Task1Notebook

September 26, 2020

1 QOSF QC Mentorship Program - Task 1

Emlyn Graham For my screening task, I chose to do the first task (see the Google Doc). It seemed like a good chance to play with a QC framework and gates, given that my only exposure to QC is through quantum information theory (a course and a project) at university.

After playing around with the problem for a bit, I came to the following approach. This solution works through - Part 1: Building the circuits - Part 2: Optimising the circuits - Part 3: Having a bit of fun

where I solve the problem as stated in Parts 1 and 2, and play with the parameters of the problem in Part 3.

1.1 Part 1: Building the circuits

I chose Qiskit, for no particular reason. To start, we need to import the right things.

[1]:
```
import numpy as np
from scipy import optimize
from qiskit import QuantumCircuit, Aer, execute
import matplotlib.pyplot as plt
%matplotlib inline
np.random.seed(0)
```

In addition, we need to choose a simulator backend. For simplicity, I started with the `statevector_simulator`, which simulates circuits without measurement or noise (just the unadulterated quantum states).

[2]:
```
statevector_backend = Aer.get_backend('statevector_simulator')
```

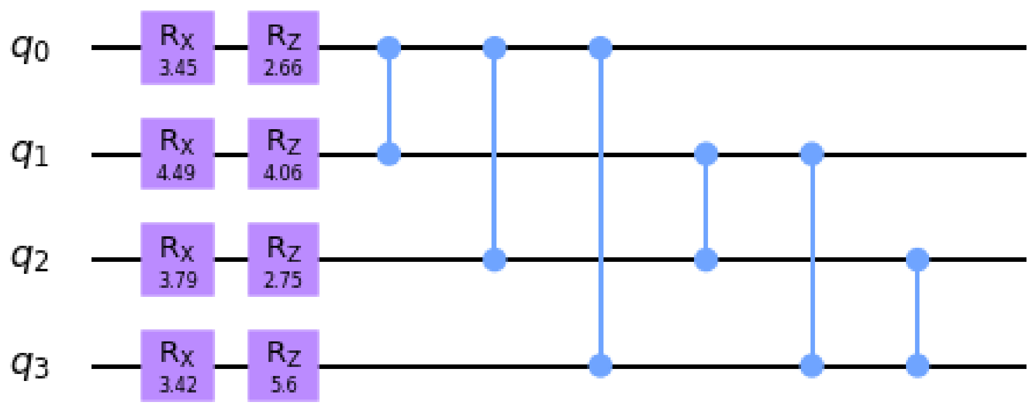
To most easily solve this problem, I settled on making two classes: one for creating and storing the specific layered circuits in the task, and another to do the optimisation. Starting with the former, which I called `layered_circuit`, I coded up the following class:

[3]:
```
class layered_circuit():
    def __init__(self, angles):
        self.angles = angles
        self.L = np.size(self.angles)//8
        self.angles = np.reshape(angles, (self.L, 8))
        self.circuit = QuantumCircuit(4)
        for i in range(self.L):
            self.add_layer(self.angles[i])

    def add_odd(self, angles):
        #Add in the x rotations
        self.circuit.rx(angles[0],0)
        self.circuit.rx(angles[1],1)
        self.circuit.rx(angles[2],2)
        self.circuit.rx(angles[3],3)
        return None

    def add_even(self, angles):
        #Add in the z rotations
        self.circuit.rz(angles[0],0)
        self.circuit.rz(angles[1],1)
        self.circuit.rz(angles[2],2)
        self.circuit.rz(angles[3],3)
        #and then add the controlled phase gates
        self.circuit.cz(0,1)
        self.circuit.cz(0,2)
        self.circuit.cz(0,3)
        self.circuit.cz(1,2)
        self.circuit.cz(1,3)
        self.circuit.cz(2,3)
        return None

    def add_layer(self, angles):
        odd_angles = angles[:4]
        even_angles = angles[4:]
        self.add_odd(odd_angles)
        self.add_even(even_angles)
        return None
```

