Block Encoding for Banded Circulant Matrices

Rishab Khurana

February 2024

Note that the math and implementations of the following sections are heavily inspired by

- Daan Camps and Lin Lin and Roel Van Beeumen and Chao Yang. Explicit Quantum-Circuits for Block Encodings of Certain Sparse Matrices, 2022

1 $BCM_3(\alpha, \beta, \gamma)$, $\alpha = 0.2, \beta = 0.3, \gamma = 0.4$

1.1 The Math

Note that the matrix given by $BCM_3(\alpha, \beta, \gamma)$ is

$$\begin{bmatrix} \alpha & \gamma & 0 & 0 & 0 & 0 & 0 & \beta \\ \beta & \alpha & \gamma & 0 & 0 & 0 & 0 & 0 \\ 0 & \beta & \alpha & \gamma & 0 & 0 & 0 & 0 \\ 0 & 0 & \beta & \alpha & \gamma & 0 & 0 & 0 \\ 0 & 0 & 0 & \beta & \alpha & \gamma & 0 & 0 \\ 0 & 0 & 0 & 0 & \beta & \alpha & \gamma & 0 \\ 0 & 0 & 0 & 0 & 0 & \beta & \alpha & \gamma \\ \gamma & 0 & 0 & 0 & 0 & 0 & \beta & \alpha \end{bmatrix}$$

We apply the following theorem to create a circuit for $BCM_3(\alpha, \beta, \gamma)$.

Theorem 1. [Theorem 4.1 in [Camps et al. 2022]] Let $c(j, l)$ be a function that gives the row index of the l'th (among a list of s) nonzero matrix elements in the j'th column of an s-sparse complex $2^n \times 2^n$ matrix A, where $s = 2^m$. If there exist unitaries O_C, O_A such that

$$\begin{aligned} O_C \ (|l\rangle \otimes |j\rangle) &= |l\rangle \otimes |c(j,l)\rangle \\ O_A \ (|0\rangle \otimes |l\rangle \otimes |j\rangle) &= \left(A_{c(j,l),j} \ |0\rangle \ + \ \sqrt{1 - |A_{c(j,l),j}|^2} \ |1\rangle \right) \otimes |l\rangle \otimes |j\rangle \end{aligned}$$

then

$$U_A = (I_2 \otimes H^{\otimes m} \otimes I_N) \ (I_2 \otimes O_C) \ O_A \ (I_2 \otimes H^{\otimes m} \otimes I_N)$$

For $BCM_3(\alpha, \beta, \gamma)$, we have that $m = 2$ because there are three non-zero elements per column, so 2 bits are needed to encode the number of non-zero elements.

We first need to find a $c(j, l)$ function for $BCM_3(\alpha, \beta, \gamma)$. Note that for a given column, α is always in the jth spot, and α is the 2nd *non*-zero element. The first non-zero element, γ, is always before α unless α is the first element of the column; in which case γ is the last element (which is -1 $(mod\ 8)$). The third non-zero element is always $after$ α but is the first element if α is the last element (which is $9(mod\ 8)$) in a column. In all cases, we can represent $c(j, l)$ for $j = 0, ..., 7$ as the following function

$$c(j,l) = \begin{cases} (j-1)\ (mod\ 8) & l = 0 \\ j & l = 1 \\ (j+1)\ (mod\ 8) & l = 2 \end{cases}$$

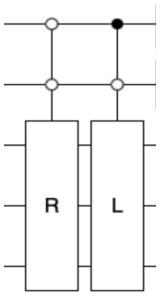
Now, we need to construct O_C. Namely, we need an O_C such that $O_C|l\rangle|j\rangle = |l\rangle|c(j,l)\rangle$. Because $l = 0, 1, 2$, we represent l with two qubits l_0, l_1. Moreoever j ranges from $0 \to 7$, we represent j with three qubits j_0, j_1, j_2. Following the definition of $c(j, l)$, if $l = 00$, we want $|j\rangle \to |(j-1)\ (mod\ 8)\rangle$; if $l = 01$, we do nothing to $|j\rangle$; if $l = 10$, we want $|j\rangle \to |(j+1)\ (mod\ 8)\rangle$. Following the methods of the paper outlined in the beginning of this paper, we can use the following circuits for transforming $|j\rangle$:

(a) *L*-shift circuit **(b)** *R*-shift circuit

Figure 6. The shift circuits.

The above circuits have $R|j\rangle = |(j-1)\ (mod\ 8)\rangle$ and $L|j\rangle = |(j+1)\ (mod\ 8)\rangle$.

