11 Programming Quantum Computers

Kristel Michielsen, Madita Willsch, Dennis Willsch
Jülich Supercomputing Centre
Forschungszentrum Jülich

Contents

1 Introduction **2**

2 Gate-based quantum computing **2**
2.1 Quantum bits and gates 2
2.2 Programming and simulating quantum circuits 8
2.3 Example: Quantum adder 10
2.4 Example: Quantum approximate optimization algorithm 13

3 Quantum annealing **17**
3.1 Optimization problems with binary variables 17
3.2 Working principle of a quantum annealer 18
3.3 Architecture of D-Wave quantum annealers 20
3.4 Limitations 21
3.5 Programming a D-Wave quantum annealer 22
3.6 Example: Garden optimization 29

A JUQCS standard gate set **31**

E. Pavarini and E. Koch (eds.)
Simulating Correlations with Computers
Modeling and Simulation Vol. 11
Forschungszentrum Jülich, 2021, ISBN 978-3-95806-529-1
http://www.cond-mat.de/events/correl21

1 Introduction

Quantum computing is a new emerging computer technology. Current quantum computing devices are at a development stage where they gradually become suitable for small real-world applications. This lecture is devoted to the practical aspects of programming such quantum computing devices.

Over the past twenty years, two major paradigms of quantum computing have emerged. The first is the *gate-based model of quantum computing* (also known as *universal quantum computing*), and the second is *quantum annealing* (also known as *adiabatic quantum computing*). From a mathematical point of view, both models have the same computational power, but in practice they operate in a fundamentally different way.

The first part of this lecture focuses on gate-based quantum computers. We will define the basic unit of computation, the quantum bit (qubit), and how a quantum computer processes information. Subsequently, basic quantum circuits (i.e., the *programs* for gate-based quantum computers) are discussed and simulated. Finally, a more complex algorithm called the *quantum approximate optimization algorithm* (QAOA), which is considered to be an approach to address small optimization problems, is introduced and discussed.

In the second part of this lecture, we give an introduction to quantum annealing and discuss how to program a quantum annealer, i.e., the quantum processing unit (QPU) that performs the quantum annealing process. The introductory part starts with a discussion of discrete optimization problems and a formulation of the particular set of problems that can be solved on currently available quantum annealers. Subsequently, we describe the working principles of quantum annealers and the architecture of the currently available quantum annealers by D-Wave Systems Inc. We also discuss physical aspects including some limitations. Finally, we demonstrate how to program a D-Wave quantum annealer by means of some example programs.

2 Gate-based quantum computing

This section provides a hands-on introduction to the programming of gate-based quantum computers. After introducing the basic notions of qubits and gates, several examples of quantum circuits are programmed and discussed. These are either fundamental building blocks in disruptive quantum circuit scenarios, such as the quantum Fourier transform in Shor's factoring algorithm [1], or potentially relevant for near-term applications such as the QAOA [2]. In this section, the term *quantum computer* always refers to the gate-based model of quantum computing.

2.1 Quantum bits and gates

2.1.1 Single qubits

A gate-based quantum computer is designed to process information in terms of quantum bits (qubits). The word *qubit* is derived from the basic unit of computation in a digital computer, a

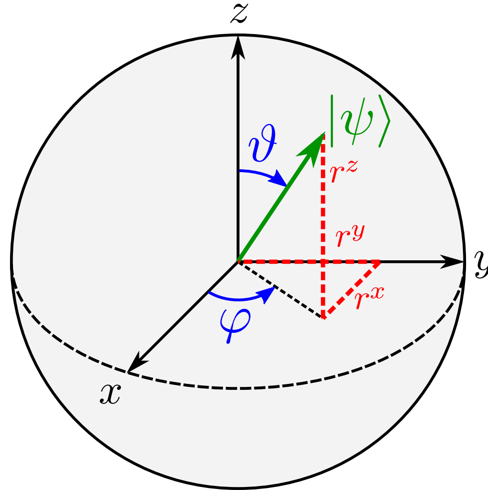


Fig. 1: *Bloch sphere representation of a pure single-qubit state $|\psi\rangle$. The azimuthal angle $\vartheta \in [0, \pi]$ and the polar angle $\varphi \in [0, 2\pi)$ are defined in Eq. (3), and the Cartesian coordinates r^x, r^y, and r^z are given by Eq. (5).*

binary digit or *bit*. While a bit in a digital computer can only ever be either 0 or 1, a qubit is a generalization of a bit in the sense that it can also be in a superposition of 0 and 1.
We describe a qubit $|\psi\rangle$ in terms of two complex numbers $\psi_0, \psi_1 \in \mathbb{C}$,

$$|\psi\rangle = \psi_0|0\rangle + \psi_1|1\rangle = \begin{pmatrix} \psi_0 \\ \psi_1 \end{pmatrix}, \tag{1}$$

which are normalized such that the norm of $|\psi\rangle$ is $\langle\psi|\psi\rangle = |\psi_0|^2 + |\psi_1|^2 = 1$. In the quantum computer model, the notions of 0 and 1 are represented by the standard vectors

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{2}$$

For the sake of programming quantum computers, these two notations are equivalent. We call $|\psi\rangle$ the *state vector* of the qubit.
Informally, a complex superposition like Eq. (1) is sometimes described as "0 and 1 at the same time", although it is important to realize that the notion of time plays no role here. Equation (1) is a well-defined mathematical object that completely describes the state of a single qubit.
A very useful representation of the general single-qubit state $|\psi\rangle$ in Eq. (1) is called the *Bloch sphere representation* that is shown in Fig. 1. It is particularly convenient to visualize the states and operations on a single qubit. We obtain the Bloch sphere representation by using the fact that $\langle\psi|\psi\rangle = |\psi_0|^2 + |\psi_1|^2 = 1$, which implies that there exists an angle $\vartheta \in [0, \pi]$ such that $|\psi_0| = \cos(\vartheta/2)$ and $|\psi_1| = \sin(\vartheta/2)$. Furthermore, as the global phase of a quantum state is irrelevant, we can choose without loss of generality $\psi_0 = \cos(\vartheta/2)$ and $\psi_1 = e^{i\varphi}\sin(\vartheta/2)$, where $\varphi \in [0, 2\pi)$ represents the relative phase between the complex coefficients. We thus obtain

$$|\psi\rangle = \cos\frac{\vartheta}{2}\,|0\rangle + e^{i\varphi}\sin\frac{\vartheta}{2}\,|1\rangle. \tag{3}$$

For all values of ϑ and φ, this state can be drawn on the surface of a 3D sphere with radius one as shown in Fig. 1.
When a qubit is measured, one always obtains one of the two discrete, digital outcomes "0" and "1". The complex coefficients of $|\psi\rangle$ determine the corresponding *probabilities* $p_0 = |\psi_0|^2$ and $p_1 = |\psi_1|^2$ to measure one of the two outcomes. On the Bloch sphere, the probabilities p_0 and p_1 can be obtained from the projection of $|\psi\rangle$ onto the z axis.

Exercise 1: Calculate ϑ and φ for the following states, visualize them on the Bloch sphere with radius one, and compute the probabilities to measure the qubit in $|0\rangle$ and $|1\rangle$:
(a) $|0\rangle$, **(b)** $|1\rangle$, **(c)** $(|0\rangle + |1\rangle)/\sqrt{2}$, **(d)** $(|0\rangle + i|1\rangle)/\sqrt{2}$, **(e)** $(\sqrt{3}/2)|0\rangle + ((1+i)/2\sqrt{2})|1\rangle$,
(f) $((1+i)/2)|0\rangle + ((1-\sqrt{3}i)/\sqrt{8})|1\rangle$ (hint: remove the global phase here first).

The Cartesian coordinates r^x, r^y, and r^z of the single-qubit state $|\psi\rangle$ in Fig. 1 can be computed as expectation values of the Pauli matrices,

$$\sigma^x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad \sigma^y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \qquad \sigma^z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{4}$$

A short calculation yields

$$\vec{r} = \begin{pmatrix} r^x \\ r^y \\ r^z \end{pmatrix} = \begin{pmatrix} \langle\psi|\sigma^x|\psi\rangle \\ \langle\psi|\sigma^y|\psi\rangle \\ \langle\psi|\sigma^z|\psi\rangle \end{pmatrix} = \begin{pmatrix} \sin\vartheta\cos\varphi \\ \sin\vartheta\sin\varphi \\ \cos\vartheta \end{pmatrix}. \tag{5}$$

2.1.2 Quantum gates

A *quantum gate* is a unitary operation that can be performed on a qubit. All single-qubit quantum gates can be visualized as rotations of $|\psi\rangle$ on the Bloch sphere in Fig. 1. One defines three elementary qubit rotations as matrix exponentials of the Pauli matrices in Eq. (4)

$$R^x(\theta) = e^{-i\theta\sigma^x/2} = \begin{pmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix}, \tag{6}$$

$$R^y(\theta) = e^{-i\theta\sigma^y/2} = \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix}, \tag{7}$$

$$R^z(\theta) = e^{-i\theta\sigma^z/2} = \begin{pmatrix} \exp(-i\theta/2) & 0 \\ 0 & \exp(i\theta/2) \end{pmatrix}. \tag{8}$$

Here, the quantum gate $R^\alpha(\theta)$ for $\alpha = x, y, z$ rotates the qubit $|\psi\rangle$ by an angle θ around the axis α according to the *right-hand rule*. This means that if the thumb of the right hand points along the corresponding axis α in Fig. 1, the sense of rotation is given by the curl of the remaining fingers, i.e., counter-clockwise when looking at the top of the thumb. An example for the gate $R^y(\pi)$ is shown in Fig. 2.

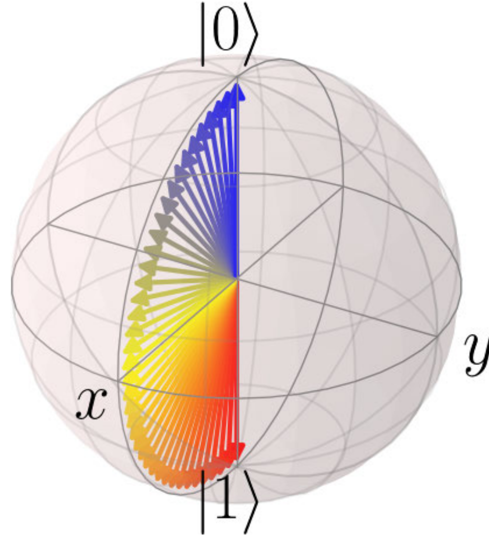


Fig. 2: *Visualization of the single-qubit gate $R^y(\pi)$ (see Eq. (7)) applied to the state $|0\rangle$ on the Bloch sphere defined in Fig. 1. The gate represents a counter-clockwise rotation around the y axis by an angle of π. Shown is the time evolution of a qubit during the application of a pulse designed to implement the gate $R^y(\pi)$. The time is encoded in the color of the arrows (from blue at the beginning over yellow to red at the end of the pulse). The data is taken from a simulation of transmon qubits [3].*

Often, at least one of these elementary qubit rotations is implemented in a hardware realization of a gate-based quantum computer. When programming quantum computers, the quantum gates are internally decomposed into products of such elementary rotations. For instance, for the current generation quantum processors of the IBM Q (which are based on superconducting transmon qubits [4]), the elementary rotations are $R^x(\pi/2)$ and $R^z(\theta)$ [5,6].
A general single-qubit rotation by an angle θ around a unit axis $\vec{n} = (n^x, n^y, n^z)$ is given by

$$R^{\vec{n}}(\theta) = e^{-i\theta\vec{n}\cdot\vec{\sigma}/2} = \cos\frac{\theta}{2}\, I - i\sin\frac{\theta}{2}\,\vec{n}\cdot\vec{\sigma}, \tag{9}$$

where $\vec{n}\cdot\vec{\sigma} = n^x\sigma^x + n^y\sigma^y + n^z\sigma^z$, and I is the $2{\times}2$ identity matrix. All single-qubit gates can be written in this form, up to an arbitrary global phase factor of the form $e^{i\alpha}$.
Besides the elementary single-qubit rotations, there are six other important gates that belong to the so-called standard gate set

$$X = \sigma^x, \qquad Y = \sigma^y, \qquad Z = \sigma^z, \tag{10}$$

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \qquad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \qquad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}. \tag{11}$$

Especially the X gate (also known as the NOT gate or *bit flip* gate) and the Hadamard gate H (which maps a state $|0\rangle$ to a uniform superposition of $|0\rangle$ and $|1\rangle$ and back) are used in many applications. A comprehensive list of common quantum gates is given in appendix A.
As each quantum gate U is unitary (i.e., $U^{-1} = U^\dagger$ where $U^\dagger$ denotes the Hermitian conjugate), the inverse of a quantum gate U is again a quantum gate.

Exercise 2: For all single-qubit gates in Eqs. (10) and (11) and their inverses, find the corresponding axes $\vec{n}$ and angles θ (and optionally the global phase factors $e^{i\alpha}$) to express them in the form of Eq. (9) and visualize their operations as rotations on the Bloch sphere, as in Fig. 2.

