Hamiltonian Simulation

Rishab Khurana

February 2024

We will simulate the solution to the Schrodinger Equation $|\psi(t)\rangle = e^{-iHt}$ for Hamiltonian $H = \sum_{j=0}^{n-1} X_j$, where X_j is the pauli X gate acting on qubit j. We fix $n = 3$. Now, suppose we have error bound $\epsilon = 0.05$, and we want to simulate at time $t = 0.05$. Suppose, we have initial state $|\varphi\rangle$, which is an arbitrary n-qubit vector – we will test on specific φ in the **Testing** section.

1 The Math

Note that the Trotter-Suzuki technique isn't necessarily needed here because the X_j's commute (we could just create a circuit for e^{-iHt} directly), but for the sake of applying what the Professor talked about in lecture notes, I implement e^{-iHt} using the Trotter-Suzuki technique.

Using the Trotter-Suzuki technique, We compute e^{-iHt} by approximating it with

$$e^{-iHt} \approx (e^{-iX_0 t/r} e^{-iX_1 t/r} ... e^{-iX_{n-1} t/r})^r$$

The error bound for the above approximation is t^2/r, and because we want error less than ϵ, we choose r such that $t^2/r < \epsilon \implies$ we set $r = \lfloor t^2/\epsilon \rfloor + 1$. So, we must create a circuit that outputs $(e^{-iX_0 t/r} e^{-iX_1 t/r} e^{-iX_2 t/r})^r |\varphi\rangle$, which approximates $e^{-iHt}|\varphi\rangle$.

We first create a circuit for $e^{-iX_j t/r}$. We diagonalize X_j and get that $X_j = \mathbf{H}_j Z_j \mathbf{H}_j$, where $\mathbf{H}_j$ is Hadamard applied to qubit j, and Z_j is Z applied to qubit j. Now we exponentiate:

$$e^{-iX_j t/r} = \mathbf{H}_j e^{-iZ_j t/r} \mathbf{H}_j$$

Note that Z's eigenvectors are the elementary basis vectors: $Z|q\rangle = (-1)^q |q\rangle$ for $q \in \{0, 1\}$. Thus, we can interchange Z_j with it's scalar effect:

$$e^{-iZ_j t/r}|q\rangle_j = e^{-i(-1)^q t/r}|q\rangle_j$$
$$\implies e^{-iZ_j t/r}|0\rangle_j = e^{-it/r}|0\rangle_j$$
$$\implies e^{-iZ_j t/r}|1\rangle_j = e^{it/r}|1\rangle_j$$

The j subscript means we talk only of the jth qubit. Using the above behavior on the basis vectors, we can conclude that

$$e^{-iZ_j t/r} = I \otimes I \otimes ... \otimes \begin{bmatrix} e^{-it/r} & 0 \\ 0 & e^{it/r} \end{bmatrix} \otimes I \otimes ... \otimes I = I \otimes I \otimes ... \otimes R_z(2t/r) \otimes I \otimes ... \otimes I$$

