Task 3 - A Simple Quantum Compiler

In [11]:
```
import numpy as np
import math
import functools
```

Task Description:

Input assumptions: A circuit conisisting of gates to be simplified, this is represented by a list of lists of gates. The possible gates are H,X,Y,Z
Output assumptions: A circuit consisting of only CZ,RX,RZ

Description of most naive approach: The most naive approach would be to simply take every single gate in the original circuit and replace th
gate decomposition.

Description of main ideas for this construction: -Palindromic circuits of even length may be removed - it is important to note here that
RX(theta),RZ(theta),RY(theta) are not self inverse, but (-theta) or (2pi-theta) of the same gate is. Also, to include multi qubit gates in a palind
between them on all qubits they act on must be palindromic. -All identity gates can be disregarded -consecutive RX gates may be turned int
and the same is true for rz gates

Creating a gate representation

- This design of gates is working off of the assumption that all angles of rotation are limited between -pi and pi, thus any input angles to pa
  out of this range will give an error of out of bounds
- The superclass gate is the parent to both singlequbitgate and twoqubitgate from which all of the gates that match that are descended

-A circuit is expected here to be a list of lists, a list of lists of gates for each qubit index. So a circuit of n qubits is a list of length n.

-This assumes that the two qubit gates are given an indication in both the lists of gates for the qubits that they act on

-gates need the attributes: matrix, id (for checking type), angles (x,y,z), basisrepresentation (list of gates that
functions: init (setting any angle if is parametric), is_palindromic(g) which checks if a gate is its inverse
getting the gate in terms of rx, rz and cz)

jupyter Screening tasks

File Edit View Insert Cell Kernel Help

In [1?0]:
```
#here we define the classes for the gates:
#note: getmatrix is for testing purposes
class gate():
    def getmatrix(self):
        return self.matrix


    def is_palindromic(self,g):
        if self.id == g.id:
            return true

    def get_id(self):
        return self.id


    def get_basisrepresentation(self):
        return self.basisrepresentation
```

In [174]:
```
class singlequbitgate(gate):
    def get_xyzrotationangles():
        return self.angles(self)

    def is_palindromic(self,g):
        if Super().is_palindromic():
            return true
    #none of H,X,Y,Z can be represented as just one Rx or RZ gate, [illegible]
    #we also define this in Rx so that they can be removed [illegible]

class Rx(singlequbitgate):
    def __init__(self,theta):
        if theta > math.pi | theta < -math.pi:
            raise Exception() [illegible]
        self.matrix = np.array([[np.cos(theta/2), [illegible]
        self.angles = (theta,0,0)
        [illegible]
```

```
        else:
            self.basisrepresentation = [self]


    def is_palindromic(g):

        if g.get_xyzrotationangles()[2] == -self.angles[2] & g.get_xyzrotationangles()[0] == 0 & g.get_xyzrotationangles()[1] ==
            return True

        return False

class RY(singlequbitgate):

    def __init__(self,theta):
        if theta > math.pi | theta < -math.pi:
            raise Exception()
        self.matrix = np.array([[np.cos(theta/2)-(np.sin(theta/2))*1j,0],[0,np.cos(theta/2)+(np.sin(theta/2))*1j]])
        self.angles = (0,theta,0)
        self.id = ('RY')
        self.basisrepresentation = [RX(-math.pi/2),RZ(theta),RX(math.pi/2)]
        if g.get_xyzrotationangles()[1] == -self.angles[1] & g.get_xyzrotationangles()[0] == 0 & g.get_xyzrotationangles()[2] ==
            return True

class H(singlequbitgate):
    def __init__(self,theta):
        self.matrix = np.array([[np.cos(theta/2),-(np.sin(theta/2))*1j],[-(np.sin(theta/2))*1j,np.cos(theta/2)]])
        self.angles = (math.pi/2,0,math.pi/2)
        self.id = ('H')
        self.basisrepresentation = [RZ(math.pi/2),RX(math.pi/2),RZ(math.pi/2)]

class X(singlequbitgate):
    def __init__(self,theta):
        self.matrix = np.array([[np.cos(theta/2)-(np.sin(theta/2))*1j,0],[0,np.cos(theta/2)+(np.sin(theta/2))*1j]])
        self.angles = (math.pi,0,math.pi)
        self.id = ('X')
        self.basisrepresentation = [RX(-math.pi/2),RZ(math.pi),RX(math.pi/2),RZ(math.pi)]


class Y(singlequbitgate):
    def __init__(self,theta):
        self.matrix = np.array([[np.cos(theta/2),-(np.sin(theta/2))*1j],[-(np.sin(theta/2))*1j,np.cos(theta/2)]])
        self.angles = (math.pi,math.pi/2,math.pi/2)
        self.id = ('Y')
```

```
            self.basisrepresentation = [self]

    #For parametric gates, if theta for the comparison gate = -theta for this
    def is_palindromic(self,g):

        if g.get_xyzrotationangles()[0] == -self.angles[0] & g.get_xyzrotation

        return False


class RZ(singlequbitgate):
    def __init__(self,theta):
        if theta > math.pi | theta < -math.pi:
            raise Exception()
        self.matrix = np.array([[np.cos(theta/2)-(np.sin(theta/2))*1j,0],[0,np.cos
        self.angles = (0,0,theta)
        self.id = ('RZ')
        if theta == 0:
            self.basisrepresentation = []
        else:
            self.basisrepresentation = [self]


    def is_palindromic(g):

        if g.get_xyzrotationangles()[2] == -self.angles[2] & g.get_xyzrotationangles()[

        return False

class RY(singlequbitgate):

    def __init__(self,theta):
        if theta > math.pi | theta < -math.pi:
            raise Exception()
        self.matrix = np.array([[np.cos(theta/2) (np.sin(theta
        self.angles = (0,theta,0)
        self.id = ('RY')
        self.basisrepresentation = [RX(-math.pi/2),RZ(theta),R
        if g.get_xyzrotationangles()[1] == -self.angles[1] &
```

jupyter Screening Tasks Last Checkpoint: 5 minutes ago (unsaved changes)