This class takes a flat array of angles, with eight needed per layer (as defined in the task). It helps to actually see what the circuit looks like, so using random inputs we can create and draw a couple using the inbuilt functionality of Qiskit:

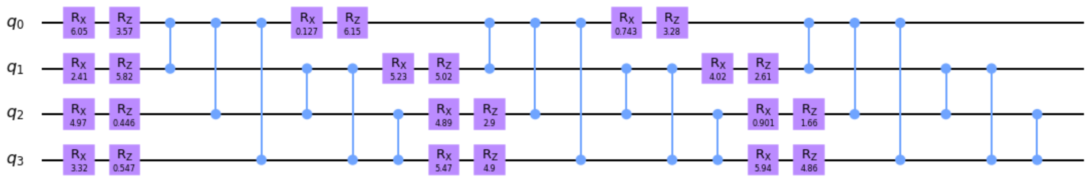
[4]:
```
single_layer_angles = 2*np.pi*np.random.rand(1,8).flatten()
single_layer_circuit = layered_circuit(single_layer_angles)
single_layer_circuit.circuit.draw('mpl')
```

[4]:



```
[5]: layered_circuit(2*np.pi*np.random.rand(3,8).flatten()).circuit.draw('mpl')
```

[5]:



Here we can see circuits with $L = 1$ and $L = 3$ respectively, with each angle randomly generated in the interval $(0, 2\pi)$.

To solve this task, these angles need to be tuned to get the circuit to output a statevector as close as possible to a predetermined (random) statevector. To do this, we need optimisation!

2 Part 2: Optimising the circuits

To do the required optimisation, I decided to create another class, layered_circuit_optimiser, that uses the previously made layered_circuit. It starts with randomly generated angles for the gates, which are restricted to the interval $(0, 2\pi)$ as defined in the task guidelines. I decided on the *COBYLA* (Constrained Optimisation BY Linear Approximation) method for scipy.optimize.minimize, which doesn't require any information about the cost function derivatives. See the scipy docs page for more information.

```
[6]: class layered_circuit_optimiser():
         def __init__(self, random_vector, L, backend, use_constraints=False,␣
      ↪tol=10**(-4), maxiter=10000, verbose=True):
        self.L = L
        self.random_vector = random_vector
        # Can use this option to force the angles
        self.use_constraints = use_constraints
        if self.use_constraints == True:
            self.constraints = []
            for element in range(self.L*8):
                l = {'type': 'ineq',
                     'fun': lambda x, lb=0.0, i=element: x[i] - lb}
                u = {'type': 'ineq',
                     'fun': lambda x, ub=2*np.pi, i=element: ub - x[i]}
                self.constraints.append(l)
                self.constraints.append(u)
        self.tol = tol
        self.maxiter = maxiter
        self.backend = backend
        self.verbose = verbose

    def cost_function(self, angles):
        circuit = layered_circuit(angles)
        job = execute(circuit.circuit, self.backend)
        result = job.result()
        outputstate = result.get_statevector(circuit.circuit)
        return np.linalg.norm(outputstate-random_vector)

    def minimise(self):
        initial_angles = 2*np.pi*np.random.rand(self.L,8).flatten()
        if self.use_constraints == True:
            result = optimize.minimize(self.cost_function, initial_angles,␣
↪method='COBYLA', constraints=self.constraints, tol=self.tol,␣
↪options={'maxiter': self.maxiter, 'disp': False})
            if self.verbose:
                if result['success'] == False:
                    print("L =",self.L, result['message'])
            return result
        else:
            result = optimize.minimize(self.cost_function, initial_angles,␣
↪method='COBYLA', tol=self.tol, options={'maxiter': self.maxiter, 'disp':␣
↪False})
            if self.verbose:
                if result['success'] == False:
                    print("L =",self.L, result['message'])
            return result
```

As required in the task guidelines, we need to pick a random statevector to minimise against. Since there are 4 qubits, this statevector is of size 16, and needs to be normalised.

