Introducing Qibocal 0.0.7

Qubit calibration using Qibo

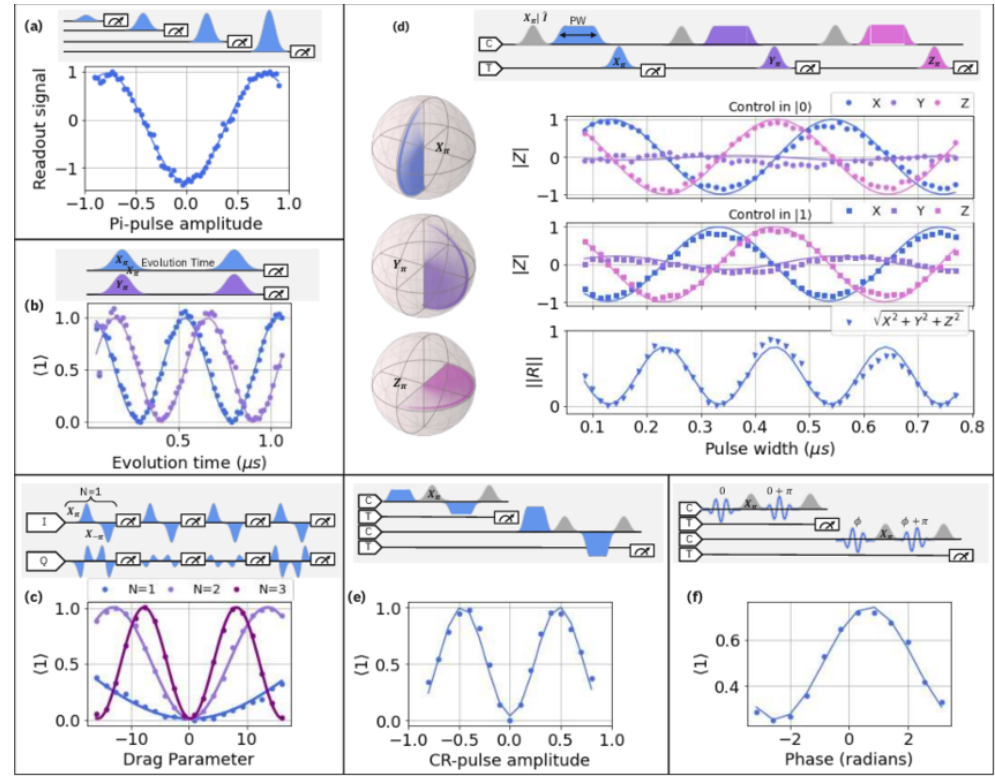
How to calibrate superconducting devices?