2.1.3 Multiple qubits

While a single-qubit state is described by two complex coefficients ψ_0 and ψ_1 (see Eq. (1)), a multi-qubit state $|\psi\rangle$ with $n > 1$ qubits is described by 2^n complex coefficients $\psi_0, \ldots, \psi_{2^n-1}$,

$$|\psi\rangle = \psi_0|0\cdots00\rangle + \psi_1|0\cdots01\rangle + \cdots + \psi_{2^n-1}|1\cdots11\rangle = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{2^n-1} \end{pmatrix}, \tag{12}$$

The corresponding basis vectors $|q_0q_1\cdots q_{n-1}\rangle$ for $q_i = 0, 1$ and $i = 0, \ldots, n-1$ are constructed from the single-qubit standard basis in Eq. (2) by means of the *tensor product* "$\otimes$" (also known as *Kronecker product*), $|q_0q_1\cdots q_{n-1}\rangle = |q_0\rangle \otimes |q_1\rangle \otimes \cdots \otimes |q_{n-1}\rangle$. For simplicity, we often do not write the tensor product explicitly. Consequently, for two qubits, the computational basis reads

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}. \tag{13}$$

One may notice that in Eq. (12), the basis state $|q_0q_1\cdots q_{n-1}\rangle$ corresponding to the coefficient ψ_j for $j = 0, \ldots, 2^n-1$ contains the binary representation of j, i.e., $\mathrm{bin}(j) = q_0q_1\cdots q_{n-1}$, or equivalently, $j = \sum_{i=0}^{n-1} q_i \times 2^{n-i-1}$. For this reason, we identify the notations $|j\rangle \equiv |\mathrm{bin}(j)\rangle \equiv |q_0q_1\cdots q_{n-1}\rangle$ so that the state in Eq. (12) is also written as

$$|\psi\rangle = \sum_{j=0}^{2^n-1} \psi_j|j\rangle. \tag{14}$$

This notation is needed for the example of the quantum Fourier transform discussed below.
Quantum gates on multiple qubits are, like single-qubit gates, unitary operations on the multi-qubit state $|\psi\rangle$. In practice, most multi-qubit gates are actually single-qubit gates acting on certain qubits in the multi-qubit state. For instance, a single-qubit gate U from Eqs. (10) or (11) acting on a certain qubit i (denoted by U_i) transforms a basis vector $|q_0\cdots q_{n-1}\rangle$ according to

$$U_i|q_0\cdots q_{n-1}\rangle = |q_0\cdots q_{i-1}\rangle(U|q_i\rangle)|q_{i+1}\cdots q_{n-1}\rangle. \tag{15}$$

In other words, U_i is given by the tensor product $U_i = I \otimes \cdots \otimes U \otimes \cdots \otimes I$.
Another common set of multi-qubit gates derived from single-qubit gates are *controlled* quantum gates. For a single-qubit gate U, the controlled-U gate (denoted by CU) acts on two qubits

i_1 and i_2 in a multi-qubit state. Its action on a basis vector $|q_0 \cdots q_{n-1}\rangle$ is defined by

$$\mathrm{C}U_{i_1 i_2}|q_0 \cdots q_{n-1}\rangle = \begin{cases} |q_0 \cdots q_{n-1}\rangle & (\text{if } q_{i_1} = 0) \\ |q_0 \cdots q_{i_2-1}\rangle(U|q_{i_2}\rangle)|q_{i_2+1} \cdots q_{n-1}\rangle & (\text{if } q_{i_1} = 1) \end{cases}. \tag{16}$$

In other words, the action is controlled by qubit q_{i_1}: Only if the control qubit q_{i_1} is in state 1, the target qubit q_{i_2} experiences the single-qubit gate U. On the two-qubit space spanned by the four basis states in Eq. (13), the matrix representation of the controlled-U gate is given by

$$\mathrm{C}U = \begin{pmatrix} I & \mathbf{0} \\ \mathbf{0} & U \end{pmatrix}, \tag{17}$$

where $\mathbf{0}$ denotes a $2{\times}2$ matrix with all elements equal to zero.
It is important to realize that for controlled quantum gates constructed in this way, the global phase of the single-qubit gate U becomes a relative phase. In particular, this means that, even though the single-qubit gates S and $R^z(\pi/2)$ are equivalent, the controlled gates CS and C$R^z(\pi/2)$ are different two-qubit gates.
Two very important two-qubit gates constructed like this are the controlled-NOT (CNOT or CX) and the controlled-phase (CZ) gates. Their matrix representations with respect to the two-qubit basis in Eq. (13) are given by

$$\mathrm{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \qquad \mathrm{C}Z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}. \tag{18}$$

The CNOT gate and the CZ gate can be converted into one another using the identity $\mathrm{CNOT} = (I \otimes H)\,\mathrm{C}Z\,(I \otimes H)$, which can be verified by computing the product of the corresponding matrix representations. On a space with more than two qubits, the same identity reads $\mathrm{CNOT}_{i_1 i_2} = H_{i_2}\,\mathrm{C}Z_{i_1 i_2}\,H_{i_2}$. More of such *circuit identities* that are useful when programming quantum computers can be found in the following exercise and in [7].

Exercise 3: Verify the following circuit identities, e.g. by computing their matrix representations on a suitable space and then confirming that they are equivalent (up to a global phase):
(a) $X = HZH$, **(b)** $Y = HYH$, **(c)** $Z = HXH$, **(d)** $I = XX = YY = ZZ = HH$,
(e) $S = TT$, **(f)** $H = SR^x(\pi/2)S$, **(g)** $HTH = R^x(\pi/4)$, **(h)** $XR^y(\theta)X = R^y(-\theta)$,
(i) $\mathrm{CNOT}_{i_1 i_2} = H_{i_2}\,\mathrm{C}Z_{i_1 i_2}\,H_{i_2}$,
(j) $\mathrm{CNOT}_{i_2 i_1} = H_{i_1} H_{i_2}\,\mathrm{CNOT}_{i_1 i_2}\,H_{i_1} H_{i_2}$,
(k) $\mathrm{CNOT}_{i_2 i_1} = H_{i_1} H_{i_2}\,\mathrm{CNOT}_{i_1 i_2}\,H_{i_1} H_{i_2}$,
(l) $\mathrm{C}Z_{i_1 i_2} = \mathrm{C}Z_{i_2 i_1}$, **(m)** $\mathrm{C}S_{i_1 i_2} = \mathrm{C}S_{i_2 i_1}$,
(n) $\mathrm{C}(e^{i\alpha} I) = R^z(\alpha) \otimes I$,
(o) $R^z_{i_1}(\theta)\,\mathrm{CNOT}_{i_1 i_2} = \mathrm{CNOT}_{i_1 i_2} R^z_{i_1}(\theta)$,
(p) $R^x_{i_2}(\theta)\,\mathrm{CNOT}_{i_1 i_2} = \mathrm{CNOT}_{i_1 i_2} R^x_{i_2}(\theta)$.

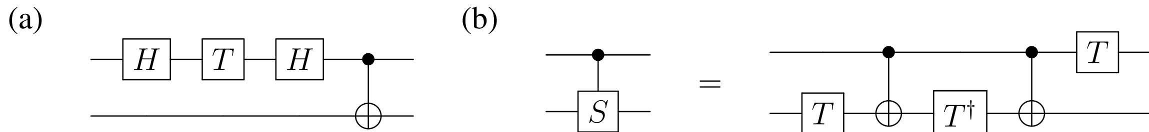


Fig. 3: *Example quantum circuit diagrams. Note that the diagrams are read from left to right. (a) Quantum circuit that generates the state* $\cos(\pi/8)|00\rangle - i\sin(\pi/8)|11\rangle$ *(which can be computed using the circuit identity from exercise 3 (g)) when starting with the initial state* $|00\rangle$ *on the left. (b) Schematic way of writing a circuit identity for the* CS *gate.*

2.2 Programming and simulating quantum circuits

A program for a gate-based quantum computer is called a *quantum circuit*. A quantum circuit is a sequence of several quantum gates. It is often expressed with a diagrammatic language that uses horizontal lines to represent the qubits and boxes to represent the quantum gates. The order of execution in the quantum gate sequence is from left to right. Controlled quantum gates such as $\mathrm{C}U_{i_1 i_2}$ in Eq. (16) are visualized with a filled dot on the control qubit and the single-qubit gate U on the target qubit. The CNOT gate in particular is visualized with an encircled plus symbol on the target qubit. Two examples using this schematic language are shown in Fig. 3.

As described in the previous section, each gate in a quantum circuit represents a unitary matrix. By multiplying all quantum gate matrices in a quantum circuit, we could, in principle, obtain a large unitary matrix that is equivalent to the quantum circuit. Note that, as the order of execution in a quantum circuit diagram is from left to right, the corresponding quantum gate matrices must be multiplied in reverse order. For the right circuit in Fig. 3(a), for instance, this matrix product would read $\mathrm{CNOT}\,(H\otimes I)\,(T\otimes I)\,(H\otimes I)$.

To simulate a quantum circuit, one could use the thus obtained matrix and apply it to a certain initial state, which is typically chosen as $|0\cdots0\rangle = (1,0,\dots,0)^T$. This method works for small quantum circuits. However, it quickly becomes prohibitive because the size of the quantum circuit matrix for N qubits is $2^N\times 2^N$.

Larger quantum circuits with, say, $N \geq 30$ qubits can be simulated on supercomputers such as the GPU-cluster JUWELS Booster using JUQCS–G [8], which is a GPU-accelerated version of the Jülich Universal Quantum Computer Simulator (JUQCS) [9, 10] that was also used for benchmarking purposes in Google's quantum supremacy experiment [11]. For reference, the standard gate set implemented by JUQCS is given in appendix A.

A `qiskit` [12] interface to JUQCS including the conversion from the `qiskit` gate-set to the JUQCS gate-set is available through the Jülich UNified Infrastructure for Quantum computing (JUNIQ) service at `http://jugit.fz-juelich.de/qip/juniq-platform`.

An example program to simulate the circuit in Fig. 3(a) using this interface is shown in listing 1. It simulates the quantum circuit by propagating the state vector, sampling from the resulting probability, and returning the counts for all sampled events (in this case "00" and "11"). Furthermore, instead of sampling, JUQCS also provides an option to return the full state vector up to a certain number of qubits. An example program for this mode of operation is shown in listing 2.

```
1  import qiskit
2
3  circuit = qiskit.QuantumCircuit(2)
4  circuit.h(0)
5  circuit.t(0)
6  circuit.h(0)
7  circuit.cx(0,1)
8  circuit.measure_all()
9
10 from juqcs import Juqcs
11 backend = Juqcs.get_backend('qasm_simulator')
12 backend.allocate(minutes=10, max_qubits=2)
13
14 job = qiskit.execute(circuit.reverse_bits(), backend=backend, shots=1000)
15 result = job.result()
16
17 print(result.get_counts())
18
19 backend.deallocate()
```

Listing 1: *Example program to simulate the quantum circuit shown in Fig. 3(a). As the result of this circuit is* $\cos(\pi/8)|00\rangle - i\sin(\pi/8)|11\rangle$*, the printed counts should correspond to the probabilities* $p_{00} = \cos(\pi/8)^2 \approx 0.85$ *and* $p_{11} = \sin(\pi/8)^2 \approx 0.15$*. Note also the usage of* `circuit.reverse_bits()`*, because* `qiskit` *uses the ordering* $|q_{n-1}\cdots q_0\rangle$ *while all standard text books as well as these lecture notes use* $|q_0\cdots q_{n-1}\rangle$*.*

```
1  import qiskit
2
3  circuit = qiskit.QuantumCircuit(2)
4  circuit.h(0)
5  circuit.t(0)
6  circuit.h(0)
7  circuit.cx(0,1)
8
9  from juqcs import Juqcs
10 backend = Juqcs.get_backend('statevector_simulator')
11 backend.allocate(minutes=10, max_qubits=2)
12
13 job = qiskit.execute(circuit.reverse_bits(), backend=backend)
14 result = job.result()
15
16 print(result.get_statevector())
17
18 backend.deallocate()
```

Listing 2: *Example program to simulate the quantum circuit shown in Fig. 3(a) using the state vector simulator. The result of this program is a numerical representation of the state* $\cos(\pi/8)|00\rangle - i\sin(\pi/8)|11\rangle$*, up to a global phase.*

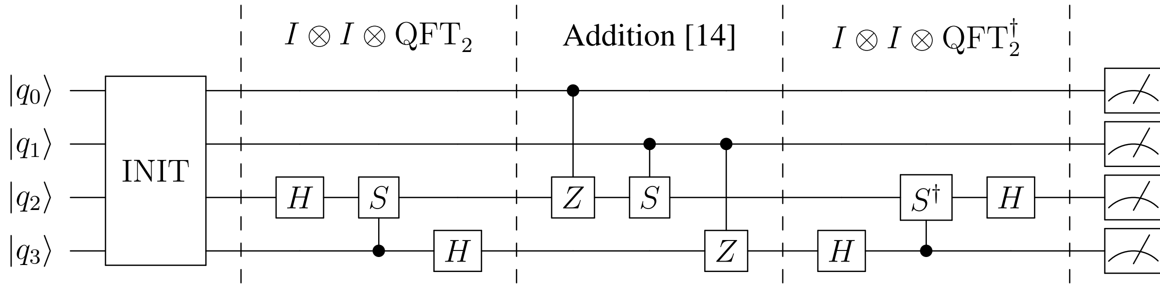


Fig. 4: *Circuit for a quantum adder [14] that adds two two-qubit registers modulo four, according to the rule Eq. (19). The circuit consists of four parts: Initialization of the qubit registers, QFT on the last two registers, the phase addition transform from [14], and another QFT on the last two registers. Note that the swaps that are part of typical QFT circuits (see [7]) are left out. Finally, a measurement of the qubits, which produces a "0" or a "1" for each qubit, is indicated at the end. To rewrite the gates, one can use identities from exercise 3 or [7].*

Instead of the JUQCS backend, one can also use a simulator from `qiskit` via the function `qiskit.Aer.get_backend`. This one only works for circuits with a small number of qubits, but it does not need the calls to `backend.allocate` and `backend.deallocate` as it does not run on a supercomputer. Furthermore, backends for real quantum devices can be accessed in a similar manner through the IBM Q Experience [13].