Because where we send $|j\rangle$ depends on $|l\rangle = |l_0 l_1\rangle$, we control using these qubits – namely, if $l_0 = 0, l_1 = 0$, we apply R to $|j\rangle$, and if $l_0 = 0, l_1 = 1$, we apply L (and if $l_0 = 1, l_1 = 0$, we do nothing). Thus, the following circuit implements O_C



where the first two lines are l_1, l_0 (in that order) and the last three lines are j_2, j_1, j_0 (in that order).

Now, we construct O_A. Namely, we must have that $O_A|0\rangle|l\rangle|j\rangle = (A_{c(j,l),j}|0\rangle + \sqrt{1 - |A_{c(j,l),j}|^2}|1\rangle) \otimes |l\rangle|j\rangle$. Thus, we need to turn $|0\rangle \to A_{c(j,l),j}|0\rangle + \sqrt{1 - |A_{c(j,l),j}|^2}|1\rangle$, where $A = BCM_3(\alpha, \beta, \gamma)$ in this case. Note that $A_{c(j,l),j}$ is *only* dependent on l – if $l = 0$, we know that $A_{c(j,l),j)} = \gamma$, if $l = 1$, we know that $A_{c(j,l),j)} = \alpha$, and if $l = 2$, we know that $A_{c(j,l),j)} = \beta$. Thus, we apply controlled $R_y(\theta_i)$ gates on $|0\rangle$ with $|l\rangle = |l_1 l_0\rangle$ as the control bits to create O_A. Specifically, we apply $R_y(\theta_0)$ if $l = |00\rangle$; we apply $R_y(\theta_1)$ if $l = |01\rangle$; we apply $R_y(\theta_2)$ if $l = |10\rangle$. We just need to determine the value of θ_i for $i = 0, 1, 2$ to create $\gamma|0\rangle + \sqrt{1 - \gamma^2}|1\rangle$, $\alpha|0\rangle + \sqrt{1 - \alpha^2}|1\rangle$, $\beta|0\rangle + \sqrt{1 - \beta^2}|1\rangle$, respectively. To do this, we follow the computation outlined in (4.24) and (4.25) in the paper cited in the beginning – this computation essentially uses the fact that $\langle 0|R_y(\theta_i)|0\rangle = \cos(\theta_i)/2$ for each i. For γ (which is created by $R_y(\theta_0)$), we have that (for notation, $C_{(i,X(j)),k}(U)$ means apply U to qubit k with qubit i as a control and qubit j as an anticontrol), we have that

$$(I_2 \otimes O_C) C_{(X(1),X(2)),0}(R_y(\theta_0))(I_2 \otimes H^{\otimes 2} \otimes I_3)|0\rangle|00\rangle|j\rangle$$

$$= \frac{1}{2}(I_2 \otimes O_C) C_{(X(1),X(2)),0}(R_y(\theta_0))|0\rangle(|00\rangle + |01\rangle + |10\rangle + |11\rangle)|j\rangle$$

$$= \frac{1}{2}(I_2 \otimes O_C)(R_y(\theta_0)|0\rangle|00\rangle + |0\rangle|01\rangle + |0\rangle|10\rangle + |0\rangle|11\rangle)|j\rangle$$

$$= \frac{1}{2}(R_y(\theta_0)|0\rangle|00\rangle|j - 1 \ (mod\ 8)\rangle + |0\rangle|01\rangle|j\rangle + |0\rangle|10\rangle|j + 1 \ (mod\ 8)\rangle + |0\rangle|11\rangle|j\rangle)$$

Now, taking the inner product of the above with $(I_2 \otimes H^{\otimes 2} \otimes I_3)|0\rangle|00\rangle|j-1(mod\ 8)\rangle$ has the following result:

$$\frac{1}{4}\langle j-1(mod\,8)|\langle 00|\langle 0|R_y(\theta_0)|0\rangle|00\rangle|j-1(mod\,8)\rangle = \frac{1}{4}\cos(\theta_0/2)\ (using\ \langle 0|R_y(\theta_0)|0\rangle = \cos(\theta_0/2))$$

Now, using the fact that the above inner product equals $\gamma/4$, we have that $\theta_0 = 2\arccos(\gamma)$. Now, we do the computation for $R_y(\theta_1)$. We have that

$$(I_2 \otimes O_C)C_{(X(1),2),0}(R_y(\theta_1))(I_2 \otimes H^{\otimes 2} \otimes I_3)|0\rangle|00\rangle|j\rangle$$

$$= \frac{1}{2}(|0\rangle|00\rangle|j-1\ (mod\ 8)\rangle + R_y(\theta_1)|0\rangle|01\rangle|j\rangle + |0\rangle|10\rangle|j+1\ (mod\ 8)\rangle + |0\rangle|11\rangle|j\rangle)$$