[7]:
```
random_vector = np.random.random(16) + np.random.random(16) * 1j
random_vector = random_vector/np.linalg.norm(random_vector)
```

Now that everything is set up, we just need to run the optimiser for various values of L, and see what the behaviour is. To save on the increased computing time, I limited L to run from $L = 0$ to $L = 10$, with a tolerance of 10^{-6} and a limit on the optimiser iteration steps of 10000.

> You will see that even this isn't enough for the algorithm to converge to within tolerance in many cases, but since I am doing on a very old laptop away from my main computer, I didn't want to wait around for too long.

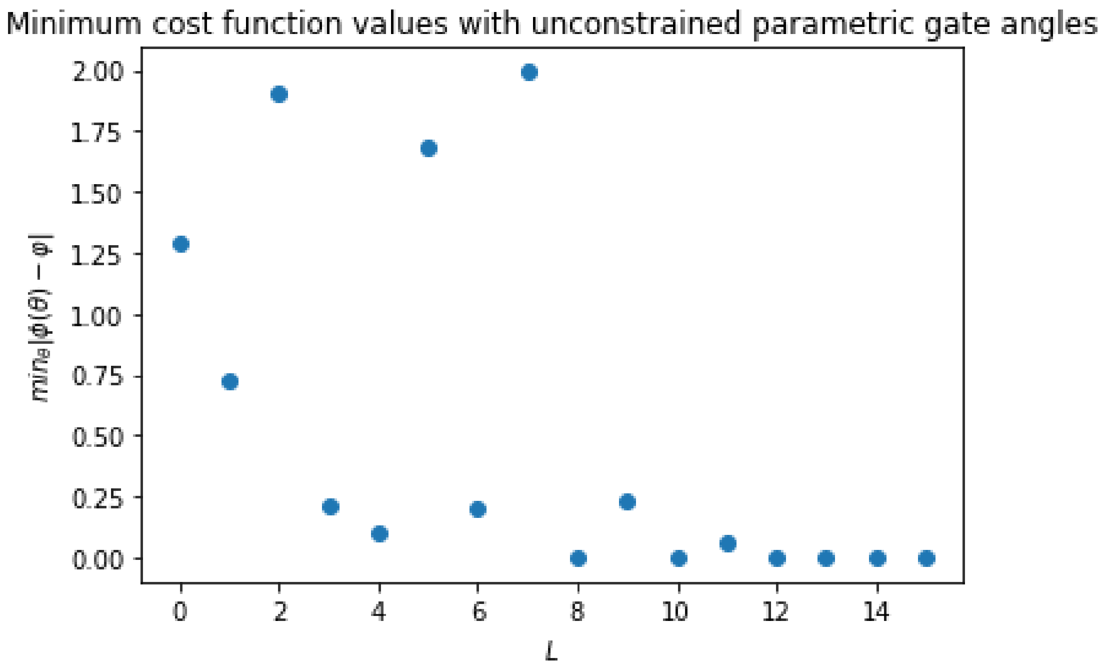
[8]:
```
n_layers = 15
tolerance = 10**(-6)
maxiterations = 10000
```

[9]:
```
number_of_layers = []
minimised_cost_functions = []
```

[10]:
```
for L in range(n_layers+1):
    optimiser = layered_circuit_optimiser(random_vector, L,␣
 ↪statevector_backend, use_constraints=True, tol=tolerance,␣
 ↪maxiter=maxiterations)
    result = optimiser.minimise()
    number_of_layers.append(L)
    minimised_cost_functions.append(result['fun'])
```

```
L = 3 Maximum number of function evaluations has been exceeded.
L = 4 Maximum number of function evaluations has been exceeded.
L = 8 Maximum number of function evaluations has been exceeded.
L = 10 Maximum number of function evaluations has been exceeded.
L = 11 Maximum number of function evaluations has been exceeded.
L = 12 Maximum number of function evaluations has been exceeded.
L = 13 Maximum number of function evaluations has been exceeded.
```

[11]:
```
fig, ax = plt.subplots()
ax.scatter(number_of_layers, minimised_cost_functions)
ax.set_ylabel(r'$min_{\theta}\|\phi(\theta) - \varphi\|$')
ax.set_xlabel(r'$L$')
ax.set_title("Minimum cost function values with unconstrained parametric gate␣
 ↪angles")
plt.show()
```



As you would expect, this shows that more layers (and thus more parametric gates) gives more ability to adjust the four qubit state, and thus the optimiser can get it closer to the given random statevector. However, it does seem that this is being limited in some way for fewer layers (of odd number). We can examine this further by playing around with the conditions of the problem.