2.3 Example: Quantum adder

As a first "real-world" example, we consider a four-qubit quantum computer program that adds two two-qubit registers according to the rule

$$|q_0 q_1\rangle|q_2 q_3\rangle \mapsto |q_0 q_1\rangle|q_0 q_1 + q_2 q_3\rangle, \tag{19}$$

where the expression $q_0 q_1 + q_2 q_3$ is to be understood as integer addition (modulo 4) of the two-bit integers with binary representations $q_0 q_1$ and $q_2 q_3$, respectively. After the application of the quantum circuit, the second register contains the sum in binary notation. Some examples of the operation of the quantum adder are

$$|2\rangle|1\rangle \quad \mapsto \quad |2\rangle|3\rangle, \tag{20}$$

$$|2\rangle\frac{|0\rangle + |1\rangle}{\sqrt{2}} \quad \mapsto \quad |2\rangle\frac{|2\rangle + |3\rangle}{\sqrt{2}}, \tag{21}$$

$$|2\rangle\frac{|0\rangle + |1\rangle + |2\rangle}{\sqrt{3}} \quad \mapsto \quad |2\rangle\frac{|2\rangle + |3\rangle + |0\rangle}{\sqrt{3}}. \tag{22}$$

The interesting thing is that it can also add superpositions in parallel, as done in Eqs. (21) and (22) (note that this works because quantum circuits are linear maps, so a circuit U applied to a state $|\psi_1\rangle + |\psi_2\rangle$ produces the state $U|\psi_1\rangle + U|\psi_2\rangle$).

Exercise 3: Use the rule Eq. (19) for the quantum adder to work out the result when the initial state is given by $(|0\rangle + |3\rangle)/\sqrt{2} \otimes (|1\rangle + |2\rangle + |3\rangle)/\sqrt{3}$.

The quantum circuit for the adder consists of four stages that are shown in Fig. 4. The purpose of the INIT block at the beginning is to initialize the registers in a certain initial state, such as the states on the left-hand side in Eqs. (20)–(22).

The second and the fourth stage contain a quantum Fourier transform (QFT) on the second register. The QFT is an operation that, loosely speaking, moves information from the registers to the phases of exponential prefactors and vice versa. For an initial N-qubit basis state $|j\rangle$ ($j = 0, \ldots, 2^N - 1$), the QFT is defined as the unitary transformation

$$|j\rangle \overset{\text{QFT}}{\mapsto} \frac{1}{2^{N/2}} \sum_{k=0}^{2^N-1} e^{2\pi i jk/2^N} |k\rangle, \tag{23}$$

where we identified j and its binary representation $q_0 \cdots q_{N-1}$ as done in Eq. (14). There is a generic quantum circuit [7] to implement this transformation using only H gates and conditional z rotations in a number of steps *polynomial* in N, as opposed to the *Fast Fourier Transform* that requires $\mathcal{O}(N2^N)$ steps. It is an important component of many quantum algorithms for which a theoretical exponential speedup is known. One such algorithm is Shor's factorization algorithm [1] in which the QFT is basically used to find the period of a suitable function (note that finding periods is a generic feature of any Fourier transform).

The third part of the quantum adder circuit in Fig. 4 is the addition transform from [14]. It uses conditional phase shifts from the first to the second register so that an addition is effectively performed in the exponent of the phase factors. For instance, if the first register represents an integer l (e.g. $|l\rangle \equiv |q_0 q_1\rangle$ above), and the second register is given by the QFT of an integer j (i.e. $\text{QFT}|j\rangle \propto \sum_k e^{2\pi i jk/2^N} |k\rangle$), the addition transform performs the operation

$$\sum_{k=0}^{2^N-1} e^{2\pi i jk/2^N} |l\rangle|k\rangle \mapsto \sum_{k=0}^{2^N-1} e^{2\pi i (j+l)k/2^N} |l\rangle|k\rangle. \tag{24}$$

After this step, the inverse QFT (given by $\text{QFT}^\dagger$ in Fig. 4) can be used to move the result $(j+l)$ of the addition from the exponent back into the second register. Note that the addition is automatically implemented modulo 2^N, because the complex exponential function is periodic with period $2\pi i$.

As an example, we consider an implementation of the quantum adder using the single-qubit gate set defined in Eqs. (9)–(11) and the CNOT gate as the only two-qubit gate. This requires rewriting (also known as *transpiling*) the gates using the circuit identities from exercise 3 and Fig. 3(b). The result is shown in Fig. 5 and listing 3. Note that the initial state in this example is chosen to be $|2\rangle(|0\rangle + |1\rangle)/\sqrt{2}$, i.e., the example from Eq. (21) above. As the output contains a superposition of the results, each result occurs as separate event in the simulation.

Fig. 5: *Example circuit for the quantum adder in Fig. 4 after transpiling some of the quantum gates using appropriate circuit identities. Note that in this example, an X and an H gate have been used to explicitly create the initial state for the example in Eq. (21).*

```
import qiskit

circuit = qiskit.QuantumCircuit(4)
circuit.x(0)
circuit.h(3)
circuit.barrier()
circuit.t(1)
circuit.h(2)
circuit.t(3)
circuit.cx(2,3)
circuit.tdg(3)
circuit.t(2)
circuit.cx(2,3)
circuit.h(2)
circuit.cx(0,2)
circuit.h(2)
circuit.cx(2,1)
circuit.tdg(1)
circuit.cx(2,1)
circuit.cx(1,3)
circuit.cx(2,3)
circuit.t(3)
circuit.cx(2,3)
circuit.h(2)
circuit.tdg(3)
circuit.measure_all()

from juqcs import Juqcs
backend = Juqcs.get_backend('qasm_simulator')
backend.allocate(minutes=10, max_qubits=circuit.num_qubits)

job = qiskit.execute(circuit.reverse_bits(), backend=backend, shots=1000)
result = job.result()

print(result.get_counts())

backend.deallocate()
```

Listing 3: *Example program for the quantum adder circuit shown in Fig. 5 for the case given by Eq. (21) with initial state $|2\rangle(|0\rangle + |1\rangle)/\sqrt{2}$. The circuit is simulated with JUQCS as shown in listing 1. The output should be the events "1010" (corresponding to state $|2\rangle|2\rangle$) and "1011" (corresponding to state $|2\rangle|3\rangle$) with roughly 50% frequency each.*

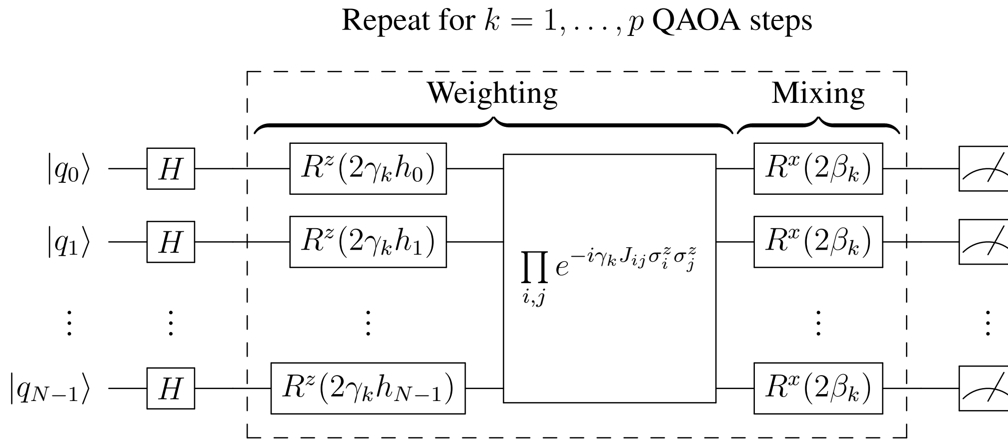


Fig. 6: *General QAOA circuit [2]. Initially, the qubits are brought into a uniform superposition over all states using H gates. Then, $k = 1, \dots, p$ QAOA steps with variational parameters $\beta_1, \dots, \beta_p$ and $\gamma_1, \dots, \gamma_p$ are applied. Each QAOA step k consists of a "weighting step" using z rotations scaled with γ_k and the parameters $\{h_i\}$ and $\{J_{ij}\}$ that define the optimization problem (see Eq. (25)), followed by a "mixing step" using x rotations with angle $2\beta_k$. Finally, the qubits are measured. The result can then be used to update the variational parameters until the energy is sufficiently low.*

2.4 Example: Quantum approximate optimization algorithm

In this section, we consider an approach to address optimization problems on gate-based quantum computers. Gate-based quantum computers are not made to solve optimization problems by design, which is a key difference to the adiabatic quantum computers covered in the next section. However, there is a standard (and by now rather common) way of addressing optimization problems in QUBO/Ising form on gate-based quantum computers, namely the quantum approximate optimization algorithm (QAOA) [2].
The optimization problems considered here are discrete Ising problems. The goal of such problems is to find the minimum of an energy function (or cost function)

$$E(s_0, \dots, s_{N-1}) = \sum_{i=0}^{N-1} h_i s_i + \sum_{i<j} J_{ij} s_i s_j, \tag{25}$$

where $s_0, \dots, s_{N-1} = \pm 1$ are the discrete, two-valued variables, N is the number of variables (which is equal to the number of required qubits), and $\{h_i\}$ and $\{J_{ij}\}$ are real numbers that define the optimization problem instance (for more information see the following section).
The QAOA is a quantum algorithm to find minima (or low-energy states) of Eq. (25). It is a variational quantum algorithm, which means that it has a number of variational parameters $\beta_1, \dots, \beta_p$ and $\gamma_1, \dots, \gamma_p$ that are varied during the iterations of the algorithm. The order of the QAOA, denoted by p, determines the number of variational parameters. It is worth mentioning that for large p, the QAOA can be related to approximate quantum annealing, which also provides a method to find a useful initialization of the variational parameters (see [8]).

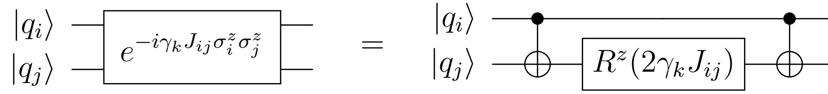


Fig. 7: *Circuit identity to implement the second part of the weighting step in Fig. 6.*

In each iteration of the QAOA, one executes the quantum circuit in Fig. 6 for a given set of variational parameters $\beta_1, \ldots, \beta_p$ and $\gamma_1, \ldots, \gamma_p$ (note that the quantum circuit depends on the fixed problem instance given by $\{h_i\}$ and $\{J_{ij}\}$). To execute the gate $\prod_{i,j} e^{-i\gamma_k J_{ij}\sigma_i^z\sigma_j^z}$ in the weighting step of the QAOA circuit in Fig. 6 using the standard gate set defined above, we need another circuit identity. This identity is shown in Fig. 7.

Exercise 4: Prove the circuit identity in Fig. 7, e.g. by computing the matrix representations of both sides and verifying that they are equivalent.

