Theory and Implementation of the Quantum Approximate Optimization Algorithm

A comprehensible introduction and case study using Qiskit and IBM quantum computers

Andreas Sturm [1]

Fraunhofer IAO, Fraunhofer Institute for Industrial Engineering,
70569 Stuttgart, Germany

January 24, 2023

[1]E-mail: andreas.sturm@iao.fraunhofer.de

Overview

Introduction

The present tutorial aims to provide a comprehensible and easily accessible introduction into the theory and implementation of the famous Quantum Approximate Optimization Algorithm (QAOA). We lay our focus on practical aspects and step-by-step guide through the realization of a proof of concept quantum application based on a real-world use case. In every step we first explain the underlying theory and subsequently provide the implementation using IBM's Qiskit. In this way we provide a thorough understanding of the mathematical modelling and the (quantum) algorithms as well as the equally important knowledge how to properly write the code implementing those theoretical concepts. As another central aspect of this tutorial we provide extensive experiments on the 27 qubits state-of-the-art quantum computer ibmq_ehningen. From the discussion of these experiments we gain an overview on the current status of quantum computers and deduce which problem sizes can meaningfully be executed on today's hardware.

Structure

This tutorial is divided into four notebooks. In Notebook 1 we introduce the use case that will accompany us all the time and on which we will demonstrate all ideas and concepts. We present the mathematical modeling of the use case as an optimization problem as well as the associated implementation in Python and Qiskit. Notebook 2 is dedicated to the theory and implementation of QAOA. Moreover, we will discuss the classical optimization that is always present in variational algorithms like QAOA and provide advanced knowledge on the gate synthesis of the QAOA operators. Subsequently, Notebook 3 deals with the topic how QAOA circuits can be optimally executed on real quantum computers. We present a transpilation pipeline that implements this task and explain how to run experiments on real quantum hardware. Additionally, we demonstrate in this notebook how the experimental results can be post-processed with the big data tool Pandas. Finally, in Notebook 4 we present a large variety of results from experiments on simulators and most of all from the quantum computer ibmq_ehningen. Among others, we discuss challenges in classical optimization, that besides the number of required qubits the amount of coupling of the variables in the underlying optimization problem is a central factor in judging if a problem is feasible for current quantum hardware, and that the transpilation pipeline and error mitigation techniques have a great effect on the quality of experimental results. While those techniques allow to get the most out from current quantum computers our experiments also clearly show today's limits with respect to the problem size and the quality of the results.

Code Availability and Technical Requirements

This tutorial is based and closely follows demonstration notebooks that were developed as a part of the project SEQUOIA within the Competence Center Quantum Computing Baden-Württemberg. The codes shown in this tutorial as well as the codes of the used helper functions are available in the SEQUOIA use case database or upon request from the author.

All codes were developed with

- `mthree==1.1.0`
- `numpy==1.19.5`
- `pandas==1.2.1`
- `plotly==4.14.3`
- `qiskit==0.39.4`
- `qiskit_aer==0.11.1`
- `qiskit_terra==0.22.2`

Acknowledgement

Contents

Notebook 1

1.1 Real World Problem

The project **LamA - Laden am Arbeitsplatz** (english: charging at work) is a joint project led by Fraunhofer IAO and funded by the federal government of Germany. Among others the goal is to build up charging infrastructure for electric vehicles at 37 institutes of the Fraunhofer society across Germany so that employees can charge their electric cars at work [5, 6, 13].

In this tutorial we aim to provide an optimal charging schedule for this charging infrastructure. Clearly, such an optimization can be implemented with respect to different aspects, e.g. by involving weather predictions we could optimize the schedule such that as much clean energy (sun, wind, etc.) as possible is used. Or, considering that the price of energy from the public electricity grid varies over time and the electric vehicles are charged during working hours, i.e. around a fairly long period of about 8 hours, we could optimize the schedule such that it takes energy from the grid when it is the cheapest. Having in mind that quantum computing is currently in the NISQ (noisy intermediate scale quantum) era we decided for this tutorial to consider the aspect of **minimizing the peak load** that is taken from the eletricity grid. This reduces costs, relieves the public electricity grid and – as we will see – can be reduced to a meaningful proof of concept problem that can be executed on today's available quantum computers.

1.2 Proof of Concept Model

Let us begin by introducing the proof of concept (POC) model that we will consider in this tutorial. It is an **optimization task** where we are **given**

- the number of cars that have to be charged,
- the arrival and departure times of these cars, and
- the required energies they need to charge.

Our **aim** is to

- minimize the peak load taken from the electricity grid,
- meet the time restrictions imposed by the arrival and departure times, and
- charge the correct amount of energies.

In order to make this optimization task feasible for a NISQ computer we make the following **simplifications**:

- We work with discrete time slots and

- we assume that we can only charge on discrete loading levels.

Let us illustrate this rather abstract optimization task with a simple toy example.

1.2.1 Toy Example

We consider a charging station with 6 charging levels (i.e. levels 0, 1, ..., 5) and 7 available time slots (i.e. slots 0, 1, ..., 6). Moreover, let us take two cars (named car_green and car_orange), where

- car_green is at the charging station at time slots 0, ..., 3 and needs to charge 8 energy units, and
- car_orange is at the charging station at time slots 1, ..., 6 and needs to charge 12 energy units.

A visualization of the example is given in Figure 1.1.
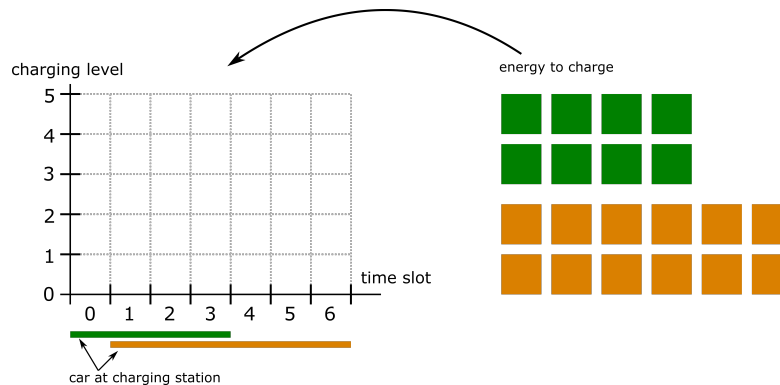
Figure 1.1: A simple example with one charging station and two cars.

An optimization process for the above situation could for example yield the charging schedules depicted in Figure 1.2.

6
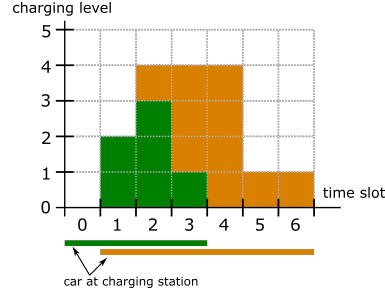
(a) A non-feasible schedule because the energy constraint is violated.
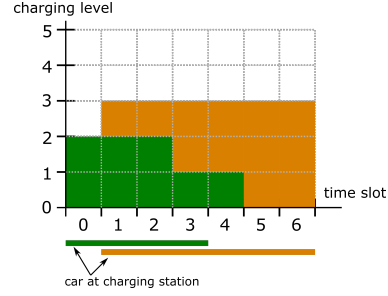
(b) A non-feasible schedule because the time constraint is violated.
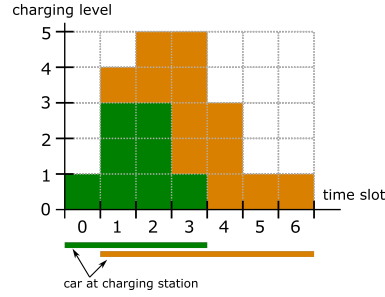
(c) A non-optimal schedule because the peak load is not as low as possible.
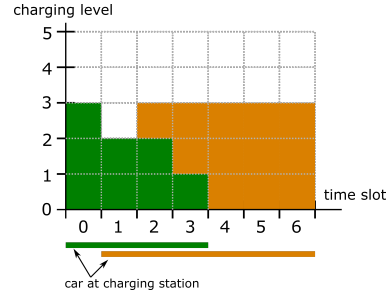
(d) An optimal schedule.

Figure 1.2: Possible outcomes of an optimization process for the situation described in Figure 1.1

Our next step is to find a formal mathematical description for our POC model.

1.3 Mathematical Description of POC Model

1.3.1 Notation

In the following we use the notation:

- $T \in \mathbb{N}$: Number of time slots $\rightarrow$ we have time slots $0, 1, \ldots, T - 1$.
- $L \in \mathbb{N}$: Number of charging levels $\rightarrow$ we have charging levels $0, 1, \ldots, L - 1$.
- $K \in \mathbb{N}$: Number of cars $\rightarrow$ we have K cars that we label as $0, 1, \ldots, K - 1$.

Moreover, for every car $k \in \{0, \ldots, K - 1\}$ we define:

- $t_k^a \in \{0, \ldots, T - 1\}$: Arrival time of car k.
- $t_k^d \in \{0, \ldots, T - 1\}$: Departure time of car k.
- $e_k \in \mathbb{N}$: Required amount of energy.

Finally, for every car $k \in \{0, \ldots, K - 1\}$ and for every time slot $t \in \{0, \ldots, T - 1\}$ we define:

- $p_k^t \in \{0, \ldots, L - 1\}$: Charging level on which car k charges at time slot t (i.e. the amount of energy packages it draws from the electricity grid on time slot t).

For a shorter notation we introduce the vectors

7

- $\vec{p}_k = \begin{pmatrix} p_k^0 \\ \vdots \\ p_k^{T-1} \end{pmatrix} \in \{0, \ldots, L-1\}^T$, where $k \in \{0, \ldots, K-1\}$,

- $\vec{p} = \begin{pmatrix} \vec{p}_0 \\ \vdots \\ \vec{p}_{K-1} \end{pmatrix} \in \{0, \ldots, L-1\}^{KT}$.

The entries of $\vec{p}_k$ form the charging curve for car k, see Figure 1.2.

Last, we define the vector $\vec{p}_{\text{sum}}$ so that its t-th component is the sum of the charging levels of all cars for time slot t:

$$\vec{p}_{\text{sum}} = \vec{p}_0 + \vec{p}_1 + \cdots + \vec{p}_{K-1} = \begin{pmatrix} p_0^0 + p_1^0 + \ldots p_{K-1}^0 \\ \vdots \\ p_0^{T-1} + p_1^{T-1} + \ldots p_{K-1}^{T-1} \end{pmatrix} .$$

It is easy to see that we have

$$\vec{p}_{\text{sum}} = \big(\underbrace{I_T \ I_T \ \cdots \ I_T}_{K \text{ times}} \big) \vec{p} \, , \tag{1.1}$$

where I_T is the $T \times T$ identity matrix.

With this notation at hand we can set up a cost function that assigns a cost to a charging schedule.

1.3.2 Cost Function

Note that the summation of all components of the vector $\vec{p}_{\text{sum}}$ gives the total energy drawn from the electricity grid. Further, observe that for the same total energy the smaller the ℓ_2-norm $\|\vec{p}_{\text{sum}}\|_2$ of $\vec{p}_{\text{sum}}$ the smaller the peak load. E.g.

$$\left\| \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \right\|_2^2 = 1^2 + 2^2 + 3^2 = 14, \qquad \left\| \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix} \right\|_2^2 = 2^2 + 2^2 + 2^2 = 12.$$

Moreover, note that

$$\|\vec{p}_{\text{sum}}\|_2^2 = \vec{p}_{\text{sum}}^t \vec{p}_{\text{sum}} = \vec{p}^{\,t} A \vec{p} \, , \qquad A = \mathbb{1}_K^t \mathbb{1}_K \otimes I_T \, ,$$

where we used (1.1) and where $\mathbb{1}_K = (1, 1 \ldots, 1)^t \in \mathbb{R}^K$.

So, we can use the following **cost function** for the minimization of the peak load:

$$f_1(\vec{p}) = \vec{p}^{\,t} A \vec{p} \, . \tag{1.2}$$

Clearly, only minimizing f_1 would yield $\vec{p} = (0, 0, \ldots, 0)^t$, which means that no charging would take place at all. Thus, we have to incorporate constraints that enforce the charging of the correct amount of energy at valid time slots.

1.3.3 Constraints

For every $k \in \{0, \ldots, K-1\}$ we define

$$C_k = \begin{pmatrix} 0 & \ldots 0 & \underbrace{1}_{t_k^a} & 1 \ldots 1 & \underbrace{1}_{t_k^d} & 0 \ldots 0 \end{pmatrix},$$

and collect all C_k in a matrix C:

$$C = \begin{pmatrix} C_0 & & & \\ & C_1 & & \\ & & \ddots & \\ & & & C_{K-1} \end{pmatrix}.$$

Then, the **constraints** (charge at valid times the right amount of energy) are given by

$$C\vec{p} = \vec{e}, \qquad \vec{e} = \begin{pmatrix} e_0 \\ \vdots \\ e_{K-1} \end{pmatrix},$$

where e_k is the amount of energy car k needs to charge, see Sections 1.3.2 and 1.3.3.

1.3.4 Minimization Problem

Putting all together we have a **quadratic constrained integer optimization** problem that reads

$$\begin{cases} \min\limits_{\vec{p} \in \{0, \ldots, L-1\}^{KT}} f_1(\vec{p}), \\ \\ \text{such that } C\vec{p} = \vec{e}. \end{cases} \tag{QCIO}$$

Next, we want to implement the logic of (QCIO) in **Python** and **Qiskit**.

1.4 Implementation of POC Model

1.4.1 Part 1: Python

First, we write a class that holds the data for a car that should be charged.

```python
from typing import List

class Car:
    def __init__(
        self,
        car_id: str, # an arbitrary name for the car
        time_slots_at_charging_unit: List[int], # time slots when the
                                                # car is at the charging unit
        required_energy: int # energy units that should be charged
```

```python
        ) -> None:
            self.car_id = car_id
            self.time_slots_at_charging_unit = time_slots_at_charging_unit
            self.required_energy = required_energy

        def __str__(self) -> str:
            return f"Car '{self.car_id}':\n" \
                "  at charging station at time slots " \
                f"{self.time_slots_at_charging_unit}\n" \
                "  requires " \
                f"{self.required_energy} energy units"
```

Next, we write a class to hold the data for a charging unit and with the possibility to register cars to charge at it. Moreover, to keep things simple we give this class the task to generate the matrices of the mathematical formulation of our POC model, see Section 1.3.

```python
[6]: import numpy as np

class ChargingUnit:
    def __init__(
        self,
        charging_unit_id: str, # an arbitrary name for the charging unit
        number_charging_levels: int,
        number_time_slots: int,
    ) -> None:
        self.charging_unit_id = charging_unit_id
        self.number_charging_levels = number_charging_levels
        self.number_time_slots = number_time_slots
        self.cars_to_charge = []

    def __str__(self) -> str:
        info_cars_registered = ""
        for car in self.cars_to_charge:
            info_cars_registered = info_cars_registered + " " + car.car_id
        return "Charging unit with\n" \
            "  charging levels: " \
            f"{list(range(self.number_charging_levels))[1:-1]}\n" \
            "  time slots: " \
            f"{list(range(self.number_time_slots))[1:-1]}\n" \
            "  cars to charge:" \
            + info_cars_registered

    def register_car_for_charging(self, car: Car) -> None:
        if max(car.time_slots_at_charging_unit) > self.number_time_slots - 1:
            raise ValueError("From car required time slots not compatible "
                " with charging unit.")
        self.cars_to_charge.append(car)
```

10

```python
    def reset_cars_for_charging(self) -> None:
        self.cars_to_charge = []

    def generate_constraint_matrix(self) -> np.ndarray:
        """Matrix with ones for times when car is at charging station
         and with zeros if car is not at charging station"""
        number_cars_to_charge = len(self.cars_to_charge)
        constraint_matrix = np.zeros(
            (number_cars_to_charge,
                number_cars_to_charge*self.number_time_slots))
        for row_index in range(0, number_cars_to_charge):
            offset = row_index*self.number_time_slots
            cols = np.array(
                self.cars_to_charge[row_index].time_slots_at_charging_unit)
            constraint_matrix[row_index, offset+cols] = 1
        return constraint_matrix

    def generate_constraint_rhs(self) -> np.ndarray:
        """Vector with required energy as entries"""
        number_cars_to_charge = len(self.cars_to_charge)
        constraint_rhs = np.zeros((number_cars_to_charge, 1))
        for row_index in range(0, number_cars_to_charge):
            constraint_rhs[row_index] = \
                self.cars_to_charge[row_index].required_energy
        return constraint_rhs

    def generate_cost_matrix(self) -> np.ndarray:
        number_cars_to_charge = len(self.cars_to_charge)
        return np.kron(
            np.ones((number_cars_to_charge, 1)) \
                @ np.ones((1, number_cars_to_charge)),
            np.eye(self.number_time_slots))
```

Toy Example: Implementation in Python

Let's instantiate the objects for our upper example:

```python
[3]: car_green = Car(
         car_id="car_green",
         time_slots_at_charging_unit=[0, 1, 2, 3],
         required_energy=8)

     car_orange = Car(
         car_id="car_orange",
         time_slots_at_charging_unit=[1, 2, 3, 4, 5,  6],
         required_energy=12)
```

```
[4]: print(car_green)
     print(car_orange)
```

```
Car 'car_green':
  at charging station at time slots [0, 1, 2, 3]
  requires 8 energy units
Car 'car_orange':
  at charging station at time slots [1, 2, 3, 4, 5, 6]
  requires 12 energy units
```

```
[5]: charging_unit = ChargingUnit(
         charging_unit_id="charging_unit",
         number_charging_levels=6,
         number_time_slots=7)
```

```
[6]: print(charging_unit)
```

```
Charging unit with
  charging levels: 0, 1, 2, 3, 4, 5
  time slots: 0, 1, 2, 3, 4, 5, 6
  cars to charge:
```

```
[7]: charging_unit.register_car_for_charging(car_green)
     charging_unit.register_car_for_charging(car_orange)
     print(charging_unit)
```

```
Charging unit with
  charging levels: 0, 1, 2, 3, 4, 5
  time slots: 0, 1, 2, 3, 4, 5, 6
  cars to charge: car_green car_orange
```

Now, let's get the cost matrix A, the constraint matrix C, and the constraint right-hand side (RHS) vector $\vec{e}$:

```
[8]: A = charging_unit.generate_cost_matrix()
     print(f"A =\n{A}")
```

```
A =
[[1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

```
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1.]]
```

[9]:
```python
C = charging_unit.generate_constraint_matrix()
e = charging_unit.generate_constraint_rhs()
print(f"C =\n{C}\n")
print(f"e =\n{e}")
```

```
C =
[[1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]]

e =
[[ 8.]
 [12.]]
```

Next, we move to Qiskit, where we can easily implement the minimization problem.

1.4.2 Part 2: Qiskit

The module `qiskit_optimization` contains the class `QuadraticProgram` which is used to represent quadratic optimization problems. In the following code we write a function that generates a `QuadraticProgram` instance according to (QCIO), where the necessary data stem from a `ChargingUnit` object.

Remark: For further information about `qiskit_optimization` and how it is used see e.g. [22].

[18]:
```python
from qiskit_optimization import QuadraticProgram

def generate_qcio(
    charging_unit: ChargingUnit,
    name: str=None
) -> QuadraticProgram:
    if name is None:
        name = ""
    qcio = QuadraticProgram(name)

    for car in charging_unit.cars_to_charge:
        qcio.integer_var_list(
            keys=[f"{car.car_id}_t{t}" \
                    for t in range(0, charging_unit.number_time_slots)],
            lowerbound=0,
            upperbound=charging_unit.number_charging_levels-1,
            name="p_")

    constraint_matrix = charging_unit.generate_constraint_matrix()
    constraint_rhs = charging_unit.generate_constraint_rhs()
    for row_index in range(0, constraint_matrix.shape[0]):
```

```
        qcio.linear_constraint(
            linear=constraint_matrix[row_index, :],
            rhs=constraint_rhs[row_index][0],
            sense="==",
            name=f"charge_correct_energy_for_{charging_unit.
→cars_to_charge[row_index].car_id}")

    cost_matrix = charging_unit.generate_cost_matrix()
    qcio.minimize(quadratic=cost_matrix)

    return qcio
```

Toy Example: Implementation in Qiskit

For our upper `charging_unit` we get:

```
[11]: qcio = generate_qcio(charging_unit, name="QCIO")
```

```
[12]: print(qcio.prettyprint())
```

```
Problem name: QCIO

Minimize
  p_car_green_t0^2 + 2*p_car_green_t0*p_car_orange_t0
  + p_car_green_t1^2 + 2*p_car_green_t1*p_car_orange_t1
  + p_car_green_t2^2 + 2*p_car_green_t2*p_car_orange_t2
  + p_car_green_t3^2 + 2*p_car_green_t3*p_car_orange_t3
  + p_car_green_t4^2 + 2*p_car_green_t4*p_car_orange_t4
  + p_car_green_t5^2 + 2*p_car_green_t5*p_car_orange_t5
  + p_car_green_t6^2 + 2*p_car_green_t6*p_car_orange_t6
  + p_car_orange_t0^2 + p_car_orange_t1^2 + p_car_orange_t2^2
  + p_car_orange_t3^2 + p_car_orange_t4^2 + p_car_orange_t5^2
  + p_car_orange_t6^2

Subject to
  Linear constraints (2)
    p_car_green_t0 + p_car_green_t1 + p_car_green_t2
    + p_car_green_t3 == 8  'charge_correct_energy_for_car_green'
    p_car_orange_t1 + p_car_orange_t2 + p_car_orange_t3
    + p_car_orange_t4 + p_car_orange_t5 + p_car_orange_t6
    == 12  'charge_correct_energy_for_car_orange'

  Integer variables (14)
    0 <= p_car_green_t0 <= 5
    0 <= p_car_green_t1 <= 5
    0 <= p_car_green_t2 <= 5
    0 <= p_car_green_t3 <= 5
```

```
0 <= p_car_green_t4 <= 5
0 <= p_car_green_t5 <= 5
0 <= p_car_green_t6 <= 5
0 <= p_car_orange_t0 <= 5
0 <= p_car_orange_t1 <= 5
0 <= p_car_orange_t2 <= 5
0 <= p_car_orange_t3 <= 5
0 <= p_car_orange_t4 <= 5
0 <= p_car_orange_t5 <= 5
0 <= p_car_orange_t6 <= 5
```

1.5 Solve POC Model with Classical Solver

In order to assess our later quantum algorithm it is of great advantage to know the exact solution of (QCIO). For small POC examples such as we are considering here this solution can be computed by a classical solver. Qiskit provides two such solvers (`CplexOptimizer` and `GurobiOptimizer`) in `qiskit_optimization.algorithms`.

Remark: Further information is available at [8].

1.5.1 Toy Example: Classical Solver

We solve `qcio` from above with `CplexOptimizer`.

```
[21]: from qiskit_optimization.algorithms import CplexOptimizer

      cplex_optimizer = CplexOptimizer()

      qcio_minimization_result = cplex_optimizer.solve(qcio)

      print("minimum point: p_min = ", qcio_minimization_result.x)
      print("minimum value: f_1(p_min) = ", qcio_minimization_result.fval)
```

```
minimum point: p_min =  [3. 2. 0. 3. 0. 0. 0. 0. 0. 3. 0. 3. 3. 3.]
minimum value: f_1(p_min) =  58.0
```

The array `qcio_minimization_result.x` corresponds to a solution $\vec{p}_{\min} = \begin{pmatrix} \vec{p}_{0,\min} \\ \vec{p}_{1,\min} \end{pmatrix}$ of (QCIO), where $\vec{p}_{0,\min}$ and $\vec{p}_{1,\min}$ correspond to solutions for car_green and car_orange, respectively.

```
[24]: print("minimum point for car_green: p_0,min = "
            f"{qcio_minimization_result.x[0:charging_unit.number_time_slots]}")
      print("minimum point for car_orange: p_1,min = "
            f"{qcio_minimization_result.x[charging_unit.number_time_slots:]}")
```

```
minimum point for car_green: p_0,min =  [3. 2. 0. 3. 0. 0. 0.]
minimum point for car_orange: p_1,min =  [0. 0. 3. 0. 3. 3. 3.]
```

We can plot the solution with the provided function `plot_charging_schedule`

```
[25]: from utils import plot_charging_schedule

     fig = plot_charging_schedule(
         charging_unit, qcio_minimization_result.x, marker_size=20)
     fig.update_layout(width=400, height=300)
     fig.show()
```
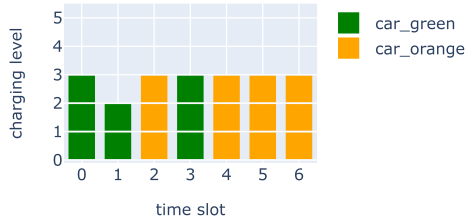

Recalling our toy example, see Section 1.2.1 and Figure 1.1, we see that this is indeed an optimal solution.

1.6   Convert QCIO to a QUBO

Later we want to solve our optimization problem (QCIO) with the quantum algorithm QAOA. This algorithm requires our optimization problem in a different form, which will derive in this section.

Recall that (QCIO) was given by

$$\min_{\vec{p}\in\{0,\dots,L\}^{KT}} f_1(\vec{p}) \qquad \text{such that } C\vec{p}=\vec{e}\,, \qquad \text{where } f_1(\vec{p})=\vec{p}^t A\vec{p}\,.$$

In the sequence we transform (QCIO) by the following two steps:

Step 1: Convert hard constraints to soft constraints.
        This means include $C\vec{p}=\vec{e}$ into the cost function.

Step 2: Binary encoding of integer variables $\vec{p}$.
        This means to transform the problem such that we have binary variables $\vec{b}$.

1.6.1   Convert Hard to Soft Constraints

For a **penalty parameter** $\varrho \geq 0$ we define

$$f_2(\vec{p};\varrho) = f_1(\vec{p}) + \varrho\|C\vec{p}-\vec{e}\|_2^2\,.$$

Note that f_2 is also a quadratic cost function with

$$f_2(\vec{p};\varrho) = \vec{p}^t \hat{A}_\varrho \vec{p} + \hat{L}_\varrho \vec{p} + \hat{c}_\varrho\,,$$

16

where

$$\hat{A}_\varrho = A + \varrho C^t C \ , \qquad \hat{L}_\varrho = -2\varrho \vec{e}^{\,t} C \ , \qquad \hat{c}_\varrho = \varrho \|\vec{e}\|_2^2 \ .$$

Now, we have (for a fixed penalty parameter ϱ) a **quadratic unconstrained integer optimization** problem

$$\min_{\vec{p} \in \{0,\dots,L\}^{KT}} f_2(\vec{p}; \varrho) \ . \tag{QUIO}$$

It is important to note that if ϱ is chosen large enough then the solution of (QUIO) is also a solution of (QCIO). On the other hand, this means that choosing ϱ too small can lead to **unfeasible** solutions, i.e. solutions $\vec{p}$ of (QUIO) that do not satisfy the constraint $C\vec{p} = \vec{e}$ of (QCIO).

1.6.2 Binary Encoding

A **binary encoding** is given by a transformation matrix B such that $\vec{p} = B\vec{b}$, where the coefficients b_i of $\vec{b}$ are binary, i.e. $b_i \in \{0, 1\}$.

By substituting $\vec{p} = B\vec{b}$ in (QUIO) we get a **quadratic unconstrained binary optimization** problem

$$\min_{\vec{b} \in \{0,1\}^{\tilde{N}}} f_3(\vec{b}; \varrho) \ , \tag{QUBO}$$

where

$$f_3(\vec{b}; \varrho) = \vec{b}^{\,t} \widetilde{A}_\varrho \vec{b} + \widetilde{L}_\varrho \vec{b} + \widetilde{c}_\varrho, \qquad \widetilde{A}_\varrho = B^t \hat{A}_\varrho B \ , \quad \widetilde{L}_\varrho = \hat{L}_\varrho B, \quad \widetilde{c}_\varrho = \hat{c}_\varrho \ .$$

In the next lines we give a simple example of a binary encoding and refer to the literature for advanced encodings.

Toy example: fixed width binary encoding

Let $w = 3$ be a fixed encoding width. Then, we can represent every component $p_i \in \{0, 1, \dots, 5\}$ of our vector $\vec{p}$ by

$$p_i = b_{i,0} \cdot 2^0 + b_{i,1} \cdot 2^1 + b_{i,2} \cdot 2^2 = \underbrace{\begin{pmatrix} 2^0 & 2^1 & 2^2 \end{pmatrix}}_{\widetilde{B}} \underbrace{\begin{pmatrix} b_{i,0} \\ b_{i,1} \\ b_{i,2} \end{pmatrix}}_{\vec{b}_i}, \quad \vec{b}_i \in \{0,1\}^3.$$

Using this, we can write

$$\vec{p} = \begin{pmatrix} p_0 \\ \vdots \\ p_{N-1} \end{pmatrix} = \underbrace{\begin{pmatrix} \widetilde{B} & & & \\ & \widetilde{B} & & \\ & & \ddots & \\ & & & \widetilde{B} \end{pmatrix}}_{B} \underbrace{\begin{pmatrix} \vec{b}_0 \\ \vdots \\ \vec{b}_{N-1} \end{pmatrix}}_{\vec{b}}, \qquad N = KT.$$

Note that the dimension increases: $\vec{p}$ has $N = KT$ entries, whereas $\vec{b}$ has $\widetilde{N} = wKT$ entries.

Other binary encodings

Further information on encondings and more examples can be found in [14]. Also the **bounded-coefficient encoding**, which **Qiskit uses by default**, is proposed in this paper (see the documention of the class `qiskit_optimization.converters.IntegerToBinary`).

Remarks

1) It is easy to prove that the matrix $\widetilde{A}_\varrho$ in (QUBO) can always be transformed such that it is an upper triangular matrix:

$$\widetilde{A}_\varrho = \begin{pmatrix} * & * & \ldots & * & * \\ 0 & * & \ldots & * & * \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \ldots & * & * \\ 0 & 0 & \ldots & 0 & * \end{pmatrix} .$$

This also holds true for A and $\hat{A}_\varrho$. We note this here because Qiskit will save the matrices in such a way.

2) In this tutorial we consider the original form of QAOA, i.e. the Quantum Approximate Optimization Algorithm. An adaption of this algorithm is the Quantum Alternating Operator Ansatz [4] (often also abbreviated with QAOA), where hard constraints are handled differently and thus Step 1 is not needed. Many examples can be found in [4] and also in [15].

1.6.3 Implementation

Next, we write a class `Converter` the implements the upper transformations. For this we use the following Qiskit classes:

- `QuadraticProgramConverter`: This is the abstract class for converters of quadratic programs in Qiskit. It enforces that we implement the methods `convert` and `interpret` in our class `Converter`.
- `LinearEqualityToPenalty`: For Step 1.
- `IntegerToBinary`: For Step 2.

```python
[30]: from typing import Union
from qiskit_optimization.converters import QuadraticProgramConverter, \
    LinearEqualityToPenalty, IntegerToBinary

class Converter(QuadraticProgramConverter):
    def __init__(
        self,
        penalty: float=None # the penalty parameter for step 1
    ) -> None:
        super().__init__()
        self._penalty = penalty
```

```
        self.linear_equality_to_penalty_converter = \
            LinearEqualityToPenalty(penalty)
        self.integer_to_binary_converter = IntegerToBinary()

    def convert(self, quadratic_program: QuadraticProgram) -> QuadraticProgram:
        return self.integer_to_binary_converter.convert(
            self.linear_equality_to_penalty_converter.convert(quadratic_program))

    def interpret(self, x: Union[np.ndarray, List[float]]) -> np.ndarray:
        return self.linear_equality_to_penalty_converter.interpret(
            self.integer_to_binary_converter.interpret(x))
```

1.6.4 Toy Example: Convert to QUBO

Let us use our `Converter` to convert `qcio`.

```
[17]: # Note: penalty <= 5.0 will give a non-feasible solution,
      # penalty > 5.0 will give a feasible solution.
      converter = Converter(penalty=5.1)
      qubo = converter.convert(qcio)
      qubo.name = "QUBO"
      print(qubo.prettyprint())
```

```
Problem name: QUBO

Minimize
  6.1*p_car_green_t0@0^2 + 24.4*p_car_green_t0@0*p_car_green_t0@1
  + 24.4*p_car_green_t0@0*p_car_green_t0@2
  + 10.2*p_car_green_t0@0*p_car_green_t1@0
  + 20.4*p_car_green_t0@0*p_car_green_t1@1
  + 20.4*p_car_green_t0@0*p_car_green_t1@2
  + 10.2*p_car_green_t0@0*p_car_green_t2@0
  + 20.4*p_car_green_t0@0*p_car_green_t2@1
  + 20.4*p_car_green_t0@0*p_car_green_t2@2
  + 10.2*p_car_green_t0@0*p_car_green_t3@0
  + 20.4*p_car_green_t0@0*p_car_green_t3@1
  + 20.4*p_car_green_t0@0*p_car_green_t3@2
  + 2*p_car_green_t0@0*p_car_orange_t0@0
  + 4*p_car_green_t0@0*p_car_orange_t0@1
  + 4*p_car_green_t0@0*p_car_orange_t0@2 + 24.4*p_car_green_t0@1^2
  + 48.8*p_car_green_t0@1*p_car_green_t0@2
  ...

Subject to
  No constraints

  Binary variables (42)
    p_car_green_t0@0 p_car_green_t0@1 p_car_green_t0@2
```

```
      p_car_green_t1@0 p_car_green_t1@1 p_car_green_t1@2
      ...
```

We can verify that the number of binary variables has grown in comparison with the number of integer variables:

```
[33]: number_integer_variables = qcio.get_num_integer_vars()
      print(f"Number integer variables: {number_integer_variables}")

      number_binary_variables = qubo.get_num_binary_vars()
      print(f"Number binary variables: {number_binary_variables}")
```

```
Number integer variables:   14
Number binary variables:   42
```

As a last step let's retrieve the matrix $\tilde{A}$ from `qubo` and make a plot of its sparsity pattern.

