Ryan Dougherty - 606085269

In [ ]:
```
from qiskit import *
from qiskit.circuit import Parameter,ParameterVector, Gate
from qiskit.quantum_info import Operator
from qiskit.circuit.library import *
from qiskit.visualization import array_to_latex, plot_histogram
from qutip import hilbert_dist, Qobj, fock_dm

import random

import pyswarms as ps
import pyswarms.backend as P
from pyswarms.backend.topology import Star

import numpy as np
import matplotlib.pyplot as plt
```

Introduction

To optimize a generic 3 qubit unitary, we can build it using a set of generic rotation gates composed of 11 single qubit unitaries and 4 two qubit unitaries. This process is outlined in the paper 'Optimal Implementation of Quantum Gates with Two Controls: A Detailed Proof' by Jens Palsberg and Nengkun Yu. In this assignement we used the methods in this paper to find an approximiate solution to the Toffoli gate (CCNOT) using only 4 two qubit gates. I was able to measure how close my generated Toffoli gate was to the original by comparing it to the Hilbert Schmidt distance which is outlined later.

The Toffoli Gate and the Hilbert Schmidt Distance

Here's the native Toffoli gate from qiskit:

In [ ]:
```
qc = QuantumCircuit(3)
qc.ccx(0, 1, 2)
ccx_gate = Operator(qc).data
array_to_latex(ccx_gate)
```

Out[ ]:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Hilbert Schmidt Distance

Next steps is to to have a function to measure the 'closeness' to an approximate matrix. This can be done simply with the Hilbert Schmidt Distance. This is given by the following fromula from Jens' paper 'Approximate Quantum Computing':

$$d(U,V) = \sqrt{1 - \frac{\|Tr(U^\dagger V)\|^2}{N^2}}$$

Where $Tr(U)$ is the trace of U and $N = 2^n$. This will give us the absolute value of the Hilbert Schmidt distance (a real value), which will be eaasy to optimize around. I've defined this function below in `my_hilbert_schmidt_distance` . I've also defined an additional function `get_hilbert_shmidt_distance` which returns the 2d imaginary Hilbert Schmidt distance. This definition comes from the qutip library and I use it later to do some further optimization.

In [ ]:
```
def get_hilbert_shmidt_distance(gate1, gate2):
    return hilbert_dist(Qobj(gate1), Qobj(gate2))

def my_hilbert_schmidt_distance(gate1, gate2):
    return np.sqrt(1 - (np.abs(np.trace(np.conj(gate1).T @ gate2))**2) / (2*
```

In [ ]:
```
print("Hilbert Schmidt Distance of CCX with itself: ", get_hilbert_shmidt_di
print("My Hilbert Schmidt Distance of CCX with itself: ", my_hilbert_schmidt
```

```
Hilbert Schmidt Distance of CCX with itself:  0.0
My Hilbert Schmidt Distance of CCX with itself:  0.0
```

Building the Circuit

We can build the circuit with 11 single qubit gates which can simply be made with qiskit's UGate. This is a generic rotation matrix of three dimensions defined by:

In [ ]:
```
class SingleQubitU(Gate):
    def __init__(self, theta, phi, lam):
        super().__init__('U1', 1, [theta, phi, lam])

    def _define(self):
        qc = QuantumCircuit(1)
        qc.unitary(self.to_matrix(), [0])
        self.definition = qc

    def to_matrix(self):
        theta = float(self.params[0])
        phi = float(self.params[1])
        lam = float(self.params[2])
        return UGate(theta, phi, lam).to_matrix()
```

Next, we can define a generic two qubit unitary using RXX, RYY, and RZZ gates like so:

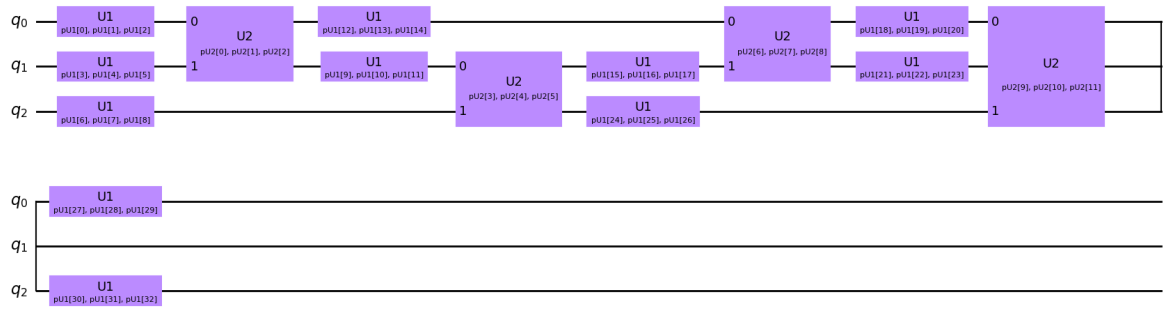
In [ ]:
```
class TwoQubitU(Gate):
    def __init__(self, alpha1, alpha2, alpha3):
        super().__init__('U2', 2, [alpha1, alpha2, alpha3])
    def _define(self):
        qc = QuantumCircuit(2)
        qc.unitary(self.to_matrix(), [0, 1])
        self.definition = qc
    def to_matrix(self):
        alpha1 = float(self.params[0])
        alpha2 = float(self.params[1])
        alpha3 = float(self.params[2])

        rxx_gate = RXXGate(alpha1).to_matrix()
        ryy_gate = RYYGate(alpha2).to_matrix()
        rzz_gate = RZZGate(alpha3).to_matrix()

        return rxx_gate @ ryy_gate @ rzz_gate
```

In [ ]:
```
class ParamVectorItterator:
    def __init__(self, param_vector):
        self.param_vector = param_vector
        self.index = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.index < len(self.param_vector):
            start = self.index
            end = self.index + 3
            params = self.param_vector[start:end]
            self.index += 3
            return params
        else:
            raise StopIteration
```

From the paper, we know its possible to build a generic gate with 4 two qubit unitaries in the order $U_{AB}U_{BC}U_{AB}U_{AC}$. We then surround the space around the 4 unitaries with 11 single qubit unitaries. The circuit is outlined below. Each Unitary has a set of 3 rotation parameters assigned to them, allowing for a total of 45 degrees of freedom.

In [ ]:
```
approx_CCNOT_qc = QuantumCircuit(3, name="Approx CCNOT")

# Define parameters
theta = Parameter('theta')
phi = Parameter('phi')
lam = Parameter('lam')

num_U1_params = 3 * 11
num_U2_params = 3 * 4

pU1 = ParameterVector('pU1', num_U1_params)
pU1_itter = ParamVectorItterator(pU1)
pU2 = ParameterVector('pU2', num_U2_params)
pU2_itter = ParamVectorItterator(pU2)

# Create the circuit
approx_CCNOT_qc.append(SingleQubitU(*next(pU1_itter)), [0])
approx_CCNOT_qc.append(SingleQubitU(*next(pU1_itter)), [1])
approx_CCNOT_qc.append(SingleQubitU(*next(pU1_itter)), [2])

approx_CCNOT_qc.append(TwoQubitU(*next(pU2_itter)), [0, 1])

approx_CCNOT_qc.append(SingleQubitU(*next(pU1_itter)), [1])

approx_CCNOT_qc.append(SingleQubitU(*next(pU1_itter)), [0])
approx_CCNOT_qc.append(TwoQubitU(*next(pU2_itter)), [1, 2])

approx_CCNOT_qc.append(SingleQubitU(*next(pU1_itter)), [1])

approx_CCNOT_qc.append(TwoQubitU(*next(pU2_itter)), [0, 1])

approx_CCNOT_qc.append(SingleQubitU(*next(pU1_itter)), [0])
approx_CCNOT_qc.append(SingleQubitU(*next(pU1_itter)), [1])
approx_CCNOT_qc.append(SingleQubitU(*next(pU1_itter)), [2])

approx_CCNOT_qc.append(TwoQubitU(*next(pU2_itter)), [0, 2])

approx_CCNOT_qc.append(SingleQubitU(*next(pU1_itter)), [0])
approx_CCNOT_qc.append(SingleQubitU(*next(pU1_itter)), [2])

approx_CCNOT_qc.draw(output='mpl')
```