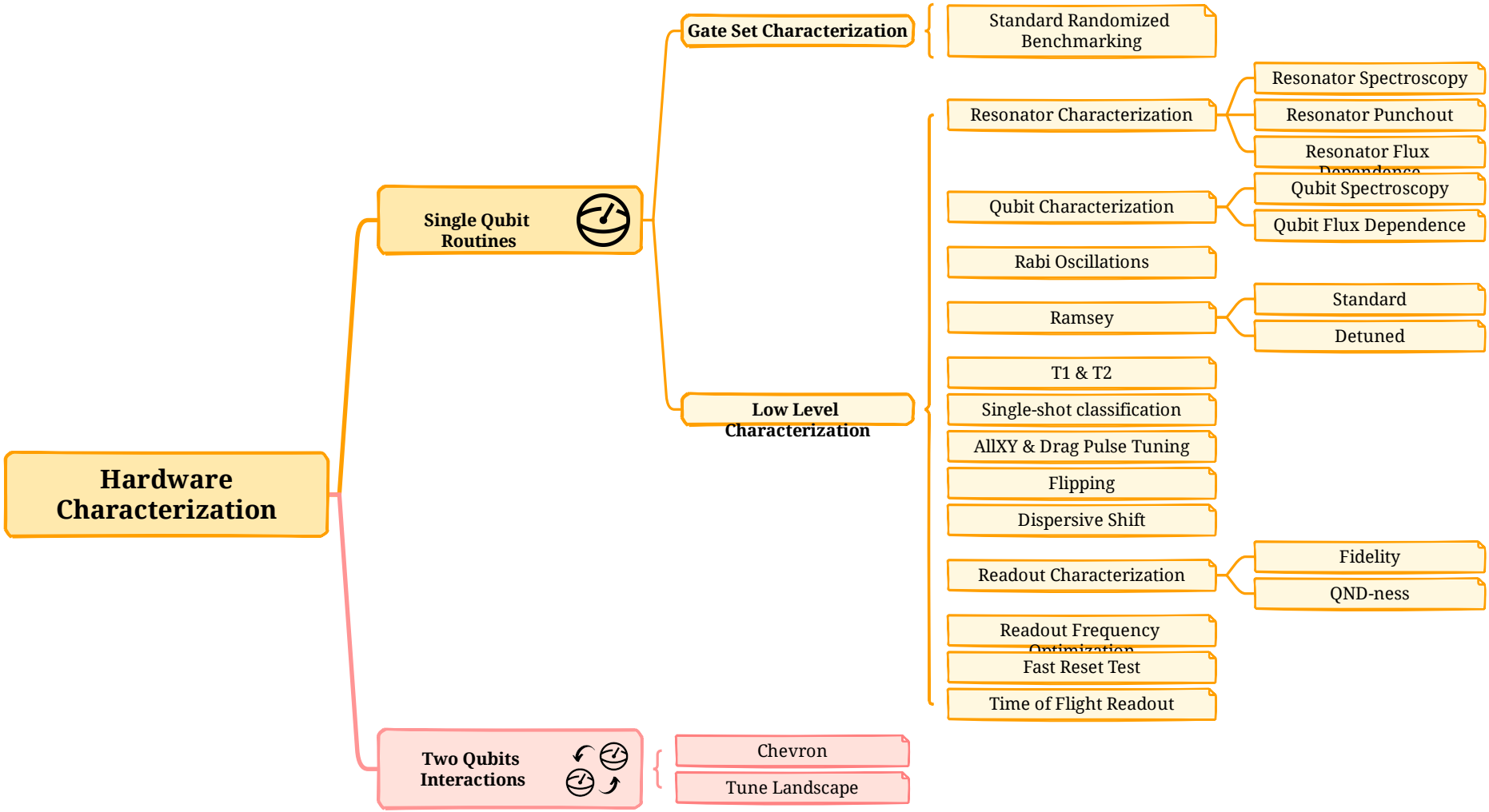
In superconducting qubits gates are implemented through microwave pulses.

Several protocols need to be executed to extract specific parameters.

