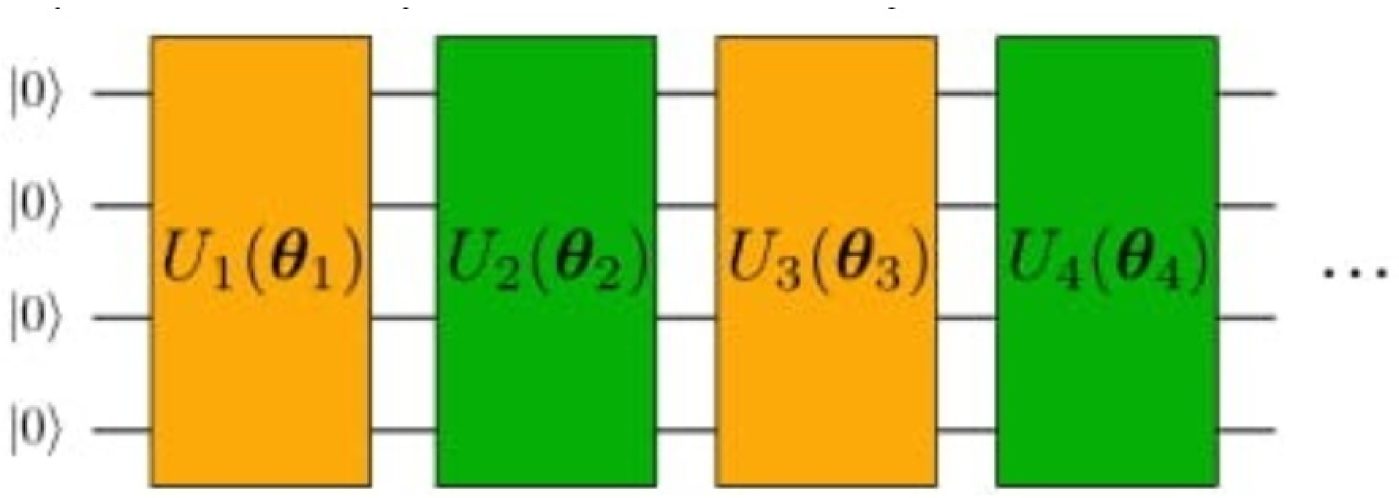
QOSF Task 1

Pamela Pajarillo

We will be implementing the following 4 qubit state $|\psi(\theta)\rangle$



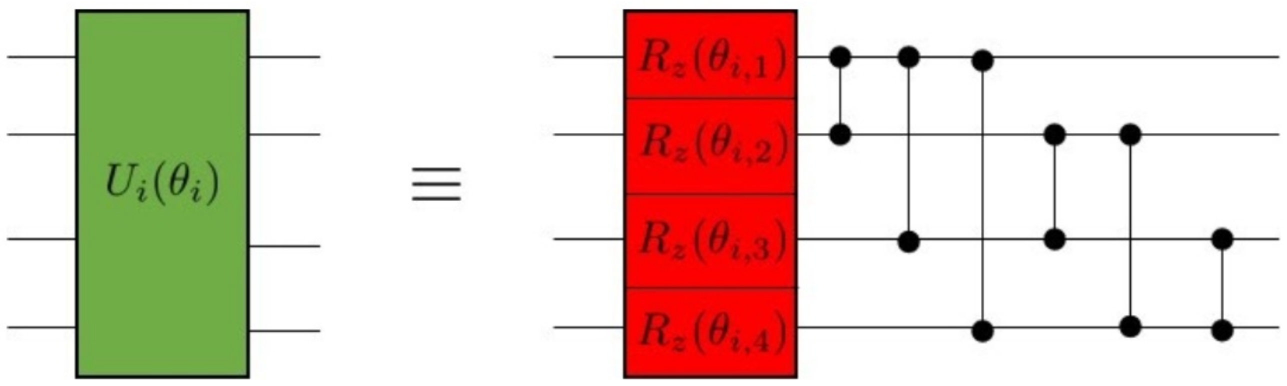
$|0\rangle$
$|0\rangle$
$|0\rangle$
$|0\rangle$
$U_1(\boldsymbol{\theta}_1)$
$U_2(\boldsymbol{\theta}_2)$
$U_3(\boldsymbol{\theta}_3)$
$U_4(\boldsymbol{\theta}_4)$
...

where the number of layers, denoted with L, is a parameter. We call "Layer" the combination of 1 yellow + 1 green block, so, for example, U1 + U2 is a layer.