$$\implies e^{-iZ_j/r} = R_z(2t/r)_j$$

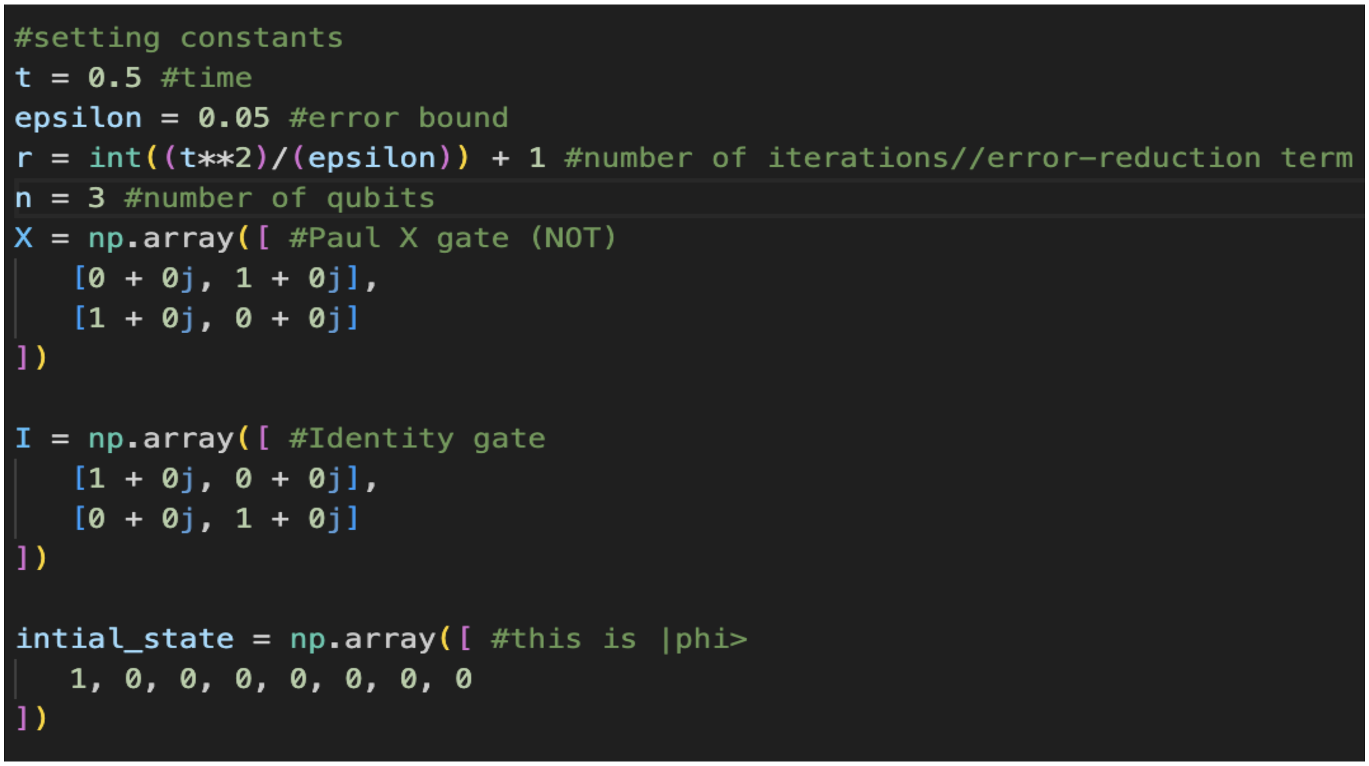
Therefore, the circuit for $e^{-iX_j t/r}$ can be implemented as $\mathbf{H}_j R_z(2t/r)_j \mathbf{H}_j$.

Now, we create the circuit for our approximation of e^{-iHt}. Using the above computation, for $n = 3$ qubits, we have that

$$e^{-iHt} \approx ((\mathbf{H}_0 R_z(2t/r)_0 \mathbf{H}_0)(\mathbf{H}_1 R_z(2t/r)_1 \mathbf{H}_1)(\mathbf{H}_2 R_z(2t/r)_2 \mathbf{H}_2))^r$$