2.1 Part 3: Having a bit of fun

2.1.1 Removing the angular constraint

The first restriction in this problem that interested me was the range of angles, $(0, 2\pi)$, particularly given how difficult it is for the solver to minimise the cost function with this restriction. It is an odd restriction, given that the Bloch sphere (single qubit) has 4π symmetry, not 2π. This can be demonstrated by taking some sets of input angles modulo 2π and 4π, and comparing the cost functions, as shown below.

[12]:
```
test_angles = np.array([4.58720534,  5.12976832,  4.35786745,  6.0319927 ,  0.
↪83087618,
        2.95726568,  1.35676681,  6.61611607,  1.47745495,  5.15871066,
        6.58166545,  4.43165288,  3.38237775, -0.35100602,  3.34341531,
        5.09094119,  4.70655196,  3.63604573,  0.36776149,  1.50521657,
        3.33197588,  5.50577004,  4.30233816,  3.28400418,  1.69197184,
        4.16878838,  5.49833625,  4.8553416 ,  5.21890076,  0.41695165,
        1.02446935,  2.19507073,  4.7542675 ,  2.68182992,  5.48189534,
        0.76287412,  6.33515418,  3.72404662,  2.41016765,  1.96144032])
```

```
base_circuit = layered_circuit(test_angles)
base_job = execute(base_circuit.circuit, statevector_backend)
mod2pi_circuit = layered_circuit(test_angles%(2*np.pi))
mod2pi_job = execute(mod2pi_circuit.circuit, statevector_backend)
mod4pi_circuit = layered_circuit(test_angles%(4*np.pi))
mod4pi_job = execute(mod4pi_circuit.circuit, statevector_backend)
print("Base cost function value: "
      , np.linalg.norm(base_job.result().get_statevector(base_circuit.circuit,␣
 ↪decimals=6)-random_vector))
print('Modulo 2 pi: '
      , np.linalg.norm(mod2pi_job.result().get_statevector(mod2pi_circuit.
 ↪circuit, decimals=6)-random_vector))
print("Modulo 4 pi: "
      , np.linalg.norm(mod4pi_job.result().get_statevector(mod4pi_circuit.
 ↪circuit, decimals=6)-random_vector))
```