Now, to isolate $R_y(\theta_1)$, we take the inner product of the above with $(I_2 \otimes H^{\otimes 2} \otimes I_3)|0\rangle|00\rangle|j\rangle$, but because there are two terms with $|j\rangle$ in the above expression, we end up with the following result:

$$\frac{1}{4}\langle j|\langle 01|\langle 0|R_y(\theta_1)|0\rangle|01\rangle|j\rangle + \frac{1}{4}\langle j|\langle 11|\langle 0|0\rangle|11\rangle|j\rangle = \frac{1}{4}\cos(\theta_1/2) + \frac{1}{4}$$

Now, using the fact that the above inner product is equal to $\alpha/4$, we have that $\theta_1 = 2\arccos(\alpha - 1)$. Finally, we find θ_2. We have that

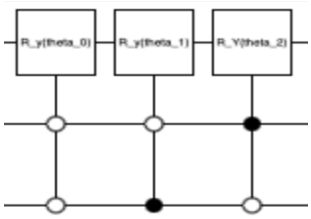
$$(I_2 \otimes O_C)C_{(1,X(2)),0}(R_y(\theta_2))(I_2 \otimes H^{\otimes 2} \otimes I_3)|0\rangle|00\rangle|j\rangle$$

$$= \frac{1}{2}(|0\rangle|00\rangle|j-1\ (mod\ 8)\rangle + |0\rangle|01\rangle|j\rangle + R_y(\theta_2)|0\rangle|10\rangle|j+1\ (mod\ 8)\rangle + |0\rangle|11\rangle|j\rangle)$$

And we take the inner product of the above with $(I_2 \otimes H^{\otimes 2} \otimes I_3)|0\rangle|00\rangle|j+1(mod\ 8)\rangle$ to isolate $R_y(\theta_2)$ and we get the following result:

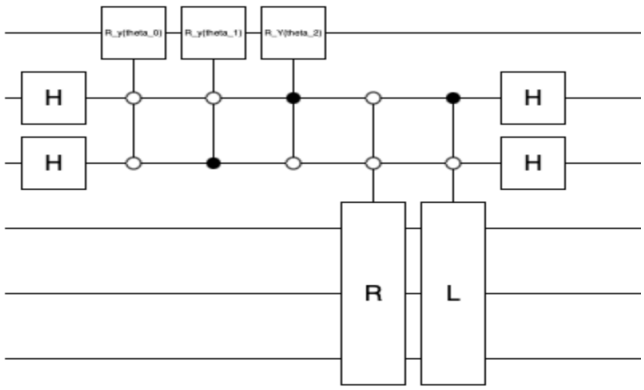
$$\frac{1}{4}\langle j+1\ (mod\ 8)|\langle 10|\langle 0|R_y(\theta_2)|0\rangle|10\rangle|j+1\ (mod\ 8)\rangle = \frac{1}{4}\cos(\theta_2/2)$$

Thus, because the above inner product is equal to $\beta/4$, we have that $\theta_2 = 2\arccos(\beta)$. Now, we can fully construct O_A:



where $R\_y(theta\_i)$ is a y rotation using the θ_i computed above; and the first line is $|0\rangle$, and the second two lines represent $|l_1\rangle, |l_0\rangle$ (in that order). Note the control bits – if $l = 00 = 0$, we know to create $\gamma|0\rangle + \sqrt{1-\gamma^2}|1\rangle$, so we rotate by $\theta_0 = 2\arccos(\gamma)$; if $l = 01 = 1$, we know to create $\alpha|0\rangle + \sqrt{1-\alpha^2}|1\rangle$, so we rotate by $\theta_1 = 2\arccos(\alpha - 1)$; if $l = 10 = 2$, we know to create $\beta|0\rangle + \sqrt{1-\beta^2}|1\rangle$, so we rotate by $\theta_2 = 2\arccos(\beta)$.

Now that we have O_A and O_C, we can construct $U_A = (I_2 \otimes H^{\otimes 2} \otimes I_8)(I_2 \otimes O_C)O_A(I_2 \otimes H^{\otimes 2} \otimes I_8)$ as the following circuit:



where the inputs are: the first line is $|0\rangle$, the second line is $|0\rangle$, the third line is $|0\rangle$, and the fourth, fifth, and sixth lines encode the input unit vector $|x\rangle$ for A. Then, to determine $A|x\rangle$, we run the above circuit, and if the first three bits are $|000\rangle$, the last three bits represent a normalized $A|x\rangle$ – so by rerunning the circuit several times, we can determine the superposition (probabilities) for each term in $A|x\rangle$. We now implement the above circuit in Qiskit so that we can test its accuracy.