In [1]:

```
# installations
!pip install --upgrade qiskit
!pip install pylatexenc

from IPython.display import clear_output
clear_output()

# import
import qiskit
import numpy as np
import random
import scipy
import matplotlib.pyplot as plt

from qiskit import QuantumCircuit
from qiskit import Aer, execute
from qiskit.circuit.random import random_circuit
from numpy import linalg as LA
from timeit import default_timer as timer

pi = np.pi
```

The even blocks, denoted in green is given by

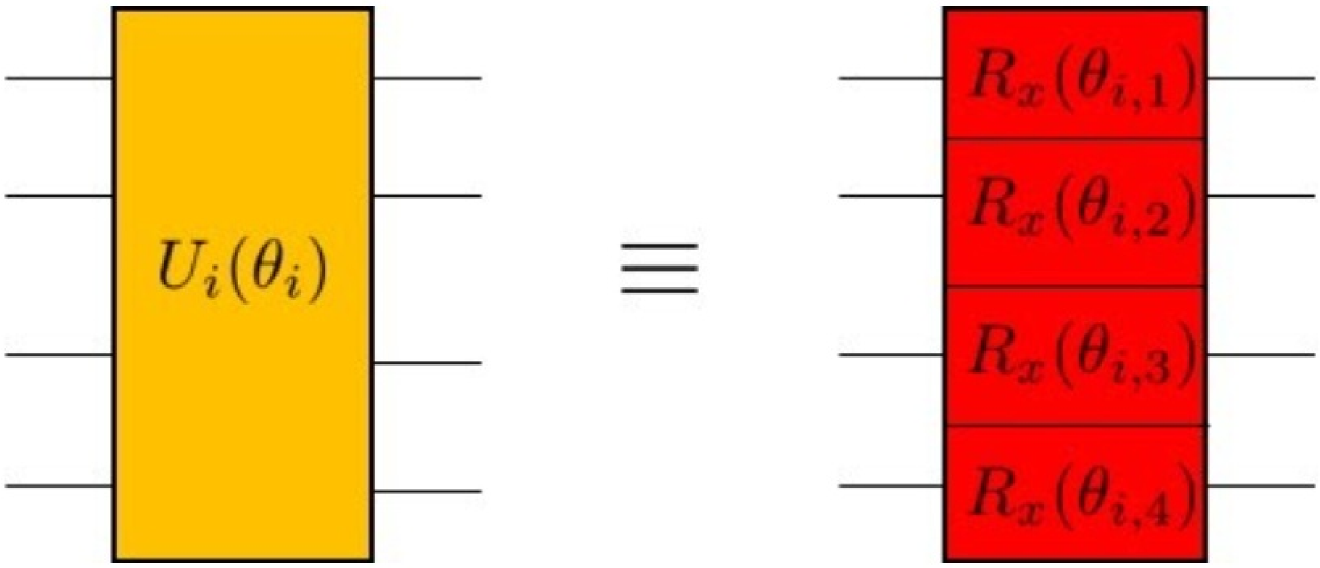


The function *green_block* implements the circuit shown above.

In [2]:

```python
# Implement Green Block - Even Block
def green_block(qcirc, rlist, index):

    '''
    Creates Green Block (Even) and appends to input circuit

    Inputs:
      qcirc - input circuit
      rzlist - list of angles for the rotation z gates
      index - index of block to place green block
    '''

    # Throw error for an odd index
    if (index % 2 != 0):
          print("Error: Must be an even index to apply a green block")
          return

    # initialize quantum circuit
    qc = QuantumCircuit(qcirc.num_qubits)

    # add rotation blocks - rz
    for i in range(qcirc.num_qubits):
        qc.rz(rlist[i], i)   # add rz gate with specified angle

    # add CZ gates
    for i in range(qcirc.num_qubits):
        for j in range(qcirc.num_qubits):
            if (j > i):
                qc.cx(i,j)    # add CX gate

    # turn above gates to green block and append to qcirc
    qc = qc.to_gate()
    qc.name = "$R_z(theta_%d)$" % (index)
    qcirc.append(qc, range(qcirc.num_qubits))
```

The odd blocks, denoted in yellow is given by



The function *yellow_block* implements the circuit shown above.

