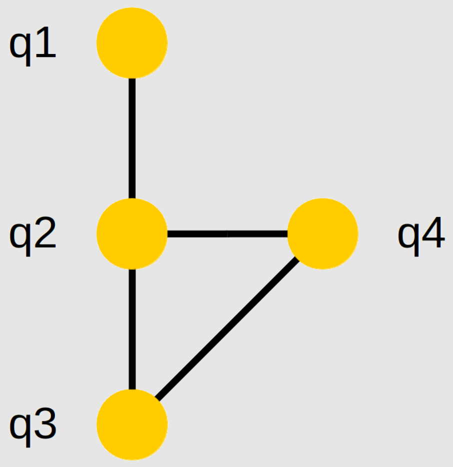
Problem - Solving the MaxCut problem with a gate-based implementation of QAOA
In this exercise you will investigate the implementation of the QAOA algorithm to solve the MaxCut problem. The main challenge here is to transform the QAOA algorithm from its mathematical form into a sequence of operations that are available in Pulser.

As mentioned in the course, there are two operating modes when dealing with neutral atom platforms, the digital (gate based) mode and the analog (Hamiltonian simulation) mode. Here we would be tempted to study this problem in an analog approach as the problem is actually an Hamiltonian simulation. Unfortunately, there is no known way to reproduce the cost Hamiltonian of the MaxCut problem without proceeding to a gate decomposition. Thus, we will be forced to proceed in digital mode.

The graph $G(V, E)$ we propose you to study is the following (triangle + edge):



First of all, copy/past the following imports in a new notebook:

```
!pip install pulser

from pulser import Sequence, Pulse, Register
from pulser.devices import Chadoq2
from pulser_simulation import Simulation
from gate_pulses import h_pulse, rx_pulse
from control_gate_sequences import cx_seq
import matplotlib.pyplot as plt
import numpy as np
```

a. **(Graph architecture)** Using the `Chadoq2` device available in Pulser, build a register that reproduces this architecture.

b. **(MaxCut)** What is the maximum-cut for this graph?

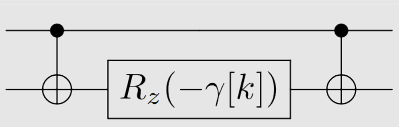
c. **(Cost Hamiltonian)** The inner loop of the algorithm first requires the application of the cost Hamiltonian evolution $e^{-i\gamma[k]H_C}$ for the k-th optimization cycle, where

$$H_C = \sum_{(i,j)\in E} \frac{1}{2}\left(I - Z_i Z_j\right).$$

Show that this evolution corresponds, for each edge of the graph, to implementing the following operation:

$$U_C(\gamma[k]) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \mathrm{e}^{-i\gamma[k]} & 0 & 0 \\ 0 & 0 & \mathrm{e}^{-i\gamma[k]} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Then verify thix matrix can be decomposed the following way:



i.e. two CX gates and a phase shift on the target qubit.

d. **(Mixer Hamiltonian)** Show that the mixer Hamiltonian evolution $\mathrm{e}^{-i\beta[k]H_M}$, where

$$H_M = \sum_i X_i,$$

corresponds, for each vertex of the graph, to implementing the following operation:

$$R_X(2\beta[k]) = \begin{bmatrix} \cos(\beta[k]) & -i\sin(\beta[k]) \\ -i\sin(\beta[k]) & \cos(\beta[k]) \end{bmatrix}$$