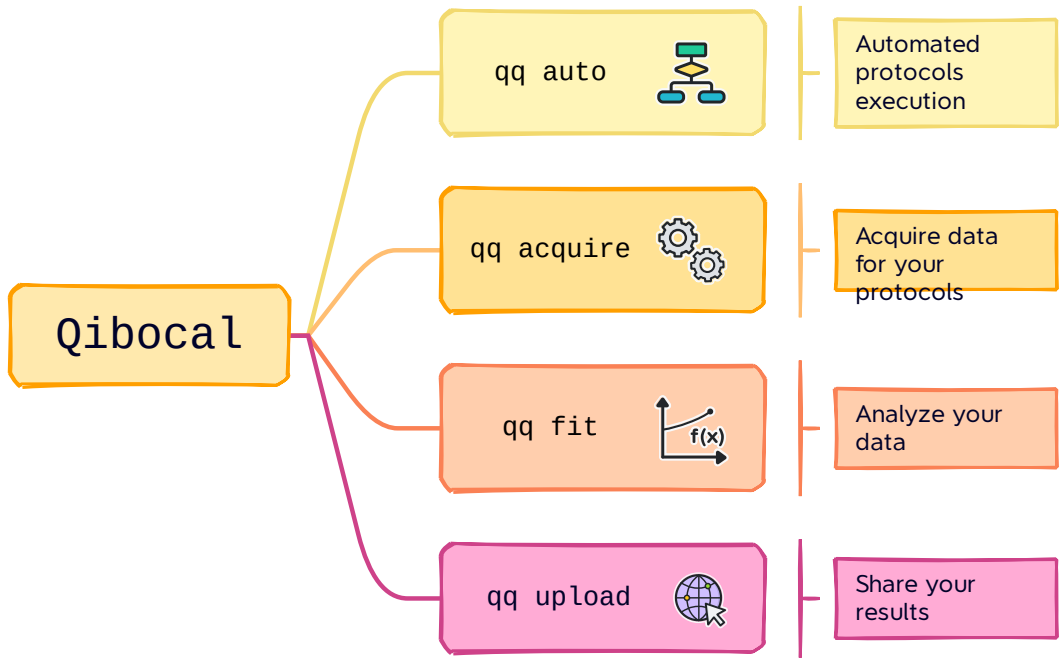
After an initial calibration more advanced experiments can be performed in order to:

- improve readout
- run benchmarking protocols
- reach optimal control




Hardware Characterization
Single Qubit Routines
Gate Set Characterization
Standard Randomized Benchmarking
Low Level Characterization
Resonator Characterization
Resonator Spectroscopy
Resonator Punchout
Resonator Flux Dependence
Qubit Characterization
Qubit Spectroscopy
Qubit Flux Dependence
Rabi Oscillations
Ramsey
Standard
Detuned
T1 & T2
Single-shot classification
AllXY & Drag Pulse Tuning
Flipping
Dispersive Shift
Readout Characterization
Fidelity
QND-ness
Readout Frequency Optimization
Fast Reset Test
Time of Flight Readout
Two Qubits Interactions
Chevron
Tune Landscape

Qibocal workflow



Routine: data acquisition

```python
from dataclasses import dataclass
from qibocal.auto.operation import Parameters, Data
from qibolab.platform import Platform
from qibolab.qubits import QubitId


@dataclass
class RoutineParameters(Parameters):
    """Input parameters for YAML runcard."""


@dataclass
class RoutineData(Data):
    """Data structure for acquisition data."""


def acquisition(params: RoutineParameters,
                platform: Platform,
                qubits: list[QubitId]) -> RoutineData:
    """Acquisition protocol."""
```

`RoutineParameters` experiment configuration.

`RoutineData` data acquired by the protocol.

`RoutineParameters` and `RoutineData` are connected through `acquisition` which is the function which will presumably use `Qibolab` code to perform acquisition.

Routine: post-processing

```python
from dataclasses import dataclass
from qibolab.platform import Platform
from qibolab.qubits import QubitId
from qibocal.auto.operation import Results


@dataclass
class RoutineResults(Results):
    """Post-processed results."""


def fit(data: RoutineData) -> RoutineResults:
    """Extracting features from data."""


def update(results: RoutineResults,
           qubit: QubitId,
           platform: Platform) -> None:
    """Updating platform parameters'."""
```

`RoutineResults` class containing the analysis of the raw data.

`fit` optional function which performs the post-processing analysis.

`update` optional function which updates specific calibration parameters computed in `Results`.

Routine: reporting

```python
import plotly.graph_objects as go
from qibolab.qubits import QubitId
from qibocal.auto.operation import Routine


def report(data: RoutineData,
           qubit: QubitId,
           results: RoutineResults) -> tuple[list[go.Figure], str]:
    """Updating platform parameters'."""




# define Routine object
routine = Routine(acquisition, fit, report, update)
```

`report` is the function that plots the data or results obtained. The return type should include the following:

- list of `go.Figure`, which are figure realized with plotly
- a str, which is a generic HTML code that can be injected and it will be rendered correctly in the report

We can create an instance of `Routine` by passing `acquisition`, `fit`, `report` and `update`.