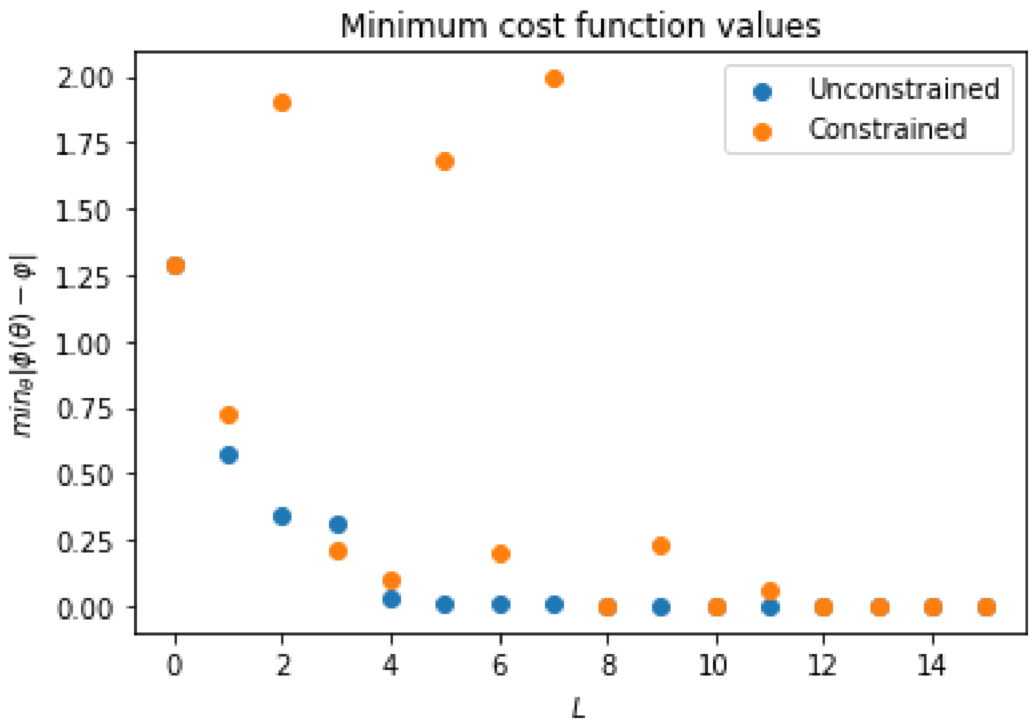
```
Base cost function value:  0.5472662324810892
Modulo 2 pi:  1.9236682850184839
Modulo 4 pi:  0.5472662324810891
```

Given that this is true, it's interesting to see whether we can get closer to the random vector by removing the constraints on the angles. We know that, whatever their final values, they are equivalent to some number in the interval $(0, 4\pi)$. This is a quick change to make, and nets the following results:

[13]:
```
minimised_cost_functions_unconstrained = []
for L in range(n_layers+1):
    optimiser = layered_circuit_optimiser(random_vector, L,␣
 ↪statevector_backend, use_constraints=False, tol=tolerance,␣
 ↪maxiter=maxiterations)
    result = optimiser.minimise()
    minimised_cost_functions_unconstrained.append(result['fun'])
fig, ax = plt.subplots()
ax.scatter(number_of_layers, minimised_cost_functions_unconstrained,␣
 ↪label="Unconstrained")
ax.scatter(number_of_layers, minimised_cost_functions, label="Constrained")
ax.set_ylabel(r'$min_{\theta}\|\phi(\theta) - \varphi\|$')
ax.set_xlabel(r'$L$')
ax.set_title("Minimum cost function values")
ax.legend()
plt.show()
```

```
L = 2 Maximum number of function evaluations has been exceeded.
L = 3 Maximum number of function evaluations has been exceeded.
L = 4 Maximum number of function evaluations has been exceeded.
L = 5 Maximum number of function evaluations has been exceeded.
L = 6 Maximum number of function evaluations has been exceeded.
L = 7 Maximum number of function evaluations has been exceeded.
```

```
L = 8 Maximum number of function evaluations has been exceeded.
L = 9 Maximum number of function evaluations has been exceeded.
L = 10 Maximum number of function evaluations has been exceeded.
L = 13 Maximum number of function evaluations has been exceeded.
```



Unconstraining the parametric gate angles seems to rid us of the poor performance of the odd-L cases. Only $L = 3$ seems to not follow the general convergence behaviour closely in this case.

