole register, so that the state of the second qubit is conditional on the value of the first. In the code snippet, we have two calls to the `viz2()` function (adapted from DD2367 course material), which allows the user to visualize the state vector of the quantum state. Page 5 of this paper shows the results of running this circuit. Before the Bell circuit is applied, the register is in state 0, with no superposition. However, after applying the Bell circuit, the register has a 50% probability of being in either state 0 or state 3. Either the qubits could both be read with value 0 or both with value 1, but they must be the same. The application of cnot to the second qubit, and therefore the value of the second qubit, was dependent on the value of the first qubit, which was in superposition. The read value of each qubit became dependent on the value of the other one. The two qubits became entangled.

C-II: Implementation of the Quantum Arithmetic primitives

```
def add_subtract(reg = None, **kwargs):
  if reg == None:
    reg = kwargs['registers'][0]
  reg.h(2)
  print("before adding")
  viz2(reg.state, 16)
  reg.ccnot(3, [2, 1, 0])
  reg.ccnot(2, [1, 0])
  reg.cnot(1, 0)
```

```
  reg.x(0)
  print("after adding")
  viz2(reg.state, 16)
  reg.x(0)
  reg.cnot(1, 0)
  reg.ccnot(2, [1, 0])
  reg.ccnot(3, [2, 1, 0])
  print("after subtracting")
  viz2(reg.state, 16)
```

Quantum arithmetic consists of both quantum increment and quantum decrement. These two operations are important because the non-cloning theorem limits our ability to use classical arithmetic with quantum registers. Quantum incrementing works in the same way as adding any binary numbers. Iterating down from the most significant bit, it considers if all lower bits are 1. If so, the most significant bit is flipped. This is easily achieved by using CCNOT, targeting the highest bit and conditioning on each lower bit. This is repeated for each lower bit until finally the least significant bit is flipped. Quantum decrementing works in the reverse of quantum incrementing. Quantum arithmetic is important because it demonstrates the correctness of several of our quantum simulator's behaviors. It is also an important primitive that can be used to build other important quantum circuits, such as the Discrete Quantum Walk.

C-III: Implementation of the Discrete Quantum Walk

```
def walk(pos=None, coin=None, steps=30, **kwargs):
  if(pos == None):
    pos = kwargs['registers'][0]
  if(coin == None):
    coin = kwargs['registers'][1]
  cindex = pos.qubits
  reg = quantum_state.combine_state(pos, coin)
  reg.x(6)
  for i in range (steps):
    reg.h(cindex)
    reg.ccnot(6, [cindex, 5, 4, 3, 2, 1, 0])
    reg.ccnot(5, [cindex, 4, 3, 2, 1, 0])
    reg.ccnot(4, [cindex, 3, 2, 1, 0])
    reg.ccnot(3, [cindex, 2, 1, 0])
    reg.ccnot(2, [cindex, 1, 0])
    reg.ccnot(1, [cindex, 0])
    reg.cnot(0, cindex)
    reg.x(cindex)
    reg.cnot(0, cindex)
    reg.ccnot(1, [cindex, 0])
    reg.ccnot(2, [cindex, 1, 0])
    reg.ccnot(3, [cindex, 2, 1, 0])
    reg.ccnot(4, [cindex, 3, 2, 1, 0])
    reg.ccnot(5, [cindex, 4, 3, 2, 1, 0])
```

```
    reg.ccnot(6, [cindex, 5, 4, 3, 2, 1, 0])
    reg.x(cindex)
  ret = reg.read(0, 6)
  return ret
```

A Random Walk is an experiment in which the subject is positioned on a number line, and based on the result of the coin flip, the position in the walk is either incremented or decremented. The Discrete Quantum Walk is a variant of the classical Random Walk, but with a quantum coin. In this case, we controlled our coin qubit with a Hadamard gate and modified the position using quantum arithmetic. Because the Hadamard gate creates superposition, the position on the number line is also in superposition until it is read. The probability distribution of position is impacted by quantum interference. We can see from the results of this experiment on page 7 that, over many runs, the distribution of final position was heavily skewed and that some positions along the line were never measured. This is because the Hadamard gate matrix holds a negative value in only one space ([1][1]), so destructive interference is more common in the direction that the negative value applies to. In this case, since we move right along the line when the coin value is 1, there is more destructive interference moving right along the line, so the graph is skewed left.

C-IV: Implementation of Quantum Fourier Transform

```
def qft3(reg=None, **kwargs):
  if(reg == None):
    reg = kwargs['registers'][0]
  print("before applying qft")
  viz2(reg.state, 8)
  #dft 1
  reg.h(2)
  reg.cphase(2, 1, -np.pi/2)
  reg.cphase(2, 0, -np.pi/4)
  #dft 2
  reg.h(1)
  reg.cphase(1, 0, -np.pi/2)
  #dft 3
  reg.h(0)
  #swap
  reg.cnot(2,0)
  reg.cnot(0,2)
  reg.cnot(2,0)
  print("after applying qft")
  viz2(reg.state, 8)
```

The QFT is a classic quantum circuit that extracts encoded frequencies from the relative phases within a quantum register. We demonstrate the QFT for a 3-qubit register. It applies a Hadamard

and controlled phase operations to each qubit then swaps the outermost qubits. By the end of the circuit, the frequency encoded in the phases will have been converted to a measurable magnitude.

C-V: Implementation of inverse Quantum Fourier Transform

```
def inv_qft(reg=None, **kwargs):
  if(reg == None):
    reg = kwargs['registers'][0]
  print("before applying inv_qft")
  viz2(reg.state, 8)
  # swap
  reg.cnot(2,0)
  reg.cnot(0,2)
  reg.cnot(2,0)
  # reverse dft 1
  reg.h(0)
  reg.cphase(0, 1, np.pi/2)
  reg.cphase(0, 2, np.pi/4)
  # reverse dft 2
  reg.h(1)
  reg.cphase(1, 2, np.pi/2)
  # reverse dft 3
  reg.h(2)
```

In order to prepare an encoded state to test the QFT with, we implemented the inverse QFT. The inverse QFT reverses the steps of the QFT so that the original value of the register is encoded as the frequency of the phases.

Sources

J.D. Hidary. *Quantum Computing: and Applied Approach*

Johnston, Eric R., Nic Harrigan, and Mercedes Gimeno-Segovia. *Programming Quantum Computers*. O'Reilly Media, 2019

Kaiser, Sarah C., and Christopher Granade. *Learn quantum computing with Python and Q#: A hands-on approach*. Simon and Schuster, 2021.

Kempe, Julia. "Quantum random walks: an introductory overview." *Contemporary Physics* 44.4 (2003): 307-327.

Yanofsky, Noson S., and Mirco A. Mannucci. *Quantum computing for computer scientists.* Cambridge University Press, 2008.

Markidis, Stefano. "Mat.7 - Tensor Product.", “1.12 - Bell Pairs & Entanglement”, “2.8 - Introduction on Quantum Fourier Transform” *Quantum Computing for Computer Scientists*, KTH Royal Institute of Technology, Lecture.