How to add a new protocol?

Suppose that we want to code a protocol to perform a RX rotation for different angles.

Parameters

First, we define the input parameters of our experiment inheriting the Qibocal *Parameters* class.

```
from dataclasses import dataclass
from ...auto.operation import Parameters
@dataclass
class RotationParameters(Parameters):
    """Parameters for rotation protocol."""
    theta_start: float
    """Initial angle."""
    theta_end: float
    """Final angle."""
    theta_step: float
    """Angle step."""
    nshots: int
    """Number of shots."""
```

Data

Secondly, we define a data structure that aims at storing both the angles and the probabilities measured for each qubit.

```
import numpy as np
import numpy.typing as npt
from dataclasses import dataclass, field
from ...auto.operation import Data

RotationType = np.dtype([("theta", np.float64), ("prob", np.float64)])

@dataclass
class RotationData(Data):
    """Rotation data."""

    data: dict[QubitId, npt.NDArray[RotationType]] =
        field(default_factory=dict)
    """Raw data acquired."""
```

Data acqusition

```python
from qibolab.platform import Platform
from ...auto.operation import Qubits
def acquisition(
    params: RotationParameters,
    platform: Platform,
    qubits: Qubits,
) -> RotationData:

    angles = np.arange(params.theta_start, params.theta_end, params.theta_step)

    data = RotationData()

    for angle in angles:

        circuit = Circuit(platform.nqubits)

        for qubit in qubits:
            circuit.add(gates.RX(qubit, theta=angle))
            circuit.add(gates.M(qubit))

        result = circuit(nshots=params.nshots)
```

In the acquisition function we are going to perform the experiment.

- We define the angle range according to the input parameters,
- allocate the data structure *RotationData*
- We build the circuit with an *RX* gate and execute it on hardware.

Data acquisition

```
        for qubit in qubits:
            circuit.add(gates.RX(qubit, theta=angle))
            circuit.add(gates.M(qubit))

        result = circuit(nshots=params.nshots)

        for qubit in qubits:

            prob = result.probabilities(qubits=[qubit])[0]

            data.register_qubit(qubit, theta=angle, prob=prob)

    return data
```

- Extract probability of 0,
- save the angles and the probabilities in the data structure,
- return the data.

Result class

Here we decided to code a generic Result that contains the fitted parameters for each quibt.

```
from qibolab.qubits import QubitId

@dataclass
class RotationResults(Results):
    """Results object for data"""
    fitted_parameters: dict[QubitId, list] = field(default_factory=dict)
```

Fit function

The following function performs a sinusoidal fit for each qubit.

```
def fit(data: RotationData) -> RotationResults:
    qubits = data.qubits
    freqs = {}
    fitted_parameters = {}

    def cos_fit(x, offset, amplitude, omega):
        return offset + amplitude * np.cos(omega*x)

    for qubit in qubits:
        qubit_data = data[qubit]
        thetas = qubit_data.theta
        popt, _ = curve_fit(cos_fit, thetas, qubit_data.prob)
        freqs[qubit] = popt[2] / 2*np.pi
        fitted_parameters[qubit]=popt.tolist()
    return RotationResults(
        fitted_parameters=fitted_parameters,
    )
```

Plot function

The report function generates a list of figures and an optional table to be shown in the html report.

```
def plot(data: RotationData, fit: RotationResults, qubit):
"""Plotting function for rotation."""
    figures = []
    fig = go.Figure()
    fitting_report = ""
    qubit_data = data[qubit]
    fig.add_trace(
        go.Scatter(
            x=qubit_data.theta,
            y=qubit_data.prob,
    ...

    return figures, fitting_report
```

For more details about this function look at the doc.