Each measurement at the end of the circuit produces a string of discrete variables $s_0, \ldots, s_{N-1} = \pm 1$, where the qubit measurement $q_i = 0$ corresponds to $s_i = +1$ and $q_i = -1$ corresponds to $s_i = -1$. Note that this convention, $q_i = (1-s_i)/2$, is standard for gate-based quantum computers [7]; for quantum annealers, one often uses $q_i = (1+s_i)/2$ instead (see below).
From several executions of the circuit, an expectation value for the energy in Eq. (25) can be computed. This result can be used to update the variational parameters and perform the next iteration. This process is continued as long as necessary until it converges.
Note that, when the QAOA is simulated using a state vector simulator (see listing 2), the expectation value can also be computed directly from the state vector $|\psi\rangle$. This is done by replacing the problem variables s_i in Eq. (25) by the Pauli matrices σ_i^z (which yields the *Ising Hamiltonian* of the problem) and computing the expectation value

$$\langle E\rangle = \langle\psi|E(\sigma_0^z, \ldots, \sigma_{N-1}^z)|\psi\rangle. \tag{26}$$

As an example, we consider a three-qubit problem characterized by the energy function

$$E(s_0, s_1, s_2) = -s_0 + \frac{1}{2}s_1 - \frac{1}{2}s_2 + \frac{1}{2}s_0s_1 + \frac{1}{2}s_1s_2. \tag{27}$$

In this case, the problem parameters for Eq. (25) are given by $(h_0, h_1, h_2) = (-1, 1/2, -1/2)$ and $(J_{01}, J_{02}, J_{12}) = (1/2, 0, 1/2)$. The minimum is given by $(s_0, s_1, s_2) = (+1, -1, +1)$ and corresponds to the state vector $|010\rangle$. The energy at the minimum is $E(+1, -1, +1) = -3$.
Constructing the QAOA circuit for $p = 1$ according to Figs. 6 and 7 yields the quantum circuit in Fig. 8. The program to simulate this circuit for a certain range of values for β_1 and γ_1 is shown in listing 4.
Figure 9 shows the expectation value of the energy, computed from the state vector $|\psi\rangle$ according to Eq. (26), and the success probability $|\langle 010|\psi\rangle|^2$. In this case, the energy minimum (see Fig. 9(a)) is very close to the point with maximum success probability (see Fig. 9(b)). Note that this is not guaranteed for larger QAOA applications [8, 15].

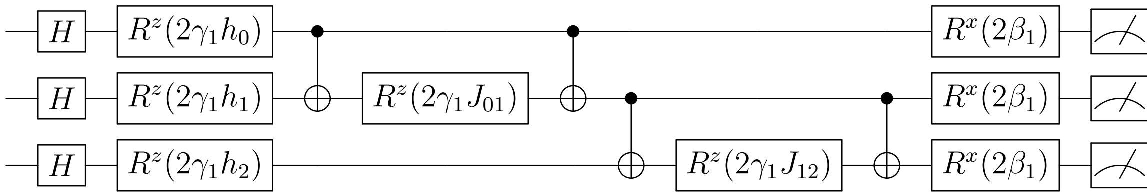


Fig. 8: *Example QAOA circuit for $p = 1$ to find the minimum of the energy function given by Eq. (27). The circuit has three qubits (one for each problem variable s_i). The qubit values q_i after the measurement at the end are related to the problem variables via $q_i = (1 - s_i)/2$ (gate-based convention). Note that only two of the blocks in Fig. 7 are necessary for the weighting step, because the third coupling parameter $J_{02} = 0$.*

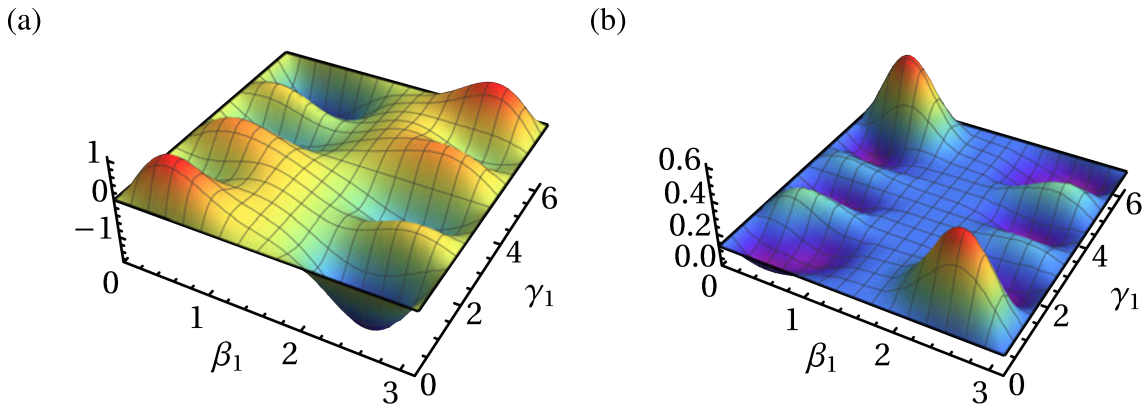


Fig. 9: *Example QAOA result for $p = 1$ that shows (a) the expectation value of the energy $\langle E \rangle$ (see Eq. (26)) and (b) the probability to measure the solution state $|\langle 010|\psi\rangle|^2$. In this case, the energy minimum at $(\beta_1, \gamma_1) \approx (2.505, 0.681)$ is very close to the point $(\beta_1, \gamma_1) \approx (2.524, 0.713)$ where the success probability is maximal. The state vector $|\psi\rangle$ has been obtained by simulating the QAOA circuit in Fig. 8 for the whole range of $\beta_1 \in [0, \pi)$ and $\gamma_1 \in [0, 2\pi)$. Beyond this range, the landscapes can be continued periodically; the periodicity in β_1 and γ_1 is due to the periodicity of the R^x and R^z gates in Fig. 8.*

Exercise 5: Run the $p = 1$ QAOA for the energy function in Eq. (27) by simulating the circuit in Fig. 8. Either perform a grid scan over $\beta_1 \in [0, \pi)$ and $\gamma_1 \in [0, 2\pi)$ (as done to produce Fig. 9), or write an optimization program to find the location of the minimum (e.g. by using the `scipy` package [16]).

Exercise 6: Construct the $p = 2$ QAOA circuit and run it to improve upon the $p = 1$ result. One way to do this is to take the best (β_1, γ_1) from exercise 5 or Fig. 9, and optimize for (β_2, γ_2). Another way would be to consider a case with larger p and take values for (β_k, γ_k) from a quantum annealing schedule as described in [8].

```
1  import qiskit
2
3  h = [-1, 1/2, -1/2]
4  J = [1/2, 1/2]
5
6  def E(s0, s1, s2):
7      return h[0]*s0 + h[1]*s1 + h[2]*s2 + J[0]*s0*s1 + J[1]*s1*s2
8
9  qaoa_circuits = []
10 qaoa_parameters = []
11
12 for beta1 in [2.0, 2.5]:
13     for gamma1 in [0.5, 0.7, 1.0]:
14         circuit = qiskit.QuantumCircuit(3)
15         circuit.h(0)
16         circuit.h(1)
17         circuit.h(2)
18         circuit.rz(2*gamma1*h[0], 0)
19         circuit.rz(2*gamma1*h[1], 1)
20         circuit.rz(2*gamma1*h[2], 2)
21         circuit.cx(0,1)
22         circuit.rz(2*gamma1*J[0], 1)
23         circuit.cx(0,1)
24         circuit.cx(1,2)
25         circuit.rz(2*gamma1*J[1], 2)
26         circuit.cx(1,2)
27         circuit.rx(2*beta1, 0)
28         circuit.rx(2*beta1, 1)
29         circuit.rx(2*beta1, 2)
30         qaoa_circuits.append(circuit.reverse_bits())
31         qaoa_parameters.append([beta1, gamma1])
32
33 from juqcs import Juqcs
34 backend = Juqcs.get_backend('statevector_simulator')
35 backend.allocate(minutes=30, max_qubits=3)
36
37 job = qiskit.execute(qaoa_circuits, backend=backend)
38 result = job.result()
39
40 print('beta1\tgamma1\tenergyexpectation\tsuccessprobability')
41 for i,(beta1,gamma1) in enumerate(qaoa_parameters):
42     psi = result.get_statevector(i)
43     prob = abs(psi)**2
44     energy  = E(+1,+1,+1)*prob[0b000] + E(+1,+1,-1)*prob[0b001]
45     energy += E(+1,-1,+1)*prob[0b010] + E(+1,-1,-1)*prob[0b011]
46     energy += E(-1,+1,+1)*prob[0b100] + E(-1,+1,-1)*prob[0b101]
47     energy += E(-1,-1,+1)*prob[0b110] + E(-1,-1,-1)*prob[0b111]
48     probability = prob[0b010]
49     print(f'{beta1}\t{gamma1}\t{energy}\t{probability}')
50
51 backend.deallocate()
```

Listing 4: *Example program for the $p = 1$ QAOA circuit shown in Fig. 8. Separate circuits are simulated for all given values of β_1 and γ_1. From the resulting state vector, both energy and success probability are evaluated. The best result in the output should be at $\beta_1 = 2.5$ and $\gamma_1 = 0.7$ with energy -1.69 and success probability 58.9%.*

3 Quantum annealing

The second part of these lecture notes focuses on quantum annealing. Besides gate-based quantum computers, quantum annealing has emerged as the second major paradigm of quantum computing. As quantum annealers are somewhat simpler to manufacture, much larger devices of 5000+ qubits are already available and the technology is closer to the verge of technological maturity.

For this reason, the quantum annealing programs discussed in section 3.5 of these lecture notes exclusively target real devices, in contrast to the simulators discussed in the gate-based case above. Furthermore, with the garden optimization problem [17] as an application, the program type reflects the typical kind of problems solved on current quantum annealers that are a little bit closer to actual real-world applications.

3.1 Optimization problems with binary variables

Optimization of parameters in high-dimensional spaces can, in general, be a (computationally) demanding task. Gradient-based algorithms as well as non-gradient based algorithms usually require many evaluations of the cost function or its gradient. Typically, they rely on the continuity of the parameter space and often also on the continuity of the function itself. In high dimensional spaces, cost functions usually have many local optima in which optimization algorithms can get stuck as well as plateaus which slow down the convergence. There exist algorithms that use an adaptive step size to mitigate these effects. However, usually it is still necessary to start the optimizer with different (random) initial parameters and take at the end the overall best solution that was returned. In that sense, these algorithms also do not guarantee that the globally optimal solution has been found.

For cost functions of discrete or binary variables, these optimization algorithms are not applicable because the functions are only defined at discrete values. In general such optimization problems are NP-hard, and only a brute-force search over all possible inputs can deliver the optimal solution. For large configuration spaces, however, this is not feasible. For many discrete optimization problems, heuristic algorithms have been developed which work well for many cases. A heuristic algorithm may find the globally optimal solution but it may also return just a locally optimal solution as the outcome usually depends on the initialization. The advantage of heuristic algorithms is that the run time is much shorter than for a brute-force search. An example for such a heuristic algorithm is the simulated annealing algorithm [18].

A common discrete optimization problem in physics is to find the ground state of the Ising spin Hamiltonian

$$H_{\text{Ising}} = \sum_{i=1}^{N} h_i \sigma_i^z + \sum_{i<j} J_{ij} \sigma_i^z \sigma_j^z, \tag{28}$$

where N denotes the number of spins, h_i denotes the local field for spin i, J_{ij} denotes the

coupling strength between spins i and j, and σ_i^z denotes the Pauli z matrix

$$\sigma^z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{29}$$

for spin i with eigenstates $|\uparrow\rangle$ and $|\downarrow\rangle$ such that $\sigma^z|\uparrow\rangle = |\uparrow\rangle$ and $\sigma^z|\downarrow\rangle = -|\downarrow\rangle$. This kind of problems can be solved on the D-Wave quantum annealer.
Another form of discrete optimization that can be solved on the D-Wave quantum annealer is Quadratic Unconstrained Binary Optimization (QUBO). The QUBO cost function to minimize is given in the form

$$C(\mathbf{x}) = \sum_{i=1}^{N} a_i x_i + \sum_{i<j} b_{ij} x_i x_j, \tag{30}$$

where N is the number of binary variables $x_i \in \{0, 1\}$, $\mathbf{x} = x_1 x_2 \ldots x_N$ denotes the bitstring obtained by concatenating the problem variables, $a_i x_i$ are the linear terms and $b_{ij} x_i x_j$ are the quadratic terms in the binary variables, and $a_i, b_{ij} \in \mathbb{R}$ are the parameters that define the problem instance to be solved. In the definition of a QUBO, the term "Unconstrained" means that there are no constraints in the optimization of $C(\mathbf{x})$ such as "subject to $f(\mathbf{x}) = 0$". Solving a QUBO is equivalent to solving for the ground state of an Ising Hamiltonian Eq. (28). Many optimization problems that can be formulated in terms of an Ising or QUBO model are discussed in Ref. [19].

3.2 Working principle of a quantum annealer

Initially, quantum annealing was invented as an heuristic algorithm for a classical computer [20–22]. It was inspired by the simulated annealing algorithm [18] where thermal fluctuations are replaced by quantum fluctuations. For simulated annealing, thin but high energy barriers are difficult to overcome as thermal hopping processes become unlikely for high barriers. Therefore, the idea was that quantum fluctuations in quantum annealing allow for tunneling through these thin but high energy barriers. The quantum annealing Hamiltonian can be expressed as

$$H_{\text{QA}}(t/t_{\text{max}}) = \Gamma(t/t_{\text{max}}) H_{\text{QF}} + H_{\text{problem}} , \tag{31}$$

where H_{problem} encodes the optimization problem to be solved, and H_{QF} denotes the Hamiltonian introducing the quantum fluctuations. The function $\Gamma(t/t_{\text{max}})$ controls the strength of these fluctuations. It has to satisfy $\Gamma(0) \gg$ energy scale of H_{problem} and $\Gamma(1) \approx 0$.
Later, adiabatic quantum computation was proposed [23, 24]. The idea to perform the computation is based on the adiabatic theorem: The quantum system is to be initialized in the known ground state of an initial Hamiltonian H_{init}. Then, the time-dependent Hamiltonian of the system

$$H(t/t_{\text{max}}) = A(t/t_{\text{max}}) H_{\text{init}} + B(t/t_{\text{max}}) H_{\text{final}} \tag{32}$$

is changed over time into the Hamiltonian H_{final} which encodes an optimization problem (e.g., the Ising Hamiltonian Eq. (28)). The annealing functions $A(t/t_{\text{max}})$ and $B(t/t_{\text{max}})$ satisfy

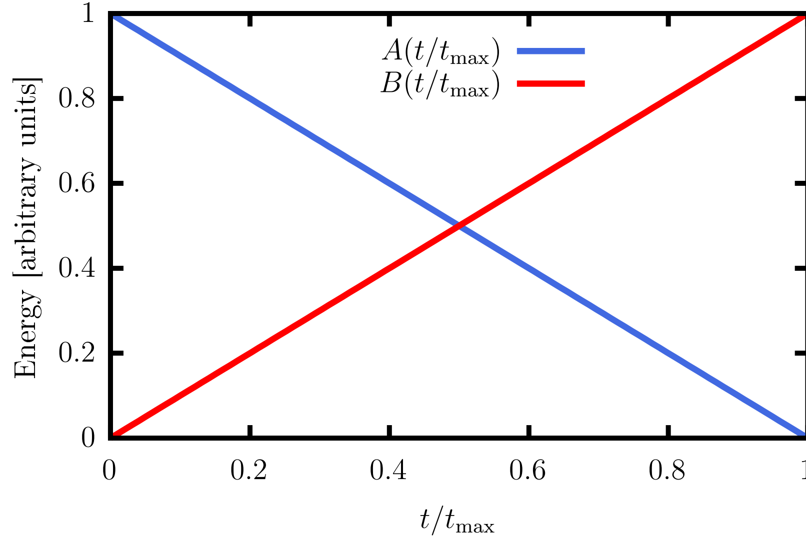


Fig. 10: *The annealing functions $A(t/t_{max})$ and $B(t/t_{max})$ for a linear annealing schedule.*

$A(0) \gg B(0)$ and $B(1) \gg A(1)$. An example for a linear annealing schedule is shown in Fig. 10. With $H_{\text{final}} = H_{\text{problem}}$, $H_{\text{init}} = H_{\text{QF}}$, $A(t/t_{\text{max}}) = \Gamma(t/t_{\text{max}})$ and $B(t/t_{\text{max}}) = 1$, Hamiltonian Eq. (32) implements the quantum annealing Hamiltonian Eq. (31). The adiabatic theorem [25] states that if the Hamiltonian $H(t/t_{\text{max}})$ changes sufficiently slowly with time, the system stays in the instantaneous ground state so that a measurement at the end of the process yields the state that encodes the solution to the optimization problem. A detailed review on the ideal (i.e., closed-system) model of adiabatic quantum computation is given in Ref. [26].
For a single spin-1/2 particle in a time-dependent external magnetic field, the closed-system model can be solved analytically exactly and is described by the Landau-Zener theory [27]: The Hamiltonian describing the spin in the external field $h_z = vt$ which changes with time t from $-\infty$ to ∞ with sweep velocity v and a time-independent transverse field h_x is given by

$$H_{\text{LZ}}(t) = -h_x\sigma^x - h_z(t)\sigma^z = -h_x\sigma^x - vt\sigma^z \tag{33}$$

where σ^x is the Pauli x matrix

$$\sigma^x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \tag{34}$$

with eigenstates $|+\rangle = (|\uparrow\rangle + |\downarrow\rangle)/\sqrt{2}$ and $|-\rangle = (|\uparrow\rangle - |\downarrow\rangle)/\sqrt{2}$. The spin is prepared in the ground state of $H_{\text{LZ}}(t \to -\infty)$ which is the state $|\psi_{\text{init}}\rangle = |\downarrow\rangle$. The spin then evolves with the Hamiltonian $H_{\text{LZ}}(t)$ and the Landau-Zener theory gives the probability to find the spin in the ground ($|\uparrow\rangle$) or excited state ($|\downarrow\rangle$) of $H_{\text{LZ}}(t)$ for $t \to \infty$

$$P_\uparrow = 1 - e^{-\pi h_x^2/v}, \quad P_\downarrow = e^{-\pi h_x^2/v}. \tag{35}$$

These probabilities show that for fast sweeping (large v), $P_\uparrow \to 0$ and $P_\downarrow \to 1$. They also show that for $h_x \to 0$, $P_\uparrow \to 0$ and $P_\downarrow \to 1$. Since the minimal energy gap is proportional to $|h_x|$, the probabilities scale not only with the sweep velocity but also with the minimal energy gap squared.

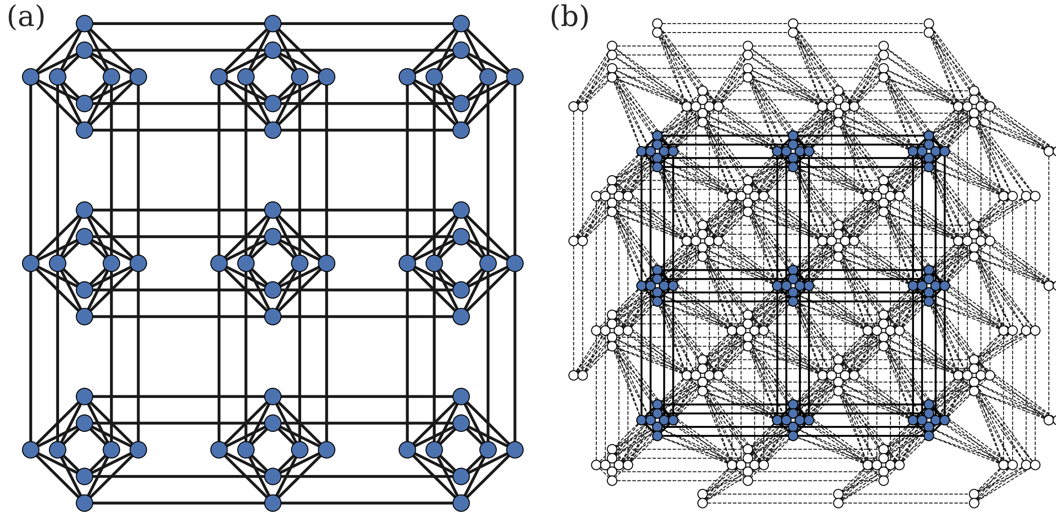


Fig. 11: *Small-size examples of the connectivity graphs of the (a) Chimera topology of the D-Wave 2000Q processors and (b) Pegasus topology of the Advantage processors. Blue nodes in the Pegasus graph show the embedding of the Chimera graph onto the Pegasus graph.*

In practice however, the annealing system is always connected to an environment at a finite temperature. This means that some sources of noise can never be removed completely. So not only a too rapid change of the Hamiltonian but also a too slow annealing procedure may excite the system.

3.3 Architecture of D-Wave quantum annealers

There are currently two generations of D-Wave quantum annealers available: The previous generation (D-Wave 2000Q) with about 2000 qubits and up to 6 couplers per qubit, and the current generation (Advantage) with more than 5000 qubits and up to 15 couplers per qubit. The qubits of the D-Wave 2000Q quantum processors are arranged in the so-called Chimera topology and the qubits of the Advantage systems are arranged in the so-called Pegasus topology (see Fig. 11). The Chimera graph is a subgraph of the Pegasus graph.
In the case of D-Wave quantum annealers, the final Hamiltonian is given by the Ising Hamiltonian

$$H_{\text{final}} = H_{\text{Ising}} = \sum_{i=1}^{N} h_i \sigma_i^z + \sum_{\substack{i<j \\ i,j \text{ neighbors}}} J_{ij} \sigma_i^z \sigma_j^z, \tag{36}$$

and the initial Hamiltonian is given by

$$H_{\text{init}} = -\sum_{i=1}^{N} \sigma_i^x . \tag{37}$$

The ground state of the initial Hamiltonian is given by the equal superposition of all computa-

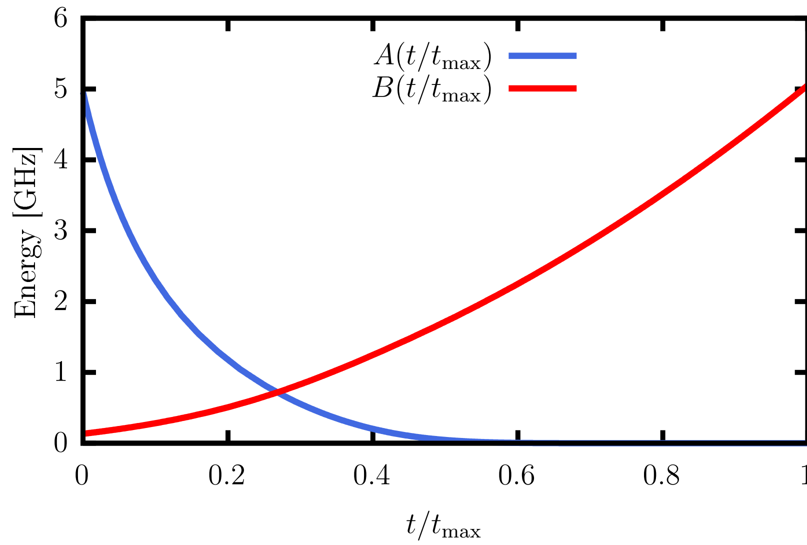


Fig. 12: *The annealing functions $A(t/t_{max})$ and $B(t/t_{max})$ of the annealing schedule of D-Wave's 5000+ qubit processor* `Advantage_system1.1`.

tional basis states,

$$|\psi_{\text{init}}\rangle = |+\rangle^{\otimes N} = \frac{1}{\sqrt{2^N}} \sum_{z_i \in \{\uparrow,\downarrow\}} |z_1 z_2 \dots z_N\rangle. \tag{38}$$

The qubits (two-level systems) of the Hamiltonians in Eqs. (36) and (37) are built of superconducting circuits. The particular design of these circuits is called flux qubits. The flux qubits as well as the couplers which allow for a tunable coupling between the qubits are controlled via external fluxes. The parameters h_i and J_{ij} of the final Hamiltonian given in Eq. (36) are controlled by time-independent external magnetic fluxes. The annealing process is controlled through time-dependent external magnetic fluxes which change the effective Hamiltonian from the initial Hamiltonian to the final Hamiltonian. As an example, in Fig. 12 we show the annealing schedule of the processor `Advantage_system1.1`. The annealing schedule is obviously different from the linear annealing schedule shown in Fig. 10. The reason for this is that the hardware design of the flux qubits does not allow for an independent control of the functions $A(t/t_{\max})$ and $B(t/t_{\max})$ [28]. Like all superconducting qubits, these systems are actually multi-level systems of which only a two-dimensional subspace (spanned by the two lowest energy eigenstates) functions as the qubit. More detailed information can be found in, for instance, Refs. [28–30].

3.4 Limitations

Current quantum annealers are subject to various limitations which may have an impact on the performance. For instance, due to the limited connectivity of the qubits (up to 6 connections per qubit on the Chimera architecture; up to 15 connections per qubit on the Pegasus architecture), the variables of a given problem may not be directly mappable to the qubits. In that case, an

embedding of the problem graph onto the hardware graph is necessary. Finding an (optimal) embedding is in general a hard problem itself. However, there exist heuristic methods to find embeddings and the `Ocean` package provides such an algorithm (cf. section 3.5.4). The need to find an embedding limits the performance due to the time overhead required to generate an embedding and it often also increases the required number of qubits (see section 3.5.4 for more details). Thus, although problem instances that feature the native hardware graphs (Chimera or Pegasus graphs) can be put on the D-Wave 2000Q and Advantage systems with sizes of 2000+ and 5000+ qubits, respectively, only 64- and 124-qubit fully-connected problems can be placed on these systems, respectively.

Other limiting factors are the restricted range and the limited precision of the parameters h_i and J_{ij}. In order for the parameter values to fit into the available range they have to be rescaled (see section 3.5.1). Thus, for problem instances which cover a large range of values but at the same time include parameters with small differences, these differences may not be resolvable anymore within the available precision, i.e., configured parameters on the hardware may differ from the specified ones.

A general limiting factor in practical quantum annealing is that in practice the quantum system can never be completely isolated from the environment and noise sources. As a consequence, the system can be thermally excited. In particular, for theoretical closed-system quantum annealing, the annealing time cannot be too long. However, for open systems, a long annealing time means long time for the system to interact with the environment and a higher probability to leave the ground state.

Apart from these practical limitations, there are also theoretical issues. The adiabatic theorem states that the quantum system stays in its instantaneous ground state if the Hamiltonian changes sufficiently slowly in time. In particular, the annealing time should satisfy ($s = t/t_{\text{max}}$) [31]

$$t_{\text{max}} \gg \max_{s\in[0,1]} \frac{\left|\langle E_n(s)|\frac{\partial H}{\partial s}|E_0(s)\rangle\right|}{\big(E_0(s)-E_n(s)\big)^2} \quad \text{for } n \neq 0. \tag{39}$$

Thus, for an adiabatic evolution and for exponentially small energy gaps, the annealing time is expected to also increase exponentially.

3.5 Programming a D-Wave quantum annealer

In this section, we focus on programming the D-Wave quantum annealer using D-Wave's python package `Ocean-SDK` [32, 33], as well as some practical aspects which are useful when programming a D-Wave quantum annealer. It is worth mentioning that, even though the previous sections contain useful technical details for background information, the only knowledge required to program a quantum annealer is about the kind of optimization problem (Ising or QUBO) that it solves. This makes quantum annealers particularly attractive for non-physicists.

3.5.1 Problem specification

Optimization problems that can be put onto D-Wave quantum annealers have to be formulated as QUBO

$$\min_{x_i=0,1}\left(\sum_{i=0}^{N-1} a_i x_i + \sum_{i<j}^{N-1} b_{ij} x_i x_j\right) = \min_{x_i=0,1}\left(\sum_{i\le j}^{N-1} x_i Q_{ij} x_j\right), \tag{40}$$

or as an Ising Hamiltonian

$$\min_{s_i=\pm 1}\left(\sum_{i=0}^{N-1} h_i s_i + \sum_{i<j}^{N-1} J_{ij} s_i s_j\right), \tag{41}$$

where $s_i \in \{-1, +1\}$ are the eigenvalues of the Pauli-z matrix σ_i^z. Reformulating a QUBO as Ising Hamiltonian can be done by substituting

$$x_i = \frac{1+s_i}{2}. \tag{42}$$

This convention is commonly used in quantum annealing and maps $x_i = 0$ and $x_i = 1$ to $s_i = -1$ and $s_i = 1$, respectively. Note that this convention is different from the one commonly used for gate-based quantum computing (see above). When converting problems between QUBO and Ising formulation, constants which arise from the substitution Eq. (42) can be neglected as they only lead to an energy shift but do not change the solution of the optimization.

To specify the problem instance using the `Ocean-SDK`, the coefficients can be stored in a dictionary `Q` = $\{$`(`i`,`j`)`$: Q_{ij}\}$ (QUBO) or two dictionaries `h` = $\{i : h_i\}$ and `J` = $\{$`(`i`,`j`)`$: J_{ij}\}$ (Ising).

In practice, the coupling strength J_{ij} can only be set to values in a certain interval $[J_{\text{min}}, J_{\text{max}}]$, where usually $J_{\text{min}} = -J_{\text{max}}$, with a limited precision. The same applies to the single-qubit bias h_i with interval $[h_{\text{min}}, h_{\text{max}}]$. Thus, all h_i and J_{ij} need to be rescaled by a factor [34]

$$r = \max\left\{\max\left[\frac{\max\{h_i\}}{h_{\text{max}}}, 0\right], \max\left[\frac{\min\{h_i\}}{h_{\text{min}}}, 0\right], \max\left[\frac{\max\{J_{ij}\}}{J_{\text{max}}}, 0\right], \max\left[\frac{\min\{J_{ij}\}}{J_{\text{min}}}, 0\right]\right\}, \tag{43}$$

to fit into the ranges $h_{\text{min}} \le h_i \le h_{\text{max}}$ and $J_{\text{min}} \le J_{ij} \le J_{\text{max}}$. If $r < 1$, rescaling is optional, but it may be useful to exhaust a larger parameter range and potentially improve the performance. When `auto_scale` is set to `true` (default), the rescaling is performed automatically when submitting a problem through `Ocean`. When auto-scaling is disabled and the problem parameters do not fit into the parameter range, the submission of the problem fails. Note that rescaling does not change the solution of the problem. In general, it is advised to have the auto-scaling feature enabled (default). However, there may be certain cases where it has to be disabled. If in such a study, unexpected results are obtained, one should check to disable the auto-scaling feature.

```
1 from dwave.system import DWaveSampler
2
3 sampler = DWaveSampler(solver='DW_2000Q_6', token='insert_your_token_here')
4 h = {0:1, 4:-0.5}
5 J = {(0,4):1, (0,5):-1}
6
7 response = sampler.sample_ising(h, J)
8 print(response)
```

Listing 5: *A minimal working example to run a program on a D-Wave quantum annealer.*

```
1 from dwave.system import DWaveSampler, EmbeddingComposite
2
3 sampler = EmbeddingComposite(DWaveSampler(solver='DW_2000Q_6', token='
     insert_your_token_here'))
4 Q = {(0,0):1, (0,1):1, (0,2):-1, (1,2):-0.8}
5
6 response = sampler.sample_qubo(Q, num_reads=100, chain_strength=2,
     annealing_time=5)
7 print(response)
```

Listing 6: *Example showing how to use an `EmbeddingComposite`.*

3.5.2 Submitting a problem to the D-Wave quantum annealer through the Ocean package

Listing 5 shows a minimal working example of a submission to the D-Wave quantum annealer. The class `DWaveSampler` takes a solver (for example the current hardware solvers, at the time of writing, '`DW_2000Q_6`' or '`Advantage_system1.1`') to which to submit the problem. If one did not create a `config`-file during or after the installation of the `Ocean-SDK`, the personal token also has to be provided to `DWaveSampler`. Depending on the formulation of the problem (Ising or QUBO), one has to create the `h` and `J` or `Q` dictionaries, respectively. The example code in listing 5 shows the case for the Ising formulation. For larger problems, the dictionaries should be generated algorithmically. The class `DWaveSampler` has the member functions `sample_ising` and `sample_qubo` which submit the specified problem to the QPU. The function `sample_ising` takes the `h` and `J` dictionaries and the function `sample_qubo` takes the `Q` dictionary.

Optional parameters of the functions `sample_ising` and `sample_qubo` are for instance the number of samples obtained per submission (`num_reads`), the `annealing_time` (in µs), or the `chain_strength` (see section 3.5.4). An example is shown in listing 6.

Listing 6 also illustrates how to use an `EmbeddingComposite` (see section 3.5.4). The return value contains the solutions returned by the quantum annealer as well as some additional information. An example output is shown in listing 7: The first column labels the different solutions returned (rows). The following N_{qubit} (in this case $N_{\text{qubit}} = 3$) columns give the values

```
1    0  1  2 energy num_oc. chain_.
2 0  0  1  1   -0.8      79     0.0
3 1  0  0  1    0.0       5     0.0
4 2  1  0  1    0.0       5     0.0
5 3  0  0  0    0.0       7     0.0
6 4  0  1  0    0.0       1     0.0
7 5  1  1  1    0.2       2     0.0
8 6  1  0  0    1.0       1     0.0
9 ['BINARY', 7 rows, 100 samples, 3 variables]
```

Listing 7: *Example output of the program given in listing 6.*

```
1 import dwave.inspector
2
3 ##################
4 #  previous code #
5 ##################
6
7 dwave.inspector.show(response)
```

Listing 8: *Example showing how to use `dwave.inspector` to visualize the result. A screenshot using the Leap IDE is shown in Fig. 13.*

of the qubits in the returned solution. The next three columns contain the energy, the number of occurrences of this solution in all samples, and the chain break fraction (see section 3.5.4), respectively. In our example output, we obtained 79 times the solution no. 0 with energy -0.8 (the energy of the ground state) and no chain breaks. In addition, the `response` also contains the information that the result is given in `BINARY`, i.e. QUBO, format (in the Ising representation it would be `SPIN`), that 7 distinct answers were returned by the quantum annealer (the number of rows), that the number of samples is 100 (equals `num_reads`), and that the number of qubits (variables) is three. Further information can be accessed through `response.info`.

Another tool of the `Ocean` package which can be handy when studying the returned results is `dwave.inspector`. With this tool, the response object can be visualized. The usage is illustrated in listing 8 and Fig. 13.

3.5.3 Implementing constraints

As mentioned previously, D-Wave quantum annealers are designed to solve QUBO or Ising problems by minimizing the corresponding energy function without constraints. However, in practice optimization problems often require constrained optimization, i.e., the objective function $C(\mathbf{x})$ needs to be minimized under a certain constraint $f(\mathbf{x}) = c$, where f is a function that takes a string of bits $\mathbf{x}$ and returns a scalar, and c is a scalar, i.e., the optimization task is to "minimize $C(\mathbf{x})$ subject to $f(\mathbf{x}) = c$".
We can consider such a constraint by formulating it as a QUBO (or Ising Hamiltonian) and adding it to the objective function: We can write the constraint $f(\mathbf{x}) = c$ as $f(\mathbf{x}) - c = 0$. The

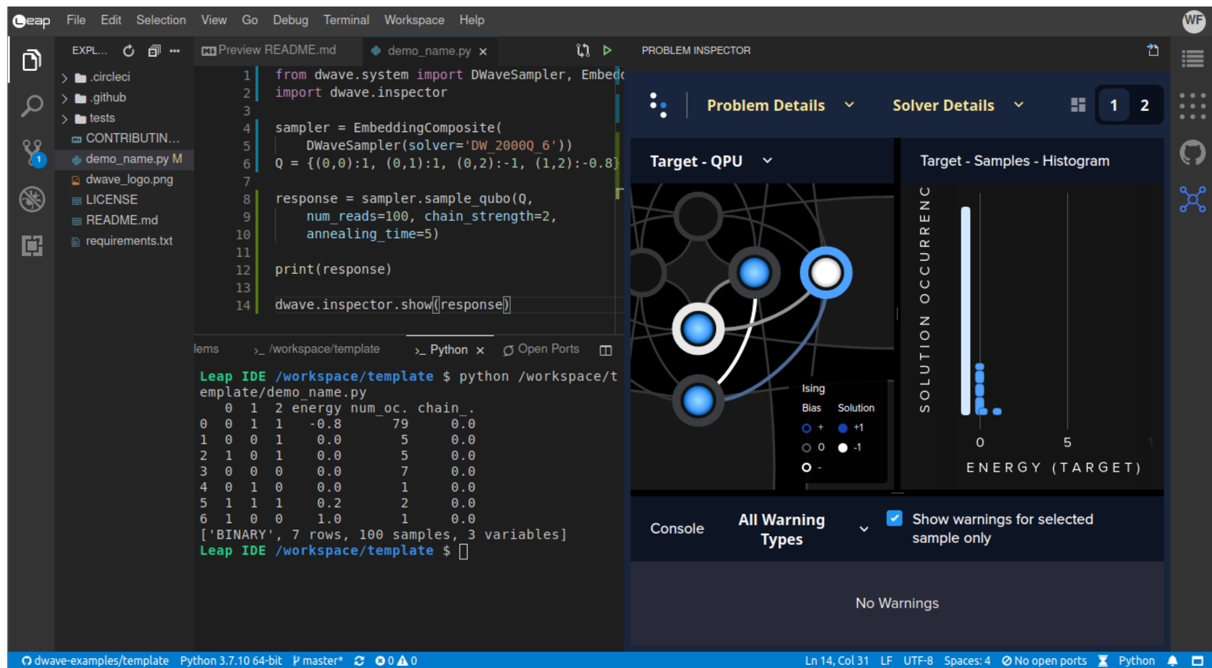


Fig. 13: *Running the example program from listing 6 and visualizing the result on the D-Wave using the inspector (see listings 7 and 8). The Leap IDE with an example template to run such an experiment can be opened at* `https://ide.dwavesys.io/#https://github.com/dwave-examples/template`.

square $(f(\mathbf{x}) - c)^2$ is always greater than zero, except if the constraint $f(\mathbf{x}) = c$ is satisfied in which case the square is equal to zero and thus minimal. By adding the square term $(f(\mathbf{x}) - c)^2$ to the objective function, we effectively add a penalty to the objective function if the constraint is not satisfied. To keep things simple, we assume that $f(\mathbf{x})$ is a linear function in the bits x_i (further information can be found in [35]). We add the penalty term to the objective function $C(\mathbf{x})$ so that the new/modified objective function reads $C(\mathbf{x}) + \lambda(f(\mathbf{x}) - c)^2$, where λ is a scalar called Lagrange multiplier and has to be chosen reasonably. The optimization task is now “minimize $C(\mathbf{x}) + \lambda(f(\mathbf{x}) - c)^2$”.

A “reasonable” choice for λ means that λ should neither be too small nor too large. If λ is chosen too small, the constraint will likely not be satisfied in the optimal solution for $C(\mathbf{x}) + \lambda(f(\mathbf{x}) - c)^2$ as it may be more favorable to accept a penalty multiplied by a small λ than to return a state with a larger cost function value $C(\mathbf{x})$. On the other hand, a too large λ will force the constraint to be satisfied but due to the rescaling of the parameters (see section 3.5.1), the problem parameters might become so small that they cannot be represented accurately enough with the limited precision, and also the energy differences become so small that excitations to higher energy states, which do not encode the optimal solution to the original problem anymore, become very likely.

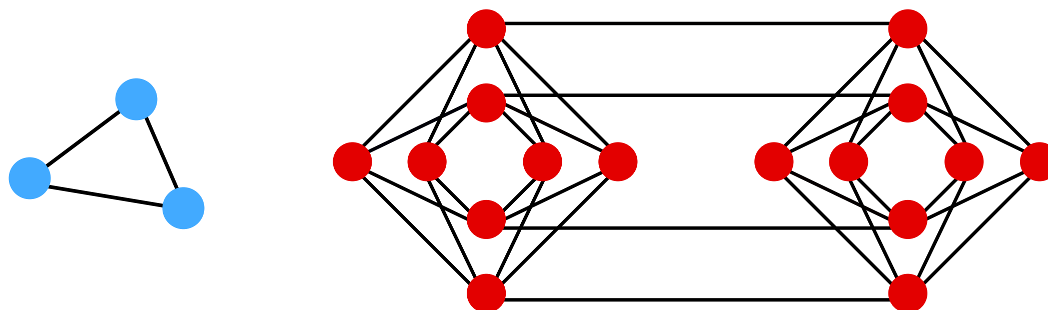


Fig. 14: *A graph with triangular connectivity (left) cannot be directly mapped onto the Chimera hardware graph (right).*

3.5.4 Embedding of problem graphs onto the hardware graph

As soon as problem sizes become larger so that we cannot immediately find a mapping of the problem graph onto the hardware graph, it is convenient to call a dedicated routine to find this mapping for us. Generating the optimal embedding of a graph onto another one is in general NP-hard, but a heuristic algorithm is provided by the `Ocean-SDK`. This algorithm uses probabilistic methods, which means that each time we call the function, it may return a different embedding and these may be of different quality. Thus, one typically tries several different embeddings and chooses the best one (see also [15, 17]). It may also be possible that the problem graph requires connectivities which are not present on the hardware graph. For instance, a triangular connectivity as shown in Fig. 14 cannot be mapped onto the Chimera graph. In such a case, we have to use more than a single physical qubit (the ones on the hardware) to represent a logical qubit (the ones in the problem specification), see Fig. 15. If we have more than a single physical qubit representing a logical qubit, i.e., a chain of two or more physical qubits represents a single logical qubit, the physical qubits should behave as a single entity, i.e., at the end of the annealing process, they should all have the same value.

To achieve this, the couplings J_{ij} between these physical qubits are set to a negative value with large magnitude. The magnitude (also called chain strength) determines how strongly these qubits couple and how easily the chain may "break". In this context, a "chain break" means that different qubits of a chain representing the same logical qubit end up in different states. The Ocean package includes post-processing procedures which, in this case, determine the value to return for the logical qubit by majority vote of the physical qubits. Thus, the values returned will always be valid for the original problem, although they could be far from optimal. Too many chain breaks should be avoided as the returned solutions become "randomized" and may be rather poor.

The optimal value of the chain strength depends on the particular problem and possibly also on the particular embedding. If the chain strength is chosen too weak, the optimal and close-to-optimal solutions may likely contain chain breaks as it is energetically favorable to break these couplings instead of the ones that encode the actual problem instance. In theory, the chain strength should be chosen as large as possible to satisfy the constraints. In practice however,

Fig. 15: *A graph with triangular connectivity (left) can be mapped onto the Chimera hardware graph (right) by using two physical qubits for one of the logical qubits.*

we have to keep in mind that the coupling strength J_{ij} can only be set to values in the interval $[J_{\text{min}}, J_{\text{max}}]$ with a limited precision (cf. section 3.5.1). Thus, all h_i and J_{ij} are rescaled by the factor r (see Eq. (43)). If the chain strength is chosen too strong, all parameters defining the problem instance will be rescaled to small values which might be no longer resolvable with the given precision. Moreover, the energy differences of the original problem are also rescaled and become very small which could more likely lead to excitations to higher energy states. This, in turn, would lead to worse results for the original problem than if a weaker chain strength had been chosen. The chain strength is a parameter called `chain_strength` that can be given to the `sample_qubo` or `sample_ising` functions.

The `Ocean` package provides several `EmbeddingComposite` classes [33] to handle the generation of embeddings:

- `EmbeddingComposite` tries to find an embedding each time one of the `sample` functions (`sample_qubo` or `sample_ising`) is called. This can be useful when submitting different problems or when studying the dependence on different embeddings.

- `LazyFixedEmbeddingComposite` tries to find an embedding the first time one of the `sample` functions is called. Later, it reuses this embedding. This can be useful when submitting the same problem with different hyperparameters such as annealing time or chain strength.

- `FixedEmbeddingComposite` takes an embedding as argument which is then used each time one of the `sample` functions is called. This can be useful when we already have an embedding for a particular problem (either by generating it ourselves or by loading a previously stored embedding) and we want to reuse it again.

- `TilingComposite` can be passed to any of the `EmbeddingComposites` to place several copies of small embedded graphs.

	Leek	Celery	Peas	Corn
Leek	○	♡	×	○
Celery	♡	○	○	×
Peas	×	○	○	♡
Corn	○	×	♡	○

Table 1: *Companion planting example of good (♡), neutral (○), and bad (×) neighbors.*

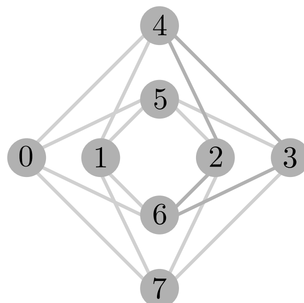


Fig. 16: *Labeling of the qubits in a Chimera graph unit cell of the D-Wave 2000Q quantum processors.*

3.6 Example: Garden optimization

As an example, we consider a simplified four-qubit problem from the garden optimization problem presented in [17]. The task is to place four plants in two pots such that good neighbors are in the same pot and bad neighbors are not. The relationships between the four plants that we consider in this example are shown in table 1.

The two pots have the labels -1 and $+1$ and the value s_i of qubit i denotes into which pot we place the plant of type i ($i \in \{$leek, celery, peas, corn$\}$). Since we consider minimization problems, we want to minimize the energy when good neighbors are placed in the same pot. Assume we place plants i and j in the same pot (i.e., $s_i = s_j$). According to Eq. (41), the energy is lowered if we choose J_{ij} negative (which we want for good neighbors i and j), and the energy is increased if we choose J_{ij} positive (which we want for bad neighbors i and j). Thus, we set $J_{\text{leek, celery}} = J_{\text{peas, corn}} = -1$ and $J_{\text{leek, peas}} = J_{\text{celery, corn}} = 1$. The energy function is then given by

$$E(s_{\text{leek}}, s_{\text{celery}}, s_{\text{peas}}, s_{\text{corn}}) = s_{\text{leek}} s_{\text{peas}} + s_{\text{celery}} s_{\text{corn}} - s_{\text{leek}} s_{\text{celery}} - s_{\text{peas}} s_{\text{corn}}. \tag{44}$$

The next step is to map the qubits onto the hardware graph. The labeling of the qubits in the first unit cell of the Chimera topology used in the D-Wave 2000Q processors is shown in Fig. 16. Since the problem is small and can be directly mapped onto the hardware graph, we define the mapping without an `EmbeddingComposite`. We use the mapping

$$\text{leek} \to 0, \quad \text{corn} \to 3, \tag{45}$$

$$\text{celery} \to 4, \quad \text{peas} \to 7, \tag{46}$$

which gives

$$J_{04} = J_{37} = -1 \quad \text{and} \quad J_{07} = J_{34} = 1. \tag{47}$$

The program to submit and solve the problem on the D-Wave 2000Q chip `DW_2000Q_6` is shown in listing 9.

Exercise 7: Consider the case that leek has already been placed in the pot with label “-1”. Replanting it would require additional work. How could this additional cost be considered in the energy function? Modify the program in listing 9 accordingly. How does the result change?

```
1  from dwave.system import DWaveSampler
2  import dwave.inspector
3
4  sampler = DWaveSampler(solver='DW_2000Q_6', token='insert_your_token_here')
5
6  h = {}
7
8  # We choose:
9  # 0 = leek
10 # 4 = celery
11 # 7 = peas
12 # 3 = corn
13 J = { (0,4): -1, (0,7): +1, (3,4): +1, (3,7): -1 }
14
15 response = sampler.sample_ising(h, J, num_reads=100)
16
17 dwave.inspector.show(response)
```

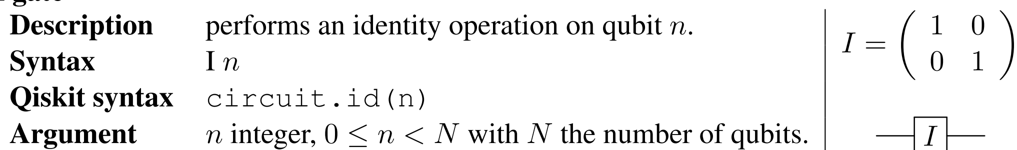
Listing 9: *Example code solving the four-qubit garden problem.*

Appendix

A JUQCS standard gate set

This appendix contains the standard gate set implemented by the Jülich Universal Quantum Computer Simulator (JUQCS) [10]. JUQCS is a large-scale simulator for gate-based quantum computers that was also used for Google's quantum supremacy experiment [11]. A GPU-accelerated version was recently used to benchmark JUWELS Booster [8] with over 2048 GPUs across 512 compute nodes. A `qiskit` [12] interface to JUQCS including the conversion from the `qiskit` gate set to the JUQCS gate set is available through the Jülich UNified Infrastructure for Quantum computing (JUNIQ) service at `https://jugit.fz-juelich.de/qip/juniq-platform`. This interface was used for the example programs in this lecture.

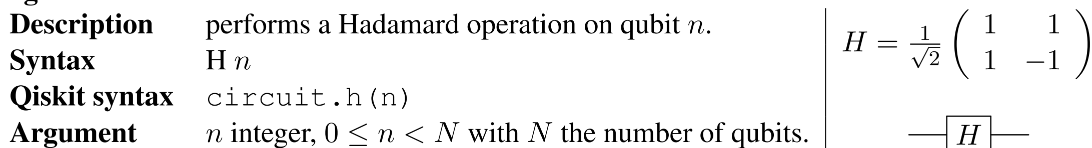
I gate



Description	performs an identity operation on qubit n.
Syntax	I n
Qiskit syntax	`circuit.id(n)`
Argument	n integer, $0 \le n < N$ with N the number of qubits.

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

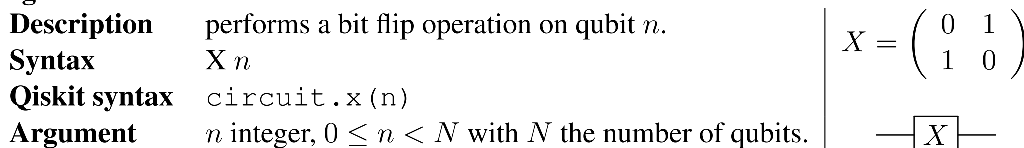
H gate



Description	performs a Hadamard operation on qubit n.
Syntax	H n
Qiskit syntax	`circuit.h(n)`
Argument	n integer, $0 \le n < N$ with N the number of qubits.

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

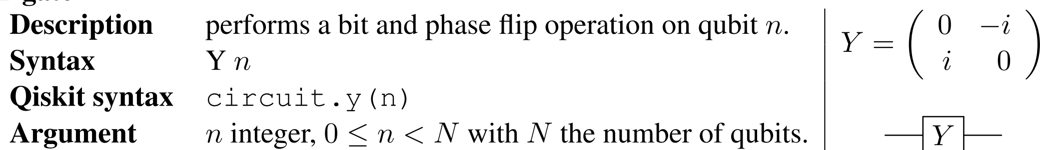
X gate



Description	performs a bit flip operation on qubit n.
Syntax	X n
Qiskit syntax	`circuit.x(n)`
Argument	n integer, $0 \le n < N$ with N the number of qubits.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

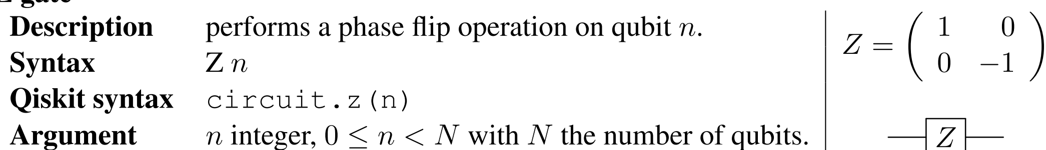
Y gate



Description	performs a bit and phase flip operation on qubit n.
Syntax	Y n
Qiskit syntax	`circuit.y(n)`
Argument	n integer, $0 \le n < N$ with N the number of qubits.

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

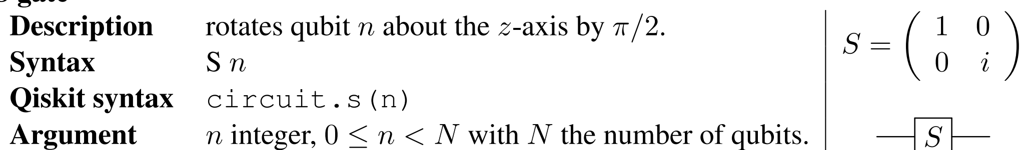
Z gate



Description	performs a phase flip operation on qubit n.
Syntax	Z n
Qiskit syntax	`circuit.z(n)`
Argument	n integer, $0 \le n < N$ with N the number of qubits.

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

S gate



Description	rotates qubit n about the z-axis by $\pi/2$.
Syntax	S n
Qiskit syntax	`circuit.s(n)`
Argument	n integer, $0 \le n < N$ with N the number of qubits.