In [3]:

```python
# Implement Yellow Block - Odd
def yellow_block(qcirc, rlist, index):

    '''
    Creates Green Block (Even) and appends to input circuit

    Inputs:
      qcirc - input circuit
      rzlist - list of angles for the rotation z gates
      index - index of block to place green block
    '''

    # Throw error for an even index
    if (index % 2 != 1):
          print("Error: Must be an odd index to apply a yellow block")
          return

    # initialize quantum circuit
    qc = QuantumCircuit(qcirc.num_qubits)

    # add rotation blocks - rx
    for i in range(qcirc.num_qubits):
        qc.rx(rlist[i], i)   # add rz gate with specified angle

    # turn above gates to yellow block and append to qcirc
    qc = qc.to_gate()
    qc.name = "$R_x (theta_%d)$" % (index)
    qcirc.append(qc, range(qcirc.num_qubits))
```

The angles $\theta_{i,n}$ are variational parameters, lying in the interval $(0, 2\pi)$, which is initialized at random. The function *random_angle_parameters* outputs an array of random θs from 0 to 2π that will be used to initialize the circuit of layers of yellow and green blocks.

In [4]:

```
# Output a random angle from 0 to 2pi
def random_angle_parameters(num_qubits, num_layers):

    '''
    Creates Initial Angle Parameters Randomly

    Inputs:
      num_qubits - number of qubits
      num_layers - number of layers

    Returns:
      Output an array of random angles from 0 to 2pi based on inputs
    '''

    theta = []
    for i in range(num_qubits * num_layers * 2):
        theta.append(2 * pi * random.uniform(0,1))

    return theta
```

We will need to obtain the statevector of the circuit. By using one of Qiskit's simulations to view the state of the system, we can get the final statevector using *get_statevector* function.

In [5]:

```
# get statevector
def output_statevector(qc):

    '''
    Get statevector from the input circuit

    Inputs:
      qcirc - input circuit

    Returns:
      Statevector for the given input circuit
    '''

    # Run the quantum circuit on a statevector simulator backend
    backend = Aer.get_backend('statevector_simulator')

    # Create a Quantum Program for execution
    job = execute(qc, backend)
    result = job.result()
    outputstate = result.get_statevector(qc)

    return outputstate
```

The function *initialize_circuit* takes in an array of angles, a quantum circuit, the number of layers, and the number of qubits specified and outputs the statevector of the system, along with the modifed circuit with the layers appended.

In [6]:

```
def initialize_circuit(angles, qc, num_layers, num_qubits):

    '''
    Initializes circuit with random angles initialized

    Parameters:
      qc - input Quantum Circuit
      num_layers - number of layers
      num_qubits - number of qubits

    Returns:
      Modifies input qc with the number layers specified
      Outputs the statevector of the circuit
    '''

    # reshape array
    theta = np.reshape(angles, (num_layers * 2, num_qubits))
    for i in range(num_layers):
        # Initialize random thetas
        yellow_theta = theta[(2*i)]
        green_theta = theta[(2*i)+1]
        # append yellow and green blocks
        yellow_block(qc, yellow_theta, (2*i)+1)   # odd index
        green_block(qc, green_theta, (2*i)+2)    # even index

    # compute return the statevector
    qc_vector = output_statevector(qc)

    return qc_vector
```

Starting with One Layer

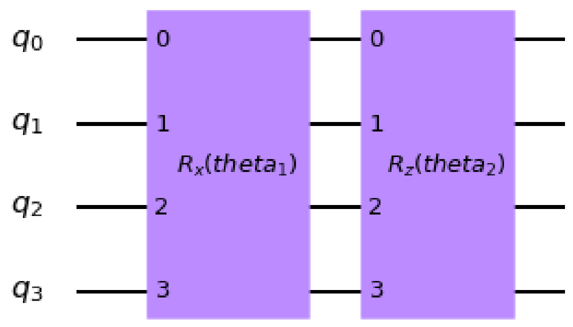
We will first start with one layer. We will create a random array of angles from 0 to 2π then initialize the circuit. We can see the amplitudes of each of the states possible for 4 qubits, shown when *qc_vector* is printed. The first element of the *qc_vector* array represents the amplitude of the $|0000\rangle$ state, the second element represents the amplitude of the $|0001\rangle$ state, etc., and the 16th element represents the amplitude of the $|1111\rangle$ state. The state of the layered circuit is denoted as $\psi(\theta)$.