1.2 Implementation

The implementation is simply just recreating the above circuit in Qiskit. First, we define constants:

```
#setting constants
alpha = 0.2
beta = 0.3
gamma = 0.4

theta_0  = 2*math.acos(gamma)
theta_1 = 2*math.acos(alpha - 1)
theta_2 = 2*math.acos(beta)
```

Then, we must go on to create O_A, which requires the use of control R_y gates with two control bits. We create these gates using the Qiskit control() function:

```
#contstructing R_Y(theta_0), R_Y(theta_1), R_Y(theta_2)
ccry_theta_0 = RYGate(theta_0).control(2, label=None)
ccry_theta_1 = RYGate(theta_1).control(2, label=None)
ccry_theta_2 = RYGate(theta_2).control(2, label=None)
```

To specify the control qubits, we specify them as the first two qubits when appending the above gates to a circuit. Now, we can create O_A (following the circuit from the math section):

```
#creating O_A (product of controlled rotations)
O_A = QuantumCircuit(3, name="O_A")
O_A.barrier()
#appending first controlled rotation
O_A.x(1)
O_A.x(2)
O_A.append(ccry_theta_0, [1,2,0])
O_A.x(1)
O_A.x(2)
O_A.barrier()
#appending second controlled rotation
O_A.x(1)
O_A.append(ccry_theta_1, [1,2,0])
O_A.x(1)
O_A.barrier()
#appending third controlled rotation
O_A.x(2)
O_A.append(ccry_theta_2, [1,2,0])
O_A.x(2)
O_A.barrier()
```

As you can see, to simulate anti-control bits, we just apply X to the control bit before and after the control gate. Now, to create O_C, we first create the controlled L and R gates (whose implementation is outlined above and in the research paper cited in the beginning):

```
#creating double controlled Right-Shift, double controlled Left-Shift circuit
R_Shift = QuantumCircuit(3, name="R")
R_Shift.x(1)
R_Shift.x(2)
R_Shift.ccx(1,2,0)
R_Shift.x(1)
R_Shift.x(2)
R_Shift.x(2)
R_Shift.cx(2,1)
R_Shift.x(2)
R_Shift.x(2)
cc_R_Shift_gate = R_Shift.to_gate().control(2, label=None)

L_Shift = QuantumCircuit(3, name="L")
L_Shift.ccx(1,2,0)
L_Shift.cx(2,1)
L_Shift.x(2)
cc_L_Shift_gate = L_Shift.to_gate().control(2, label=None)
```

Then, using the above implementations of R and L, we create O_C by following the circuit from the math section:

```
#creating O_C
O_C = QuantumCircuit(5, name="O_C")
O_C.x(0)
O_C.x(1)
O_C.append(cc_R_Shift_gate, [0,1,2,3,4])
O_C.x(0)
O_C.x(1)
O_C.barrier()
O_C.x(1)
O_C.append(cc_L_Shift_gate, [0,1,2,3,4])
O_C.x(1)
```

Finally, now that we have O_A and O_C, we can create the $BCM_3(\alpha, \beta, \gamma)$ circuit:

```
#creating BCM circuit
BCM = QuantumCircuit(6, 6)
BCM.reset(range(6))

#start put instructions here for encoding input vector |x>
#i.e. BCM.h(5)

#end

BCM.h(1)
BCM.h(2)
BCM.barrier()
BCM.append(O_A, [0,1,2])
BCM.barrier()
BCM.append(O_C, [1,2,3,4,5])
BCM.barrier()
BCM.h(1)
BCM.h(2)
BCM.barrier()
BCM.measure([i for i in range(6)], [i for i in range(6)])
```

Note the comment in the beginning of the implementation – we can input instructions to encode $|x\rangle$ and therefore find what $BCM_3(\alpha, \beta, \gamma)|x\rangle$ is. This is what we do in the next section – we test the above implementation on a few test cases.

1.3 Testing

Note that we refer to $BCM_3(\alpha, \beta, \gamma)$ as A. We will test the above implementations on four test cases. The first is $|x\rangle = (1\ 0\ 0\ 0\ 0\ 0\ 0\ 0)^T$. The second is $|x\rangle = (0\ 0\ 0\ 1\ 0\ 0\ 0\ 0)^T$. The third is $|x\rangle = (\frac{1}{\sqrt{2}}\ \frac{1}{\sqrt{2}}\ 0\ 0\ 0\ 0\ 0\ 0)^T$. The fourth is $|x\rangle = (\frac{1}{2}\ 0\ \frac{1}{2}\ 0\ \frac{1}{2}\ 0\ \frac{1}{2}\ 0)^T$.

