Qibolab

Getting started tutorial

`qibolab_hello_world.py`

Instead of contents...

```python
from qibolab import create_platform
from qibolab.pulses import PulseSequence
from qibolab.execution_parameters import ExecutionParameters

platform = create_platform("myplatform")

sequence = PulseSequence()
ro_pulse = platform.create_MZ_pulse(qubit=0, start=0)
sequence.add(ro_pulse)

options = ExecutionParameters(nshots=1000)
results = platform.execute_pulse_sequence(sequence, options)
```

`Platform` represents the lab configuration, containing all information about the available qubits and orchestrating the instruments.

`PulseSequence` contains the pulses to be executed. Pulses can be constructed manually through the pulse API, or via the `platform`.

The experiment is deployed using the `Platform`.

Pulse API

```python
from qibolab.pulses import Pulse, PulseType, Rectangular

pulse = Pulse(
    start=0,          # Timing in nanoseconds (ns)
    duration=40,      # Pulse duration in ns
    amplitude=0.5,    # Amplitude of the waveform
    frequency=1e8,    # Frequency in Hz
    relative_phase=0, # Phase in radians
    shape=Rectangular(),
    type=PulseType.DRIVE,
    qubit=0,
)
```

`PulseSequence` is a list of pulses

```python
sequence = PulseSequence()
sequence.add(Pulse( ... ))
```

Pulse waveforms can have different shapes

- `Rectangular`
- `Exponential`
- `Gaussian`
- `GaussianSquare`
- `Drag`

Pulses can have different types

- `PulseType.READOUT`
- `PulseType.DRIVE`
- `PulseType.FLUX`

Platform

```python
@dataclass
class Platform:
    qubits: QubitMap
    pairs: QubitPairMap
    instruments: InstrumentMap

    def connect(self):

    def disconnect(self):

    def execute_pulse_sequence(
        self,
        sequences: PulseSequence,
        options: ExecutionParameters
    ):

    def sweep(
        self,
        sequence: PulseSequence,
        options: ExecutionParameters,
        *sweepers: Sweeper
    ):
```

Platform contains information about

- `qubits`: characterization and native single-qubit gates
- `pairs`: connectivity and native two-qubit gates
- `instruments`: used to deploy pulses *(drivers)*

```python
@dataclass
class Qubit:
    readout_frequency: int
    drive_frequency: int

    readout: Optional[Channel]
    feedback: Optional[Channel]
    drive: Optional[Channel]

    native_gates: SingleQubitNatives
```

Qubits are connected to instruments via channels.

Creating platforms

```
def create():
    instrument = DummyInstrument("myinstr", "0.0.0.0:0")

    channels = ChannelMap()
    channels |= Channel(
        "readout",
        port=instrument.ports("o1")
    )
    channels |= Channel(
        "feedback",
        port=instrument.ports("i1", output=False)
    )

    qubit = Qubit(0)
    qubit.readout = channels["readout"]
    qubit.feedback = channels["feedback"]

    return Platform(
        "myplatform",
        qubits={qubit.name: qubit},
        pairs={},
        instruments={instrument.name: instrument},
    )
```

Instantiate instrument objects.

Create channels and connect them to instruments.

Create qubits and connect them to channels.

Instantiate platform with all the information.

Qubit parameters

```python
native_gates = SingleQubitNatives(
    MZ=NativePulse(
        name="MZ",
        duration=1000,
        amplitude=0.005,
        shape="Rectangular()",
        pulse_type=PulseType.READOUT,
        qubit=qubit,
        frequency=int(7e9),
    ),
    RX=NativePulse(
        ...
    ),
)

qubit = Qubit(
    name=0,
    readout_frequency=7e9,
    drive_frequency=4.5e9,
    native_gates=native_gates,
)
```

```yaml
native_gates:
    single_qubit:
        0:
            MZ:
                duration: 1000
                amplitude: 0.005
                frequency: 7_000_000_000
                shape: Rectangular()
                type: ro # readout
                start: 0
                phase: 0
            RX:
                ...

characterization:
    single_qubit:
        0:
            readout_frequency: 7_000_000_000
            drive_frequency: 4_500_000_000
```

Let's now put it all together...

Creating platforms

using `qibolab.serialize`

```
qibolab_platforms/
    myplatform/
        platform.py
        parameters.yml  # → parameters.json
        kernels.npz # (optional)
```

- `platform.py`: Contains the `create` method that initializes the `Platform`.
- `parameters.yml`: Contains parameters that are updated during calibration.
- other files (integration weights, etc.) can also be provided and loaded in `create`.

```
export QIBOLAB_PLATFORMS=./qibolab_platforms
```

```python
FOLDER = Path(__file__).parent

def create():
    instrument = DummyInstrument("myinstr", "0.0.0.0:0")

    channels = ChannelMap()
    channels |= ...

    runcard = load_runcard(FOLDER)
    qubits, couplers, pairs = load_qubits(runcard)

    qubits[0].readout = channels["readout"]
    qubits[0].feedback = channels["feedback"]
    qubits[0].drive = channels["drive"]

    return Platform(
        "myplatform",
        qubits,
        pairs,
        instruments={instrument.name: instrument},
    )
```

Acquiring results

```
platform = create_platform("myplatform")

sequence = PulseSequence()
ro_pulse = platform.create_MZ_pulse(qubit=0, start=0)
sequence.add(ro_pulse)

options = ExecutionParameters(
    nshots=1000,
    relaxation_time=100000,
    acquisition_type=AcquisitionType.DISCRIMINATION,
    averaging_mode=AveragingMode.SINGLESHOT

)
results = platform.execute_pulse_sequence(sequence, options)

print(results[ro_pulse.serial].samples)
print(results[0].samples)
```

`results` is a `dict` from `pulse.serial` to a results object.

Acquisition types:

- `RAW`: (I, Q) waveform
- `INTEGRATION`: (I, Q) voltage
- `DISCRIMINATION`: samples

Averaging modes:

- `SEQUENTIAL` *(not recommended)*
- `CYCLIC`
- `SINGLESHOT`

Real-time sweeps

```
platform = create_platform("myplatform")

sequence = PulseSequence()
ro_pulse = platform.create_MZ_pulse(qubit=0, start=0)
sequence.add(ro_pulse)

sweeper = Sweeper(
    parameter=Parameter.frequency,
    values=np.arange(-2e8, +2e8, 1e6),
    pulses=[ro_pulse],
    type=SweeperType.OFFSET,
)

options = ExecutionParameters(
    nshots=1000,
    relaxation_time=1000,
    acquisition_type=AcquisitionType.INTEGRATION,
    averaging_mode=AveragingMode.CYCLIC
)
results = platform.sweep(sequence, options, sweeper)
```

Executing sweeps in real time is usually faster because it requires less communication with the instruments.

Sequence unrolling

```
platform = create_platform("myplatform")

nsequences = 20
sequences = []
for _ in range(nsequences):
    sequence = PulseSequence()
    sequence.add(platform.create_MZ_pulse(qubit=0, start=sequence.finish))
    sequences.append(sequence)

options = ExecutionParameters(
    nshots=1000,
    relaxation_time=100000,
    acquisition_type=AcquisitionType.DISCRIMINATION,
    averaging_mode=AveragingMode.SINGLESHOT,
)
results = platform.execute_pulse_sequences(sequences, options)
```

Passing multiple sequences in a single call is usually faster because it requires less communication with the instruments.

Sometimes sequences need to be *batched* in order to fit in the instruments memory (WIP).

Thanks