e. **(Implementation of QAOA)** Complete the definition of the following function that takes as inputs the $\gamma[k]$ and the $\beta[k]$ as list objects to build the QAOA sequence in Pulser:

```
def qaoa_maxcut(reg, device, gamma, beta):

  # Sequence initialization
  seq = Sequence(reg, device)
  seq.declare_channel("digital", "raman_local")
  seq.declare_channel("rydberg", "rydberg_local")

  # Starting with state |+,+,+,+> (i.e. an eigenstate of the driver Hamiltonian)
  ## vertex (1)
  seq.target(..., ...)
  seq.add(..., ...)
  ## vertex (2)
  ...
  ...
  ## vertex (3)
  ...
  ...
  ## vertex (4)
  ...
```

```
    ...

    # Optimization cycles
    for i in range(len(gamma)):

      # Cost Hamiltonian evolution:
      ## edge (2) <-> (3)
      seq = cx_seq(seq, ..., ..., blockade_radius=...)
      seq.phase_shift(..., ..., basis="digital")
      ...
      ## edge (2) <-> (4)
      ...
      ...
      ...
      ## edge (3) <-> (4)
      ...
      ...
      ...
      ## edge (1) <-> (2)
      ...
      ...
      ...

      # Driver Hamiltonian evolution:
      ## vertex (1)
      ...
      ...
      ## vertex (2)
      ...
      ...
      ## vertex (3)
      ...
      ...
      ## vertex (4)
      ...
      ...

    # Measurement apparatus
    seq.measure(basis="digital")

    # Simulation
    sim = Simulation(seq)
    res = sim.run()

    return res
```

f. **(Displaying results)** Complete the definition of the following function that displays as a bar chart the results of 4096 executions of QAOA as well as the success probability (i.e. the probability of getting a maximum cut):

```
def display_results(res):
    # Results
    counts = res.sample_final_state(N_samples=4096)

    ## 3-cut (i.e. MaxCut) contributions
    plt.bar(
        [..., ..., ..., ..., ..., ...],
        [..., ..., ..., ..., ..., ...],
        label='3-cut',
        color='r'
        )

    ## 2-cut contributions
    plt.bar(
        [..., ..., ..., ..., ..., ...],
        [..., ..., ..., ..., ..., ...],
        label="2-cut",
        color='b'
        )

    ## 1-cut contributions
    plt.bar(
        [..., ...],
        [..., ...],
        label="1-cut",
        color='y'
        )

    ## 0-cut contributions
    plt.bar(
        [..., ...],
        [..., ...],
        label="0-cut",
        color='g'
        )

    plt.xticks(rotation=60)
    plt.legend()
    plt.show()

    # Results summary
    print(counts)

    # Success probability
    N_success = ...
    p_success = ... / ...
    print("Succes probability :", p_success)
```

g. **(QAOA with one cycle of optimization)** Try your code with the following parameters that were found through numerical grid searches: $\gamma[0] = 0.208\pi$ and $\beta[0] = 0.105\pi$.

h. **(QAOA with two cycles of optimization)** Try your code with the following parameters that were found through numerical grid searches: $\gamma[0] = 0.2\pi$, $\gamma[1] = 0.4\pi$, $\beta[0] = 0.15\pi$ and $\beta[1] = 0.05\pi$. Have you noticed an improvement with two cycles of optimization instead of one?

