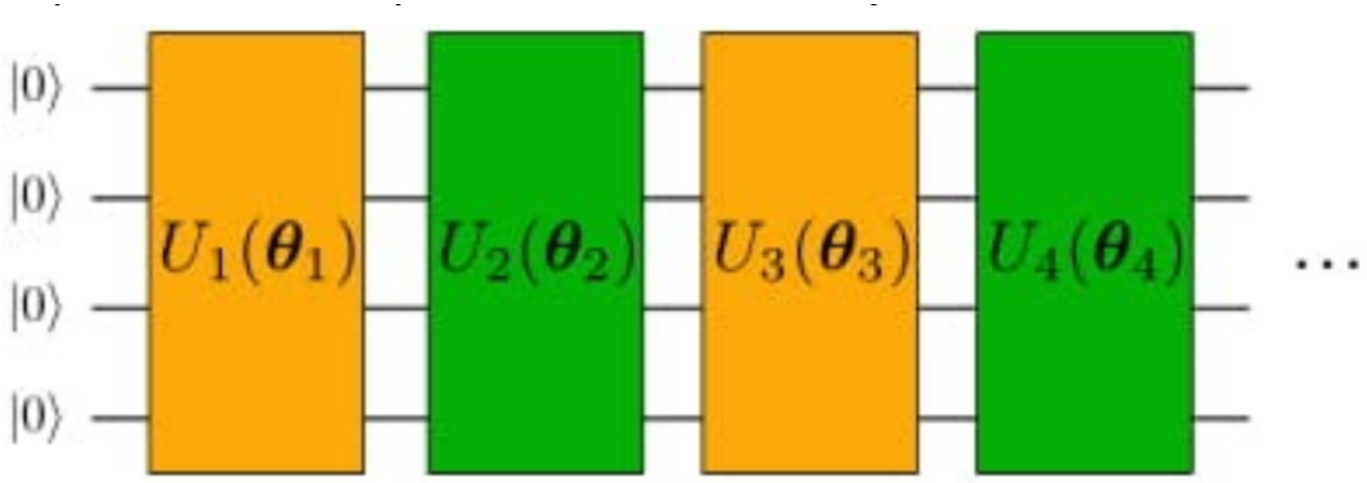
QOSF Task 1

Pamela Pajarillo

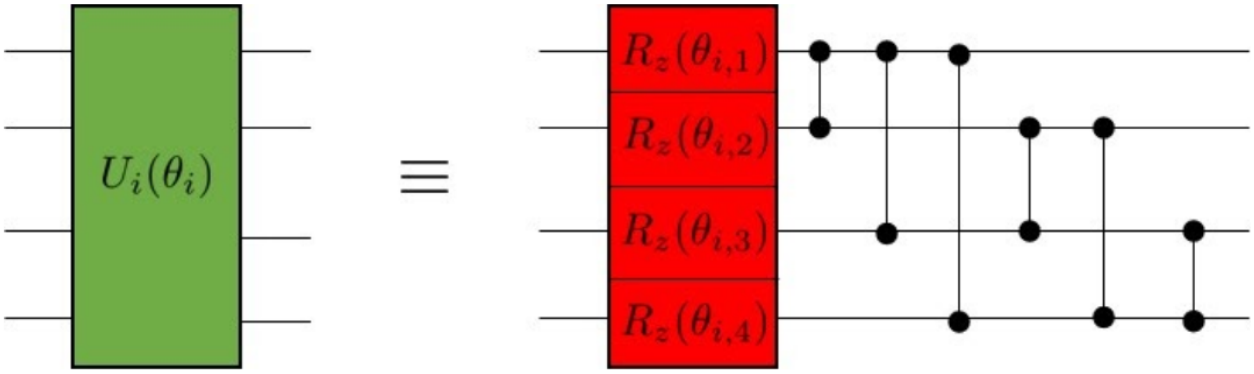
We will be implementing the following 4 qubit state $|\psi(\theta)\rangle$