2.1.2 Changing the random vector

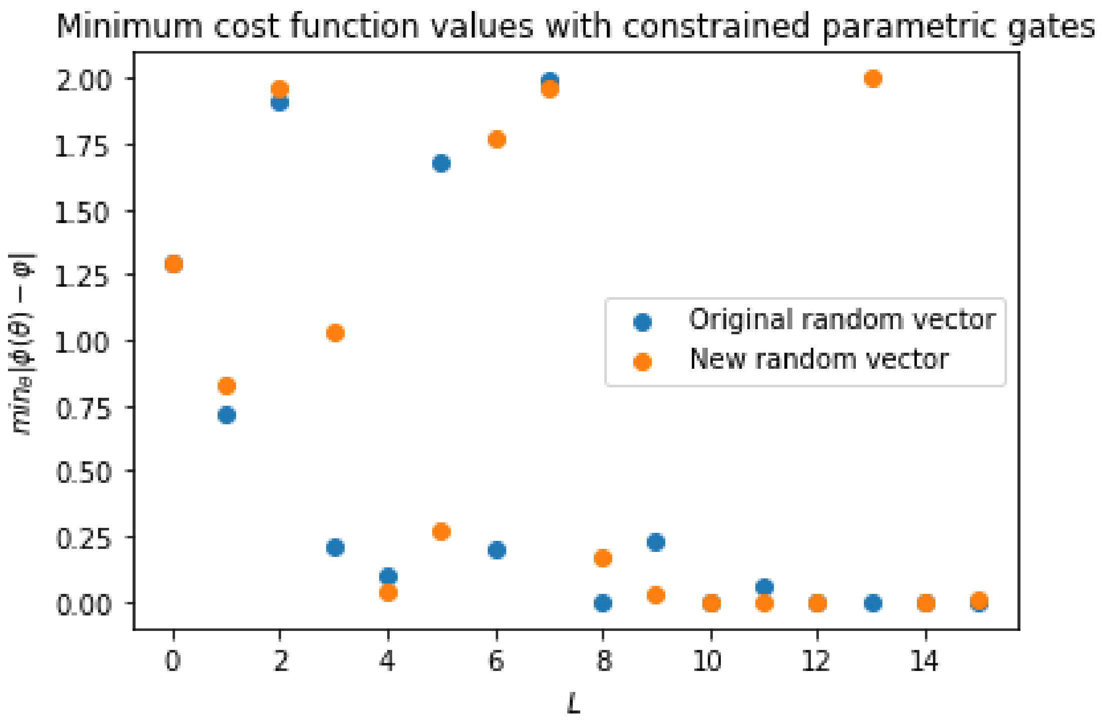
Another point of interest is whether changing the random vector we are optimising to will change how close we are able to get. To do that, you can change the seed for the random numbers, and then re-run the optimisation, comparing the results to those we have already found.

[14]:
```
np.random.seed(15)
v2 = np.random.random(16) + np.random.random(16) * 1j
v2 = v2/np.linalg.norm(v2)
```

[15]:
```
minimised_cost_functions_v2 = []
minimised_cost_functions_unconstrained_v2 = []
for L in range(n_layers+1):
    optimiser = layered_circuit_optimiser(v2, L, statevector_backend,␣
 ↪use_constraints=True, tol=tolerance, maxiter=maxiterations)
    result = optimiser.minimise()
    minimised_cost_functions_v2.append(result['fun'])
for L in range(n_layers+1):
    optimiser = layered_circuit_optimiser(v2, L, statevector_backend,␣
 ↪use_constraints=False, tol=tolerance, maxiter=maxiterations)
    result = optimiser.minimise()
    minimised_cost_functions_unconstrained_v2.append(result['fun'])
```