The code for testing is below:

```python
#Test-bench:
outs = []
count = 0
backend = Aer.get_backend("aer_simulator") #accurate simualtor using no noise
bit_str_dict = {}
total_zeros = 0
iters = 5000
normalization_factor = math.sqrt(alpha**2 + beta**2 + gamma**2) #changes for each test case |x>

for i in range(iters):
    job = backend.run(BCM.decompose(reps = 6), shots=1, memory=True)
    output = job.result().get_memory()[0][::-1] #need to reverse do to formatting (big-endian but needs to be little endian)
    if (output[0] == '0' and output[1] == '0' and output[2] == '0'):
      bit_str = output[3:]
      if (bit_str in bit_str_dict):
        bit_str_dict[bit_str] += 1
      else:
        bit_str_dict[bit_str] = 1
      total_zeros += 1

for key in bit_str_dict:
  print("coefficient of " + str(key) + ": " + str(normalization_factor*math.sqrt(bit_str_dict[key]/total_zeros)))
```

The idea is that we run the circuit thousands of times, and for each output that has 000 as qubits $0, 1, 2$, we know that qubits $3, 4, 5$ is a measurement of $A|x\rangle = a_0|000\rangle + ... + a_7|111\rangle$. Thus, by running the circuit several times, we can determine probabilities – $|a_i|^2$ – and thus compute the superposition of a normalized $A|x\rangle$ (by taking the square root of the probabilities). Then, we can see if the output is correct by multiplying the normalized coefficients by $||A|x\rangle||$. So, we measure the frequency of the outputs for qubits $3, 4, 5$.

We will start with $|x\rangle = (1\ 0\ 0\ 0\ 0\ 0\ 0\ 0)^T$. We have that

$$A|x\rangle = (\alpha\ \beta\ 0\ 0\ 0\ 0\ 0\ \gamma)^T = (0.2\ 0.3\ 0\ 0\ 0\ 0\ 0\ 0.4)^T$$

Note that the encoded $|x\rangle$ is just $|000\rangle$. Thus, we need not input any instructions into our BCM circuit to test the output for this $|x\rangle$. The output is $A|x\rangle = \alpha|000\rangle + \beta|001\rangle + \gamma|111\rangle = 0.2|000\rangle$ with norm $||A|x\rangle|| = \sqrt{\alpha^2 + \beta^2 + \gamma^2}$. Thus, to test this $|x\rangle$, we set normalization_factor = math.sqrt(alpha**2 + beta**2 + gamma**2), and we see the resulting coefficients. We have the following output:

```
(base) user@wifi-131-179-53-4 Banded Circulant Matrix Encoding\n % python3 circulant_matrix_sim.py
coefficient of 001: 0.2916619047230315
coefficient of 000: 0.2062361106434403
coefficient of 111: 0.4029888335921978
```

Thus, our only outputs are 001, 000, and 111, which is exactly what we want because $A|x\rangle = \alpha|000\rangle + \beta|001\rangle + \gamma|111\rangle = 0.2|000\rangle + 0.3|001\rangle + 0.4|111\rangle$. Moreoever, we can see that the coefficients are almost the same as the coefficients for $A|x\rangle$ – the very minimal difference is perfectly described by the fact that we are using frequencies instead of theoretical probabilities.

We now test with $|x\rangle = (0\ 0\ 0\ 1\ 0\ 0\ 0\ 0)^T = |011\rangle$. To get this encoding of $|x\rangle$, we must apply X to the 5th and 6th qubits before running BCM (because the last three qubits represent input vectors into BCM). Thus, we add the following code:

```
#creating BCM circuit
BCM = QuantumCircuit(6, 6)
BCM.reset(range(6))

#start put instructions here for encoding input vector |x>
#i.e. BCM.h(5)

#for x = (0 0 0 1 0 0 0 0)^T
BCM.x(4)
BCM.x(5)
#end

BCM.h(1)
BCM.h(2)
BCM.barrier()
BCM.append(O_A, [0,1,2])
BCM.barrier()
BCM.append(O_C, [1,2,3,4,5])
BCM.barrier()
BCM.h(1)
BCM.h(2)
BCM.barrier()
BCM.measure([i for i in range(6)], [i for i in range(6)])
```

Our desired output is

$$A|x\rangle = (0\ 0\ \gamma\ \alpha\ \beta\ 0\ 0\ 0)^T = \gamma|010\rangle + \alpha|011\rangle + \beta|100\rangle = 0.4|010\rangle + 0.2|011\rangle + 0.3|100\rangle$$

with norm $||A|x\rangle|| = \sqrt{\alpha^2 + \beta^2 + \gamma^2}$ – so we set the normalization factor as normalization_factor = math.sqrt(alpha**2 + beta**2 + gamma**2). Thus, running the test bench, we get the following output:

```
(base) user@wifi-131-179-53-4 Banded Circulant Matrix Encoding\n % python3 circulant_matrix_sim.py
coefficient of 010: 0.4132526803031988
coefficient of 100: 0.29495762407505255
coefficient of 011: 0.17950549357115014
```

As you can see, the correct bit sequences are outputted, and the coefficients match very well (with little difference that, again, can be attributed to the fact that we are working with frequencies).