Solution:

```
!pip install pulser

from pulser import Sequence, Pulse, Register
from pulser.devices import Chadoq2
from pulser_simulation import Simulation
from gate_pulses import h_pulse, rx_pulse
from control_gate_sequences import cx_seq
import matplotlib.pyplot as plt
import numpy as np

# Reproducing the vertices arrangement
device = Chadoq2
qubits = {"q1":(0,4), "q2":(0,0), "q3":(0,-4), "q4":(4,0)}
reg = Register(qubits)
reg.draw()


def qaoa_maxcut(reg, device, gamma, beta):

  # Sequence initialization
  seq = Sequence(reg, device)
  seq.declare_channel("digital", "raman_local")
  seq.declare_channel("rydberg", "rydberg_local")

  # Starting with state |+,+,+,+> (i.e. an eigenstate of the driver Hamiltonian)
  ## vertex (1)
  seq.target("q1", "digital")
  seq.add(h_pulse(),"digital")
  ## vertex (2)
  seq.target("q2", "digital")
  seq.add(h_pulse(),"digital")
  ## vertex (3)
  seq.target("q3", "digital")
  seq.add(h_pulse(),"digital")
  ## vertex (4)
  seq.target("q4", "digital")
  seq.add(h_pulse(),"digital")

  # Optimization cycles
  for i in range(len(gamma)):

    # Cost Hamiltonian evolution:
    ## edge (2) <-> (3)
    seq = cx_seq(seq, "q3", "q2", blockade_radius=8)
    seq.phase_shift(-gamma[i], "q2", basis="digital")
    seq = cx_seq(seq, "q3", "q2", blockade_radius=8)
    ## edge (2) <-> (4)
    seq = cx_seq(seq, "q4", "q2", blockade_radius=8)
    seq.phase_shift(-gamma[i], "q2", basis="digital")
    seq = cx_seq(seq, "q4", "q2", blockade_radius=8)
    ## edge (3) <-> (4)
    seq = cx_seq(seq, "q4", "q3", blockade_radius=8)
```

```
        seq.phase_shift(-gamma[i], "q3", basis="digital")
        seq = cx_seq(seq, "q4", "q3", blockade_radius=8)
        ## edge (1) <-> (2)
        seq = cx_seq(seq, "q1", "q2", blockade_radius=8)
        seq.phase_shift(-gamma[i], "q2", basis="digital")
        seq = cx_seq(seq, "q1", "q2", blockade_radius=8)

        # Driver Hamiltonian evolution:
        ## vertex (1)
        seq.target("q1", "digital")
        seq.add(rx_pulse(2*beta[i]), "digital")
        ## vertex (2)
        seq.target("q2", "digital")
        seq.add(rx_pulse(2*beta[i]), "digital")
        ## vertex (3)
        seq.target("q3", "digital")
        seq.add(rx_pulse(2*beta[i]), "digital")
        ## vertex (4)
        seq.target("q4", "digital")
        seq.add(rx_pulse(2*beta[i]), "digital")

    # Measurement apparatus
    seq.measure(basis="digital")

    # Simulation
    sim = Simulation(seq)
    res = sim.run()

    return res


def display_results(res):

    # Results
    counts = res.sample_final_state(N_samples=4096)

    ## 3-cut (i.e. MaxCut) contributions
    plt.bar(
        ['0100', '0101', '0110', '1001', '1010', '1011'],
        [counts['0100'], counts['0101'], counts['0110'],
         counts['1001'], counts['1010'], counts['1011']],
        label='3-cut',
        color='r'
        )

    ## 2-cut contributions
    plt.bar(
        ['0001', '0010', '0011', '1100', '1101', '1110'],
        [counts['0001'], counts['0010'], counts['0011'],
         counts['1100'], counts['1101'], counts['1110']],
        label="2-cut",
        color='b'
        )
```

```
  ## 1-cut contributions
  plt.bar(
      ['0111', '1000'],
      [counts['0111'], counts['1000']],
      label="1-cut",
      color='y'
      )

  ## 0-cut contributions
  plt.bar(
      ['0000', '1111'],
      [counts['0000'], counts['1111']],
      label="0-cut",
      color='g'
      )

  plt.xticks(rotation=60)
  plt.legend()
  plt.show()

  # Results summary
  print(counts)

  # Success probability
  N_success = counts['1001'] + counts['0101'] + counts['1011'] +
  counts['0100'] + counts['1010'] + counts['0110']
  p_success = N_success / 4096
  print("Succes probability :", p_success)


# QAOA with one cycle of optimization
gamma = [0.208 * np.pi]
beta = [0.105 * np.pi]
counts_with_one_cycle = qaoa_maxcut(reg, device, gamma, beta)
display_results(counts_with_one_cycle)

# QAOA with two cycles of optimization
gamma = [0.2 * np.pi, 0.4 * np.pi]
beta = [0.15 * np.pi, 0.05 * np.pi]
counts_with_two_cycles = qaoa_maxcut(reg, device, gamma, beta)
display_results(counts_with_two_cycles)
```