In [7]:

```
# Create new circuit
num_qubits = 4
num_layers = 1

# Create array of random angles
theta_list = random_angle_parameters(num_qubits, num_layers)

# Initialize circuit with layers using random angles
qc = QuantumCircuit(num_qubits)
qc_vector = initialize_circuit(theta_list, qc, num_layers, num_qubits)
qc_vector = output_statevector(qc)
print(qc_vector)
qc.draw('mpl')
```

```
[-5.61823583e-03+0.00000000e+00j -1.21196165e-01+1.87546918e-02j
 -1.92299363e-02-1.00768724e-02j -2.55276554e-02+1.88563914e-02j
  2.71050185e-02-6.62135691e-03j  2.46025974e-02+2.12264371e-03j
  3.27994535e-03+2.88994210e-03j  1.00934295e-01-1.21057673e-01j
 -9.74490861e-02+9.28403201e-02j -4.20279437e-03-2.92276208e-03j
 -2.91027416e-04-8.58225840e-04j -1.31182332e-01+7.48906585e-01j
  1.00981592e-03+5.80316257e-04j  4.61811791e-01-3.69735872e-01j
  1.04646876e-01-4.07404740e-03j  6.53602030e-03-7.52437416e-04j]
```

Out[7]:



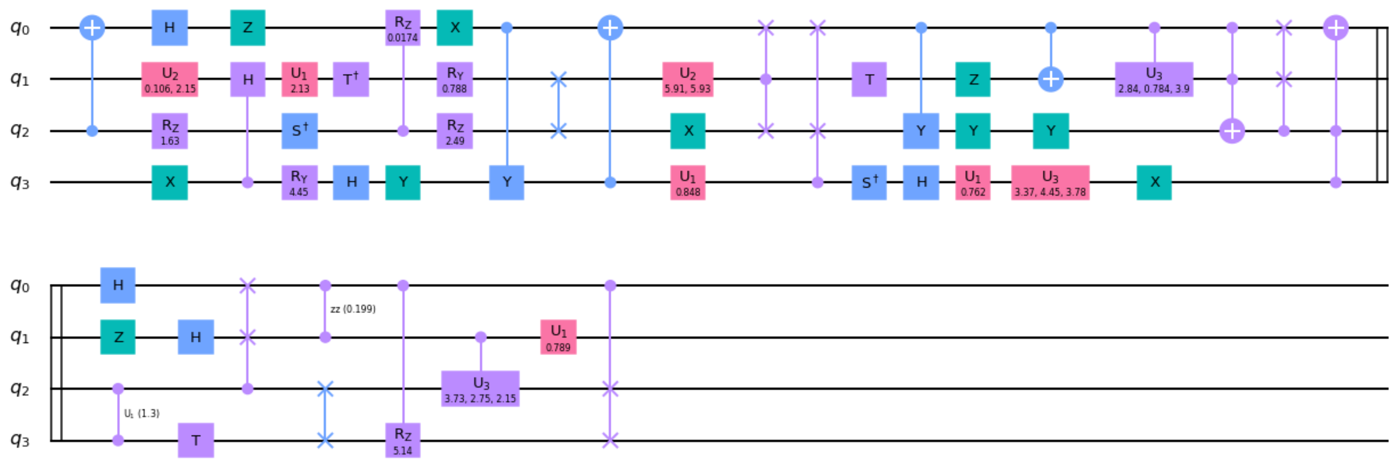
Now we want to create a random state ϕ, which will be fixed throughout the run. We can create a random circuit, using Qiskit's *random_circuit* and then obtain its statevector. The circuit of the random state is drawn below.