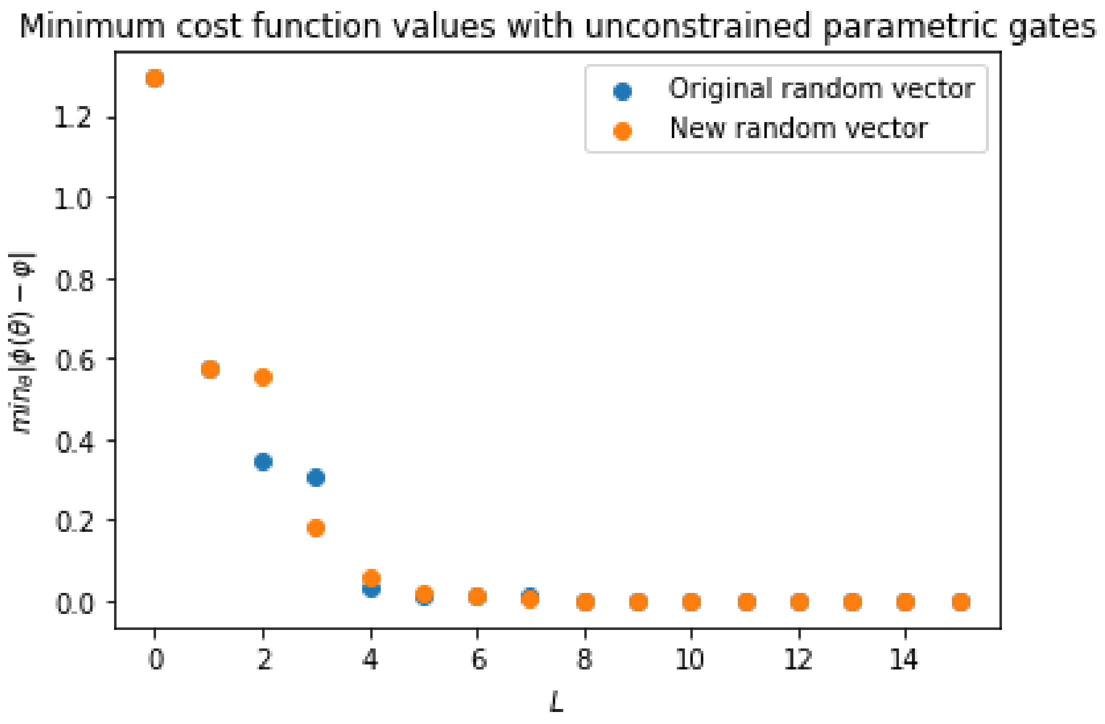
```
L = 4 Maximum number of function evaluations has been exceeded.
L = 9 Maximum number of function evaluations has been exceeded.
L = 10 Maximum number of function evaluations has been exceeded.
L = 11 Maximum number of function evaluations has been exceeded.
L = 12 Maximum number of function evaluations has been exceeded.
L = 2 Maximum number of function evaluations has been exceeded.
L = 3 Maximum number of function evaluations has been exceeded.
L = 4 Maximum number of function evaluations has been exceeded.
L = 5 Maximum number of function evaluations has been exceeded.
L = 6 Maximum number of function evaluations has been exceeded.
L = 7 Maximum number of function evaluations has been exceeded.
L = 8 Maximum number of function evaluations has been exceeded.
L = 9 Maximum number of function evaluations has been exceeded.
L = 13 Maximum number of function evaluations has been exceeded.
```

[16]:
```
fig, ax = plt.subplots()
ax.scatter(number_of_layers, minimised_cost_functions, label="Original random␣
 ↪vector")
ax.scatter(number_of_layers, minimised_cost_functions_v2, label="New random␣
 ↪vector")
ax.set_ylabel(r'$min_{\theta}\|\phi(\theta) - \varphi\|$')
ax.set_xlabel(r'$L$')
ax.set_title("Minimum cost function values with constrained parametric gates")
ax.legend()
plt.show()
```



