1 SETTING UP

In this chapter we describe the environment we use to do our test, the tool

1.1 PYTHON ENVIRONMENT

The language used to interface with quantum computer 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 test we manage pyton environments with `conda`, let's start creating the virtual env named `quantum` and activate it with:

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

For our test and to follow the various example presented both by IBM and D-Wave is also useful to been able of running a Jupyter notebook. We can install Jupyter with:

```
pip install jupyter
```

1.2 IBM QISKIT

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

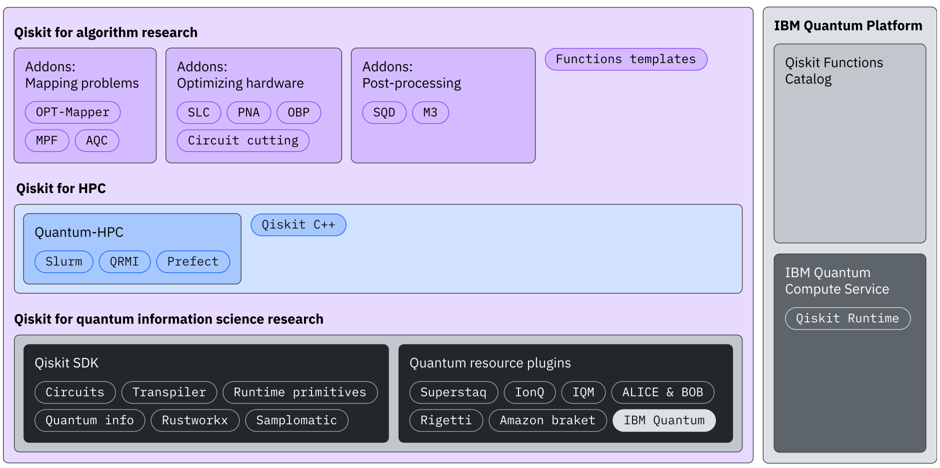


Figure 1.1: Qiskit software stack

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 computer.

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 install Aer, that is a high performance simulator for quantum circuits written in Qiskit. Aer includes realistic noise models, and we will use later to test our circuit.

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

If the setup had success we are now able to run a small test to build a Bell state (two entangled qubits). The following code assemble the gates, show the final circuit and use 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: hello qiskit

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 witch we want to run our program. This phase of rewriting, followed by an optimization, is called transpilation.

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

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

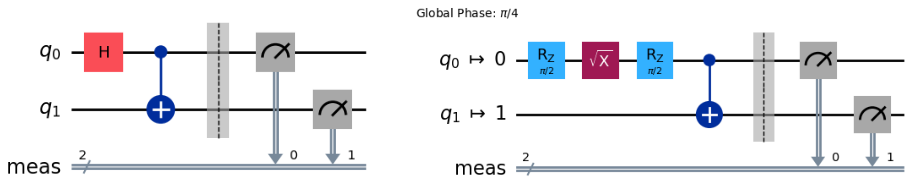
```
from qiskit_ibm_runtime.fake_provider import FakeWashingtonV2
from qiskit.transpiler import generate_preset_pass_manager

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

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

Listing 2: Transpilation

The following picture show (1.2a) the quantum circuit that build 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 that allow us to simulate also the noise of a real quantum hardware. We can execute our program with:

```
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 could observe that some answers present non entangled qbit, 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 state `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 resolved on a D-Wave quantum computer we use the Ocean software stack. Ocean, also, allow us to interact with D-Wave hardware, submit a problem and to simulate the execution on a classical CPU.

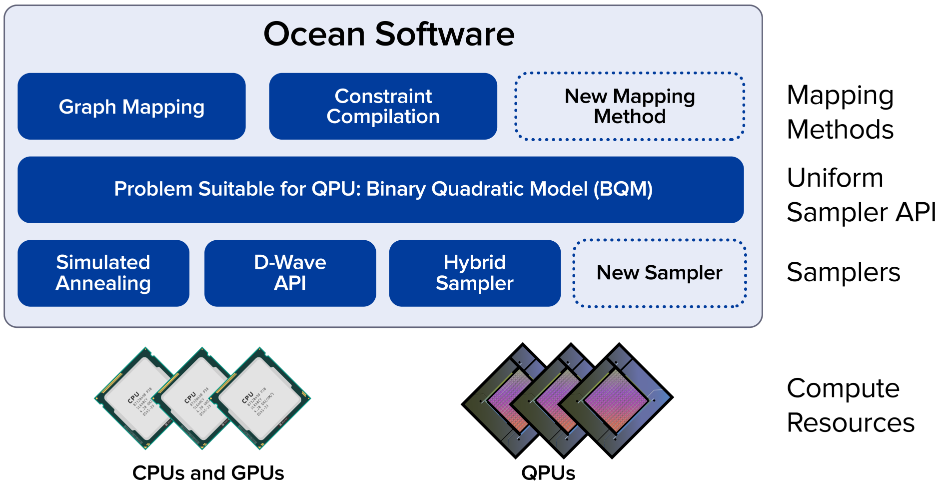


Figure 1.3: Ocean software stack

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 guide us through a full setup. During the setup is also possible adding a API token or connecting to the D-Wave account to import directly a key to use the quantum hardware.

1.3.1 Hello World

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

The reduction from MVC to Ising formulation is well known, the cost function, that we want to minimize, could be expressed by:

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

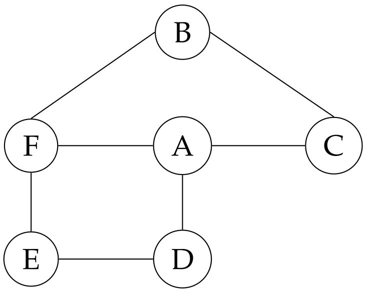
Where $v_i \in \{-1, 1\}$ and if $v_i = 1$ means that $v_i \in V'$, otherwise $v_i = -1$.

Like all problem in Ising form we can express the cost as a symmetrical matrix, so our function become

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

Were 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 present a possible implementation of the Ising model described above. We have defined two dictionary to memorize the matrix coefficient. The last line of code founds ten possible answers to the problem using the simulated annealing function implemented by D-Wave.

```python
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 could 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 answer 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 ?? 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 working at higher level defining the function like:

$$cost = objective + \lambda \cdot penalty$$