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
```

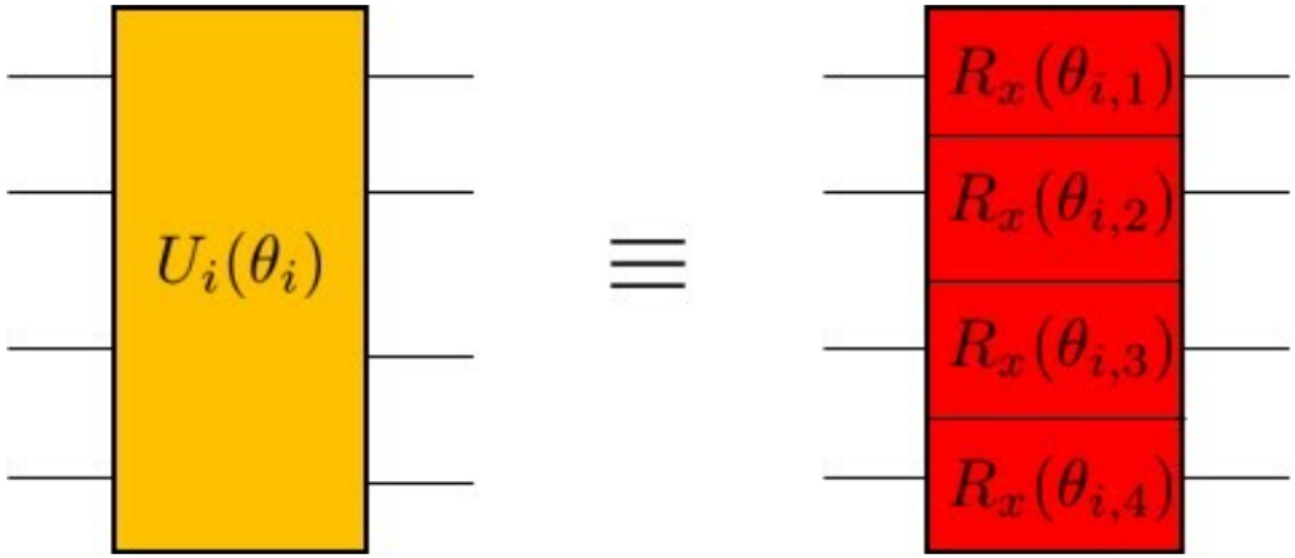
The even blocks, denoted in green is given by



The function *green_block* implements the circuit shown above.

In [2]:
```
# 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 CZ 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]:
```
# 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
```