Out[ ]:



Finding a Solution Using Swarm Optimization

We need to find a set of parameters that will closely approximate a Toffoli using this circuit. An algorithm I've chosen to use is Swarm Optimization. A detailed description of the algorithm can be found here: https://machinelearningmastery.com/a-gentle-introduction-to-particle-swarm-optimization/

In a nutshell, it uses a randomly distributed amount of points that query the function. They all move about the inout space and have the ability to 'talk' to eachother, where if one particle finds a global minimum, it'll tell the rest. This causes nearby particles to 'swarm' towards it. The grouped up particles will repeatedly swarm about the current global minimum until they unconver an even smaller minimum. Then the process is repeated.

In [ ]:
```
def opt_func(vals):
    distances = []
    for p in vals:
        # Get the circuit
        bound_approx_CCNOT = approx_CCNOT_qc.bind_parameters({pU1: p[:num_U1

        # Get the matrix
        approx_CCNOT_matrix = Operator(bound_approx_CCNOT).data

        # Get the distance
        distances.append(my_hilbert_schmidt_distance(ccx_gate, approx_CCNOT_

    return distances
```

Here I use the `opt_func` defined above and run swarm optimization using the python library Pyswarms. I've choesn my c1 parameter to be 0.5 and my c2 parameter to be 0.3. The algorithm contains 200 particles and is set to run for 2500 iterations. I had all the variables bounded between 0 and $2\pi and they all started at random values.

In [ ]:
```
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}
bounds = (np.repeat(0, num_U1_params + num_U2_params), np.repeat(2*np.pi, nu
optimizer = ps.single.GlobalBestPSO(n_particles=200, dimensions=num_U1_param
cost, pos = optimizer.optimize(opt_func, iters=2500)

print("Total cost: ", cost)
print("Best position: ", pos)
```

```
2023-02-23 17:31:48,913 - pyswarms.single.global_best - INFO - Optimize for
2500 iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%|██████████|2500/2500, best_cost=0.429
2023-02-23 17:52:48,688 - pyswarms.single.global_best - INFO - Optimization
finished | best cost: 0.4287530618177782, best pos: [3.2881299  3.99184176
3.18553378 3.2863268  1.47877481 5.09025655
 2.06475121 2.60927009 3.17647204 2.59969531 3.36538605 3.78667736
 2.66786559 2.8671362  1.69989823 2.57387439 1.70459726 2.55508443
 2.32916717 4.76582611 2.1363075  3.41773472 1.99808937 2.85485426
 3.51766326 2.50742741 3.89748721 2.82860762 4.14244896 3.91760327
 1.92736293 2.2492523  3.38994099 1.57787831 1.57906084 4.01130051
 4.89219643 1.63048575 4.10654436 4.73169744 1.55157382 3.99735117
 4.76154822 1.95726347 4.46895444]
Total cost:  0.4287530618177782
Best position:  [3.2881299  3.99184176 3.18553378 3.2863268  1.47877481 5.0
9025655
 2.06475121 2.60927009 3.17647204 2.59969531 3.36538605 3.78667736
 2.66786559 2.8671362  1.69989823 2.57387439 1.70459726 2.55508443
 2.32916717 4.76582611 2.1363075  3.41773472 1.99808937 2.85485426
 3.51766326 2.50742741 3.89748721 2.82860762 4.14244896 3.91760327
 1.92736293 2.2492523  3.38994099 1.57787831 1.57906084 4.01130051
 4.89219643 1.63048575 4.10654436 4.73169744 1.55157382 3.99735117
 4.76154822 1.95726347 4.46895444]
```

In [ ]:
```
# Saving the best position thus far
best_pos_thus_far = pos
```

Results

As we can see, the swarm optimization did quite well. I was able to achieve a HS distance of 0.428 - which is relatively close to the accepted minimum of ≈ 0.38. Let's take a look at the output matrix.

