1 SETTING UP

In this chapter we describe the environment, libraries and tools we use to execute our tests.

In the following sections we install the SDKs to develop and interact with quantum computers from IBM and D-Wave. We also present two other useful tools to easily write optimization problems.

1.1 PYTHON ENVIRONMENT

The language used to interface with quantum computers is usually Python. In this section we create a virtual environment in Python in order to communicate with the IBM quantum computer and the D-Wave quantum computer.

For our tests we manage Python environments with `conda`. Let's start by creating the virtual environment named `quantum` and activating it with:

```
conda create --name quantum python=3.12 pip
conda activate quantum
```

For our tests and to follow the various examples presented both by IBM and D-Wave, it is also useful to be able to run a Jupyter notebook. We can install Jupyter with:

```
pip install jupyter
```

1.2 IBM QISKIT

To program a gate-based architecture and to access IBM quantum computers we use the *Qiskit* software stack. The name Qiskit is a general term referring to a collection of softwares for executing programs on quantum computers.

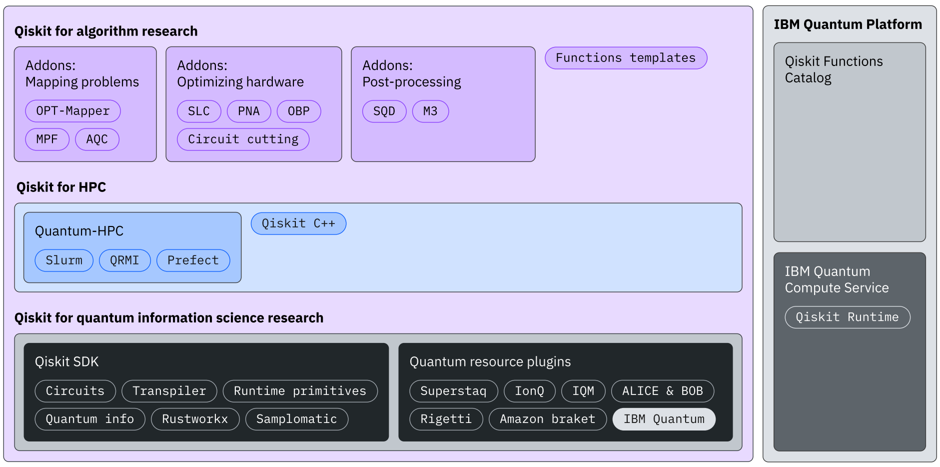


Figure 1.1: Qiskit software stack

The core components are *Qiskit SDK* and *Qiskit Runtime*. The first one is completely open source and allows the developer to define his circuit; the second one is a cloud-based service for executing quantum computations on IBM quantum computers.

1.2.1 Hello World

Following the IBM documentation[1] we can install the SDK and the Runtime with:

```
1 pip install qiskit matplotlib qiskit[visualization]
2 pip install qiskit-ibm-runtime
3 pip install qiskit-aer
```

Line 3 installs Aer, which is a high-performance simulator for quantum circuits written in Qiskit. Aer includes realistic noise models, and we will use it later to test our circuit.

Sometimes the Qiskit stack suffers from incompatibilities between the various software components that compose the environment. At the moment of writing, the latest packages seem to work without any problem. For our tests we will use `qiskit: 2.2.3`, `qiskit-ibm-runtime: 0.43.1` and `qiskit-aer: 0.17.2`.

If the setup is successful we are now able to run a small test to build a Bell state (two entangled qubits). The following code assembles the gates, shows the final circuit and uses a sampler to simulate on the CPU the result of 1024 runs of the program.

```python
from qiskit import QuantumCircuit
from qiskit.primitives import StatevectorSampler

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all()

sampler = StatevectorSampler()
result = sampler.run([qc], shots=1024).result()
print(result[0].data.meas.get_counts())
qc.draw("mpl")
```

Listing 1: Building Bell state

1.2.2 Transpilation

Each Quantum Processing Unit (QPU) has a specific topology. We need to rewrite our quantum circuit in order to match the topology of the selected device on which we want to run our program. This phase of rewriting, followed by an optimization, is called transpilation.

1 https://quantum.cloud.ibm.com/docs/en/guides/install-qiskit

Considering, for now, a fake hardware (so we do not need an API key) we can transpile the quantum circuit `qc`, from the code above, to match the topology of a specific QPU:

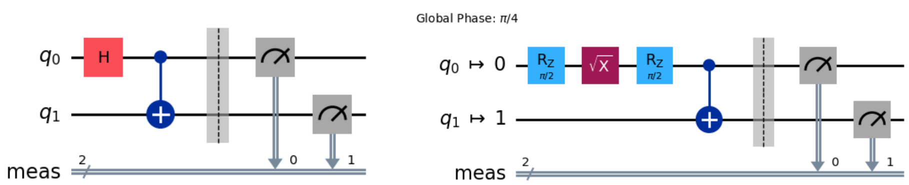
```python
from qiskit_ibm_runtime.fake_provider import FakeWashingtonV2
from qiskit.transpiler import generate_preset_pass_manager

backend = FakeWashingtonV2()
pass_manager = generate_preset_pass_manager(backend=backend)

transpiled = pass_manager.run(qc)
transpiled.draw("mpl")
```

Listing 2: Transpilation

The following picture shows (1.2a) the quantum circuit that builds a Bell state, and (1.2b) the transpiled version where the Hadamard gate is replaced to match the actual topology of the QPU.



(a) Original circuit **(b)** Transpiled circuit

Figure 1.2: Transpilation example

1.2.3 Execution

To test our transpiled circuit we use Aer, which allows us to simulate also the noise of real quantum hardware. We can execute our program with:

```python
from qiskit_aer.primitives import SamplerV2

sampler = SamplerV2.from_backend(backend)
job = sampler.run([transpiled], shots=1024)
result = job.result()
print(f"counts for Bell circuit : {result[0].data.meas.get_counts()}")
```

Listing 3: Simulated execution

If we look at the results of the execution we can observe that some answers present non-entangled qubits; this is caused by the (simulated) noise of the quantum device. A typical output of the execution could be:

```
> counts for Bell circuit : {'00': 504, '11': 503, '01': 10, '10': 7}
```

Where states `01` and `10` should not be present in an ideal execution with no errors.

1.2.4 A complete example on real hardware

1.3 D-WAVE OCEAN

To define an optimization problem that can be solved on a D-Wave quantum computer we use the Ocean software stack. Ocean also allows us to interact with D-Wave hardware, submit a problem, and simulate the execution on a classical CPU.

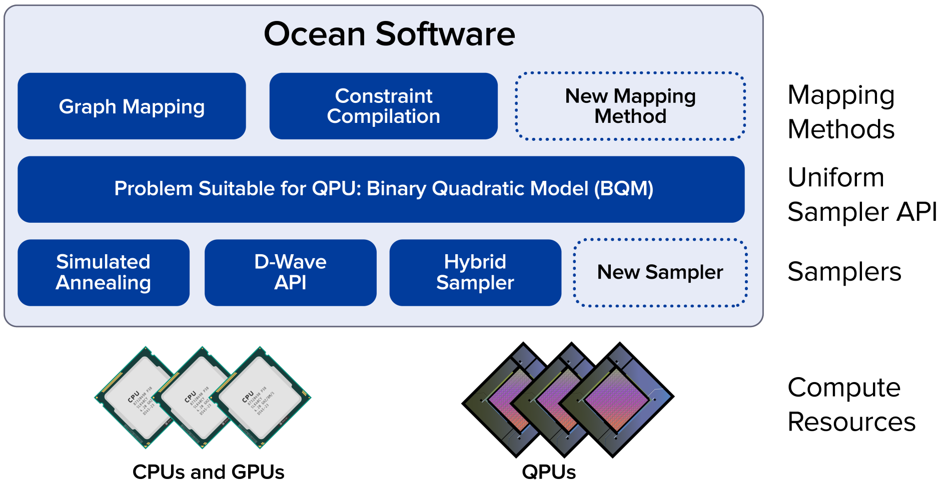


Figure 1.3: Ocean software stack

All tools that implement the steps needed to solve your problem on a CPU, a D-Wave quantum computer, or a quantum-classical hybrid solver can be installed with:

```
pip install dwave-ocean-sdk
```

After the installation, running the command `dwave setup` will start an interactive prompt that guides us through a full setup. During the setup it is also possible to add an API token or connect to the D-Wave account to import a key directly to use the quantum hardware.

1.3.1 Hello World

To present a simple optimization program we consider the minimum vertex cover (MVC) problem. Given a graph $G = (V, E)$, the problem asks to find a subset $V' \subseteq V$ such that, for each edge $\{u, v\} \in E$, at least one of u or v belongs to V', and the number of nodes in V' ($|V'|$) is the lowest possible.

The reduction from MVC to an Ising formulation is well known. The cost function that we want to minimize can be expressed by:

$$\text{cost} = \sum_{i=1}^{|V|} v_i + 2 \cdot \sum_{\{i,j\} \in E} (1 - v_i - v_j + v_i v_j)$$

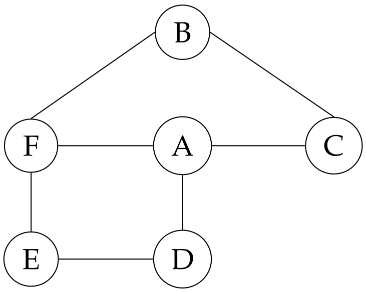
where $v_i \in \{-1, 1\}$ and $v_i = 1$ means that $v_i \in V'$, otherwise $v_i = -1$.

Like all problems in Ising form we can express the cost as a symmetric matrix, so our function becomes

$$\text{cost} = v^{T} \times \mathbf{M} \times v$$

where v is the vector containing the binary variables v_i.

The figure shows an example graph (1.4a) and the corresponding matrix (1.4b) expressing the cost function.



(a) Original graph

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 2 & 2 & 0 & 2 \\ 0 & 1 & 2 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) Corresponding matrix

Figure 1.4: Ising formulation

The following code presents a possible implementation of the Ising model described above. We have defined two dictionaries to store the matrix coefficients. The last line of code finds ten possible answers to the problem using the simulated annealing function implemented by D-Wave.

```
from dwave.samplers import SimulatedAnnealingSampler
linear = {'A': 1, 'B': 1, 'C': 1, 'D': 1, 'E': 1, 'F': 1}
quadratic = {('B', 'C'): 2, ('B', 'F'): 2, ('C', 'A'): 2, ('D',
↪  'A'): 2, ('E', 'D'): 2, ('E', 'F'): 2, ('F', 'A'): 2}
sampler = SimulatedAnnealingSampler()
result = sampler.sample_ising(linear, quadratic, num_reads=10)
```

Listing 4: Ising example

If we print the results with `print(result.aggregate())` we can observe something similar to this:

```
  A  B  C  D  E  F energy num_oc.
0 -1 -1 +1 +1 -1 +1  -14.0       6
1 +1 +1 -1 -1 +1 -1  -14.0       4
['SPIN', 2 rows, 10 samples, 6 variables]
```

The two different results represent the two correct answers to our particular instance of the MVC problem.

1.3.2 Example on real hardware

1.3.3 Minor embedding

1.4 PYQUBO AND QUBOVERT

In listing 4 we have manually built the matrix representing the function that we want to minimize. It can be useful to have some tools that allow us to work at a higher level, defining cost functions like **??** that we defined in the section about quantum annealing (**??**).

Considering again the MVC problem, the objective function tends to minimize the number of nodes in our subset, while the penalty increases the cost if we leave out some edges. This interpretation allows us to transform the Ising model into the more familiar —from the point of view of a computer scientist— QUBO model, where all variables $x_i \in \{0, 1\}$. Let's see how PyQUBO and qubovert help us in this task.

1.4.1 PyQUBO

Reading from the documentation on the PyQUBO site[2], PyQUBO allows us to create QUBOs or Ising models from flexible mathematical expressions easily. Some of the features of PyQUBO are:

- Python based (C++ backend);
- Fully integrated with Ocean SDK;
- Automatic validation of constraints;
- Placeholder for parameter tuning.

We can install PyQUBO with `pip install pyqubo` and rewrite our MVC problem by defining the Hamiltonian that we want to minimize.

```python
from pyqubo import Binary, Placeholder, Constraint
from dwave.samplers import SimulatedAnnealingSampler

A, B, C, D, E, F  = Binary('A'), Binary('B'), Binary('C'), Binary('D'),
↪  Binary('E'), Binary('F')

H_objective = (A + B + C + D + E + F)
H_penalty = Constraint(((1 - A - C + A*C) +\
(1 - A - D + A*D) +\
(1 - A - F + A*F) +\
(1 - B - C + B*C) +\
(1 - B - F + B*F) +\
(1 - D - E + D*E) +\
(1 - E - F + E*F)) ,label='cnstr0')

L = Placeholder('L')
H = H_objective + L*H_penalty
H_internal = H.compile()
bqm = H_internal.to_bqm(feed_dict={'L': 2})

sampler = SimulatedAnnealingSampler()
result = sampler.sample(bqm, num_reads=10)
```

Listing 5: Rewriting MVC with pyQUBO

2 https://pyqubo.readthedocs.io/en/latest/

Listing 5 presents a possible re-implementation of listing 4, where we also see how PyQUBO interfaces with the Ocean SDK (line 17), and how to create (lines 14–16) and instantiate (line 17) a parametric Hamiltonian.

1.4.2 qubovert

As written in the documentation[3], qubovert is the one-stop package for formulating, simulating, and solving problems in boolean and spin form. Using our nomenclature, boolean and spin form are respectively QUBO and Ising form.

Qubovert allows us to define various types of optimization problems that can be solved by brute force, with qubovert's simulated annealing, or with D-Wave's solver. Models defined in qubovert are:

- QUBO: Quadratic Unconstrained Boolean Optimization;
- QUSO: Quadratic Unconstrained Spin Optimization (Ising model);
- PUBO: Polynomial Unconstrained Boolean Optimization;
- PUSO: Polynomial Unconstrained Spin Optimization;
- PCBO: Polynomial Constrained Boolean Optimization;
- PCSO: Polynomial Constrained Spin Optimization.

In addition to generic models, qubovert has a library of famous NP-complete problems mapped to QUBO and Ising forms.

```
from qubovert import boolean_var
from dwave.samplers import SimulatedAnnealingSampler

A, B, C, D, E, F  = boolean_var('A'), boolean_var('B'),
↪  boolean_var('C'), boolean_var('D'), boolean_var('E'),
↪  boolean_var('F')

model = A + B + C + D + E + F
model.add_constraint_OR(A, C, lam=2)
model.add_constraint_OR(A, D, lam=2)
model.add_constraint_OR(A, F, lam=2)
model.add_constraint_OR(B, C, lam=2)
model.add_constraint_OR(B, F, lam=2)
model.add_constraint_OR(D, E, lam=2)
model.add_constraint_OR(E, F, lam=2)

qubo = model.to_qubo()
dwave_qubo = qubo.Q

sampler = SimulatedAnnealingSampler()
result = sampler.sample_qubo(dwave_qubo, num_reads=10)
```

Listing 6: Rewriting MVC with qubovert

3 https://qubovert.readthedocs.io/en/latest/index.html

Listing 6 shows a possible implementation of the MVC problem using the tools provided by qubovert. Qubovert allows us to express our problem as a PCBO; we use this formulation to express constraints in a more natural way. In our example we ensure that each edge is covered simply by enforcing that at least one of the nodes linked by the edge is present in the solution. This constraint is repeated for each edge in the graph (lines 7–13). To specify the Lagrange multiplier (equation **??**) we use the keyword `lam`.

Qubovert, like PyQUBO, can interface with the Ocean SDK, transforming a PCBO problem into a QUBO problem (line 15) and then rewriting it in the format accepted by the D-Wave solver (or sampler).

1.5 CONCLUSION

In this chapter we have set up an environment to run our future experiments and tests. We have also shown some small examples to present the main characteristics and test the tools we will use in our work.

Following this setup allows anyone to recreate exactly the same configuration we use, avoiding (for what we know and test) incompatibilities between Python packages.