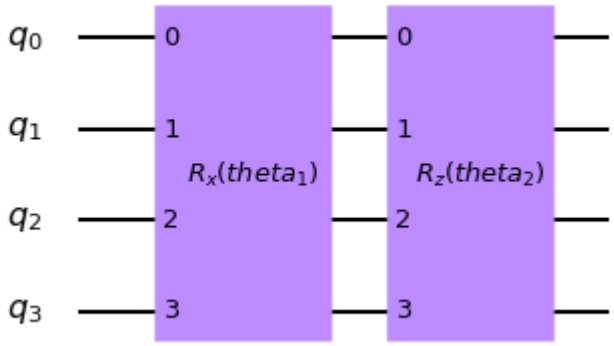
Starting with One Layer

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')
```

```
[ 1.29019047e-02+0.00000000e+00j -3.47622531e-03-1.96208772e-02j
  1.99516152e-02+2.10077564e-02j -7.40162722e-03-4.89459649e-03j
 -1.14033915e-02+1.30833721e-02j -9.29767524e-03+1.15318098e-02j
  2.02756658e-03-2.14421001e-02j  1.15053927e-02-3.17962686e-03j
  2.84357353e-01-3.12968979e-01j  3.71758526e-04-4.81077066e-04j
 -6.50394388e-05+8.81583477e-04j -2.81862568e-01+7.16688918e-02j
 -5.29423411e-04+1.09275012e-05j  7.48134831e-02+4.79699625e-01j
 -4.75445978e-01-5.21766702e-01j  3.07867755e-04+1.94578455e-04j]
```

Out[7]:

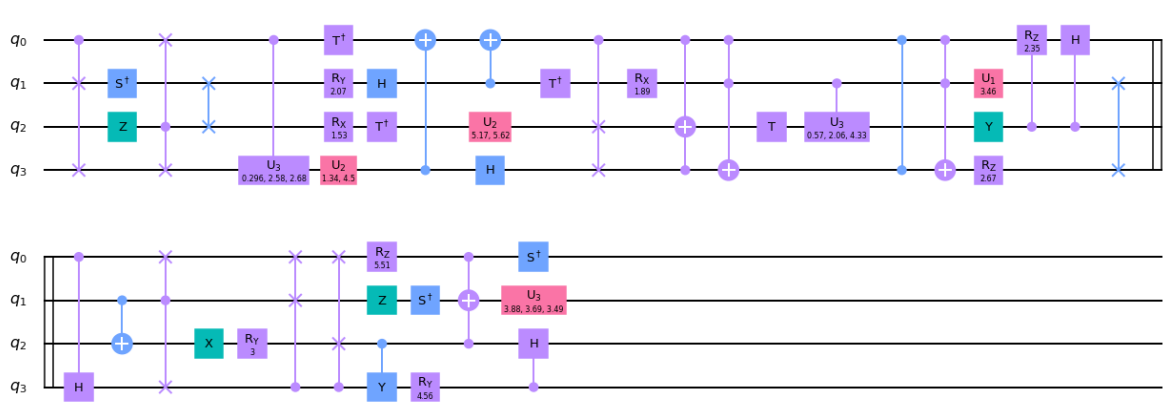


Now we want to create a random state ϕ, which will be fixed throughout the run. We can create a random circuit, using Qiskit's *random_circuit* and then obtain its statevector. The circuit of the random state is drawn below.

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')
```

```
[ 0.23555295-0.00788251j -0.06888923-0.19384045j -0.02617805-0.05038374j
  0.24034679+0.13581225j -0.14408886+0.02889191j -0.22856151-0.1458717j
  0.21712666+0.01046292j  0.11932138+0.13797767j  0.26711162+0.15217231j
 -0.05371827+0.04343178j -0.06260456-0.11377887j  0.38169878+0.32424317j
 -0.03907236+0.25801196j  0.1774657 +0.22893583j  0.11784884-0.08900835j
  0.25599729+0.20183113j]
```

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: 0.7114111514823231
 hess_inv: array([[ 4.83913047e+00, -1.29579252e+00,  5.05651681e-01,
         2.92658878e-01,  9.78109178e-01,  1.16861755e+00,
        -2.70091636e+00, -1.02667352e+00],
       [-1.29579252e+00,  5.97136374e+00, -3.08790159e+00,
        -1.77745067e+00,  4.77163648e-01, -1.21987428e+00,
         1.35444975e-01, -2.52840219e-01],
       [ 5.05651681e-01, -3.08790159e+00,  5.13448326e+00,
         3.16448861e+00, -8.26069054e-01, -2.06284608e+00,
         6.15922427e-01,  2.16185461e-01],
       [ 2.92658878e-01, -1.77745067e+00,  3.16448861e+00,
         4.27895775e+00, -1.05279592e+00, -3.18400368e+00,
         5.85746030e-02,  1.15935294e+00],
       [ 9.78109178e-01,  4.77163648e-01, -8.26069054e-01,
        -1.05279592e+00,  3.22932887e+00,  2.01676856e+00,
        -2.49716150e+00, -2.08775441e+00],
       [ 1.16861755e+00, -1.21987428e+00, -2.06284608e+00,
        -3.18400368e+00,  2.01676856e+00,  8.35259424e+01,
         6.48881140e+00, -3.71767677e+00],
       [-2.70091636e+00,  1.35444975e-01,  6.15922427e-01,
         5.85746030e-02, -2.49716150e+00,  6.48881140e+00,
         1.01785449e+01,  1.34209740e+00],
       [-1.02667352e+00, -2.52840219e-01,  2.16185461e-01,
         1.15935294e+00, -2.08775441e+00, -3.71767677e+00,
         1.34209740e+00,  2.87054706e+00]])
      jac: array([-5.66244125e-07, -2.74926424e-06, -3.33786011e-06,  3.63588
333e-06,
        1.14738941e-06,  3.12924385e-07,  4.09781933e-07, -6.25848770e-07])
  message: 'Optimization terminated successfully.'
     nfev: 330
      nit: 31
     njev: 33
   status: 0
  success: True
        x: array([1.97675271, 0.2425638 , 7.03312747, 4.3939622 , 1.72873835,
       1.1338893 , 2.40639731, 5.25690582])
```

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(f
inal_epsilon))
```

```
Initial Epsilon: 1.368089506732105, Final Epsilon: 0.7114111514823231
```