Now, we test with $|x\rangle = (\frac{1}{\sqrt{2}}\ \frac{1}{\sqrt{2}}\ 0\ 0\ 0\ 0\ 0\ 0)^T = \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|001\rangle$. To get this encoding, we just need to apply Hadamard to the last qubit. Thus, we add the following code:

```
#start put instructions here for encoding input vector |x>
#i.e. BCM.h(5)

#for x = (1/sqrt(2) 1/sqrt(2) 0 0 0 0 0 0)^T
BCM.h(5)
#end
```

Our desired output is

$$A|x\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} \alpha+\gamma \\ \beta+\alpha \\ \beta \\ 0 \\ 0 \\ 0 \\ 0 \\ \gamma \end{bmatrix} = 0.4243|000\rangle + 0.3536|001\rangle + 0.2121|010\rangle + 0.2828|111\rangle$$

with norm $||A|x\rangle|| = \sqrt{(\alpha+\gamma)^2/2 + (\beta+\alpha)^2/2 + \beta^2/2 + \gamma^2/2}$ – so we set the normalization_factor acccordingly. Running this on the test bench, we get the following output:

```
(base) user@wifi-131-179-53-4 Banded Circulant Matrix Encoding\n % python3 circulant_matrix_sim.py
coefficient of 111: 0.3126136157099891
coefficient of 000: 0.41551355591787276
coefficient of 010: 0.22830070547681652
coefficient of 001: 0.32787192621510003
```

As you can see again, the correct bit sequences are there, and the coefficients match decently well, but there is more error with this test case. This is likely because there are more possibilities despite the number of iterations remaining the same – thus, the frequencies will take longer to converge. In increasing the number of iterations, the simulation took too long, so I leave this to the interested reader.

Now, we move onto $|x\rangle = (\frac{1}{2}\ 0\ \frac{1}{2}\ 0\ \frac{1}{2}\ 0\ \frac{1}{2}\ 0)^T = \frac{1}{2}|000\rangle + \frac{1}{2}|010\rangle + \frac{1}{2}|100\rangle + \frac{1}{2}|110\rangle = |+\rangle|+\rangle|0\rangle$. Thus, to get this encoding, we apply H to the 4th and 5th qubit and leave the 6th qubit as $|0\rangle$. Thus, we add the following code:

```
#start put instructions here for encoding input vector |x>
#i.e. BCM.h(5)

#for x = (1/2 0 1/2 0 1/2 0 1/2 0)^T
BCM.h(3)
BCM.h(4)
#end
```

Our desired output is

$$A|x\rangle = \frac{1}{2}\begin{bmatrix} \alpha \\ \beta+\gamma \\ \alpha \\ \beta+\gamma \\ \alpha \\ \beta+\gamma \\ \alpha \\ \gamma+\beta \end{bmatrix} = \frac{1}{2}\alpha(|000\rangle+|010\rangle+|100\rangle+|110\rangle)+\frac{1}{2}(\beta+\gamma)(|001\rangle+|011\rangle+|101\rangle+|111\rangle)$$

$$= 0.1(|000\rangle + |010\rangle + |100\rangle + |110\rangle) + 0.35(|001\rangle + |011\rangle + |101\rangle + |111\rangle)$$

with norm $||A|x\rangle|| = \sqrt{\alpha^2 + (\beta+\gamma)^2}$, so we set the normalization_factor variable as such. Running this on the test bench, we get the following output:

```
(base) user@wifi-131-179-53-4 Banded Circulant Matrix Encoding\n % python3 circulant_matrix_sim.py
coefficient of 111: 0.32323410515963696
coefficient of 001: 0.37238935957603303
coefficient of 101: 0.35139251914199954
coefficient of 011: 0.35675760070821533
coefficient of 000: 0.1307546335452338
coefficient of 110: 0.08716975569682255
coefficient of 100: 0.09745874966007237
coefficient of 010: 0.04358487784841127
```

All the possible outputs are there, which is desired, and most of the coefficients closely match the coefficients of $A|x\rangle$, but the coefficient of 010 is quite a bit lower. I believe this can again be attributed to the fact that the number of iterations is the same despite the number of possibilities doubling in this case – to accommodate for correct computation of $A|x\rangle$, one must run the circuit thousands of more times to get better probabilities/frequencies.