Create Routine object

```
rotation = Routine(acquisition, fit, plot)
"""Rotation Routine  object."""
```

Add routine to Operation Enum

```
# other imports ...
from rotate import rotation

class Operation(Enum):
## other protocols ...
rotation = rotation
```

Write a runcard

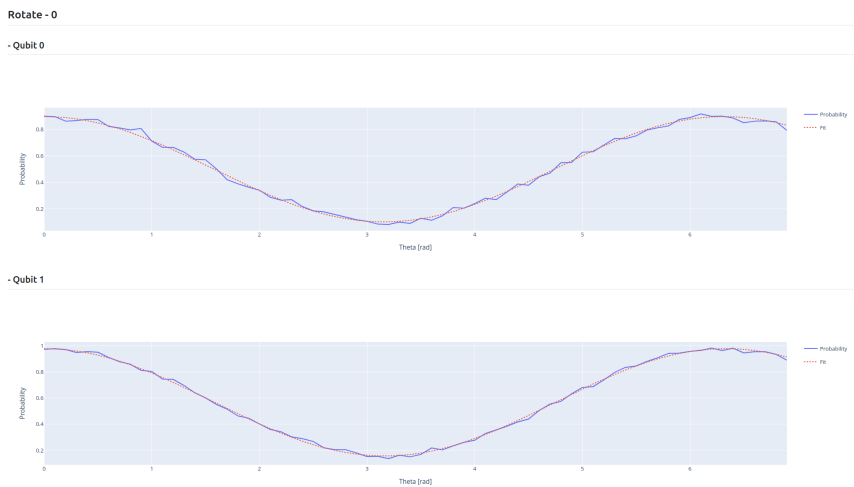
```
platform: dummy

qubits: [0,1]


actions:
    - id: rotate
      priority: 0
      operation: rotation
      parameters:
        theta_start: 0
        theta_end: 7
        theta_step: 20
        nshots: 1024
```

Run the routine

After running `qq auto` a report is generated.



Qibocal as a library

Qibocal also allows executing protocols without the standard interface.

We show how to run a single protocol using Qibocal as a library

```
from qibocal.protocols.characterization import Operation
from qibolab import create_platform

# allocate platform
platform = create_platform("....")
# get qubits from platform
qubits = platform.qubits

# we select the protocol
experiment = Operation.single_shot_classification.value
```

In order to run a protocol the user needs to specify the parameters.

```
parameters = experiment.parameters_type.load(dict(nshots=1024))
```

The user can perform the acquisition using *experiment.acquisition*

```
data, acquisition_time = experiment.acquisition(
    params=parameters, platform=platform, qubits=qubits)
```

The fitting corresponding to the experiment can be launched in the following way:

```
fit, fit_time = experiment.fit(data)
```

it is also possible to access the plots and the tables with the following lines.

```
# Plot for qubit 0
qubit = 0
figs, html_content = experiment.report(
    data=data, qubit=0, fit=fit)
```

Thanks for listening

Useful pages in documentation

How to execute a single protocol in `Qibocal`?

In `Qibocal` we adopt a declarative programming paradigm, i.e. the user should specify directly what he wants to do without caring about the underlying implementation.

This paradigm is implemented in `Qibocal` in the form of runcards. A runcard is essentally a set of instructions that are specified in a file.

Down below we present how to write a runcard to execute a single protocol using *qq*.

```
backend: <qibo backend>

platform: <qibolab platform name>

qubits: <list of qubit ids where all the protocols will be performed.>


actions:
    - id: <protocol id>
      priority: 0
```

Advanced examples

How to use Qibocal as a library

Qibocal also allows executing protocols without the standard interface.

In the following tutorial we show how to run a single protocol using Qibocal as a library. For this particular example we will focus on the single shot classification protocol.

```
from qibocal.protocols.characterization import Operation
from qibolab import create_platform

# allocate platform
platform = create_platform("....")
# get qubits from platform
qubits = platform.qubits

# we select the protocol
protocol = Operation.single_shot_classification.value
```

`protocol` is a Routine object which contains all the necessary methods to execute the experiment.