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_li
st)
    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_qu
bits)

    # 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: 5.633117334999952
Number of Layers: 2, Time Taken: 24.29353364299982
Number of Layers: 3, Time Taken: 65.02018368499989
Number of Layers: 4, Time Taken: 245.928385616
Number of Layers: 5, Time Taken: 308.4367288679998
Number of Layers: 6, Time Taken: 409.54594497799985
```

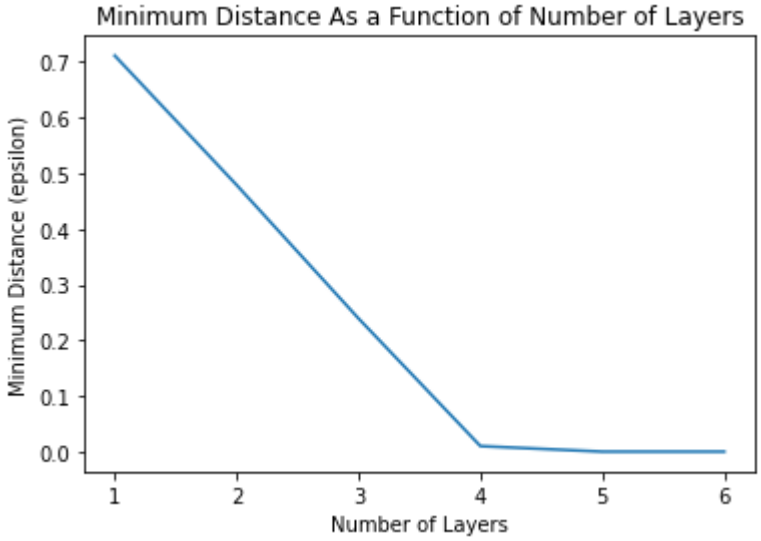
Results

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: [1.5213752901332538, 1.3492083490850204, 1.4462965735720574,
1.3646899486956119, 1.5349471900104033, 1.5339028032690574]
Optimized epsilon: [0.7114111514308868, 0.4789607485397122, 0.239245182888353
53, 0.009968645422525122, 2.1776097480940998e-06, 2.248279996765389e-06]
Number of function Evaluations: [340, 1062, 2262, 7072, 7362, 8561]
```

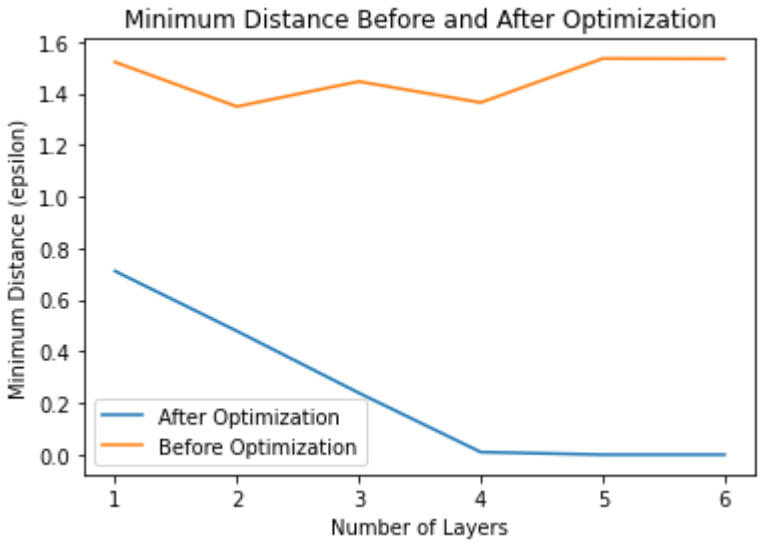
We can now see the effects of the additional layers on the minimum distance between $\psi(\theta)$ and ϕ.