In [ ]:
```
def pos_to_matrix(pos):
    bound_approx_CCNOT = approx_CCNOT_qc.bind_parameters({pU1: pos[:num_U1_p
    approx_CCNOT_matrix = Operator(bound_approx_CCNOT).data
    return approx_CCNOT_matrix
```

In [ ]:
```
array_to_latex(pos_to_matrix(best_pos_thus_far))
```

Out[ ]:

$$\begin{bmatrix}
0.76866 - 0.15965i & 0.15426 + 0.31358i & -0.09028 - 0.18448i & 0.08586 - 0.1 \\
-0.2417 + 0.20675i & 0.4901 + 0.12602i & 0.59797 - 0.12651i & 0.09808 + 0.0 \\
0.21766 - 0.28419i & -0.42095 - 0.42157i & 0.58803 + 0.15945i & 0.11423 - 0.1 \\
-0.01731 - 0.10988i & -0.06438 + 0.01535i & -0.07918 + 0.03198i & 0.18388 + 0.6 \\
0.21389 - 0.03186i & 0.04818 + 0.33657i & 0.28136 + 0.19236i & -0.0824 + 0.0 \\
-0.04262 - 0.12944i & 0.33495 - 0.08285i & 0.05343 - 0.02007i & -0.08735 - 0. \\
0.21151 - 0.03403i & -0.06794 - 0.14012i & 0.22197 - 0.10344i & -0.326 + 0.4 \\
0.13403 - 0.11313i & 0.02965 - 0.05196i & 0.15836 - 0.03555i & 0.22175 + 0.1
\end{bmatrix}$$

As we can see, this matrix is quite good. All of the positions where we expect to see 1s are values very close to such.

Attempt at Further Optimization

This matrix is quite good. However, we could optimize it to get a bit closer.

To start, lets consider the imaginary version of the Hilbert Schmidt Distance; which is of course 2-dimentional

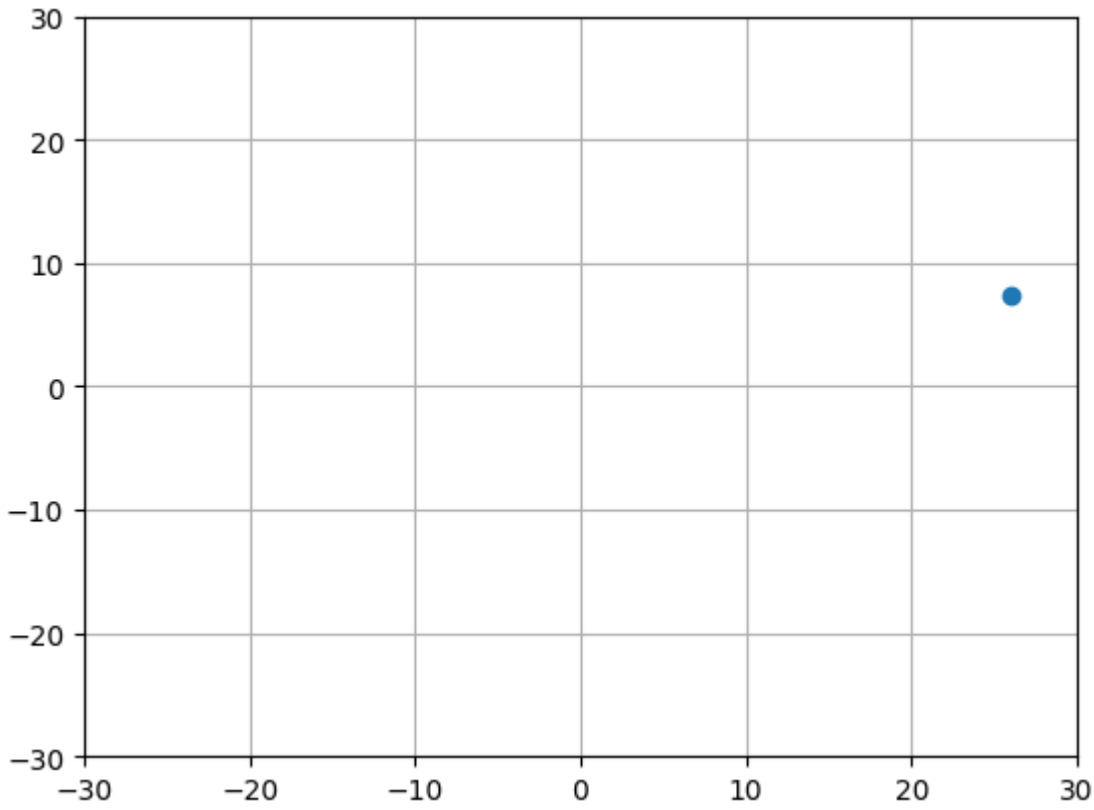
In [ ]:
```
qutip_hs_dist = get_hilbert_shmidt_distance(ccx_gate, Operator(bound_approx_
print(qutip_hs_dist)
```

```
(25.981858991915477+7.390696236864459j)
```