In [8]:

```
# Create random state
rand_circ = random_circuit(4, 20, measure=False)
rand_vector = output_statevector(rand_circ)
print(rand_vector)
rand_circ.draw('mpl')
```

```
[ 0.0732264 +0.02386672j  0.05852649+0.06010763j  0.14662777-0.27120193j
  0.22030105-0.19846538j  0.00830147-0.166037j    0.23281311+0.11687544j
 -0.12057697+0.12651978j  0.42312586+0.01307785j  0.22207563-0.1131693j
  0.1565931 -0.17964445j  0.21263294+0.01589725j -0.15346548+0.07697183j
  0.34622235+0.18938546j  0.3711526 -0.00357954j -0.00107222-0.03733494j
 -0.09874639+0.01724012j]
```

Out[8]:



We want to measure the difference between the two states. This can be done by using numpy's linear algebra function *norm*. We first subtract the amplitudes of the initial state prepared with random angles and the random state prepared using a random circuit. The distance is given as

$$\epsilon = \min_\theta$$
$$|||\psi(\theta)\rangle$$
$$- |\phi\rangle||$$

In [9]:

```
initial_epsilon = LA.norm(np.subtract(qc_vector, rand_vector))
```

Optimization of Angles

The correct set of the variational parameters $\theta_{i,n}$ need to found such that ϵ defined above is the minimum. We will be using *scipy.optimize.minimize*, a Scipy function that minimizes scalar functions of one or more variables. We first need to create an objective function to be minimized, in this case *optimize_angles*. The default method solver will be used, BFGS (quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno), which uses first derivatives.

In [10]:

```
# Optimize Variational Parameters
def optimize_angles(angles):

    '''
    Function to optimize angles

    Parameters:
      angles - array of angles

    Returns:
      Statevector for the given input circuit
    '''

    global rand_vector
    global num_qubits
    global num_layers

    qc = QuantumCircuit(num_qubits)
    qc_vector = initialize_circuit(angles, qc, num_layers, num_qubits)

    return LA.norm(np.subtract(qc_vector, rand_vector))
```

We will input the objective function *optimize_angles* and the initial angles into *scipy.optimize.minimize*. This will output an *OptimizeResult* object, which has several attributes, such as *x*, which is the the solution array, and *nfev*, which is the number of evaluations of the objective function.

In [11]:

```
# Optimize variational parameters
final_optimized_angles = scipy.optimize.minimize(optimize_angles, theta_list)
print(final_optimized_angles)
```

```
      fun: 0.8647504764029831
 hess_inv: array([[ 7.13921923,  0.41499663, -2.75530494,  1.06360888,  0.43980766,
        -6.28361441,  0.39474113,  0.06424018],
       [ 0.41499663,  4.59137032, -0.47255172,  0.16392159,  0.87104411,
        -1.61212643,  0.81413353, -1.72627313],
       [-2.75530494, -0.47255172,  6.51205504, -0.12891993,  0.19815149,
         5.14683508, -0.9229395 ,  0.22572556],
       [ 1.06360888,  0.16392159, -0.12891993,  5.3341275 ,  0.28683356,
        -2.36396988, -0.13779397,  0.17970204],
       [ 0.43980766,  0.87104411,  0.19815149,  0.28683356,  4.11583257,
        -1.4936381 , -0.69841576, -2.49610714],
       [-6.28361441, -1.61212643,  5.14683508, -2.36396988, -1.4936381 ,
        27.17612738, -1.68918233,  0.19732698],
       [ 0.39474113,  0.81413353, -0.9229395 , -0.13779397, -0.69841576,
        -1.68918233,  1.51026512,  0.10130604],
       [ 0.06424018, -1.72627313,  0.22572556,  0.17970204, -2.49610714,
         0.19732698,  0.10130604,  3.21007694]])
      jac: array([ 2.13086605e-06, -1.54227018e-06,  2.40653753e-06,  2.83122063e-07,
       -2.55554914e-06, -4.91738319e-07,  2.40653753e-06, -2.55554914e-06])
  message: 'Optimization terminated successfully.'
     nfev: 310
      nit: 30
     njev: 31
   status: 0
  success: True
        x: array([4.33133164, 0.54525004, 4.14661033, 4.46519881, 0.91107248,
       2.20637925, 2.12505258, 1.70524866])
```

```
        3.20637925, 2.12505358, 1.79524866])
```

Now that we have the solution angles from the optimizer, we can set up the circuit with the optimized angles, and compare the ϵ between the initial random angles and the optimized angles given by the optimizer.