```
[34]: A_tilde = qubo.objective.quadratic.to_array()
      print(f"Dimension: {A_tilde.shape}")
```

```
Dimension:   (42, 42)
```

```
[51]: import matplotlib.pyplot as plt
      fig = plt.figure()
      ax = fig.add_subplot(1, 1, 1)
      ax.spy(A_tilde, markersize=3)
```
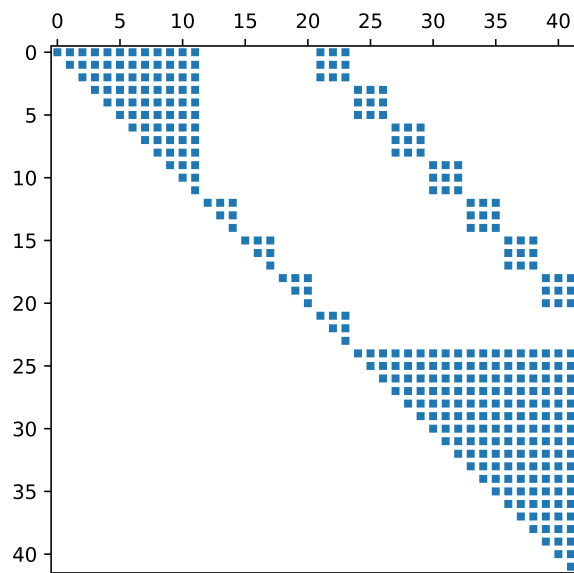

In Notebook 4 we will discuss the meaning and the importance of the sparsity pattern.

1.6.5   Solve with a Classical Solver

Analog to (QCIO) we can use a classical solver to solve (QUBO). In the next cells we do this for our `qubo` from above.

```
[21]: qubo_minimization_result = cplex_optimizer.solve(qubo)

      print("minimum point (binary): b_min = ", qubo_minimization_result.x)
      print("minimum value: f_3(b_min) = ", qubo_minimization_result.fval)
```

minimum point (binary): b_min = [1. 0. 1. 1. 0. 0. 1. 0. 1. 1. 0. 0. 0. 0. 0. 0.␣
→0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 1. 0. 1. 1. 0. 1. 1. 0.]
minimum value: f_3(b_min) = 57.999999999999886

The array `qubo_minimization_result.x` corresponds to a solution $\vec{b}_{\min}$ of (QUBO). With the method `interpret` of `converter` we can transform the binary vector $\vec{b}_{\min}$ to the integer vector $\vec{p}_{\min}$ (which is a feasible solution of (QCIO) if ϱ is chosen large enough).

```
[22]: b_min = qubo_minimization_result.x
      p_min = converter.interpret(b_min)

      print("minimum point (integer): p_min = ", p_min)
      print("minimum value: f_1(p_min) = ", qcio.objective.evaluate(p_min))
      print("minimum point feasible = ", qcio.is_feasible(p_min))
```

minimum point (integer): p_min = [3. 1. 3. 1. 0. 0. 0. 0. 2. 0. 2. 2. 3. 3.]
minimum value: f_1(p_min) = 58.0
minimum point feasible = True

Last, let's plot the solution:

```
[23]: fig = plot_charging_schedule(charging_unit, p_min, marker_size=20)
      fig.update_layout(width=400, height=300)
      fig.show()
```
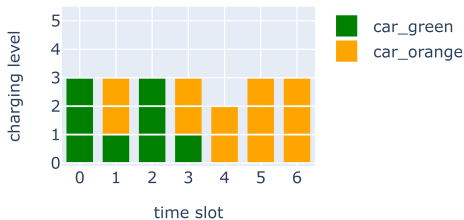
Notebook 2

2.1 Introduction

In Notebook 1 we have seen how to model and implement (a proof of concept version of) a real-world problem on optimizing the charging schedules for electric vehicles. Moreover, we showed how to transform and implement this problem as a QUBO which is the required form for the **Quantum Approximate Optimization Algorithm (QAOA)** [2] .

In this notebook we will introduce QAOA, explain how a QAOA circuit can be derived from a QUBO and how it can be implemented in Qiskit.

We will demonstrate these points with a small example from the use case introduced in Notebook 1.

2.2 Example for this Notebook

Our example for this whole notebook is a **charging station** with **4 charging levels** and **4 available time slots**. **One car** is at the charging station at **time slots 0, 1 and 2**, and needs to **charge 4 energy units**. The situation is depicted in Figure 2.1.
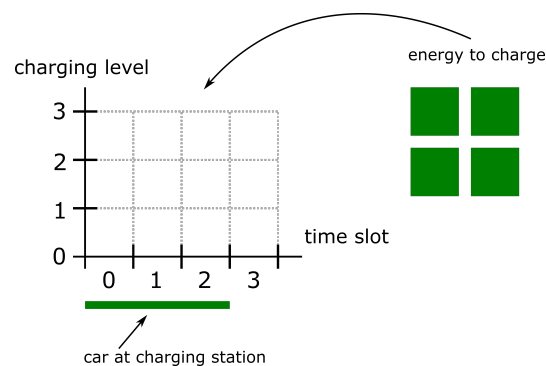

Figure 2.1: Example for this notebook.

The following cell implements the upper situation based on our codes from Notebook 1.

```
[1]: from codes_notebook_1 import generate_example
charging_unit, car_green, qcio, converter, qubo, \
```

```
        number_binary_variables, qubo_minimization_result = generate_example()
```

[2]:
```
print(charging_unit)
print(car_green)
```

```
Charging unit with
  charging levels: 0, 1, 2, 3
  time slots: 0, 1, 2, 3
  cars to charge: car_green
Car 'car_green':
  at charging station at time slots [0, 1, 2]
  requires 4 energy units
```

Let us compute an exact solution so that we have a reference to compare our quantum solution with.

[3]:
```
b_min = qubo_minimization_result.x
f_3_min = qubo_minimization_result.fval
print("minimum point (binary): b_min = ", b_min)
print("minimum value: f_3(b_min) = ", f_3_min)
```

```
minimum point (binary): b_min =  [0. 1. 1. 0. 1. 0. 0. 0.]
minimum value: f_3(b_min) =  5.999999999999993
```

[4]:
```
p_min = converter.interpret(b_min)
f_1_min = qcio.objective.evaluate(p_min)
p_min_feasible = qcio.is_feasible(p_min)

print("minimum point (integer): p_min = ", p_min)
print("minimum value: f_1(p_min) = ", f_1_min)
print("minimum point feasible = ", p_min_feasible)
```

```
minimum point (integer): p_min =  [2. 1. 1. 0.]
minimum value: f_1(p_min) =  6.0
minimum point feasible =  True
```

[5]:
```
from utils import plot_charging_schedule
fig = plot_charging_schedule(
    charging_unit, p_min, marker_size=30)
fig.update_layout(width=350, height=300)
fig.show()
```

2.3 Revision: QUBO

In Notebook 1 we discussed how a QUBO can be derived from an optimization use case. For this notebook we move the use case more in the background and write a QUBO in the general form

$$\min_{\vec{b}\in\{0,1\}^n} f_3(\vec{b}), \qquad f_3(\vec{b}) = \vec{b}^t A\vec{b} + L\vec{b} + c\,, \qquad A \in \mathbb{R}^{n\times n},\ L \in \mathbb{R}^{n\times 1},\ c \in \mathbb{R}. \qquad \text{(QUBO)}$$

From now on, if we refer to (QUBO) we mean the problem above.

A first step to solve (QUBO) with QAOA is to transform it into an Ising Hamiltonian [3, 18].

2.4 Convert QUBO to Ising Hamiltonian

2.4.1 Theory

We write the QUBO cost function f_3 as

$$f_3(\vec{b}) = \sum_{i=0}^{n-1}\sum_{j>i}^{n-1} a_{ij}b_i b_j + \sum_{i=0}^{n-1} l_i b_i + c\,,$$

where a_{ij} and l_i are the entries of A and L, respectively. Subsequently, we replace

$$b_i \leftrightarrow \tfrac{1}{2}\left(I^{\otimes n} - \sigma_Z^{(i)}\right),$$

where

$$I^{\otimes n} = \underbrace{I \otimes \cdots \otimes I}_{n\ \text{times}}, \qquad I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\,,$$

and

$$\sigma_Z^{(i)} = I \otimes \cdots \otimes I \otimes \underbrace{Z}_{\text{position } i} \otimes I \otimes \cdots \otimes I\,, \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\,.$$

This gives the **cost (or problem) Hamiltonian** $\mathcal{H}_P$, which is of the form

$$\mathcal{H}_P = \sum_{i=0}^{n-1}\sum_{j>i}^{n-1} h_{ij}\sigma_Z^{(i)}\sigma_Z^{(j)} + \sum_{i=0}^{n-1} h_i'\sigma_Z^{(i)} + h''I^{\otimes n} \ , \tag{2.1}$$

where the coefficients h_{ij}, h_i' and h'' can be computed from a_{ij}, l_i and c.

The following theorem establishes the connection between the QUBO cost function f_3 and the cost Hamiltonian $\mathcal{H}_P$.

Theorem For a quantum state $|\psi\rangle$ with amplitudes λ_b, i.e.

$$|\psi\rangle = \sum_{\vec{b}\in\{0,1\}^n} \lambda_b |b\rangle \ , \qquad \lambda_b \in \mathbb{C} \ , \qquad \sum_{\vec{b}\in\{0,1\}^n} |\lambda_b|^2 = 1 \ ,$$

we have

$$\langle\psi|\mathcal{H}_P|\psi\rangle = \sum_{\vec{b}\in\{0,1\}^n} |\lambda_b|^2 f_3(\vec{b}) \ . \tag{2.2}$$

Remark 1: By writing b instead of $\vec{b}$ we mean the bitstring $b = b_0 b_1 \ldots b_{n-1}$ associated with the vector $\vec{b} = (b_0, b_1, \ldots, b_{n-1})^t$.

Remark 2: Since this tutorial is concerned with practical aspects of quantum computing we leave out the mathematical details and derivations. However, it is a good exercise to do the computation QUBO $\rightarrow$ Ising and to prove the theorem.

2.4.2 Implementation

In Notebook 1 we already got to know the class `QuadraticProgram`. In particular, `qubo` is an object of this class. Now, we need this class' method `to_ising` since it implements the transformation discussed above. However, note that `to_ising` splits $\mathcal{H}_P$ into two parts, namely

- `ising` that contains the first two terms of the RHS of (2.1), and

- `ising_offset` which contains the third term of the RHS of (2.1).

```
[6]: ising, ising_offset = qubo.to_ising()
```

```
[7]: print("ising: ", ising)
```

```
ising:  -3.3 * IIIIIIIZ
- 6.599999999999998 * IIIIIIZI
- 3.3000000000000007 * IIIIIZII
- 6.6 * IIIIZIII
- 3.2999999999999994 * IIIZIIII
- 6.599999999999999 * IIZIIIII
+ 4.6 * IIIIIIZZ
+ 1.8 * IIIIIZIZ
+ 3.6 * IIIIIZZI
+ 3.6 * IIIIZIIZ
+ 7.2 * IIIIZIZI
```

```
+ 4.6 * IIIIZZII
+ 1.8 * IIIZIIIZ
+ 3.6 * IIIZIIZI
+ 1.8 * IIIZIZII
+ 3.6 * IIIZZIII
+ 3.6 * IIZIIIIZ
+ 7.2 * IIZIIIZI
+ 3.6 * IIZIIZII
+ 7.2 * IIZIZIII
+ 4.6 * IIZZIIII
- 1.5 * IZIIIIII
+ 1.0 * ZZIIIIII
- 3.0 * ZIIIIIII
```

[8]:
```
print("ising_offset:", ising_offset)
```

```
ising_offset: 28.400000000000013
```

Let's verify Equation (2.2) with the following state

$$|\psi\rangle = 0.7\,|00011100\rangle + 0.2\mathrm{i}\,|10101010\rangle + (0.6 + \sqrt{0.11}\mathrm{i})\,|11110000\rangle \ .$$

For this purpose we use the following classes from `qiskit.opflow`:

- `DictStateFn` to represent $|\psi\rangle$ and

- `OperatorStateFn` with parameter `is_measurement=True` to represent the observable $\mathcal{H}_P$ and to calculate the expecation value $\langle\psi|\mathcal{H}_P|\psi\rangle$.

Remark: For a tutorial on `qiskit.opflow` see [7].

[9]:
```
import numpy as np
from qiskit.opflow import DictStateFn, OperatorStateFn

psi = DictStateFn({"00011100": 0.7,
                   "10101010": 0.2j,
                   "11110000": 0.6+1j*np.sqrt(0.11)})

# Note that observable does not contain ising_offset
observable = OperatorStateFn(ising, is_measurement=True)

expectation_wo_offset = np.real(observable.eval(psi))
expectation = expectation_wo_offset + ising_offset

print("<psi | H_P | psi> = ", expectation)
```

```
<psi | H_P | psi> =  16.268000000000015
```

Before we calculate the RHS of (2.2) we recall that Qiskit uses a different ordering of the qubits

than most textbooks and different to the default ordering in Python. In fact, we have

$$\text{ordering textbooks/Python: } |b_0 \dots b_{n-1}\rangle \qquad \leftrightarrow \qquad \text{ordering Qiskit: } |b_{n-1} \dots b_0\rangle$$

It is important to note that methods like `QuadraticObjective.evaluate` or `QuadraticProgramConverter.interpret` expect their input in the **textbook/Python ordering**.

With this knowledge we can implement the RHS of (2.2):

```
[10]: result = 0
      for bitstring, amplitude in psi.primitive.items():
          # Convert string to array and CHANGE ORDERRING
          bitarray = np.fromiter(bitstring, int)[::-1]
          result += (np.abs(amplitude)**2) * qubo.objective.evaluate(bitarray)
      print("sum_b |lambda_b|^2 f_3(b) = ", result)
```

```
sum_b |lambda_b|^2 f_3(b) =  16.267999999999994
```

2.5   QAOA: Theory

QAOA aims to construct a quantum state

$$|\psi\rangle = \sum_{\vec{b} \in \{0,1\}^n} \lambda_b |b\rangle$$

that has amplitudes λ_b with large absolute value for those basis states $|b\rangle$ where $f_3(\vec{b})$ is small (or even better where f_3 is minimal). **Measuring** such a quantum state will result with **high probability** in a bitstring $\vec{b}^*$ that is a **(nearly) optimal solution** of (QUBO).

Equation (2.2) describes that such a quantum state can be obtained by minimizing the expectation value $\langle\psi|\mathcal{H}_P|\psi\rangle$. In fact, for

$$|\psi_{\min}\rangle = \sum_{\vec{b} \in \{0,1\}^n} \lambda_{b,\min} |b\rangle \qquad \text{with} \qquad |\psi_{\min}\rangle = \operatorname*{argmin}_{|\psi\rangle \in \mathbb{H}^{\otimes n}} \langle\psi|\mathcal{H}_P|\psi\rangle$$

we have that every basis state $|b\rangle$ with $\lambda_{b,\min} \neq 0$ is a minimum of f_3. Here $\mathbb{H}^{\otimes n}$ is the space of all n qubit states.

In order to construct an **approximation** of such an optimal state QAOA starts at the **uniform superposition**

$$|+\rangle^{\otimes n} = \underbrace{|+\rangle \otimes \cdots \otimes |+\rangle}_{n \text{ times}}, \qquad |+\rangle = \tfrac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big),$$

and then alternatingly applies p times a **phase operator** $\mathcal{U}_P$ and a **mixing operator** $\mathcal{U}_M$:

$$|\psi_{\text{QAOA}}(\vec{\beta}, \vec{\gamma})\rangle = \mathcal{U}_M(\beta_{p-1})\mathcal{U}_P(\gamma_{p-1}) \cdots \mathcal{U}_M(\beta_0)\mathcal{U}_P(\gamma_0)|+\rangle^{\otimes n}. \qquad \text{(QAOA-1)}$$

Here, $\vec{\beta} = (\beta_0, \ldots, \beta_{p-1})^t \in [0, \pi]^p$ and $\vec{\gamma} = (\gamma_0, \ldots, \gamma_{p-1})^t \in \mathbb{R}^p$ are parameters, and the phase and mixing operator are given by

$$\mathcal{U}_P(\gamma) = \exp(-\mathrm{i}\gamma \mathcal{H}_P) \quad \text{and} \quad \mathcal{U}_M(\beta) = \exp(-\mathrm{i}\beta \mathcal{H}_M),$$

respectively. The **mixer** $\mathcal{H}_M$ is defined by

$$\mathcal{H}_M = \sum_{i=0}^{n-1} \sigma_X^{(i)}, \qquad \sigma_X^{(i)} = I \otimes \cdots \otimes I \otimes \underbrace{X}_{\text{place } i} \otimes \, I \otimes \cdots \otimes I, \qquad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Following the reasoning above the parameters $\vec{\beta}$ and $\vec{\gamma}$ in (QAOA-1) should be chosen such that the expectation value

$$e(\vec{\beta}, \vec{\gamma}) = \langle \psi_{\text{QAOA}}(\vec{\beta}, \vec{\gamma}) | \mathcal{H}_P | \psi_{\text{QAOA}}(\vec{\beta}, \vec{\gamma}) \rangle \tag{QAOA-2}$$

is minimized, i.e.

$$(\vec{\beta}^*, \vec{\gamma}^*) = \operatorname*{argmin}_{\vec{\beta}, \vec{\gamma}} e(\vec{\beta}, \vec{\gamma}) . \tag{QAOA-3}$$

Then, measuring $|\psi_{\text{QAOA}}(\vec{\beta}^*, \vec{\gamma}^*)\rangle$ should result in a bitstring $\vec{b}^*$ that is a good approximation of the minimum of f_3, i.e.

$$f_3(\vec{b}^*) \approx \min_{\vec{b} \in \{0,1\}^n} f_3(\vec{b}) .$$

2.5.1 Classical Optimizers

The minimization task (QAOA-3) can be carried out by a classical optimizer (as e.g. provided in `qiskit.algorithms.optimizers`).

Very roughly speaking such an optimizer works in the following way: Starting from initial guesses $\vec{\beta}^{(0)}$ and $\vec{\gamma}^{(0)}$ it computes iteratively a next set of parameters $\vec{\beta}^{(j+1)}$ and $\vec{\gamma}^{(j+1)}$ with $e(\vec{\beta}^{(j+1)}, \vec{\gamma}^{(j+1)}) \leq e(\vec{\beta}^{(j)}, \vec{\gamma}^{(j)})$, $j = 0, 1, \ldots$. In this way $\vec{\beta}^{(j+1)}$ and $\vec{\gamma}^{(j+1)}$ approach $\vec{\beta}^*$ and $\vec{\gamma}^*$ of (QAOA-3) with every iteration. Clearly, in order to do the step $\vec{\beta}^{(j)}, \vec{\gamma}^{(j)} \curvearrowright \vec{\beta}^{(j+1)}, \vec{\gamma}^{(j+1)}$ the value of e in $(\vec{\beta}^{(j)}, \vec{\gamma}^{(j)})$ and (usually) the values of $e(\vec{\beta}^{(j)} + \vec{h}_\beta, \vec{\gamma}^{(j)} + \vec{h}_\gamma)$ for some $\vec{h}_\beta, \vec{h}_\gamma$ (to compute a gradient) are needed.

We see that this is a **hybrid workflow**, where for a given set of parameters the quantum state in (QAOA-1) is constructed on a quantum computer and the evaluation (QAOA-2) is computed from measurements of this quantum state. From this evaluation (and usually evaluations of small variations of the given parameters to obtain a gradient) a classical optimizer proposes a next set of parameters and the process starts from the beginning. In this way (QAOA-3) is iteratively approximated.

We note that this optimization is usually a difficult task and for many applications, initial guesses, hyperparameter choices, etc. the classical optimizer gets stuck in a **local minimum** and is not able to converge to the **global minimum** given by the optimal parameters $\vec{\beta}^*$ and $\vec{\gamma}^*$. Another issue are **barren plateaus** where the gradient vanishes exponentially. We refer to [1, 16] for more insights into these problems.

Remark 1: We usually omit the dependence on $\vec{\beta}$ and $\vec{\gamma}$ and just write $|\psi_{\text{QAOA}}\rangle = |\psi_{\text{QAOA}}(\vec{\beta}, \vec{\gamma})\rangle$.

Remark 2: Note that $\mathcal{U}_P(\gamma)$, $\mathcal{U}_M(\beta)$, $\mathcal{H}_P$, and $\mathcal{H}_M$ are $2^n \times 2^n$ matrices whereas A is a $n \times n$ matrix. Also note that if f_3 depends on ϱ also $\mathcal{H}_P$, $\mathcal{U}_P(\gamma)$ and $|\psi_{\text{QAOA}}\rangle$ depend on ϱ.

2.6 QAOA: Implementation

A quantum circuit that implements (QAOA-1) can be obtained from the class `QAOAAnsatz`.

In the next cell we use this class to get a QAOA circuit with $p = 2$ for our cost Hamiltonian `ising` from above.

```
[11]: from qiskit.circuit.library import QAOAAnsatz


      qaoa_reps = 2 # this corresponds to the parameter p in (QAOA)
      qaoa_ansatz = QAOAAnsatz(cost_operator=ising, reps=qaoa_reps, name='qaoa')
      qaoa_ansatz.measure_active()
```

```
[12]: qaoa_ansatz.decompose(reps=1).draw(scale=0.5, fold=-1)
```

[12]:
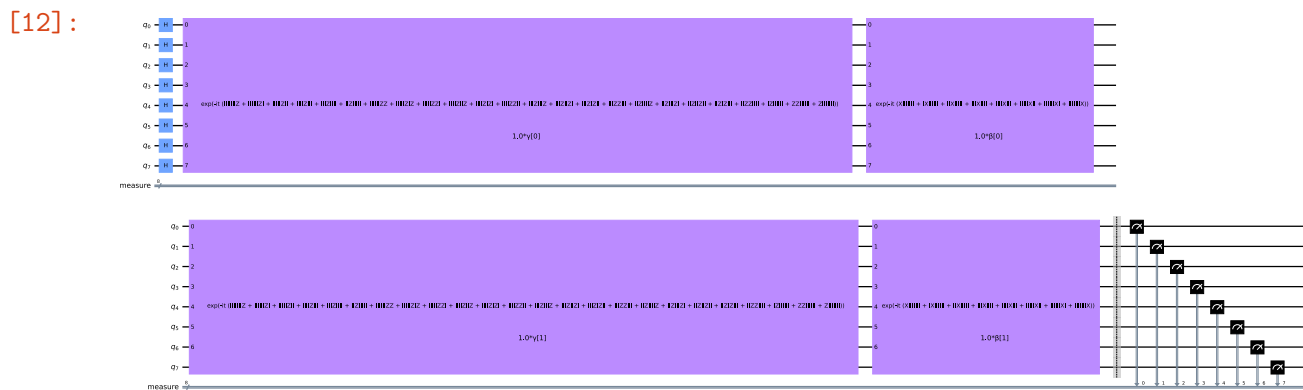
Note in which order `qaoa_ansatz` expects its parameters:

```
[13]: qaoa_ansatz.parameters
```

```
[13]: ParameterView([ParameterVectorElement(β[0]), ParameterVectorElement(β[1]),
      ParameterVectorElement(γ[0]), ParameterVectorElement(γ[1])])
```

Let us execute the circuit on a simulator for some arbitrarily chosen parameters and visualize the result.
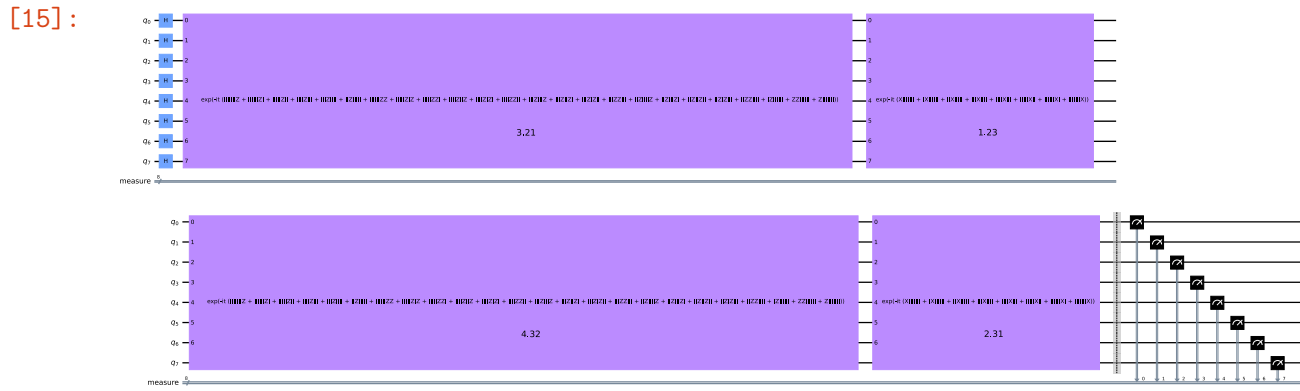
```
[28]: betas = [1.23, 2.31]
      gammas = [3.21, 4.32]


      # See above the ordering that qaoa_ansatz expects
      parameter_values = [*betas, *gammas]
      parameters = qaoa_ansatz.parameters
```

```
parameter_bindings = dict(zip(parameters, parameter_values))

qaoa_with_parameters = qaoa_ansatz.bind_parameters(parameter_bindings)
```

[15]:
```
qaoa_with_parameters.decompose(reps=1).draw(scale=0.5, fold=-1)
```

[15]:

[16]:
```
from qiskit_aer import AerSimulator

number_of_shots = 8000
aer_simulator = AerSimulator(method="statevector", shots=number_of_shots)

# Decompose the circuit such that the gates can be simulated with the
# AerSimulator
qaoa_with_parameters_decomposed = qaoa_with_parameters.decompose(reps=3)

result = aer_simulator.run(qaoa_with_parameters_decomposed).result()
```
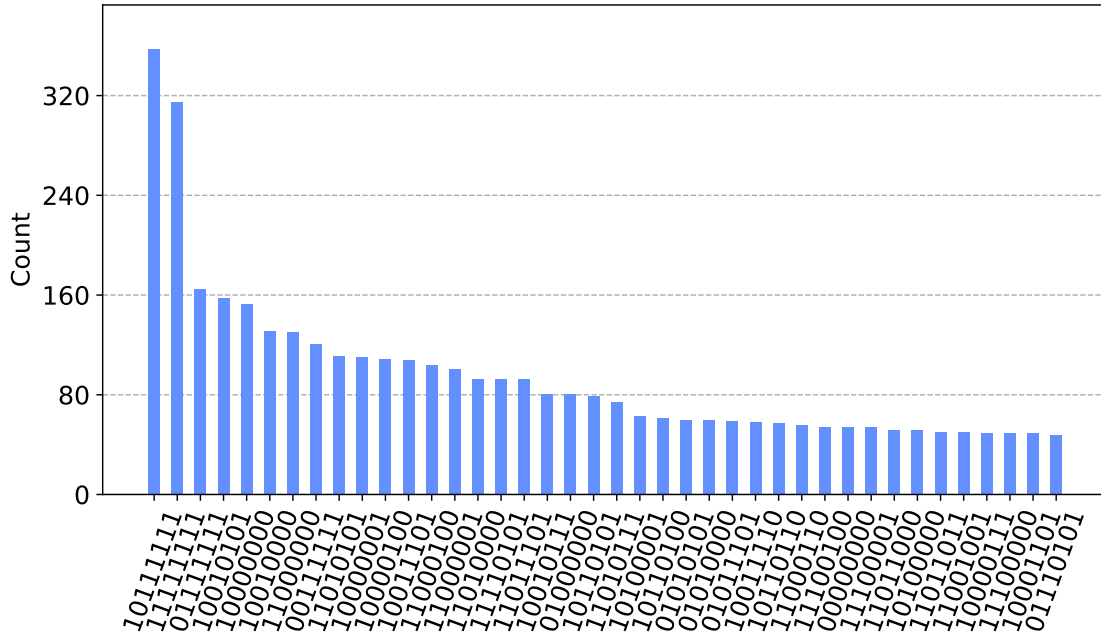
[17]:
```
from qiskit.visualization import plot_histogram

counts = result.get_counts()

# Plot the 40 bitstrings with the highest count.
plot_histogram(
    counts,
    number_to_keep=40,
    sort="value_desc",
    bar_labels=False,
    figsize=(10,5))
```

[17]:

Next, let us calculate the expectation value for our result and for the bitstring with the highest count/probability.

```python
sqrt_probabilities = {bitstring: np.sqrt(count/number_of_shots) \
                         for bitstring, count in counts.items()}


expectation = np.real(observable.eval(sqrt_probabilities)) \
                + ising_offset

print(f"e({betas}, {gammas}) = ", expectation)
```

```
e([1.23, 2.31], [3.21, 4.32]) =  39.16945000000001
```

```python
bitstring_highest_prob = max(
    sqrt_probabilities, key=lambda key: sqrt_probabilities[key])

print("bitstring with highest probability:", bitstring_highest_prob)
```

```
bitstring with highest probability: 10111111
```

```python
expectation = np.real(observable.eval(bitstring_highest_prob)) \
                + ising_offset

print(f"<{bitstring_highest_prob} | H_P | {bitstring_highest_prob}> = "
    f"{expectation}")
```

```
<10111111 | H_P | 10111111> = 121.0
```

Remembering from above that the minimum value of f_3 (and of f_1) is 6.0, we see that the solution of QAOA is not good. A main reason for this is that our parameters `betas` and `gammas` were not optimized. We do this in the next section.

2.7 QAOA: Classical Optimization

First, we need to write a function that has as input numerical values for the QAOA parameters $\vec{\beta}, \vec{\gamma}$ and computes the expectation value $e(\vec{\beta}, \vec{\gamma})$, see (QAOA-2), as output. We name this function `energy_evaluation` to be consistent with other Qiskit implementations, most importantly `qiskit.algorithms.QAOA`. The term energy comes from applications of the VQE algorithm and means the same as expectation. It is *not* connected to the term energy as used in our electric vehicle use case, where it refers to the energy the electric cars need to charge.

```
[22]:  from typing import List


def energy_evaluation(parameter_values: List[float]):
    """parameter_values is expected to be of
    the form [beta_0, beta_1, ..., gamma_0, gamma_1, ...]"""
    parameter_bindings = dict(zip(qaoa_ansatz.parameters, parameter_values))

    qaoa_with_parameters = qaoa_ansatz.bind_parameters(parameter_bindings)
    qaoa_with_parameters_decomposed = qaoa_with_parameters.decompose(reps=3)

    result = aer_simulator.run(qaoa_with_parameters_decomposed).result()
    counts = result.get_counts()
    sqrt_probabilities = {bitstring: np.sqrt(count/number_of_shots) \
                          for bitstring, count in counts.items()}

    expectation = np.real(observable.eval(sqrt_probabilities)) + ising_offset


    return expectation
```

We test our implementation with `betas` and `gammas` from Section 2.6.

```
[22]:  energy_evaluation([*betas, *gammas])
```

```
[22]:  39.09017500000002
```

We see that we get nearly the same result as above. The small difference is due to statistical errors stemming from the fact that from a finite number of shots (see `number_shots` in the code) the expectation value $\langle\psi|\mathcal{H}_P|\psi\rangle$ cannot be computed exactly. This effect is also known as **shot noise** [12].

Having this function we can use a classical optimizer, as e.g. provided in `qiskit.algorithms.optimizers`. We will use the COBYLA optimizer and random initial guesses for our parameters. These can be implemented e.g. with NumPy's default random generator.

32

```
[23]:  # Run this cell only once if you want changing random numbers in the cell below
       random_generator = np.random.default_rng(1234)
```

```
[24]:  from qiskit.algorithms.optimizers import COBYLA

       betas_initial_guess = np.pi*random_generator.random(qaoa_reps)
       gammas_initial_guess = 2*np.pi*random_generator.random(qaoa_reps)
       parameter_values_initial_guess = [*betas_initial_guess, *gammas_initial_guess]

       cobyla_optimizer = COBYLA()

       result_optimization = cobyla_optimizer.minimize(
           fun=energy_evaluation, x0=parameter_values_initial_guess)

       parameter_values_optimized = result_optimization.x
       energy_optimized = result_optimization.fun
       # Number of evalutions of energy_evaluation
       number_function_evaluations = result_optimization.nfev

       print(f"Optimized parameters: {parameter_values_optimized}")
       print(f"Expectation value: {energy_optimized}")
       print(f"Number function evaluations: {number_function_evaluations}")
```

```
Optimized parameters: [3.18220113 1.19369043 5.75551235 1.65220983]
Expectation value: 19.25527500000001
Number function evaluations: 45
```

Let's run the QAOA circuit with the optimized parameters.

```
[25]:  parameter_bindings = dict(
           zip(qaoa_ansatz.parameters, parameter_values_optimized))

       qaoa_with_parameters = qaoa_ansatz.bind_parameters(parameter_bindings)
       qaoa_with_parameters_decomposed = qaoa_with_parameters.decompose(reps=3)

       result = aer_simulator.run(qaoa_with_parameters_decomposed).result()
       counts = result.get_counts()
```

```
[26]:  # Plot the 40 bitstrings with the highest count.
       plot_histogram(
           counts,
           number_to_keep=40,
           sort="value_desc",
           bar_labels=False,
           figsize=(10,5))
```

[26]:
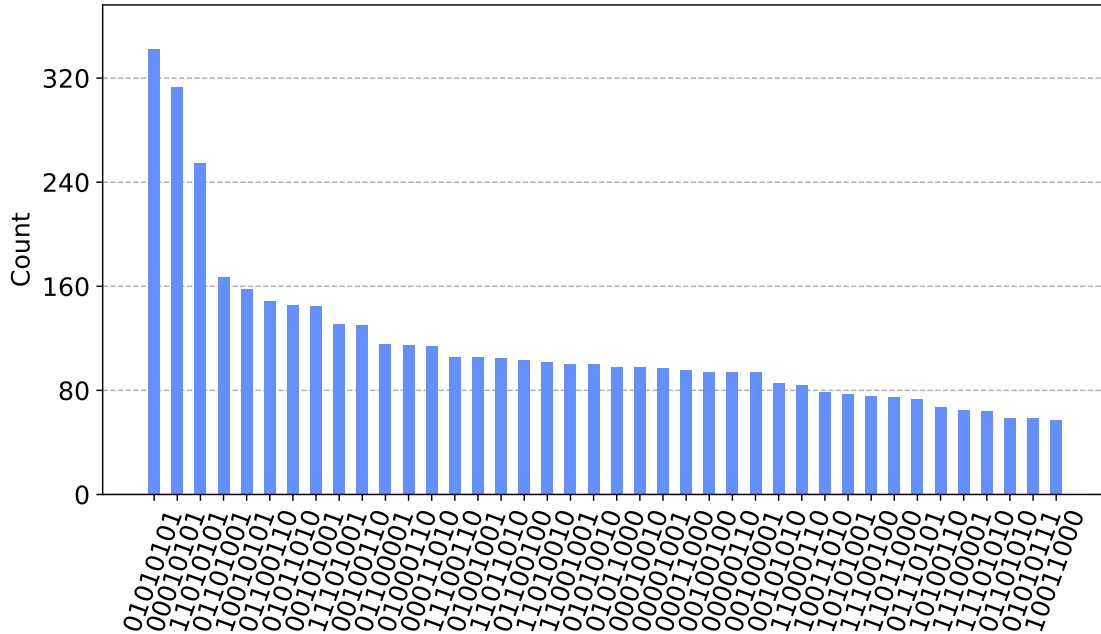
```
[28]: sqrt_probabilities = {bitstring: np.sqrt(count/number_of_shots) \
                             for bitstring, count in counts.items()}


      bitstring_highest_prob = max(
          sqrt_probabilities, key=lambda key: sqrt_probabilities[key])
      print(f"bitstring with highest probability: {bitstring_highest_prob}")


      expectation = np.real(observable.eval(bitstring_highest_prob)) \
                    + ising_offset
      print(f"<{bitstring_highest_prob} | H_P | {bitstring_highest_prob}> = "
            f"{expectation}")
```

```
bitstring with highest probability: 01010101
<01010101 | H_P | 01010101> = 7.6000000000000085
```

We see that the result is considerably improved compared with the non-optimized parameters from above. You can also rerun the optimization procedure so that other inital values are used.

2.8   (Optional) Advanced Knowledge: Gate Synthesis

In the remaining part of the notebook we discuss how (QAOA-1) can be implemented, i.e. which gates are needed to synthesize the unitary operations and verify that Qiskit's implementation of QAOA indeed uses theses gates. This will become technical and you can skip to the next notebook without problems.

The calculations below rely on the following identities:

1) For *commuting* matrices A and B, i.e. $[A, B] = AB - BA = 0$, we have

$$e^{A+B} = e^A e^B = e^B e^A.$$

2) For a Pauli-string $S = P_1 \otimes P_2 \otimes \cdots \otimes P_n$, $P_j \in \{I, X, Y, Z\}$, we have

$$\exp(i\theta S) = \cos(\theta) I^{\otimes n} + i \sin(\theta) S .$$

In particular, this means

$$\exp(i\theta \sigma_P^{(k)}) = I^{\otimes k} \otimes \exp(i\theta P) \otimes I^{\otimes(n-k-1)}, \qquad P \in \{X, Y, Z\}.$$

Note that e^A and $\exp(A)$ are the same function and we use the form that gives the clearest notation.

2.8.1 Gates for Uniform Superposition

The initial state $|+\rangle^{\otimes n}$ can easily be obtained by

$$|+\rangle^{\otimes n} = H^{\otimes n} |0\rangle^{\otimes n} = (\underbrace{H \otimes \cdots \otimes H}_{n \text{ times}}) |\underbrace{0 \ldots 0}_{n \text{ times}}\rangle, \qquad H = \tfrac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} .$$

Here, H is the Hadamard gate.

2.8.2 Gates for Mixing Operator

For the mixing operator we have

$$
\begin{aligned}
\mathcal{U}_M(\beta) &= \exp(-i\beta \mathcal{H}_M) \\
&= \exp\!\left(-i\beta \sum_k \sigma_X^{(k)}\right) \\
&= \prod_k \exp\!\left(-i\beta \sigma_X^{(k)}\right) \\
&= \left(\exp(-i\beta X) \otimes I \otimes \cdots \otimes I\right) \cdots \left(I \otimes \ldots I \otimes \exp(-i\beta X)\right)
\end{aligned}
$$

Recall that the RX gate is given by

$$\mathrm{RX}(\theta) = \exp(-i\tfrac{\theta}{2} X) .$$

This means the RX gate applied on qubit k is

$$\mathrm{RX}_k(\theta) = I^{\otimes k} \otimes \exp(-i\tfrac{\theta}{2} X) \otimes I^{n-k-1} .$$

Thus, the mixing operator can be implemented as

$$\mathcal{U}_M(\beta) = \mathrm{RX}_0(2\beta) \mathrm{RX}_1(2\beta) \cdots \mathrm{RX}_{n-1}(2\beta) .$$

Note that since every rotation acts on a different qubit all gates can be implemented in parallel.

2.8.3 Gates for Phase Operator

For the phase operator we have

$$
\begin{aligned}
\mathcal{U}_P(\gamma) &= \exp(-\mathrm{i}\gamma\mathcal{H}_P) \\
&= \exp\!\Big(-\mathrm{i}\gamma\big(\sum_k\sum_{l>k} h_{kl}\sigma_Z^{(k)}\sigma_Z^{(l)} + \sum_k h_k'\sigma_Z^{(k)} + h''I^{\otimes n}\big)\Big) \\
&= \exp\!\Big(-\mathrm{i}\gamma\sum_k\sum_{l>k} h_{kl}\sigma_Z^{(k)}\sigma_Z^{(l)}\Big)\exp\!\Big(-\mathrm{i}\gamma\sum_k h_k'\sigma_Z^{(k)}\Big)\exp\!\Big(-\mathrm{i}\gamma h''I^{\otimes n}\Big) .
\end{aligned}
$$

The third factor on the RHS is just a multiplication with a global phase $\mathrm{e}^{-\mathrm{i}\gamma h''}$. Since this part is not included in Qiskit's `ising` (see `ising_offset` above) we ignore it for the remaining section.

By the same computations as for the mixing operator the second factor can be written as

$$
\exp\!\Big(-\mathrm{i}\gamma\sum_k h_k'\sigma_Z^{(k)}\Big) = \mathrm{RZ}_0(2\gamma h_0')\mathrm{RZ}_1(2\gamma h_1')\cdots\mathrm{RZ}_{n-1}(2\gamma h_{n-1}') ,
$$

where the RZ gate is given by

$$
\mathrm{RZ}(\theta) = \exp(-\mathrm{i}\tfrac{\theta}{2}Z) .
$$

Note that contrary to the mixing operator now the **rotation angle also depends on the coefficients** h_k'.

For the first factor we have

$$
\begin{aligned}
\exp\!\Big(-\mathrm{i}\gamma\sum_k\sum_{l>k} h_{kl}\sigma_Z^{(k)}\sigma_Z^{(l)}\Big) &= \prod_k\prod_{l>k}\exp(-\mathrm{i}\gamma h_{kl}\sigma_Z^{(k)}\sigma_Z^{(l)}) \\
&= \prod_k\prod_{l>k}\mathrm{RZZ}_{k,l}(2\gamma h_{kl}) \\
&= \mathrm{RZZ}_{0,1}(2\gamma h_{01})\mathrm{RZZ}_{0,2}(2\gamma h_{02})\cdots\mathrm{RZZ}_{n-2,n-1}(2\gamma h_{n-2,n-1}) ,
\end{aligned}
$$

since

$$
\sigma_Z^{(k)}\sigma_Z^{(l)} = I^{\otimes k}\otimes Z\otimes I^{\otimes(l-k-1)}\otimes Z\otimes I^{\otimes(n-l-1)} ,
$$

and since the RZZ gate is defined by

$$
\mathrm{RZZ}(\theta) = \exp\!\big(-\mathrm{i}\tfrac{\theta}{2}Z\otimes Z\big) .
$$

Again note that the **rotation angle not only depends on** γ **but also on the coeffiecients** h_{kl}.

2.8.4 Implementation

Let us verify that indeed we get a quantum circuit with the gates discussed in the previous three sections. To see this we have to `decompose qaoa_ansatz` a different number of times. For simplicity we now choose $p = 1$.

```
[28]: qaoa_ansatz = QAOAAnsatz(cost_operator=ising, reps=1, name='qaoa')
```
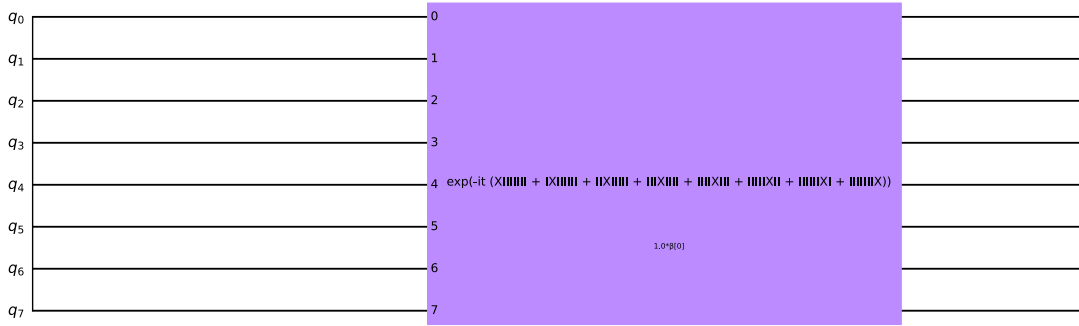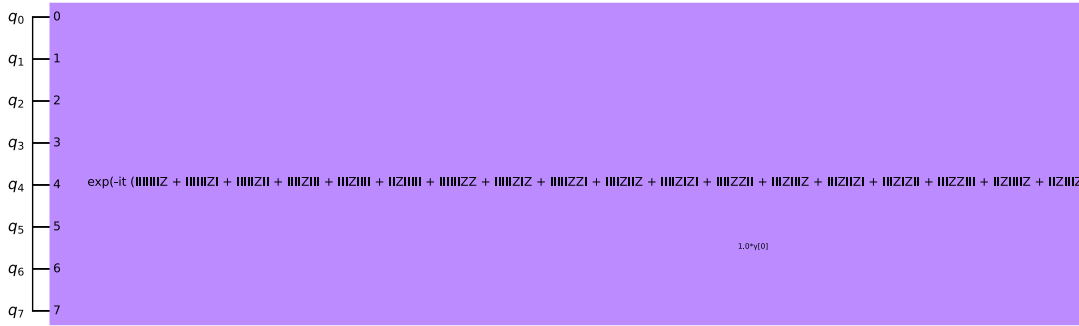
Decompose once

With this decomposition we see that

$$|\psi_{\text{QAOA}}\rangle = \mathcal{U}_M(\beta_0)\mathcal{U}_P(\gamma_0)\,|+\rangle^{\otimes n}$$

$$= \exp\left(-\mathrm{i}\beta_0 \sum_k \sigma_X^{(k)}\right) \exp\left(-\mathrm{i}\gamma_0\big(\sum_k \sum_{l>k} h_{kl}\sigma_Z^{(k)}\sigma_Z^{(l)} + \sum_k h'_k\sigma_Z^{(k)}\big)\right) H^{\otimes n}\,|0\rangle^{\otimes n}\ .$$

Recall: Mathematical formulas are read right to left whereas quantum circuits are read left to right.

```
[29]: qaoa_ansatz.decompose(reps=1).draw(scale=0.5, fold=-1)
```

[29]:

q_0 — H

q_1 — H

q_2 — H

q_3 — H

q_4 — H

q_5 — H

q_6 — H

q_7 — H

q_0 — 0

q_1 — 1

q_2 — 2

q_3 — 3

q_4 — 4 exp(-it (IIIIIIZ + IIIIIZI + IIIIIZII + IIIIZIII + IIIZIIII + IIZIIIII + IIIIIZZ + IIIIIZIZ + IIIIIZZI + IIIIZIIZ + IIIIZIZI + IIIIZZII + IIIZIIIZ + IIIZIIZI + IIIZIZII + IIIZZIII + IIZIIIIZ + IIZIIIZI

q_5 — 5

q_6 — 6   1.0*γ[0]

q_7 — 7

q_0 — 0

q_1 — 1

q_2 — 2

q_3 — 3

q_4 — 4 exp(-it (XIIIIII + IXIIIII + IIXIIIII + IIIXIIII + IIIIXIII + IIIIIXII + IIIIIIXI + IIIIIIIX))

q_5 — 5

q_6 — 6   1.0*β[0]
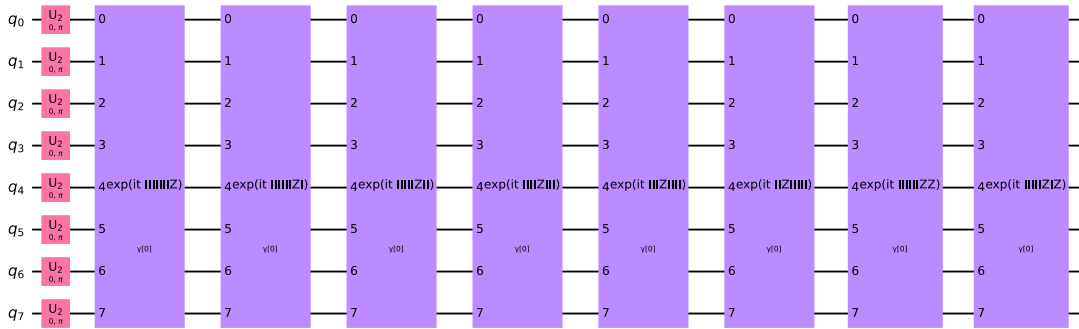
q_7 — 7

Decompose two times

Here we see

$$\mathcal{U}_M(\beta_0) = \prod_k \exp\big(-\mathrm{i}\beta_0\sigma_X^{(k)}\big) \; ,$$

and

$$\mathcal{U}_P(\gamma_0) = \prod_k \prod_{l>k} \exp\big(-\mathrm{i}\gamma_0 h_{kl}\sigma_Z^{(k)}\sigma_Z^{(l)}\big) \prod_k \exp\big(-\mathrm{i}\gamma_0 h_k'\sigma_Z^{(k)}\big) \; .$$

```
[30]: qaoa_ansatz.decompose(reps=2).draw(scale=0.5, fold=-1)
```
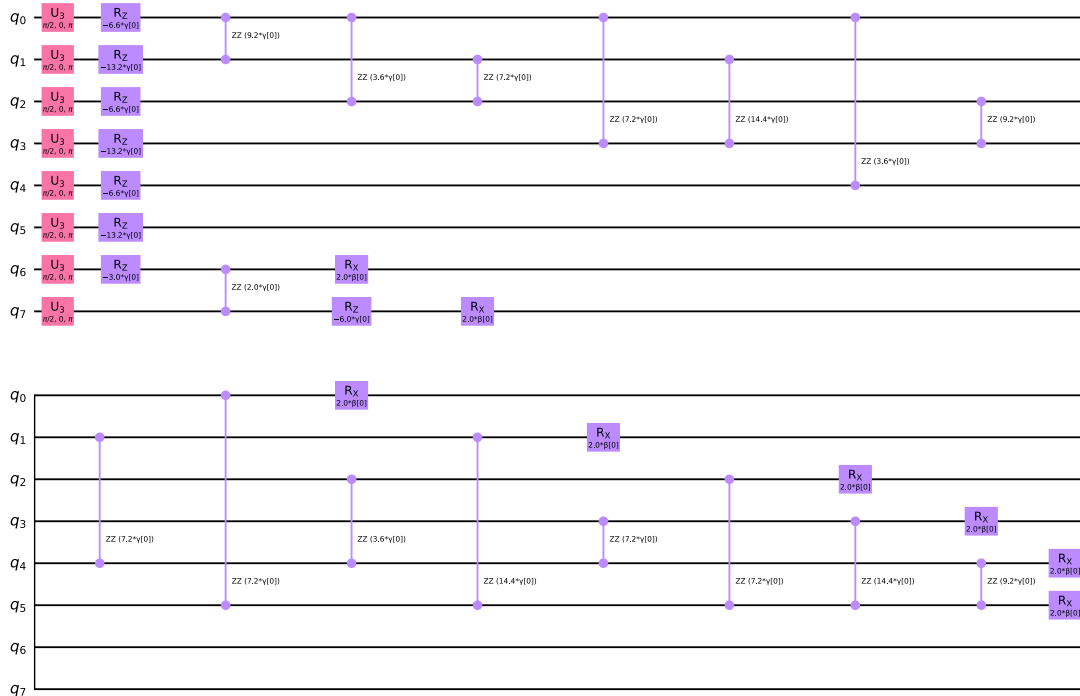
[30]:
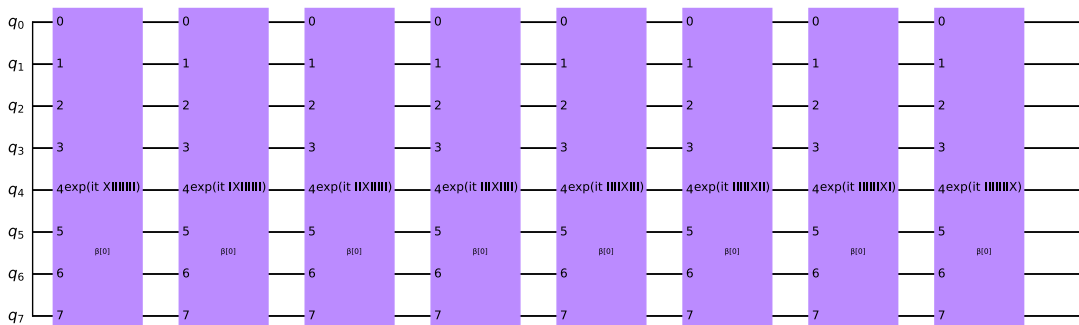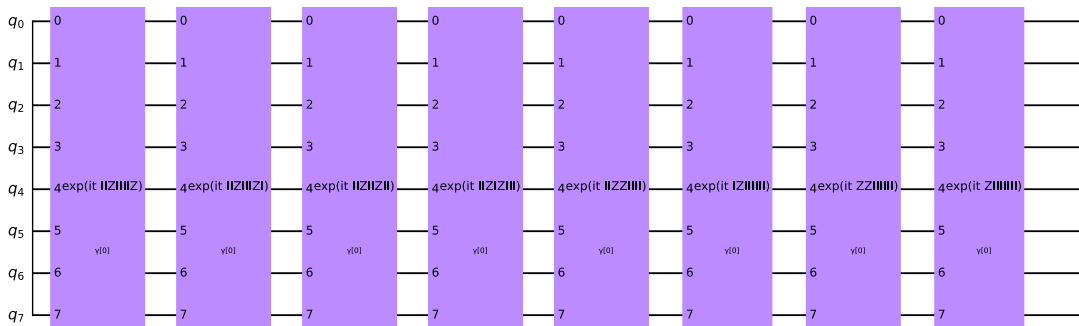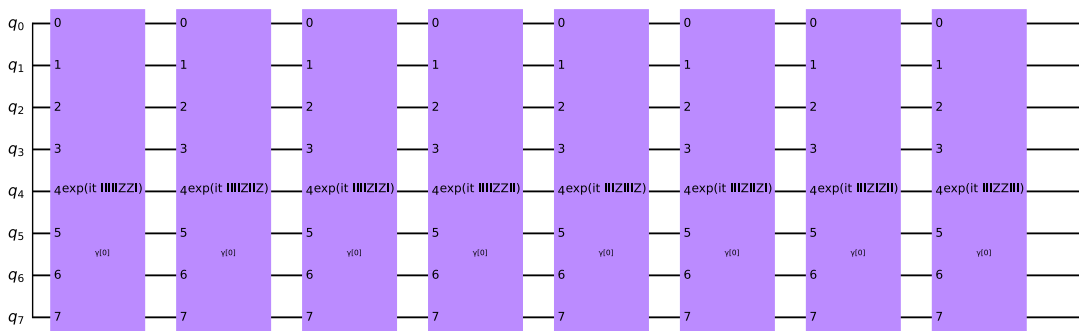
Decompose three times

With this decomposition we see

$$\mathcal{U}_M(\beta_0) = \mathrm{RX}_0(2\beta_0)\mathrm{RX}_1(2\beta_0)\cdots\mathrm{RX}_{n-1}(2\beta_0) \ ,$$

as well as

$$\mathcal{U}_P(\gamma_0) = \mathrm{RZZ}_{0,1}(2\gamma_0 h_{01})\mathrm{RZZ}_{0,2}(2\gamma_0 h_{02})\cdots\mathrm{RZZ}_{n-2,n-1}(2\gamma_0 h_{n-2,n-1})\,\mathrm{RZ}_0(2\gamma_0 h_0')\cdots\mathrm{RZ}_{n-1}(2\gamma_0 h_{n-1}') \ .$$

```
[31]: qaoa_ansatz.decompose(reps=3).draw(scale=0.5, fold=-1)
```

[31]:

Compare this also with the coefficient of `ising` (but recall the different ordering of Qiskit)

```
[32]: print(ising)
```

```
-3.3 * IIIIIIIZ
- 6.599999999999998 * IIIIIIZI
- 3.300000000000007 * IIIIIZII
- 6.6 * IIIIZIII
- 3.2999999999999994 * IIIZIIII
- 6.599999999999999 * IIZIIIII
```

```
+ 4.6 * IIIIIIZZ
+ 1.8 * IIIIIZIZ
+ 3.6 * IIIIIZZI
+ 3.6 * IIIIZIIZ
+ 7.2 * IIIIZIZI
+ 4.6 * IIIIZZII
+ 1.8 * IIIZIIIZ
+ 3.6 * IIIZIIZI
+ 1.8 * IIIZIZII
+ 3.6 * IIIZZIII
+ 3.6 * IIZIIIIZ
+ 7.2 * IIZIIIZI
+ 3.6 * IIZIIZII
+ 7.2 * IIZIZIII
+ 4.6 * IIZZIIII
- 1.5 * IZIIIIII
+ 1.0 * ZZIIIIII
- 3.0 * ZIIIIIII
```

Notebook 3

3.1 Introdution

In Notebook 1 we presented a real-world use case for optimizing charging schedules for electric cars, reduced it to a proof of concept model, and transformed it to a QUBO. Then, in Notebook 2 we presented the quantum algorithm QAOA and how the associated quantum circuits can be obtained. However, these circuits cannot directly be executed on a real quantum computer but they first need to be **transpiled**. In this notebook we will implement this process. Moreover, we will present how results from a quantum computer can be **post-processed**.

We will demonstrate these points with the same example as used in Notebook 2 enriched with the implementations that where derived there.

3.2 Example for this Notebook

```
[1]: from codes_notebook_2 import generate_example
     charging_unit, car_green, qcio, converter, qubo, \
         number_binary_variables, qubo_minimization_result, \
         ising, ising_offset, qaoa_reps, qaoa_circuit = generate_example()
```

```
[2]: print(charging_unit)
     print(car_green)
```

```
Charging unit with
  charging levels: 0, 1, 2, 3
  time slots: 0, 1, 2, 3
  cars to charge: car_green
Car 'car_green':
  at charging station at time slots [0, 1, 2]
  requires 4 energy units
```

```
[3]: b_min = qubo_minimization_result.x
     f_3_min = qubo_minimization_result.fval
     print("minimum point (binary): b_min = ", b_min)
     print("minimum value: f_3(b_min) = ", f_3_min)
```

```
minimum point (binary): b_min =  [0. 1. 1. 0. 1. 0. 0. 0.]
minimum value: f_3(b_min) =  5.999999999999993
```

```
[4]: p_min = converter.interpret(b_min)
     f_1_min = qcio.objective.evaluate(p_min)
     p_min_feasible = qcio.is_feasible(p_min)

     print("minimum point (integer): p_min = ", p_min)
     print("minimum value: f_1(p_min) = ", f_1_min)
     print("minimum point feasible = ", p_min_feasible)
```

```
minimum point (integer): p_min =  [2. 1. 1. 0.]
minimum value: f_1(p_min) =  6.0
minimum point feasible =  True
```

```
[5]: from utils import plot_charging_schedule
     fig = plot_charging_schedule(
         charging_unit, p_min, marker_size=30)
     fig.update_layout(width=350, height=300)
     fig.show()
```

```
[6]: print(ising)
```

```
-3.3 * IIIIIIIZ
- 6.599999999999998 * IIIIIIZI
- 3.3000000000000007 * IIIIIZII
- 6.6 * IIIIZIII
- 3.2999999999999994 * IIIZIIII
- 6.599999999999999 * IIZIIIII
+ 4.6 * IIIIIIZZ
+ 1.8 * IIIIIZIZ
+ 3.6 * IIIIIZZI
+ 3.6 * IIIIZIIZ
+ 7.2 * IIIIZIZI
+ 4.6 * IIIIZZII
+ 1.8 * IIIZIIIZ
+ 3.6 * IIIZIIZI
+ 1.8 * IIIZIZII
+ 3.6 * IIIZZIII
+ 3.6 * IIZIIIIZ
+ 7.2 * IIZIIIZI
+ 3.6 * IIZIIZII
+ 7.2 * IIZIZIII
+ 4.6 * IIZZIIII
- 1.5 * IZIIIIII
+ 1.0 * ZZIIIIII
- 3.0 * ZIIIIIII
```

3.3 Access to IBMQ

We start this notebook with the access to IBMQ systems. Note that this can (and probably will) change in the future and also depends on your membership to an IBMQ Hub. We therefore cannot guarantee that the following code cells work for everybody. A starting point what things need to be adapted can be found here or ask your local IBMQ Hub admin.

In order to access IBMQ systems one needs a personal token. For a more comfortable access we can save this token in a binary `pickle` file with the provided function `save_token`. After having saved your token you can load it with `load_token`.

If you **don't have access to real backends** you can skip the following four cells and use a **fake backend** as described next.

```python
from utils import save_token, load_token

# Note: For different URLs you might need different tokens.
# Note: You only have to do this once per token.
# save_token(
#     token="_your_token_",
#     file_name="_filename_for_your_saved_token_")

# Use the following line if you have saved your token:
# token = load_token("_filename_for_your_saved_token_")
# Otherwise, put your token in the following line:
token = "_your_token_"
```

```python
from qiskit import IBMQ
from qiskit.providers.ibmq.exceptions import IBMQAccountError, \
    IBMQProviderError
from qiskit.providers.ibmq.api.exceptions import RequestsApiError

# The following URL belongs to the Fraunhofer-DE hub:
api_url = "https://auth.de.quantum-computing.ibm.com/api"
# The following URL belongs to the US hub:
# api_url = "https://auth.quantum-computing.ibm.com/api"

try:
    IBMQ.enable_account(token, api_url)
except IBMQAccountError as e:
    if not (e.args[0] == "An IBM Quantum Experience account "
                         "is already in use for the session."):
        raise IBMQAccountError
except RequestsApiError as e:
    print(f"Error: {e.message}\n"
          "-> Check if the URL and your token are correct.")
```

```python
[ ]: from qiskit.providers.ibmq.exceptions import IBMQProviderError

try:
    # If you belong to the Fraunhofer Hub your parameters should
    # look likes this EXCEPT you need to adapat project='...':
    provider = IBMQ.get_provider(
        hub='fraunhofer-de',
        group='fhg-all',
        project='fiao01')
    # If you use the open access quantum systems your parameters
    # should look likes this:
    #    provider = IBMQ.get_provider(
    #        group='open',
    #        project='main')
except IBMQProviderError as e:
    print(f"Error: {e.message}\n"
          "-> Check your parameters in IBMQ.get_provider\n"
          "-> If you don't have access to IBMQ quantum systems "
          "use the fake backend below.")
```

```python
[ ]: from qiskit.providers.exceptions import QiskitBackendNotFoundError

# Depending on your access to real backends choose an appropriate backend here.
try:
    real_backend = provider.get_backend("ibmq_ehningen")
except NameError:
    print("Error: You first have to instantiate a provider object "
          "with the code in the cell above.")
except QiskitBackendNotFoundError as e:
    list_backends = [backend.name() for backend in provider.backends()]
    print(f"Error: backend not found. You only have access to: {list_backends}")
```

Fake backends are provided among others in `qiskit.providers.fake_provider`.

```python
[7]: from qiskit.providers.fake_provider import FakeKolkata
fake_kolkata = FakeKolkata()
```

In the next cell we decide which backend we want to use for the remaining notebook.

```python
[8]: # backend = real_backend
backend = fake_kolkata

number_shots = 8000
backend.set_options(shots=number_shots)
```

3.4 Transpilation Pipeline

It is important to know that **current quantum computers** (still) suffer from

- a limited set of gates that they can execute (see basis gates below),
- noisy qubits and erroneous gates (see x, sx and cx errors below),
- limited connectivity between individual qubits (see coupling map below),
- measurement errors (see error map below), and
- many other shortcomings.

Among others the task of a **transpilation** is to address these issues. Before we build a transpilation pipeline let us shortly visualize these problems.

3.4.1 Analysis of Quantum Backend

In the following code cells we access and visualize different properties of our (fake) quantum backend. If you don't have access to a real quantum computer you can look at the plots for the system ibmq_ehningen in Appendix A.
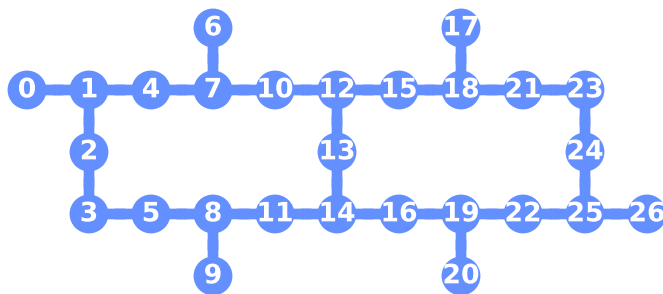
```
[9]: print(f"Basis gates of {backend.name()}: "
           f"{backend.configuration().basis_gates}")
```

Basis gates of fake_kolkata: ['id', 'rz', 'sx', 'x', 'cx', 'reset']

```
[10]: from qiskit.visualization import plot_gate_map

      print("Coupling map of", backend.name())
      plot_gate_map(backend)
```

Coupling map of fake_kolkata

[10]:


```
[11]: backend.configuration().coupling_map
```

```
[11]: [[0, 1],
       [1, 0],
```

```
    [1, 2],
    [1, 4],
    [2, 1],
    [2, 3],
    [3, 2],
    [3, 5],
    [4, 1],
    ...
    [24, 25],
    [25, 22],
    [25, 24],
    [25, 26],
    [26, 25]]
```
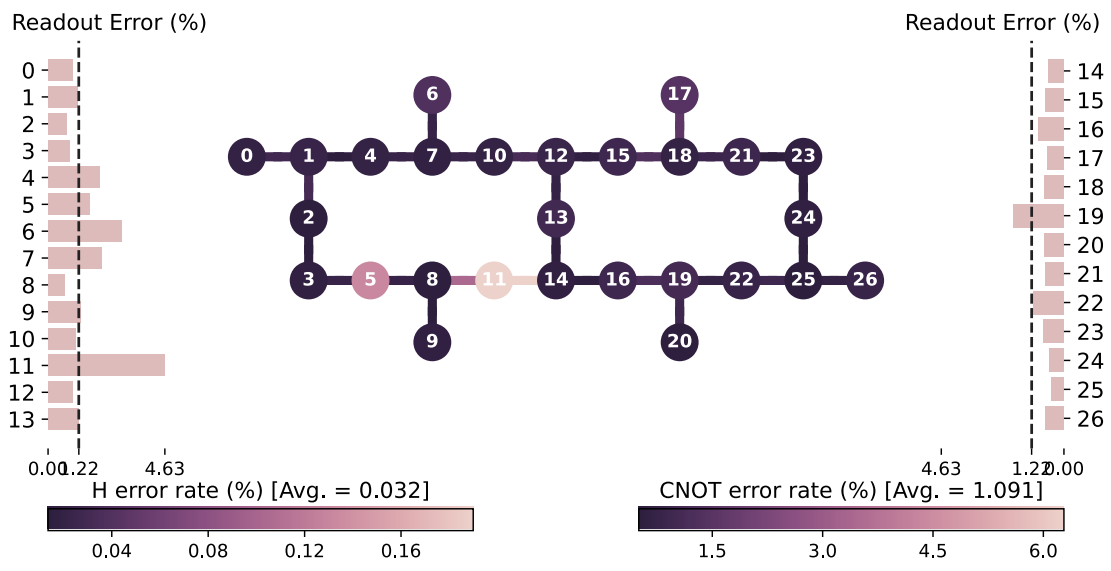
[12]:
```
from qiskit.visualization import plot_error_map

plot_error_map(backend, figsize=(10,6))
```

[12]:

fake_kolkata Error Map

[13]:
```
x_errors = []
sx_errors = []
cx_errors = []

backend_properties = backend.properties()

for qubit in range(backend.configuration().n_qubits):
```

```
        x_errors.append(backend_properties.gate_error("x", qubit))
        sx_errors.append(backend_properties.gate_error("sx", qubit))

for qubit0, qubit1 in backend.configuration().coupling_map:
        cx_errors.append(backend_properties.gate_error("cx", [qubit0, qubit1]))
```

[14]:
```python
import numpy as np
import plotly.graph_objects as go

fig = go.Figure()
fig.add_trace(go.Scatter(
    x=np.arange(backend.configuration().n_qubits),
    y=x_errors,
    mode="lines+markers",
    name="x error",
    marker_size=10
))
fig.add_trace(go.Scatter(
    x=np.arange(backend.configuration().n_qubits),
    y=sx_errors,
    mode="lines+markers",
    name="sx error",
    marker_symbol="x"
))
fig.update_xaxes(
    title="qubit")
fig.update_yaxes(
    title="error")
```
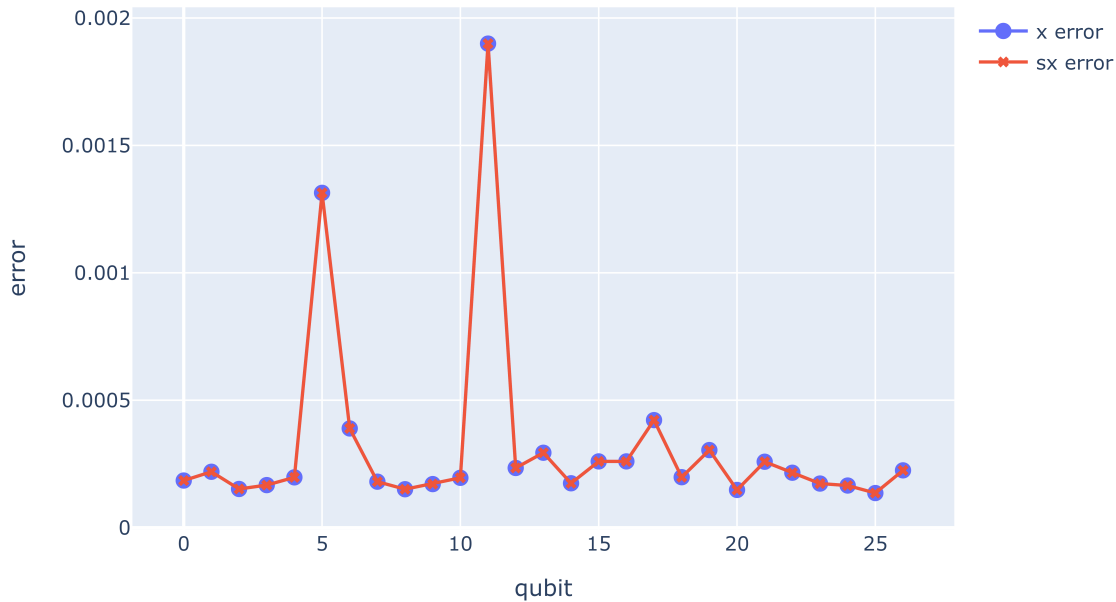
```
[15]: fig = go.Figure()
      fig.add_trace(go.Scatter(
          x=list(map(str, backend.configuration().coupling_map)),
          y=cx_errors,
          mode="lines+markers",
          name="CNOT error"
      ))
      fig.update_xaxes(
          title="[qubit0, qubit1]")
      fig.update_yaxes(
          title="CNOT error")
```
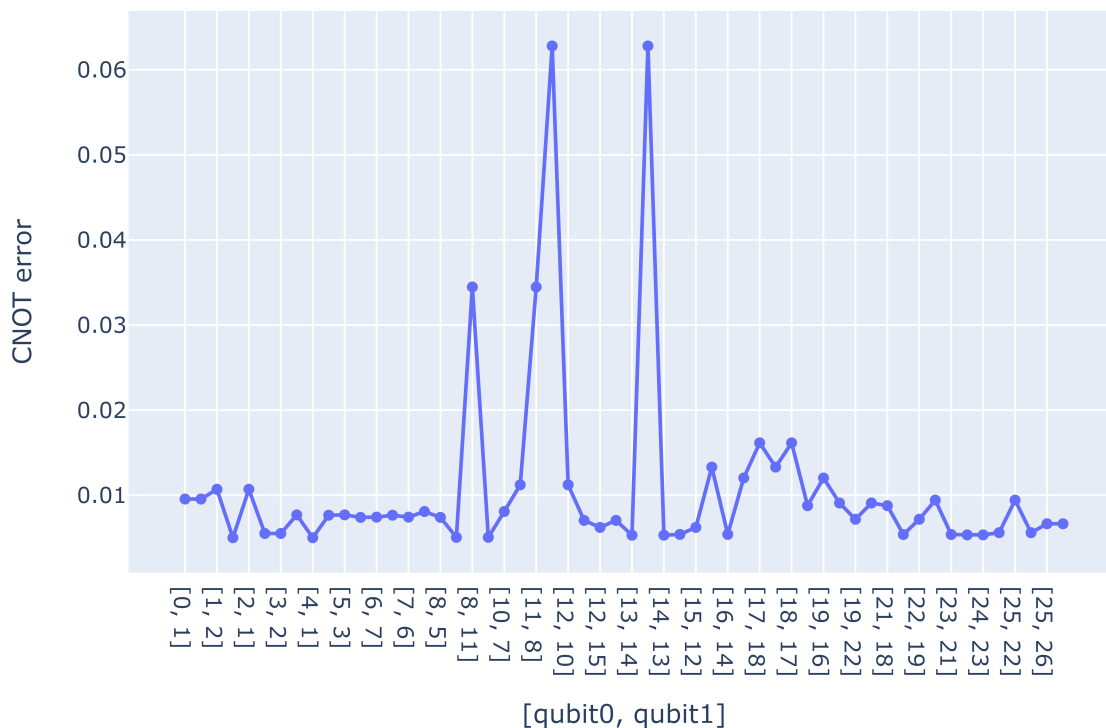

3.4.2 Standard Transpilation

In the following, we demonstrate how to build a transpilation pipeline based on the QAOA example from Notebook 2. Since the transpilation process in Qiskit is changed and improved frequently we only briefly sketch how a good transpilation pipeline looks like today (end of 2022). For future use [10, 21] could provide a state of the art pipeline.
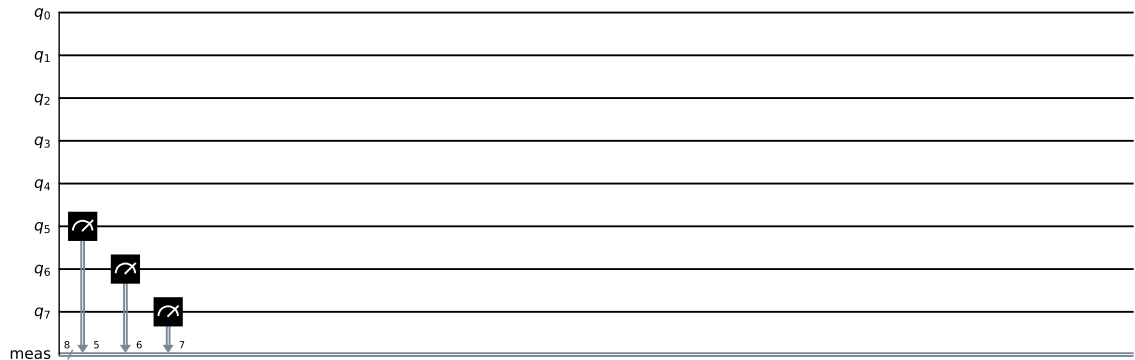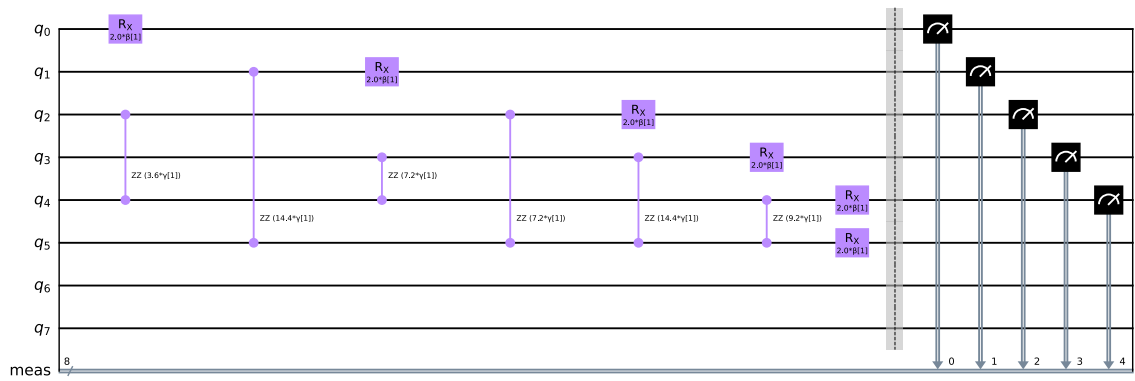
Let's begin by drawing the QAOA circuit.

```
[16]: qaoa_circuit.decompose(reps=3).draw(scale=1)
```

[16]:
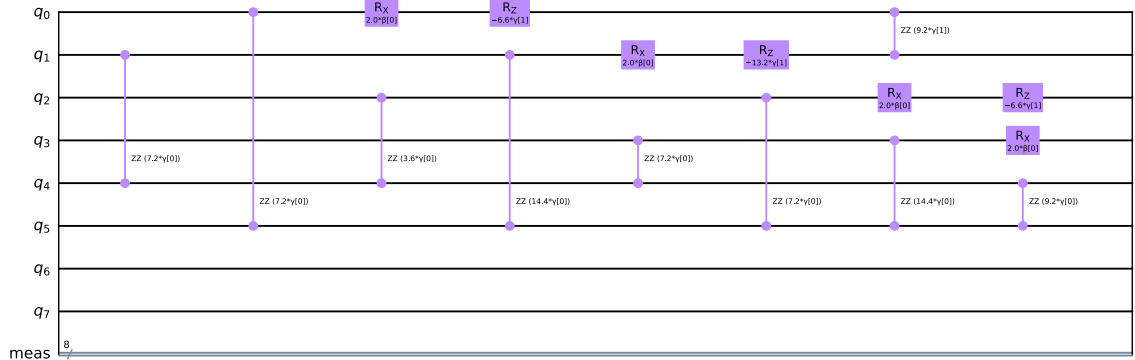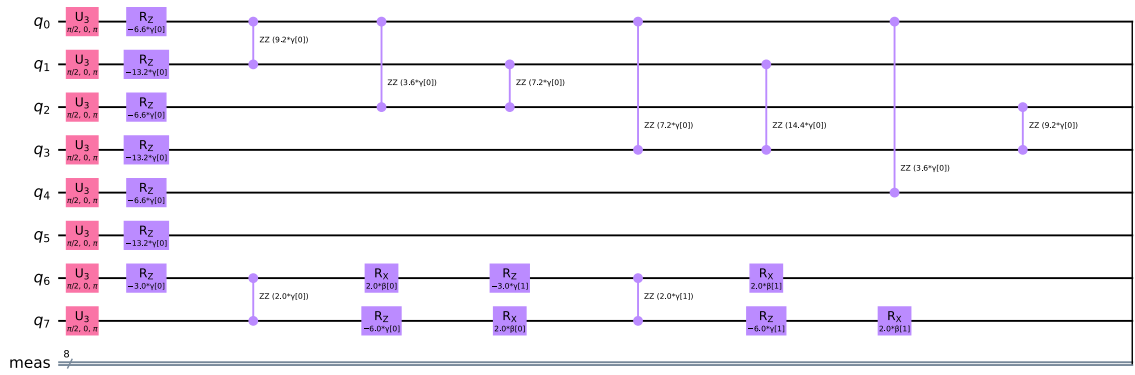
Now, we want to use **Qiskit's transpiler** to transpile this circuit. It is important to know that the transpiler has **stochastic components** which can be varied to obtain a good transpilation of a given quantum circuit. The stochastic components can be controlled with a **seed**. If we provide the transpiler with a list of seeds we will get a list of differently transpiled circuits.

```python
[17]: from qiskit.compiler import transpile

      seeds_for_transpiler = [k for k in range(40)]
      number_seeds_for_transpiler = len(seeds_for_transpiler)

      qaoa_circuit_transpilations = transpile(
          [qaoa_circuit]*number_seeds_for_transpiler,
          backend=backend,
          optimization_level=3,
          seed_transpiler=seeds_for_transpiler)

      print(f"We have {len(qaoa_circuit_transpilations)} "
            "different transpilations of the QAOA circuit.")
```
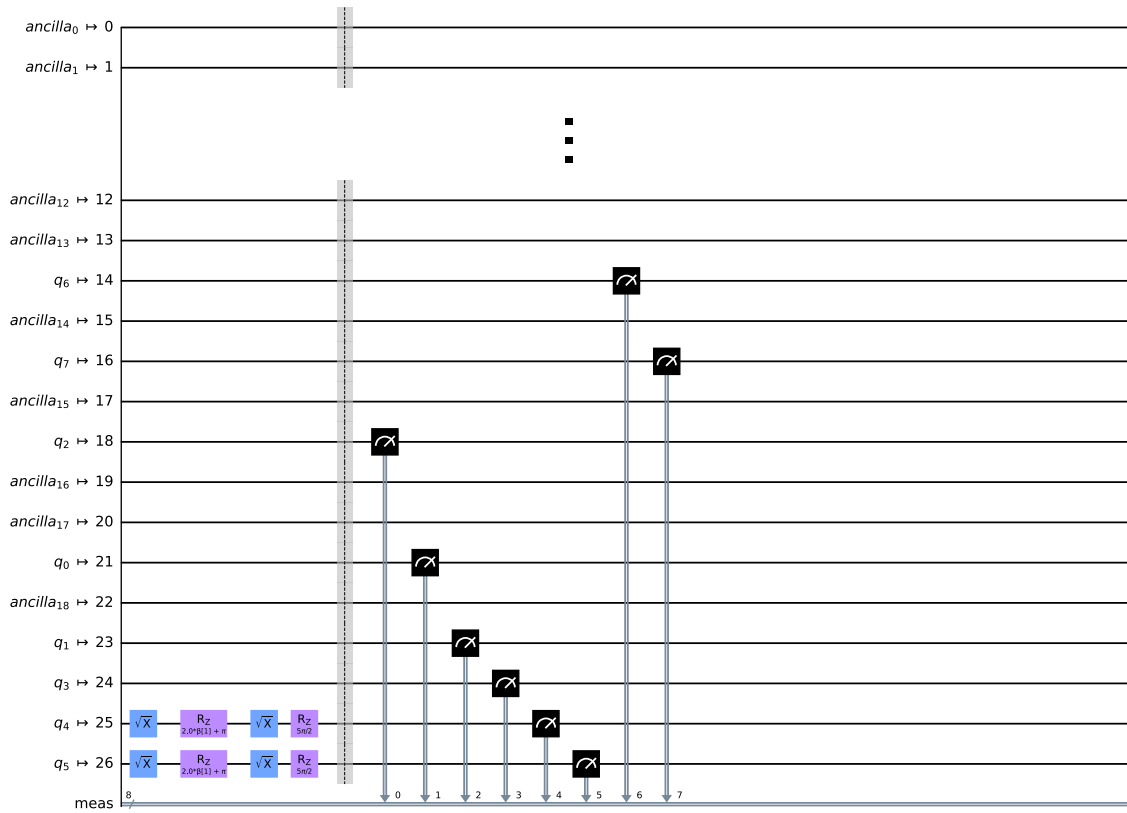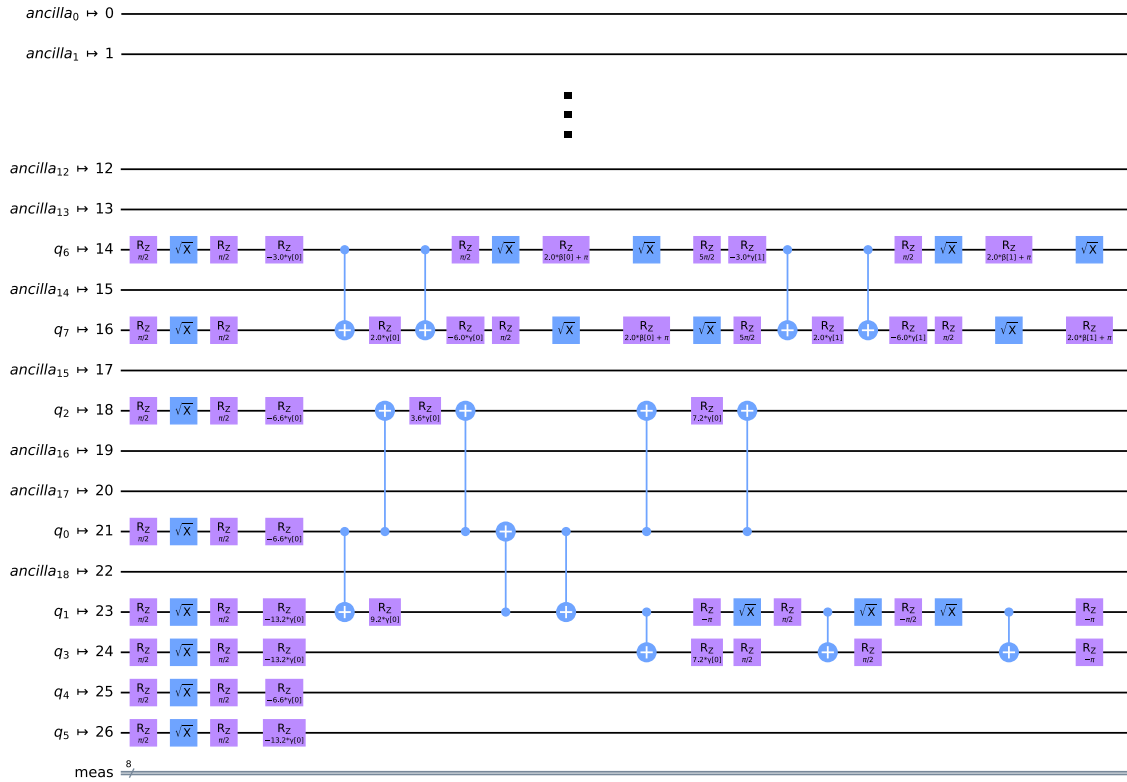
```
We have 40 different transpilations of the QAOA circuit.
```

```python
[18]: # Put here a number between 0 and number_seeds_for_transpiler
      number = 6

      qaoa_circuit_transpilations[number].draw()
```

[18]:

Clearly, now the question arises which circuit is the best. There are many metrics one can consider in order to answer this question. Probably the most simple one is to count the CNOT gates and take the circuit with the smallest count. The background for this metric is that the CNOT gate is the gate with highest error rate (see above).

```python
[19]: number_cnots = np.array([circuit.count_ops()["cx"]
                               for circuit in qaoa_circuit_transpilations])
      index_circuit_with_least_cnots = np.argmin(number_cnots)
      print(f"The circuit with index {index_circuit_with_least_cnots} "
            "has the least CNOTs.")
```

```
The circuit with index 12 has the least CNOTs.
```

```python
[20]: qaoa_circuit_best = \
          qaoa_circuit_transpilations[index_circuit_with_least_cnots]
```

Now that we have our best transpiled circuit we can apply some techniques to mitigate errors.

3.4.3 Lowering Decoherence and Dephasing via Dynamical Decoupling

Dynamical decoupling is a technique to lower decoherence and dephasing. In its simplest form we apply a X-X-gate sequence. For further details see [19].

```python
[21]: from qiskit.circuit.library import XGate
      from qiskit.transpiler import PassManager, InstructionDurations
      from qiskit.transpiler.passes import PadDynamicalDecoupling, ALAPScheduleAnalysis

      instruction_durations = InstructionDurations.from_backend(backend)
      dynamical_decoupling_sequence = [XGate(), XGate()]
      pulse_alignment = backend.configuration().timing_constraints["pulse_alignment"]
      pass_manager = PassManager([
          ALAPScheduleAnalysis(instruction_durations),
          PadDynamicalDecoupling(
              instruction_durations,
              dynamical_decoupling_sequence,
              pulse_alignment=pulse_alignment
          )
      ])

      qaoa_circuit_best = pass_manager.run(qaoa_circuit_best)
```

```python
[22]: qaoa_circuit_best.draw()
```

```
[22]:
```
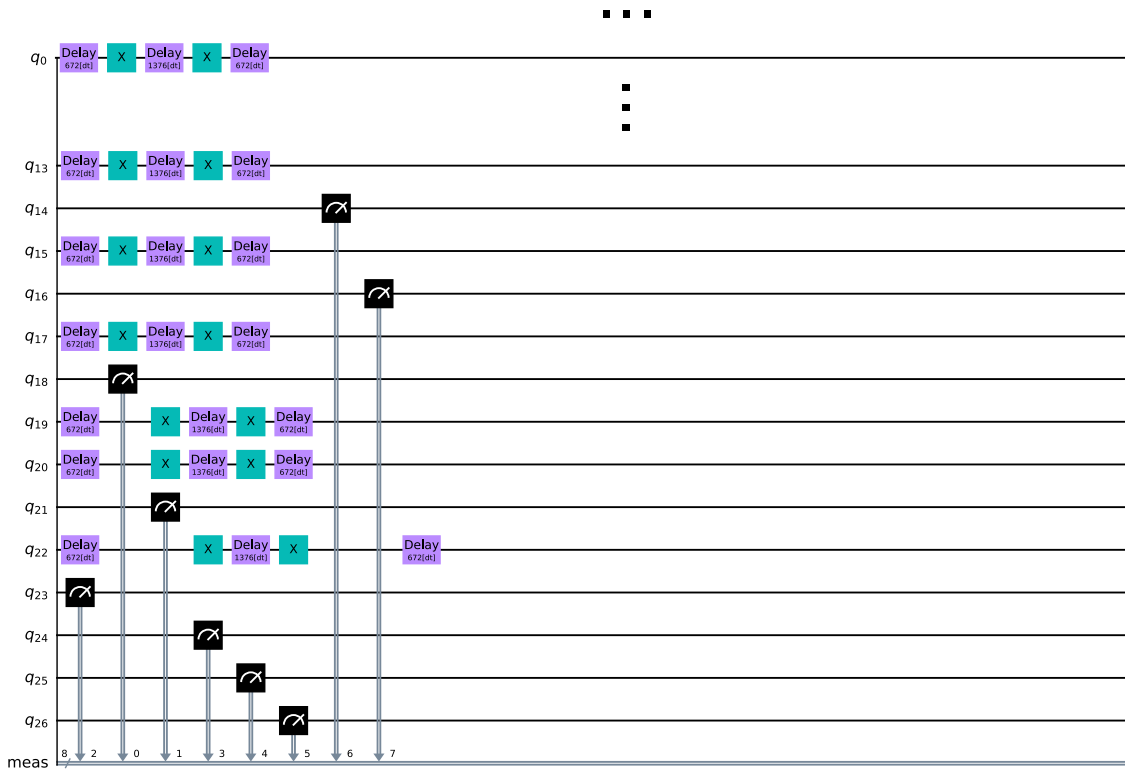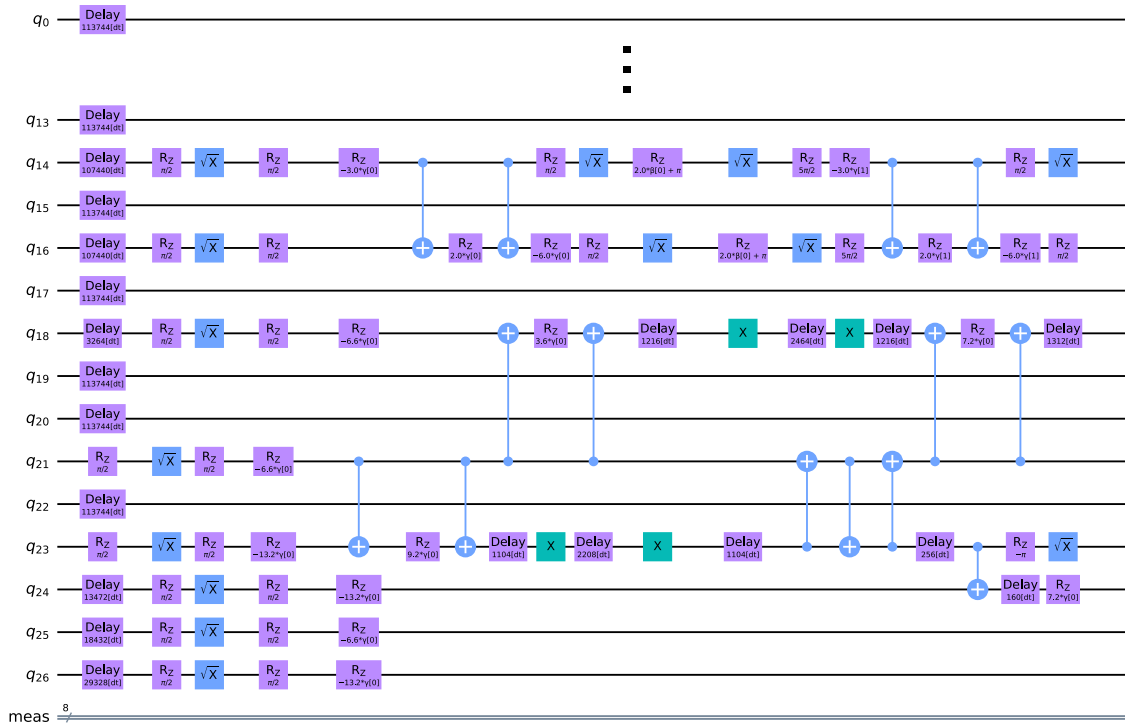
3.4.4 Measurement Error Mititgation

Measurement errors [12] can e.g. be mitigated with the M3 package [11, 20].

```
[23]: import mthree
      mit = mthree.M3Mitigation(backend)
      mit_final_measurement_mapping = \
          mthree.utils.final_measurement_mapping(qaoa_circuit_best)
```

3.4.5 Run Transpiled Circuit on Backend

Now let's run our circuit on the backend.

```
[24]: # The below provided values for the QAOA parameters
      # stem from an optimization as discussed in Notebook 2
      betas = [3.99890724, 2.72012026]
      gammas = [6.11303759, 1.75840967]
      qaoa_parameter_values = [*betas, *gammas]

      parameter_bindings = \
          dict(zip(qaoa_circuit_best.parameters, qaoa_parameter_values))
      qaoa_circuit_best_with_parameters = \
          qaoa_circuit_best.bind_parameters(parameter_bindings)
```

Note: If you have a real backend you will get in the **job queue** when executing the next cells. Instead you can also **proceed to Section 3.4.6** and work with the provided data.

```
[25]: # Run job to get calibration data for measurement error mitigation.
      # Ignore a possible warning.
      mit.cals_from_system(mit_final_measurement_mapping, async_cal=True)
```

```
[26]: # Run transpiled QAOA circuit
      job = backend.run(qaoa_circuit_best_with_parameters, job_name="qaoa_test_job")
      print("job is sent to backend")
      print("job_id: ", job.job_id())
```

```
      job is sent to backend
      job_id:   4cb2e677-b52a-4d20-a854-2ffac1905101
```

```
[27]: from qiskit_aer.jobs import AerJob

      # Write info about job into job_list.txt (if using a real backend)
      if not isinstance(job, AerJob):
          with open("job_list.txt", "at") as f:
              f.write(f"creation date: {job.creation_date()}, "
                  + f"backend: {backend.name()}, "
                  + f"job name: {job.name()}, "
                  + f"job id: {job.job_id()}\n")
```

```
[28]:  # Get measurement error mitigation matrices.
       mit_matrices = mit.cals_to_matrices()
```

```
[29]:  # Wait for jobs to finish.
       print("Status qaoa job:", job.status())
```

Status qaoa job: JobStatus.DONE

Save results

```
[30]:  from pathlib import Path
       import pickle
       from utils import convert_to_date_and_time_string

       assert job.done(), "Job is not finished."

       result = job.result()

       experiment_data = {
           "result": result,
           "qaoa_circuit_best_with_parameters": qaoa_circuit_best_with_parameters,
           "backend_properties": backend.properties(),
           "mit_matrices": mit_matrices,
           "mit_final_measurement_mapping": mit_final_measurement_mapping,
           # and all other meta data you need
       }

       # Add more data in the case of real backend.
       if not isinstance(job, AerJob):
           experiment_data["job_properties"] = job.properties()
           # and all other meta data you need

       result_time_stamp = convert_to_date_and_time_string(result.date)
       experiment_data_file_path = Path(
           f"{result_time_stamp}_"
           f"{backend.name()}_"
           "qaoa_experiment_data").with_suffix(".pickle")
       with open(experiment_data_file_path, "wb") as f:
           pickle.dump(experiment_data, f)
```

3.4.6 Postprocessing

In the rest of the notebook we are concerned with the processing of our result from the backend for which we will use the big data tool **pandas**. But first let us load our saved results.

```
[31]:  from pathlib import Path
       import pickle
```

```python
# Load your data ...
# experiment_data_file_path = \
#     Path("path_to_your_experiment_data").with_suffix(".pickle")
#
# or use the provided data from ibmq_ehningen ...
# experiment_data_file_path = \
#     Path("2022_09_22-14h47m_ibmq_ehningen_qaoa_experiment_data").with_suffix(".
# ↪pickle")
experiment_data_file_path = \
    Path("2022_09_23-08h42m_ibmq_ehningen_qaoa_experiment_data").with_suffix(".
# ↪pickle")
#
# or from fake_kolakata
# experiment_data_file_path = \
#     Path("2022_12_06-15h31m_fake_kolakata_qaoa_experiment_data").with_suffix(".
# ↪pickle")

with open(experiment_data_file_path, "rb") as f:
    experiment_data = pickle.load(f)

result = experiment_data["result"]
mit_matrices = experiment_data["mit_matrices"]
mit_final_measurement_mapping = experiment_data["mit_final_measurement_mapping"]
```

Let us retrieve the counts from our `result` object and put it in a `DataFrame` together with the respective bitstring. Then, let us add a column with the probabilities, which we can compute from the counts.

```python
[32]: import pandas as pd

      experiment_counts = result.get_counts()
      number_shots = sum(experiment_counts.values())

      # Create the DataFrame from experiment_counts.
      experiment_df = pd.DataFrame.from_dict(data={
          "bit_string": experiment_counts.keys(),
          "count": experiment_counts.values()})

      # Add column probability.
      experiment_df["probability"] = \
          experiment_df["count"]/number_shots

      experiment_df
```

```
[32]:    bit_string   count   probability
      0    00000000      49      0.006125
```

```
1       00000001      58      0.007250
2       00010000     112      0.014000
3       00010001     167      0.020875
4       00010010     130      0.016250
..           ...     ...           ...
249     11111011       6      0.000750
250     11111100       6      0.000750
251     11111101       7      0.000875
252     11111110      12      0.001500
253     11111111       4      0.000500

[254 rows x 3 columns]
```

Next, we create a `DataFrame` with the probabilities stemming from applying the measurement error mitigation to the counts from our experiment.

```
[33]: import mthree

      mit = mthree.M3Mitigation()
      mit.cals_from_matrices(mit_matrices)

      experiment_quasi_distribution = \
          mit.apply_correction(experiment_counts, mit_final_measurement_mapping)
      experiment_mitigated_probabilities = \
          experiment_quasi_distribution.nearest_probability_distribution()

      experiment_mitigated_df = pd.DataFrame.from_dict(data={
          "bit_string": experiment_mitigated_probabilities.keys(),
          "probability_mit": experiment_mitigated_probabilities.values()})

      experiment_mitigated_df
```

```
[33]:      bit_string  probability_mit
      0       01100001         0.000017
      1       10000000         0.000019
      2       10000101         0.000022
      3       01001010         0.000024
      4       01100100         0.000030
      ..           ...              ...
      246     00011000         0.025836
      247     00110110         0.026571
      248     00010110         0.029771
      249     00011110         0.034682
      250     00111110         0.034965

      [251 rows x 2 columns]
```

Now, let's merge our two `DataFrame`s and visualize the result.

```
[34]: experiment_df = experiment_df.merge(
          experiment_mitigated_df, how="outer", on="bit_string")
      experiment_df.fillna(0.0, inplace=True)

      experiment_df
```

```
[34]:       bit_string  count  probability  probability_mit
      0       00000000     49     0.006125         0.005830
      1       00000001     58     0.007250         0.007232
      2       00010000    112     0.014000         0.013703
      3       00010001    167     0.020875         0.022115
      4       00010010    130     0.016250         0.016376
      ..           ...    ...          ...              ...
      249     11111011      6     0.000750         0.000755
      250     11111100      6     0.000750         0.000704
      251     11111101      7     0.000875         0.000893
      252     11111110     12     0.001500         0.001518
      253     11111111      4     0.000500         0.000455

      [254 rows x 4 columns]
```
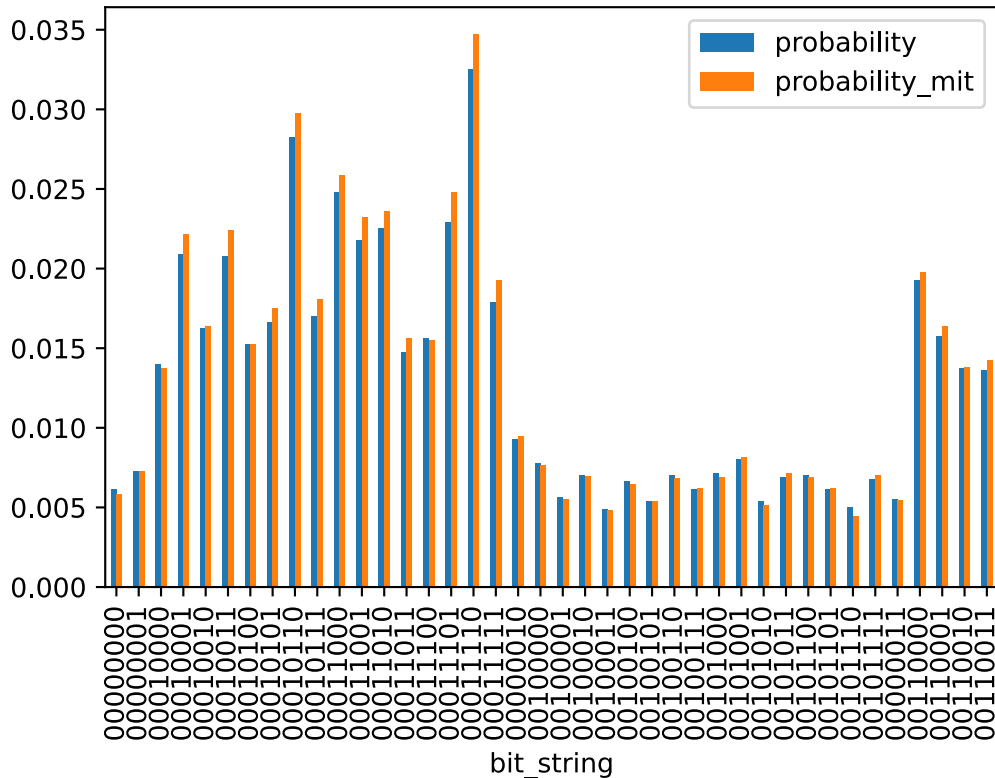
```
[35]: # Plot first 40 bit_strings and probability
      experiment_df.iloc[0:40,:].plot.bar(
          x="bit_string", y=["probability", "probability_mit"])
```

```
[35]: <AxesSubplot:xlabel='bit_string'>
```

For the remaining section let's focus on the 20 data with the highest probability. We first do some preparations, namely adding columns with `bit_array` and `integer_array`, so that we can apply `qubo.objective.evaluate` and `qcio.is_feasible` later (see Notebook 1, also note the different ordering conventions).

```
[36]: import numpy as np

experiment_df_top20 = \
    experiment_df.sort_values("probability", ascending=False).iloc[0:20, :]

experiment_df_top20["bit_array"] = \
    experiment_df_top20["bit_string"].apply(
        lambda bitstring: np.fromiter(bitstring, dtype=int)[::-1])

experiment_df_top20["integer_array"] = \
    experiment_df_top20["bit_array"].apply(converter.interpret)

# Change order of columns.
experiment_df_top20 = experiment_df_top20[
    ["bit_string", "bit_array","integer_array",
     "count", "probability", "probability_mit"]]
```

```
experiment_df_top20.reset_index(inplace=True, drop=True)

experiment_df_top20
```

[36]:   bit_string   bit_array   integer_array count \
     0   00111110 [0, 1, 1, 1, 1, 1, 0, 0] [2.0, 3.0, 3.0, 0.0]   261
     1   00011110 [0, 1, 1, 1, 1, 0, 0, 0] [2.0, 3.0, 1.0, 0.0]   260
     2   00010110 [0, 1, 1, 0, 1, 0, 0, 0] [2.0, 1.0, 1.0, 0.0]   226
     3   00110110 [0, 1, 1, 0, 1, 1, 0, 0] [2.0, 1.0, 3.0, 0.0]   204
     4   00011000 [0, 0, 0, 1, 1, 0, 0, 0] [0.0, 2.0, 1.0, 0.0]   198
     5   00011101 [1, 0, 1, 1, 1, 0, 0, 0] [1.0, 3.0, 1.0, 0.0]   183
     6   00011010 [0, 1, 0, 1, 1, 0, 0, 0] [2.0, 2.0, 1.0, 0.0]   180
     7   00011001 [1, 0, 0, 1, 1, 0, 0, 0] [1.0, 2.0, 1.0, 0.0]   174
     8   00010001 [1, 0, 0, 0, 1, 0, 0, 0] [1.0, 0.0, 1.0, 0.0]   167
     9   00010011 [1, 1, 0, 0, 1, 0, 0, 0] [3.0, 0.0, 1.0, 0.0]   166
     10   00110100 [0, 0, 1, 0, 1, 1, 0, 0] [0.0, 1.0, 3.0, 0.0]   159
     11   00110000 [0, 0, 0, 0, 1, 1, 0, 0] [0.0, 0.0, 3.0, 0.0]   154
     12   00111001 [1, 0, 0, 1, 1, 1, 0, 0] [1.0, 2.0, 3.0, 0.0]   152
     13   00111010 [0, 1, 0, 1, 1, 1, 0, 0] [2.0, 2.0, 3.0, 0.0]   151
     14   00011111 [1, 1, 1, 1, 1, 0, 0, 0] [3.0, 3.0, 1.0, 0.0]   143
     15   00111000 [0, 0, 0, 1, 1, 1, 0, 0] [0.0, 2.0, 3.0, 0.0]   140
     16   00010111 [1, 1, 1, 0, 1, 0, 0, 0] [3.0, 1.0, 1.0, 0.0]   136
     17   00111100 [0, 0, 1, 1, 1, 1, 0, 0] [0.0, 3.0, 3.0, 0.0]   134
     18   00110111 [1, 1, 1, 0, 1, 1, 0, 0] [3.0, 1.0, 3.0, 0.0]   133
     19   00010101 [1, 0, 1, 0, 1, 0, 0, 0] [1.0, 1.0, 1.0, 0.0]   133

         probability probability_mit
     0   0.032625   0.034965
     1   0.032500   0.034682
     2   0.028250   0.029771
     3   0.025500   0.026571
     4   0.024750   0.025836
     5   0.022875   0.024802
     6   0.022500   0.023610
     7   0.021750   0.023181
     8   0.020875   0.022115
     9   0.020750   0.022423
     10   0.019875   0.020464
     11   0.019250   0.019782
     12   0.019000   0.020227
     13   0.018875   0.019463
     14   0.017875   0.019263
     15   0.017500   0.017681
     16   0.017000   0.018043
     17   0.016750   0.017082
     18   0.016625   0.017772
     19   0.016625   0.017476

```
[37]:  experiment_df_top20.plot.bar(
           x="bit_string", y=["probability", "probability_mit"])
```

```
[37]:  <AxesSubplot:xlabel='bit_string'>
```
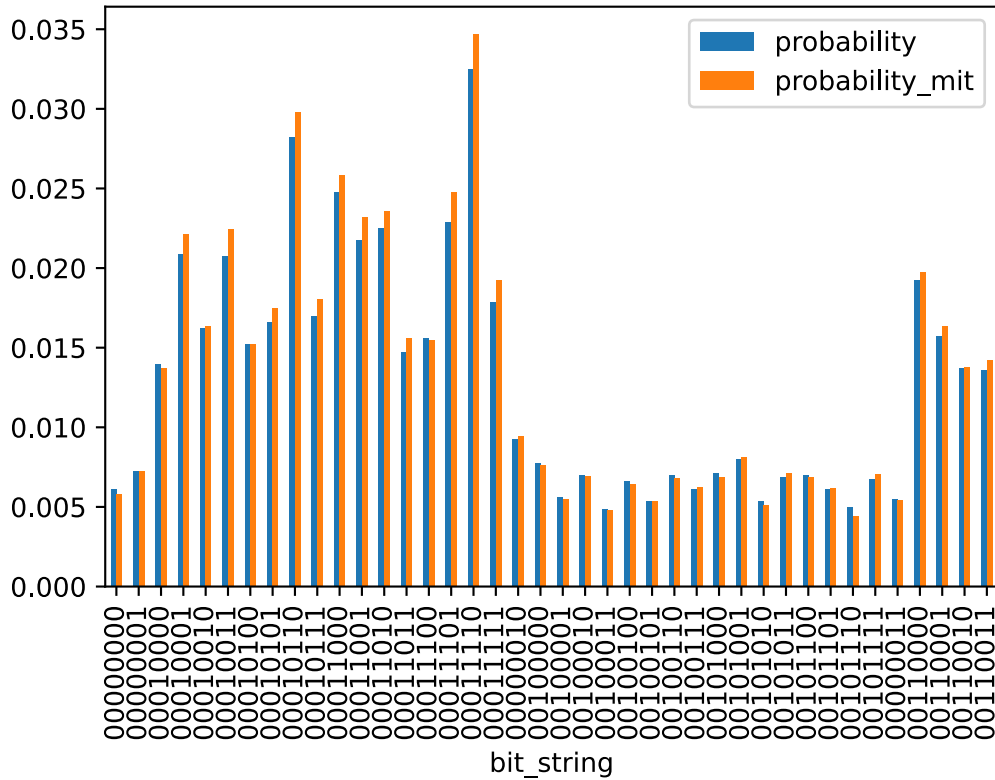

Since we only have 20 rows in our dataframe it is not too costly to apply `qubo.objective.evaluate` and `qcio.is_feasible`. This gives us a good overview what could be a good solution candidate (i.e. it is feasible and has low cost).

```
[38]:  experiment_df_top20["cost"] = \
           experiment_df_top20["bit_array"].apply(qubo.objective.evaluate)

       experiment_df_top20["is_feasible"] = \
           experiment_df_top20["integer_array"].apply(qcio.is_feasible)

       experiment_df_top20
```

```
[38]:     bit_string              bit_array        integer_array   count  \
       0    00111110  [0, 1, 1, 1, 1, 1, 0, 0]  [2.0, 3.0, 3.0, 0.0]    261
       1    00011110  [0, 1, 1, 1, 1, 0, 0, 0]  [2.0, 3.0, 1.0, 0.0]    260
       2    00010110  [0, 1, 1, 0, 1, 0, 0, 0]  [2.0, 1.0, 1.0, 0.0]    226
       3    00110110  [0, 1, 1, 0, 1, 1, 0, 0]  [2.0, 1.0, 3.0, 0.0]    204
```

```
4    00011000  [0, 0, 0, 1, 1, 0, 0, 0]  [0.0, 2.0, 1.0, 0.0]  198
5    00011101  [1, 0, 1, 1, 1, 0, 0, 0]  [1.0, 3.0, 1.0, 0.0]  183
6    00011010  [0, 1, 0, 1, 1, 0, 0, 0]  [2.0, 2.0, 1.0, 0.0]  180
7    00011001  [1, 0, 0, 1, 1, 0, 0, 0]  [1.0, 2.0, 1.0, 0.0]  174
8    00010001  [1, 0, 0, 0, 1, 0, 0, 0]  [1.0, 0.0, 1.0, 0.0]  167
9    00010011  [1, 1, 0, 0, 1, 0, 0, 0]  [3.0, 0.0, 1.0, 0.0]  166
10   00110100  [0, 0, 1, 0, 1, 1, 0, 0]  [0.0, 1.0, 3.0, 0.0]  159
11   00110000  [0, 0, 0, 0, 1, 1, 0, 0]  [0.0, 0.0, 3.0, 0.0]  154
12   00111001  [1, 0, 0, 1, 1, 1, 0, 0]  [1.0, 2.0, 3.0, 0.0]  152
13   00111010  [0, 1, 0, 1, 1, 1, 0, 0]  [2.0, 2.0, 3.0, 0.0]  151
14   00011111  [1, 1, 1, 1, 1, 0, 0, 0]  [3.0, 3.0, 1.0, 0.0]  143
15   00111000  [0, 0, 0, 1, 1, 1, 0, 0]  [0.0, 2.0, 3.0, 0.0]  140
16   00010111  [1, 1, 1, 0, 1, 0, 0, 0]  [3.0, 1.0, 1.0, 0.0]  136
17   00111100  [0, 0, 1, 1, 1, 1, 0, 0]  [0.0, 3.0, 3.0, 0.0]  134
18   00110111  [1, 1, 1, 0, 1, 1, 0, 0]  [3.0, 1.0, 3.0, 0.0]  133
19   00010101  [1, 0, 1, 0, 1, 0, 0, 0]  [1.0, 1.0, 1.0, 0.0]  133


     probability  probability_mit  cost  is_feasible
0       0.032625         0.034965  79.6        False
1       0.032500         0.034682  28.4        False
2       0.028250         0.029771   6.0         True
3       0.025500         0.026571  28.4        False
4       0.024750         0.025836   8.6        False
5       0.022875         0.024802  14.6        False
6       0.022500         0.023610  12.6        False
7       0.021750         0.023181   6.0         True
8       0.020875         0.022115  16.4        False
9       0.020750         0.022423  10.0         True
10      0.019875         0.020464  10.0         True
11      0.019250         0.019782  12.6        False
12      0.019000         0.020227  28.4        False
13      0.018875         0.019463  49.4        False
14      0.017875         0.019263  51.4        False
15      0.017500         0.017681  16.6        False
16      0.017000         0.018043  14.6        False
17      0.016750         0.017082  32.4        False
18      0.016625         0.017772  51.4        False
19      0.016625         0.017476   6.6        False
```

Let's plot two solutions from the dataframe above.

```
[39]:  row_number_1 = 2 # choose a number between 0 and 19
       row_number_2 = 6 # choose a number between 0 and 19
```
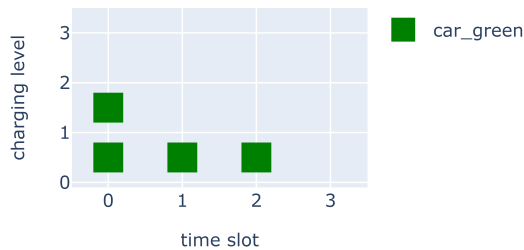
```
[40]:  fig = plot_charging_schedule(
           charging_unit,
           experiment_df_top20["integer_array"].iloc[row_number_1],
```

```
        marker_size=20)
fig.update_layout(width=400, height=300)
fig.show()
```



```
[41]: fig = plot_charging_schedule(
          charging_unit,
          experiment_df_top20["integer_array"].iloc[row_number_2],
          marker_size=20)
      fig.update_layout(width=400, height=300)
      fig.show()
```


3.4.7   Compare with Exact Simulation

In order to judge how good QAOA was executed on the real quantum computer we can compare it
with the result of an exact simulation (a statevector simulation without shot noise).

```
[42]: from qiskit.providers.aer import AerSimulator
      from qiskit.compiler import transpile

      aer_simulator = AerSimulator(method="statevector")

      qaoa_circuit_for_exact_simulation = qaoa_circuit.copy()
      qaoa_circuit_for_exact_simulation.remove_final_measurements()
      qaoa_circuit_for_exact_simulation.save_state()

      qaoa_circuit_for_exact_simulation = transpile(
          qaoa_circuit_for_exact_simulation,
          basis_gates=aer_simulator.configuration().basis_gates)

      parameter_bindings = dict(
          zip(qaoa_circuit_for_exact_simulation.parameters, qaoa_parameter_values))
      qaoa_circuit_for_exact_simulation = \
```

64

```
    qaoa_circuit_for_exact_simulation.bind_parameters(parameter_bindings)

job = aer_simulator.run(qaoa_circuit_for_exact_simulation)
result_exact_simulation = job.result()
```

Such a simulation has a `Statevetor` object as result.

[43]:
```
statevector_exact_simulation = result_exact_simulation.get_statevector()
print(statevector_exact_simulation)
```

```
Statevector([-2.19237435e-03+4.73527294e-05j,
              9.36596020e-03+4.63441187e-03j,
              2.20883527e-06+5.24193322e-04j,
              8.90791252e-03+6.83746374e-03j,
              9.36596020e-03+4.63441187e-03j,
              2.09261296e-02+1.38949582e-01j,
              5.14211808e-06-9.66312497e-04j,
             -4.59808754e-03+3.90829328e-02j,
              2.20883527e-06+5.24193322e-04j,
              ...
              8.24173563e-04-8.22140766e-04j,
             -5.44605429e-04-1.92611140e-03j,
              3.05595401e-03-2.21578751e-03j,
              8.24173563e-04-8.22140766e-04j,
              1.98750175e-03-1.13018901e-04j],
            dims=(2, 2, 2, 2, 2, 2, 2, 2))
```

Let's transform this into a `DictStateFn` to be compatible with the data type of the result from the real backend.

[44]:
```
from qiskit.opflow import VectorStateFn

dict_state_fn_exact_simulation = \
    VectorStateFn(statevector_exact_simulation).to_dict_fn()
```

[45]:
```
exact_simulation_df = pd.DataFrame(data={
    "bit_string": dict_state_fn_exact_simulation.primitive.keys(),
    "amplitude": dict_state_fn_exact_simulation.primitive.values()
})
exact_simulation_df["probability"] = \
    exact_simulation_df["amplitude"].abs()**2
```

[46]:
```
exact_simulation_df
```

[46]:
```
    bit_string            amplitude    probability
0     00000000  -0.002192+0.000047j   4.808748e-06
1     00000001   0.009366+0.004634j   1.091990e-04
2     00000010   0.000002+0.000524j   2.747835e-07
```

```
3      00000011  0.008908+0.006837j  1.261018e-04
4      00000100  0.009366+0.004634j  1.091990e-04
..        ...                  ...                  ...
251    11111011  0.000824-0.000822j  1.355178e-06
252    11111100 -0.000545-0.001926j  4.006500e-06
253    11111101  0.003056-0.002216j  1.424857e-05
254    11111110  0.000824-0.000822j  1.355178e-06
255    11111111  0.001988-0.000113j  3.962936e-06

[256 rows x 3 columns]
```

Now, we can merge the dataframes from the real backend and from the simulation and plot the results.

```
[47]: experiment_df = experiment_df.merge(
          exact_simulation_df[["bit_string", "probability"]],
          how="outer",
          on="bit_string",
          suffixes=["", "_exact"])
      experiment_df.fillna(0.0, inplace=True)
```

```
[48]: experiment_df
```

```
[48]:      bit_string  count  probability  probability_mit  probability_exact
      0      00000000   49.0     0.006125         0.005830       4.808748e-06
      1      00000001   58.0     0.007250         0.007232       1.091990e-04
      2      00010000  112.0     0.014000         0.013703       1.091990e-04
      3      00010001  167.0     0.020875         0.022115       1.974489e-02
      4      00010010  130.0     0.016250         0.016376       9.337863e-07
      ..        ...     ...          ...              ...                ...
      251    11111101    7.0     0.000875         0.000893       1.424857e-05
      252    11111110   12.0     0.001500         0.001518       1.355178e-06
      253    11111111    4.0     0.000500         0.000455       3.962936e-06
      254    01101000    0.0     0.000000         0.000000       7.846307e-08
      255    11101101    0.0     0.000000         0.000000       9.549872e-06

[256 rows x 5 columns]
```

We conclude this notebook with two plots comparing the probabilities of the results stemming from the real backend (with and without measurement error mitigation) and from the exact simulation.

```
[49]: experiment_df.sort_values(
          "probability_exact", ascending=False).iloc[0:40, :].plot.bar(
              x="bit_string",
              y=["probability", "probability_mit", "probability_exact"])
```

```
[49]: <AxesSubplot:xlabel='bit_string'>
```
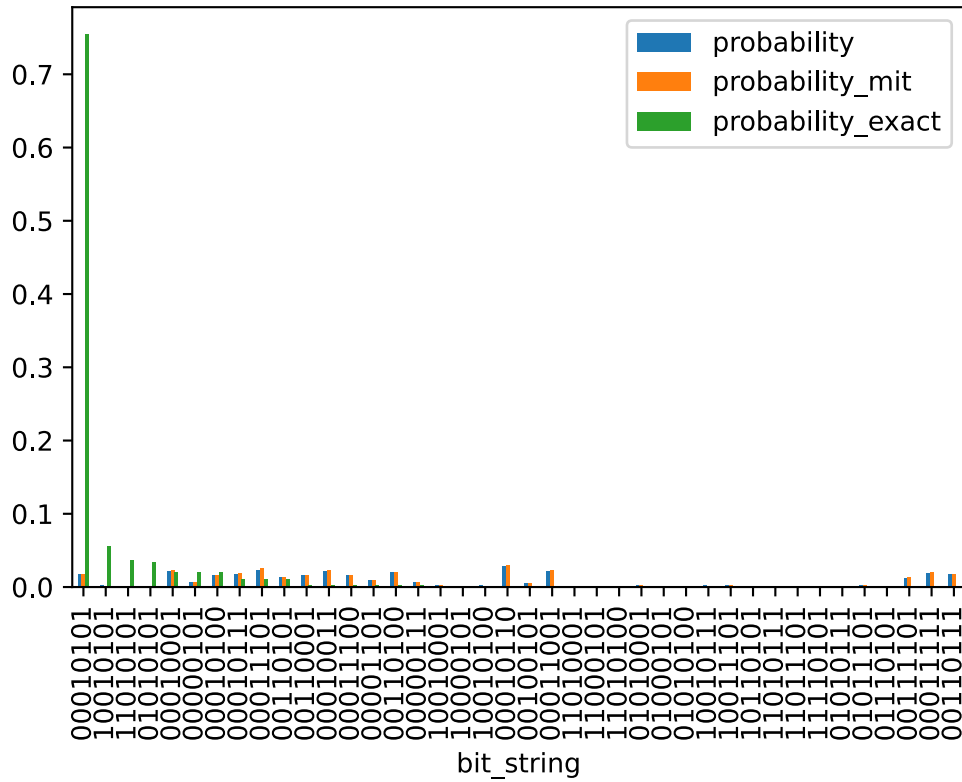
```
[50]: experiment_df.sort_values(
          "probability", ascending=False).iloc[0:40, :].plot.bar(
              x="bit_string",
              y=["probability", "probability_mit", "probability_exact"])
```

```
[50]: <AxesSubplot:xlabel='bit_string'>
```

Notebook 4

4.1 Introduction

In Notebook 1 we presented a real-world use case for optimizing charging schedules for electric cars, reduced it to a proof of concept model, and transformed it to a QUBO. Then, in Notebook 2 we presented the quantum algorithm QAOA and how the associated quantum circuits can be obtained. Subsequently, in Notebook 3 we explained how these quantum circuits can be transpiled and run on real quantum computers, and how the results of such experiments can be postprocessed. Building on this pipeline one can build a **series of experiments** in order to **study how good quantum computing** (with all its **limitations** in the current **NISQ era**) can be employed **for our charging schedule optimization use case**. It is exactly a series of such experiments that we will present in this notebook.

Note: Our main aim with this notebook is to provide and discuss results from quantum computing experiments. In order to keep this notebook at a reasonable scope we thus don't provide the detailed codes but only give the most important parts of them. Together with the knowledge of the previous notebooks the reader should be able to develop codes by him/herself (if this is desired)

We begin with introducing the two example series on which all our experiments will be based.

4.2 Examples for this Notebook

For all examples in this notebook we consider **1 charging station** with **4 charging levels** and **4 available time slots**.

4.2.1 Example Series 1

As a first series of examples (denoted by example1pX, X=0, 1, 2 or 3) we assume that **1 car** is at the charging station and needs to **charge 4 energy units**. The **examples differ** in the **duration** that the car is **at the charging station**, namely:

- example1p0 → car is at charging station at time slot 0.
- example1p1 → car is at charging station at time slots 0, 1.
- example1p2 → car is at charging station at time slots 0, 1, 2.
- example1p3 → car is at charging station at time slots 0, 1, 2, 3.
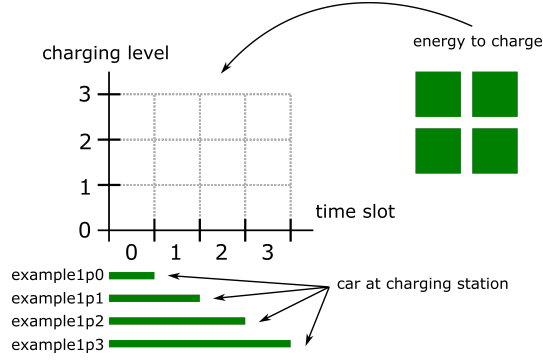
Figure 4.1: Visualization of Example Series 1.

4.2.2 Example Series 2

Our second series of examples (denoted by example2pX, X=0, 1, 2, 3 or 4) considers the situation where **2 cars** are at the charging station and **both** need to **charge 4 energy units**. Again, the **examples differ** in the **duration** that the cars are **at the charging station**:

- example2p0 $\rightarrow$ time slots green car: 0, time slots orange car: 1.
- example2p1 $\rightarrow$ time slots green car: 0, 1, time slots orange car: 1, 2.
- example2p2 $\rightarrow$ time slots green car: 0, 1, 2, time slots orange car: 1, 2, 3.
- example2p3 $\rightarrow$ time slots green car: 0, 1, 2, 3, time slots orange car: 1, 2, 3.
- example2p4 $\rightarrow$ time slots green car: 0, 1, 2, 3, time slots orange car: 0, 1, 2, 3.
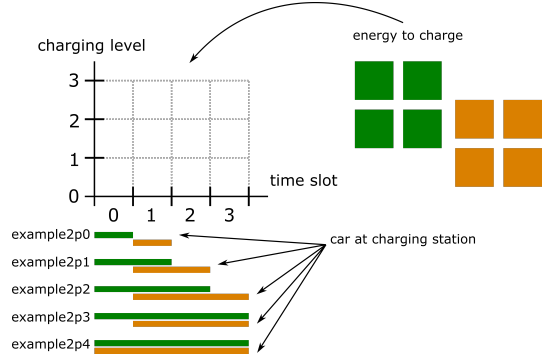
Figure 4.2: Visualization of Example Series 2.

Our first experiments are concerned with the classical optimization part of QAOA.

4.3 Classical Optimization

Recall from Notebook 2 that we want to find parameters $\vec{\beta}$ and $\vec{\gamma}$ such that the expectation value e, given by

$$e(\vec{\beta}, \vec{\gamma}) = \langle \psi_{\text{QAOA}}(\vec{\beta}, \vec{\gamma}) | \mathcal{H}_P | \psi_{\text{QAOA}}(\vec{\beta}, \vec{\gamma}) \rangle \ ,$$

is minimized. This means we search for $\vec{\beta}^*$ and $\vec{\gamma}^*$ that satisfy

70

$$(\vec{\beta}^*, \vec{\gamma}^*) = \operatorname*{argmin}_{\vec{\beta}, \vec{\gamma}} e(\vec{\beta}, \vec{\gamma}) \ .$$

4.3.1 Optimization Landscape

For $p = 1$ we only have **two parameters** β_0 and γ_0 so that the **expectation value** $e(\beta_0, \gamma_0)$ can be **visualized as a heatmap**. We have done this for the following setting:

- We visualize the optimization landscape for parameters in the domain $[0, \pi] \times [0, 2\pi]$. For this purpose we use a fine, equidistant discretization of $[0, \pi] \times [0, 2\pi]$ with 100×200 gridpoints $(\beta_{0,j}, \gamma_{0,k})$, $j = 0, \ldots, 99$, $k = 0, \ldots, 199$.
- For every grid point $(\beta_{0,j}, \gamma_{0,k})$ we compute the expectation value $e(\beta_{0,j}, \gamma_{0,k})$. This gives a 100×200 matrix $E = (e(\beta_{0,j}, \gamma_{0,k}))_{j,k}$.
- For the transformation to a QUBO we use as value for the penalty ϱ the minimum value such that the solution of (QUBO) is a solution of (QCIO).
- We use an exact state vector simulation to compute the expectation value.

In the following figures we see the results, where the **expectation value** $e(\beta_{0,j}, \gamma_{0,k})$ is **visualized** by the **color** of the point $(\beta_{0,j}, \gamma_{0,k})$ (see the colorscale in the plots).
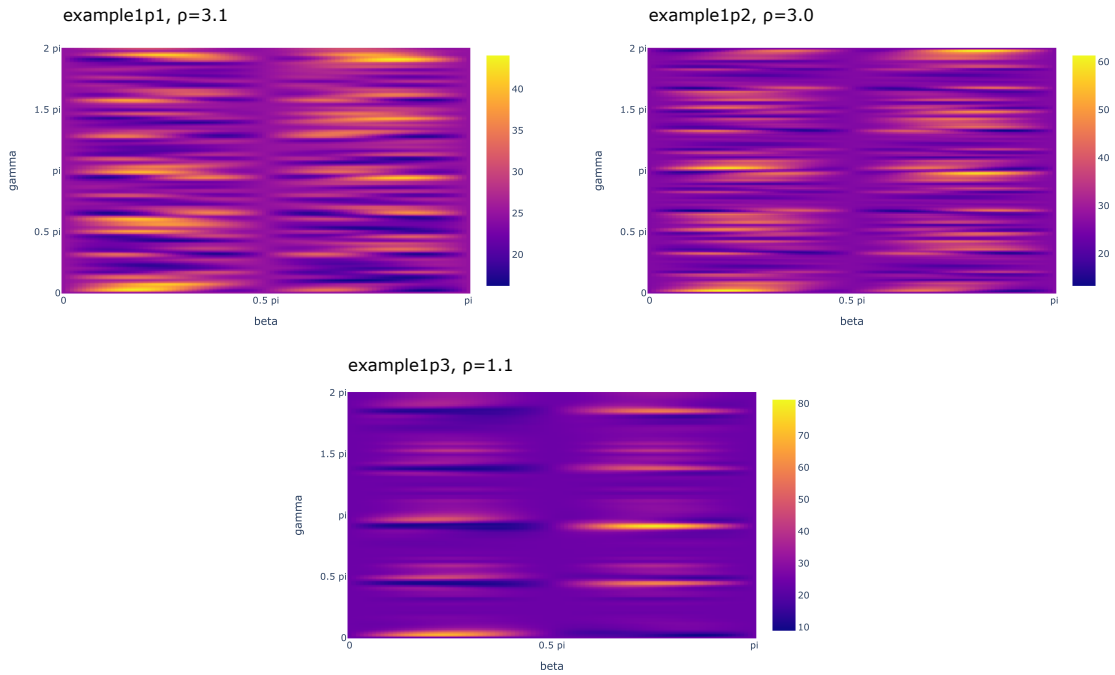
Figure 4.3: Optimization landscapes for Example Series 1.
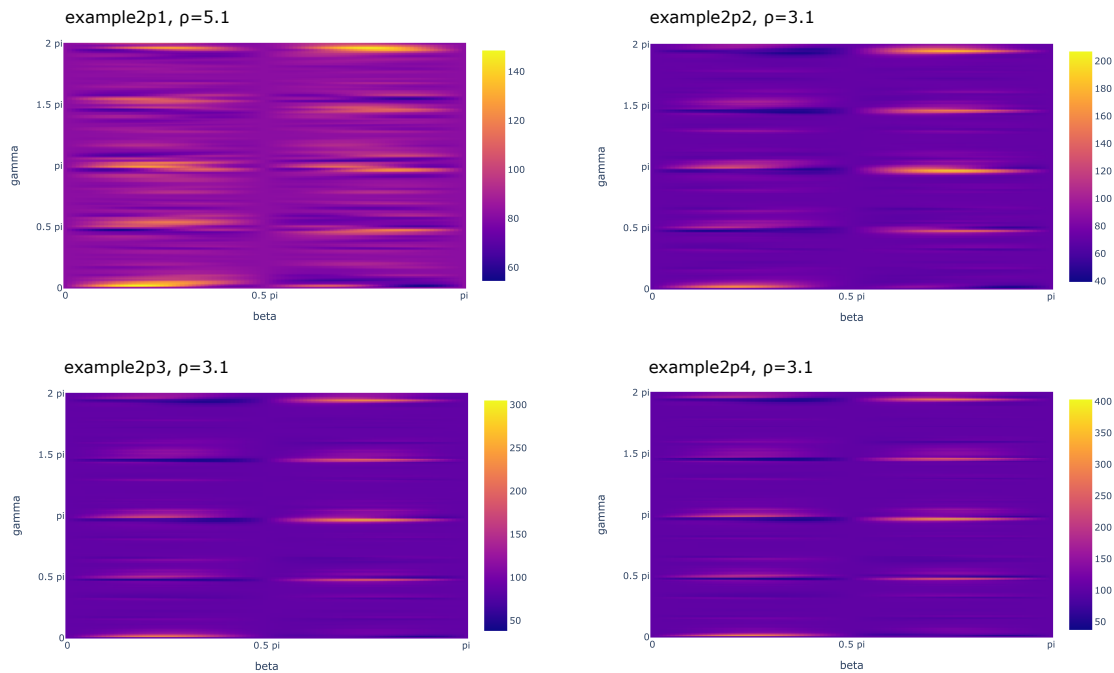
Figure 4.4: Optimization landscapes for Example Series 2.

We see that the optimization landscapes are fairly complicated with lots of **local extrema**. This indicates that it is a **hard task** for classical optimizers to find the **global minimum** (depending on the initial guess (=starting point) for a local optimizer it will get stuck in a local minimum). Moreover, we already can see that for the bigger examples gradients vanish, see further the so-called **barren plateaus** phenomenon [1, 16].

In the next section we report results from a classical optimizer and will indeed experience that finding good parameters $\vec{\beta}$ and $\vec{\gamma}$ is a difficult task.

Code snippet:

```python
from joblib import Parallel, delayed
import plotly.graph_objects as go

beta_mesh = np.linspace(0, np.pi, beta_mesh_grid_points)
gamma_mesh = np.linspace(0, 2*np.pi, gamma_mesh_grid_points)
energies = Parallel(n_jobs=-1)(
    delayed(
        energy_evaluation # this was defined in Notebook 2
    )(
        [beta, gamma]
    ) for beta in beta_mesh for gamma in gamma_mesh
)
```

```
energy_matrix = np.reshape(
    energies,
    (gamma_mesh.size, beta_mesh.size),
    order='F')

fig = go.Figure(
    data =
        go.Heatmap(
            z=np.real(energy_matrix),
            x=beta_mesh,
            y=gamma_mesh,
))
```

4.3.2 Results with Optimizer COBYLA

For now following results we used the **optimizer COBYLA** to minimize $e(\vec{\beta}, \vec{\gamma})$ for QAOA with $p = 1, 2$ and 3. Moreover, we used

- 10 different values for the penalty ϱ, starting from the minimum that gives a feasible solution and advancing in steps of 0.1.
- For every combination of p and ϱ we ran 50 COBYLA minimizations with random initial guesses $\vec{\beta}^{(0)} \in [0, \pi]^p$, $\vec{\gamma}^{(0)} \in [0, 2\pi]^p$. The remaining parameters of COBYLA where left by their default values.
- We used an exact state vector simulation to compute the expectation value.

We report the **expectation value** $e(\vec{\beta^*}, \vec{\gamma^*})$ (termed cost in the figures) below, where $\vec{\beta^*}$ **and** $\vec{\gamma^*}$ are the **result of the COBYLA minimization**:

$$\text{random } \vec{\beta}^{(0)}, \vec{\gamma}^{(0)} \overset{\text{COBYLA}}{\rightsquigarrow} \vec{\beta^*}, \vec{\gamma^*} .$$

The **dashed line** in the plots corresponds to the **solution of the original minimization problem**, i.e. this is the value we try to reach with $e(\vec{\beta^*}, \vec{\gamma^*})$.

Remark: In the subsequence we will often plot results as **box plots**. In these plots the ends of the box represent the lower and upper quartiles, while the median (second quartile) is marked by a line inside the box.

73
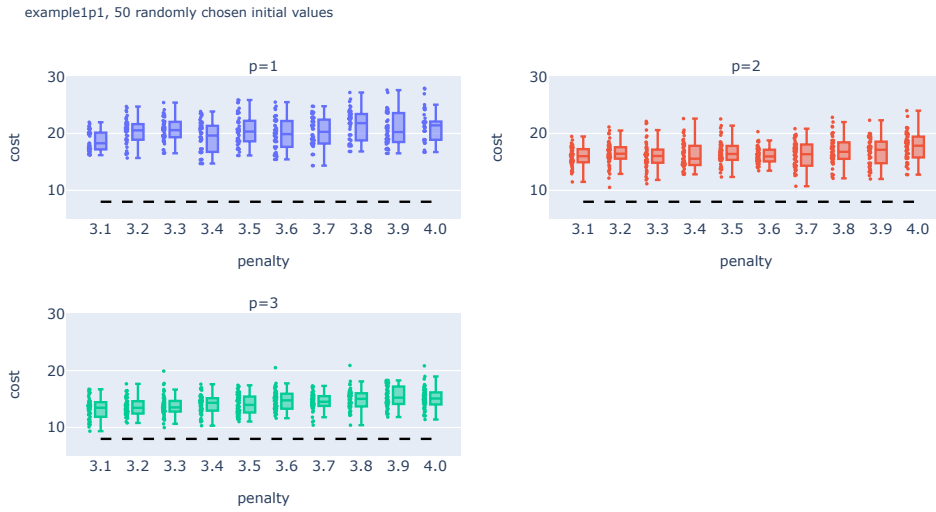
(a) Results for example1p1.
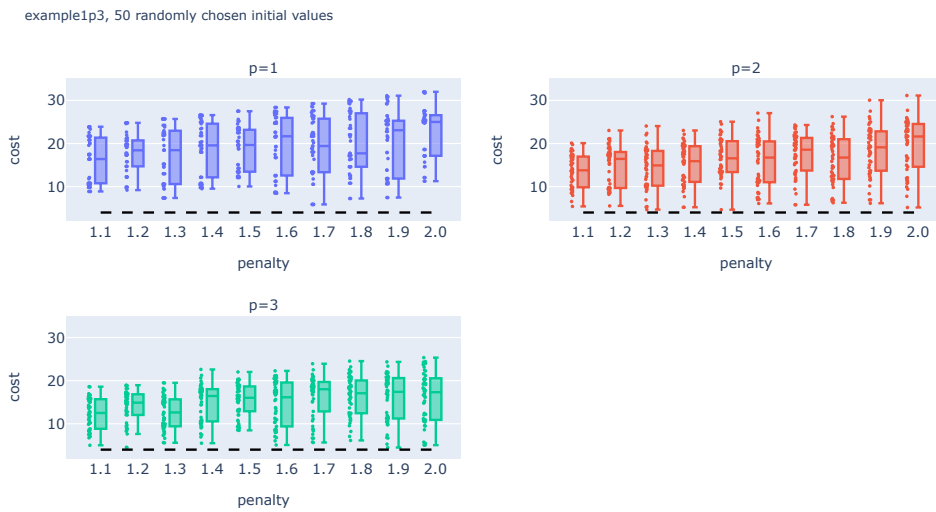
(b) Results for example1p3.

Figure 4.5: Results of COBYLA optimization for two subexamples of Example Series 1. We used $p = 1, 2$, and 3, and different penalties ϱ. Every dot corresponds to a randomly chosen initial guess $\vec{\beta}^{(0)}, \vec{\gamma}^{(0)}$. The dasehd line indicates the minimum cost, i.e. the exact solution.
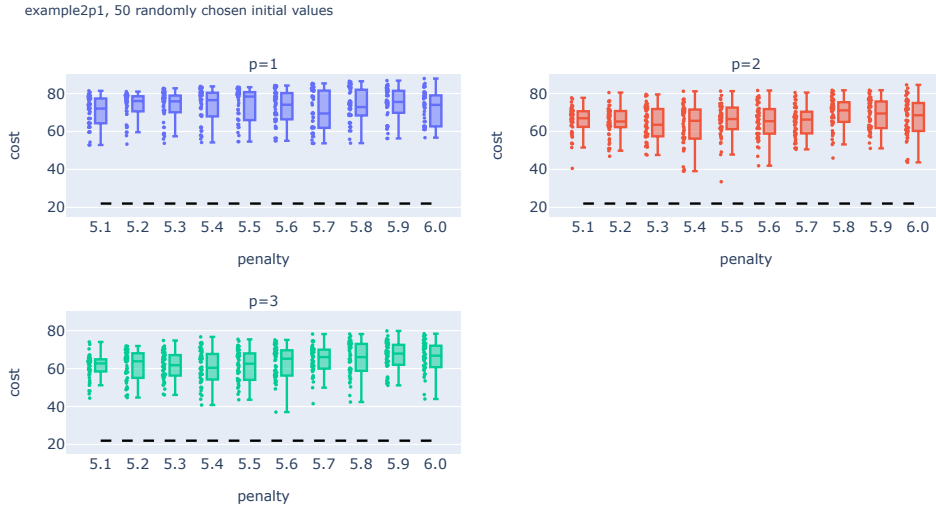
(a) Results for example2p1.

(b) Reults for example2p4.

Figure 4.6: Results of COBYLA optimization for two subexamples of Example Series 2. We used $p = 1, 2$, and 3, and different penalties ϱ. Every dot corresponds to a randomly chosen initial guess $\vec{\beta}^{(0)}, \vec{\gamma}^{(0)}$. The dasehd line indicates the minimum cost, i.e. the exact solution.

In general, we see a **strong dependency** (and thus a **high variance**) of the result of the optimization with the **choice of the initial values** $\vec{\beta}^{(0)}$ and $\vec{\gamma}^{(0)}$ (every dot in the upper figures corresponds to one random choice of the initial values). Comparing the simpler examples with the more complicated ones, i.e. example1p1 with example1p3 and example2p1 with example2p4, we observe a higher variance and that QAOA with $p = 1$ performs poorly for the more complicated problems (note the accumulation of results at very high expectation values).

Another observation is that a **higher parameter** p (as expected) **improves the quality of the QAOA solution**. In particuar, for example1p1 we observe that QAOA with $p = 3$ yields expectation values near to the exact solution whereas QAOA with $p = 1$ is bounded away from the exact solution for every initial choice $\vec{\beta}^{(0)}$ and $\vec{\gamma}^{(0)}$.

In summary, we have seen that optimizing the parameters $\vec{\beta}$ and $\vec{\gamma}$ is indeed a difficult task. So, we can conclude that different optimizers should be tested and at least a few different initial values should be compared. We also observe that the choice of penalty plays a role but can hardly conclude a recommendation from the data.

With this we end our experiments that solely used a simulator. The **following sections** deal with **results from real quantum computers**. An important **first step** here is to **analyze the transpilation** of our QAOA circuits and the **number of basis gates** (i.e. the number of gates that the quantum computing device natively supports) it requires in order to evaluate how well it can be executed on NISQ computers.

4.4   Analysis of Transpiled QAOA Circuits

In Section 1.6.4 we already saw a sparsity plot of the QUBO matrix. Now, we explain how the sparsity pattern affects the transpilation of the QAOA circuit associated with the QUBO.

4.4.1   Sparsity of QUBO Matrix and Number of RZZ Gates

Recall from Notebooks 1 and 2 the QUBO cost function

$$f_3(\vec{b}) = \vec{b}^t A \vec{b} + L \vec{b} + c = \sum_{i=0}^{n-1} \sum_{j>i}^{n-1} a_{ij} b_i b_j + \sum_{i=0}^{n-1} l_i b_i + c \ ,$$

and that the transformation $b_i \leftrightarrow \frac{1}{2}\left(I^{\otimes n} - \sigma_Z^{(i)}\right)$ resulted in the cost Hamiltonian

$$\mathcal{H}_P = \sum_{i=0}^{n-1} \sum_{j>i}^{n-1} h_{ij} \sigma_Z^{(i)} \sigma_Z^{(j)} + \sum_{i=0}^{n-1} h_i' \sigma_Z^{(i)} + h'' I^{\otimes n} \ .$$

Moreover, recall the mixing and the phase operators

$$\mathcal{U}_M(\beta) = \exp(-\mathrm{i}\beta \mathcal{H}_M), \qquad \text{and} \qquad \mathcal{U}_P(\gamma) = \exp(-\mathrm{i}\gamma \mathcal{H}_P) \ ,$$

respectively, as well as the gates needed to implement them:

$$\mathcal{U}_M(\beta) = \mathrm{RX}_0(2\beta) \mathrm{RX}_1(2\beta) \cdots \mathrm{RX}_{n-1}(2\beta) \ ,$$

$$\mathcal{U}_P(\gamma) = \mathrm{RZZ}_{0,1}(2\gamma h_{01}) \cdots \mathrm{RZZ}_{n-2,n-1}(2\gamma h_{n-2,n-1}) \ \mathrm{RZ}_0(2\gamma h_0') \cdots \mathrm{RZ}_{n-1}(2\gamma h_{n-1}') \ .$$

Observe that we need the **one-qubit gates** RX and RZ as well as the **two-qubit gate** RZZ. More precisely, we obtain one RZZ gate for every coefficient $h_{ij} \neq 0$. It is easy to see that we have

$h_{ij} \neq 0 \Leftrightarrow a_{ij} \neq 0$. This means: **the sparser** the matrix A ($=$ the lesser the number of non-zero entries in A) the **lesser** RZZ **gates** we have in the circuit:

$$\text{number of non-zero entries of } A \quad \Longleftrightarrow \quad \text{number of RZZ gates in the circuit .}$$

Remark: One can include L into A (in the QUBO cost function f_3) because $b_i^2 = b_i$. The resulting matrix, let's call it $\tilde{A}$, then can have *non-zero* diagonal entries $\tilde{a}_{ii} = l_i$. However, these entries *do not give rise* to a RZZ gate. In this case we have

$$\text{number of non-zero (off-diagonal) entries of } \tilde{A} \quad \Longleftrightarrow \quad \text{number of RZZ gates in the circuit .}$$

Clearly, the question arises why we should be mainly interested in the number of RZZ gates and can neglect (to a certain degree) the number of the single qubit gates. In order to understand this we have to analyze the **transpilations** of the gates.

4.4.2 Transpilation of RX, RZ and RZZ

Let us write a quantum circuit with one RX, one RZ, and one RZZ gate and transpile it to the **basis gates** of the current IBM quantum computers (see Notebook 3).

```
[1]: from qiskit.circuit import QuantumCircuit, Parameter
     from qiskit.compiler import transpile

     theta = Parameter("$\\theta$")
     qc = QuantumCircuit(2)
     qc.rx(theta, 0)
     qc.barrier()
     qc.rz(theta, 0)
     qc.barrier()
     qc.rzz(theta, 0, 1)
     qc.draw()
```

[1]:
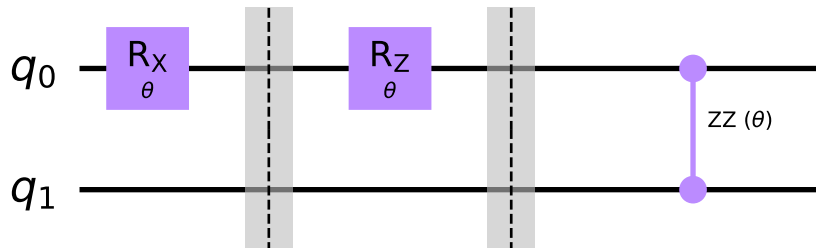

```
[2]: # See Notebook 3
     basis_gates = ['id', 'rz', 'sx', 'x', 'cx', 'reset']
```

```
qc_transpiled = transpile(qc, basis_gates=basis_gates)
qc_transpiled.draw()
```

[2]:

Global Phase: $3\pi/2$
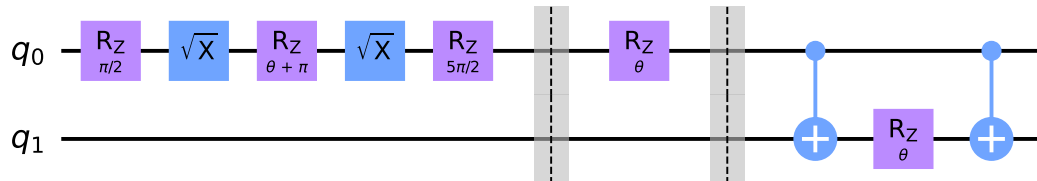
Note that the RZ gate is a virtual gate and let us thus call the set of basis gates without RZ the **hardware gates**. Then, we see that RX requires two single-qubit hardware gates, the RZ gate needs none, and RZZ needs **two two-qubit hardware gates** (i.e. the two CNOT gates). Recalling from Notebook 3 that the CNOT **gate** is the **most erroneous hardware gate** (around one magnitude higher error rate than the two single qubit gates) explains why we have to lay our focus on the RZZ gates when analyzing how well a QAOA circuit can be executed on real quantum hardware.

4.4.3 CNOT **Gates and Coupling Map**

Recall from Notebook 3 that not all qubits in a quantum computer are connected with each other (see also the coupling map in Figure 4.10). So, let's see how a CNOT gate is transpiled for two qubits that are not connected.

[3]:
```
qc = QuantumCircuit(3)
qc.cx(0, 2)
qc.draw()
```

[3]:
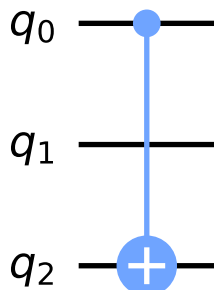
```
[4]:  # Only qubits 0-1, 1-2 are connected.
      coupling_map=[[0, 1], [1, 0], [1, 2], [2, 1]]
      qc_transpiled = transpile(
          qc,
          coupling_map=coupling_map,
          initial_layout=[0, 1, 2],
          seed_transpiler=123)
      qc_transpiled.draw()
```

[4]:

$$q_0 \mapsto 0$$

$$q_1 \mapsto 1$$

$$q_2 \mapsto 2$$
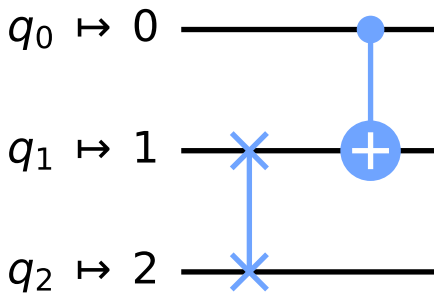
We see that the transpiler needs to insert a SWAP gate between qubits 1 and 2 in order to realize the CNOT gate between qubits 0 and 2. Clearly, this raises the question how a SWAP gate is transpiled to basis gates:

```
[5]:  qc_transpiled = transpile(
          qc_transpiled,
          coupling_map=coupling_map,
          initial_layout=[0, 1, 2],
          basis_gates=basis_gates,
          seed_transpiler=123)
      qc_transpiled.draw()
```

[5]:
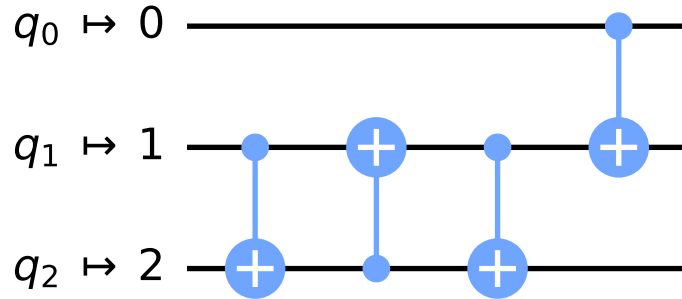
We see that a SWAP **gate** requires **three** CNOT **gates**! Having in mind the limited coupling map of current (IBM) quantum computers it is easy to imagine that the denser the QUBO matrix A is populated (i.e. the more non-zero entries) the more SWAP gates (and therefore CNOT gates) are needed because many RZZ gates between many different qubits have to be transpiled.

Now, let us observe these effects on our examples.

4.4.4 Example Series 1

Sparsity Pattern of QUBO Matrix

We begin by recalling Example Series 1:
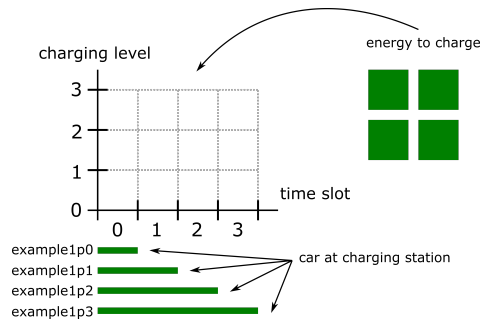
Figure 4.7: Reminder of Example series 1, see Section 4.2.1.

The next plot shows the sparsity pattern of the QUBO matrices.
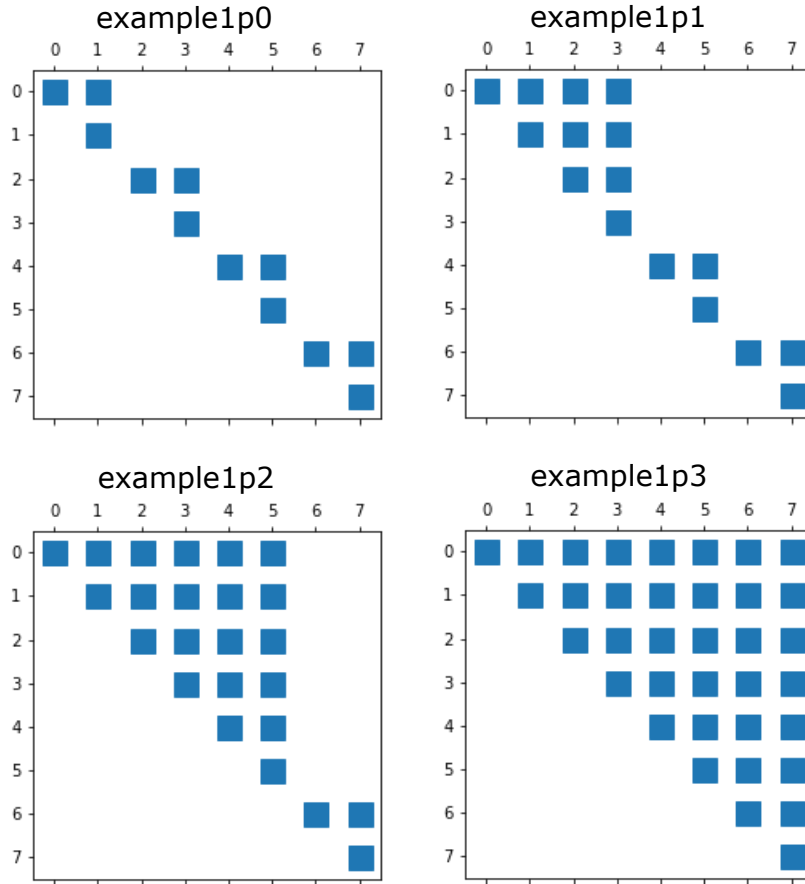
Figure 4.8: Sparsity pattern of QUBO matrices for Example Series 1.

For these simple examples we can easily derive the structure of the QUBO matrices: We need to represent the charging level of the car for every time slot 0, 1, 2 and 3. Since we have 4 charging levels we need 2 qubits to represent the charging level for every time slot. So, we need $4 \cdot 2 = 8$ qubits for every example in Example Series 1.

In the plot for example1p0 you can see that we only have a coupling (off-diagonal element) between qubits $0 - 1$, $2 - 3$, $4 - 5$, and $6 - 7$. Every two qubit pair represents the charging level for a time slot. Since the car in this example is only at the charging station at time slot 0 no coupling between the different time slots is necessary. This changes for example1p1 where we see a coupling between qubits $0 - 1 - 2 - 3$. This stems from the fact that in this example the car is at the charging station at time slots 0 and 1, and thus also the qubits involved must be coupled. And so it goes on until example1p3, where the car is at the charging station for all the time slots and we observe a full coupling between all the qubits.

Now, let us investigate how the sparsity pattern of the different examples affect the count of basis gates for QAOA with $p = 1$.

Gate Count: Fully Connected Topology

First, let us assume that we have a **fully connected topology**, i.e. each qubit is connected with all other qubits. In the following figure we plot the number of hardware gates, i.e. of X, $\sqrt{X}$ and CNOT (in Qiskit: `x`, `sx` and `cx`), and the depth of the circuit for all subexamples of Example Series 1.

qaoa p=1, 8 qubits, fully connected topology, no optimization
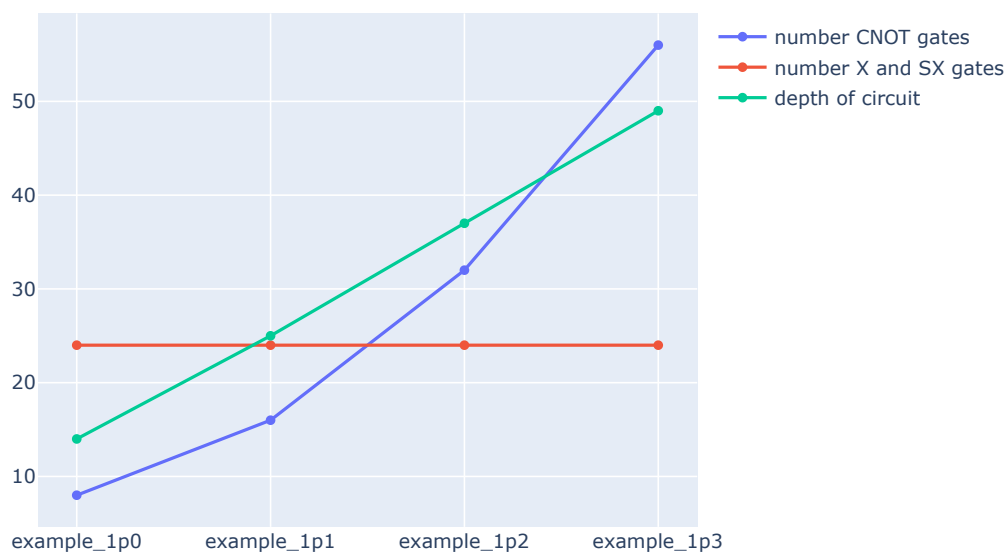
Figure 4.9: Number of X, $\sqrt{X}$, and CNOT gates and depth of the transpiled circuits for Example Series 1 for a fully connected topology.

We see that the number of CNOT gates grows whereas the number of single qubit gates stays constant. The reason is that the subexamples only differ in the number of time slots that the car is at the charging station and (as we discussed above) this directly affects the number of RZZ gates which then directly translate to the number of CNOT gates.

Code snippet:

```
# See Notebook 3.
backend_basis_gates = ["id", "rz", "sx", "x", "cx", "reset"]
backend_single_qubit_basis_gates = ["sx", "x"]

qaoa_ansatz_transpiled = transpile(
    qaoa_ansatz,
    basis_gates=backend_basis_gates,
```

```
    optimization_level=0) # has no effect in this case


number_cnots = qaoa_ansatz_transpiled.count_ops()['cx']
# The function count_gates is provided in utils.
number_sx_x = count_gates(
    qaoa_ansatz_transpiled,
    backend_single_qubit_basis_gates)
depth = qaoa_ansatz_transpiled.depth()
```

Gate Count: ibmq_ehningen

Next, we transpile the QAOA circuits for our example series to the **ibmq_ehningen backend**. The hardware gates are the same as above but recall the **limited connectivity between the qubits** (see Figure 4.10) that makes it necessary to introduce SWAP gates (which we have seen are transpiled to three CNOT gates).
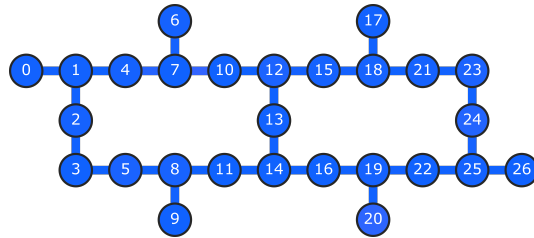
Figure 4.10: Coupling map of ibmq_ehningen.

Figure 4.11 shows the results for Example Series 1 transpiled to ibmq_ehningen.
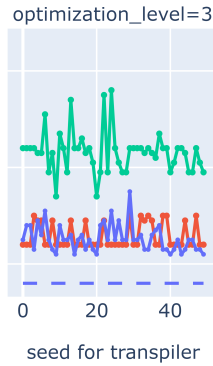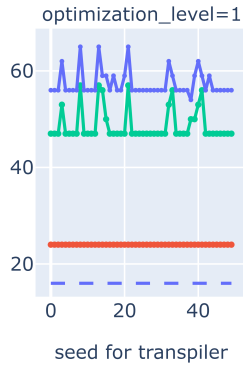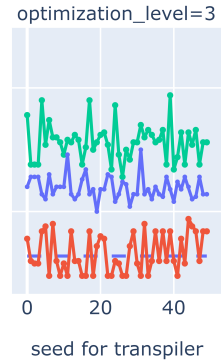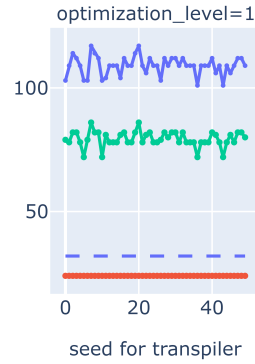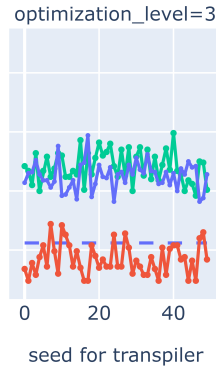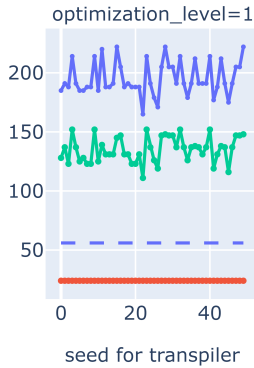
Figure 4.11: Number of X, $\sqrt{X}$, and CNOT gates and depth of transpiled circuits for Example Series 1 for ibmq_ehningen. We used 50 different transpiler seeds and optimization levels 1 and 3.

As discussed in Notebook 3 there are many parameters to **control the transpilation process**. Two important ones are the `optimization_level` and the `seed_transpiler`. For the later we already explained that it controls the stochastic part of the transpilation process. Looking at the figures above we see that this can make a significant difference. As demonstrated in Notebook 3 it is thus a **good practice** to run a transpilation several times and **choose the circuit with the least CNOT gates**. Moreover, we see that a transpilation can be done in many different ways and so there is much potential for optimization. This is controlled with the `optimization_level`. The higher the number, the more optimized the transpiled circuit is but at the expense of more computation time. We refer to [9, 10, 21] for further information. Looking at the severe restrictions of NISQ hardware we would recommend `optimization_level=3` in order to obtain the best circuits and take the most of the current potential. However, keep in mind that a higher optimization level comes at higher cost on the classical computer.

Code snippet:

```
ibmq_ehningen = provider.get_backend("ibmq_ehningen")
backend_coupling_map = ibmq_ehningen.configuration().coupling_map
backend_basis_gates = ibmq_ehningen.configuration().basis_gates

seeds_for_transpiler = [k for k in range(50)]
number_seeds_for_transpiler = len(seeds_for_transpiler)

quantum_circuit_transpilations = transpile(
    [qaoa_ansatz]*number_seeds_for_transpiler,
    coupling_map=backend_coupling_map,
    basis_gates=backend_basis_gates,
    optimization_level=3, # 0, 1, 2 or 3
    seed_transpiler=seeds_for_transpiler)
```

Gate Count: Comparison

As a last plot for Example Series 1 let us **compare** the **number of** CNOT **gates** between the **fully connected topology** and **ibmq_ehningen**:
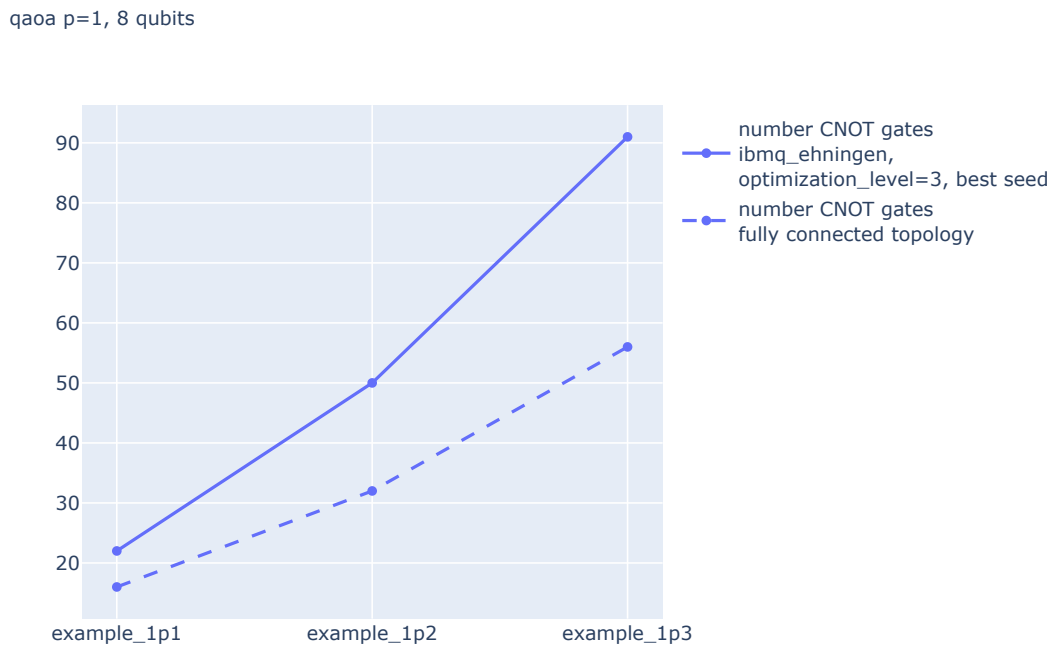
Figure 4.12: Comparison of number of CNOT gates for Example Series 1 between a fully connected topology and the best transpilation for ibmq_ehningen.

We see that the difference in the number of CNOT gates **diverges** from example_1p1 over example_1p2 to example_1p3. This has the reason that the **coupling between the variables** (i.e. the non-zero entries in the QUBO matrix) increases from example_1p1 to example_1p3 and that this coupling is realized via a RZZ gates between the corresponding qubits. The more qubits are connected via RZZ gates the more SWAP gates have to be used in order to compensate for the limited connectivity of ibmq_ehningen and thus the divergence.

4.4.5   Example Series 2

Now, we present the figures for Example Series 2 for the same experiments as for Example Series 1. We will observe the same effects and thus will only make a few comments.

Sparsity Pattern of QUBO Matrix

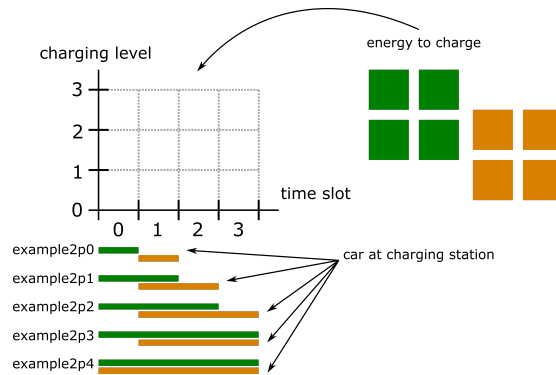As for Example Series 1 we start by recalling Example Series 2:

Figure 4.13: Reminder of Example Series 2, see Section 4.2.2.

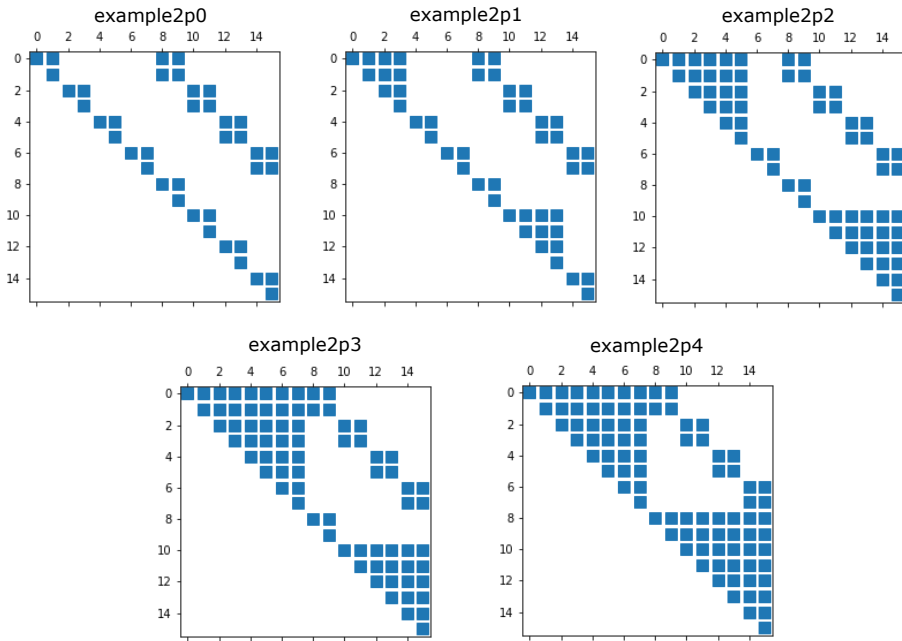The sparsity pattern of the QUBO matrices is given in the next figure:

Figure 4.14: Sparsity pattern of the QUBO matrices for Example Series 2.

Note the following differences to Examples Series 1: Clearly, we now need 16 qubits – 8 qubits for the green car and 8 qubits for the orange car. Moreover, note that we have coupling of qubits $0 - 1 - 8 - 9$, $2 - 3 - 10 - 11$, $4 - 5 - 12 - 13$, $6 - 7 - 14 - 15$. These are needed for every time slot to add the charging level for the green and the orange car. Altogether, we see a more complicated sparsity pattern of the QUBO matrix and expect that this will translate to deeper circuits with more gates and eventually a poorer quality from results of real quantum computers.

Gate Count: Fully Connected Topology

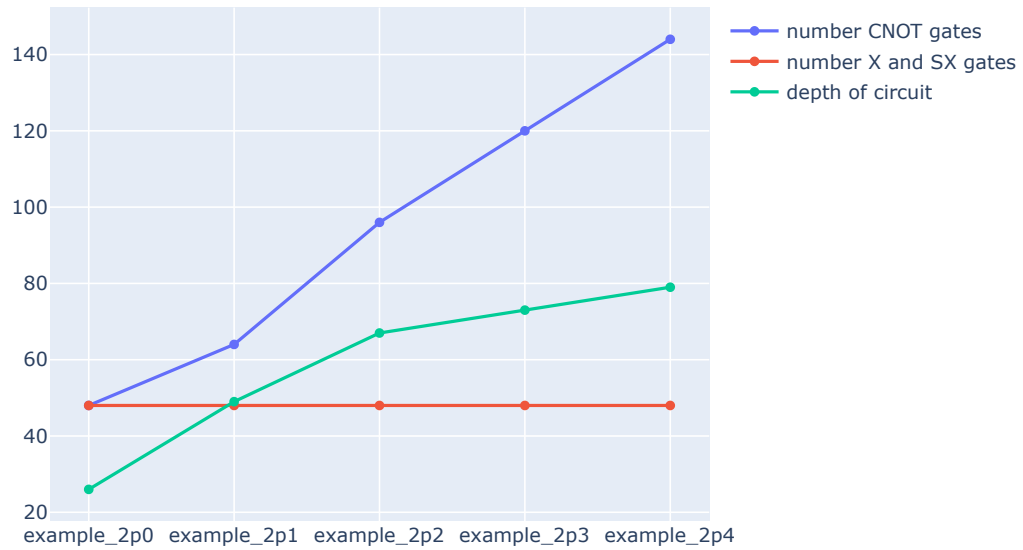qaoa p=1, 16 qubits, fully connected topology, no optimization

Figure 4.15: Number of X, $\sqrt{\text{X}}$, and CNOT gates and depth of transpiled circuits for Example Series 2 for a fully connected topology.
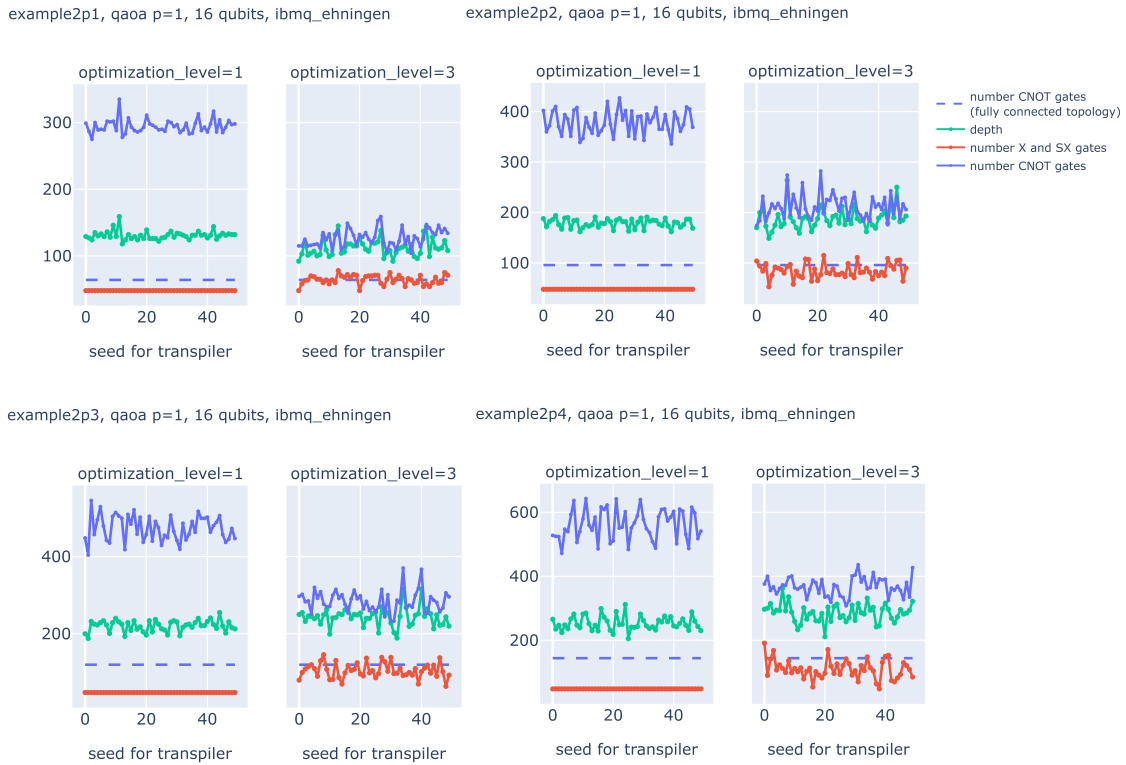
Gate Count: ibmq_ehningen

Figure 4.16: Number of X, $\sqrt{X}$, and CNOT gates and depth of transpiled circuits for Example Series 2 for ibmq_ehningen. We used 50 different transpiler seeds and optimization levels 1 and 3.
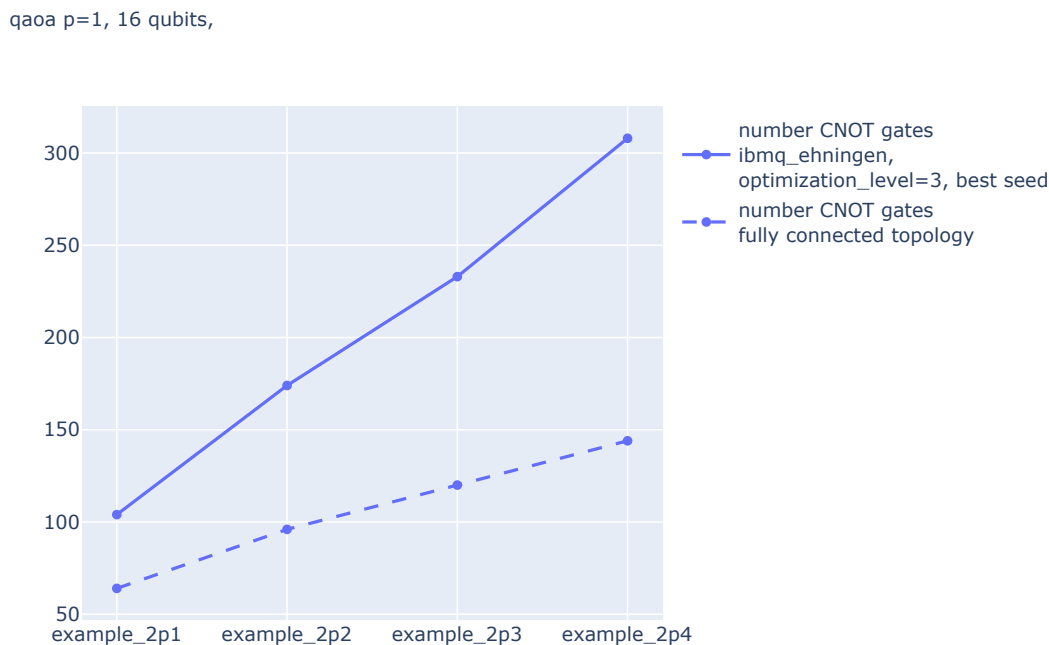
Gate Count: Comparison

qaoa p=1, 16 qubits,

number CNOT gates
ibmq_ehningen,
optimization_level=3, best seed

number CNOT gates
fully connected topology

Figure 4.17: Comparison of number of CNOT gates for Example Series 2 between a fully connected topology and the best transpilation for ibmq_ehningen. Here we see the divergence in the number of CNOT gates between a fully connected quantum computer and ibmq_ehningen even stronger than in Example Series 1.

In the last part of this notebook we present results from solving Example Series 1 and 2 with QAOA on ibmq_ehningen.

4.5   QAOA Results on ibmq_ehnigen

Our set up for the experiments reported below is the following:

- We use the **best parameters** $\vec{\beta}$, $\vec{\gamma}$, and ϱ that we found in our simulator experiments in Section 4.3, see Table 4.1. Thus, we can analyze the **performance of ibmq_ehningen** for the **best possible QAOA circuits**.
- We **transpile** the QAOA circuits with **75 different seeds** and **optimization level 3**. This transpilation is abbreviated with std in the figures below.
- We additionally add **dynamical decoupling** to the transpiled circuits (abbreviation then is dd).
- We employ **measurement error mitigation** to the std and dd circuits via the package `mthree` (abbreviation then is xx_mit, where xx = std or dd)

For more information on the error mitigation techniques see Notebook 3.

90

4.5.1 Quality Metric

There are many ways how we can measure the quality of the result of a computation on a real quantum computer. We will use the **fidelity** and the **expectation value**.

Fidelity

In the following we will mainly measure the quality of the probability distribution $\{q_i\}$ stemming from running the QAOA circuit on **ibmq_ehningen** by computing the **fidelity** F with respect to the probability distribution $\{p_i\}$ from an **exact state vector simulation**. Thereby, the fidelity is defined as

$$F(\{p_i\}, \{q_i\}) = \sum_i \sqrt{p_i q_i} \ ,$$

see further [17, Chapter 9]. We note that the fidelity is between 0 and 1, where 0 is the worst case and 1 is the best case. In Qiskit the fidelity can be computed via `qiskit.quantum_info.hellinger_fidelity`.

Code snippet:

```
from qiskit.quantum_info import hellinger_fidelity

# See the end of Notebook 3 for experiment_df
fidelity = hellinger_fidelity(
    experiment_df["probability_exact"].to_dict(),
    experiment_df["probability"].to_dict())
```

Expectation Value

In Notebook 2 we explained that the expectation value

$$e(\vec{\beta}, \vec{\gamma}) = \langle \psi_{\mathrm{QAOA}}(\vec{\beta}, \vec{\gamma}) | \mathcal{H}_P | \psi_{\mathrm{QAOA}}(\vec{\beta}, \vec{\gamma}) \rangle$$

is connected to the QUBO cost function f_3 by

$$e(\vec{\beta}, \vec{\gamma}) = \sum_{b \in \{0,1\}^n} |\lambda_b(\vec{\beta}, \vec{\gamma})|^2 f_3(\vec{b}) \ ,$$

where the amplitudes $\lambda_b(\vec{\beta}, \vec{\gamma})$ belong to the state $|\psi_{\mathrm{QAOA}}(\vec{\beta}, \vec{\gamma})\rangle$, i.e.

$$|\psi_{\mathrm{QAOA}}(\vec{\beta}, \vec{\gamma})\rangle = \sum_{\vec{b} \in \{0,1\}^n} \lambda_b(\vec{\beta}, \vec{\gamma}) |b\rangle \ .$$

This means a low expectation value e indicates that QAOA generates a quantum state $|\psi_{\text{QAOA}}(\vec{\beta}, \vec{\gamma})\rangle$ with **large amplitudes** for **bit strings** b with **low cost** $f_3(\vec{b})$. Thus, measuring e is a meaningful quality metric, in particular if one compares it to the minimum value of f_3, i.e. $\min_{\vec{b} \in \{0,1\}^n} f_3(\vec{b})$.

In the following sections we first present results from experiments for Example Series 1 and then close this notebook by presenting results for Example Series 2.

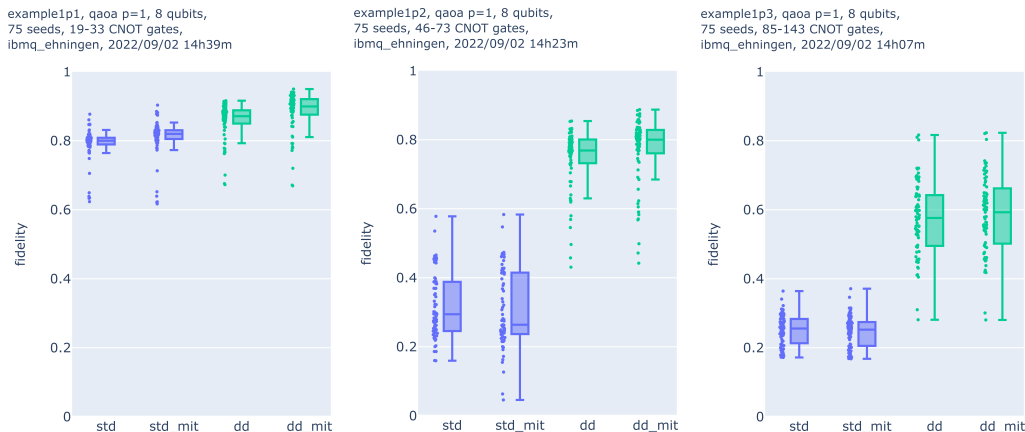### 4.5.2 Fidelity, Example Series 1, $p = 1$

Figure 4.18: Fidelity vs. transpilation method for example1p1 (left), example1p2 (center), and example1p3 (right). Every dot stems from one of 75 transpilation seeds.

In the upper three plots we can clearly see the influence of the number of CNOT gates. On the left, we see that a **moderate number of** CNOTs leads to a **fairly high fidelity** – with and without dynamical decoupling. We see some outliers but in general all transpilations lead to a good quality (the variance between results is small). In the middle we see that for circuits with **more** CNOTs the **fidelity** for the **standard transpilation drops significantly**. There are still some circuits that lead to medium fidelities but also many with poor performance, i.e. the variance of the results is very high. However, the **circuit depth** seems to be **low enough** so that **dynamical decoupling can mitigate many errors** and yields considerably better results. On the right plot we see CNOT **numbers** that are **definitely too high for ibmq_ehningen** so that without dynamical decoupling the **fidelity is poor for all transpilations**. Adding dynamical decoupling can in some cases give better results but looking at the high variance we see that there is no guarantee that it works in general.

4.5.3 Fidelity, $p = 1$, Different Dates

The **error rates** of quantum computers are not static but **significantly change over time**. In the following figures we observe that this has a **drastic effect** on the **quality** of the results obtained from ibmq_ehningen.
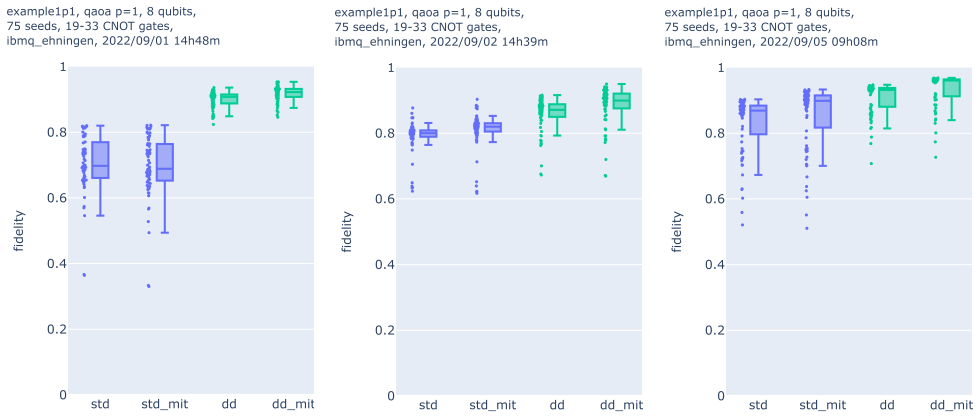
(a) Example1p1.
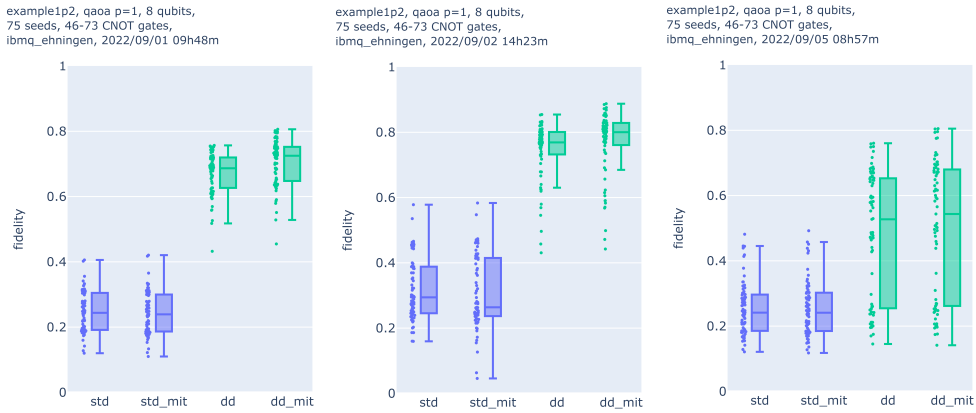
(b) Example1p2.

Figure 4.19: Fidelity vs. transpilation method for example1p1 (upper plots) and example1p2 (lower plots) on three different dates. Every dot stems from one of 75 transpilation seeds.

Seeing how drastically the quality of our results changes from day to day we recommend running experiments on a series of different dates and (if possible) on different quantum computers.

4.5.4 Fidelity and Expectation Value, $p = 1$ and $p = 2$

In this section we want to analyze the effect of the parameter p. In particular, we are interested in the **trade-off** between **better approximation quality** but **longer circuits** that come with higher values of p.
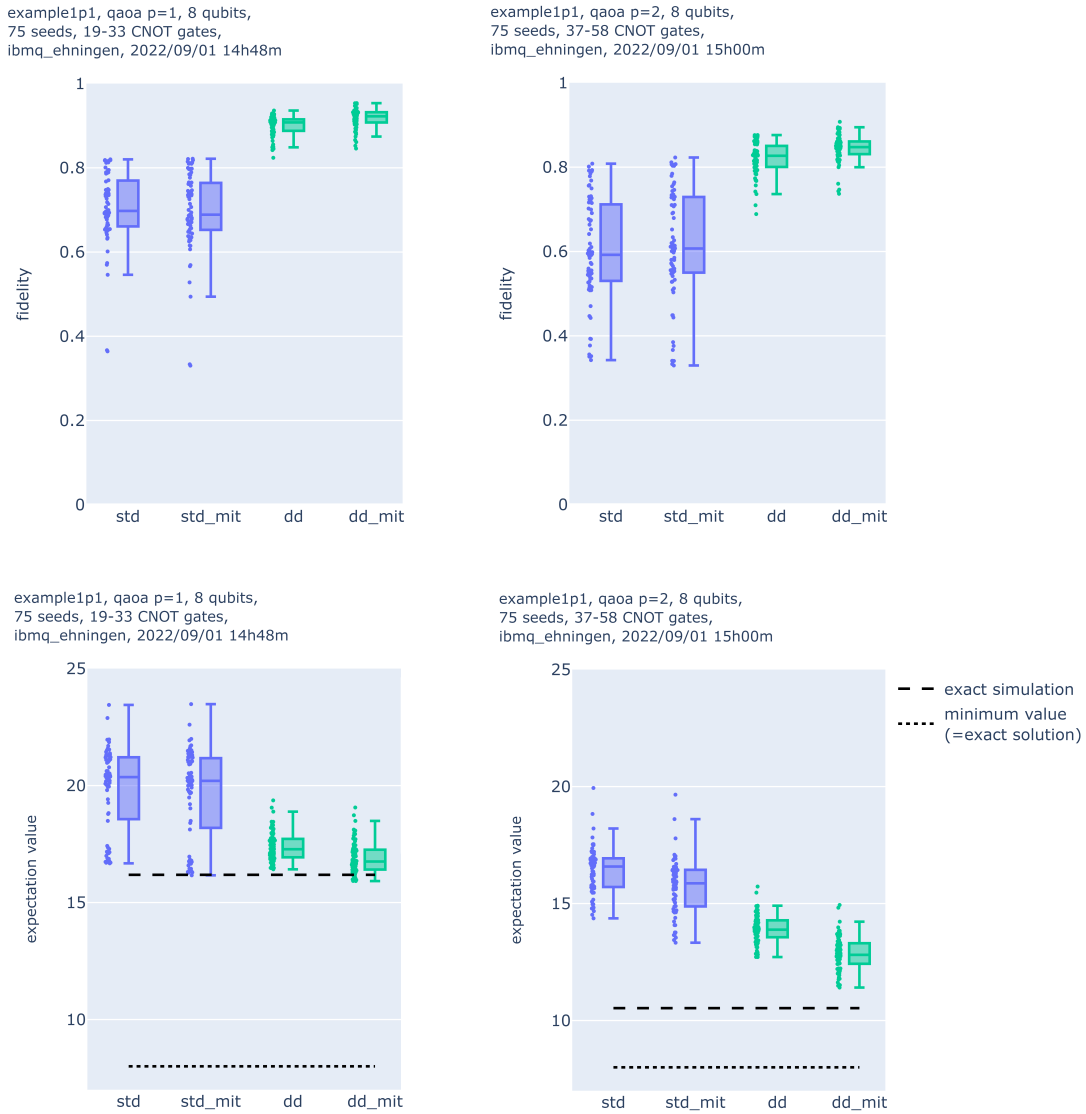
Figure 4.20: Fidelity (upper plots) and expectation value (lower plots) vs. transpilation method for example1p1. On the left we have $p = 1$ and on the right $p = 2$. Every dot stems from one of 75 transpilation seeds.

For the simplest example that we consider (i.e. example1p1) we see in Figure 4.20 that choosing $p = 2$ gives a better result (in terms of a lower expectation value). In theory this is expected, compare the dashed lines in the two lower plots. However, in practice on real quantum computers the deeper circuits for $p = 2$ could be a problem, but for the example at hand the circuits are shallow enough so that we don't run into problems when executing them on ibmq_ehningen. This will change in the next examples.
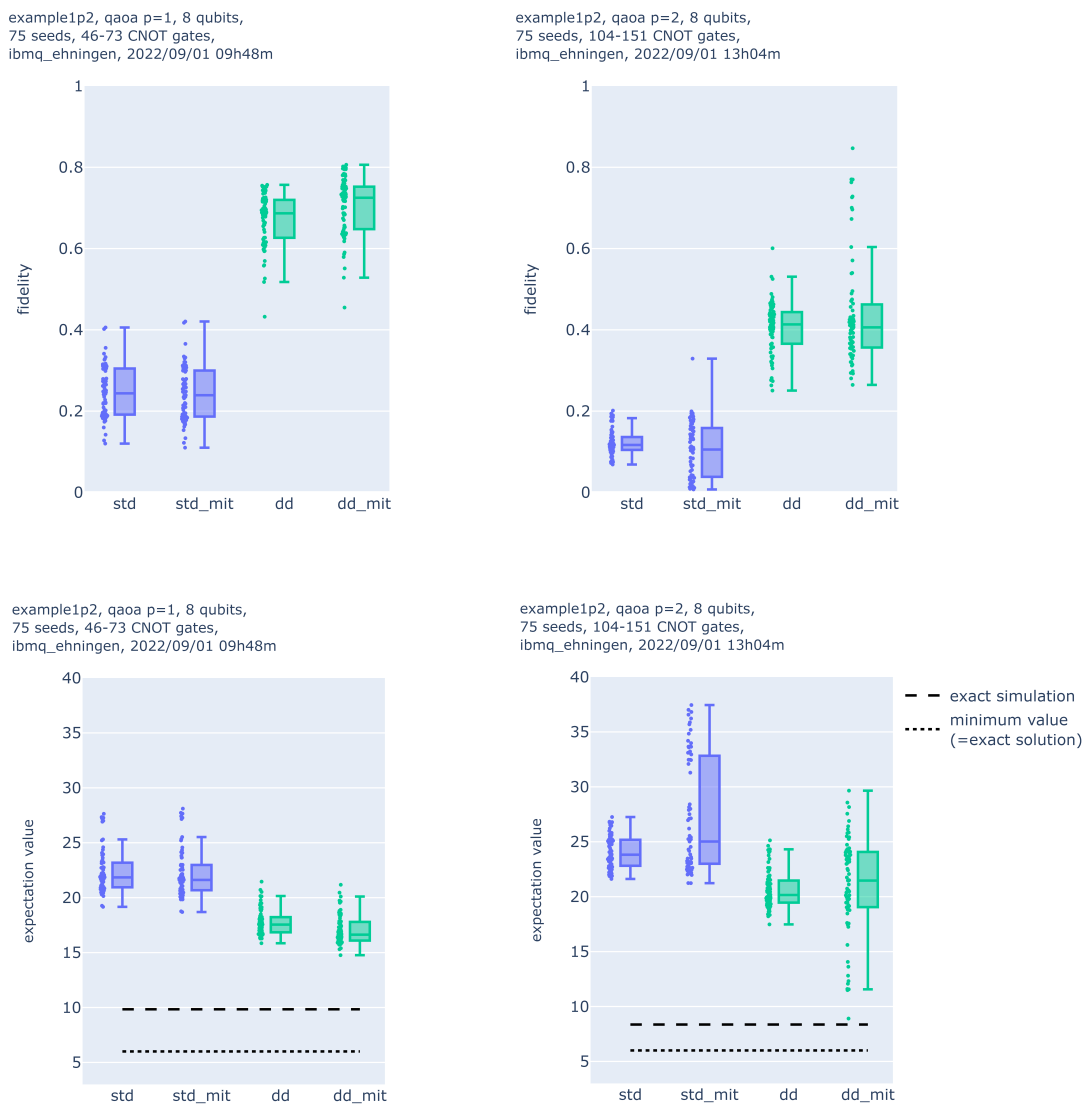
example1p2, qaoa p=1, 8 qubits,
75 seeds, 46-73 CNOT gates,
ibmq_ehningen, 2022/09/01 09h48m

example1p2, qaoa p=2, 8 qubits,
75 seeds, 104-151 CNOT gates,
ibmq_ehningen, 2022/09/01 13h04m

example1p2, qaoa p=1, 8 qubits,
75 seeds, 46-73 CNOT gates,
ibmq_ehningen, 2022/09/01 09h48m

example1p2, qaoa p=2, 8 qubits,
75 seeds, 104-151 CNOT gates,
ibmq_ehningen, 2022/09/01 13h04m

Figure 4.21: Fidelity (upper plots) and expectation value (lower plots) vs. transpilation method for example1p2. On the left we have $p = 1$ and on the right $p = 2$. Every dot stems from one of 75 transpilation seeds.

Already for example1p2 we see in Figure 4.21 that the trade-off between better theoretical solution and practical result speak rather in favor of $p = 1$. For this choice we get more certainly a solution with a good expectation value. Choosing $p = 2$ might give a better solution but looking at the high variance this is pretty uncertain.
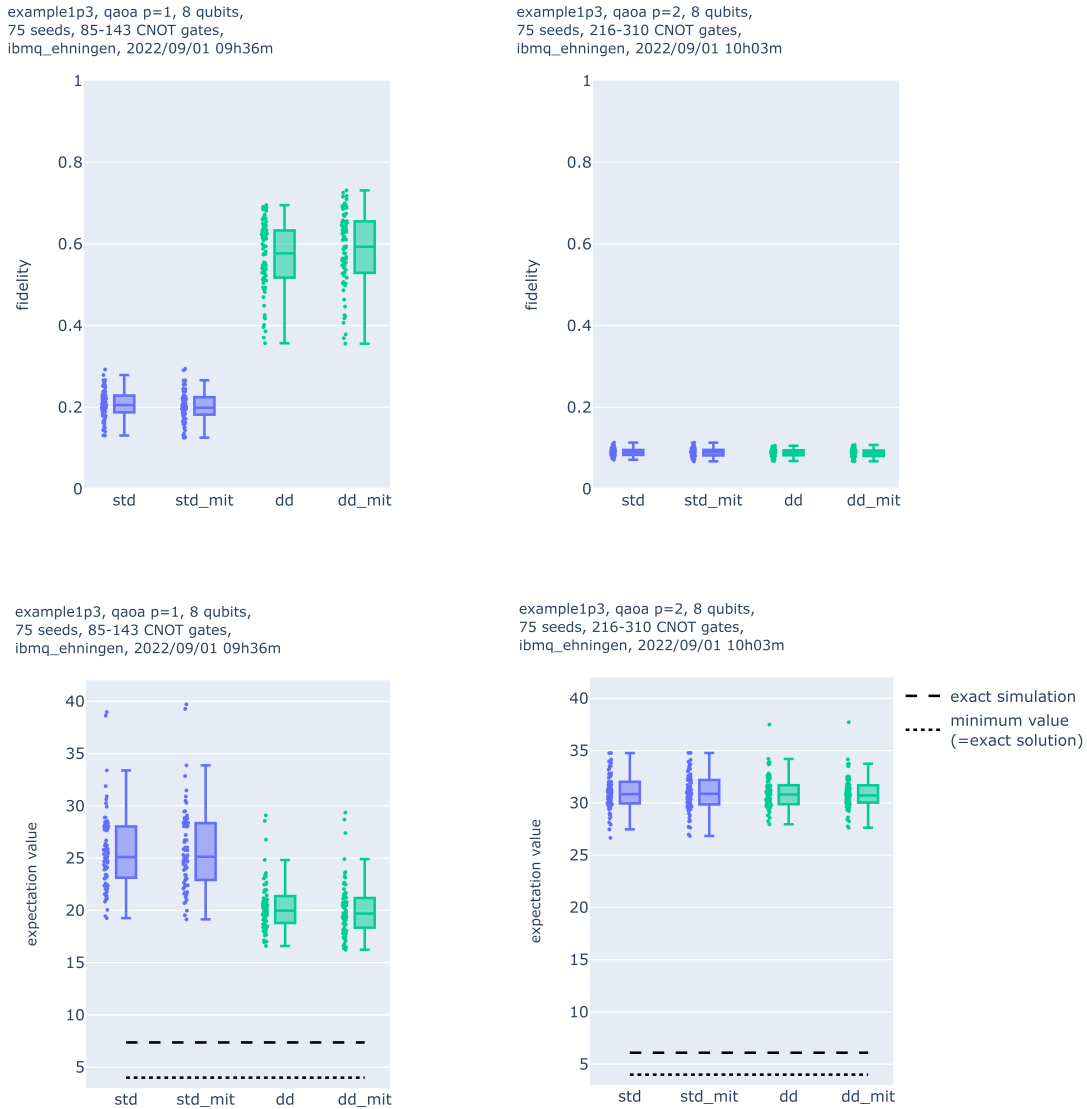
Figure 4.22: Fidelity (upper plots) and expectation value (lower plots) vs. transpilation method for example1p3. On the left we have $p = 1$ and on the right $p = 2$. Every dot stems from one of 75 transpilation seeds.

For example1p3 the situation is clear: The QAOA circuits for $p = 2$ are too deep for ibmq_ehningen so that no meaningful result can be obtained, see Figure 4.22.

4.5.5 Number CNOT Gates vs. Fidelity

Next, we present figures showing the dependence of the fidelity that the circuit achieved with the number of CNOT gates in the circuit.
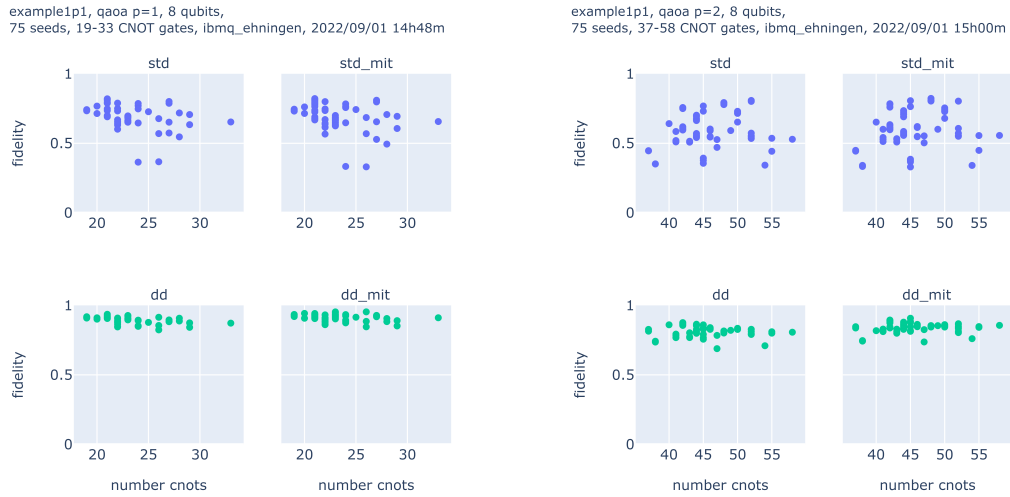
Figure 4.23: Fidelity vs. number of CNOT gates for example1p1 for different transpilations and $p = 1$ (left plots) as well as $p = 2$ (right plots). Every dot stems from one of 75 transpilation seeds.

Comparing the left and the right plots (i.e. comparing $p = 1$ and $p = 2$) we see that circuits with more CNOT gates give (in general) results with a lower fidelity. Moreover, we see in the top plots that for the standard transpilation we don't have the expected decrease in fidelity when increasing the number of CNOTs. In particular, in the top right plot we see the best results for a medium number of CNOT gates and a very poor quality for the lowest number of CNOTs. The explanation is probably that the **standard transpilation method** is **not aware of all kinds of errors** that appear in a quantum device and thus did not choose the best qubits. A hint in this direction is also the remarkable fact that for the circuits with dynamical decoupling the fidelity stays nearly constant for the range of number of CNOT gates appearing in the examples here. It seems that **dynamical decoupling removed the errors that spoiled the quality** for the top plots.

For example1p2 and example1p3 below we make the same observations, but additionally observe that dynamical decoupling and measurement error mitigation are not sufficient to remove all appearing errors, see the plots with CNOT range between 40 and 150. (One reason might be cross-talk between the qubits). For examples with more than 200 CNOT gates we don't get a meaningful solution.
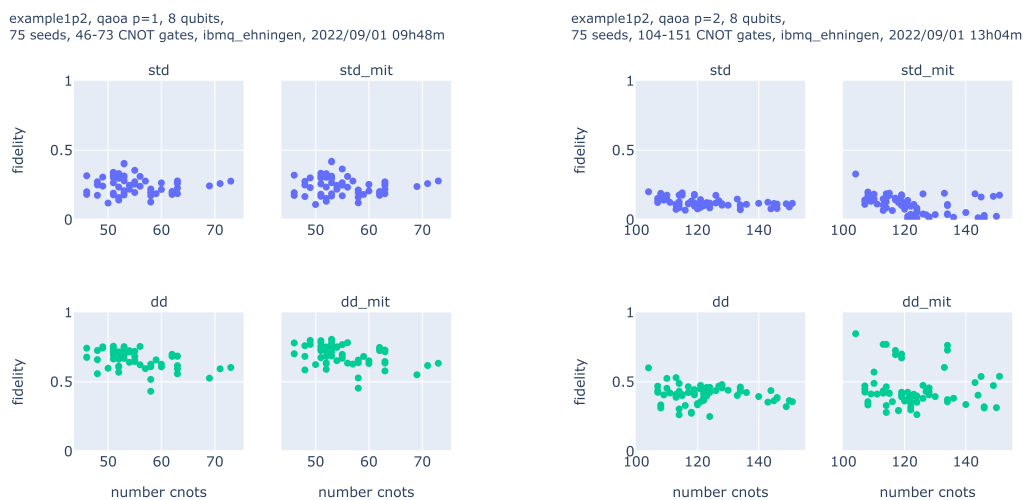
Figure 4.24: Fidelity vs. number of CNOT gates for example1p2 for different transpilations and $p = 1$ (left plots) as well as $p = 2$ (right plots). Every dot stems from one of 75 transpilation seeds.
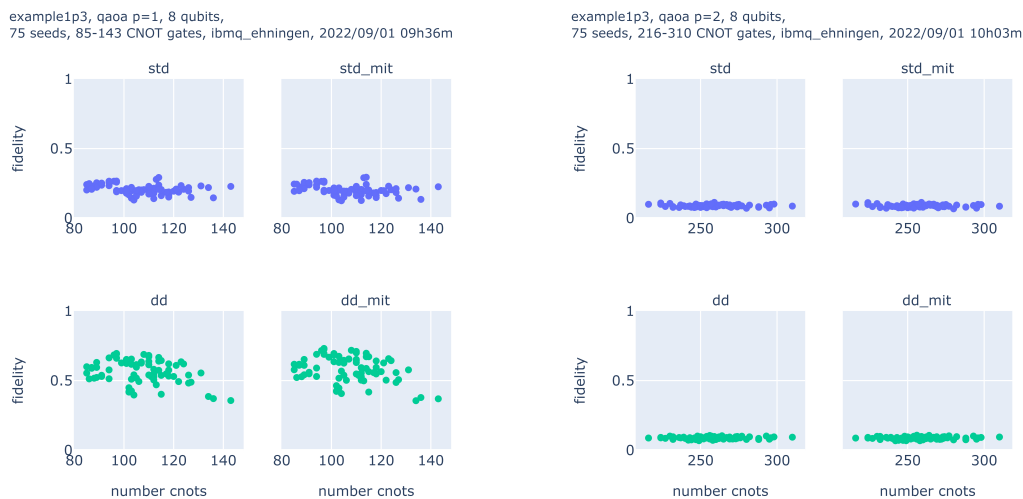

Figure 4.25: Fidelity vs. number of CNOT gates for example1p3 for different transpilations and $p = 1$ (left plots) as well as $p = 2$ (right plots). Every dot stems from one of 75 transpilation seeds.

Example1p3: Different Dates

As mentioned above the error rates of quantum computers change significantly over time. The effect on the quality of our results is clearly visible in Figure 4.26.
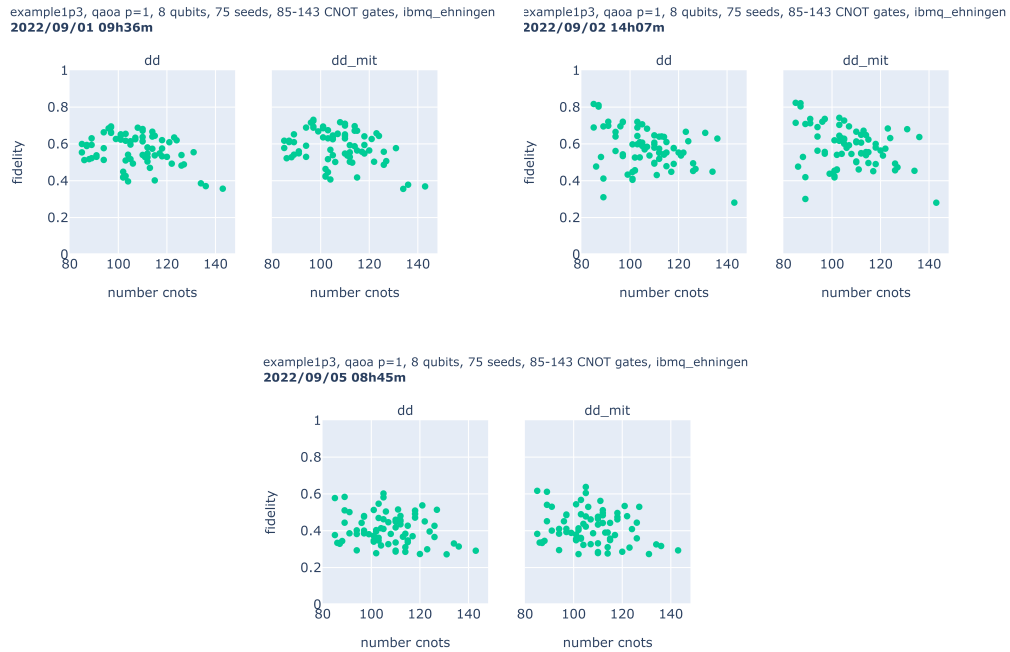
Figure 4.26: Fidelity vs. number of CNOT gates for example1p3 for $p = 1$. The circuits for upper left plot were run on 2022/09/01, for the upper right on 2022/09/02 and for the lower plot on 2022/09/05. Every dot stems from one of 75 transpilation seeds.

We end this notebook with results from Example Series 2.

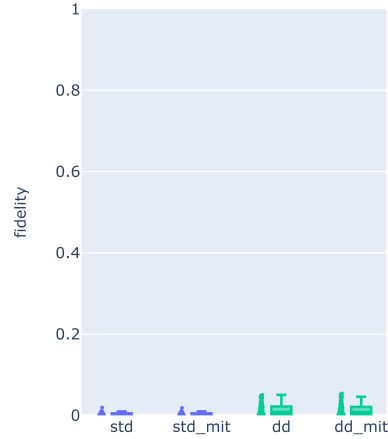### 4.5.6   Fidelity and Expectation Value, $p = 1$ and $p = 2$

For all examples of Example Series 2 we are in regimes of number of CNOT gates that are too high for current quantum computers. Thus, we see poor fidelities and expectation values that stay bounded away from an exact simulation and also from the exact solution.

Example2p1

example2p1, qaoa p=1, 16 qubits,
75 seeds, 92-162 CNOT gates,
ibmq_ehningen, 2022/09/01 10h31m
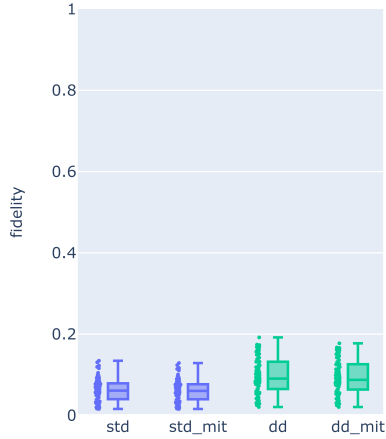
example2p1, qaoa p=2, 16 qubits,
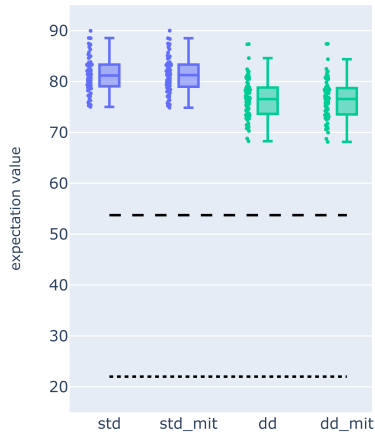75 seeds, 211-325 CNOT gates,
ibmq_ehningen, 2022/09/01 13h37m

example2p1, qaoa p=1, 16 qubits,
75 seeds, 92-162 CNOT gates,
ibmq_ehningen, 2022/09/01 10h31m

example2p1, qaoa p=2, 16 qubits,
75 seeds, 211-325 CNOT gates,
ibmq_ehningen, 2022/09/01 13h37m

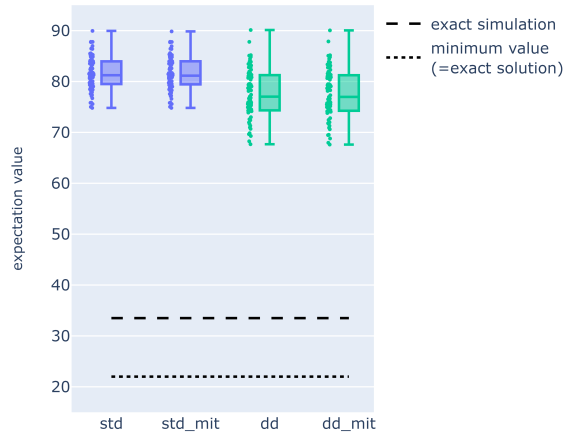Figure 4.27: Fidelity (upper plots) and expectation value (lower plots) vs. transpilation method for example2p1. On the left we have $p = 1$ and on the right $p = 2$. Every dot stems from one of 75 transpilation seeds.

Example2p4

Figure 4.28: Fidelity (upper plots) and expectation value (lower plots) vs. transpilation method for example2p4. On the left we have $p = 1$ and on the right $p = 2$. Every dot stems from one of 75 transpilation seeds.
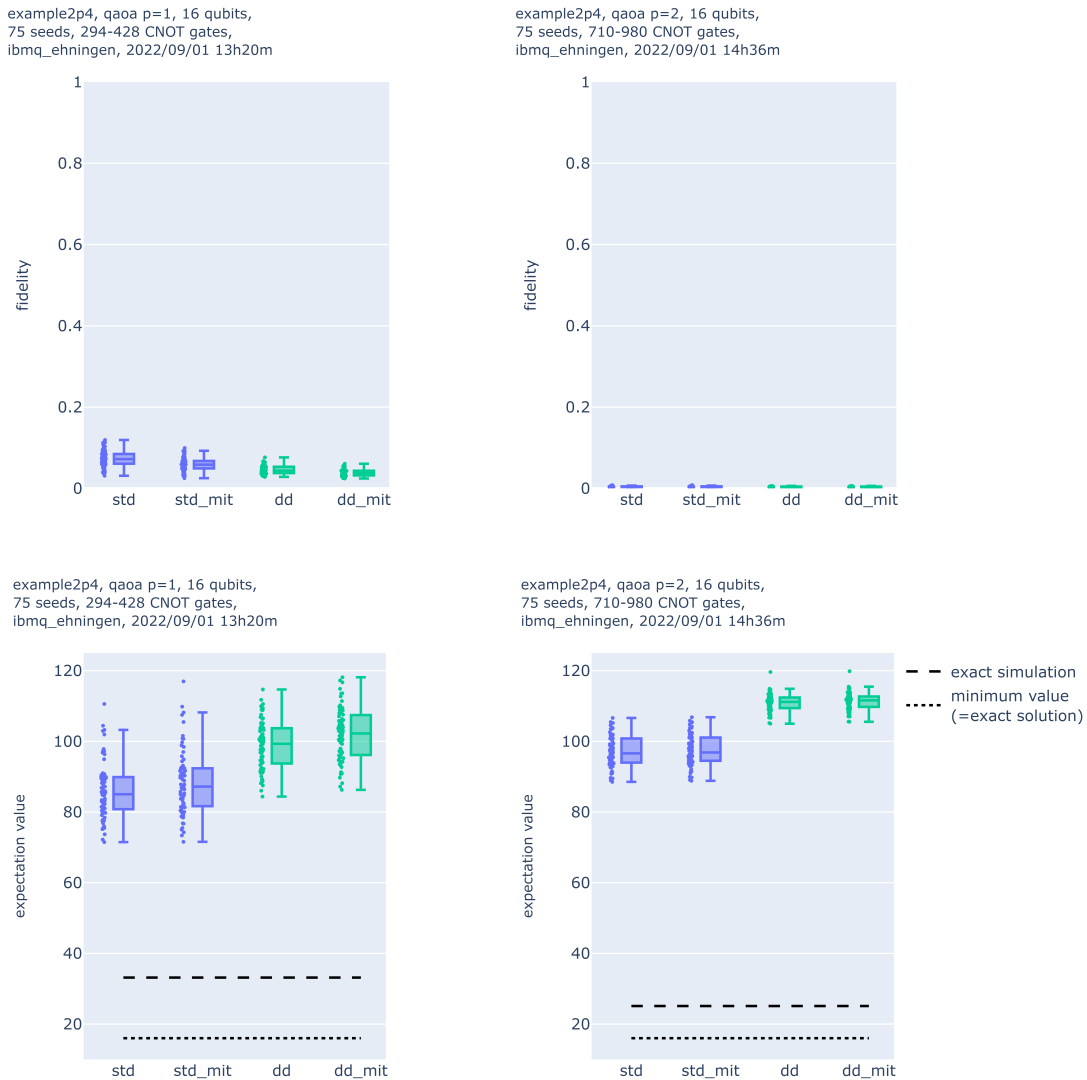
4.6 Remarks on Implementation

All experiments in this notebook were run using the following package versions:

- `mthree==1.0`
- `qiskit==0.37`
- `qiskit_aer==0.10.4`
- `qiskit_terra==0.21.0`

Example	penalty ϱ	QAOA parameters	
example1p1, $p = 1$	$\varrho = 3.1$	$\beta_0 = 0.86470653,$	$\gamma_0 = -0.31959353$
example1p2, $p = 1$	$\varrho = 3.5$	$\beta_0 = 0.70449479,$	$\gamma_0 = 4.48896374$
example1p3, $p = 1$	$\varrho = 1.3$	$\beta_0 = 2.48199357,$	$\gamma_0 = 4.86664818$
example1p1, $p = 2$	$\varrho = 3.2$	$\beta_0 = 1.50891553,$	$\gamma_0 = 2.59570247$
		$\beta_1 = 2.37353917,$	$\gamma_1 = 2.3102415$
example1p2, $p = 2$	$\varrho = 3.6$	$\beta_0 = 3.99890724,$	$\gamma_0 = 6.11303759$
		$\beta_1 = 2.72012026,$	$\gamma_1 = 1.75840967$
example1p3, $p = 2$	$\varrho = 1.6$	$\beta_0 = 0.48954047,$	$\gamma_0 = 1.98033881$
		$\beta_1 = 0.9098274,$	$\gamma_1 = 3.92848365$
example2p1, $p = 1$	$\varrho = 5.7$	$\beta_0 = 0.48207365,$	$\gamma_0 = 6.11153126$
example2p4, $p = 1$	$\varrho = 3.1$	$\beta_0 = 2.77112559,$	$\gamma_0 = 0.02398761$
example2p1, $p = 2$	$\varrho = 5.5$	$\beta_0 = 1.60523831,$	$\gamma_0 = 0.08836239$
		$\beta_1 = 3.88382278,$	$\gamma_1 = 6.19852592$
example2p4, $p = 2$	$\varrho = 3.3$	$\beta_0 = 1.50645202,$	$\gamma_0 = 5.76044569$
		$\beta_1 = 3.84358483,$	$\gamma_1 = 3.7570749$

Table 4.1: Parameters for the experiments in Section 4.5.

For the experiments in Section 4.5 we used the parameters given in Table 4.1.

Appendix A

Properties of ibmq_ehningen

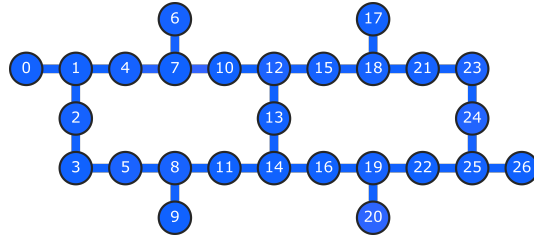The following properties are from November, 11, 2022.

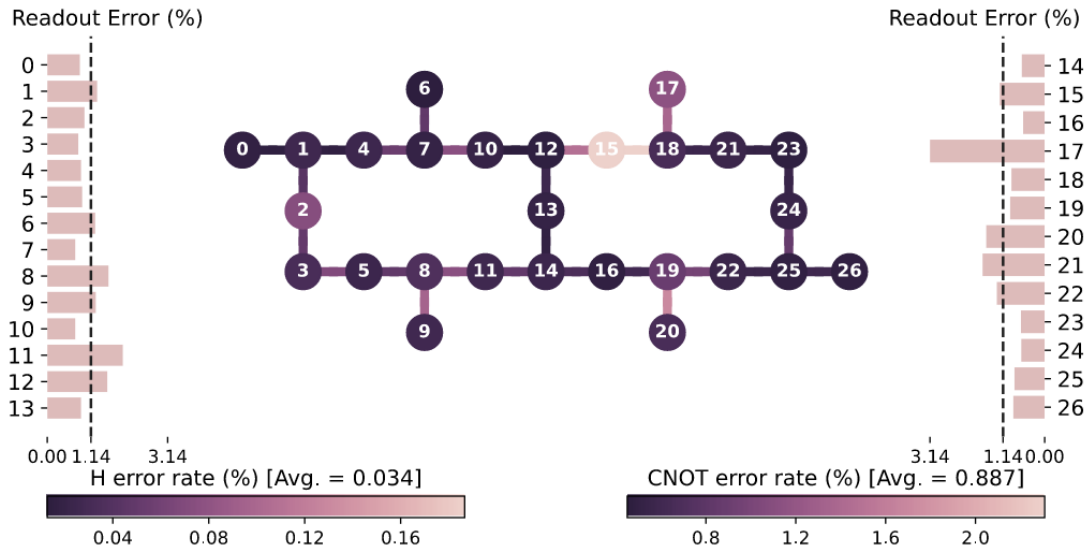Figure A.1: Coupling map of ibmq_ehningen.
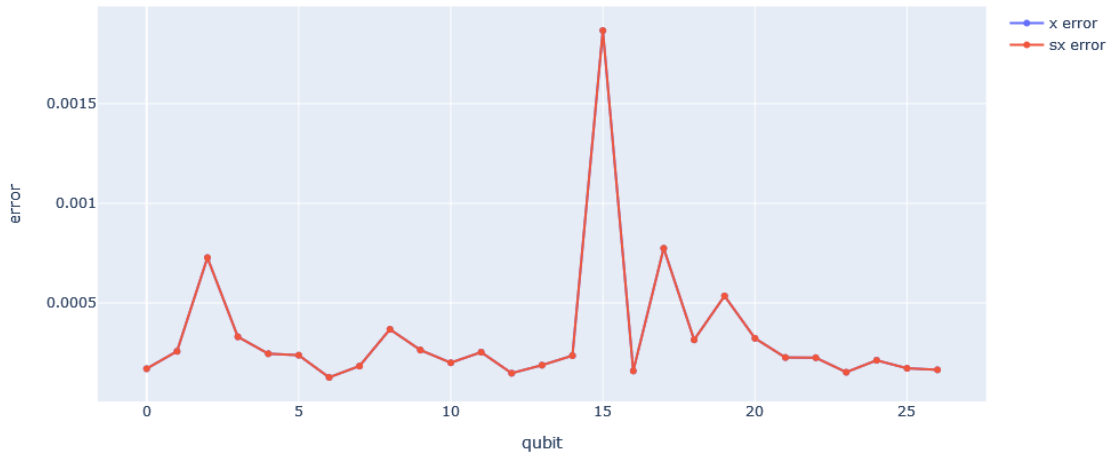

Figure A.2: Error map of ibmq_ehningen.

Figure A.3: X and $\sqrt{\mathrm{X}}$ errors of ibmq_ehningen.
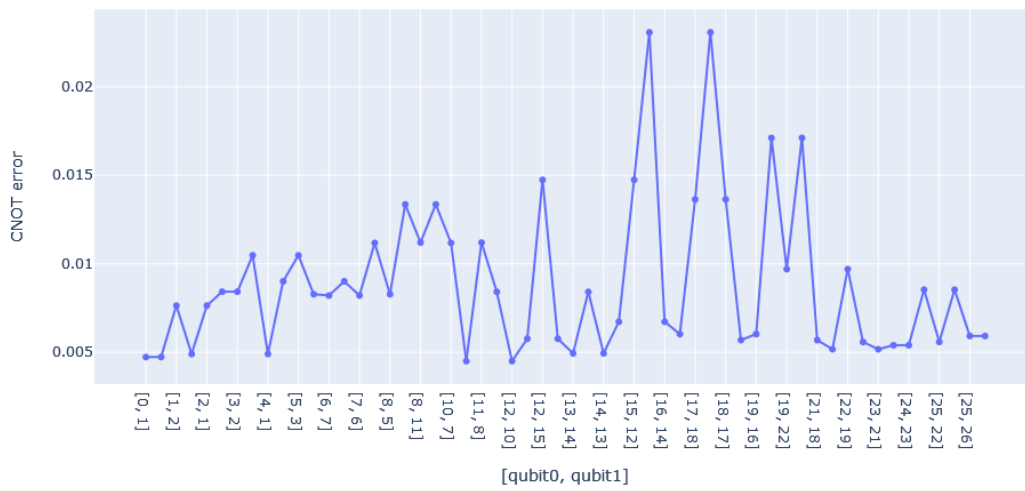

Figure A.4: CNOT errors of ibmq_ehningen.

Appendix B

Codes of Helper Functions

Code for `codes_notebook_1.py`:

```python
from typing import List, Union
import numpy as np
from qiskit_optimization import QuadraticProgram
from qiskit_optimization.converters import QuadraticProgramConverter,␣
 ↪LinearEqualityToPenalty, IntegerToBinary
from qiskit_optimization.algorithms import CplexOptimizer

## --- Codes from Notebook 1 --- ##

class Car:
    def __init__(
        self,
        car_id: str, # an arbitrary name for the car
        time_slots_at_charging_unit: List[int], # time slots when the car is at␣
 ↪the charging unit
        required_energy: int # energy units that should be charged
    ) -> None:
        self.car_id = car_id
        self.time_slots_at_charging_unit = time_slots_at_charging_unit
        self.required_energy = required_energy

    def __str__(self) -> str:
        return f"Car '{self.car_id}':\n" \
            f"  at charging station at time slots {self.
 ↪time_slots_at_charging_unit}\n" \
            f"  requires {self.required_energy} energy units"

class ChargingUnit:
    def __init__(
        self,
```

```python
        charging_unit_id: str, # an arbitrary name for the charging unit
        number_charging_levels: int,
        number_time_slots: int,
    ) -> None:
        self.charging_unit_id = charging_unit_id
        self.number_charging_levels = number_charging_levels
        self.number_time_slots = number_time_slots
        self.cars_to_charge = []

    def __str__(self) -> str:
        info_cars_registered = ""
        for car in self.cars_to_charge:
            info_cars_registered = info_cars_registered + " " + car.car_id
        return "Charging unit with\n" \
            "  charging levels: " + str(list(range(self.
→number_charging_levels)))[1:-1] + "\n" \
            "  time slots: " + str(list(range(self.number_time_slots)))[1:-1] +␣
→"\n" \
            "  cars to charge:" + info_cars_registered

    def register_car_for_charging(self, car: Car) -> None:
        if max(car.time_slots_at_charging_unit) > self.number_time_slots - 1:
            raise ValueError("From car required time slots not compatible with␣
→charging unit.")
        self.cars_to_charge.append(car)

    def reset_cars_for_charging(self) -> None:
        self.cars_to_charge = []

    def generate_constraint_matrix(self) -> np.ndarray:
        """Matrix with ones for times when car is at charging station
         and with zeros if car is not at charging station"""
        number_cars_to_charge = len(self.cars_to_charge)
        constraint_matrix = np.zeros(
            (number_cars_to_charge, number_cars_to_charge*self.
→number_time_slots))
        for row_index in range(0, number_cars_to_charge):
            offset = row_index*self.number_time_slots
            cols = np.array(self.cars_to_charge[row_index].
→time_slots_at_charging_unit)
            constraint_matrix[row_index, offset+cols] = 1
        return constraint_matrix

    def generate_constraint_rhs(self) -> np.ndarray:
        """Vector with required energy as entries"""
        number_cars_to_charge = len(self.cars_to_charge)
        constraint_rhs = np.zeros((number_cars_to_charge, 1))
```

```python
        for row_index in range(0, number_cars_to_charge):
            constraint_rhs[row_index] = self.cars_to_charge[row_index].
 ↪required_energy
        return constraint_rhs

    def generate_cost_matrix(self) -> np.ndarray:
        number_cars_to_charge = len(self.cars_to_charge)
        return np.kron(
            np.ones((number_cars_to_charge, 1)) @ np.ones((1,␣
 ↪number_cars_to_charge)),
            np.eye(self.number_time_slots))

def generate_qcio(
    charging_unit: ChargingUnit,
    name: str=None
) -> QuadraticProgram:
    if name is None:
        name = ""
    qcio = QuadraticProgram(name)

    for car in charging_unit.cars_to_charge:
        qcio.integer_var_list(
            keys=[f"{car.car_id}_t{t}" for t in range(0, charging_unit.
 ↪number_time_slots)],
            lowerbound=0,
            upperbound=charging_unit.number_charging_levels-1,
            name="power.")

    constraint_matrix = charging_unit.generate_constraint_matrix()
    constraint_rhs = charging_unit.generate_constraint_rhs()
    for row_index in range(0, constraint_matrix.shape[0]):
        qcio.linear_constraint(
            linear=constraint_matrix[row_index, :],
            rhs=constraint_rhs[row_index][0],
            sense="==",
            name=f"charge_correct_energy_for_{charging_unit.
 ↪cars_to_charge[row_index].car_id}")

    cost_matrix = charging_unit.generate_cost_matrix()
    qcio.minimize(quadratic=cost_matrix)

    return qcio

class Converter(QuadraticProgramConverter):
    def __init__(
        self,
        penalty: float=None # the penalty paramter for step 1
```

```python
    ) -> None:
        super().__init__()
        self._penalty = penalty
        self.linear_equality_to_penalty_converter =␣
 ↪LinearEqualityToPenalty(penalty)
        self.integer_to_binary_converter = IntegerToBinary()

    def convert(self, quadratic_program: QuadraticProgram) -> QuadraticProgram:
        return self.integer_to_binary_converter.convert(
            self.linear_equality_to_penalty_converter.convert(quadratic_program))

    def interpret(self, x: Union[np.ndarray, List[float]]) -> np.ndarray:
        return self.linear_equality_to_penalty_converter.interpret(
            self.integer_to_binary_converter.interpret(x))


## --- Example for Notebook 2 --- ##

def generate_example():
    charging_unit = ChargingUnit(
        charging_unit_id="charging_unit",
        number_charging_levels=4,
        number_time_slots=4)
    car_green = Car(
        car_id="car_green",
        time_slots_at_charging_unit=[0, 1, 2],
        required_energy=4)
    charging_unit.register_car_for_charging(car_green)

    qcio = generate_qcio(charging_unit, name="QCIO")
    converter = Converter(penalty=3.6)
    qubo = converter.convert(qcio)
    number_binary_variables = qubo.get_num_binary_vars()

    cplex_optimizer = CplexOptimizer()
    qubo_minimization_result = cplex_optimizer.solve(qubo)

    return charging_unit, car_green, qcio, converter, qubo,␣
 ↪number_binary_variables, qubo_minimization_result
```

Code for `codes_notebook_2.py`:

```python
from qiskit.circuit.library.n_local import QAOAAnsatz
from codes_notebook_1 import generate_example as generate_example_notebook_1
```

```python
def generate_example():
    charging_unit, car_green, qcio, converter, qubo, number_binary_variables,␣
↪qubo_minimization_result = generate_example_notebook_1()

    ising, ising_offset = qubo.to_ising()

    qaoa_reps = 2
    qaoa_circuit = QAOAAnsatz(cost_operator=ising, reps=qaoa_reps)
    qaoa_circuit.measure_all()

    return charging_unit, car_green, qcio, converter, qubo,␣
↪number_binary_variables, qubo_minimization_result, ising, ising_offset,␣
↪qaoa_reps, qaoa_circuit
```

Code for utils.py:

```python
    from typing import Union, List
    import numpy as np
    from datetime import datetime
    from pathlib import Path
    import pickle
    import plotly.graph_objects as go
    from qiskit.circuit import QuantumCircuit

    def plot_charging_schedule(
            charging_unit,
            minimization_result_x,
            marker_size=50,
    ) -> go.Figure:
        marker_colors = ["green", "orange", "blue", "red", "magenta",␣
↪"goldenrod"]
        time_slots = np.arange(0, charging_unit.number_time_slots)
        fig = go.Figure()
        already_in_legend = []
        for t in time_slots:
            offset = 0
            for car_num in np.arange(0, len(charging_unit.cars_to_charge)):
                car_id_current_car = charging_unit.cars_to_charge[car_num].car_id
                minimization_result_x_current_car = minimization_result_x[
                    car_num*charging_unit.number_time_slots:
↪(car_num+1)*charging_unit.number_time_slots]
                power_t = minimization_result_x_current_car[t]
                fig.add_trace(go.Scatter(
                    x=[t+0.5]*int(power_t),
```

```python
                    y=offset + np.arange(0, power_t),
                    mode="markers",
                    marker_symbol="square",
                    marker_size=marker_size,
                    marker_color=marker_colors[car_num],
                    name=car_id_current_car,
                    showlegend=False if car_id_current_car in already_in_legend␣
↪else True
                ))
                offset += power_t
                if power_t > 0:
                    already_in_legend.append(car_id_current_car)

    fig.update_xaxes(
        tick0=1,
        dtick=1,
        range=[0.01, charging_unit.number_time_slots],
        tickvals=np.arange(0.5, charging_unit.number_time_slots),
        ticktext=np.arange(0, charging_unit.number_time_slots),
        title="time slot",
        title_font_size=12,
    )
    fig.update_yaxes(
        range=[-0.6, charging_unit.number_charging_levels-1],
        tickvals=np.arange(-0.5, charging_unit.number_charging_levels-0.5),
        ticktext=np.arange(0, charging_unit.number_charging_levels),
        title="charging level",
        title_font_size=12,
        zeroline=False
    )
    return fig

def convert_to_date_and_time_string(time_stamp: Union[datetime, str]):
    if isinstance(time_stamp, datetime):
        output = str(time_stamp.year) + "_" + \
            str(time_stamp.month).rjust(2, '0') + "_" + \
            str(time_stamp.day).rjust(2, '0') + "-" + \
            str(time_stamp.hour).rjust(2, '0') + "h" + \
            str(time_stamp.minute).rjust(2, '0') + "m"
    elif isinstance(time_stamp, str):
        output = time_stamp[0:17].replace('-', '_').replace('T', '-').
↪replace(':', 'h', 1).replace(':', 'm', 1)
    else:
        raise ValueError("data type of 'time_stamp' not supported")
    return output

def save_token(token: str, file_name: str):
```

```python
        path_token_file = Path(file_name).with_suffix(".pickle")
        if path_token_file.exists():
            print("Token already saved.")
        else:
            with open(path_token_file, 'wb') as file:
                pickle.dump(token, file)
            print(f"Token has been saved in '{file_name}.pickle'.")

    def load_token(file_name: str):
        path_token_file = Path(file_name).with_suffix(".pickle")
        try:
            with open(path_token_file, 'rb') as file:
                token = pickle.load(file)
        except FileNotFoundError:
            raise FileNotFoundError("Token has not been saved. Use the function␣
↪save_token to to save your token.")
        print("Token loaded.")
        return token

    def count_gates(
            quantum_circuit: QuantumCircuit,
            gates_to_consider: List[str]
        ) -> int:
        result = 0
        for gate in gates_to_consider:
            try:
                count_gate = quantum_circuit.count_ops()[gate]
            except KeyError:
                count_gate = 0
            result = result + count_gate
        return result
```

Bibliography

[1] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio, and P. J. Coles. Variational Quantum Algorithms. *Nature Reviews Physics*, pages 1–20, 2021. ISSN 2522-5820. https://www.nature.com/articles/s42254-021-00348-9.

[2] E. Farhi, J. Goldstone, and S. Gutmann. A Quantum Approximate Optimization Algorithm, 2014. http://arxiv.org/pdf/1411.4028v1.

[3] F. Guerra. Spin Glasses, 2005. https://arxiv.org/pdf/cond-mat/0507581.

[4] S. Hadfield, Z. Wang, B. O'Gorman, E. G. Rieffel, D. Venturelli, and R. Biswas. From the Quantum Approximate Optimization Algorithm to a Quantum Alternating Operator Ansatz. *Algorithms*, 12(2):34, 2019. ISSN 1999-4893. http://arxiv.org/pdf/1709.03489v2.

[5] Fraunhofer IAO. LamA Laden am Arbeitsplatz, January 20, 2023 (online). https://www.iao.fraunhofer.de/de/forschung/smart-energy-and-mobility-solutions/lama-laden-am-arbeitsplatz.html.

[6] Fraunhofer IAO. LamA - Laden am Arbeitsplatz, January 20, 2023 (online). https://www.lama.zone/.

[7] IBM. Operator Flow, January 20, 2023 (online). https://github.com/Qiskit/qiskit-tutorials/blob/master/tutorials/operators/01_operator_flow.ipynb.

[8] IBM. Using Classical Optimization Solvers and Models with Qiskit Optimization, January 20, 2023 (online). https://qiskit.org/documentation/optimization/tutorials/11_using_classical_optimization_solvers_and_models.html.

[9] IBM. Transpiler Passes and Pass Manager, January 20, 2023 (online). https://qiskit.org/documentation/tutorials/circuits_advanced/04_transpiler_passes_and_passmanager.html.

[10] IBM. Using the Qiskit Compiler, January 20, 2023 (online). https://quantum-enablement.org/how-to/compiling/compiling.html.

[11] IBM. Mesurement Error Mitigation Using M3, January 20, 2023 (online). https://quantum-enablement.org/how-to/mitigation/M3/m3_mitigation.html.

[12] IBM. Noisy Real Hardware, Noise in Quantum Computers, January 20, 2023 (online). https://learn.qiskit.org/summer-school/2022/noisy-real-hardware-noise-quantum-computers.

[13] Fraunhofer IEQ. LamA - Laden am Arbeitsplatz (Charging at Work), January 20, 2023 (online). https://www.ieg.fraunhofer.de/en/references/lama.html.

[14] S. Karimi and P. Ronagh. Practical Integer-to-Binary Mapping for Quantum Annealers. *Quantum Information Processing*, 18(4):042314, 2019. ISSN 1570-0755. https://arxiv.org/pdf/1706.01945.

[15] G. Koßmann, L. Binkowski, C. Tutschku, and R. Schwonnek. Open-Shop Scheduling with Hard Constraints, 2023. https://arxiv.org/abs/2211.05822.

[16] J. R. McClean, S. Boixo, V. N. Smelyanskiy, R. Babbush, and H. Neven. Barren Plateaus in Quantum Neural Network Training Landscapes. *Nature Communications*, 9(1):4812, 2018. ISSN 2041-1723. https://www.nature.com/articles/s41467-018-07090-4.

[17] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 10th anniversary edition edition, 2010. ISBN 9781107002173.

[18] Encyclopedia of Mathematics. Ising Model, January 20, 2023 (online). https://encyclopediaofmath.org/index.php?title=Ising_model.

[19] IBM Quantum Enabling Technologies Team. Dynamical decoupling, January 20, 2023 (online). https://quantum-enablement.org/how-to/dynamical_decoupling/dynamical_decoupling.html.

[20] Mthree Team. mthree, January 20, 2023 (online). https://qiskit.org/documentation/partners/mthree/.

[21] Qiskit Development Team. Transpiler, January 20, 2023 (online). https://qiskit.org/documentation/apidoc/transpiler.html.

[22] Qiskit Optimization Development Team. Optimization Tutorials, January 20, 2023 (online). https://qiskit.org/documentation/optimization/tutorials/index.html.