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

S† gate

Description	rotates qubit n about the z-axis by $-\pi/2$	$S^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$
Syntax	S+ n	
Qiskit syntax	`circuit.sdg(n)`	
Argument	n integer, $0 \le n < N$ with N the number of qubits.	$S^\dagger$

T gate

Description	rotates qubit n about the z-axis by $\pi/4$	$T = \begin{pmatrix} 1 & 0 \\ 0 & (1+i)/\sqrt{2} \end{pmatrix}$
Syntax	T n	
Qiskit syntax	`circuit.t(n)`	
Argument	n integer, $0 \le n < N$ with N the number of qubits.	T

T† gate

Description	rotates qubit n about the z-axis by $-\pi/4$	$T^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & (1-i)/\sqrt{2} \end{pmatrix}$
Syntax	T+ n	
Qiskit syntax	`circuit.tdg(n)`	
Argument	n integer, $0 \le n < N$ with N the number of qubits.	$T^\dagger$

U1 gate

Description	performs a U1(λ) operation [6] on qubit n.	$U1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}$
Syntax	U1 n λ	
Qiskit syntax	`circuit.p(lam, n)`	
Arguments	n integer, $0 \le n < N$ with N the number of qubits, λ angle in radians (floating point or integer).	$U1(\lambda)$

U2 gate

Description	performs a U2(ϕ, λ) operation [6] on qubit n.	$U2(\phi,\lambda) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{pmatrix}$
Syntax	U2 n ϕ λ	
Qiskit syntax	`circuit.u(pi/2, phi, lam, n)`	
Arguments	n integer, $0 \le n < N$ with N the number of qubits, ϕ, λ angles in radians (floating point or integer).	$U2(\phi,\lambda)$

U3 gate

Description	performs a U3(θ, ϕ, λ) operation [6] on qubit n.	$U3(\theta,\phi,\lambda) =$
Syntax	U3 n θ ϕ λ	$\begin{pmatrix} \cos(\frac{\theta}{2}) & -e^{i\lambda}\sin(\frac{\theta}{2}) \\ e^{i\phi}\sin(\frac{\theta}{2}) & e^{i(\phi+\lambda)}\cos(\frac{\theta}{2}) \end{pmatrix}$
Qiskit syntax	`circuit.u(theta, phi, lam, n)`	
Arguments	n integer, $0 \le n < N$ with N the number of qubits, θ, ϕ, λ angles in radians (floating point or integer).	$U3(\theta,\phi,\lambda)$

+X gate

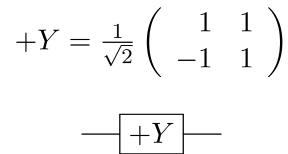
Description	rotates qubit n by $-\pi/2$ about the x-axis.	$+X = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & i \\ i & 1 \end{pmatrix}$
Syntax	+X n	
Qiskit syntax	`circuit.rx(-pi/2, n)`	
Argument	n integer, $0 \le n < N$ with N the number of qubits.	$+X$

-X gate

Description	rotates qubit n by $+\pi/2$ about the x-axis.	$-X = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix}$
Syntax	-X n	
Qiskit syntax	`circuit.rx(pi/2, n)`	
Argument	n integer, $0 \le n < N$ with N the number of qubits.	$-X$

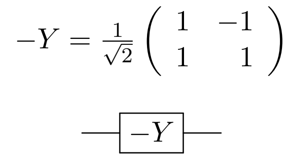
+Y gate

Description rotates qubit n by $-\pi/2$ about the y-axis.
Syntax +Y n
Qiskit syntax `circuit.ry(-pi/2, n)`
Argument n integer, $0 \leq n < N$ with N the number of qubits.



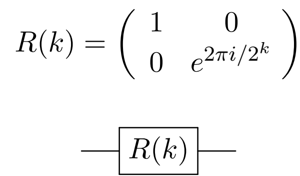
-Y gate

Description rotates qubit n by $+\pi/2$ about the y-axis.
Syntax -Y n
Qiskit syntax `circuit.ry(pi/2, n)`
Argument n integer, $0 \leq n < N$ with N the number of qubits.



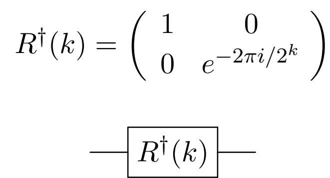
R(k) gate

Description changes the phase of qubit n by $2\pi/2^k$.
Syntax R n k
Qiskit syntax `circuit.p(2*pi/2**k, n)`
Arguments n integer, $0 \leq n < N$ with N the number of qubits, k is non-negative integer.



R†(k) gate

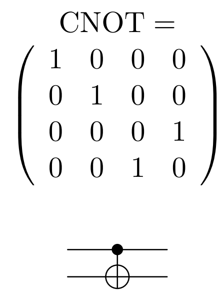
Description changes the phase of qubit n by $-2\pi/2^k$.
Syntax R n $-k$
Qiskit syntax `circuit.p(-2*pi/2**k, n)`
Arguments n integer, $0 \leq n < N$ with N the number of qubits, k is non-negative integer.



Two-qubit gates

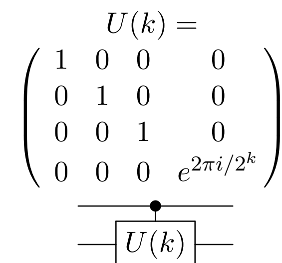
CNOT gate

Description flips the target qubit if the control qubit is 1.
Syntax CNOT *control target*
Qiskit syntax `circuit.cx(control, target)`
Arguments *control* $\neq$ *target* integers in the range $0, \ldots, N-1$ with N the number of qubits.
Note The matrix looks different from the `qiskit` documentation as `qiskit` uses the ordering $|q_{n-1} \cdots q_0\rangle$ while all standard text books as well as these lecture notes use $|q_0 \cdots q_{n-1}\rangle$. To fix this, the supplied example programs invoke the `circuit.reverse_bits()` function.



U(k) gate

Description shifts the phase of the target qubit by $2\pi/2^k$ if the control qubit is 1.
Syntax U *control target k*
Qiskit syntax `circuit.cp(2*pi/2**k, control, target)`
Arguments *control* $\neq$ *target* integers in the range $0, \ldots, N-1$ with N the number of qubits, k non-negative integer.



U†(k) gate

Description shifts the phase of the target qubit by $-2\pi/2^k$ if the control qubit is 1.

Syntax U *control target* $-k$

Qiskit syntax `circuit.cp(-2*pi/2**k, control,target)`

Arguments *control* $\neq$ *target* integers in the range $0, \ldots, N-1$ with N the number of qubits, k non-negative integer.

$$U^\dagger(k) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{-2\pi i/2^k} \end{pmatrix}$$

Toffoli gate

Description flips the target qubit if both control qubits are 1.

Syntax TOFFOLI $control_1$ $control_2$ $target$

Qiskit syntax `circuit.ccx(control1,control2,target)`

Arguments $control_1 \neq control_2 \neq target \neq control_1$ integers in the range $0, \ldots, N-1$ with N the number of qubits.

Note The matrix looks different from the `qiskit` documentation because `qiskit` uses the ordering $|q_{n-1} \cdots q_0\rangle$ while all standard text books as well as these lecture notes use $|q_0 \cdots q_{n-1}\rangle$. To fix this, the supplied example programs invoke the `circuit.reverse_bits()` function.

$$\text{TOFFOLI} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

References

[1] P.W. Shor, in *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 124–134

[2] E. Farhi, J. Goldstone, and S. Gutmann, arXiv:1411.4028

[3] D. Willsch: *Supercomputer simulations of transmon quantum computers* Ph.D. thesis, RWTH Aachen University, Aachen (2020)

[4] J. Koch, T.M. Yu, J. Gambetta, A.A. Houck, D.I. Schuster, J. Majer, A. Blais, M.H. Devoret, S.M. Girvin, and R.J. Schoelkopf, Phys. Rev. A **76**, 042319 (2007)

[5] D.C. McKay, C.J. Wood, S. Sheldon, J.M. Chow, and J.M. Gambetta, Phys. Rev. A **96**, 022330 (2017)

[6] A.W. Cross, L.S. Bishop, J.A. Smolin, and J.M. Gambetta, arXiv:1707.03429

[7] M. Nielsen and I. Chuang: *Quantum Computation and Quantum Information* (Cambridge University Press, 2010), 10th anniversary edition ed.

[8] D. Willsch, M. Willsch, F. Jin, K. Michielsen, and H. De Raedt, arXiv:2104.03293

[9] K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, Th. Lippert, H. Watanabe, and N. Ito, Comput. Phys. Commun. **176**, 121 (2007)

[10] H. De Raedt, F. Jin, D. Willsch, M. Willsch, N. Yoshioka, N. Ito, S. Yuan, and K. Michielsen, Comput. Phys. Commun. **237**, 47 (2019)

[11] F. Arute, K. Arya, R. Babbush, D. Bacon, J.C. Bardin, R. Barends, R. Biswas, S. Boixo, F.G.S.L. Brandao, D.A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M.P. Harrigan, M.J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T.S. Humble, S.V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P.V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J.R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M.Y. Niu, E. Ostby, A. Petukhov, J.C. Platt, C. Quintana, E.G. Rieffel, P. Roushan, N.C. Rubin, D. Sank, K.J. Satzinger, V. Smelyanskiy, K.J. Sung, M.D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z.J. Yao, P. Yeh, A. Zalcman, H. Neven, and J.M. Martinis, Nature **574**, 505 (2019)

[12] *Qiskit: An Open-source Framework for Quantum Computing* (2019) `http://doi.org/10.5281/zenodo.2562110`

[13] IBM Q team: *Quantum Experience* (2016) `http://quantum-computing.ibm.com/`

[14] T.G. Draper, arXiv:quant-ph/0008033

[15] M. Willsch, D. Willsch, F. Jin, H. De Raedt, and K. Michielsen, Quantum Inf. Process. **19**, 197 (2020)

[16] E. Jones, T. Oliphant, P. Peterson et al.: *SciPy: Open source scientific tools for Python* (2001) `http://www.scipy.org/`

[17] C.D.G. Calaza, D. Willsch, and K. Michielsen, arXiv:2101.10827

[18] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, Science **220**, 671 (1983)

[19] A. Lucas, Front. Phys. **2**, 5 (2014)

[20] B. Apolloni, C. Carvalho, and D. de Falco, Stoch. Process. Their Appl. **33**, 233 (1989)

[21] A. Finnila, M. Gomez, C. Sebenik, C. Stenson, and J. Doll, Chem. Phys. Lett. **219**, 343 (1994)

[22] T. Kadowaki and H. Nishimori, Phys. Rev. E **58**, 5355 (1998)

[23] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser, arXiv:quant-ph/0001106

[24] A.M. Childs, E. Farhi, and J. Preskill, Phys. Rev. A **65**, 012322 (2001)

[25] M. Born and V. Fock, Z. Phys. **51**, 165 (1928)

[26] T. Albash and D.A. Lidar, Rev. Mod. Phys. **90**, 015002 (2018)

[27] C. Zener, Proc. R. Soc. London, Ser A **137**, 696 (1932)

[28] R. Harris, M.W. Johnson, T. Lanting, A.J. Berkley, J. Johansson, P. Bunyk, E. Tolkacheva, E. Ladizinsky, N. Ladizinsky, T. Oh, F. Cioata, I. Perminov, P. Spear, C. Enderud, C. Rich, S. Uchaikin, M.C. Thom, E.M. Chapple, J. Wang, B. Wilson, M.H.S. Amin, N. Dickson, K. Karimi, B. Macready, C.J.S. Truncik, and G. Rose, Phys. Rev. B **82**, 024511 (2010)

[29] R. Harris, T. Lanting, A.J. Berkley, J. Johansson, M.W. Johnson, P. Bunyk, E. Ladizinsky, N. Ladizinsky, T. Oh, and S. Han, Phys. Rev. B **80**, 052506 (2009)

[30] R. Harris, J. Johansson, A.J. Berkley, M.W. Johnson, T. Lanting, S. Han, P. Bunyk, E. Ladizinsky, T. Oh, I. Perminov, E. Tolkacheva, S. Uchaikin, E.M. Chapple, C. Enderud, C. Rich, M. Thom, J. Wang, B. Wilson, and G. Rose, Phys. Rev. B **81**, 134510 (2010)

[31] M.H.S. Amin, Phys. Rev. Lett. **102**, 220401 (2009)

[32] *D-Wave Ocean SDK* `https://github.com/dwavesystems/dwave-ocean-sdk`

[33] *D-Wave Ocean Software Documentation* `https://docs.ocean.dwavesys.com/en/stable/`

[34] D-Wave Systems: *D-Wave Solver Properties and Parameters*. Tech. rep., D-Wave Systems Inc., Burnaby, BC, Canada (2021). D-Wave User Manual 09-1169A-S `https://docs.dwavesys.com/docs/latest/doc_solver_ref.html`

[35] *Problem-Solving Handbook: Reformulating a Problem* `https://docs.dwavesys.com/docs/latest/handbook_reformulating.html`