In [12]:

```
# Extract angle from optimization
theta = np.array(final_optimized_angles.x)

# Set up circuit with optimized angles
final_qc = QuantumCircuit(num_qubits)
final_statevector = initialize_circuit(theta, final_qc, num_layers, num_qubits)
final_epsilon = LA.norm(np.subtract(final_statevector, rand_vector))

print("Initial Epsilon: " + str(initial_epsilon) + ", Final Epsilon: " + str(final_epsil
on))
```

```
Initial Epsilon: 1.2242936682982757, Final Epsilon: 0.8647504764029831
```

Increasing the Number of Layers

Now, we will iteratively increase the number of layers to see the effects on the minimum distance between $\psi(\theta)$ and ϕ.

In [13]:

```
max_layers = 7
num_qubits = 4

optimized_epsilon = []
initial_epsilon = []
times = []
num_fev = []

for i in range(1,max_layers):
    # Create new circuit
    num_layers = i
    qc = QuantumCircuit(num_qubits)
    theta_list = random_angle_parameters(num_qubits, num_layers)
    qc_vector = initialize_circuit(theta_list, qc, num_layers, num_qubits)

    # time the process of the optimization
    start = timer()
    # optimize the thetas (variational parameters)
    final_optimized_angles = scipy.optimize.minimize(optimize_angles, theta_list)
    end = timer()
    time_taken = end - start
    print("Number of Layers: " + str(i) + ", Time Taken: " + str(time_taken))

    # print(final_optimized_angles)

    # Extract the solution of theta from optimization
    theta = np.array(final_optimized_angles.x)

    # Set up circuit with optimized angles
    final_qc = QuantumCircuit(num_qubits)
    final_statevector = initialize_circuit(theta, final_qc, num_layers, num_qubits)

    # Calculate epsilon after optimization
    final_epsilon = LA.norm(np.subtract(final_statevector, rand_vector))

    # Save additional variables to to printed in plots
    initial_epsilon.append(LA.norm(np.subtract(qc_vector, rand_vector)))
    optimized_epsilon.append(final_epsilon)
    num_fev.append(final_optimized_angles.nfev)
    times.append(time_taken)
```

```
Number of Layers: 1, Time Taken: 5.657966693999981
Number of Layers: 2, Time Taken: 49.299136774999994
Number of Layers: 2, Time Taken: 49.299130774999994
Number of Layers: 3, Time Taken: 118.13956350000001
Number of Layers: 4, Time Taken: 345.72493145400006
Number of Layers: 5, Time Taken: 319.865991573
Number of Layers: 6, Time Taken: 375.10242994300006
```

Results

In [14]:

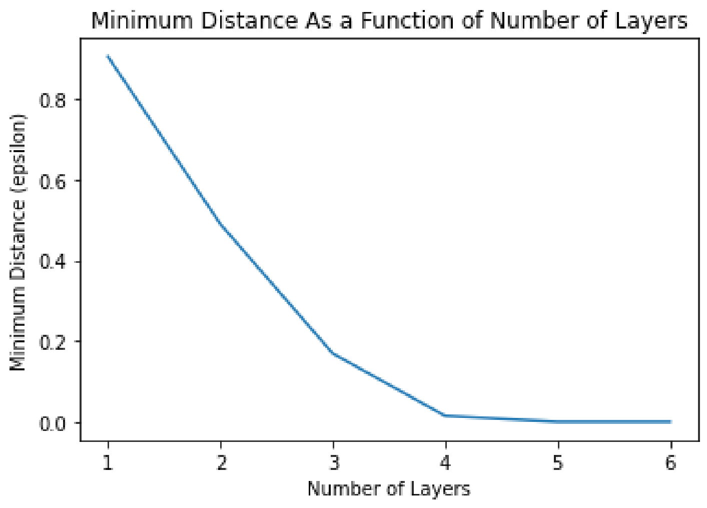
```
# Print out arrays
print("Initial epsilon: " + str(initial_epsilon))
print("Optimized epsilon: " + str(optimized_epsilon))
print("Number of function Evaluations: " + str(num_fev))
```

```
Initial epsilon: [1.4098606269258178, 1.1194220548331335, 1.4393732292312604, 1.441866752
928699, 1.6429590953999367, 1.4759326851739893]
Optimized epsilon: [0.9056359879121924, 0.4904200566193999, 0.16939272549931106, 0.014584
398107421187, 2.2290965081458256e-06, 2.8736125409152535e-06]
Number of function Evaluations: [410, 2538, 4628, 10710, 8370, 8511]
```

We can now see the effects of the additional layers on the minimum distance between $\psi(\theta)$ and ϕ.

In [15]:

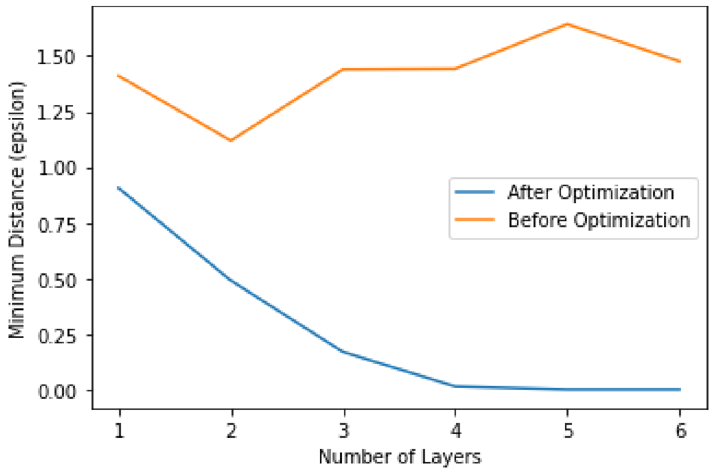
```
fig, ax1 = plt.subplots()

ax1.plot(range(1,max_layers), optimized_epsilon)
ax1.set_xlabel('Number of Layers')
ax1.set_ylabel('Minimum Distance (epsilon)')
ax1.set_title('Minimum Distance As a Function of Number of Layers')
fig.show()
```



We can see the more layers added, the mininum distance between $\psi(\theta)$ and the random state ϕ generated decreases as the number of layers increases and the distance is close to zero after 4 layers. This is because there are so many parameters to tune that it's likely for $\psi(\theta)$ and ϕ to be similar. The plot below shows the ϵ before and after optimization.

In [16]:

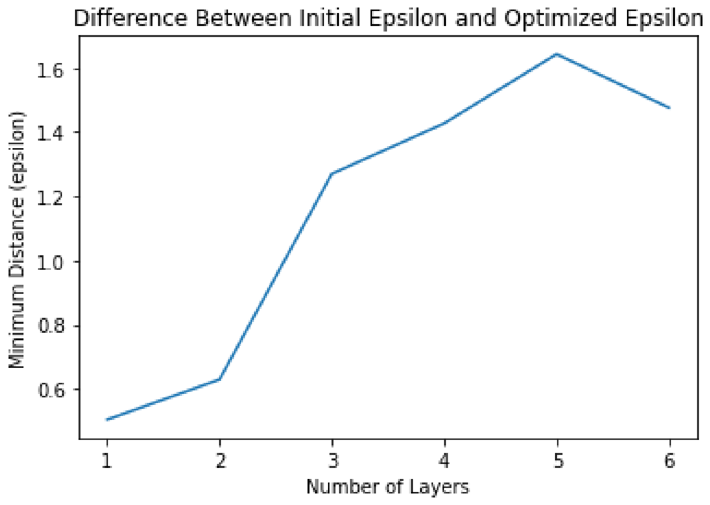
```
fig, ax1 = plt.subplots()

ax1.plot(range(1,max_layers), optimized_epsilon, label = "After Optimization")
ax1.plot(range(1,max_layers), initial_epsilon, label = "Before Optimization")
ax1.set_xlabel('Number of Layers')
ax1.set_ylabel('Minimum Distance (epsilon)')
ax1.set_title('Minimum Distance Before and After Optimization')
ax1.legend()
fig.show()
```



We can then plot the difference of ϵ before and after optimization of angles.