Now, lets plot the HS distance on a polar plot centered around zero

In [ ]:
```
plt.scatter([qutip_hs_dist.real], [qutip_hs_dist.imag])
plt.grid()
plt.xlim([-30, 30])
plt.ylim([-30, 30])
```

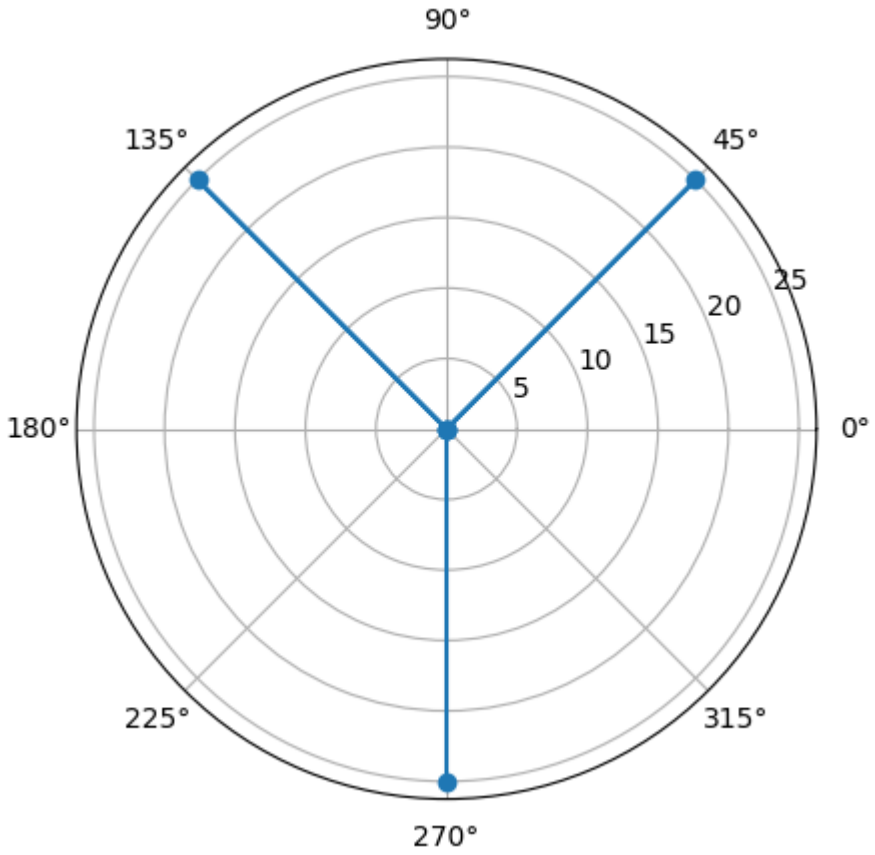
Out[ ]:
```
(-30.0, 30.0)
```



Lets choose 3 points we can optimize around. Here they are at pi/4, 3pi/4 and 3pi/2

In [ ]:
```
opt_point1 = 25 * np.exp(1j * np.pi/4)
opt_point2 = 25 * np.exp(1j * 3*np.pi/4)
opt_point3 = 25 * np.exp(1j * 3*np.pi/2)

plt.polar([0, np.angle(opt_point1), 0, np.angle(opt_point2),0,np.angle(opt_p
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f211c711990>]
```



If we can possibly optimize a set of matrices centered around *different* points. Then we can use approximate computation (similar to QUEST) by calculating an average matrix of all of them.