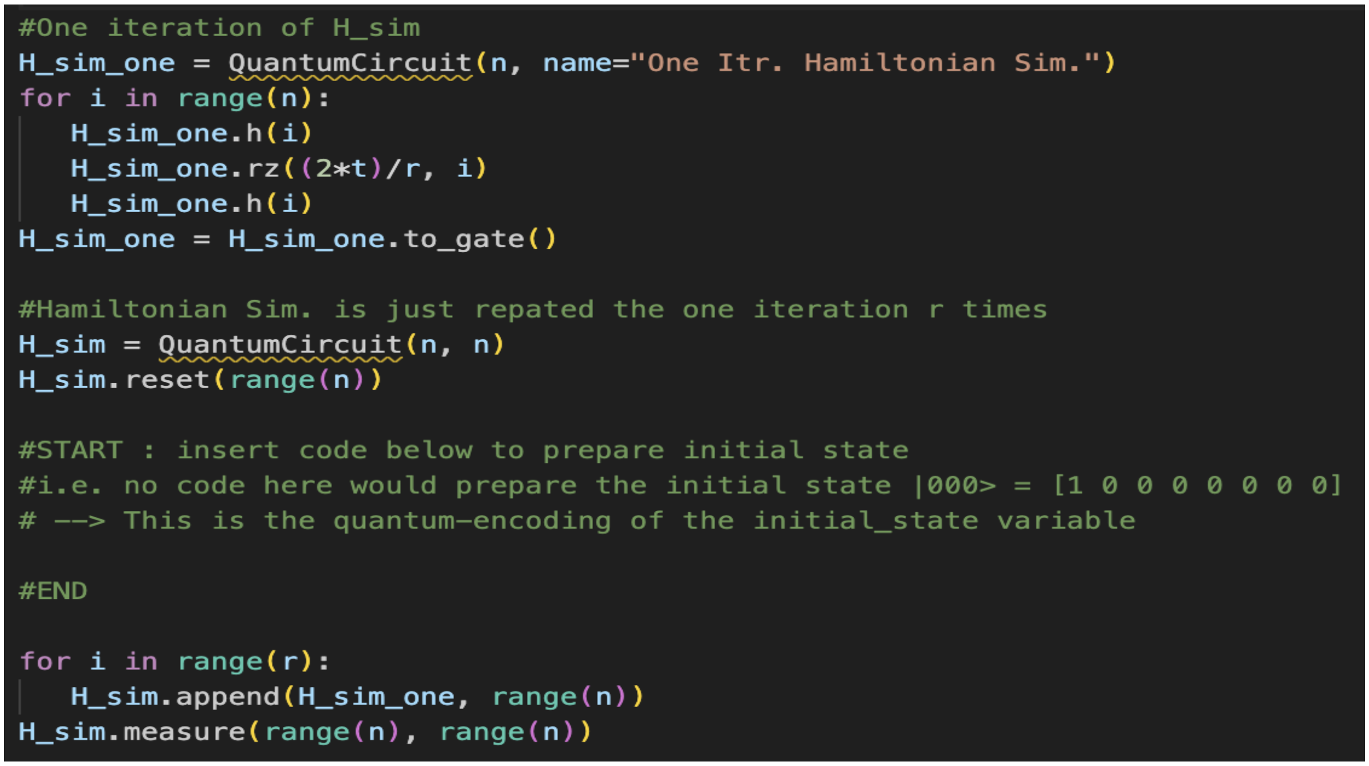
2 Implementation

We implement the above circuit/formula in Qiskit. The first thing we do is define our constants in code.



```
#setting constants
t = 0.5 #time
epsilon = 0.05 #error bound
r = int((t**2)/(epsilon)) + 1 #number of iterations//error-reduction term
n = 3 #number of qubits
X = np.array([ #Paul X gate (NOT)
    [0 + 0j, 1 + 0j],
    [1 + 0j, 0 + 0j]
])

I = np.array([ #Identity gate
    [1 + 0j, 0 + 0j],
    [0 + 0j, 1 + 0j]
])

intial_state = np.array([ #this is |phi>
    1, 0, 0, 0, 0, 0, 0, 0
])
```

The declaration of the matrix version of the X and I gates are for use in the **Testing** section; same goes for the initial_state vector. The actual implementation of the Hamiltonian simulation is simple:



```
#One iteration of H_sim
H_sim_one = QuantumCircuit(n, name="One Itr. Hamiltonian Sim.")
for i in range(n):
    H_sim_one.h(i)
    H_sim_one.rz((2*t)/r, i)
    H_sim_one.h(i)
H_sim_one = H_sim_one.to_gate()

#Hamiltonian Sim. is just repated the one iteration r times
H_sim = QuantumCircuit(n, n)
H_sim.reset(range(n))

#START : insert code below to prepare initial state
#i.e. no code here would prepare the initial state |000> = [1 0 0 0 0 0 0 0]
# --> This is the quantum-encoding of the initial_state variable

#END

for i in range(r):
    H_sim.append(H_sim_one, range(n))
H_sim.measure(range(n), range(n))
```

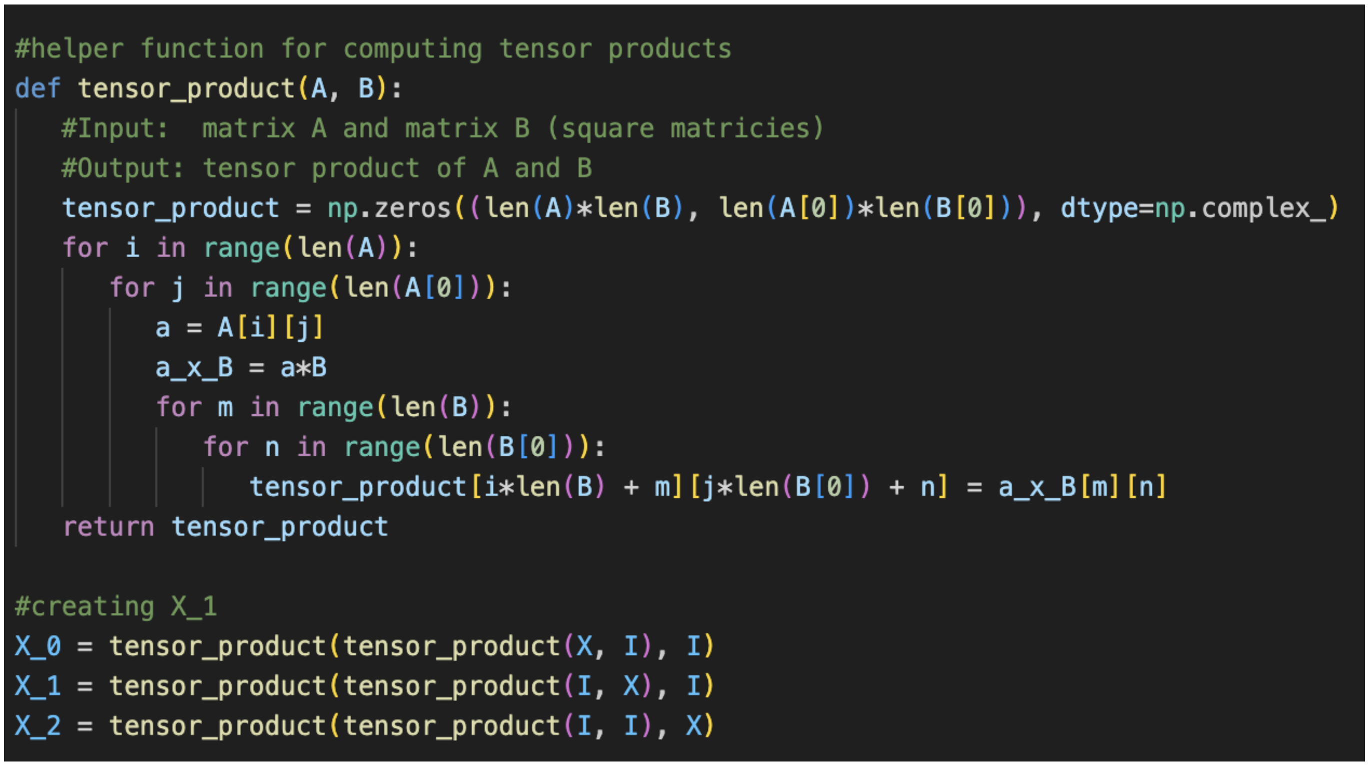
We first create a QuantumCircuit that implements

$$(\mathbf{H}_0 R_z(2t/r)_0 \mathbf{H}_0)(\mathbf{H}_1 R_z(2t/r)_1 \mathbf{H}_1)(\mathbf{H}_2 R_z(2t/r)_2 \mathbf{H}_2)$$

As you can see above, this is in H_sim_one. Just like the formula, we apply Hadamard, the Z rotation of $2t/r$, and another Hadadmard to each qubit. Then, to implement the Hamiltonian simulation, we just need to repeat H_sim_one r times, just as we saw in the **Math** section; and the superposition we measure at the end approximates $e^{-iHt}|\varphi\rangle$ – in the above screenshot, there is no preparing of the qubits before running H_sim, so we have that $|\varphi\rangle = |000\rangle$, but we test with different $|\varphi\rangle$ in the next section.

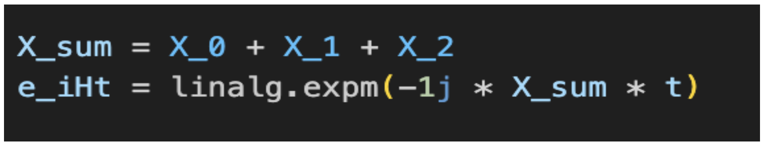
3 Testing

To test the accuracy of our implementation of the Hamiltonian simulation, we must classically compute e^{-iHt} for $H = X_0 + X_1 + X_2$. Thus, we need to first create $X_0 = X \otimes I \otimes I$, $X_1 = I \otimes X \otimes I$, $X_2 = I \otimes I \otimes X$. We do this below:



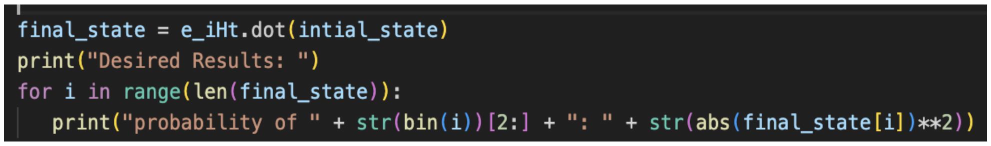
```
#helper function for computing tensor products
def tensor_product(A, B):
    #Input:  matrix A and matrix B (square matricies)
    #Output: tensor product of A and B
    tensor_product = np.zeros((len(A)*len(B), len(A[0])*len(B[0])), dtype=np.complex_)
    for i in range(len(A)):
        for j in range(len(A[0])):
            a = A[i][j]
            a_x_B = a*B
            for m in range(len(B)):
                for n in range(len(B[0])):
                    tensor_product[i*len(B) + m][j*len(B[0]) + n] = a_x_B[m][n]
    return tensor_product

#creating X_1
X_0 = tensor_product(tensor_product(X, I), I)
X_1 = tensor_product(tensor_product(I, X), I)
X_2 = tensor_product(tensor_product(I, I), X)
```

We use a helper tensor_product function (this is grabbed from last quarter's homework in which we created a Quantum Simulator). Then, using that function, we create X_0, X_1, and X_2. Now, we sum the three and take the matrix exponential of $-i(X_0 + X_1 + X_2)t$. We do this below:



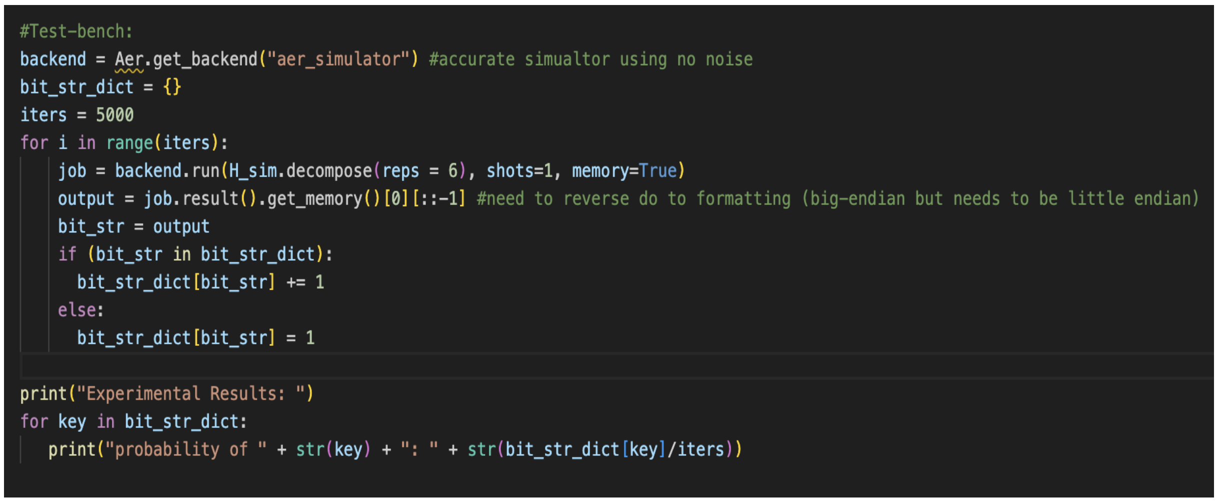
```
X_sum = X_0 + X_1 + X_2
e_iHt = linalg.expm(-1j * X_sum * t)
```

Note that the linalg.expm() function takes the exponential of a matrix. Now, we can apply e_iHt to any initial state of our choosing to see our desired output:


```python
final_state = e_iHt.dot(intial_state)
print("Desired Results: ")
for i in range(len(final_state)):
    print("probability of " + str(bin(i))[2:] + ": " + str(abs(final_state[i])**2))
```

We print the probabilities (the norm squared of each coefficient) because we can only test the correctness of our simulation by comparing the output frequencies of each possibility to the theoretical/desired probabilities – this is because e_iHt is a matrix with complex entries, so we cannot create a complex output from our simulation if we just know the norm squared/the frequencies.

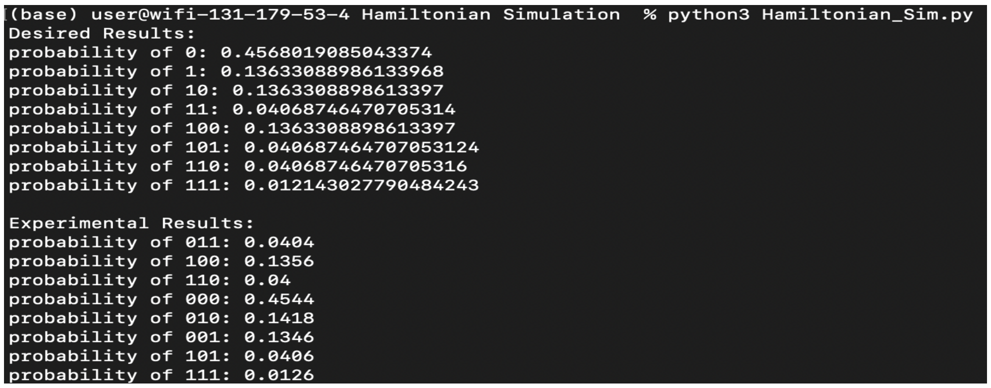
Now, our test bench for our simulation is below:


```python
#Test-bench:
backend = Aer.get_backend("aer_simulator") #accurate simualtor using no noise
bit_str_dict = {}
iters = 5000
for i in range(iters):
    job = backend.run(H_sim.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)
    bit_str = output
    if (bit_str in bit_str_dict):
      bit_str_dict[bit_str] += 1
    else:
      bit_str_dict[bit_str] = 1

print("Experimental Results: ")
for key in bit_str_dict:
    print("probability of " + str(key) + ": " + str(bit_str_dict[key]/iters))
```

What we do is get the count of each possibility and divide by the total number of iterations to get the frequency of each output; this gives us an estimate of the probability of each outcome, and we can compare this estimate to the theoretical probabilities determined from classically applying the e_iHt matrix to our initial_state vector.

Now, we are ready to run a few test cases. We can first do a test of inital_state =

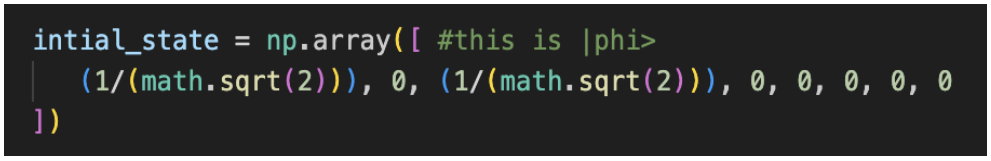
$[1\ 0\ 0\ 0\ 0\ 0\ 0\ 0]^T$. In quantum, this is the qubit state $|000\rangle$. So, we can run the circuit we currently have and see our output (because we $|000\rangle$ is the default prepared input). We have that:



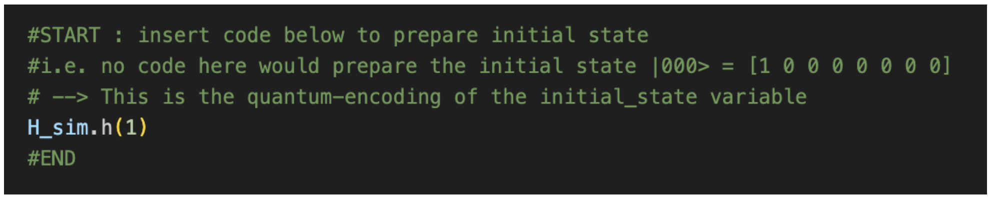
```
(base) user@wifi-131-179-53-4 Hamiltonian Simulation  % python3 Hamiltonian_Sim.py
Desired Results:
probability of 0: 0.4568019085043374
probability of 1: 0.13633088986133968
probability of 10: 0.1363308898613397
probability of 11: 0.04068746470705314
probability of 100: 0.1363308898613397
probability of 101: 0.040687464707053124
probability of 110: 0.04068746470705316
probability of 111: 0.012143027790484243

Experimental Results:
probability of 011: 0.0404
probability of 100: 0.1356
probability of 110: 0.04
probability of 000: 0.4544
probability of 010: 0.1418
probability of 001: 0.1346
probability of 101: 0.0406
probability of 111: 0.0126
```

Notice that for each outcome, the experimental frequency closely matches the desired probability. Thus, this test case yielded very good results.

Now, we can run initial_state = $[\frac{1}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ 0\ 0\ 0\ 0\ 0]^T$. In quantum, this is the state $|\varphi\rangle = \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|010\rangle = |0\rangle|+\rangle|0\rangle$. Thus, to create this state in our program, we apply Hadamard to the second qubit. We insert the following two pieces of code:

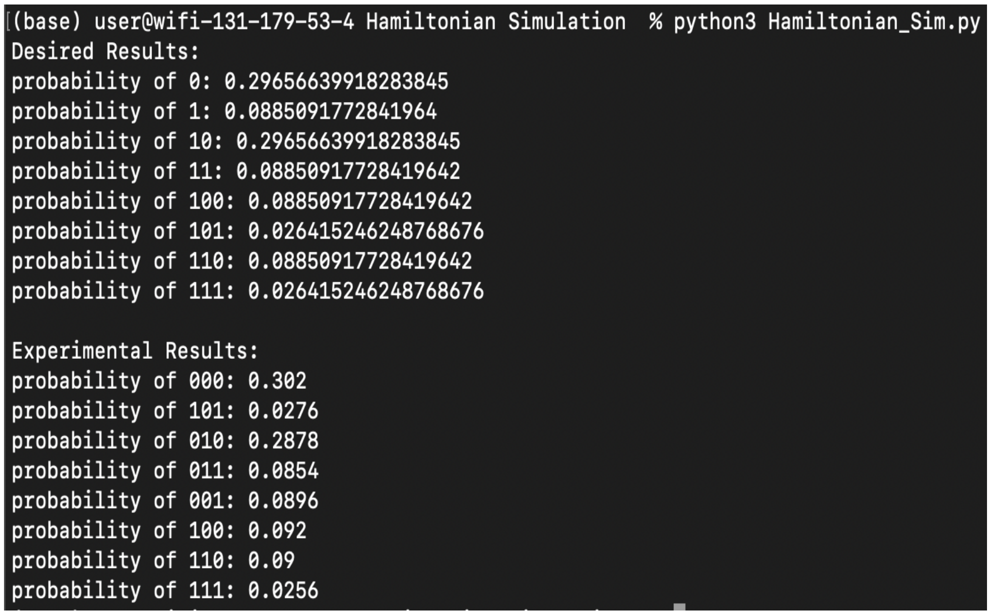


```
intial_state = np.array([ #this is |phi>
    (1/(math.sqrt(2))), 0, (1/(math.sqrt(2))), 0, 0, 0, 0, 0
])
```



```
#START : insert code below to prepare initial state
#i.e. no code here would prepare the initial state |000> = [1 0 0 0 0 0 0 0]
# --> This is the quantum-encoding of the initial_state variable
H_sim.h(1)
#END
```

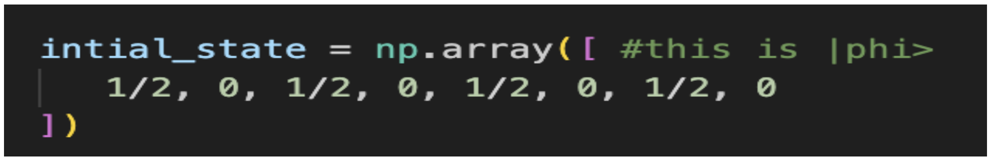
And we get the following output:



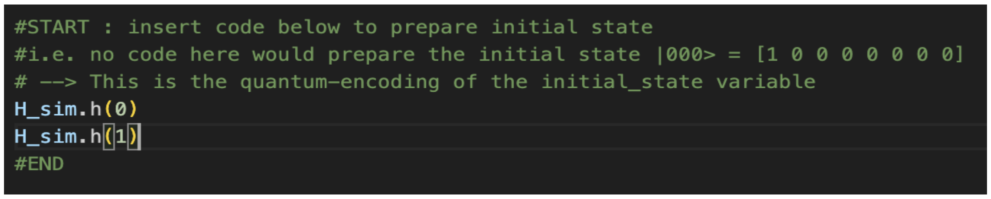
```
[(base) user@wifi-131-179-53-4 Hamiltonian Simulation  % python3 Hamiltonian_Sim.py
Desired Results:
probability of 0: 0.29656639918283845
probability of 1: 0.0885091772841964
probability of 10: 0.29656639918283845
probability of 11: 0.08850917728419642
probability of 100: 0.08850917728419642
probability of 101: 0.026415246248768676
probability of 110: 0.08850917728419642
probability of 111: 0.026415246248768676

Experimental Results:
probability of 000: 0.302
probability of 101: 0.0276
probability of 010: 0.2878
probability of 011: 0.0854
probability of 001: 0.0896
probability of 100: 0.092
probability of 110: 0.09
probability of 111: 0.0256
```

Again, we see that the frequencies of the outcomes closely match the desired probabilities, meaning this test case checks out. Thus, our simulation holds up in its correctness.

We do one more test case: initial_state $= [\frac{1}{2}\ 0\ \frac{1}{2}\ 0\ \frac{1}{2}\ 0\ \frac{1}{2}\ 0]^T$. In quantum, this is $|\varphi\rangle = \frac{1}{2}|000\rangle + \frac{1}{2}|010\rangle + \frac{1}{2}|100\rangle + \frac{1}{2}|110\rangle = |+\rangle|+\rangle|0\rangle$. Thus, to create this state in code, we apply Hadamard to the first two qubits. We this insert the following two pieces of code:

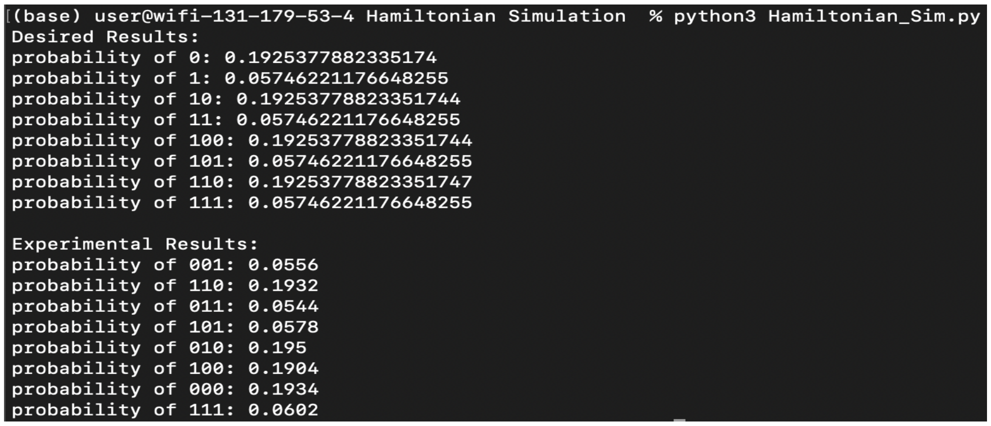


```
intial_state = np.array([ #this is |phi>
    1/2, 0, 1/2, 0, 1/2, 0, 1/2, 0
])
```



```
#START : insert code below to prepare initial state
#i.e. no code here would prepare the initial state |000> = [1 0 0 0 0 0 0 0]
# --> This is the quantum-encoding of the initial_state variable
H_sim.h(0)
H_sim.h(1)
#END
```

Running the code, we get the following output:



```
(base) user@wifi-131-179-53-4 Hamiltonian Simulation  % python3 Hamiltonian_Sim.py
Desired Results:
probability of 0: 0.1925377882335174
probability of 1: 0.05746221176648255
probability of 10: 0.19253778823351744
probability of 11: 0.05746221176648255
probability of 100: 0.19253778823351744
probability of 101: 0.05746221176648255
probability of 110: 0.19253778823351747
probability of 111: 0.05746221176648255

Experimental Results:
probability of 001: 0.0556
probability of 110: 0.1932
probability of 011: 0.0544
probability of 101: 0.0578
probability of 010: 0.195
probability of 100: 0.1904
probability of 000: 0.1934
probability of 111: 0.0602
```

We can thus see that the experimental results match the desired results again, supporting the correctness of our Hamiltonian simulator.