In [17]:

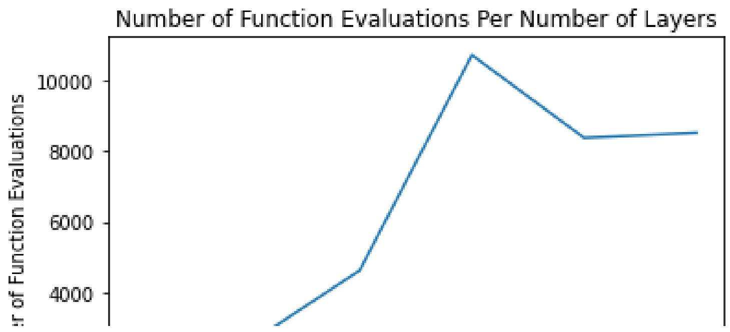
```
fig, ax1 = plt.subplots()

diff_epsilon = np.subtract(initial_epsilon, optimized_epsilon)
ax1.plot(range(1,max_layers), diff_epsilon)
ax1.set_xlabel('Number of Layers')
ax1.set_ylabel('Minimum Distance (epsilon)')
ax1.set_title('Difference Between Initial Epsilon and Optimized Epsilon')
fig.show()
```



The number of times *optimize_angles* was called during optimization is plotted below. As the number of layers increase, the number of the function *optimize_angles* evaluations increases.

In [18]:

```
fig, ax1 = plt.subplots()

ax1.plot(range(1,max_layers), num_fev)
ax1.set_xlabel('Number of Layers')
ax1.set_ylabel('Number of Function Evaluations')
ax1.set_title('Number of Function Evaluations Per Number of Layers')
fig.show()
```

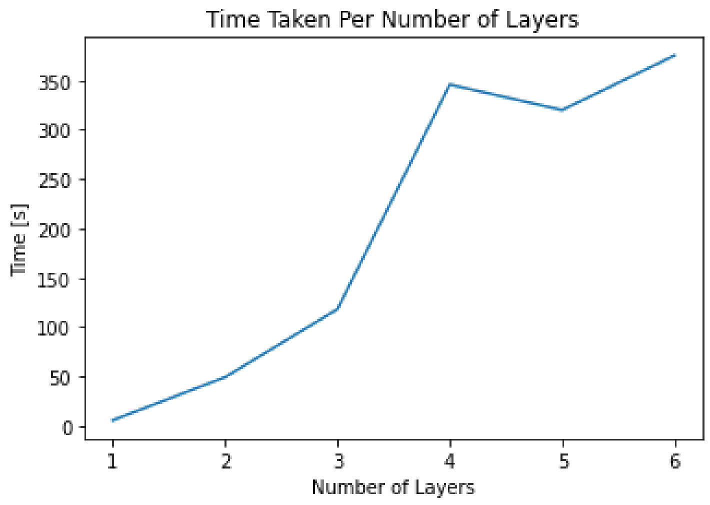


Numbe
2000
0
1
2
3
4
5
6
Number of Layers

Additionally, we can analyze the runtime for the number of layers specified. As the number of layers increase, the time it takes to optimize increases.

In [19]:

```
fig, ax1 = plt.subplots()

ax1.plot(range(1,max_layers), times)
ax1.set_xlabel('Number of Layers')
ax1.set_ylabel('Time [s]')
ax1.set_title('Time Taken Per Number of Layers')
fig.show()
```



Conclusion

We implemented the 4-qubit state, which depends on layers of one yellow and one green block. These blocks depend on variational parameters, $\theta_{i,n}$, which lie in the interval $(0, 2\pi)$. These variational parameters are initialized at random, which then need to be optimized to give the minimum distance ϵ. *Scipy.optimize.minimize* is used to give the correct $\theta_{i,n}$. The number of layers are then varied to see the effects of ϵ. We concluded that the more layers added, the mininum distance between $\psi(\theta)$ and the random state ϕ generated by Qiskit's *random_circuit* function decreases as the number of layers increases and the distance is close to zero after four layers. Four layers minimizes the distance between $\psi(\theta)$ and ϕ best as time to optimize and function evaluation calls increases with the number of layers, however, there is very little change in the minimum distance between $\psi(\theta)$ and ϕ when the number of layers increase.