Lets define a new optimization function that takes in a shifted value from the HS distance. Then we perform swam optimization once more on each of these shifted distances

In [ ]:
```
def swarm(center_point, num_particles=100, num_iters=1000):
    options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}
    bounds = (np.repeat(0, num_U1_params + num_U2_params), np.repeat(2*np.pi
    optimizer = ps.single.GlobalBestPSO(n_particles=num_particles, dimension
    cost, pos = optimizer.optimize(opt_func_shift, iters=num_iters, center=c
    return cost, pos
```

In [ ]:
```
def opt_func_shift(vals, center):
    distances = []
    for p in vals:
        # Get the circuit
        bound_approx_CCNOT = approx_CCNOT_qc.bind_parameters({pU1: p[:num_U1

        # Get the matrix
        approx_CCNOT_matrix = Operator(bound_approx_CCNOT).data

        # Get the distance
        distances.append(get_hilbert_shmidt_distance(ccx_gate, approx_CCNOT_

    return  distances + center
```

In [ ]:
```
cost_pos1, pos_pos1 = swarm(opt_point1, num_particles=100, num_iters=1000)
cost_pos2, pos_pos2 = swarm(opt_point2, num_particles=100, num_iters=1000)
cost_pos3, pos_pos3 = swarm(opt_point3, num_particles=100, num_iters=1000)
```

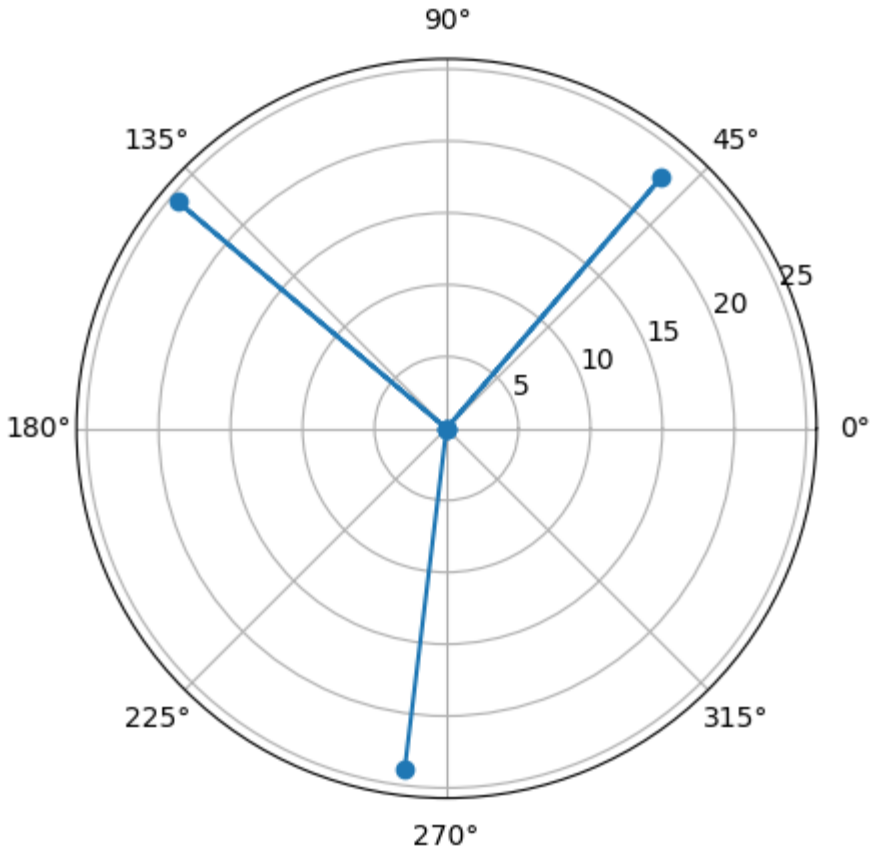
```
2023-02-24 18:51:38,870 - pyswarms.single.global_best - INFO - Optimize for
1000 iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%|██████████|1000/1000, best_cost=14.9+17.5
j
2023-02-24 19:04:25,041 - pyswarms.single.global_best - INFO - Optimization
finished | best cost: (14.947570760863105+17.49957230658889j), best pos:
[4.94852239 5.31990243 2.72016298 2.52436342 5.07355072 3.8872982
 2.90433733 1.47252178 4.87271872 1.472074   3.29548535 1.20690037
 3.28364791 4.07272721 3.05815181 5.13989982 3.56239027 2.45110943
 2.81320373 3.47888178 2.00305014 2.99351045 2.38845607 3.27279178
 3.76155527 1.73367278 3.60487945 1.57368081 3.47171195 4.63285325
 3.64088159 1.90793813 4.07857064 3.84096104 5.31429011 2.7523648
 5.35741858 2.64941561 3.77728354 3.41822824 0.33558311 2.59505671
 3.40351711 2.26134192 2.91022288]
2023-02-24 19:04:25,051 - pyswarms.single.global_best - INFO - Optimize for
1000 iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%|██████████|1000/1000, best_cost=-18.7+15.
8j
2023-02-24 19:17:08,493 - pyswarms.single.global_best - INFO - Optimization
finished | best cost: (-18.70159143943443+15.803896261701501j), best pos:
[3.65813699 2.20360965 4.96830455 0.22032852 2.21555011 1.12597795
 2.59663447 0.8376655  2.38897694 3.85048812 1.92046155 1.29820114
 3.53804372 3.06051318 3.47328148 3.41827793 1.93526526 3.49683556
 3.79325234 4.09740866 4.13758411 6.13515788 2.91750183 2.85017197
 2.30059826 3.17825267 3.44485304 3.73410987 2.17827143 3.94234458
 4.50626074 3.55246323 4.83272395 1.63899373 1.38762747 4.22771967
 1.94659978 2.78654343 3.02964671 1.91887559 4.56346941 1.55961505
 3.67840085 0.94189995 3.06790657]
2023-02-24 19:17:08,506 - pyswarms.single.global_best - INFO - Optimize for
1000 iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%|██████████|1000/1000, best_cost=-2.91-23.
7j
2023-02-24 19:30:02,637 - pyswarms.single.global_best - INFO - Optimization
finished | best cost: (-2.910532419972098-23.731059806466718j), best pos:
[3.31937089 4.08372469 4.93494652 3.10549009 3.0658602  2.51631658
 4.61207406 3.31491016 3.21046124 4.28797036 3.47631791 3.28448812
 2.85181143 2.69417498 3.8968485  3.30453134 4.35808709 3.72272129
 4.25351947 2.6790646  3.29649022 2.02026226 3.15979941 3.41766571
 1.26042811 3.30254734 1.38923825 2.27978336 3.12086983 3.69036325
 2.66367921 2.46728138 5.18724274 3.17041132 2.76535835 3.39010443
 2.95430432 2.52661284 3.84968976 3.68019831 3.33942254 3.02380828
 3.61595185 3.42158961 3.57915215]
```