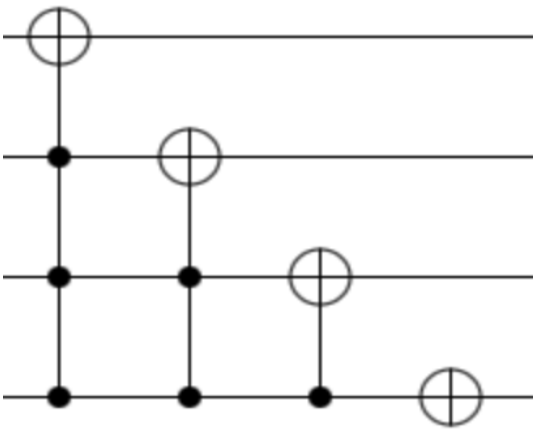
Thus, these test cases do show a flaw with block encoding. To correctly compute $A|x\rangle$, one must run the BCM circuit *a lot* to get accurate results. So the question of whether or not it is more efficient than classical computing is important to consider given these test results.

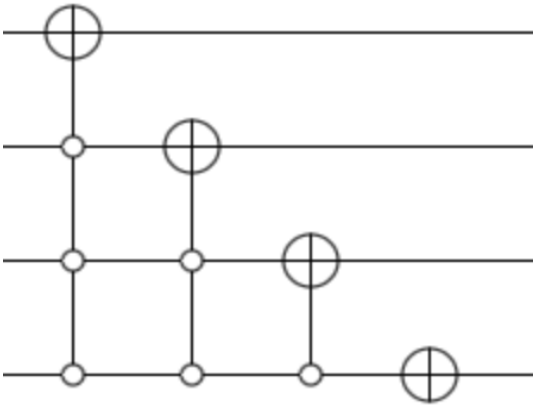
2 $BCM_4(\alpha, \beta, \gamma)$, $\alpha = 0.2, \beta = 0.3, \gamma = 0.4$

We skip the math portion for this section because it is very similar to the math portion for the previous section. The only real differences are things like changes from *mod* 8 to *mod* 16 and the number of qubits for j changing to 4 instead of 3. l stays the same, however, because the matrix still only contains 3 elements per row (so it is still 4-sparse). And because O_A only relies on l and because the matrix is still 4-sparse, most of the circuit is the same. The only thing that needs to be discussed is O_C, which we do in the implementation section.

2.1 Implementation

The only real difference for the implementation has to do with the L and R unitaries constructed in the previous section's implementation. This time, we have L as the left circuit and R as the right circuit:

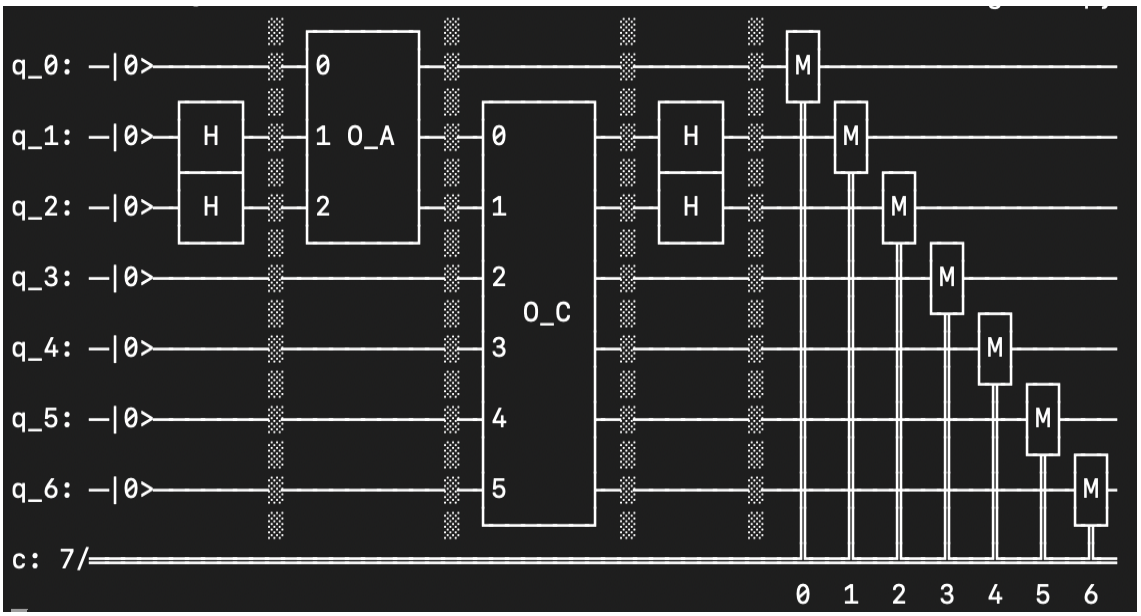




The construction is essentially the same as the paper's construction for three qubits – it is just extended to work for 4 because $|j\rangle$ is now a 4 qubit input. In code, we have R and L as

```
#creating double controlled Right-Shift, double controlled Left-Shift circuit
R_Shift = QuantumCircuit(4, name="R")

R_Shift.x(1)
R_Shift.x(2)
R_Shift.x(3)
R_Shift.mcx([1,2,3],0)
R_Shift.x(1)
R_Shift.x(2)
R_Shift.x(3)

R_Shift.x(2)
R_Shift.x(3)
R_Shift.ccx(2,3,1)
R_Shift.x(2)
R_Shift.x(3)

R_Shift.x(3)
R_Shift.cx(3, 2)
R_Shift.x(3)

R_Shift.x(3)

cc_R_Shift_gate = R_Shift.to_gate().control(2, label=None)

L_Shift = QuantumCircuit(4, name="L")
L_Shift.mcx([1,2,3], 0)
L_Shift.ccx(2,3, 1)
L_Shift.cx(3, 2)
L_Shift.x(3)
cc_L_Shift_gate = L_Shift.to_gate().control(2, label=None)
```

Other than the above, the code from BCM_3 is nearly identical to the code for BCM_4 – BCM_4 just has another input qubit in the circuit. Below is the circuit for BCM_4 (compiled by qiskit):



We now test BCM_4.

2.2 Testing

We will test with $|x\rangle = (1\ 0\ 0\ 0\ 0 ...\ 0)^T$ and $|x\rangle = (0\ 0\ 0\ 0\ 1\ 0\ ...\ 0)^T$ (we use less cases because BCM_4 takes significantly longer to run). The test bench is the same as the previous one as well – we just now read the last 4 qubits instead of 3 (and we deal with a $2^4 \times 2^4$ matrix A this time). The number of iterations also had to be decreased because BCM_4 takes a lot longer to simulate (due to there being 7 qubits).

For $|x\rangle = (1\ 0\ 0\ 0\ 0 ...\ 0) = |0000\rangle$, we need not change anything in BCM_4 to encode $|x\rangle$ because it is the just the $|0^4\rangle$ qubit. We have that

$$A|x\rangle = \alpha|0000\rangle + \beta|0001\rangle + \gamma|1111\rangle = 0.2|0000\rangle + 0.3|0001\rangle + 0.4|1111\rangle$$

And thus our normalization factor is $||A|x\rangle|| = \sqrt{\alpha^2 + \beta^2 + \gamma^2}$. Putting this into our code, we get the following output for coefficients:

```
(base) user@wifi-131-179-53-4 Banded Circulant Matrix Encoding\n % python3 BCM_4_matrix_sim.py
coefficient of 1111: 0.4454447806761002
coefficient of 0001: 0.27625312572654126
coefficient of 0000: 0.12354415362426847
```

The possibilities are all correct (0000, 0001, 1111), but the accuracies are much lower here – this is because I ran with less iterations due to how long it takes to run with 7 qubits. The coefficient of 1111 is slightly off, the coefficient of 0001 is slightly off, but the coefficient of 0000 is the worst.

Now, we test with $|x\rangle = (0\ 0\ 0\ 0\ 1\ 0\ ...\ 0)^T = |0100\rangle$. To send this $|x\rangle$ as input, we simply just apply X to the 5th qubit. Thus, we insert the following code before running BCM:

```
#start put instructions here for encoding input vector |x>
#i.e. BCM.h(5)
BCM.x(4)
#end
```

The desired output is

$$A|x\rangle = \gamma|0011\rangle + \alpha|0100\rangle + \beta|0101\rangle = 0.4|0011\rangle + 0.2|0100\rangle + 0.3|0101\rangle$$

And our normalization factor is $\sqrt{\alpha^2 + \beta^2 + \gamma^2}$ again. Running the code, we get the following output:

```
(base) user@wifi-131-179-53-4 Banded Circulant Matrix Encoding\n % python3 BCM_4_matrix_sim.py
coefficient of 0100: 0.20354009783964297
coefficient of 0101: 0.3109126351029605
coefficient of 0011: 0.38974961437410294
```

Thus, all the possibilities are correct, and the coefficients are very close here – in fact, there is less error in this case compared to the other case. However, I believe the outputs are close enough to the actual value to conclude that the algorithm is correct (and error is due to frequencies//not enough iterations) – the same can be said for the previous part.

However, like what we saw in the previous part, block encoding is hard because it requires running the algorithm several times, and it can be quite computationally expensive, especially with BCM_4.