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()
```



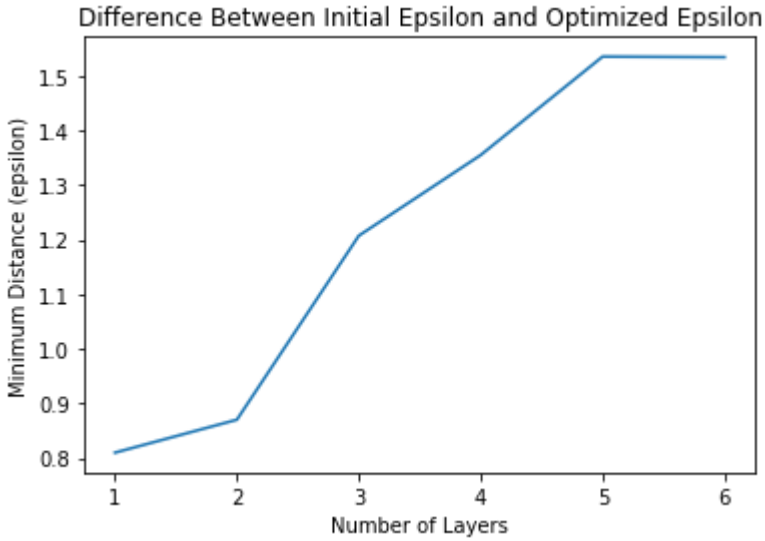
We can see the more layers added, the mininum distance between $\psi(\theta)$ and the random state ϕ generated decreases as the number of layers increases and the distance is close to zero after 4 layers. This is because there are so many parameters to tune that it's likely for $\psi(\theta)$ and ϕ to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



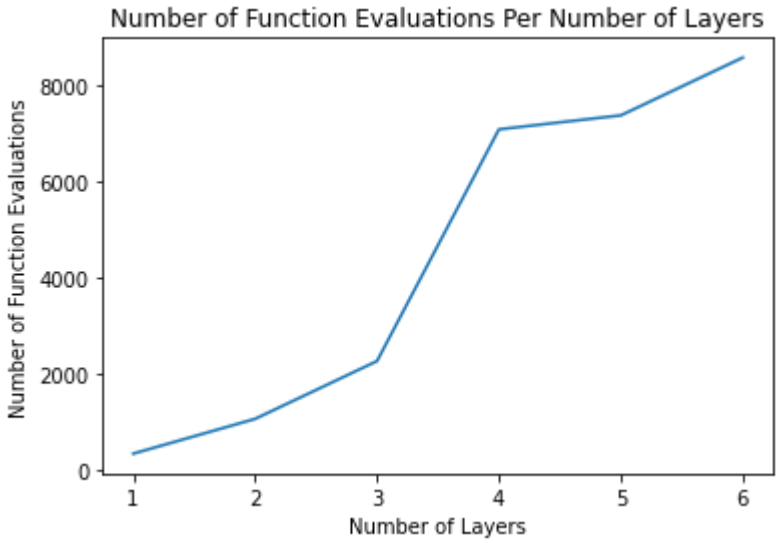
We can then plot the difference of ϵ before and after optimization of angles.

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()
```



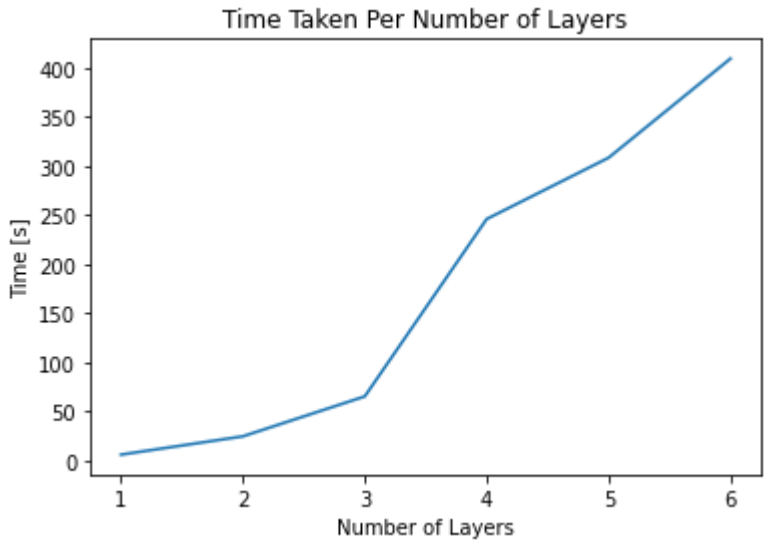
The number of times *optimize_angles* was called during optimization is plotted below. As the number of layers increase, the number of the function *optimize_angles* evaluations increases.

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()
```



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.