In [ ]:
```
tot_cost_pos1 = cost_pos1-opt_point1
tot_cost_pos2 = cost_pos2-opt_point2
tot_cost_pos3 = cost_pos3-opt_point3
print(tot_cost_pos1)
```

```
(-2.730098768800584-0.17809722307479348j)
```

In [ ]:
```
#plt.polar([0, np.angle(cost_pos1)], [0, np.abs(cost_pos1)], marker='o')
plt.polar([0, np.angle(cost_pos1), 0, np.angle(cost_pos2),0,np.angle(cost_po
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f21175c9990>]
```



In [ ]:
```
array_to_latex(pos_to_matrix(pos_pos1))
```

Out[ ]:

$$\begin{bmatrix} 0.67054 - 0.04844i & 0.19283 - 0.06341i & -0.24934 - 0.23378i & 0.24221 - 0.0 \\ -0.40004 - 0.04437i & 0.77973 - 0.18047i & -0.12443 - 0.14944i & -0.10744 + 0. \\ 0.19655 - 0.43396i & 0.12752 - 0.27401i & 0.49702 - 0.07831i & 0.18275 + 0.0 \\ -0.11847 + 0.10084i & 0.26484 + 0.03139i & -0.04296 + 0.17028i & 0.11337 - 0. \\ -0.26075 - 0.07888i & -0.15578 + 0.12252i & -0.27248 + 0.37328i & 0.20961 - 0.0 \\ -0.02643 - 0.07i & 0.01298 + 0.25i & 0.15581 + 0.08201i & 0.07178 + 0. \\ 0.18726 + 0.04434i & 0.20628 + 0.1008i & -0.52155 + 0.17461i & -0.04911 + 0. \\ -0.13293 + 0.01902i & 0.00781 + 0.03246i & -0.01081 + 0.11897i & 0.62984 + 0.4 \end{bmatrix}$$