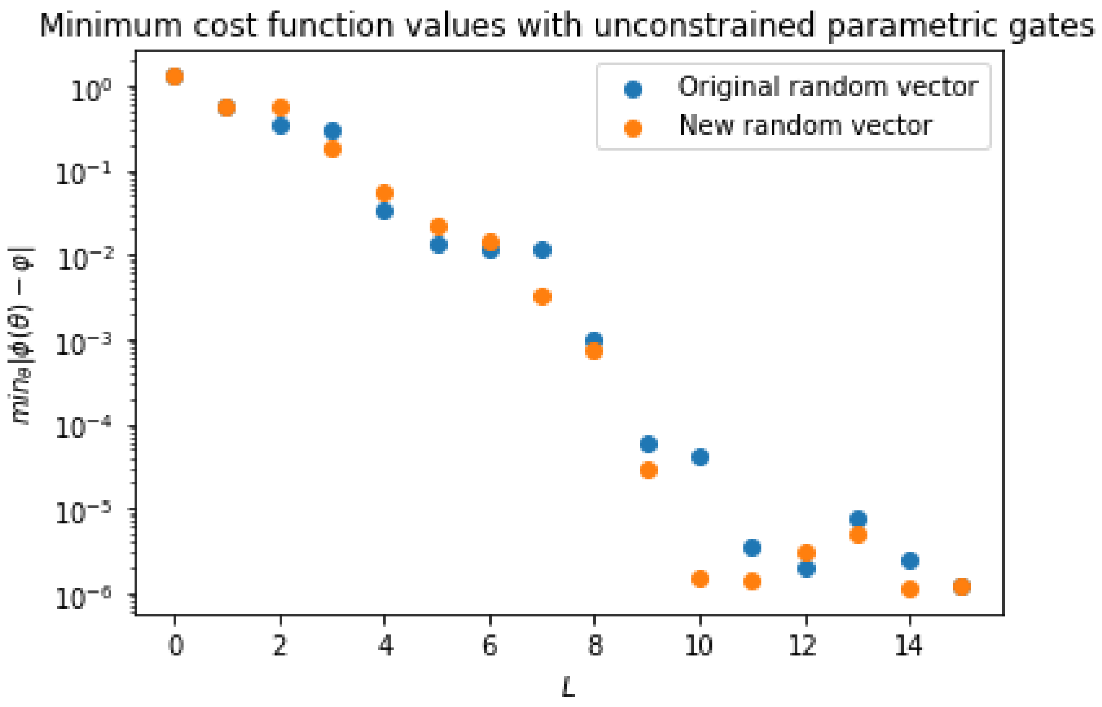
Comparing the constrained gates, we can see that the general behaviour is similar, albeit with different values at specific L.

[17]:
```
fig, ax = plt.subplots()
ax.scatter(number_of_layers, minimised_cost_functions_unconstrained,␣
 ↪label="Original random vector")
ax.scatter(number_of_layers, minimised_cost_functions_unconstrained_v2,␣
 ↪label="New random vector")
ax.set_ylabel(r'$min_{\theta}\|\phi(\theta) - \varphi\|$')
ax.set_xlabel(r'$L$')
ax.set_title("Minimum cost function values with unconstrained parametric gates")
ax.legend()
plt.show()
```



Examining the unconstrained gates, we can see the behaviour is very similar, with only some variance around $L = 2$ and $L = 3$. To see more clearly what is happening at higher L, we can plot this on a log scale.

[18]:
```
fig, ax = plt.subplots()
ax.scatter(number_of_layers, minimised_cost_functions_unconstrained,␣
 ↪label="Original random vector")
ax.scatter(number_of_layers, minimised_cost_functions_unconstrained_v2,␣
 ↪label="New random vector")
ax.set_ylabel(r'$min_{\theta}\|\phi(\theta) - \varphi\|$')
ax.set_xlabel(r'$L$')
ax.set_title("Minimum cost function values with unconstrained parametric gates")
ax.set_yscale('log')
ax.legend()
plt.show()
```



This shows that the behaviour really is very similar, with the minimised cost function values quite close at each L. This gives us evidence that the exact vector being optimised to doesn't make a whole lot of difference to the performance of the minimisation.

2.2 Conclusion and takeaways

Given this (decidedly limited) investigation, we have some indication that the more layers of $RX/RZ/CZ$ gates we have, the closer we can tune them to get a desired quantum statevector out at the end. Restricting the parameters these gates take somewhat impacts on this estimation ability, but the exact statevector being estimated doesn't make much of a difference.

Given some more time and computing power, I would look more into how this behaviour continues for higher L, and where (or if) it plateaus. I would also try to improve the rigour, by making sure each L has converged sufficiently.

I am interested in how the results change with the order of the gates, and more generally with exactly which (parametric) gates are chosen. I would also be interested to see just how much impact measurement and noise have on the ability of an optimiser to replicate a given quantum statevector. I might play around in my spare time in the coming weeks!

> Note: *Thank you for giving up your time to read this work of mine. It's been a surprisingly fun exercise, and a great way to remove the rust off both my physics knowledge and coding ability (given I use neither in my*

day job)