File Edit View Insert Cell Kernel Help

Run Code

```
    def __init__(self,theta):
        self.matrix = np.array([[np.cos(theta/2)-(np.sin(theta/2))*1j,0],[0,np.cos(theta/2)+(np.sin(theta/2))*1j
        self.angles = (math.pi,0,math.pi)
        self.id = ('X')
        self.basisrepresentation = [RX(-math.pi/2),RZ(math.pi),RX(math.pi/2),RZ(math.pi)]


class Y(singlequbitgate):
    def __init__(self,theta):
        self.matrix = np.array([[np.cos(theta/2),-(np.sin(theta/2))*1j],[-(np.sin(theta/2))*1j,np.cos(theta/2)]])
        self.angles = (math.pi,math.pi/2,math.pi/2)
        self.id = ('Y')
        self.basisrepresentation = [RZ(math.pi/2),RX(-math.pi/2),RZ(math.pi),RX(math.pi/2),RZ(math.pi/2)]


class Z(singlequbitgate):
    def __init__(self,theta):
        self.matrix = np.array([[np.cos(theta/2)-(np.sin(theta/2))*1j,0],[0,np.cos(theta/2)+(np.sin(theta/2))*1j]])
        self.angles = (0,0,math.pi)
        self.id = ('Z')
        self.basisrepresentation = [RZ(math.pi)]


class I(singlequbitgate):
    def __init__(self,theta):
        self.matrix = np.array([[1,0],[0,1]])
        self.angles = (0,0,0)
        self.id = 'I'
        self.basisrepresentation = []

    def is_palindromic(self,g):
        if g.get_XYZrotationangles() == (0,0,0):
            return True
        return super().is_palindromic(g
```

Description of palindrome finding protocol

localhost:8888/notebooks/Screening%20%20Tasks.ipynb#

jupyter Screening Tasks Last Checkpoint: 5 minutes ago (unsaved changes)

File Edit View Insert Cell Kernel Help

Trusted

Run Code

Description of palindrome finding protocol

The process of moving palindromes shall be called remove_palindromes(self).

The steps of this function are defined as such: -for each qubit in the circuit, call removepalindromesfromqubit(index)

removepalindromesfromqubit is defined as: -while removelargestpalindrome() > 1: #this is when there is no palindrome left -call removelargestevenpalindrome()

removelargestevenpalindrome is defined as: -find start and end indices of largest palindrome in qubit line -set that line of

In [172]:
```
#defining the two qubit gates

class twoqubitgate(gate):
    def __init__(self,controlindex,actindex):
        self.controlindex = controlindex
        self.actindex = actindex

    def get_actindex(self):
        return self.actindex

    def get_controlindex(self):
        return self.controlindex

    def is_palindromic(self,g,qubitindexon,circuit,dist):
        if g.id == self.id & g.get_controlindex() == self.controlindex & g.get_actindex == self.actindex:
            if qubitindexon > min(self.actindex,self.controlindex): #if the index passed to it is larger than one of the
                return False

            if circuit[max(self.actindex,self.controlindex)].index(g) - circuit[max(self.actindex,self.controlindex)].index(

                return False


        return False #TODO

class CX(twoqubitgate):
    def __init__(self,controlindex,actindex):
        if controlindex == actindex:
            raise Exception
        self.controlindex = controlindex
        self.actindex = actindex
```

localhost:8888/notebooks/Screening%20%20Tasks.ipynb#

jupyter Screening Tasks Last Checkpoint: 5 minutes ago (unsaved changes)

File Edit View Insert Cell Kernel Help Trusted

Run Code

```
        self.basisrepresentation = [H.getmatrix(),CZ(controlindex,actindex),H.getmatrix]

class CZ(twoqubitgate):
    def __init__(self,controlindex,actindex):
        if controlindex == actindex:
            raise Exception
        self.controlindex = controlindex
        self.actindex = actindex
        self.id = ('CZ')
        #if self.actindex > self.controlindex:
        #    self.matrix = np.array([[1.+0.j 0.+0.j 0.+0.j 0.+0.j],[0.+0.j 1.+0.j 0.+0.j 0.+0.j],[0.+0.j 0.+0.j 0.+0.j 1.
        #else:
        #    self.matrix = np.array([[1.+0.j 0.+0.j 0.+0.j 0.+0.j],[0.+0.j 0.+0.j 0.+0.j 1.+0.j],[0.+0.j 0.+0.j 1.+0.j 0.

        self.basisrepresentation = [self]
```

In [ ]:

```
class Circuit():
    def __init__(self,numqubits):
        self.circuit = []
        for i in range(numqubits):
            self.circuit.append([])
        self.reducedcircuit = []

    def add_gate(self,qubitindex,g):
        self.circuit[qubitindex].append(g)
        #If a controlled gate, put reference in both qubits
        if g.get_id() == 'CX' | g.get_id() == 'CZ':
            if qubitindex == g.get_controlindex():
                self.circuit[g.get_actindex()].append(g)
            else:
                self.circuit[g.get_controlindex()].append(g)

    def get_compiledcircuit():

        #This step replaces all of the gates with their basis representations, if they are the I got
        for index in len(self.circuit):
            for g in self.circuit[index]:
                self.reducedcircuit[index].append(g.get_basisrepresentation())


    def removepalindromes(self):
```