In [ ]:
```
array_to_latex(pos_to_matrix(pos_pos2))
```

Out[ ]:

$$\begin{bmatrix} 0.55903+0.26577i & 0.30404-0.21834i & 0.01134+0.13835i & -0.02806-0. \\ -0.42695-0.38775i & 0.22969+0.47832i & -0.02469+0.11892i & 0.04828-0.0 \\ 0.01395+0.02364i & -0.21711+0.02924i & 0.38826+0.43931i & 0.12834-0. \\ 0.0936-0.05242i & 0.08926-0.1318i & -0.0458-0.422i & 0.40988+0.4 \\ -0.37095-0.18335i & -0.07492-0.62335i & 0.15519-0.04476i & -0.03532-0. \\ -0.18978-0.24113i & 0.22541-0.22884i & -0.073+0.00269i & -0.06708-0. \\ -0.01082-0.0346i & 0.05026-0.04096i & -0.23382-0.48438i & -0.21016-0. \\ -0.01297+0.07135i & -0.0445+0.03837i & -0.03321-0.34674i & 0.14225-0. \end{bmatrix}$$

In [ ]:
```
array_to_latex(pos_to_matrix(pos_pos3))
```

Out[ ]:

$$\begin{bmatrix} 0.59526-0.47335i & -0.10777-0.20738i & 0.0361+0.26107i & 0.20596-0.1 \\ -0.03202-0.01246i & 0.49957-0.04574i & 0.14781+0.21945i & -0.06865+0. \\ 0.02362+0.29389i & -0.08189+0.14434i & 0.48692-0.54809i & 0.1847+0.0 \\ -0.13827+0.08496i & 0.01722+0.18098i & -0.09977+0.11849i & 0.06262-0. \\ 0.14571+0.48578i & -0.24578-0.12794i & 0.0598+0.41276i & 0.01237-0.1 \\ 0.09467-0.03456i & 0.17666+0.66544i & -0.00122+0.16053i & 0.24408+0.2 \\ -0.12731-0.04825i & -0.0089-0.16755i & -0.09172+0.07995i & -0.11067+0. \\ -0.12263-0.0788i & -0.22513+0.05904i & -0.26769+0.11342i & 0.28204+0.2 \end{bmatrix}$$

In [ ]:
```
average_mat = (pos_to_matrix(pos_pos1) + pos_to_matrix(pos_pos2) + pos_to_ma
array_to_latex(average_mat)
```

Out[ ]:

$$\begin{bmatrix} 0.60827-0.08534i & 0.1297-0.16305i & -0.0673+0.05521i & 0.14003-0.1 \\ -0.28634-0.14819i & 0.503+0.08404i & -0.00043+0.06297i & -0.04261+0. \\ 0.07804-0.03881i & -0.05716-0.03347i & 0.4574-0.06236i & 0.16526-0.0 \\ -0.05438+0.04446i & 0.12377+0.02686i & -0.06284-0.04441i & 0.19529-0.1 \\ -0.162+0.07452i & -0.15883-0.20959i & -0.01916+0.24709i & 0.06222-0.1 \\ -0.04052-0.11523i & 0.13835+0.22887i & 0.0272+0.08174i & 0.08293+0.1 \\ 0.01638-0.01283i & 0.08255-0.0359i & -0.28236-0.0766i & -0.12331-0. \\ -0.08951+0.00386i & -0.08727+0.04329i & -0.1039-0.03812i & 0.35137+0.1 \end{bmatrix}$$

In [ ]:
```
get_hilbert_shmidt_distance(ccx_gate, average_mat)
```

Out[ ]:
```
(0.9684075801453074-0.3784580050822781j)
```

In [ ]:
```
my_hilbert_schmidt_distance(ccx_gate, average_mat)
```

Out[ ]:
```
0.8768918012485036
```

In [ ]: