C3

Nicolas Wittler, Federico Roy, Kevin Pack, Anurag Saha Roy, Nikla

Dec 23, 2021

API DOCUMENTATION:

1 Introduction to C^3 Toolset **3**
- 1.1 The Building Blocks . . . 3
  - 1.1.1 Quantum Device Model . . . 3
  - 1.1.2 Classical Control Electronics . . . 3
  - 1.1.3 Instructions . . . 3
- 1.2 Parameter Map . . . 4
- 1.3 Experiments . . . 4
- 1.4 Optimizers . . . 4
- 1.5 Libraries . . . 4

2 Parameter handling **5**
- 2.1 The opt_map . . . 6
- 2.2 Optimizer scaling . . . 8
- 2.3 Storing and reading . . . 8

3 Setup of a two-qubit chip with C^3 **11**
- 3.1 Imports . . . 11
- 3.2 Model components . . . 12
- 3.3 SPAM errors . . . 14
- 3.4 Control signals . . . 15
- 3.5 Gates-set and Parameter map . . . 17
- 3.6 Instructions . . . 19
- 3.7 The experiment . . . 20
- 3.8 Simulation . . . 21

4 Dynamics **23**

5 Open-loop optimal control **27**

6 Entangling gate on two coupled qubits **31**
- 6.1 Imports . . . 31
- 6.2 Model components . . . 32
- 6.3 Control signals . . . 33
- 6.4 Gates-set and Parameter map . . . 34
- 6.5 Instructions . . . 35
- 6.6 The experiment . . . 36
- 6.7 Dynamics . . . 37
- 6.8 Open-loop optimal control . . . 41
- 6.9 Results of the optimisation . . . 43

7 Simulated calibration **47**

7.1 ORBIT - Single-length randomized benchmarking . . . 47
7.2 Communication with the experiment . . . 48
7.3 Optimization . . . 49
7.4 Representation of the experiment within C^3 . . . 50
7.5 Algorithms . . . 53
7.6 Analysis . . . 55

8 Model Learning **57**
8.1 Imports . . . 57
8.1.1 The Dataset . . . 57
8.1.2 The Model for Model Learning . . . 60
8.2 Define Constants . . . 60
8.3 Model . . . 61
8.4 Generator . . . 61
8.5 Gateset . . . 62
8.6 Experiment . . . 64
8.6.1 Optimizer . . . 64
8.6.2 Model Learning . . . 65
8.7 Result of Model Learning . . . 65
8.7.1 Visualisation & Analysis of Results . . . 65
8.8 Open, Clean-up and Convert Logfiles . . . 66
8.9 Summary of Logs . . . 66
8.10 Plotting . . . 67
8.11 Qubit Anharmonicity . . . 67
8.12 Qubit Frequency . . . 68
8.13 Goal Function . . . 69

9 Sensitivity Analysis **71**

10 Logs and current optimization status **75**

11 C3 Simulator as a backend for Qiskit Experiments **77**
11.1 Define a basic Quantum circuit . . . 77
11.2 Get the C3 Provider and Backend . . . 77
11.2.1 Let's view how the Qiskit Transpiler will convert the circuit . . . 78
11.3 Run an ideal device simulation using C3 . . . 78
11.4 Run Simulation and verify results on Qiskit simulator . . . 78

12 API Documentation **79**
12.1 C3objs . . . 79
12.2 Experiment module . . . 80
12.3 Model module . . . 83
12.4 Parameter map . . . 85
12.5 Main module . . . 87
12.6 Module contents . . . 87
12.7 Subpackages . . . 87
12.7.1 Generator package . . . 87
12.7.1.1 Submodules . . . 87
12.7.1.2 Devices module . . . 87
12.7.1.3 Generator module . . . 94
12.7.1.4 Module contents . . . 95
12.7.2 Libraries package . . . 95
12.7.2.1 Algorithms module . . . 95
12.7.2.2 Chip module . . . 99
12.7.2.3 Constants module . . . 105

12.7.2.4 Envelopes module . . . 105
12.7.2.5 Estimators module . . . 107
12.7.2.6 Fidelities module . . . 108
12.7.2.7 Hamiltonians module . . . 112
12.7.2.8 Sampling module . . . 114
12.7.2.9 Tasks module . . . 115
12.7.2.10 Module contents . . . 116
12.7.3 Optimizers . . . 116
12.7.3.1 C1 - Optimal control . . . 116
12.7.3.2 C2 - Calibration . . . 117
12.7.3.3 C3 - Characterization . . . 118
12.7.3.4 Optimizer module . . . 119
12.7.3.5 Sensitivity analysis . . . 121
12.7.3.6 Module contents . . . 122
12.7.4 Signal package . . . 122
12.7.4.1 Submodules . . . 122
12.7.4.2 Gates module . . . 122
12.7.4.3 Pulse module . . . 123
12.7.4.4 Module contents . . . 124
12.7.5 Utilities package . . . 124
12.7.5.1 Qutip utilities module . . . 124
12.7.5.2 Tensorflow utilities module . . . 127
12.7.5.3 Log Reader utilities module . . . 130
12.7.5.4 Miscellaneous utilities module . . . 130
12.7.5.5 Module contents . . . 131
12.7.6 Qiskit modules for C3 . . . 131
12.7.6.1 C3 Backend module . . . 131
12.7.6.2 C3 Provider module . . . 134
12.7.6.3 C3 Job module . . . 134
12.7.6.4 C3 Exceptions module . . . 134
12.7.6.5 C3 Backend Utilities module . . . 135
12.7.6.6 Module contents . . . 136

13 Indices and tables **137**

Index **139**

The C^3 software package provides tools to simulate and interact with experiments to perform common control and characterization tasks. Modules can be used individually or combined to achieve a certain goal. The main focus are three optimizations:

- C_1 Open-loop optimal control: Given a model, find the pulse shapes which maximize fidelity with a target operation.
- C_2 Closed-loop calibration: Given pulses, calibrate their parameters to maximize a figure of merit measured by the actual experiment, thus improving beyond the limits of a deficient model.
- C_3 Model learning: Given control pulses and their experimental measurement outcome, optimize model parameters to best reproduce the results.

When combined in sequence, these three procedures represent a recipe for system characterization.

Note: This documentation is work-in-progress.

CHAPTER

ONE

INTRODUCTION TO C^3 TOOLSET

In this section, we go over the foundational components and concepts in C^3 with the primary objective of understanding how the different sub-modules inside the `c3-toolset` are structured, the purpose they serve and how to tie them together into a complete Automated Quantum Device Bring-up workflow. For more detailed examples of how to use the `c3-toolset` to perform a specific Quantum Control task, please check out the *Setup of a two-qubit chip with C^3* or the *Simulated calibration* sections or refer to the *API Documentation* for descriptions of Classes and Functions.

1.1 The Building Blocks

There are three basic building blocks that form the foundation of all the modelling and calibration tasks one can perform using `c3-toolset`, and depending on the use-case, some or all of these blocks might be useful. These are the following:

1.1.1 Quantum Device Model

A theoretical Physics-based model of the Quantum Processing Unit. This is encapsulated by the `Model` class which consists of objects from the `chip` and `tasks` library. `chip` contains Hamiltonian models of different kinds of qubit realisations, along with their couplings while `tasks` let you perform common operations such as qubit initialisation or readout. A typical `Model` object would contain objects encapsulating qubits along with their interactions as drive lines and tasks, if any.

1.1.2 Classical Control Electronics

A digital twin of the electronic control stack associated with the Quantum Processing Unit. The `Generator` class contains the required encapsulation in the form of `devices` which help model the behaviour of the classical control electronics taking account of their imperfections and physical realisations. The devices e.g, an LO or an AWG or a Mixer are wired together in the `Generator` object to form a complete representation of accessory electronics.

1.1.3 Instructions

Once there is a software model for the QPU and the control electronics, one would need to define Instructions or operations to be perform on this device. For gate-based quantum computing , this is in the form of gates and their underlying pulse operations. Pulse shapes are described through a `Envelope` along with a `Carrier`, which are then wrapped up in the form of `Instruction` objects. The sequence in which these gates are applied are not defined at this stage.

Warning: Components inside the `c3/generator/` and `c3/signal/` sub-modules will be restructured in an upcoming release to be more consistent with how the `Model` class encapsulates smaller blocks present in the `c3/libraries` sub-module.

1.2 Parameter Map

The `ParameterMap` helps to obtain an optimizable vector of parameters from the various theoretical models previously defined. This allows for a simple interface to the optimization algorithms which are tasked with optimizing different sets of variables used to define some entity, e.g, optimizing pulse parameters by calibrating on hardware or providing an optimal gate-set through model-based quantum control.

1.3 Experiments

With the building blocks in place, we can bring them all together through an `Experiment` object that encapsulates the device model, the control signals, the instructions and the parameter map. Note that depending on the use only some of the blocks are essential when building the experiment.

1.4 Optimizers

At its core, `c3-toolset` is an optimization framework and all of the three steps - Open-Loop, Calibration and Model Learning can be defined as a optimization task. The `optimizers` contain classes that provide helpful encapsulation for these steps. These objects take as arguments the previously defined `Experiment` and `ParameterMap` objects along with an `algorithm` e.g, `CMA-eS` or `L-BFGS` which performs the iterative optimization steps.

1.5 Libraries

The `c3/libraries` sub-module includes various helpful library of components that are used somewhat like lego pieces when building the bigger blocks, e.g, `hamiltonians` for the `chip` present in the `Model` or `envelopes` defining a control `pulse`. More details about these components are available in the *Libraries package* section.

CHAPTER

TWO

PARAMETER HANDLING

The tool within C^3 to manipulate the parameters of both the model and controls is the `ParameterMap`. It provides methods to present the same data for human interaction, i.e. structured information with physical units and for numerical optimization algorithms that prefer a linear vector of scale 1. Here, we'll show some example usage. We'll use the `ParameterMap` of the model also used in the simulated calibration example.

```
from single_qubit_blackbox_exp import create_experiment

exp = create_experiment()
pmap = exp.pmap
```

The pmap contains a list of all parameters and their values:

```
pmap.get_full_params()
```

```
{'Q1-freq': 5.000 GHz 2pi,
 'Q1-anhar': -210.000 MHz 2pi,
 'Q1-temp': 0.000 K,
 'init_ground-init_temp': -3.469 aK,
 'resp-rise_time': 300.000 ps,
 'v_to_hz-V_to_Hz': 1.000 GHz/V,
 'id[0]-d1-no_drive-amp': 1.000 V,
 'id[0]-d1-no_drive-delta': 0.000 V,
 'id[0]-d1-no_drive-freq_offset': 0.000 Hz 2pi,
 'id[0]-d1-no_drive-xy_angle': 0.000 rad,
 'id[0]-d1-no_drive-sigma': 5.000 ns,
 'id[0]-d1-no_drive-t_final': 7.000 ns,
 'id[0]-d1-carrier-freq': 5.050 GHz 2pi,
 'id[0]-d1-carrier-framechange': 5.933 rad,
 'rx90p[0]-d1-gauss-amp': 450.000 mV,
 'rx90p[0]-d1-gauss-delta': -1.000 ,
 'rx90p[0]-d1-gauss-freq_offset': -50.500 MHz 2pi,
 'rx90p[0]-d1-gauss-xy_angle': -444.089 arad,
 'rx90p[0]-d1-gauss-sigma': 1.750 ns,
 'rx90p[0]-d1-gauss-t_final': 7.000 ns,
 'rx90p[0]-d1-carrier-freq': 5.050 GHz 2pi,
 'rx90p[0]-d1-carrier-framechange': 0.000 rad,
 'ry90p[0]-d1-gauss-amp': 450.000 mV,
 'ry90p[0]-d1-gauss-delta': -1.000 ,
 'ry90p[0]-d1-gauss-freq_offset': -50.500 MHz 2pi,
 'ry90p[0]-d1-gauss-xy_angle': 1.571 rad,
```

(continues on next page)

(continued from previous page)

```
 'ry90p[0]-d1-gauss-sigma': 1.750 ns,
 'ry90p[0]-d1-gauss-t_final': 7.000 ns,
 'ry90p[0]-d1-carrier-freq': 5.050 GHz 2pi,
 'ry90p[0]-d1-carrier-framechange': 0.000 rad,
 'rx90m[0]-d1-gauss-amp': 450.000 mV,
 'rx90m[0]-d1-gauss-delta': -1.000 ,
 'rx90m[0]-d1-gauss-freq_offset': -50.500 MHz 2pi,
 'rx90m[0]-d1-gauss-xy_angle': 3.142 rad,
 'rx90m[0]-d1-gauss-sigma': 1.750 ns,
 'rx90m[0]-d1-gauss-t_final': 7.000 ns,
 'rx90m[0]-d1-carrier-freq': 5.050 GHz 2pi,
 'rx90m[0]-d1-carrier-framechange': 0.000 rad,
 'ry90m[0]-d1-gauss-amp': 450.000 mV,
 'ry90m[0]-d1-gauss-delta': -1.000 ,
 'ry90m[0]-d1-gauss-freq_offset': -50.500 MHz 2pi,
 'ry90m[0]-d1-gauss-xy_angle': 4.712 rad,
 'ry90m[0]-d1-gauss-sigma': 1.750 ns,
 'ry90m[0]-d1-gauss-t_final': 7.000 ns,
 'ry90m[0]-d1-carrier-freq': 5.050 GHz 2pi,
 'ry90m[0]-d1-carrier-framechange': 0.000 rad}
```

To access a specific parameter, e.g. the frequency of qubit 1, we use the identifying tuple `('Q1','freq')`.

```
pmap.get_parameter(('Q1','freq'))
```

```
5.000 GHz 2pi
```

2.1 The opt_map

To deal with multiple parameters we use the `opt_map`, a nested list of identifyers.

```
opt_map = [
    [
        ("Q1", "freq")
    ],
    [
        ("Q1", "anhar")
    ],
]
```

Here, we get a list of the parameter values:

```
pmap.get_parameters(opt_map)
```

```
[5.000 GHz 2pi, -210.000 MHz 2pi]
```

Let's look at the amplitude values of two gaussian control pulses, rotations about the X and Y axes repsectively.

```
opt_map = [
    [
```

(continues on next page)

(continued from previous page)

```
        ('rx90p[0]','d1','gauss','amp')
    ],
    [
        ('ry90p[0]','d1','gauss','amp')
    ],
]
```

```
pmap.get_parameters(opt_map)
```

```
[450.000 mV, 450.000 mV]
```

We can set the parameters to new values.

```
pmap.set_parameters([0.5, 0.6], opt_map)
```

```
pmap.get_parameters(opt_map)
```

```
[500.000 mV, 600.000 mV]
```

The opt_map also allows us to specify that two parameters should have identical values. Here, let's demand our X and Y rotations use the same amplitude.

```
opt_map_ident = [
    [
        ('rx90p[0]','d1','gauss','amp'),
        ('ry90p[0]','d1','gauss','amp')
    ],
]
```

The grouping here means that these parameters share their numerical value.

```
pmap.set_parameters([0.432], opt_map_ident)
pmap.get_parameters(opt_map_ident)
```

```
[432.000 mV]
```

```
pmap.get_parameters(opt_map)
```

```
[432.000 mV, 432.000 mV]
```

During an optimization, the varied parameters do not change, so we fix the opt_map

```
pmap.set_opt_map(opt_map)
```

```
pmap.get_parameters()
```

```
[432.000 mV, 432.000 mV]
```

2.2 Optimizer scaling

To be independent of the choice of numerical optimizer, they should use the methods

```
pmap.get_parameters_scaled()
```

```
array([-0.68, -0.68])
```

To provide values bound to $[-1, 1]$. Let's set the parameters to their allowed minimum an maximum value with

```
pmap.set_parameters_scaled([1.0,-1.0])
```

```
pmap.get_parameters()
```

```
[600.000 mV, 400.000 mV]
```

As a safeguard, when setting values outside of the unit range, their physical values get looped back in the specified limits.

```
pmap.set_parameters_scaled([2.0, 3.0])
```

```
pmap.get_parameters()
```

```
[500.000 mV, 400.000 mV]
```

2.3 Storing and reading

For optimization purposes, we can store and load parameter values in HJSON format.

```
pmap.store_values("current_vals.c3log")
```

```
!cat current_vals.c3log
```

```
{
  opt_map:
  [
    [
      rx90p[0]-d1-gauss-amp
    ]
    [
      ry90p[0]-d1-gauss-amp
    ]
  ]
  units:
  [
    V
    V
  ]
  optim_status:
```

(continues on next page)

(continued from previous page)

```
  {
    params:
    [
      0.5
      0.4000000059604645
    ]
  }
}
```

```
pmap.load_values("current_vals.c3log")
```

CHAPTER

THREE

SETUP OF A TWO-QUBIT CHIP WITH C^3

In this example we will set-up a two qubit quantum processor and define a simple gate.

3.1 Imports

```
# System imports
import copy
import numpy as np
import time
import itertools
import matplotlib.pyplot as plt
import tensorflow as tf
import tensorflow_probability as tfp

# Main C3 objects
from c3.c3objs import Quantity as Qty
from c3.parametermap import ParameterMap as PMap
from c3.experiment import Experiment as Exp
from c3.model import Model as Mdl
from c3.generator.generator import Generator as Gnr

# Building blocks
import c3.generator.devices as devices
import c3.signal.gates as gates
import c3.libraries.chip as chip
import c3.signal.pulse as pulse
import c3.libraries.tasks as tasks

# Libs and helpers
import c3.libraries.algorithms as algorithms
import c3.libraries.hamiltonians as hamiltonians
import c3.libraries.fidelities as fidelities
import c3.libraries.envelopes as envelopes
import c3.utils.qt_utils as qt_utils
import c3.utils.tf_utils as tf_utils
```

3.2 Model components

We first create a qubit. Each parameter is a Quantity (`Qty()`) object with bounds and a unit. In C^3, the default multi-level qubit is a Transmon modelled as a Duffing oscillator with frequency ω and anharmonicity δ :

$$H/\hbar = \omega b^\dagger b - \frac{\delta}{2}\left(b^\dagger b - 1\right) b^\dagger b$$

The "name" will be used to identify this qubit (or other component) later and should thus be chosen carefully.

```
qubit_lvls = 3
freq_q1 = 5e9
anhar_q1 = -210e6
t1_q1 = 27e-6
t2star_q1 = 39e-6
qubit_temp = 50e-3

q1 = chip.Qubit(
    name="Q1",
    desc="Qubit 1",
    freq=Qty(
        value=freq_q1,
        min_val=4.995e9 ,
        max_val=5.005e9 ,
        unit='Hz 2pi'
    ),
    anhar=Qty(
        value=anhar_q1,
        min_val=-380e6 ,
        max_val=-120e6 ,
        unit='Hz 2pi'
    ),
    hilbert_dim=qubit_lvls,
    t1=Qty(
        value=t1_q1,
        min_val=1e-6,
        max_val=90e-6,
        unit='s'
    ),
    t2star=Qty(
        value=t2star_q1,
        min_val=10e-6,
        max_val=90e-3,
        unit='s'
    ),
    temp=Qty(
        value=qubit_temp,
        min_val=0.0,
        max_val=0.12,
        unit='K'
    )
)
```

And the same for a second qubit.

```
freq_q2 = 5.6e9
anhar_q2 = -240e6
t1_q2 = 23e-6
t2star_q2 = 31e-6
q2 = chip.Qubit(
    name="Q2",
    desc="Qubit 2",
    freq=Qty(
        value=freq_q2,
        min_val=5.595e9 ,
        max_val=5.605e9 ,
        unit='Hz 2pi'
    ),
    anhar=Qty(
        value=anhar_q2,
        min_val=-380e6 ,
        max_val=-120e6 ,
        unit='Hz 2pi'
    ),
    hilbert_dim=qubit_lvls,
    t1=Qty(
        value=t1_q2,
        min_val=1e-6,
        max_val=90e-6,
        unit='s'
    ),
    t2star=Qty(
        value=t2star_q2,
        min_val=10e-6,
        max_val=90e-6,
        unit='s'
    ),
    temp=Qty(
        value=qubit_temp,
        min_val=0.0,
        max_val=0.12,
        unit='K'
    )
)
```

A static coupling between the two is realized in the following way. We supply the type of coupling by selecting `int_XX` $(b_1+b_1^\dagger)(b_2+b_2^\dagger)$ from the hamiltonian library. The "connected" property contains the list of qubit names to be coupled, in this case "Q1" and "Q2".

```
coupling_strength = 20e6
q1q2 = chip.Coupling(
    name="Q1-Q2",
    desc="coupling",
    comment="Coupling qubit 1 to qubit 2",
    connected=["Q1", "Q2"],
    strength=Qty(
        value=coupling_strength,
        min_val=-1 * 1e3 ,
```

(continues on next page)

(continued from previous page)

```
        max_val=200e6 ,
        unit='Hz 2pi'
    ),
    hamiltonian_func=hamiltonians.int_XX
)
```

In the same spirit, we specify control Hamiltonians to drive the system. Again "connected" connected tells us which qubit this drive acts on and "name" will later be used to assign the correct control signal to this drive line.

```
drive = chip.Drive(
    name="d1",
    desc="Drive 1",
    comment="Drive line 1 on qubit 1",
    connected=["Q1"],
    hamiltonian_func=hamiltonians.x_drive
)
drive2 = chip.Drive(
    name="d2",
    desc="Drive 2",
    comment="Drive line 2 on qubit 2",
    connected=["Q2"],
    hamiltonian_func=hamiltonians.x_drive
)
```

3.3 SPAM errors

In experimental practice, the qubit state can be mis-classified during read-out. We simulate this by constructing a *confusion matrix*, containing the probabilities for one qubit state being mistaken for another.

```
m00_q1 = 0.97 # Prop to read qubit 1 state 0 as 0
m01_q1 = 0.04 # Prop to read qubit 1 state 0 as 1
m00_q2 = 0.96 # Prop to read qubit 2 state 0 as 0
m01_q2 = 0.05 # Prop to read qubit 2 state 0 as 1
one_zeros = np.array([0] * qubit_lvls)
zero_ones = np.array([1] * qubit_lvls)
one_zeros[0] = 1
zero_ones[0] = 0
val1 = one_zeros * m00_q1 + zero_ones * m01_q1
val2 = one_zeros * m00_q2 + zero_ones * m01_q2
min_val = one_zeros * 0.8 + zero_ones * 0.0
max_val = one_zeros * 1.0 + zero_ones * 0.2
confusion_row1 = Qty(value=val1, min_val=min_val, max_val=max_val, unit="")
confusion_row2 = Qty(value=val2, min_val=min_val, max_val=max_val, unit="")
conf_matrix = tasks.ConfusionMatrix(Q1=confusion_row1, Q2=confusion_row2)
```

The following task creates an initial thermal state with given temperature.

```
init_temp = 50e-3
init_ground = tasks.InitialiseGround(
    init_temp=Qty(
```

(continues on next page)

(continued from previous page)

```
        value=init_temp,
        min_val=-0.001,
        max_val=0.22,
        unit='K'
    )
)
```

We collect the parts specified above in the Model.

```
model = Mdl(
    [q1, q2], # Individual, self-contained components
    [drive, drive2, q1q2],  # Interactions between components
    [conf_matrix, init_ground] # SPAM processing
)
```

Further, we can decide between coherent or open-system dynamics using set_lindbladian() and whether to eliminate the static coupling by going to the dressed frame with set_dressed().

```
model.set_lindbladian(False)
model.set_dressed(True)
```

3.4 Control signals

With the system model taken care of, we now specify the control electronics and signal chain. Complex shaped controls are often realized by creating an envelope signal with an arbitrary waveform generator (AWG) with limited bandwith and mixing it with a fast, stable local oscillator (LO).

```
sim_res = 100e9 # Resolution for numerical simulation
awg_res = 2e9 # Realistic, limited resolution of an AWG
lo = devices.LO(name='lo', resolution=sim_res)
awg = devices.AWG(name='awg', resolution=awg_res)
mixer = devices.Mixer(name='mixer')
```

Waveform generators exhibit a rise time, the time it takes until the target voltage is set. This has a smoothing effect on the resulting pulse shape.

```
resp = devices.Response(
    name='resp',
    rise_time=Qty(
        value=0.3e-9,
        min_val=0.05e-9,
        max_val=0.6e-9,
        unit='s'
    ),
    resolution=sim_res
)
```

In simulation, we translate between AWG resolution and simulation (or “analog”) resolution by including an up-sampling device.

```
dig_to_an = devices.DigitalToAnalog(
    name="dac",
    resolution=sim_res
)
```

Control electronics apply voltages to lines, whereas in a Hamiltonian we usually write the control fields in energy or frequency units. In practice, this conversion can be highly non-trivial if it involves multiple stages of attenuation and for example the conversion of a line voltage in an antenna to a dipole field coupling to the qubit. The following device represents a simple, linear conversion factor.

```
v2hz = 1e9
v_to_hz = devices.VoltsToHertz(
    name='v_to_hz',
    V_to_Hz=Qty(
        value=v2hz,
        min_val=0.9e9,
        max_val=1.1e9,
        unit='Hz/V'
    )
)
```

The generator combines the parts of the signal generation and assigns a signal chain to each control line.

```
generator = Gnr(
        devices={
            "LO": devices.LO(name='lo', resolution=sim_res, outputs=1),
            "AWG": devices.AWG(name='awg', resolution=awg_res, outputs=1),
            "DigitalToAnalog": devices.DigitalToAnalog(
                name="dac",
                resolution=sim_res,
                inputs=1,
                outputs=1
            ),
            "Response": devices.Response(
                name='resp',
                rise_time=Qty(
                    value=0.3e-9,
                    min_val=0.05e-9,
                    max_val=0.6e-9,
                    unit='s'
                ),
                resolution=sim_res,
                inputs=1,
                outputs=1
            ),
            "Mixer": devices.Mixer(name='mixer', inputs=2, outputs=1),
            "VoltsToHertz": devices.VoltsToHertz(
                name='v_to_hz',
                V_to_Hz=Qty(
                    value=1e9,
                    min_val=0.9e9,
                    max_val=1.1e9,
                    unit='Hz/V'
```

(continues on next page)

(continued from previous page)

```
                ),
                inputs=1,
                outputs=1
            )
        },
        chains= {
            "d1": {
                "LO": [],
                "AWG": [],
                "DigitalToAnalog": ["AWG"],
                "Response": ["DigitalToAnalog"],
                "Mixer": ["LO", "Response"],
                "VoltsToHertz": ["Mixer"]
            },
            "d2": {
                "LO": [],
                "AWG": [],
                "DigitalToAnalog": ["AWG"],
                "Response": ["DigitalToAnalog"],
                "Mixer": ["LO", "Response"],
                "VoltsToHertz": ["Mixer"]
            }
        }
    )
```

3.5 Gates-set and Parameter map

It remains to write down what kind of operations we want to perform on the device. For a gate based quantum computing chip, we define a gate-set.

We choose a gate time of 7ns and a Gaussian envelope shape with a list of parameters.

```
t_final = 7e-9   # Time for single qubit gates
sideband = 50e6
gauss_params_single = {
    'amp': Qty(
        value=0.5,
        min_val=0.4,
        max_val=0.6,
        unit="V"
    ),
    't_final': Qty(
        value=t_final,
        min_val=0.5 * t_final,
        max_val=1.5 * t_final,
        unit="s"
    ),
    'sigma': Qty(
        value=t_final / 4,
        min_val=t_final / 8,
```

(continues on next page)

(continued from previous page)

```
        max_val=t_final / 2,
        unit="s"
    ),
    'xy_angle': Qty(
        value=0.0,
        min_val=-0.5 * np.pi,
        max_val=2.5 * np.pi,
        unit='rad'
    ),
    'freq_offset': Qty(
        value=-sideband - 3e6 ,
        min_val=-56 * 1e6 ,
        max_val=-52 * 1e6 ,
        unit='Hz 2pi'
    ),
    'delta': Qty(
        value=-1,
        min_val=-5,
        max_val=3,
        unit=""
    )
}
```

Here we take `gaussian_nonorm()` from the libraries as the function to define the shape.

```
gauss_env_single = pulse.Envelope(
    name="gauss",
    desc="Gaussian comp for single-qubit gates",
    params=gauss_params_single,
    shape=envelopes.gaussian_nonorm
)
```

We also define a gate that represents no driving.

```
nodrive_env = pulse.Envelope(
    name="no_drive",
    params={
        't_final': Qty(
            value=t_final,
            min_val=0.5 * t_final,
            max_val=1.5 * t_final,
            unit="s"
        )
    },
    shape=envelopes.no_drive
)
```

We specify the drive tones with an offset from the qubit frequencies. As is done in experiment, we will later adjust the resonance by modulating the envelope function.

```
lo_freq_q1 = 5e9  + sideband
carrier_parameters = {
    'freq': Qty(
```

(continues on next page)

(continued from previous page)

```
        value=lo_freq_q1,
        min_val=4.5e9 ,
        max_val=6e9 ,
        unit='Hz 2pi'
    ),
    'framechange': Qty(
        value=0.0,
        min_val= -np.pi,
        max_val= 3 * np.pi,
        unit='rad'
    )
}
carr = pulse.Carrier(
    name="carrier",
    desc="Frequency of the local oscillator",
    params=carrier_parameters
)
```

For the second qubit drive tone, we copy the first one and replace the frequency. The deepcopy is to ensure that we don't just create a pointer to the first drive.

```
lo_freq_q2 = 5.6e9  + sideband
carr_2 = copy.deepcopy(carr)
carr_2.params['freq'].set_value(lo_freq_q2)
```

3.6 Instructions

We define the gates we want to perform with a "name" that will identify them later and "channels" relating to the control Hamiltonians and drive lines we specified earlier. As a start we write down 90 degree rotations in the positive x-direction and identity gates for both qubits. Then we add a carrier and envelope to each.

```
rx90p_q1 = gates.Instruction(
    name="rx90p", targets=[0], t_start=0.0, t_end=t_final, channels=["d1", "d2"]
)
rx90p_q2 = gates.Instruction(
    name="rx90p", targets=[1], t_start=0.0, t_end=t_final, channels=["d1", "d2"]
)

rx90p_q1.add_component(gauss_env_single, "d1")
rx90p_q1.add_component(carr, "d1")


rx90p_q2.add_component(copy.deepcopy(gauss_env_single), "d2")
rx90p_q2.add_component(carr_2, "d2")
```

When later compiling gates into sequences, we have to take care of the relative rotating frames of the qubits and local oscillators. We do this by adding a phase after each gate that realigns the frames.

```
rx90p_q1.add_component(nodrive_env, "d2")
rx90p_q1.add_component(copy.deepcopy(carr_2), "d2")
```

(continues on next page)

(continued from previous page)

```
rx90p_q1.comps["d2"]["carrier"].params["framechange"].set_value(
    (-sideband * t_final) * 2 * np.pi % (2 * np.pi)
)

rx90p_q2.add_component(nodrive_env, "d1")
rx90p_q2.add_component(copy.deepcopy(carr), "d1")
rx90p_q2.comps["d1"]["carrier"].params["framechange"].set_value(
    (-sideband * t_final) * 2 * np.pi % (2 * np.pi)
)
```

The remainder of the gates-set can be derived from the RX90p gate by shifting its phase by multiples of $\pi/2$.

```
ry90p_q1 = copy.deepcopy(rx90p_q1)
ry90p_q1.name = "ry90p"
rx90m_q1 = copy.deepcopy(rx90p_q1)
rx90m_q1.name = "rx90m"
ry90m_q1 = copy.deepcopy(rx90p_q1)
ry90m_q1.name = "ry90m"
ry90p_q1.comps['d1']['gauss'].params['xy_angle'].set_value(0.5 * np.pi)
rx90m_q1.comps['d1']['gauss'].params['xy_angle'].set_value(np.pi)
ry90m_q1.comps['d1']['gauss'].params['xy_angle'].set_value(1.5 * np.pi)
single_q_gates = [rx90p_q1, ry90p_q1, rx90m_q1, ry90m_q1]


ry90p_q2 = copy.deepcopy(rx90p_q2)
ry90p_q2.name = "ry90p"
rx90m_q2 = copy.deepcopy(rx90p_q2)
rx90m_q2.name = "rx90m"
ry90m_q2 = copy.deepcopy(rx90p_q2)
ry90m_q2.name = "ry90m"
ry90p_q2.comps['d2']['gauss'].params['xy_angle'].set_value(0.5 * np.pi)
rx90m_q2.comps['d2']['gauss'].params['xy_angle'].set_value(np.pi)
ry90m_q2.comps['d2']['gauss'].params['xy_angle'].set_value(1.5 * np.pi)
single_q_gates.extend([rx90p_q2, ry90p_q2, rx90m_q2, ry90m_q2])
```

With every component defined, we collect them in the parameter map, our object that holds information and methods to manipulate and examine model and control parameters.

```
parameter_map = PMap(instructions=all_1q_gates_comb, model=model, generator=generator)
```

3.7 The experiment

Finally everything is collected in the experiment that provides the functions to interact with the system.

```
exp = Exp(pmap=parameter_map)
```

3.8 Simulation

With our experiment all set-up, we can perform simulations. We first decide which basic gates to simulate, in this case only the 90 degree rotation on one qubit and the identity.

```
exp.set_opt_gates(['RX90p:Id', 'Id:Id'])
```

The actual numerical simulation is done by calling `exp.compute_propagators()`. This is the most resource intensive part as it involves solving the equations of motion for the system.

```
unitaries = exp.compute_propagators()
```

```

```

CHAPTER

FOUR

DYNAMICS

To investigate dynamics, we define the ground state as an initial state.

```
psi_init = [[0] * 9]
psi_init[0][0] = 1
init_state = tf.transpose(tf.constant(psi_init, tf.complex128))
```

```
init_state
```

```
<tf.Tensor: shape=(9, 1), dtype=complex128, numpy=
array([[1.+0.j],
       [0.+0.j],
       [0.+0.j],
       [0.+0.j],
       [0.+0.j],
       [0.+0.j],
       [0.+0.j],
       [0.+0.j],
       [0.+0.j]])>
```

Since we stored the process matrices, we can now relatively inexpensively evaluate sequences. We start with just one gate

```
barely_a_seq = ['rx90p[0]']
```

and plot system dynamics.

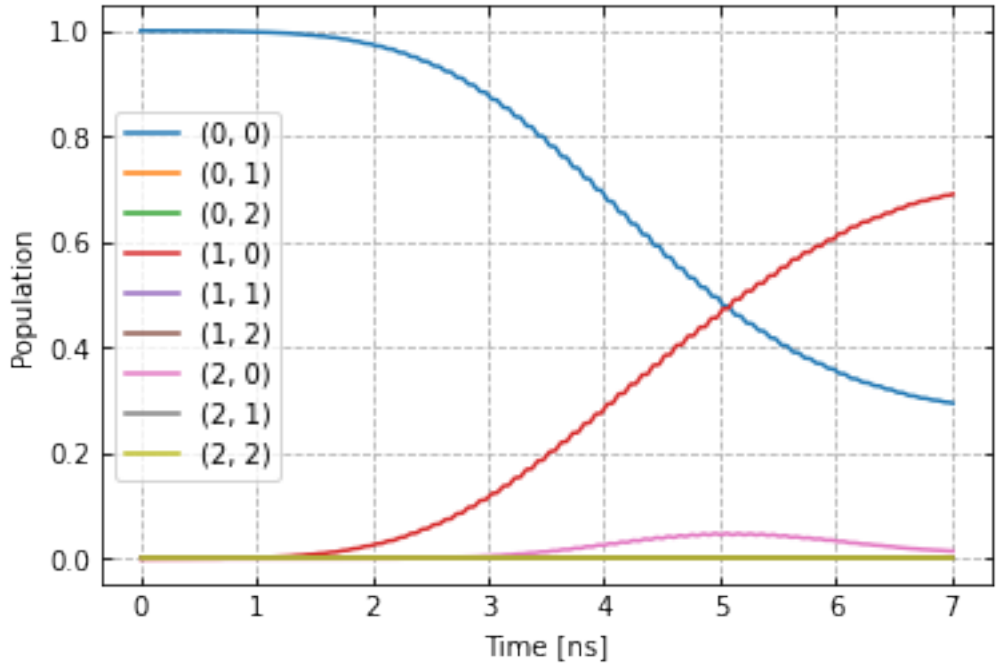
```
def plot_dynamics(exp, psi_init, seq, goal=-1):
        """
        Plotting code for time-resolved populations.

        Parameters
        ----------
        psi_init: tf.Tensor
            Initial state or density matrix.
        seq: list
            List of operations to apply to the initial state.
        goal: tf.float64
            Value of the goal function, if used.
        debug: boolean
            If true, return a matplotlib figure instead of saving.
```

(continues on next page)

(continued from previous page)

```
        """
        model = exp.pmap.model
        dUs = exp.partial_propagators
        psi_t = psi_init.numpy()
        pop_t = exp.populations(psi_t, model.lindbladian)
        for gate in seq:
            for du in dUs[gate]:
                psi_t = np.matmul(du.numpy(), psi_t)
                pops = exp.populations(psi_t, model.lindbladian)
                pop_t = np.append(pop_t, pops, axis=1)

        fig, axs = plt.subplots(1, 1)
        ts = exp.ts
        dt = ts[1] - ts[0]
        ts = np.linspace(0.0, dt*pop_t.shape[1], pop_t.shape[1])
        axs.plot(ts / 1e-9, pop_t.T)
        axs.grid(linestyle="--")
        axs.tick_params(
            direction="in", left=True, right=True, top=True, bottom=True
        )
        axs.set_xlabel('Time [ns]')
        axs.set_ylabel('Population')
        plt.legend(model.state_labels)
        pass
```

```
plot_dynamics(exp, init_state, barely_a_seq)
```

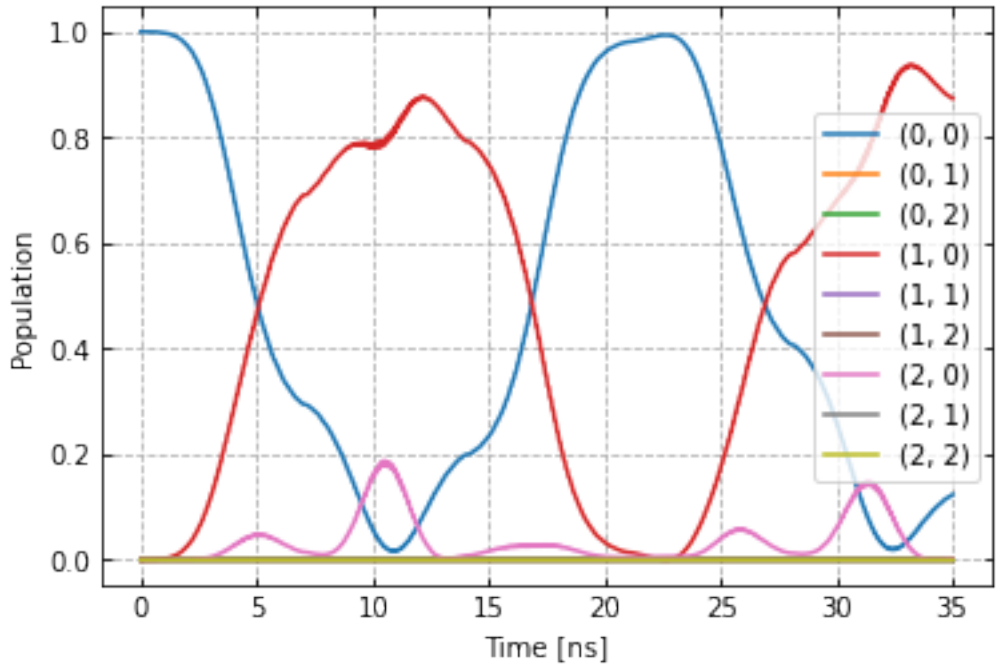


We can see an ill-defined un-optimized gate. The labels indicate qubit states in the product basis. Next we increase the number of repetitions of the same gate.

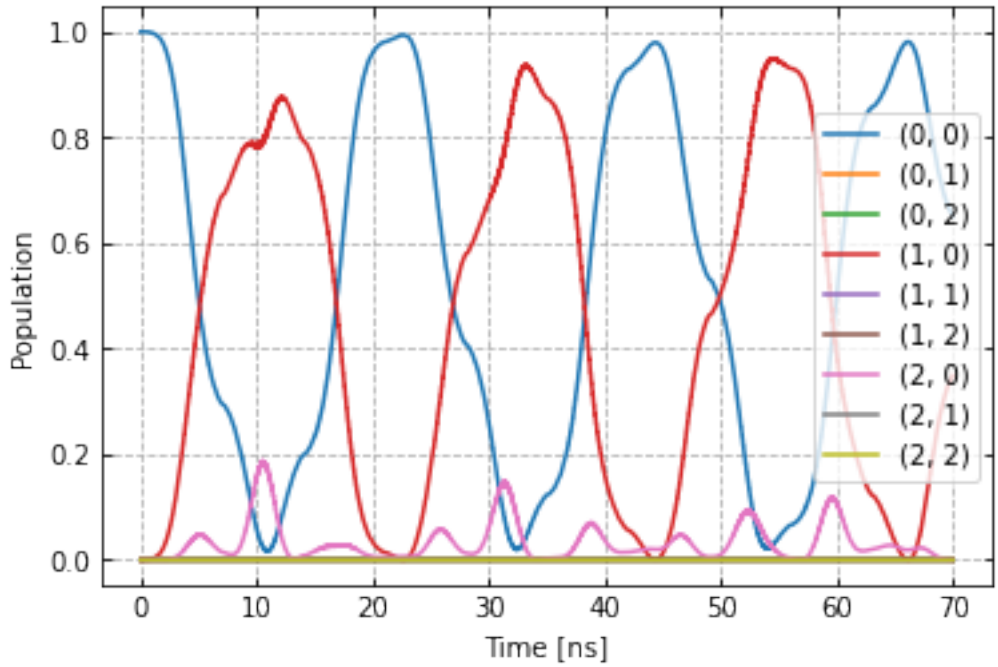
```
barely_a_seq * 10
```

```
['rx90p[0]',
 'rx90p[0]',
 'rx90p[0]',
 'rx90p[0]',
 'rx90p[0]',
 'rx90p[0]',
 'rx90p[0]',
 'rx90p[0]',
 'rx90p[0]',
 'rx90p[0]']
```

```
plot_dynamics(exp, init_state, barely_a_seq * 5)
```




```
plot_dynamics(exp, init_state, barely_a_seq * 10)
```



Note that at this point, we only multiply already computed matrices. We don't need to solve the equations of motion again for new sequences.

CHAPTER

FIVE

OPEN-LOOP OPTIMAL CONTROL

In order to improve the gate from the previous example *Setup of a two-qubit chip with C^3*, we create the optimizer object for open-loop optimal control. Examining the previous dynamics .. image:: dyn_singleX.png

in addition to over-rotation, we notice some leakage into the $|2, 0>$ state and enable a DRAG option. Details on DRAG can be found here. The main principle is adding a phase-shifted component proportional to the derivative of the original signal. With automatic differentiation, our AWG can perform this operation automatically for arbitrary shapes.

```
generator.devices['AWG'].enable_drag_2()
```

At the moment there are two implementations of DRAG, variant 2 is independent of the AWG resolution.

To define which parameters we optimize, we write the gateset_opt_map, a nested list of tuples that identifies each parameter.

```
opt_gates = ["rx90p[0]"]
gateset_opt_map=[
    [
      ("rx90p[0]", "d1", "gauss", "amp"),
    ],
    [
      ("rx90p[0]", "d1", "gauss", "freq_offset"),
    ],
    [
      ("rx90p[0]", "d1", "gauss", "xy_angle"),
    ],
    [
      ("RX90p:Id", "d1", "gauss", "delta"),
    ]
]
parameter_map.set_opt_map(gateset_opt_map)
```

We can look at the parameter values this opt_map specified with

```
parameter_map.print_parameters()
```

```
rx90p[0]-d1-gauss-amp                : 500.000 mV
rx90p[0]-d1-gauss-freq_offset        : -53.000 MHz 2pi
rx90p[0]-d1-gauss-xy_angle           : -444.089 arad
rx90p[0]-d1-gauss-delta              : -1.000
```

```
from c3.optimizers.optimalcontrol import OptimalControl
import c3.libraries.algorithms as algorithms
```

The OptimalControl object will handle the optimization for us. As a fidelity function we choose average fidelity as well as LBFG-S (a wrapper of the scipy implementation) from our library. See those libraries for how these functions are defined and how to supply your own, if necessary.

```
import os
import tempfile

# Create a temporary directory to store logfiles, modify as needed
log_dir = os.path.join(tempfile.TemporaryDirectory().name, "c3logs")

opt = OptimalControl(
    dir_path=log_dir,
    fid_func=fidelities.average_infid_set,
    fid_subspace=["Q1", "Q2"],
    pmap=parameter_map,
    algorithm=algorithms.lbfgs,
    options={"maxfun" : 10},
    run_name="better_X90"
)
```

Finally we supply our defined experiment.

```
exp.set_opt_gates(opt_gates)
opt.set_exp(exp)
```

Everything is in place to start the optimization.

```
opt.optimize_controls()
```

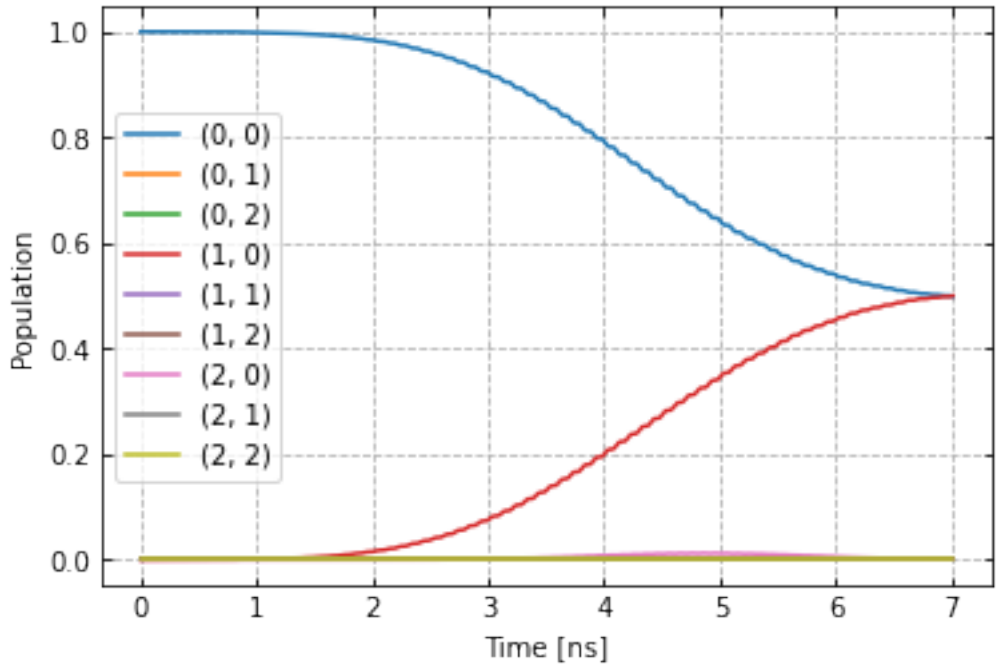
After a few steps we have improved the gate significantly, as we can check with

```
opt.current_best_goal
```

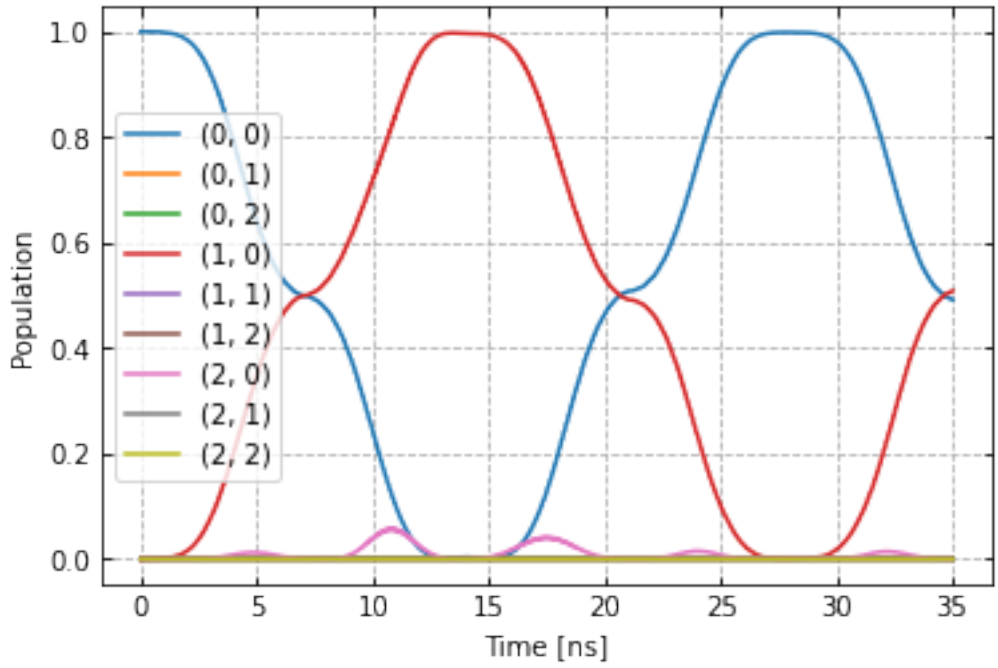
```
0.00063
```

And by looking at the same sequences as before.

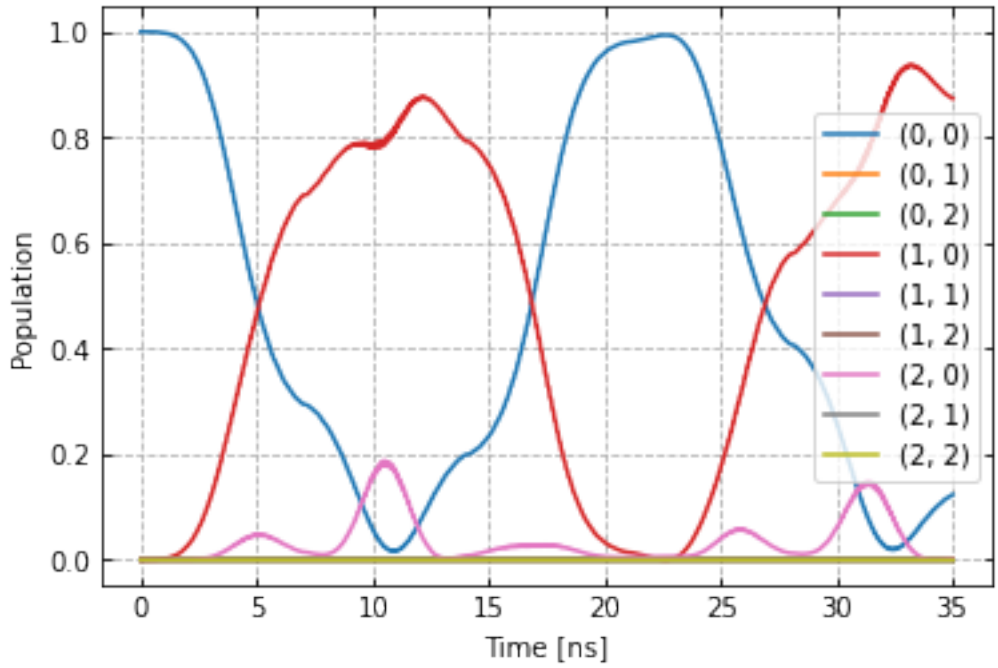
```
plot_dynamics(exp, init_state, barely_a_seq)
```




```
plot_dynamics(exp, init_state, barely_a_seq * 5)
```



Compared to before the optimization.


1.0
0.8
0.6
0.4
0.2
0.0
Population
0
5
10
15
20
25
30
35
Time [ns]
(0, 0)
(0, 1)
(0, 2)
(1, 0)
(1, 1)
(1, 2)
(2, 0)
(2, 1)
(2, 2)

CHAPTER

SIX

ENTANGLING GATE ON TWO COUPLED QUBITS

6.1 Imports

```
!pip install -q -U pip
!pip install -q matplotlib
```

```
# System imports
import copy
import numpy as np
import time
import itertools
import matplotlib.pyplot as plt
import tensorflow as tf
import tensorflow_probability as tfp
from typing import List

# Main C3 objects
from c3.c3objs import Quantity as Qty
from c3.parametermap import ParameterMap as PMap
from c3.experiment import Experiment as Exp
from c3.model import Model as Mdl
from c3.generator.generator import Generator as Gnr

# Building blocks
import c3.generator.devices as devices
import c3.signal.gates as gates
import c3.libraries.chip as chip
import c3.signal.pulse as pulse
import c3.libraries.tasks as tasks

# Libs and helpers
import c3.libraries.algorithms as algorithms
import c3.libraries.hamiltonians as hamiltonians
import c3.libraries.fidelities as fidelities
import c3.libraries.envelopes as envelopes
import c3.utils.qt_utils as qt_utils
import c3.utils.tf_utils as tf_utils
```

6.2 Model components

The model consists of two qubits with 3 levels each and slightly different parameters:

```
qubit_lvls = 3
freq_q1 = 5e9
anhar_q1 = -210e6
t1_q1 = 27e-6
t2star_q1 = 39e-6
qubit_temp = 50e-3

q1 = chip.Qubit(
    name="Q1",
    desc="Qubit 1",
    freq=Qty(value=freq_q1, min_val=4.995e9, max_val=5.005e9, unit='Hz 2pi'),
    anhar=Qty(value=anhar_q1, min_val=-380e6, max_val=-120e6, unit='Hz 2pi'),
    hilbert_dim=qubit_lvls,
    t1=Qty(value=t1_q1, min_val=1e-6, max_val=90e-6, unit='s'),
    t2star=Qty(value=t2star_q1, min_val=10e-6, max_val=90e-3, unit='s'),
    temp=Qty(value=qubit_temp, min_val=0.0, max_val=0.12, unit='K')
)

freq_q2 = 5.6e9
anhar_q2 = -240e6
t1_q2 = 23e-6
t2star_q2 = 31e-6
q2 = chip.Qubit(
    name="Q2",
    desc="Qubit 2",
    freq=Qty(value=freq_q2, min_val=5.595e9, max_val=5.605e9, unit='Hz 2pi'),
    anhar=Qty(value=anhar_q2, min_val=-380e6, max_val=-120e6, unit='Hz 2pi'),
    hilbert_dim=qubit_lvls,
    t1=Qty(value=t1_q2, min_val=1e-6, max_val=90e-6,unit='s'),
    t2star=Qty(value=t2star_q2, min_val=10e-6, max_val=90e-6, unit='s'),
    temp=Qty(value=qubit_temp, min_val=0.0, max_val=0.12, unit='K')
)
```

There is a static coupling in x-direction between them: $(b_1 + b_1^\dagger)(b_2 + b_2^\dagger)$

```
coupling_strength = 50e6
q1q2 = chip.Coupling(
    name="Q1-Q2",
    desc="coupling",
    comment="Coupling qubit 1 to qubit 2",
    connected=["Q1", "Q2"],
    strength=Qty(
        value=coupling_strength,
        min_val=-1 * 1e3 ,
        max_val=200e6 ,
        unit='Hz 2pi'
    ),
    hamiltonian_func=hamiltonians.int_XX
)
```

and each qubit has a drive line

```
drive1 = chip.Drive(
    name="d1",
    desc="Drive 1",
    comment="Drive line 1 on qubit 1",
    connected=["Q1"],
    hamiltonian_func=hamiltonians.x_drive
)
drive2 = chip.Drive(
    name="d2",
    desc="Drive 2",
    comment="Drive line 2 on qubit 2",
    connected=["Q2"],
    hamiltonian_func=hamiltonians.x_drive
)
```

All parts are collected in the model. The initial state will be thermal at a non-vanishing temperature.

```
init_temp = 50e-3
init_ground = tasks.InitialiseGround(
    init_temp=Qty(value=init_temp, min_val=-0.001, max_val=0.22, unit='K')
)

model = Mdl(
    [q1, q2], # Individual, self-contained components
    [drive1, drive2, q1q2],  # Interactions between components
    [init_ground] # SPAM processing
)
model.set_lindbladian(False)
model.set_dressed(True)
```

6.3 Control signals

The devices for the control line are set up

```
sim_res = 100e9 # Resolution for numerical simulation
awg_res = 2e9 # Realistic, limited resolution of an AWG
v2hz = 1e9

lo = devices.LO(name='lo', resolution=sim_res)
awg = devices.AWG(name='awg', resolution=awg_res)
mixer = devices.Mixer(name='mixer')
resp = devices.Response(
    name='resp',
    rise_time=Qty(value=0.3e-9, min_val=0.05e-9, max_val=0.6e-9, unit='s'),
    resolution=sim_res
)
dig_to_an = devices.DigitalToAnalog(name="dac", resolution=sim_res)
v_to_hz = devices.VoltsToHertz(
    name='v_to_hz',
```

(continues on next page)

(continued from previous page)

```
    V_to_Hz=Qty(value=v2hz, min_val=0.9e9, max_val=1.1e9, unit='Hz/V')
)
```

The generator combines the parts of the signal generation and assignes a signal chain to each control line.

```
generator = Gnr(
        devices={
            "LO": lo,
            "AWG": awg,
            "DigitalToAnalog": dig_to_an,
            "Response": resp,
            "Mixer": mixer,
            "VoltsToHertz": v_to_hz
        },
        chains={
            "d1": ["LO", "AWG", "DigitalToAnalog", "Response", "Mixer", "VoltsToHertz"],
            "d2": ["LO", "AWG", "DigitalToAnalog", "Response", "Mixer", "VoltsToHertz"]
        }
    )
```

6.4 Gates-set and Parameter map

Following a general cross resonance scheme, both qubits will be resonantly driven at the frequency of qubit 2 with a Gaussian envelope. We drive qubit 1 (the control) at the frequency of qubit 2 (the target) with a higher amplitude to compensate for the reduced Rabi frequency.

```
t_final = 45e-9
sideband = 50e6
gauss_params_single_1 = {
    'amp': Qty(value=0.8, min_val=0.2, max_val=3, unit="V"),
    't_final': Qty(value=t_final, min_val=0.5 * t_final, max_val=1.5 * t_final, unit="s
↪"),
    'sigma': Qty(value=t_final / 4, min_val=t_final / 8, max_val=t_final / 2, unit="s"),
    'xy_angle': Qty(value=0.0, min_val=-0.5 * np.pi, max_val=2.5 * np.pi, unit='rad'),
    'freq_offset': Qty(value=-sideband - 3e6, min_val=-56 * 1e6, max_val=-52 * 1e6, unit=
↪'Hz 2pi'),
    'delta': Qty(value=-1, min_val=-5, max_val=3, unit="")
}

gauss_params_single_2 = {
    'amp': Qty(value=0.03, min_val=0.02, max_val=0.6, unit="V"),
    't_final': Qty(value=t_final, min_val=0.5 * t_final, max_val=1.5 * t_final, unit="s
↪"),
    'sigma': Qty(value=t_final / 4, min_val=t_final / 8, max_val=t_final / 2, unit="s"),
    'xy_angle': Qty(value=0.0, min_val=-0.5 * np.pi, max_val=2.5 * np.pi, unit='rad'),
    'freq_offset': Qty(value=-sideband - 3e6, min_val=-56 * 1e6, max_val=-52 * 1e6, unit=
↪'Hz 2pi'),
    'delta': Qty(value=-1, min_val=-5, max_val=3, unit="")
}
```

(continues on next page)

(continued from previous page)

```
gauss_env_single_1 = pulse.Envelope(
    name="gauss1",
    desc="Gaussian envelope on drive 1",
    params=gauss_params_single_1,
    shape=envelopes.gaussian_nonorm
)
gauss_env_single_2 = pulse.Envelope(
    name="gauss2",
    desc="Gaussian envelope on drive 2",
    params=gauss_params_single_2,
    shape=envelopes.gaussian_nonorm
)
```

The carrier signal of each drive is set to the resonance frequency of the target qubit.

```
lo_freq_q1 = freq_q1 + sideband
lo_freq_q2 = freq_q2 + sideband

carr_1 = pulse.Carrier(
    name="carrier",
    desc="Carrier on drive 1",
    params={
        'freq': Qty(value=lo_freq_q2, min_val=0.9 * lo_freq_q2, max_val=1.1 * lo_freq_q2,
↪ unit='Hz 2pi'),
        'framechange': Qty(value=0.0, min_val=-np.pi, max_val=3 * np.pi, unit='rad')
    }
)

carr_2 = pulse.Carrier(
    name="carrier",
    desc="Carrier on drive 2",
    params={
        'freq': Qty(value=lo_freq_q2, min_val=0.9 * lo_freq_q2, max_val=1.1 * lo_freq_q2,
↪ unit='Hz 2pi'),
        'framechange': Qty(value=0.0, min_val=-np.pi, max_val=3 * np.pi, unit='rad')
    }
)
```

6.5 Instructions

The instruction to be optimised is a CNOT gates controlled by qubit 1.

```
# CNOT comtrolled by qubit 1
cnot12 = gates.Instruction(
    name="cnot12", targets=[0, 1], t_start=0.0, t_end=t_final, channels=["d1", "d2"],
    ideal=np.array([
        [1,0,0,0],
        [0,1,0,0],
        [0,0,0,1],
        [0,0,1,0]
```

(continues on next page)

(continued from previous page)

```
    ])
)
cnot12.add_component(gauss_env_single_1, "d1")
cnot12.add_component(carr_1, "d1")
cnot12.add_component(gauss_env_single_2, "d2")
cnot12.add_component(carr_2, "d2")
cnot12.comps["d1"]["carrier"].params["framechange"].set_value(
    (-sideband * t_final) * 2 * np.pi % (2 * np.pi)
)
```

6.6 The experiment

All components are collected in the parameter map and the experiment is set up.

```
parameter_map = PMap(instructions=[cnot12], model=model, generator=generator)
exp = Exp(pmap=parameter_map)
```

Calculate and print the propagator before the optimisation.

```
unitaries = exp.compute_propagators()
print(unitaries[cnot12.get_key()])
```

```
tf.Tensor(
[[ 5.38699071e-01-7.17750563e-02j -8.34752005e-01+8.73275022e-02j
  -6.95346256e-03-2.15875540e-03j -4.35619589e-03+3.35449682e-03j
  -1.06942994e-02+4.11831376e-03j -6.46672021e-05-3.73989900e-05j
  -1.67838080e-04-2.08026492e-04j -6.43312053e-05-7.70584828e-07j
  -3.76227149e-07-6.49845314e-07j]
 [-8.22954017e-01+1.64865789e-01j -5.35373070e-01+9.17248769e-02j
  -7.01716357e-03+7.68563193e-03j -1.04194796e-02+4.75452421e-03j
  -1.61239175e-02-5.34774092e-03j -2.42060738e-04-1.19946128e-05j
   3.81855912e-05+8.66289943e-06j -1.30621879e-04-2.10380577e-04j
  -8.82654253e-07-1.33276919e-06j]
 [-7.61570279e-03+7.68089055e-04j -4.61417534e-03+9.02462832e-03j
   3.59132066e-01-9.32828470e-01j -9.10153028e-05-6.83262609e-05j
  -2.24711912e-04+8.79671466e-05j  2.62921224e-02-1.48696337e-03j
  -4.75883791e-04-4.20508543e-05j  3.46114778e-05+1.64470496e-04j
   2.10121296e-04+1.48066297e-04j]
 [ 4.65531318e-03-6.63491197e-05j  8.62792565e-03+8.22022317e-03j
  -5.58701973e-05+1.08666061e-04j  6.94902895e-02-7.11528641e-01j
  -6.81737268e-01-1.53183314e-01j -2.09824678e-03-1.43761730e-03j
   1.48197730e-02-1.51149441e-02j -6.85074400e-03+1.43594091e-03j
   4.07440635e-05-6.43168354e-05j]
 [ 9.49155432e-03+6.86731461e-03j  4.92068252e-03+1.60041286e-02j
   1.71300460e-04+1.83910737e-04j -6.94165643e-01-7.98008223e-02j
   1.68675369e-01-6.94722446e-01j  2.75768137e-03-5.72343874e-03j
  -6.67593164e-03+1.87532770e-03j  1.07707017e-02+7.28665794e-03j
   1.40030301e-04-6.25646793e-05j]
 [ 3.43460967e-05+8.01438338e-05j  1.86345824e-04+1.52916372e-04j
  -1.74936595e-02-1.96833938e-02j -2.61695107e-03-5.33671505e-04j
```

(continues on next page)

(continued from previous page)

```
   1.02116861e-03-6.21800378e-03j -4.07849502e-01+9.12571012e-01j
   7.51460471e-05-1.15167196e-04j  2.32056836e-04-2.97650209e-04j
   2.03278960e-04+1.15047574e-02j]
 [ 2.54853797e-04-1.25904275e-04j  6.64845849e-05-1.08876861e-05j
   2.38628329e-04-2.95318799e-04j -2.10696691e-02+5.90348860e-05j
   4.21445291e-03+6.01993253e-03j -1.32690530e-04-2.44975772e-05j
   5.90859776e-01+4.84056180e-01j -6.08336007e-01-2.14442516e-01j
   3.13146026e-03+2.83895304e-03j]
 [ 2.96366741e-05-8.10052801e-05j  2.39607442e-04-8.47647458e-05j
  -2.60360838e-04+2.04175607e-04j  4.95127881e-03+5.19423708e-03j
  -5.00047077e-03-1.18242204e-02j -3.71631612e-04-5.78977628e-05j
  -6.29480118e-01-1.40758384e-01j -7.57820104e-01+9.68476237e-02j
   1.32060361e-03+7.25998662e-03j]
 [ 8.28054635e-07-3.59336781e-07j  1.64602058e-06-1.47364829e-06j
  -2.13361477e-04+2.05358711e-04j -5.70978380e-05+4.73283539e-05j
  -1.48466829e-04-3.89352221e-06j  1.00811226e-02-5.54615336e-03j
   4.21887172e-03+1.38103179e-03j  3.74182763e-03+6.21303072e-03j
  -5.89257172e-01+8.07818774e-01j]], shape=(9, 9), dtype=complex128)
```

6.7 Dynamics

The system is initialised in the state $|0,1\rangle$ so that a transition to $|1,1\rangle$ should be visible.

```
psi_init = [[0] * 9]
psi_init[0][0] = 1
init_state = tf.transpose(tf.constant(psi_init, tf.complex128))
print(init_state)
```

```
tf.Tensor(
[[1.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]], shape=(9, 1), dtype=complex128)
```

```
def plot_dynamics(exp, psi_init, seq):
        """
        Plotting code for time-resolved populations.

        Parameters
        ----------
        psi_init: tf.Tensor
            Initial state or density matrix.
        seq: list
            List of operations to apply to the initial state.
```

(continues on next page)

(continued from previous page)

```
        """
        model = exp.pmap.model
        dUs = exp.partial_propagators
        psi_t = psi_init.numpy()
        pop_t = exp.populations(psi_t, model.lindbladian)
        for gate in seq:
            for du in dUs[gate]:
                psi_t = np.matmul(du.numpy(), psi_t)
                pops = exp.populations(psi_t, model.lindbladian)
                pop_t = np.append(pop_t, pops, axis=1)

        fig, axs = plt.subplots(1, 1)
        ts = exp.ts
        dt = ts[1] - ts[0]
        ts = np.linspace(0.0, dt*pop_t.shape[1], pop_t.shape[1])
        axs.plot(ts / 1e-9, pop_t.T)
        axs.grid(linestyle="--")
        axs.tick_params(
            direction="in", left=True, right=True, top=True, bottom=True
        )
        axs.set_xlabel('Time [ns]')
        axs.set_ylabel('Population')
        plt.legend(model.state_labels)
        pass

def getQubitsPopulation(population: np.array, dims: List[int]) -> np.array:
    """
    Splits the population of all levels of a system into the populations of levels per␣
↪subsystem.
    Parameters
    ----------
    population: np.array
        The time dependent population of each energy level. First dimension: level index,
↪ second dimension: time.
    dims: List[int]
        The number of levels for each subsystem.
    Returns
    -------
    np.array
        The time-dependent population of energy levels for each subsystem. First␣
↪dimension: subsystem index, second
        dimension: level index, third dimension: time.
    """
    numQubits = len(dims)

    # create a list of all levels
    qubit_levels = []
    for dim in dims:
        qubit_levels.append(list(range(dim)))
    combined_levels = list(itertools.product(*qubit_levels))

    # calculate populations
```

(continues on next page)

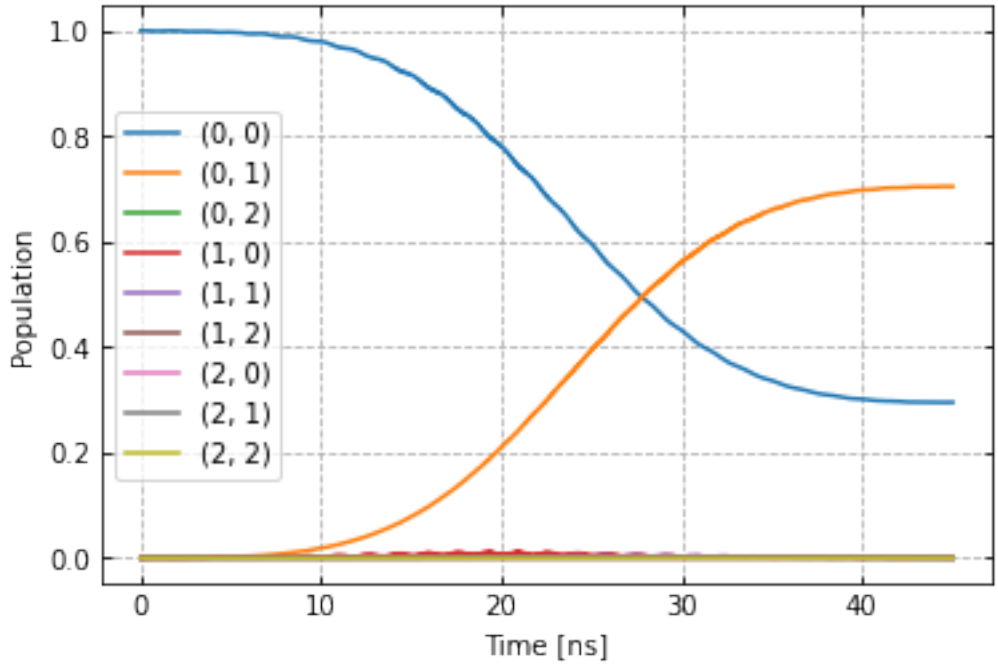
(continued from previous page)

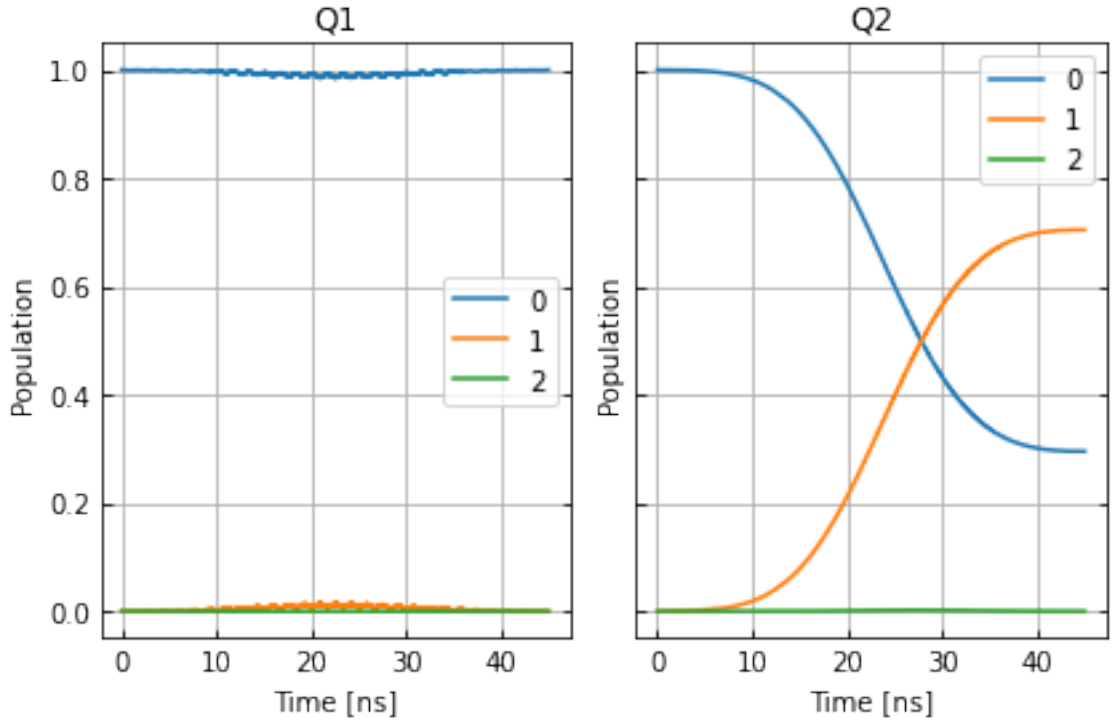
```
    qubitsPopulations = np.zeros((numQubits, dims[0], population.shape[1]))
    for idx, levels in enumerate(combined_levels):
        for i in range(numQubits):
            qubitsPopulations[i, levels[i]] += population[idx]
    return qubitsPopulations

def plotSplittedPopulation(
    exp: Exp,
    psi_init: tf.Tensor,
    sequence: List[str]
) -> None:
    """
    Plots time dependent populations for multiple qubits in separate plots.
    Parameters
    ----------
    exp: Experiment
        The experiment containing the model and propagators
    psi_init: np.array
        Initial state vector
    sequence: List[str]
        List of gate names that will be applied to the state
    -------
    """
    # calculate the time dependent level population
    model = exp.pmap.model
    dUs = exp.partial_propagators
    psi_t = psi_init.numpy()
    pop_t = exp.populations(psi_t, model.lindbladian)
    for gate in sequence:
        for du in dUs[gate]:
            psi_t = np.matmul(du, psi_t)
            pops = exp.populations(psi_t, model.lindbladian)
            pop_t = np.append(pop_t, pops, axis=1)
    dims = [s.hilbert_dim for s in model.subsystems.values()]
    splitted = getQubitsPopulation(pop_t, dims)

    # timestamps
    dt = exp.ts[1] - exp.ts[0]
    ts = np.linspace(0.0, dt * pop_t.shape[1], pop_t.shape[1])

    # create both subplots
    titles = list(exp.pmap.model.subsystems.keys())
    fig, axs = plt.subplots(1, len(splitted), sharey="all")
    for idx, ax in enumerate(axs):
        ax.plot(ts / 1e-9, splitted[idx].T)
        ax.tick_params(direction="in", left=True, right=True, top=False, bottom=True)
        ax.set_xlabel("Time [ns]")
        ax.set_ylabel("Population")
        ax.set_title(titles[idx])
        ax.legend([str(x) for x in np.arange(dims[idx])])
        ax.grid()
```

(continues on next page)

(continued from previous page)

```
    plt.tight_layout()
    plt.show()

sequence = [cnot12.get_key()]
plot_dynamics(exp, init_state, sequence)
plotSplittedPopulation(exp, init_state, sequence)
```





6.8 Open-loop optimal control

Now, open-loop optimisation with DRAG enabled is set up.

```
generator.devices['AWG'].enable_drag_2()

opt_gates = [cnot12.get_key()]
exp.set_opt_gates(opt_gates)

gateset_opt_map=[
    [(cnot12.get_key(), "d1", "gauss1", "amp")],
    [(cnot12.get_key(), "d1", "gauss1", "freq_offset")],
    [(cnot12.get_key(), "d1", "gauss1", "xy_angle")],
    [(cnot12.get_key(), "d1", "gauss1", "delta")],
    [(cnot12.get_key(), "d1", "carrier", "framechange")],
    [(cnot12.get_key(), "d2", "gauss2", "amp")],
    [(cnot12.get_key(), "d2", "gauss2", "freq_offset")],
    [(cnot12.get_key(), "d2", "gauss2", "xy_angle")],
    [(cnot12.get_key(), "d2", "gauss2", "delta")],
    [(cnot12.get_key(), "d2", "carrier", "framechange")]
]
parameter_map.set_opt_map(gateset_opt_map)

parameter_map.print_parameters()
```

```
cnot12[0, 1]-d1-gauss1-amp            : 800.000 mV
cnot12[0, 1]-d1-gauss1-freq_offset    : -53.000 MHz 2pi
cnot12[0, 1]-d1-gauss1-xy_angle       : -444.089 arad
cnot12[0, 1]-d1-gauss1-delta          : -1.000
cnot12[0, 1]-d1-carrier-framechange   : 4.712 rad
cnot12[0, 1]-d2-gauss2-amp            : 30.000 mV
cnot12[0, 1]-d2-gauss2-freq_offset    : -53.000 MHz 2pi
cnot12[0, 1]-d2-gauss2-xy_angle       : -444.089 arad
cnot12[0, 1]-d2-gauss2-delta          : -1.000
cnot12[0, 1]-d2-carrier-framechange   : 0.000 rad
```

As a fidelity function we choose unitary fidelity as well as LBFG-S (a wrapper of the scipy implementation) from our library.

```
import os
import tempfile
from c3.optimizers.optimalcontrol import OptimalControl

log_dir = os.path.join(tempfile.TemporaryDirectory().name, "c3logs")
opt = OptimalControl(
    dir_path=log_dir,
    fid_func=fidelities.unitary_infid_set,
    fid_subspace=["Q1", "Q2"],
    pmap=parameter_map,
    algorithm=algorithms.lbfgs,
    options={
        "maxfun": 25
    },
```

(continues on next page)

(continued from previous page)

```
    run_name="cnot12"
)
```

Start the optimisation

```
exp.set_opt_gates(opt_gates)
opt.set_exp(exp)
opt.optimize_controls()
```

```
C3:STATUS:Saving as: /tmp/tmpjx66lyg2/c3logs/cnot12/2021_12_08_T_12_27_05/open_loop.log
1 0.8790556354859858
2 0.9673489008768812
3 0.758622722337525
4 0.7679637459613755
5 0.696230145207080 2
6 0.541321232138175
7 0.5682335581707882
8 0.382921410272719
9 0.43114251105289114
10 0.30099424375388173
11 0.32449492775751976
12 0.26537726105532744
13 0.2653362073570743
14 0.25121669688810866
15 0.23925168937407626
16 0.18551042816386099
17 0.1305543307431979
18 0.07413739981051659
19 0.031551815290153495
20 0.017447484467834062
21 0.007924221221055072
22 0.006483318391815374
23 0.005732979353259449
24 0.005594385264244273
25 0.0055582927728303755
26 0.005521343169743842
```

The final parameters and the fidelity are

```
parameter_map.print_parameters()
print(opt.current_best_goal)
```

```
cnot12[0, 1]-d1-gauss1-amp            : 2.359 V
cnot12[0, 1]-d1-gauss1-freq_offset    : -53.252 MHz 2pi
cnot12[0, 1]-d1-gauss1-xy_angle       : 587.818 mrad
cnot12[0, 1]-d1-gauss1-delta          : -743.473 m
cnot12[0, 1]-d1-carrier-framechange   : -815.216 mrad
cnot12[0, 1]-d2-gauss2-amp            : 56.719 mV
cnot12[0, 1]-d2-gauss2-freq_offset    : -53.176 MHz 2pi
cnot12[0, 1]-d2-gauss2-xy_angle       : -135.515 mrad
cnot12[0, 1]-d2-gauss2-delta          : -519.864 m
cnot12[0, 1]-d2-carrier-framechange   : 598.919 mrad
```

(continues on next page)

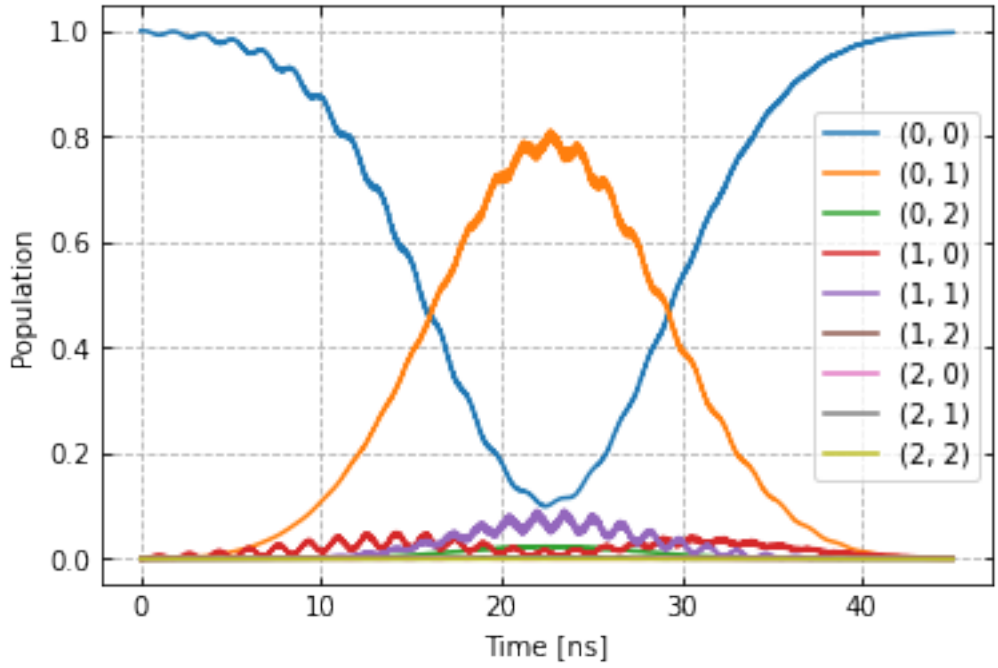
(continued from previous page)

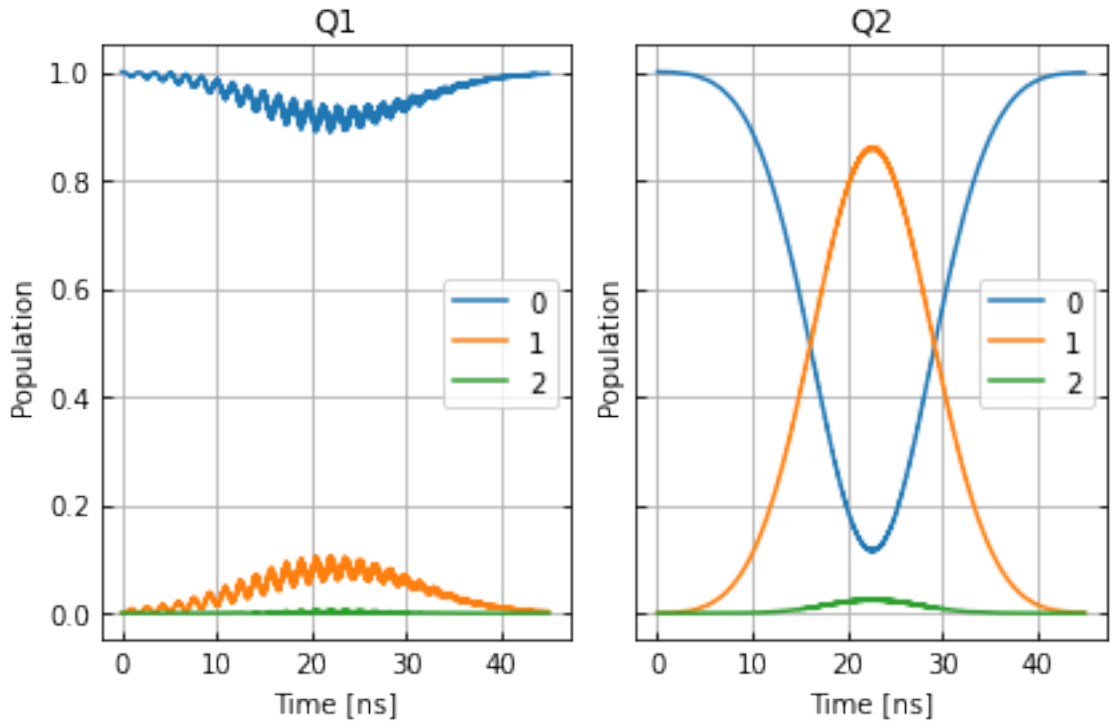
```
0.005521343169743842
```

6.9 Results of the optimisation

Plotting the dynamics with the same initial state:

```
plot_dynamics(exp, init_state, sequence)
plotSplittedPopulation(exp, init_state, sequence)
```

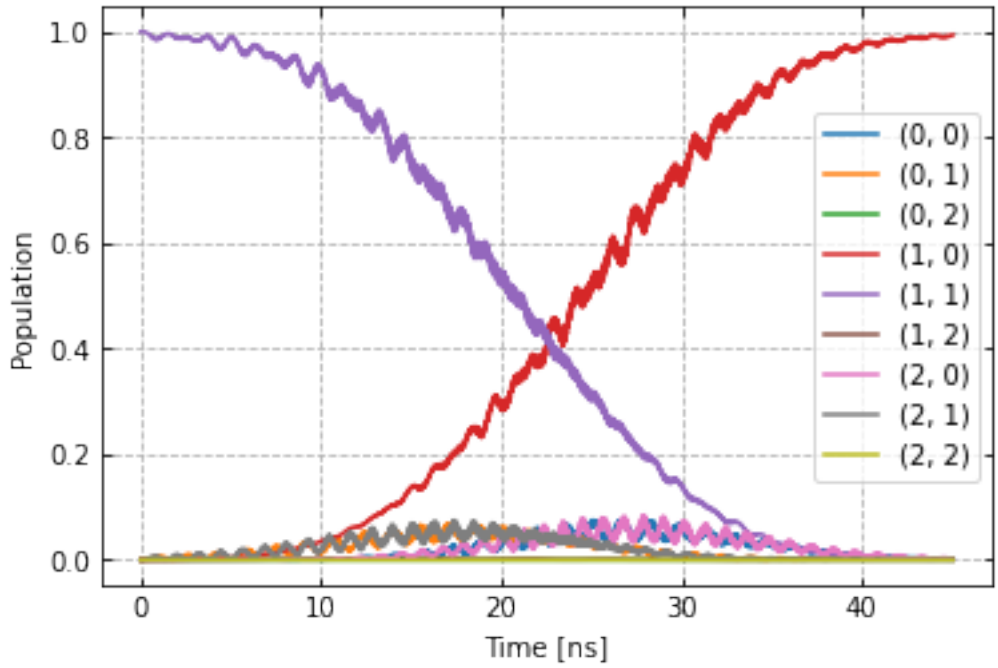


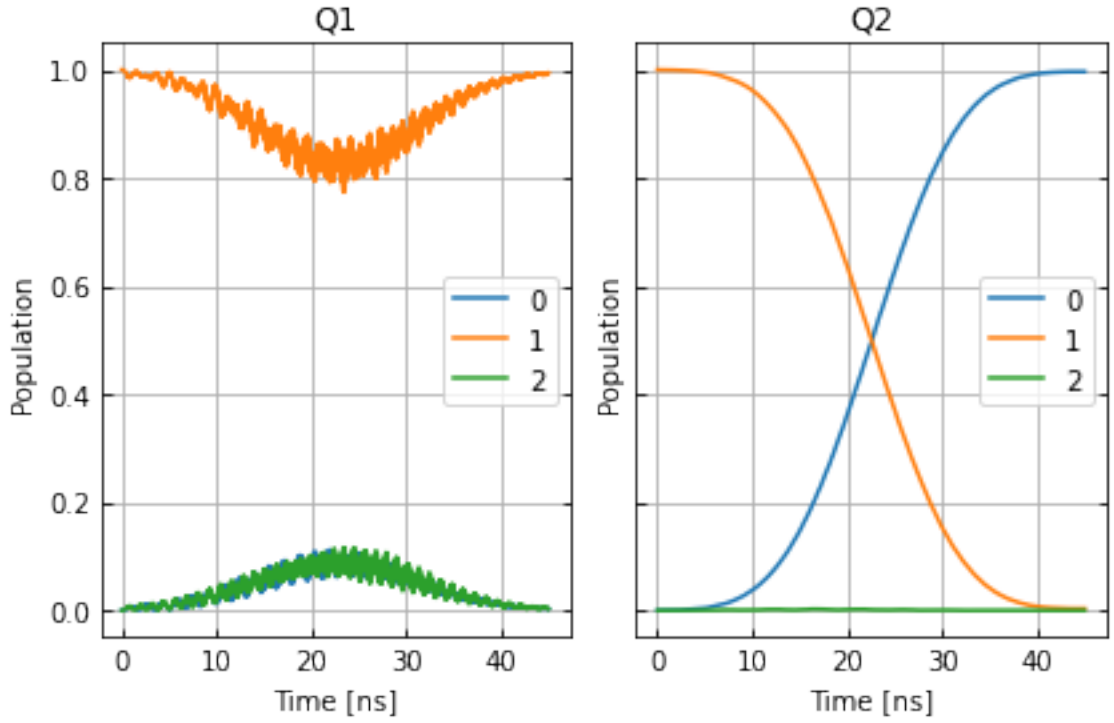


Now we plot the dynamics for the control in the excited state.

```
psi_init = [[0] * 9]
psi_init[0][4] = 1
init_state = tf.transpose(tf.constant(psi_init, tf.complex128))
print(init_state)

plot_dynamics(exp, init_state, sequence)
plotSplittedPopulation(exp, init_state, sequence)
```

```
tf.Tensor(
[[0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [1.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]], shape=(9, 1), dtype=complex128)
```





As intended, the dynamics of the target is dependent on the control qubit performing a flip if the control is excited and an identity otherwise.

CHAPTER

SEVEN

SIMULATED CALIBRATION

Calibration of control pulses is the process of fine-tuning parameters in a feedback-loop with the experiment. We will simulate this process here by constructing a black-box simulation and interacting with it exactly like an experiment.

We have manange imports and creation of the black-box the same way as in the previous example in a helper `single_qubit_blackbox_exp.py`.

```
from single_qubit_blackbox_exp import create_experiment

blackbox = create_experiment()
```

This blackbox is constructed the same way as in the OptimalControl example. The difference will be in how we interact with it. First, we decide on what experiment we want to perform and need to specify it as a python function. A general, minimal example would be

```
def exp_communication(params):
  # Send parameters to experiment controller
  # and receive a measurement result.
  return measurement_result
```

Again, `params` is a linear vector of bare numbers. The measurement result can be a single number or a set of results. It can also include additional information about statistics, like averaging, standard deviation, etc.

7.1 ORBIT - Single-length randomized benchmarking

The following defines an ORBIT procedure. In short, we define sequences of gates that result in an identity gate if our individual gates are perfect. Any deviation from identity gives us a measure of the imperfections in our gates. Our helper `qt_utils` provides these sequences.

```
from c3.utils import qt_utils
```

```
qt_utils.single_length_RB(
            RB_number=1, RB_length=5, target=0
    )
```

```
[['ry90m[0]',
  'rx90p[0]',
  'rx90m[0]',
  'rx90p[0]',
  'ry90p[0]',
```

(continues on next page)

(continued from previous page)

```
  'ry90p[0]',
  'ry90p[0]',
  'rx90p[0]',
  'ry90m[0]',
  'rx90p[0]']]
```

The desired number of 5 gates is selected from a specific set (the Clifford group) and has to be decomposed into the available gate-set. Here, this means 4 gates per Clifford, hence a sequence of 20 gates.

7.2 Communication with the experiment

Some of the following code is specific to the fact that this a *simulated* calibration. The interface of C^2 to the experiment is simple: parameters in $\rightarrow$ results out. Thus, we have to wrap the blackbox by defining the target states and the `opt_map`.

```
import numpy as np
import tensorflow as tf

def ORBIT_wrapper(p):
    def ORBIT(params, exp, opt_map, qubit_labels, logdir):
        ### ORBIT meta-parameters ###
        RB_length = 60 # How long each sequence is
        RB_number = 40  # How many sequences
        shots = 1000    # How many averages per readout

        ################################
        ### Simulation specific part ###
        ################################

        do_noise = False  # Whether to add artificial noise to the results

        qubit_label = list(qubit_labels.keys())[0]
        state_labels = qubit_labels[qubit_label]
        state_label = [tuple(l) for l in state_labels]

        # Creating the RB sequences #
        seqs = qt_utils.single_length_RB(
                RB_number=RB_number, RB_length=RB_length, target=0
        )

        # Transmitting the parameters to the experiment #
        exp.pmap.set_parameters(params, opt_map)
        exp.set_opt_gates_seq(seqs)

        # Simulating the gates #
        U_dict = exp.compute_propagators()

        # Running the RB sequences and read-out the results #
        pops = exp.evaluate(seqs)
        pop1s, _ = exp.process(pops, labels=state_label)
```

(continues on next page)

(continued from previous page)

```
        results = []
        results_std = []
        shots_nums = []

        # Collecting results and statistics, add noise #
        if do_noise:
            for p1 in pop1s:
                draws = tf.keras.backend.random_binomial(
                    [shots],
                    p=p1[0],
                    dtype=tf.float64,
                )
                results.append([np.mean(draws)])
                results_std.append([np.std(draws)/np.sqrt(shots)])
                shots_nums.append([shots])
        else:
            for p1 in pop1s:
                results.append(p1.numpy())
                results_std.append([0])
                shots_nums.append([shots])

        ########################################
        ### End of Simulation specific part ###
        ########################################

        goal = np.mean(results)
        return goal, results, results_std, seqs, shots_nums
    return ORBIT(
                p, blackbox, gateset_opt_map, state_labels, "/tmp/c3logs/blackbox"
            )
```

7.3 Optimization

We first import algorithms and the correct optimizer object.

```
import copy

from c3.experiment import Experiment as Exp
from c3.c3objs import Quantity as Qty
from c3.parametermap import ParameterMap as PMap
from c3.libraries import algorithms, envelopes
from c3.signal import gates, pulse
from c3.optimizers.calibration import Calibration
```

7.4 Representation of the experiment within C^3

At this point we have to make sure that the gates ("RX90p", etc.) and drive line ("d1") are compatible to the experiment controller operating the blackbox. We mirror the blackbox by creating an experiment in the C^3 context:

```
t_final = 7e-9   # Time for single qubit gates
sideband = 50e6
lo_freq = 5e9 + sideband

 # ### MAKE GATESET
gauss_params_single = {
    'amp': Qty(
        value=0.45,
        min_val=0.4,
        max_val=0.6,
        unit="V"
    ),
    't_final': Qty(
        value=t_final,
        min_val=0.5 * t_final,
        max_val=1.5 * t_final,
        unit="s"
    ),
    'sigma': Qty(
        value=t_final / 4,
        min_val=t_final / 8,
        max_val=t_final / 2,
        unit="s"
    ),
    'xy_angle': Qty(
        value=0.0,
        min_val=-0.5 * np.pi,
        max_val=2.5 * np.pi,
        unit='rad'
    ),
    'freq_offset': Qty(
        value=-sideband - 0.5e6,
        min_val=-53 * 1e6,
        max_val=-47 * 1e6,
        unit='Hz 2pi'
    ),
    'delta': Qty(
        value=-1,
        min_val=-5,
        max_val=3,
        unit=""
    )
}

gauss_env_single = pulse.Envelope(
    name="gauss",
    desc="Gaussian comp for single-qubit gates",
    params=gauss_params_single,
```

(continues on next page)

(continued from previous page)

```
    shape=envelopes.gaussian_nonorm
)
nodrive_env = pulse.Envelope(
    name="no_drive",
    params={
        't_final': Qty(
            value=t_final,
            min_val=0.5 * t_final,
            max_val=1.5 * t_final,
            unit="s"
        )
    },
    shape=envelopes.no_drive
)
carrier_parameters = {
    'freq': Qty(
        value=lo_freq,
        min_val=4.5e9,
        max_val=6e9,
        unit='Hz 2pi'
    ),
    'framechange': Qty(
        value=0.0,
        min_val= -np.pi,
        max_val= 3 * np.pi,
        unit='rad'
    )
}
carr = pulse.Carrier(
    name="carrier",
    desc="Frequency of the local oscillator",
    params=carrier_parameters
)

rx90p = gates.Instruction(
    name="rx90p",
    t_start=0.0,
    t_end=t_final,
    channels=["d1"]
)
QId = gates.Instruction(
    name="id",
    t_start=0.0,
    t_end=t_final,
    channels=["d1"]
)

rx90p.add_component(gauss_env_single, "d1")
rx90p.add_component(carr, "d1")
QId.add_component(nodrive_env, "d1")
QId.add_component(copy.deepcopy(carr), "d1")
QId.comps['d1']['carrier'].params['framechange'].set_value(
```

(continues on next page)

(continued from previous page)

```
    (-sideband * t_final * 2 * np.pi) % (2*np.pi)
)
ry90p = copy.deepcopy(rx90p)
ry90p.name = "ry90p"
rx90m = copy.deepcopy(rx90p)
rx90m.name = "rx90m"
ry90m = copy.deepcopy(rx90p)
ry90m.name = "ry90m"
ry90p.comps['d1']['gauss'].params['xy_angle'].set_value(0.5 * np.pi)
rx90m.comps['d1']['gauss'].params['xy_angle'].set_value(np.pi)
ry90m.comps['d1']['gauss'].params['xy_angle'].set_value(1.5 * np.pi)

parameter_map = PMap(instructions=[QId, rx90p, ry90p, rx90m, ry90m])

# ### MAKE EXPERIMENT
exp = Exp(pmap=parameter_map)
```

Next, we define the parameters we whish to calibrate. See how these gate instructions are defined in the experiment setup example or in `single_qubit_blackbox_exp.py`. Our gate-set is made up of 4 gates, rotations of 90 degrees around the x and y-axis in positive and negative direction. While it is possible to optimize each parameters of each gate individually, in this example all four gates share parameters. They only differ in the phase ϕ_{xy} that is set in the definitions.

```
gateset_opt_map =   [
    [
      ("rx90p[0]", "d1", "gauss", "amp"),
      ("ry90p[0]", "d1", "gauss", "amp"),
      ("rx90m[0]", "d1", "gauss", "amp"),
      ("ry90m[0]", "d1", "gauss", "amp")
    ],
    [
      ("rx90p[0]", "d1", "gauss", "delta"),
      ("ry90p[0]", "d1", "gauss", "delta"),
      ("rx90m[0]", "d1", "gauss", "delta"),
      ("ry90m[0]", "d1", "gauss", "delta")
    ],
    [
      ("rx90p[0]", "d1", "gauss", "freq_offset"),
      ("ry90p[0]", "d1", "gauss", "freq_offset"),
      ("rx90m[0]", "d1", "gauss", "freq_offset"),
      ("ry90m[0]", "d1", "gauss", "freq_offset")
    ],
    [
      ("id[0]", "d1", "carrier", "framechange")
    ]
  ]

parameter_map.set_opt_map(gateset_opt_map)
```

As defined above, we have 16 parameters where 4 share their numerical value. This leaves 4 values to optimize.

```
parameter_map.print_parameters()
```

```
rx90p[0]-d1-gauss-amp                 : 450.000 mV
ry90p[0]-d1-gauss-amp
rx90m[0]-d1-gauss-amp
ry90m[0]-d1-gauss-amp

rx90p[0]-d1-gauss-delta               : -1.000
ry90p[0]-d1-gauss-delta
rx90m[0]-d1-gauss-delta
ry90m[0]-d1-gauss-delta

rx90p[0]-d1-gauss-freq_offset         : -50.500 MHz 2pi
ry90p[0]-d1-gauss-freq_offset
rx90m[0]-d1-gauss-freq_offset
ry90m[0]-d1-gauss-freq_offset

id[0]-d1-carrier-framechange          : 4.084 rad
```

It is important to note that in this example, we are transmitting only these four parameters to the experiment. We don't know how the blackbox will implement the pulse shapes and care has to be taken that the parameters are understood on the other end. Optionally, we could specifiy a virtual AWG within C^3 and transmit pixilated pulse shapes directly to the physiscal AWG.

7.5 Algorithms

As an optimization algoritm, we choose CMA-Es and set up some options specific to this algorithm.

```
alg_options = {
    "popsize" : 10,
    "maxfevals" : 300,
    "init_point" : "True",
    "tolfun" : 0.01,
    "spread" : 0.25
  }
```

We define the subspace as both excited states $\{|1>, |2>\}$, assuming read-out can distinguish between 0, 1 and 2.

```
state_labels = {
      "excited" : [(1,), (2,)]
  }
```

In the real world, this setup needs to be handled in the experiment controller side. We construct the optimizer object with the options we setup:

```
import os
import tempfile

# Create a temporary directory to store logfiles, modify as needed
log_dir = os.path.join(tempfile.TemporaryDirectory().name, "c3logs")

opt = Calibration(
    dir_path=log_dir,
    run_name="ORBIT_cal",
```

(continues on next page)

(continued from previous page)

```
    eval_func=ORBIT_wrapper,
    pmap=parameter_map,
    exp_right=exp,
    algorithm=algorithms.cmaes,
    options=alg_options
)
opt.set_exp(exp)
```

And run the calibration:

```
x = parameter_map.get_parameters_scaled()
```

```
opt.optimize_controls()



C3:STATUS:Saving as: /tmp/tmpicnnbliz/c3logs/ORBIT_cal/2021_01_28_T_15_17_30/calibration.
↪log
(5_w,10)-aCMA-ES (mu_w=3.2,w_1=45%) in dimension 4 (seed=912463, Thu Jan 28 15:17:30␣
↪2021)
C3:STATUS:Adding initial point to CMA sample.
Iterat #Fevals   function value  axis ratio  sigma  min&max std  t[m:s]
    1     10 1.446744168975211e-01 1.0e+00 2.11e-01  2e-01  2e-01 1:18.9
    2     20 2.07435937466505 0e-01 1.4e+00 1.96e-01  1e-01  2e-01 2:28.5
    3     30 1.042216610303495e-01 1.5e+00 1.76e-01  1e-01  2e-01 3:36.4
    4     40 1.720244494886762e-01 1.9e+00 1.88e-01  1e-01  2e-01 4:46.5
    5     50 9.761264536669531e-02 2.2e+00 2.05e-01  1e-01  2e-01 6:15.4
    6     60 1.956493007802809e-01 2.8e+00 1.75e-01  8e-02  2e-01 7:17.9
    7     70 6.625917264980545e-02 3.0e+00 2.20e-01  9e-02  3e-01 8:22.8
    8     80 7.697621753428294e-02 4.1e+00 2.19e-01  8e-02  3e-01 9:25.8
    9     90 8.826758030850271e-02 4.7e+00 1.85e-01  6e-02  3e-01 10:28.7
   10    100 9.099567192014653e-02 5.3e+00 1.59e-01  4e-02  2e-01 11:32.7
   11    110 6.673347151005890e-02 6.9e+00 1.49e-01  3e-02  2e-01 12:27.9
   12    120 6.822093884865452e-02 7.6e+00 1.68e-01  4e-02  2e-01 13:26.6
   13    130 6.307315835232992e-02 8.1e+00 1.42e-01  3e-02  2e-01 14:22.8
   14    140 6.301017013241370e-02 7.8e+00 1.42e-01  2e-02  2e-01 15:18.7
   15    150 6.795728963072037e-02 9.3e+00 1.32e-01  2e-02  2e-01 16:15.8
   16    160 7.675314380135559e-02 9.2e+00 1.03e-01  2e-02  1e-01 17:12.9
   17    170 6.806172046778505e-02 9.1e+00 8.05e-02  1e-02  1e-01 18:11.5
   18    180 5.698438523961635e-02 1.0e+01 7.42e-02  9e-03  9e-02 19:06.1
   19    190 5.536707419037251e-02 1.1e+01 6.89e-02  8e-03  9e-02 20:00.6
   20    200 4.924177790655197e-02 1.2e+01 7.31e-02  8e-03  9e-02 20:58.2
   21    210 5.836136870997249e-02 1.2e+01 8.20e-02  8e-03  1e-01 21:55.1
   22    220 5.463139088536284e-02 1.3e+01 8.29e-02  9e-03  1e-01 22:51.0
   23    230 4.562693294212217e-02 1.4e+01 8.66e-02  9e-03  1e-01 23:48.3
   24    240 5.188441161313757e-02 1.6e+01 7.74e-02  7e-03  1e-01 24:46.1
   25    250 5.199237655967553e-02 1.7e+01 7.41e-02  6e-03  9e-02 25:47.1
   26    260 5.684400595430246e-02 1.6e+01 6.41e-02  5e-03  9e-02 26:43.7
   27    270 4.441763519087279e-02 1.8e+01 5.12e-02  4e-03  7e-02 27:36.2
   28    280 4.994977609185950e-02 1.8e+01 5.51e-02  5e-03  8e-02 28:33.9
   29    290 6.108777009078262e-02 1.8e+01 5.14e-02  4e-03  7e-02 29:30.4
   30    300 5.658962789881571e-02 1.8e+01 4.65e-02  4e-03  6e-02 30:28.0
```

(continues on next page)

(continued from previous page)

```
   31    310 5.765354335022381e-02 1.8e+01 4.77e-02  4e-03  6e-02 31:26.9
termination on maxfevals=300
final/bestever f-value = 5.765354e-02 4.441764e-02
incumbent solution: [-0.4739081748676816, -0.09828275146514219, -1.0504851431889897, 0.
↪9108808620989909]
std deviation: [0.013780217516583012, 0.0038070906112681576, 0.02460767003734409, 0.
↪05816700836608336]
```

7.6 Analysis

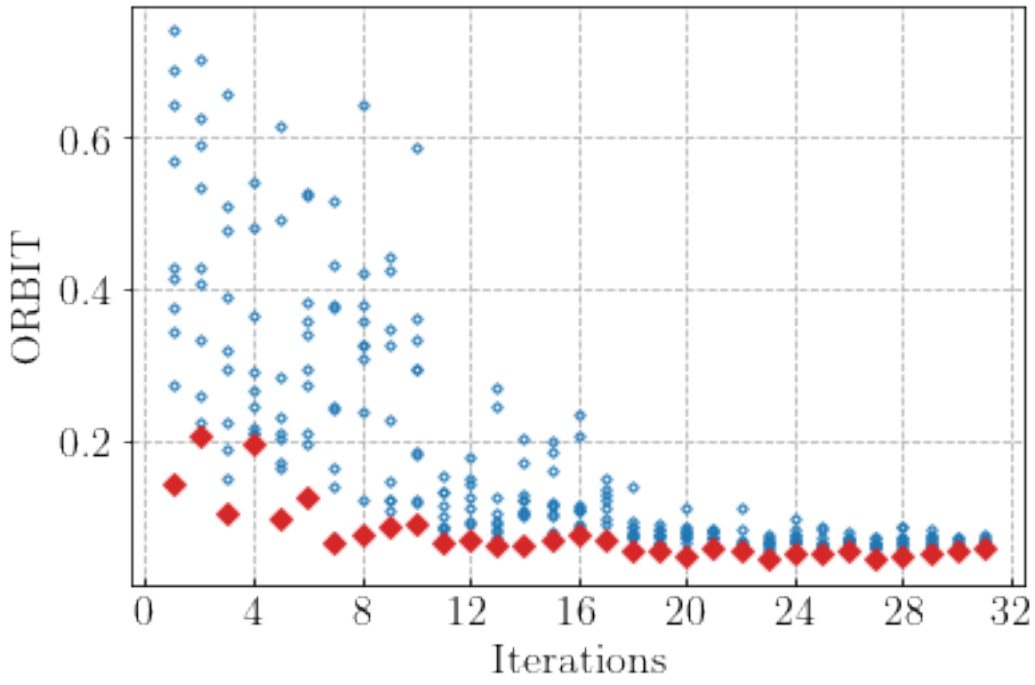
The following code uses matplotlib to create an ORBIT plot from the logfile.

```
import json
from matplotlib.ticker import MaxNLocator
from  matplotlib import rcParams
from matplotlib import cycler
import matplotlib as mpl
import matplotlib.pyplot as plt

rcParams['xtick.direction'] = 'in'
rcParams['axes.grid'] = True
rcParams['grid.linestyle'] = '--'
rcParams['markers.fillstyle'] = 'none'
rcParams['axes.prop_cycle'] = cycler(
    'linestyle', ["-", "--"]
)
rcParams['text.usetex'] = True
rcParams['font.size'] = 16
rcParams['font.family'] = 'serif'

logfilename = opt.logdir + "calibration.log"
with open(logfilename, "r") as filename:
    log = filename.readlines()


options = json.loads(log[7])

goal_function = []
batch = 0
batch_size = options["popsize"]


eval = 0
for line in log[9:]:
    if line[0] == "{":
        if not eval % batch_size:
            batch = eval // batch_size
            goal_function.append([])
        eval += 1
        point = json.loads(line)
```

(continues on next page)

(continued from previous page)

```
        if 'goal' in point.keys():
            goal_function[batch].append(point['goal'])

# Clean unfinished batch
if len(goal_function[-1])<batch_size:
    goal_function.pop(-1)

fig, ax = plt.subplots(1)
means = []
bests = []
for ii in range(len(goal_function)):
    means.append(np.mean(np.array(goal_function[ii])))
    bests.append(np.min(np.array(goal_function[ii])))
    for pt in goal_function[ii]:
        ax.plot(ii+1, pt, color='tab:blue', marker="D", markersize=2.5, linewidth=0)

ax.xaxis.set_major_locator(MaxNLocator(integer=True))
ax.set_ylabel('ORBIT')
ax.set_xlabel('Iterations')
ax.plot(
    range(1, len(goal_function)+1), bests, color="tab:red", marker="D",
    markersize=5.5, linewidth=0, fillstyle='full'
)
```



CHAPTER

EIGHT

MODEL LEARNING

In this notebook, we will use a dataset from a simulated experiment, more specifically, the `Simulated_calibration.ipynb` example notebook and perform Model Learning on a simple 1 qubit model.

8.1 Imports

```
import pickle
from pprint import pprint
import copy
import numpy as np
import os
import ast
import pandas as pd

from c3.model import Model as Mdl
from c3.c3objs import Quantity as Qty
from c3.parametermap import ParameterMap as PMap
from c3.experiment import Experiment as Exp
from c3.generator.generator import Generator as Gnr
import c3.signal.gates as gates
import c3.libraries.chip as chip
import c3.generator.devices as devices
import c3.libraries.hamiltonians as hamiltonians
import c3.signal.pulse as pulse
import c3.libraries.envelopes as envelopes
import c3.libraries.tasks as tasks
from c3.optimizers.modellearning import ModelLearning
```

8.1.1 The Dataset

We first take a look below at the dataset and its properties. To explore more details about how the dataset is generated, please refer to the `Simulated_calibration.ipynb` example notebook.

```
DATAFILE_PATH = "data/small_dataset.pkl"
```

```
with open(DATAFILE_PATH, "rb+") as file:
    data = pickle.load(file)
```

```
data.keys()
```

```
dict_keys(['seqs_grouped_by_param_set', 'opt_map'])
```

Since this dataset was obtained from an ORBIT (arXiv:1403.0035) calibration experiment, we have the `opt_map` which will tell us about the gateset parameters being optimized.

```
data["opt_map"]
```

```
[['rx90p[0]-d1-gauss-amp',
  'ry90p[0]-d1-gauss-amp',
  'rx90m[0]-d1-gauss-amp',
  'ry90m[0]-d1-gauss-amp'],
 ['rx90p[0]-d1-gauss-delta',
  'ry90p[0]-d1-gauss-delta',
  'rx90m[0]-d1-gauss-delta',
  'ry90m[0]-d1-gauss-delta'],
 ['rx90p[0]-d1-gauss-freq_offset',
  'ry90p[0]-d1-gauss-freq_offset',
  'rx90m[0]-d1-gauss-freq_offset',
  'ry90m[0]-d1-gauss-freq_offset'],
 ['id[0]-d1-carrier-framechange']]
```

This `opt_map` implies the calibration experiment focussed on optimizing the amplitude, delta and frequency offset of the gaussian pulse, along with the framechange angle

Now onto the actual measurement data from the experiment runs

```
seqs_data = data["seqs_grouped_by_param_set"]
```

How many experiment runs do we have?

```
len(seqs_data)
```

```
41
```

What does the data from each experiment look like?

We take a look at the first data point

```
example_data_point = seqs_data[0]
```

```
example_data_point.keys()
```

```
dict_keys(['params', 'seqs', 'results', 'results_std', 'shots'])
```

These `keys` are useful in understanding the structure of the dataset. We look at them one by one.

```
example_data_point["params"]
```

```
[450.000 mV, -1.000 , -50.500 MHz 2pi, 4.084 rad]
```

These are the parameters for our parameterised gateset, for the first experiment run. They correspond to the optimization parameters we previously discussed.

The `seqs` key stores the sequence of gates that make up this ORBIT calibration experiment. Each ORBIT sequence consists of a set of gates, followed by a measurement operation. This is then repeated for some `n` number of shots (eg, `1000` in this case) and we only store the averaged result along with the standard deviation of these readout shots. Each experiment in turn consists of a number of these ORBIT sequences. The terms *sequence*, *set* and *experiment* are used somewhat loosely here, so we show below what these look like.

A single ORBIT sequence

```
example_data_point["seqs"][0]
```

```
['ry90p[0]',
 'rx90p[0]',
 'rx90p[0]',
 'rx90m[0]',
 'ry90p[0]',
 'ry90p[0]',
 'rx90p[0]',
 'ry90p[0]',
 'rx90p[0]',
 'rx90p[0]',
 'ry90p[0]',
 'rx90m[0]',
 'rx90p[0]',
 'rx90p[0]',
 'ry90p[0]',
 'ry90p[0]',
 'rx90p[0]',
 'ry90p[0]',
 'ry90m[0]',
 'rx90p[0]',
 'rx90p[0]',
 'ry90m[0]',
 'rx90p[0]',
 'rx90p[0]',
 'rx90p[0]',
 'rx90p[0]']
```

Total number of ORBIT sequences in an experiment

```
len(example_data_point["seqs"])
```

```
20
```

Total number of Measurement results

```
len(example_data_point["results"])
```

```
20
```

The measurement results and the standard deviation look like this

```
example_results = [
    (example_data_point["results"][i], example_data_point["results_std"][i])
```

(continues on next page)

(continued from previous page)

```
    for i in range(len(example_data_point["results"]))
]
```

```
pprint(example_results)
```

```
[([0.745], [0.013783141876945182]),
 ([0.213], [0.012947239087929134]),
 ([0.137], [0.0108734079294396]),
 ([0.224], [0.013184233007649706]),
 ([0.434], [0.015673034167001616]),
 ([0.105], [0.009694070352540258]),
 ([0.214], [0.012969348480166613]),
 ([0.112], [0.009972762907038352]),
 ([0.318], [0.014726710426975877]),
 ([0.122], [0.010349685985574633]),
 ([0.348], [0.015063067416698366]),
 ([0.122], [0.010349685985574633]),
 ([0.558], [0.01570464899321217]),
 ([0.186], [0.012304633273690004]),
 ([0.096], [0.009315793041926168]),
 ([0.368], [0.01525044261652756l]),
 ([0.146], [0.011166198995181842]),
 ([0.121], [0.010313049985334118]),
 ([0.748], [0.013729384545565035]),
 ([0.692], [0.01459917805905524])]
```

8.1.2 The Model for Model Learning

An initial model needs to be provided, which we refine by fitting to our calibration data. We do this below. If you want to learn more about what the various components of the model mean, please refer back to the `two_qubits.ipynb` notebook or the documentation.

8.2 Define Constants

```
lindblad = False
dressed = True
qubit_lvls = 3
freq = 5.001e9
anhar = -210.001e6
init_temp = 0
qubit_temp = 0
t_final = 7e-9  # Time for single qubit gates
sim_res = 100e9
awg_res = 2e9
sideband = 50e6
lo_freq = 5e9 + sideband
```

8.3 Model

```
q1 = chip.Qubit(
    name="Q1",
    desc="Qubit 1",
    freq=Qty(
        value=freq,
        min_val=4.995e9,
        max_val=5.005e9,
        unit="Hz 2pi",
    ),
    anhar=Qty(
        value=anhar,
        min_val=-250e6,
        max_val=-150e6,
        unit="Hz 2pi",
    ),
    hilbert_dim=qubit_lvls,
    temp=Qty(value=qubit_temp, min_val=0.0, max_val=0.12, unit="K"),
)

drive = chip.Drive(
    name="d1",
    desc="Drive 1",
    comment="Drive line 1 on qubit 1",
    connected=["Q1"],
    hamiltonian_func=hamiltonians.x_drive,
)
phys_components = [q1]
line_components = [drive]

init_ground = tasks.InitialiseGround(
    init_temp=Qty(value=init_temp, min_val=-0.001, max_val=0.22, unit="K")
)
task_list = [init_ground]
model = Mdl(phys_components, line_components, task_list)
model.set_lindbladian(lindblad)
model.set_dressed(dressed)
```

8.4 Generator

```
generator = Gnr(
    devices={
        "LO": devices.LO(name="lo", resolution=sim_res, outputs=1),
        "AWG": devices.AWG(name="awg", resolution=awg_res, outputs=1),
        "DigitalToAnalog": devices.DigitalToAnalog(
            name="dac", resolution=sim_res, inputs=1, outputs=1
        ),
        "Response": devices.Response(
            name="resp",
```

(continues on next page)

(continued from previous page)

```
            rise_time=Qty(value=0.3e-9, min_val=0.05e-9, max_val=0.6e-9, unit="s"),
            resolution=sim_res,
            inputs=1,
            outputs=1,
        ),
        "Mixer": devices.Mixer(name="mixer", inputs=2, outputs=1),
        "VoltsToHertz": devices.VoltsToHertz(
            name="v_to_hz",
            V_to_Hz=Qty(value=1e9, min_val=0.9e9, max_val=1.1e9, unit="Hz/V"),
            inputs=1,
            outputs=1,
        ),
    },
    chains={
        "d1": {
            "LO": [],
            "AWG": [],
            "DigitalToAnalog": ["AWG"],
            "Response": ["DigitalToAnalog"],
            "Mixer": ["LO", "Response"],
            "VoltsToHertz": ["Mixer"]
        }
    },
)
generator.devices["AWG"].enable_drag_2()
```

8.5 Gateset

```
gauss_params_single = {
    "amp": Qty(value=0.45, min_val=0.4, max_val=0.6, unit="V"),
    "t_final": Qty(
        value=t_final, min_val=0.5 * t_final, max_val=1.5 * t_final, unit="s"
    ),
    "sigma": Qty(value=t_final / 4, min_val=t_final / 8, max_val=t_final / 2, unit="s"),
    "xy_angle": Qty(value=0.0, min_val=-0.5 * np.pi, max_val=2.5 * np.pi, unit="rad"),
    "freq_offset": Qty(
        value=-sideband - 0.5e6,
        min_val=-60 * 1e6,
        max_val=-40 * 1e6,
        unit="Hz 2pi",
    ),
    "delta": Qty(value=-1, min_val=-5, max_val=3, unit=""),
}

gauss_env_single = pulse.Envelope(
    name="gauss",
    desc="Gaussian comp for single-qubit gates",
    params=gauss_params_single,
    shape=envelopes.gaussian_nonorm,
)
```

(continues on next page)

(continued from previous page)

```
nodrive_env = pulse.Envelope(
    name="no_drive",
    params={
        "t_final": Qty(
            value=t_final, min_val=0.5 * t_final, max_val=1.5 * t_final, unit="s"
        )
    },
    shape=envelopes.no_drive,
)
carrier_parameters = {
    "freq": Qty(
        value=lo_freq,
        min_val=4.5e9,
        max_val=6e9,
        unit="Hz 2pi",
    ),
    "framechange": Qty(value=0.0, min_val=-np.pi, max_val=3 * np.pi, unit="rad"),
}
carr = pulse.Carrier(
    name="carrier",
    desc="Frequency of the local oscillator",
    params=carrier_parameters,
)

rx90p = gates.Instruction(
    name="rx90p", t_start=0.0, t_end=t_final, channels=["d1"], targets=[0]
)
QId = gates.Instruction(
    name="id", t_start=0.0, t_end=t_final, channels=["d1"], targets=[0]
)

rx90p.add_component(gauss_env_single, "d1")
rx90p.add_component(carr, "d1")
QId.add_component(nodrive_env, "d1")
QId.add_component(copy.deepcopy(carr), "d1")
QId.comps["d1"]["carrier"].params["framechange"].set_value(
    (-sideband * t_final) % (2 * np.pi)
)
ry90p = copy.deepcopy(rx90p)
ry90p.name = "ry90p"
rx90m = copy.deepcopy(rx90p)
rx90m.name = "rx90m"
ry90m = copy.deepcopy(rx90p)
ry90m.name = "ry90m"
ry90p.comps["d1"]["gauss"].params["xy_angle"].set_value(0.5 * np.pi)
rx90m.comps["d1"]["gauss"].params["xy_angle"].set_value(np.pi)
ry90m.comps["d1"]["gauss"].params["xy_angle"].set_value(1.5 * np.pi)
```

8.6 Experiment

```
parameter_map = PMap(
    instructions=[QId, rx90p, ry90p, rx90m, ry90m], model=model, generator=generator
)

exp = Exp(pmap=parameter_map)
```

```
exp_opt_map = [[('Q1', 'anhar')], [('Q1', 'freq')]]
exp.pmap.set_opt_map(exp_opt_map)
```

8.6.1 Optimizer

```
datafiles = {"orbit": DATAFILE_PATH} # path to the dataset
run_name = "simple_model_learning" # name of the optimization run
dir_path = "ml_logs" # path to save the learning logs
algorithm = "cma_pre_lbfgs" # algorithm for learning
# this first does a grad-free CMA-ES and then a gradient based LBFGS
options = {
    "cmaes": {
        "popsize": 12,
        "init_point": "True",
        "stop_at_convergence": 10,
        "ftarget": 4,
        "spread": 0.05,
        "stop_at_sigma": 0.01,
    },
    "lbfgs": {"maxfun": 50, "disp": 0},
} # options for the algorithms
sampling = "high_std" # how data points are chosen from the total dataset
batch_sizes = {"orbit": 2} # how many data points are chosen for learning
state_labels = {
    "orbit": [
        [
            1,
        ],
        [
            2,
        ],
    ]
} # the excited states of the qubit model, in this case it is 3-level
```

```
opt = ModelLearning(
    datafiles=datafiles,
    run_name=run_name,
    dir_path=dir_path,
    algorithm=algorithm,
    options=options,
    sampling=sampling,
    batch_sizes=batch_sizes,
```

(continues on next page)

(continued from previous page)

```
    state_labels=state_labels,
    pmap=exp.pmap,
)

opt.set_exp(exp)
```

8.6.2 Model Learning

We are now ready to learn from the data and improve our model

```
opt.run()
```

```
C3:STATUS:Saving as: /home/users/anurag/c3/examples/ml_logs/simple_model_learning/2021_
↪06_30_T_08_59_07/model_learn.log
(6_w,12)-aCMA-ES (mu_w=3.7,w_1=40%) in dimension 2 (seed=125441, Wed Jun 30 08:59:07␣
↪2021)
C3:STATUS:Adding initial point to CMA sample.
Iterat #Fevals   function value  axis ratio  sigma  min&max std  t[m:s]
    1     12 3.767977884544180e+00 1.0e+00 4.89e-02  4e-02  5e-02 0:31.1
termination on ftarget=4
final/bestever f-value = 3.767978e+00 3.767978e+00
incumbent solution: [-0.22224933524057258, 0.17615005514516885]
std deviation: [0.0428319357676611, 0.04699011947850928]
C3:STATUS:Saving as: /home/users/anurag/c3/examples/ml_logs/simple_model_learning/2021_
↪06_30_T_08_59_07/confirm.log
```

8.7 Result of Model Learning

```
opt.current_best_goal
```

```
-0.031570491979011794
```

```
print(opt.pmap.str_parameters(opt.pmap.opt_map))
```

```
Q1-anhar                                : -210.057 MHz 2pi
Q1-freq                                 : 5.000 GHz 2pi
```

8.7.1 Visualisation & Analysis of Results

The Model Learning logs provide a useful way to visualise the learning process and also understand what's going wrong (or right). We now process these logs to read some data points and also plot some visualisations of the Model Learning process

8.8 Open, Clean-up and Convert Logfiles

```
LOGDIR = opt.logdir
```

```
logfile = os.path.join(LOGDIR, "model_learn.log")
with open(logfile, "r") as f:
    log = f.readlines()
```

```
params_names = [
    item for sublist in (ast.literal_eval(log[3].strip("\n"))) for item in sublist
]
print(params_names)
```

```
['Q1-anhar', 'Q1-freq']
```

```
data_list_dict = list()
for line in log[9:]:
    if line[0] == "{":
        temp_dict = ast.literal_eval(line.strip("\n"))
        for index, param_name in enumerate(params_names):
            temp_dict[param_name] = temp_dict["params"][index]
        temp_dict.pop("params")
        data_list_dict.append(temp_dict)
```

```
data_df = pd.DataFrame(data_list_dict)
```

8.9 Summary of Logs

```
data_df.describe()
```

Best Point

```
best_point_file = os.path.join(LOGDIR, 'best_point_model_learn.log')
```

```
with open(best_point_file, "r") as f:
    best_point = f.read()
    best_point_log_dict = ast.literal_eval(best_point)

best_point_dict = dict(zip(params_names, best_point_log_dict["optim_status"]["params"]))
best_point_dict["goal"] = best_point_log_dict["optim_status"]["goal"]
print(best_point_dict)
```

```
{'Q1-anhar': -210057285.60876995, 'Q1-freq': 5000081146.481342, 'goal': -0.
↪031570491979011794}
```

8.10 Plotting

We use `matplotlib` to produce the plots below. Please make sure you have the same installed in your python environment.

```
!pip install -q matplotlib
```

```
[33mWARNING: You are using pip version 21.1.2; however, version 21.1.3 is available.
You should consider upgrading via the '/home/users/anurag/.conda/envs/c3-qopt/bin/python␣
↪-m pip install --upgrade pip' command.[0m
```

```
from matplotlib.ticker import MaxNLocator
from  matplotlib import rcParams
from matplotlib import cycler
import matplotlib as mpl
import matplotlib.pyplot as plt
```

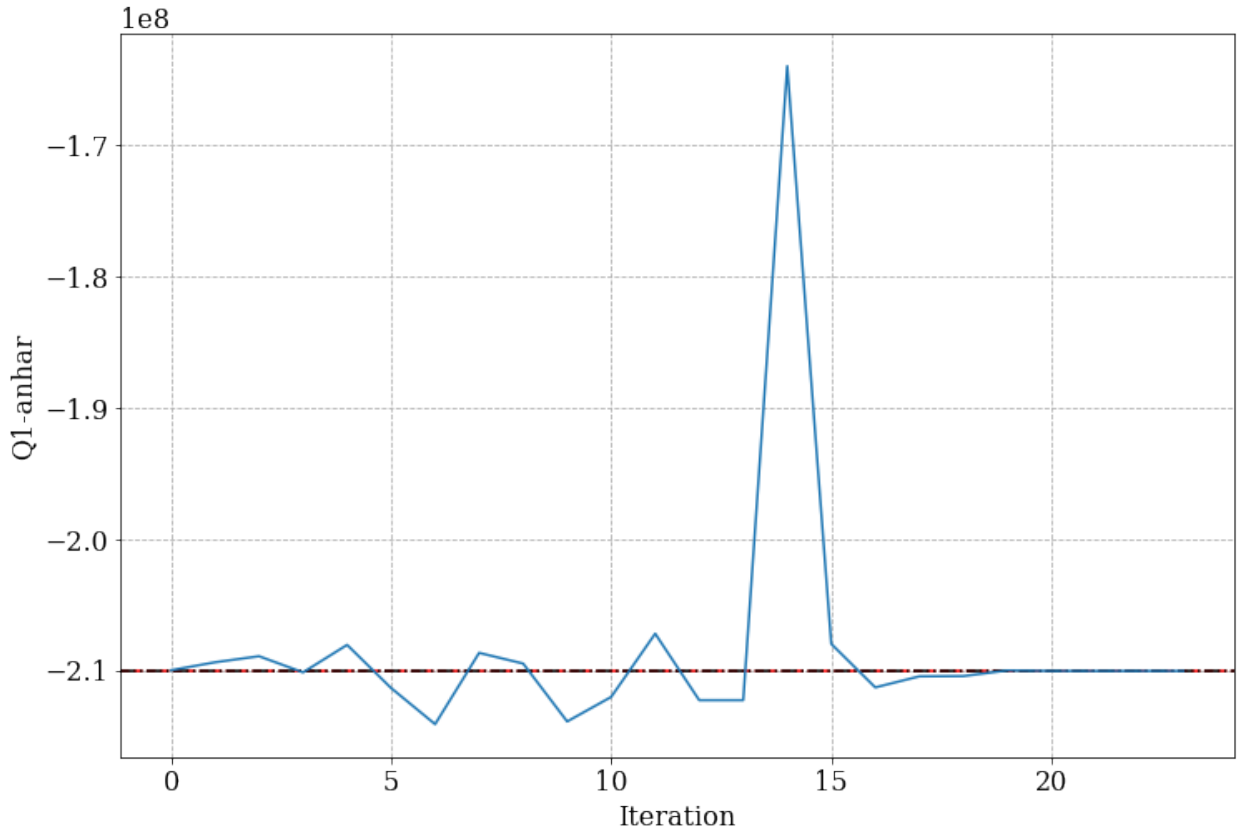
```
rcParams["axes.grid"] = True
rcParams["grid.linestyle"] = "--"

# enable usetex by setting it to True if LaTeX is installed
rcParams["text.usetex"] = False
rcParams["font.size"] = 16
rcParams["font.family"] = "serif"
```

In the plots below, the blue line shows the progress of the parameter optimization while the black and the red lines indicate the converged and true value respectively

8.11 Qubit Anharmonicity

```
plot_item = "Q1-anhar"
true_value = -210e6

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)
ax.set_xlabel("Iteration")
ax.set_ylabel(plot_item)
ax.axhline(y=true_value, color="red", linestyle="--")
ax.axhline(y=best_point_dict[plot_item], color="black", linestyle="-.")
ax.plot(data_df[plot_item])
```

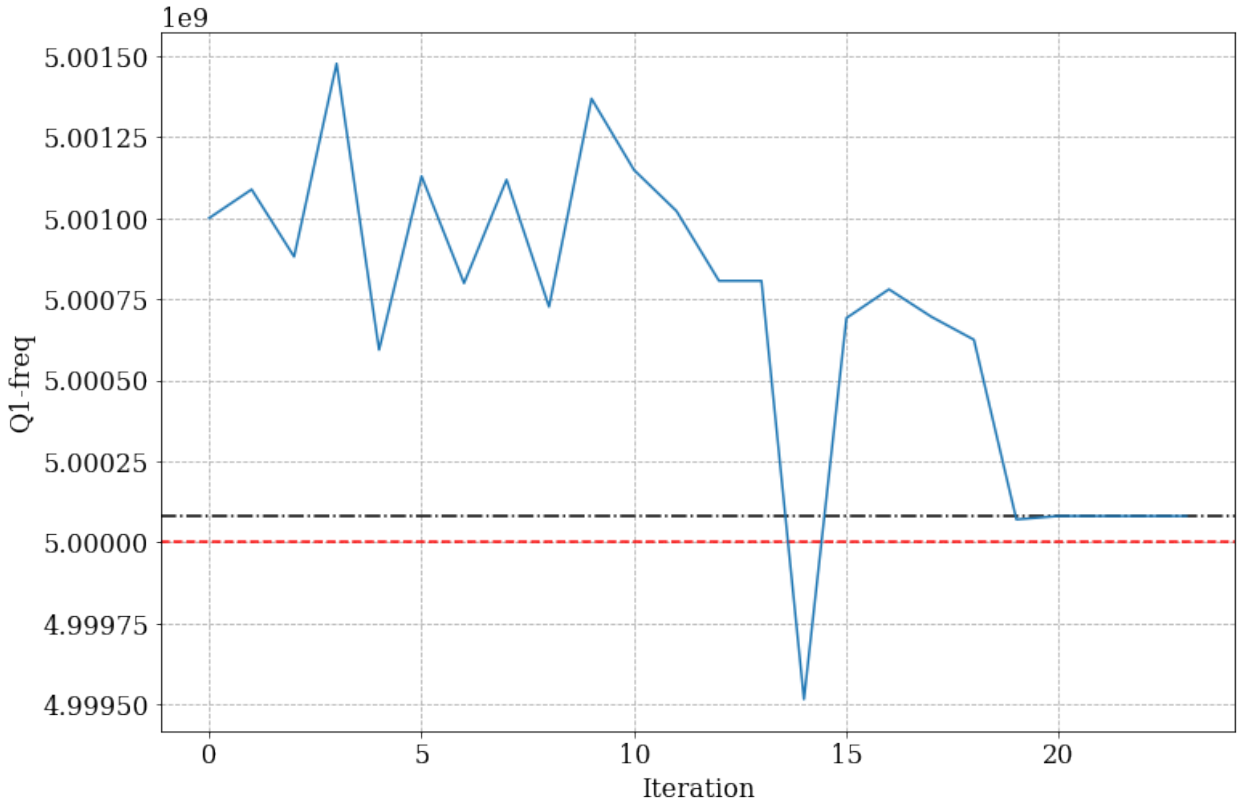
```
[<matplotlib.lines.Line2D at 0x7fc3c5ab5f70>]
```



8.12 Qubit Frequency

```
plot_item = "Q1-freq"
true_value = 5e9

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)
ax.set_xlabel("Iteration")
ax.set_ylabel(plot_item)
ax.axhline(y=true_value, color="red", linestyle="--")
ax.axhline(y=best_point_dict[plot_item], color="black", linestyle="-.")
ax.plot(data_df[plot_item])
```

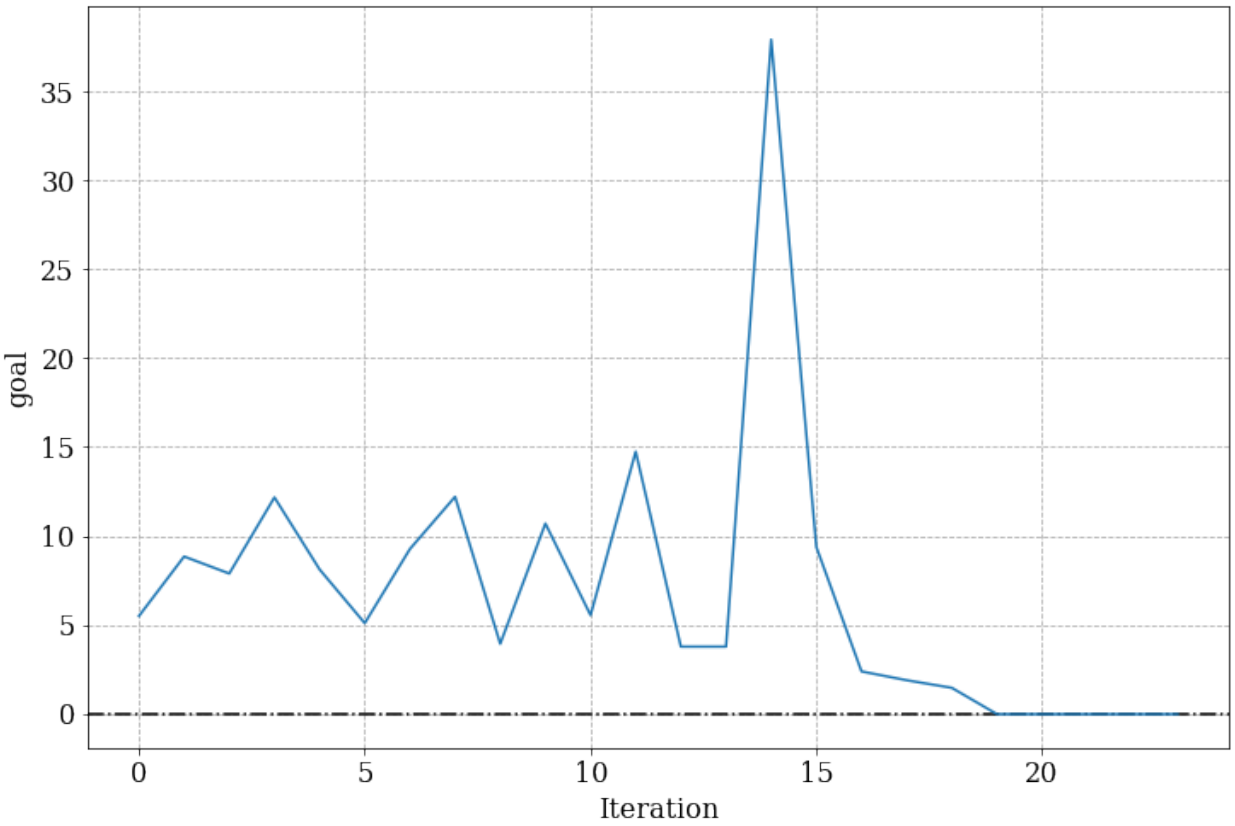
```
[<matplotlib.lines.Line2D at 0x7fc3c59aa340>]
```



8.13 Goal Function

```
plot_item = "goal"

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)
ax.set_xlabel("Iteration")
ax.axhline(y=best_point_dict[plot_item], color="black", linestyle="-.")
ax.set_ylabel(plot_item)

ax.plot(data_df[plot_item])
```

```
[<matplotlib.lines.Line2D at 0x7fc3c591d910>]
```


35
30
25
20
15
10
5
0
goal
0
5
10
15
20
Iteration

CHAPTER

NINE

SENSITIVITY ANALYSIS

Another interesting study to understand if our dataset is indeed helpful in improving certain model parameters is to perform a Sensitivity Analysis. The purpose of this exercise is to scan the Model Parameters of interest (eg, qubit frequency or anharmonicity) across a range of values and notice a prominent dip in the Model Learning Goal Function around the best-fit values

```
run_name = "Sensitivity"
dir_path = "sensi_logs"
algorithm = "sweep"
options = {"points": 20, "init_point": [-210e6, 5e9]}
sweep_bounds = [
    [-215e6, -205e6],
    [4.9985e9, 5.0015e9],
]
```

```
sense_opt = Sensitivity(
    datafiles=datafiles,
    run_name=run_name,
    dir_path=dir_path,
    algorithm=algorithm,
    options=options,
    sampling=sampling,
    batch_sizes=batch_sizes,
    state_labels=state_labels,
    pmap=exp.pmap,
    sweep_bounds=sweep_bounds,
    sweep_map=exp_opt_map,
)

sense_opt.set_exp(exp)
```

```
sense_opt.run()
```

```
C3:STATUS:Sweeping [['Q1-anhar']]: [-215000000.0, -205000000.0]
C3:STATUS:Saving as: /home/users/anurag/c3/examples/sensi_logs/Sensitivity/2021_07_05_T_
↪20_56_46/sensitivity.log
C3:STATUS:Sweeping [['Q1-freq']]: [4998500000.0, 5001500000.0]
C3:STATUS:Saving as: /home/users/anurag/c3/examples/sensi_logs/Sensitivity/2021_07_05_T_
↪20_57_38/sensitivity.log
```

```
LOGDIR = sense_opt.logdir_list[0]
```

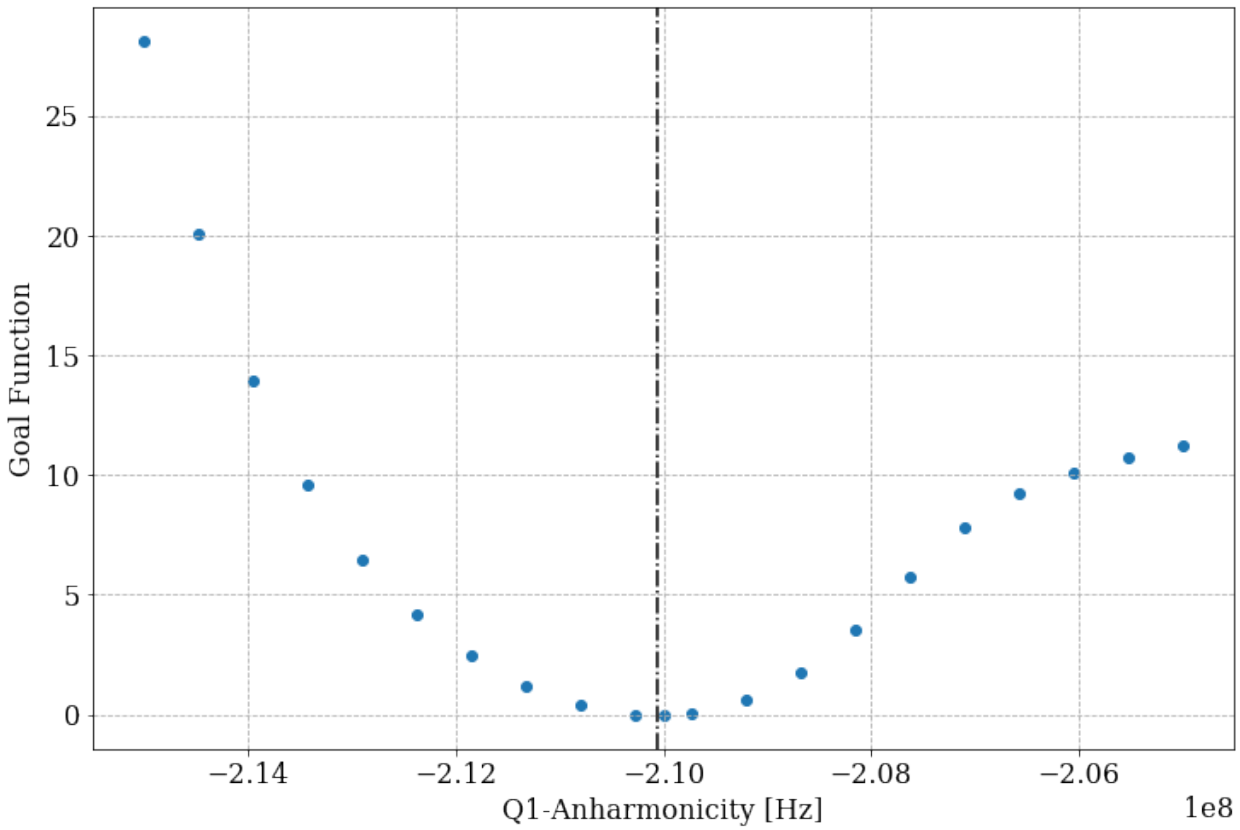
```
logfile = os.path.join(LOGDIR, "sensitivity.log")
with open(logfile, "r") as f:
    log = f.readlines()
```

```
data_list_dict = list()
for line in log[9:]:
    if line[0] == "{":
        temp_dict = ast.literal_eval(line.strip("\n"))
        param = temp_dict["params"][0]
        data_list_dict.append({"param": param, "goal": temp_dict["goal"]})
```

```
data_df = pd.DataFrame(data_list_dict)
```

```
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)
ax.set_xlabel("Q1-Anharmonicity [Hz]")
ax.set_ylabel("Goal Function")
ax.axvline(x=best_point_dict["Q1-anhar"], color="black", linestyle="-.")
ax.scatter(data_df["param"], data_df["goal"])
```

```
<matplotlib.collections.PathCollection at 0x7f917a341d30>
```



```
LOGDIR = sense_opt.logdir_list[1]
```

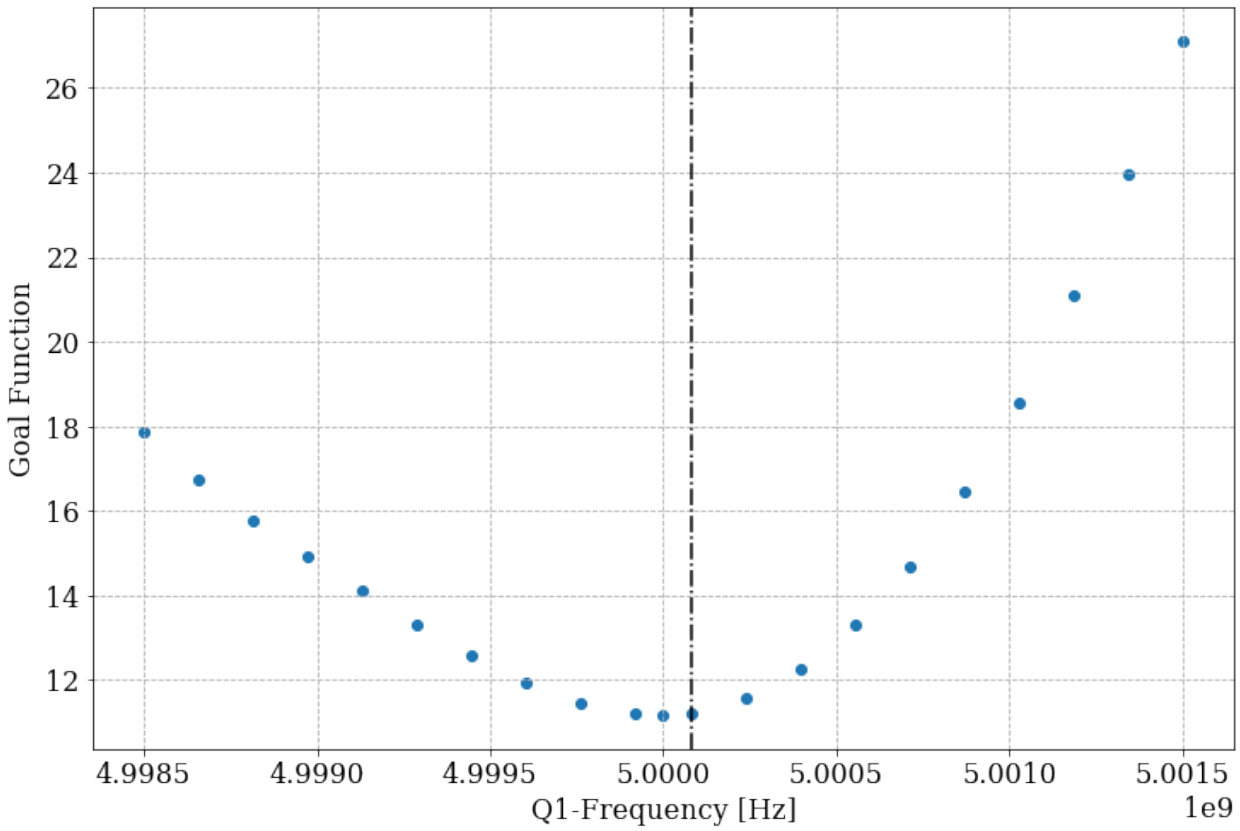
```
logfile = os.path.join(LOGDIR, "sensitivity.log")
with open(logfile, "r") as f:
    log = f.readlines()
```

```
data_list_dict = list()
for line in log[9:]:
    if line[0] == "{":
        temp_dict = ast.literal_eval(line.strip("\n"))
        param = temp_dict["params"][0]
        data_list_dict.append({"param": param, "goal": temp_dict["goal"]})
```

```
data_df = pd.DataFrame(data_list_dict)
```

```
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)
ax.set_xlabel("Q1-Frequency [Hz]")
ax.set_ylabel("Goal Function")
ax.axvline(x=best_point_dict["Q1-freq"], color="black", linestyle="-.")
ax.scatter(data_df["param"], data_df["goal"])
```

```
<matplotlib.collections.PathCollection at 0x7f917a203370>
```



CHAPTER

TEN

LOGS AND CURRENT OPTIMIZATION STATUS

During optimizations (optimal control, calibration, model learning), a current best point is stored in the log folder to monitor progress. Called on a log file it will print a rich table of the current status. With the `-w` or `-- watch` options the table will keep updating.

```
c3/utils/log_reader.py -h
```

```
usage: log_reader.py [-h] [-w WATCH] log_file

positional arguments:
  log_file

optional arguments:
  -h, --help            show this help message and exit
  -w WATCH, --watch WATCH
                        Update the table every WATCH seconds.
```

Using the example log from the test folder:

```
c3/utils/log_reader.py test/sample_optim_log.c3log
```

```
Optimization reached 0.00462 at Tue Aug 17 15:28:09 2021


 Parameter                       Value            Gradient

┌ rx90p[0]-d1-gauss-amp          │      497.311 mV │        18.720 mV │
│ rx90p[0]-d1-gauss-freq_offset  │ -52.998 MHz 2pi │ -414.237 µHz 2pi │
│ rx90p[0]-d1-gauss-xy_angle     │    -47.409 mrad │       2.904 mrad │
└ rx90p[0]-d1-gauss-delta        │          -1.077 │          6.648 m │
```

CHAPTER

ELEVEN

C3 SIMULATOR AS A BACKEND FOR QISKIT EXPERIMENTS

This notebook demonstrates the use of the C3 Simulator with a high-level quantum programming framework Qiskit. You must additionally install qiskit and matplotlib to run this example.

```
!pip install -q qiskit==0.25.0 matplotlib==3.3.4
```

```
import numpy as np
from c3.qiskit import C3Provider
from qiskit import transpile, execute, QuantumCircuit, Aer
from qiskit.tools.visualization import plot_histogram
```

11.1 Define a basic Quantum circuit

```
qc = QuantumCircuit(6, 6)
qc.rx(np.pi/2, 0)
qc.rx(np.pi/2, 1)
qc.measure([0, 1, 2, 3, 4, 5], [0, 1, 2, 3, 4, 5])
```

```
<qiskit.circuit.instructionset.InstructionSet at 0x7f08b3621280>
```

```
qc.draw()
```

11.2 Get the C3 Provider and Backend

```
c3_provider = C3Provider()
c3_backend = c3_provider.get_backend("c3_qasm_perfect_simulator")
```

```
config = c3_backend.configuration()

print("Name: {0}".format(config.backend_name))
print("Version: {0}".format(config.backend_version))
print("Max Qubits: {0}".format(config.n_qubits))
print("OpenPulse Support: {0}".format(config.open_pulse))
print("Basis Gates: {0}".format(config.basis_gates))
```

```
Name: c3_qasm_perfect_simulator
Version: 0.1
Max Qubits: 20
OpenPulse Support: False
Basis Gates: ['cx', 'cz', 'iSwap', 'id', 'x', 'y', 'z', 'rx', 'ry', 'rz', 'rzx']
```

11.2.1 Let's view how the Qiskit Transpiler will convert the circuit

```
trans_qc = transpile(qc, c3_backend)
```

```
trans_qc.draw()
```

11.3 Run an ideal device simulation using C3

```
c3_backend.set_device_config("quickstart.hjson")
c3_backend.disable_flip_labels()
c3_job = execute(qc, c3_backend, shots=1000)
result = c3_job.result()
```

```
res_counts = result.get_counts(qc)
print(res_counts)
```

```
{'000000': 250, '010000': 250, '100000': 250, '110000': 250}
```

```
plot_histogram(res_counts, title='C3 Perfect Devices simulation')
```

11.4 Run Simulation and verify results on Qiskit simulator

```
qiskit_simulator = Aer.get_backend('qasm_simulator')
qiskit_result = execute(qc, qiskit_simulator, shots=1000).result()
counts = qiskit_result.get_counts(qc)
plot_histogram(counts, title='Qiskit simulation')
```

CHAPTER

TWELVE

API DOCUMENTATION

12.1 C3objs

Basic custom objects.

class c3.c3objs.**C3obj**(*name*, *desc=''*, *comment=''*, *params=None*)
Bases: object

Represents an abstract object with parameters. To be inherited from.

Parameters

- **name** (*str*) – short name that will be used as identifier
- **desc** (*str*) – longer description of the component
- **comment** (*str*) – additional information about the component
- **params** (*dict*) – Parameters in this dict can be accessed and optimized

asdict() → dict

class c3.c3objs.**Quantity**(*value*, *unit='undefined'*, *min_val=None*, *max_val=None*, *symbol='\\alpha'*)
Bases: object

Represents any physical quantity used in the model or the pulse specification. For arithmetic operations just the numeric value is used. The value itself is stored in an optimizer friendly way as a float between -1 and 1. The conversion is given by

scale (value + 1) / 2 + offset

Parameters

- **value** (*np.array(np.float64) or np.float64*) – value of the quantity
- **min_val** (*np.array(np.float64) or np.float64*) – minimum this quantity is allowed to take
- **max_val** (*np.array(np.float64) or np.float64*) – maximum this quantity is allowed to take
- **unit** (*str*) – physical unit
- **symbol** (*str*) – latex representation

add(*val*)

asdict() → dict
Return a config-compatible dictionary representation.

get_limits()

get_opt_value() → numpy.ndarray
Get an optimizer friendly representation of the value.

get_value(*val: tf.float64 = None*, *dtype: <module 'tensorflow._api.v2.dtypes' from '/home/docs/checkouts/readthedocs.org/user_builds/c3-toolset/envs/master/lib/python3.7/site-packages/tensorflow/_api/v2/dtypes/__init__.py'> = None*) → tensorflow.python.framework.ops.Tensor
Return the value of this quantity as tensorflow.

Parameters

- **val** (*tf.float64*) –
- **dtype** (*tf.dtypes*) –

numpy() → numpy.ndarray
Return the value of this quantity as numpy.

set_limits(*min_val*, *max_val*)

set_opt_value(*val: float*) → None
Set value optimizer friendly.

Parameters **val** (*tf.float64*) – Tensorflow number that will be mapped to a value between -1 and 1.

set_value(*val*, *extend_bounds=False*)

subtract(*val*)

c3.c3objs.**hjson_decode**(*z*)

c3.c3objs.**hjson_encode**(*z*)

c3.c3objs.**jsonify_list**(*data*, *transform_arrays=True*)

12.2 Experiment module

Experiment class that models and simulates the whole experiment.

It combines the information about the model of the quantum device, the control stack and the operations that can be done on the device.

Given this information an experiment run is simulated, returning either processes, states or populations.

class c3.experiment.**Experiment**(*pmap: Optional[c3.parametermap.ParameterMap] = None*, *prop_method=None*)
Bases: object

It models all of the behaviour of the physical experiment, serving as a host for the individual parts making up the experiment.

Parameters **pmap** (*ParameterMap*) – including model: Model

The underlying physical device.

generator: Generator The infrastructure for generating and sending control signals to the device.

gateset: GateSet A gate level description of the operations implemented by control pulses.

asdict() → Dict
Return a dictionary compatible with config files.

compute_propagators()
Compute the unitary representation of operations. If no operations are specified in self.opt_gates the complete gateset is computed.

Returns A dictionary of gate names and their unitary representation.

Return type dict

compute_states() → Dict[c3.signal.gates.Instruction, List[tensorflow.python.framework.ops.Tensor]]
Employ a state solver to compute the trajectory of the system.

Returns List of states of the system from simulation.

Return type List[tf.tensor]

enable_qasm() → None
Switch the sequencing format to QASM. Will become the default.

evaluate_legacy(*sequences*)
Compute the population values for a given sequence of operations.

Parameters **sequences** (*str list*) – A list of control pulses/gates to perform on the device.

Returns A list of populations

Return type list

evaluate_qasm(*sequences*)
Compute the population values for a given sequence (in QASM format) of operations.

Parameters **sequences** (*dict list*) – A list of control pulses/gates to perform on the device in QASM format.

Returns A list of populations

Return type list

expect_oper(*state*, *lindbladian*, *oper*)

from_dict(*cfg: Dict*) → None
Load experiment from dictionary

get_VZ(*target*, *params*)
Returns the appropriate Z-rotation.

get_perfect_gates(*gate_keys: Optional[list] = None*) → Dict[str, numpy.ndarray]
Return a perfect gateset for the gate_keys.

Parameters **gate_keys** (*list*) – (Optional) List of gates to evaluate.

Returns A dictionary of gate names and np.array representation of the corresponding unitary

Return type Dict[str, np.array]

Raises **Exception** – Raise general exception for undefined gate

load_quick_setup(*filepath: str*) → None
Load a quick setup file.

Parameters **filepath** (*str*) – Location of the configuration file

lookup_gate(*name*, *qubits*, *params=None*) → tensorflow.python.framework.constant_op.constant
Returns a fixed operation or a parametric virtual Z gate. To be extended to general parametric gates.

populations(*state*, *lindbladian*)
Compute populations from a state or density vector.

Parameters

- **state** (*tf.Tensor*) – State or densitiy vector.
- **lindbladian** (*boolean*) – Specify if conversion to density matrix is needed.

Returns Vector of populations.

Return type tf.Tensor

process(*populations*, *labels=None*)
Apply a readout procedure to a population vector. Very specialized at the moment.

Parameters

- **populations** (*list*) – List of populations from evaluating.
- **labels** (*list*) – List of state labels specifying a subspace.

Returns A list of processed populations.

Return type list

quick_setup(*cfg*) → None
Load a quick setup cfg and create all necessary components.

Parameters **cfg** (*Dict*) – Configuration options

read_config(*filepath: str*) → None
Load a file and parse it to create a Model object.

Parameters **filepath** (*str*) – Location of the configuration file

set_created_by(*config*)
Store the config file location used to created this experiment.

set_enable_store_unitaries(*flag*, *logdir*, *exist_ok=False*)
Saving of unitary propagators.

Parameters

- **flag** (*boolean*) – Enable or disable saving.
- **logdir** (*str*) – File path location for the resulting unitaries.

set_opt_gates(*gates*)
Specify a selection of gates to be computed.

Parameters **gates** (*Identifiers of the gates of interest. Can contain duplicates.*) –

set_opt_gates_seq(*seqs*)
Specify a selection of gates to be computed.

Parameters **seqs** (*Identifiers of the sequences of interest. Can contain duplicates.*) –

set_prop_method(*prop_method=None*) → None
Configure the selected propagation method by either linking the function handle or looking it up in the library.

store_Udict(*goal*)
Save unitary as text and pickle.

goal: tf.float64 Value of the goal function, if used.

write_config(*filepath: str*) → None
Write dictionary to a HJSON file.

12.3 Model module

The model class, containing information on the system and its modelling.

class c3.model.**Model**(*subsystems=None*, *couplings=None*, *tasks=None*, *max_excitations=0*)
Bases: object

What the theorist thinks about from the system.

Class to store information about our system/problem/device. Different models can represent the same system.

Parameters

- **subsystems** (*list*) – List of individual, non-interacting physical components like qubits or resonators
- **couplings** (*list*) – List of interaction operators between subsystems, like couplings or drives.
- **tasks** (*list*) – Badly named list of processing steps like line distortions and read out modeling
- **max_excitations** (*int*) – Allow only up to max_excitations in the system

asdict() → dict
Return a dictionary compatible with config files.

blowup_excitations(*op*)

cut_excitations(*op*)

fromdict(*cfg: dict*) → None
Load a file and parse it to create a Model object.

Parameters **cfg** (*dict*) – configuration file

get_Frame_Rotation(*t_final: numpy.float64*, *freqs: dict*, *framechanges: dict*)
Compute the frame rotation needed to align Lab frame and rotating Eigenframes of the qubits.

Parameters

- **t_final** (*tf.float64*) – Gate length
- **freqs** (*list*) – Frequencies of the local oscillators.
- **framechanges** (*list*) – List of framechanges. A phase shift applied to the control signal to compensate relative phases of drive oscillator and qubit.

Returns A (diagonal) propagator that adjust phases

Return type tf.Tensor

get_Hamiltonian(*signal=None*)
Get a hamiltonian with an optional signal. This will return an hamiltonian over time. Can be used e.g. for tuning the frequency of a transmon, where the control hamiltonian is not easily accessible. If max.excitation is non-zero the resulting Hamiltonian is cut accordingly

get_Hamiltonians()

get_Lindbladians()

get_dephasing_channel(*t_final*, *amps*)
Compute the matrix of the dephasing channel to be applied on the operation.

Parameters

- **t_final** (*tf.float64*) – Duration of the operation.
- **amps** (*dict of tf.float64*) – Dictionary of average amplitude on each drive line.

Returns Matrix representation of the dephasing channel.

Return type tf.tensor

get_ground_state() → tensorflow.python.framework.constant_op.constant

get_qubit_freqs() → List[float]

get_sparse_Hamiltonian(*signal=None*)

get_sparse_Hamiltonians()

get_state_indeces(*states: List[Tuple]*) → List[int]

get_state_index(*state: Tuple*) → int

list_parameters()

read_config(*filepath: str*) → None
Load a file and parse it to create a Model object.

Parameters **filepath** (*str*) – Location of the configuration file

set_FR(*use_FR*)
Setter for the frame rotation option for adjusting the individual rotating frames of qubits when using gate sequences

set_components(*subsystems*, *couplings=None*, *max_excitations=0*) → None

set_dephasing_strength(*dephasing_strength*)

set_dressed(*dressed*)
Go to a dressed frame where static couplings have been eliminated.

Parameters **dressed** (*boolean*) –

set_lindbladian(*lindbladian*)
Set whether to include open system dynamics.

Parameters **lindbladian** (*boolean*) –

set_max_excitations(*max_excitations*) → None
Set the maximum number of excitations in the system used for propagation.

set_tasks(*tasks*) → None

update_Hamiltonians()
Recompute the matrix representations of the Hamiltonians.

update_Lindbladians()
Return Lindbladian operators and their prefactors.

update_dressed(*ordered=True*)
Compute the Hamiltonians in the dressed basis by diagonalizing the drift and applying the resulting transformation to the control Hamiltonians.

update_drift_eigen(*ordered=True*)
Compute the eigendecomposition of the drift Hamiltonian and store both the Eigenenergies and the transformation matrix.

update_model(*ordered=True*)

write_config(*filepath: str*) → None
Write dictionary to a HJSON file.

12.4 Parameter map

ParameterMap class

class c3.parametermap.**ParameterMap**(*instructions: List[c3.signal.gates.Instruction] = []*, *generator=None*, *model=None*)

Bases: `object`

Collects information about control and model parameters and provides different representations depending on use.

asdict(*instructions_only=True*) → dict
Return a dictionary compatible with config files.

check_limits(*opt_map*)
Check if all elements of equal ids have the same limits. This has to be checked against if setting values optimizer friendly.
Parameters **opt_map** –

fromdict(*cfg: dict*) → None

get_full_params() → Dict[str, c3.c3objs.Quantity]
Returns the full parameter vector, including model and control parameters.

get_key_from_scaled_index(*idx*, *opt_map=None*) → str
Get the key of the value at position *idx* of the scaled_parameters output :param idx: :param opt_map:

get_not_opt_params(*opt_map=None*) → Dict[str, c3.c3objs.Quantity]

get_opt_limits()

get_opt_map(*opt_map=None*) → List[List[str]]

get_opt_units() → List[str]
Returns a list of the units of the optimized quantities.

get_parameter(*par_id: Tuple[str, ...]*) → c3.c3objs.Quantity
Return one the current parameters.
Parameters **par_id** (*tuple*) – Hierarchical identifier for parameter.
Returns
Return type Quantity

get_parameter_dict(*opt_map=None*) → Dict[str, c3.c3objs.Quantity]
Return the current parameters in a dictionary including keys. :param opt_map:
Returns
Return type Dictionary with Quantities

get_parameters(*opt_map=None*) → List[c3.c3objs.Quantity]
Return the current parameters.
Parameters **opt_map** (*list*) – Hierarchical identifier for parameters.

Returns
Return type list of Quantity

get_parameters_scaled(*opt_map=None*) → numpy.ndarray
Return the current parameters. This fuction should only be called by an optimizer. Are you an optimizer?
Parameters **opt_map** (*tuple*) – Hierarchical identifier for parameters.
Returns
Return type list of Quantity

load_values(*init_point*)
Load a previous parameter point to start the optimization from.
Parameters **init_point** (*str*) – File location of the initial point

print_parameters(*opt_map=None*) → None
Print current parameters to stdout.

read_config(*filepath: str*) → None
Load a file and parse it to create a ParameterMap object.
Parameters **filepath** (*str*) – Location of the configuration file

set_opt_map(*opt_map*) → None
Set the opt_map, i.e. which parameters will be optimized.

set_parameters(*values: Union[List, numpy.ndarray]*, *opt_map=None*, *extend_bounds=False*) → None
Set the values in the original instruction class.
Parameters
- **values** (*list*) – List of parameter values. Can be nested, if a parameter is matrix valued.
- **opt_map** (*list*) – Corresponding identifiers for the parameter values.
- **extend_bounds** (*bool*) – If true bounds of quantity objects will be extended.

set_parameters_scaled(*values: Union[tensorflow.python.framework.constant_op.constant, tensorflow.python.ops.variables.Variable]*, *opt_map=None*) → None
Set the values in the original instruction class. This fuction should only be called by an optimizer. Are you an optimizer?
Parameters **values** (*list*) – List of parameter values. Matrix valued parameters need to be flattened.

store_values(*path: str*, *optim_status=None*) → None
Write current parameter values to file. Stores the numeric values, as well as the names in form of the opt_map and physical units. If an optim_status is given that will be used.
Parameters
- **path** (*str*) – Location of the resulting logfile.
- **optim_status** (*dict*) – Dictionary containing current parameters and goal function value.

str_parameters(*opt_map: Optional[Union[List[List[Tuple[str]]], List[List[str]]]] = None*) → str
Return a multi-line human-readable string of the optmization parameter names and current values.
Parameters **opt_map** (*list*) – Optionally use only the specified parameters.
Returns Parameters and their values
Return type str

update_parameters()

write_config(*filepath: str*) → None
Write dictionary to a HJSON file.

12.5 Main module

Base script to run the C3 code from a main config file.

c3.main.**run_cfg**(*cfg*, *opt_config_filename*, *debug=False*)
Execute an optimization problem described in the cfg file.

Parameters

- **cfg** (*Dict[str, Union[str, int, float]]*) – Configuration file containing optimization options and information needed to completely setup the system and optimization problem.
- **debug** (*bool, optional*) – Skip running the actual optimization, by default False

12.6 Module contents

12.7 Subpackages

12.7.1 Generator package

12.7.1.1 Submodules

12.7.1.2 Devices module

class c3.generator.devices.**AWG**(***props*)
Bases: c3.generator.devices.Device

AWG device, transforms digital input to analog signal.

Parameters **logdir** (*str*) – Filepath to store generated waveforms.

asdict() → dict

create_IQ(*instr: c3.signal.gates.Instruction*, *chan: str*) → dict
Construct the in-phase (I) and quadrature (Q) components of the signal. These are universal to either experiment or simulation. In the xperiment these will be routed to AWG and mixer electronics, while in the simulation they provide the shapes of the instruction fields to be added to the Hamiltonian.

Parameters

- **channel** (*str*) – Identifier for the selected drive line.
- **components** (*dict*) – Separate signals to be combined onto this drive line.
- **t_start** (*float*) – Beginning of the signal.
- **t_end** (*float*) – End of the signal.

Returns Waveforms as I and Q components.

Return type dict

create_IQ_pwc(*instr: c3.signal.gates.Instruction*, *chan: str*) → dict
Construct the in-phase (I) and quadrature (Q) components of the signal. These are universal to either experiment or simulation. In the xperiment these will be routed to AWG and mixer electronics, while in the simulation they provide the shapes of the instruction fields to be added to the Hamiltonian.

Parameters

- **channel** (*str*) – Identifier for the selected drive line.
- **components** (*dict*) – Separate signals to be combined onto this drive line.
- **t_start** (*float*) – Beginning of the signal.
- **t_end** (*float*) – End of the signal.

Returns Waveforms as I and Q components.

Return type dict

enable_drag()

enable_drag_2()

enable_pwc()

get_I(*line*)

get_Q(*line*)

get_average_amp(*line*)
Compute average and sum of the amplitudes. Used to estimate effective drive power for non-trivial shapes.

Returns Average and sum.

Return type tuple

log_shapes()

class c3.generator.devices.Additive_Noise(***props*)
Bases: c3.generator.devices.Device

Noise applied to a signal

get_noise(*sig*)

process(*instr*, *chan*, *signal*)
Distort signal by adding noise.

class c3.generator.devices.Crosstalk(***props*)
Bases: c3.generator.devices.Device

Device to phenomenologically include crosstalk in the model by explicitly mixing drive lines.

Parameters **crosstalk_matrix** (*tf.constant*) – Matrix description of how to mix drive channels.

Examples

```
xtalk = Crosstalk(
    name="crosstalk",
    channels=["TC1", "TC2"],
    crosstalk_matrix=Quantity(
        value=[[1, 0], [0, 1]],
        min_val=[[0, 0], [0, 0]],
        max_val=[[1, 1], [1, 1]],
        unit="",
    ),
)
```

process(*signal: Dict[str, Any]*) → Dict[str, Any]
Mix channels in the input signal according to a crosstalk matrix.

Parameters **signal** (*Dict[str, Any]*) – Dictionary of several signals identified by their channel as dict keys, e.g.

```
signal = {
    "TC1": {"values": [0, 0.5, 1, 1, ...]},
    "TC2": {"values": [1, 1, 1, 1, ...],
}
```

Returns **signal**

Return type Dict[str, Any]

class c3.generator.devices.**DC_Noise**(***props*)
Bases: c3.generator.devices.Additive_Noise

Add a random constant offset to the signals

get_noise(*sig*)

class c3.generator.devices.**DC_Offset**(***props*)
Bases: c3.generator.devices.Device

Noise applied to a signal

process(*instr*, *chan*, *signal*)
Distort signal by adding noise.

class c3.generator.devices.**Device**(***props*)
Bases: c3.c3objs.C3obj

A Device that is part of the stack generating the instruction signals.

Parameters **resolution** (*np.float64*) – Number of samples per second this device operates at.

asdict() → Dict[str, Any]

calc_slice_num(*t_start: numpy.float64*, *t_end: numpy.float64*) → None
Effective number of time slices given start, end and resolution.

Parameters

- **t_start** (*np.float64*) – Starting time for this device.
- **t_end** (*np.float64*) – End time for this device.

create_ts(*t_start: numpy.float64*, *t_end: numpy.float64*, *centered: bool = True*) → tensorflow.python.framework.constant_op.constant
Compute time samples.

Parameters

- **t_start** (*np.float64*) – Starting time for this device.
- **t_end** (*np.float64*) – End time for this device.
- **centered** (*boolean*) – Sample in the middle of an interval, otherwise at the beginning.

write_config(*filepath: str*) → None
Write dictionary to a HJSON file.

class c3.generator.devices.**DigitalToAnalog**(***props*)
Bases: c3.generator.devices.Device

Take the values at the awg resolution to the simulation resolution.

process(*instr: c3.signal.gates.Instruction*, *chan: str*, *awg_signal: Dict[str, Any]*) → Dict[str, Any]
Resample the awg values to higher resolution.

Parameters

- **instr** (*Instruction*) – The logical instruction or qubit operation for which the signal is generated.
- **chan** (*str*) – Specifies which channel is being processed if needed.
- **awg_signal** (*dict*) – Dictionary of several signals identified by their channel as dict keys.

Returns Inphase and Quadrature compontent of the upsampled signal.

Return type dict

class c3.generator.devices.**ExponentialIIR**(***props*)
Bases: c3.generator.devices.StepFuncFilter

Implement IIR filter with step response of the form s(t) = (1 + A * exp(-t / t_iir))

Parameters

- **time_iir** (*Quantity*) – Time constant for the filtering.
- **amp** (*Quantity*) –

step_response_function(*ts*)

class c3.generator.devices.**Filter**(***props*)
Bases: c3.generator.devices.Device

Apply a filter function to the signal.

process(*instr: c3.signal.gates.Instruction*, *chan: str*, *Hz_signal: Dict[str, Any]*) → Dict[str, Any]
Apply a filter function to the signal.

class c3.generator.devices.**FluxTuning**(***props*)
Bases: c3.generator.devices.Device

Flux tunable qubit frequency.

Parameters

- **phi_0** (*Quantity*) – Flux bias.
- **phi** (*Quantity*) – Current flux.

- **omega_0** (*Quantity*) – Maximum frequency.

get_factor(*phi*)

get_freq(*phi*)

process(*instr: c3.signal.gates.Instruction*, *chan: str*, *signal_in*)
Compute the qubit frequency resulting from an applied flux.

Parameters **signal** (*tf.float64*) –

Returns Qubit frequency.

Return type tf.float64

class c3.generator.devices.**FluxTuningLinear**(***props*)
Bases: c3.generator.devices.Device

Flux tunable qubit frequency linear adjustment.

Parameters

- **phi_0** (*Quantity*) – Flux bias.
- **Phi** (*Quantity*) – Current flux.
- **omega_0** (*Quantity*) – Maximum frequency.

frequency(*signal: tf.float64*) → tensorflow.python.framework.constant_op.constant
Compute the qubit frequency resulting from an applied flux.

Parameters **signal** (*tf.float64*) –

Returns Qubit frequency.

Return type tf.float64

class c3.generator.devices.**HighpassExponential**(***props*)
Bases: c3.generator.devices.StepFuncFilter

Implement Highpass filter based on exponential with step response of the form s(t) = exp(-t / t_hp)

Parameters

- **time_iir** (*Quantity*) – Time constant for the filtering.
- **amp** (*Quantity*) –

step_response_function(*ts*)

class c3.generator.devices.**HighpassFilter**(***props*)
Bases: c3.generator.devices.Device

Introduce a highpass filter

Parameters

- **cutoff** (*Quantity*) – cutoff frequency of highpass filter
- **keep_mean** (*bool*) – should the mean of the signal be restored

convolve(*signal: list*, *resp_shape: list*)
Compute the convolution with a function.

Parameters

- **signal** (*list*) – Potentially unlimited signal samples.
- **resp_shape** (*list*) – Samples of the function to model limited bandwidth.

Returns Processed signal.

Return type tf.Tensor

process(*instr*, *chan*, *iq_signal*)
Apply a highpass cutoff to an IQ signal.

Parameters **iq_signal** (*dict*) – I and Q components of an AWG signal.

Returns Filtered IQ signal.

Return type dict

class c3.generator.devices.**LO**(***props*)
Bases: c3.generator.devices.Device

Local oscillator device, generates a constant oscillating signal.

process(*instr: c3.signal.gates.Instruction*, *chan: str*) → dict

class c3.generator.devices.**LONoise**(***props*)
Bases: c3.generator.devices.Device

Noise applied to the local oscillator

process(*instr*, *chan*, *lo_signal*)
Distort signal by adding noise.

class c3.generator.devices.**Mixer**(***props*)
Bases: c3.generator.devices.Device

Mixer device, combines inputs from the local oscillator and the AWG.

process(*instr: c3.signal.gates.Instruction*, *chan: str*, *in1: dict*, *in2: dict*)
Combine signal from AWG and LO.

Parameters

- **lo_signal** (*dict*) – Local oscillator signal.
- **awg_signal** (*dict*) – Waveform generator signal.

Returns Mixed signal.

Return type dict

class c3.generator.devices.**Pink_Noise**(***props*)
Bases: c3.generator.devices.Additive_Noise

Device creating pink noise, i.e. 1/f noise.

get_noise(*sig*)

class c3.generator.devices.**Readout**(***props*)
Bases: c3.generator.devices.Device

Mimic the readout process by multiplying a state phase with a factor and offset.

Parameters

- **factor** (*Quantity*) –
- **offset** (*Quantity*) –

readout(*phase*)
Apply the readout rescaling

Parameters **phase** (*tf.float64*) – Raw phase of a quantum state

Returns Rescaled readout value

Return type tf.float64

class c3.generator.devices.**Response**(***props*)
Bases: c3.generator.devices.Device

Make the AWG signal physical by convolution with a Gaussian to limit bandwith.

Parameters **rise_time** (*Quantity*) – Time constant for the gaussian convolution.

convolve(*signal: list*, *resp_shape: list*)
Compute the convolution with a function.

Parameters

- **signal** (*list*) – Potentially unlimited signal samples.
- **resp_shape** (*list*) – Samples of the function to model limited bandwidth.

Returns Processed signal.

Return type tf.Tensor

process(*instr*, *chan*, *iq_signal*)
Apply a Gaussian shaped limiting function to an IQ signal.

Parameters **iq_signal** (*dict*) – I and Q components of an AWG signal.

Returns Bandwidth limited IQ signal.

Return type dict

class c3.generator.devices.**ResponseFFT**(***props*)
Bases: c3.generator.devices.Device

Make the AWG signal physical by convolution with a Gaussian to limit bandwith.

Parameters **rise_time** (*Quantity*) – Time constant for the gaussian convolution.

process(*instr*, *chan*, *iq_signal*)
Apply a Gaussian shaped limiting function to an IQ signal.

Parameters **iq_signal** (*dict*) – I and Q components of an AWG signal.

Returns Bandwidth limited IQ signal.

Return type dict

class c3.generator.devices.**SkinEffectResponse**(***props*)
Bases: c3.generator.devices.StepFuncFilter

Implement Highpass filter based on exponential with step response of the form s(t) = exp(-t / t_hp)

Parameters

- **time_iir** (*Quantity*) – Time constant for the filtering.
- **amp** (*Quantity*) –

step_response_function(*ts*)

class c3.generator.devices.**StepFuncFilter**(***props*)
Bases: c3.generator.devices.Device

Base class for filters that are based on the step response function Step function has to be defined explicetly

process(*instr*, *chan*, *signal_in*)

step_response_function(*ts*)

class c3.generator.devices.**VoltsToHertz**(***props*)
Bases: c3.generator.devices.Device

Convert the voltage signal to an amplitude to plug into the model Hamiltonian.

Parameters

- **V_to_Hz** (*Quantity*) – Conversion factor.
- **offset** (*tf.float64*) – Drive frequency offset.

process(*instr: c3.signal.gates.Instruction*, *chan: str*, *mixed_signal: Dict[str, Any]*) → Dict[str, Any]
Transform signal from value of V to Hz.

Parameters **mixed_signal** (*tf.Tensor*) – Waveform as line voltages after IQ mixing

Returns Waveform as control amplitudes

Return type tf.Tensor

c3.generator.devices.**dev_reg_deco**(*func: Callable*) → Callable
Decorator for making registry of functions

12.7.1.3 Generator module

Signal generation stack.

Contrary to most quanutm simulators, C^3 includes a detailed simulation of the control stack. Each component in the stack and its functions are simulated individually and combined here.

Example: A local oscillator and arbitrary waveform generator signal are put through via a mixer device to produce an effective modulated signal.

class c3.generator.generator.**Generator**(*devices: Optional[dict] = None*, *chains: Optional[dict] = None*, *resolution: numpy.float64 = 0.0*, *callback: Optional[Callable] = None*)
Bases: object

Generator, creates signal from digital to what arrives to the chip.

Parameters

- **devices** (*list*) – Physical or abstract devices in the signal processing chain.
- **resolution** (*np.float64*) – Resolution at which continuous functions are sampled.
- **callback** (*Callable*) – Function that is called after each device in the signal line.

asdict() → dict
Return a dictionary compatible with config files.

fromdict(*cfg: dict*) → None

generate_signals(*instr: c3.signal.gates.Instruction*) → dict
Perform the signal chain for a specified instruction, including local oscillator, AWG generation and IQ mixing.

Parameters **instr** (*Instruction*) – Operation to be performed, e.g. logical gate.

Returns Signal to be applied to the physical device.

Return type dict

read_config(*filepath: str*) → None
Load a file and parse it to create a Generator object.

Parameters **filepath** (*str*) – Location of the configuration file

write_config(*filepath: str*) → None
Write dictionary to a HJSON file.

12.7.1.4 Module contents

12.7.2 Libraries package

Libraries contain a collection of functions that all share a signature to be used interchangably. One entry of a library is selected in the corresponding config file.

12.7.2.1 Algorithms module

Collection of (optimization) algorithms. All entries share a common signature with optional arguments.

c3.libraries.algorithms.**adaptive_scan**(*x_init*, *fun=None*, *fun_grad=None*, *grad_lookup=None*, *options={}*)
One dimensional scan of the function values around the initial point, using adaptive sampling

Parameters

- **x_init** (*float*) – Initial point
- **fun** (*callable*) – Goal function
- **fun_grad** (*callable*) – Function that computes the gradient of the goal function
- **grad_lookup** (*callable*) – Lookup a previously computed gradient
- **options** (*dict*) – Options include

  accuracy_goal: float Targeted accuracy for the sampling algorithm

  probe_list [list] Points to definitely include in the sampling

  init_point [boolean] Include the initial point in the sampling

c3.libraries.algorithms.**algo_reg_deco**(*func*)
Decorator for making registry of functions

c3.libraries.algorithms.**cma_pre_lbfgs**(*x_init*, *fun=None*, *fun_grad=None*, *grad_lookup=None*, *options={}*)
Performs a CMA-Es optimization and feeds the result into LBFG-S for further refinement.

c3.libraries.algorithms.**cmaes**(*x_init*, *fun=None*, *fun_grad=None*, *grad_lookup=None*, *options={}*)
Wrapper for the pycma implementation of CMA-Es. See also:

http://cma.gforge.inria.fr/apidocs-pycma/

Parameters

- **x_init** (*float*) – Initial point.
- **fun** (*callable*) – Goal function.
- **fun_grad** (*callable*) – Function that computes the gradient of the goal function.
- **grad_lookup** (*callable*) – Lookup a previously computed gradient.

- **options** (*dict*) – Options of pycma and the following custom options.

  noise [float] Artificial noise added to a function evaluation.

  init_point [boolean] Force the use of the initial point in the first generation.

  spread [float] Adjust the parameter spread of the first generation cloud.

  stop_at_convergence [int] Custom stopping condition. Stop if the cloud shrunk for this number of generations.

  stop_at_sigma [float] Custom stopping condition. Stop if the cloud shrunk to this standard deviation.

Returns Parameters of the best point.

Return type np.ndarray

c3.libraries.algorithms.**gcmaes**(*x_init*, *fun=None*, *fun_grad=None*, *grad_lookup=None*, *options={}*)
EXPERIMENTAL CMA-Es where every point in the cloud is optimized with LBFG-S and the resulting cloud and results are used for the CMA update.

c3.libraries.algorithms.**grid2D**(*x_init*, *fun=None*, *fun_grad=None*, *grad_lookup=None*, *options={}*)
Two dimensional scan of the function values around the initial point.

Parameters

- **x_init** (*float*) – Initial point
- **fun** (*callable*) – Goal function
- **fun_grad** (*callable*) – Function that computes the gradient of the goal function
- **grad_lookup** (*callable*) – Lookup a previously computed gradient
- **options** (*dict*) – Options include points : int

  The number of samples

  bounds [list] Range of the scan for both dimensions

c3.libraries.algorithms.**lbfgs**(*x_init*, *fun=None*, *fun_grad=None*, *grad_lookup=None*, *options={}*)
Wrapper for the scipy.optimize.minimize implementation of LBFG-S. See also:

https://docs.scipy.org/doc/scipy/reference/optimize.minimize-lbfgsb.html

Parameters

- **x_init** (*float*) – Initial point
- **fun** (*callable*) – Goal function
- **fun_grad** (*callable*) – Function that computes the gradient of the goal function
- **grad_lookup** (*callable*) – Lookup a previously computed gradient
- **options** (*dict*) – Options of scipy.optimize.minimize

Returns Scipy result object.

Return type Result

c3.libraries.algorithms.**lbfgs_grad_free**(*x_init*, *fun=None*, *fun_grad=None*, *grad_lookup=None*, *options={}*)
Wrapper for the scipy.optimize.minimize implementation of LBFG-S. We let the algorithm determine the gradient by its own.

See also:

https://docs.scipy.org/doc/scipy/reference/optimize.minimize-lbfgsb.html

Parameters

- **x_init** (*float*) – Initial point
- **fun** (*callable*) – Goal function
- **fun_grad** (*callable*) – Function that computes the gradient of the goal function
- **grad_lookup** (*callable*) – Lookup a previously computed gradient
- **options** (*dict*) – Options of scipy.optimize.minimize

Returns Scipy result object.

Return type Result

c3.libraries.algorithms.**oneplusone**(*x_init*, *goal_fun*)

c3.libraries.algorithms.**single_eval**(*x_init*, *fun=None*, *fun_grad=None*, *grad_lookup=None*, *options={}*)
Return the function value at given point.

Parameters

- **x_init** (*float*) – Initial point
- **fun** (*callable*) – Goal function
- **fun_grad** (*callable*) – Function that computes the gradient of the goal function
- **grad_lookup** (*callable*) – Lookup a previously computed gradient
- **options** (*dict*) – Algorithm specific options

c3.libraries.algorithms.**sweep**(*x_init*, *fun=None*, *fun_grad=None*, *grad_lookup=None*, *options={}*)
One dimensional scan of the function values around the initial point.

Parameters

- **x_init** (*float*) – Initial point
- **fun** (*callable*) – Goal function
- **fun_grad** (*callable*) – Function that computes the gradient of the goal function
- **grad_lookup** (*callable*) – Lookup a previously computed gradient
- **options** (*dict*) – Options include points : int

  The number of samples

  bounds [list] Range of the scan

c3.libraries.algorithms.**tf_adadelta**(*x_init: numpy.ndarray*, *fun: Optional[Callable] = None*, *fun_grad: Optional[Callable] = None*, *grad_lookup: Optional[Callable] = None*, *options: dict = {}*) → scipy.optimize.optimize.OptimizeResult
Optimize using TensorFlow Adadelta https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adadelta

Parameters

- **x_init** (*np.ndarray*) – starting value of parameter(s)
- **fun** (*Callable, optional*) – function to minimize, by default None
- **fun_grad** (*Callable, optional*) – gradient of function to minimize, by default None

- **grad_lookup** (*Callable, optional*) – lookup stored gradients, by default None
- **options** (*dict, optional*) – optional parameters for optimizer, by default {}

Returns SciPy OptimizeResult type object with final parameters

Return type OptimizeResult

c3.libraries.algorithms.**tf_adam**(*x_init: numpy.ndarray*, *fun: Optional[Callable] = None*, *fun_grad: Optional[Callable] = None*, *grad_lookup: Optional[Callable] = None*, *options: dict = {}*) → scipy.optimize.optimize.OptimizeResult

Optimize using TensorFlow ADAM https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam

Parameters

- **x_init** (*np.ndarray*) – starting value of parameter(s)
- **fun** (*Callable, optional*) – function to minimize, by default None
- **fun_grad** (*Callable, optional*) – gradient of function to minimize, by default None
- **grad_lookup** (*Callable, optional*) – lookup stored gradients, by default None
- **options** (*dict, optional*) – optional parameters for optimizer, by default {}

Returns SciPy OptimizeResult type object with final parameters

Return type OptimizeResult

c3.libraries.algorithms.**tf_rmsprop**(*x_init: numpy.ndarray*, *fun: Optional[Callable] = None*, *fun_grad: Optional[Callable] = None*, *grad_lookup: Optional[Callable] = None*, *options: dict = {}*) → scipy.optimize.optimize.OptimizeResult

Optimize using TensorFlow RMSProp https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/RMSprop

Parameters

- **x_init** (*np.ndarray*) – starting value of parameter(s)
- **fun** (*Callable, optional*) – function to minimize, by default None
- **fun_grad** (*Callable, optional*) – gradient of function to minimize, by default None
- **grad_lookup** (*Callable, optional*) – lookup stored gradients, by default None
- **options** (*dict, optional*) – optional parameters for optimizer, by default {}

Returns SciPy OptimizeResult type object with final parameters

Return type OptimizeResult

c3.libraries.algorithms.**tf_sgd**(*x_init: numpy.ndarray*, *fun: Optional[Callable] = None*, *fun_grad: Optional[Callable] = None*, *grad_lookup: Optional[Callable] = None*, *options: dict = {}*) → scipy.optimize.optimize.OptimizeResult

Optimize using TensorFlow Stochastic Gradient Descent with Momentum https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD

Parameters

- **x_init** (*np.ndarray*) – starting value of parameter(s)
- **fun** (*Callable, optional*) – function to minimize, by default None
- **fun_grad** (*Callable, optional*) – gradient of function to minimize, by default None
- **grad_lookup** (*Callable, optional*) – lookup stored gradients, by default None
- **options** (*dict, optional*) – optional parameters for optimizer, by default {}

Returns SciPy OptimizeResult type object with final parameters

Return type OptimizeResult

12.7.2.2 Chip module

Component class and subclasses for the components making up the quantum device.

class c3.libraries.chip.**CShuntFluxQubit**(*name: str*, *desc: Optional[str] = None*, *comment: Optional[str] = None*, *hilbert_dim: Optional[int] = None*, *calc_dim: Optional[int] = None*, *EC: Optional[c3.c3objs.Quantity] = None*, *EJ: Optional[c3.c3objs.Quantity] = None*, *EL: Optional[c3.c3objs.Quantity] = None*, *phi: Optional[c3.c3objs.Quantity] = None*, *phi_0: Optional[c3.c3objs.Quantity] = None*, *gamma: Optional[c3.c3objs.Quantity] = None*, *d: Optional[c3.c3objs.Quantity] = None*, *t1: Optional[c3.c3objs.Quantity] = None*, *t2star: Optional[c3.c3objs.Quantity] = None*, *temp: Optional[c3.c3objs.Quantity] = None*, *anhar: Optional[c3.c3objs.Quantity] = None*, *params={}*, *resolution=None*)

Bases: c3.libraries.chip.Qubit

get_Hamiltonian(*signal: Optional[Union[dict, bool]] = None*, *transform: Optional[tensorflow.python.framework.ops.Tensor] = None*) → tensorflow.python.framework.ops.Tensor

Calculate the hamiltonian :returns: Hamiltonian :rtype: tf.Tensor

get_anharmonicity(*phi_sig=0*)

get_freq(*phi_sig=0*)

get_frequency(*phi_sig=0*)

get_minimum_phi_var(*init_phi_variable: tf.float64 = 0*, *phi_sig=0*)

get_potential_function(*phi_variable*, *deriv_order=1*, *phi_sig=0*)

get_third_order_prefactor(*phi_sig=0*)

init_Hs(*ann_oper*)

initialize Hamiltonians for cubic hamiltinian :param ann_oper: Annihilation operator in the full Hilbert space :type ann_oper: np.array

class c3.libraries.chip.**CShuntFluxQubitCos**(*name: str*, *desc: Optional[str] = None*, *comment: Optional[str] = None*, *hilbert_dim: Optional[int] = None*, *calc_dim: Optional[int] = None*, *EC: Optional[c3.c3objs.Quantity] = None*, *EJ: Optional[c3.c3objs.Quantity] = None*, *EL: Optional[c3.c3objs.Quantity] = None*, *phi: Optional[c3.c3objs.Quantity] = None*, *phi_0: Optional[c3.c3objs.Quantity] = None*, *gamma: Optional[c3.c3objs.Quantity] = None*, *d: Optional[c3.c3objs.Quantity] = None*, *t1: Optional[c3.c3objs.Quantity] = None*, *t2star: Optional[c3.c3objs.Quantity] = None*, *temp: Optional[c3.c3objs.Quantity] = None*, *anhar: Optional[c3.c3objs.Quantity] = None*, *params=None*)

Bases: c3.libraries.chip.Qubit

cosm(*var*, *a=1*, *b=0*)

get_Hamiltonian(*signal: Optional[Union[dict, bool]] = None*, *transform: Optional[tensorflow.python.framework.ops.Tensor] = None*)

Compute the Hamiltonian. Multiplies the number operator with the frequency and anharmonicity with the Duffing part and returns their sum.

Returns Hamiltonian

Return type tf.Tensor

get_freq()

get_n_variable()

get_phase_variable()

init_Hs(*ann_oper*)

Initialize the qubit Hamiltonians. If the dimension is higher than two, a Duffing oscillator is used.

Parameters **ann_oper** (*np.array*) – Annihilation operator in the full Hilbert space

init_exponentiated_vars(*ann_oper*)

class c3.libraries.chip.**Coupling**(*name*, *desc=None*, *comment=None*, *strength: Optional[c3.c3objs.Quantity] = None*, *connected: Optional[List[str]] = None*, *params=None*, *hamiltonian_func=None*)

Bases: c3.libraries.chip.LineComponent

Represents a coupling behaviour between elements.

Parameters

- **strength** (*Quantity*) – coupling strength
- **connected** (*list*) – all physical components coupled via this specific coupling

get_Hamiltonian(*signal: Optional[Union[dict, bool]] = None*, *transform: Optional[tensorflow.python.framework.ops.Tensor] = None*)

init_Hs(*opers_list*)

class c3.libraries.chip.**Drive**(***props*)

Bases: c3.libraries.chip.LineComponent

Represents a drive line.

Parameters **connected** (*list*) – all physical components receiving driving signals via this line

get_Hamiltonian(*signal: Optional[Union[Dict, bool]] = None*, *transform: Optional[tensorflow.python.framework.ops.Tensor] = None*) → tensorflow.python.framework.ops.Tensor

init_Hs(*ann_opers: list*)

class c3.libraries.chip.**Fluxonium**(*name: str*, *desc: Optional[str] = None*, *comment: Optional[str] = None*, *hilbert_dim: Optional[int] = None*, *calc_dim: Optional[int] = None*, *EC: Optional[c3.c3objs.Quantity] = None*, *EJ: Optional[c3.c3objs.Quantity] = None*, *EL: Optional[c3.c3objs.Quantity] = None*, *phi: Optional[c3.c3objs.Quantity] = None*, *phi_0: Optional[c3.c3objs.Quantity] = None*, *gamma: Optional[c3.c3objs.Quantity] = None*, *t1: Optional[c3.c3objs.Quantity] = None*, *t2star: Optional[c3.c3objs.Quantity] = None*, *temp: Optional[c3.c3objs.Quantity] = None*, *params=None*)

Bases: c3.libraries.chip.CShuntFluxQubit

get_potential_function(*phi_variable*, *deriv_order=1*, *phi_sig=0*) → tf.float64

class c3.libraries.chip.**LineComponent**(***props*)

Bases: c3.c3objs.C3obj

Represents the components connecting chip elements and drives.

Parameters **connected** (*list*) – specifies the component that are connected with this line

asdict() → dict

class c3.libraries.chip.**PhysicalComponent**(***props*)

Bases: c3.c3objs.C3obj

Represents the components making up a chip.

Parameters **hilbert_dim** (*int*) – Dimension of the Hilbert space of this component

asdict() → dict

get_Hamiltonian(*signal: Optional[Union[dict, bool]] = None*, *transform: Optional[tensorflow.python.framework.ops.Tensor] = None*) → Dict[str, tensorflow.python.framework.ops.Tensor]

Compute the Hamiltonian. :param signal: dictionary with signals to be used a time dependend Hamiltonian. By default "values" key will be used.

If *true* value control hamiltonian will be returned, used for later combination of signal and hamiltonians.

Parameters **transform** – transform the hamiltonian, e.g. for expressing the hamiltonian in the expressed basis. Use this function if transform will be necessary and signal is given, in order to apply the *transform* only on single hamiltonians instead of all timeslices.

get_transformed_hamiltonians(*transform: Optional[tensorflow.python.framework.ops.Tensor] = None*)

get transformed hamiltonians with given applied transformation. The Hamiltonians are assumed to be stored in *Hs*. :param transform: transform to be applied to the hamiltonians. Default: None for returning the hamiltonians without transformation applied.

set_subspace_index(*index*)

class c3.libraries.chip.**Qubit**(*name*, *hilbert_dim*, *desc=None*, *comment=None*, *freq: Optional[c3.c3objs.Quantity] = None*, *anhar: Optional[c3.c3objs.Quantity] = None*, *t1: Optional[c3.c3objs.Quantity] = None*, *t2star: Optional[c3.c3objs.Quantity] = None*, *temp: Optional[c3.c3objs.Quantity] = None*, *params=None*)

Bases: c3.libraries.chip.PhysicalComponent

Represents the element in a chip functioning as qubit.

Parameters

- **freq** (*Quantity*) – frequency of the qubit
- **anhar** (*Quantity*) – anharmonicity of the qubit. defined as w01 - w12
- **t1** (*Quantity*) – t1, the time decay of the qubit due to dissipation
- **t2star** (*Quantity*) – t2star, the time decay of the qubit due to pure dephasing
- **temp** (*Quantity*) – temperature of the qubit, used to determine the Boltzmann distribution of energy level populations

get_Hamiltonian(*signal: Optional[Union[dict, bool]] = None*, *transform: Optional[tensorflow.python.framework.ops.Tensor] = None*)

Compute the Hamiltonian. Multiplies the number operator with the frequency and anharmonicity with the Duffing part and returns their sum.

Returns Hamiltonian

Return type tf.Tensor

get_Lindbladian(*dims*)

Compute the Lindbladian, based on relaxation, dephasing constants and finite temperature.

Returns Hamiltonian

Return type tf.Tensor

init_Hs(*ann_oper*)

Initialize the qubit Hamiltonians. If the dimension is higher than two, a Duffing oscillator is used.

Parameters **ann_oper** (*np.array*) – Annihilation operator in the full Hilbert space

init_Ls(*ann_oper*)

Initialize Lindbladian components.

Parameters **ann_oper** (*np.array*) – Annihilation operator in the full Hilbert space

class c3.libraries.chip.**Resonator**(***props*)

Bases: c3.libraries.chip.PhysicalComponent

Represents the element in a chip functioning as resonator.

Parameters **freq** (*Quantity*) – frequency of the resonator

get_Hamiltonian(*signal: Optional[Union[dict, bool]] = None*, *transform: Optional[tensorflow.python.framework.ops.Tensor] = None*)

Compute the Hamiltonian.

get_Lindbladian(*dims*)

NOT IMPLEMENTED

init_Hs(*ann_oper*)

Initialize the Hamiltonian as a number operator

Parameters **ann_oper** (*np.array*) – Annihilation operator in the full Hilbert space.

init_Ls(*ann_oper*)
NOT IMPLEMENTED

class c3.libraries.chip.**SNAIL**(*name: str*, *desc: str = ' '*, *comment: str = ' '*, *hilbert_dim: int = 4*, *freq: Optional[c3.c3objs.Quantity] = None*, *anhar: Optional[c3.c3objs.Quantity] = None*, *beta: Optional[c3.c3objs.Quantity] = None*, *t1: Optional[c3.c3objs.Quantity] = None*, *t2star: Optional[c3.c3objs.Quantity] = None*, *temp: Optional[c3.c3objs.Quantity] = None*, *params: Optional[dict] = None*)
Bases: c3.libraries.chip.Qubit

Represents the element in a chip functioning as a three wave mixing element also knwon as a SNAIL. Reference: https://arxiv.org/pdf/1702.00869.pdf :param freq: frequency of the qubit :type freq: Quantity :param anhar: anharmonicity of the qubit. defined as w01 - w12 :type anhar: Quantity :param beta: third order non_linearity of the qubit. :type beta: Quantity :param t1: t1, the time decay of the qubit due to dissipation :type t1: Quantity :param t2star: t2star, the time decay of the qubit due to pure dephasing :type t2star: Quantity :param temp: temperature of the qubit, used to determine the Boltzmann distribution

of energy level populations

Parameters

- **beta.** (*Class is mostly an exact copy of the Qubit class. The only difference is the added third order non linearity with a prefactor*) –
- **linearity** (*The only modification is the get hamiltonian and init hamiltonian definition. Also imported the necessary third order non*) –
- **library.** (*from the hamiltonian*) –

get_Hamiltonian(*signal: Optional[Union[dict, bool]] = None*, *transform: Optional[tensorflow.python.framework.ops.Tensor] = None*)
Compute the Hamiltonian. Multiplies the number operator with the frequency and anharmonicity with the Duffing part and returns their sum. :returns: Hamiltonian :rtype: tf.Tensor

init_Hs(*ann_oper*)
Initialize the SNAIL Hamiltonians. :param ann_oper: Annihilation operator in the full Hilbert space :type ann_oper: np.array

class c3.libraries.chip.**Transmon**(*name: str*, *desc: Optional[str] = None*, *comment: Optional[str] = None*, *hilbert_dim: Optional[int] = None*, *freq: Optional[c3.c3objs.Quantity] = None*, *anhar: Optional[c3.c3objs.Quantity] = None*, *phi: Optional[c3.c3objs.Quantity] = None*, *phi_0: Optional[c3.c3objs.Quantity] = None*, *gamma: Optional[c3.c3objs.Quantity] = None*, *d: Optional[c3.c3objs.Quantity] = None*, *t1: Optional[c3.c3objs.Quantity] = None*, *t2star: Optional[c3.c3objs.Quantity] = None*, *temp: Optional[c3.c3objs.Quantity] = None*, *params=None*)
Bases: c3.libraries.chip.PhysicalComponent

Represents the element in a chip functioning as tunanble transmon qubit.

Parameters

- **freq** (*Quantity*) – base frequency of the Transmon
- **phi_0** (*Quantity*) – half period of the phase dependant function
- **phi** (*Quantity*) – flux position

get_Hamiltonian(*signal: Optional[Union[dict, bool]] = None*, *transform: Optional[tensorflow.python.framework.ops.Tensor] = None*)
Compute the Hamiltonian. :param signal: dictionary with signals to be used a time dependend Hamiltonian. By default "values" key will be used.

If *true* value control hamiltonian will be returned, used for later combination of signal and hamiltonians.

Parameters **transform** – transform the hamiltonian, e.g. for expressing the hamiltonian in the expressed basis. Use this function if transform will be necessary and signal is given, in order to apply the *transform* only on single hamiltonians instead of all timeslices.

get_Lindbladian(*dims*)
Compute the Lindbladian, based on relaxation, dephasing constants and finite temperature.

Returns Hamiltonian

Return type tf.Tensor

get_anhar()

get_factor(*phi_sig=0*)

get_freq(*phi_sig=0*)

init_Hs(*ann_oper*)

init_Ls(*ann_oper*)
Initialize Lindbladian components.

Parameters **ann_oper** (*np.array*) – Annihilation operator in the full Hilbert space

class c3.libraries.chip.**TransmonExpanded**(*name: str*, *desc: Optional[str] = None*, *comment: Optional[str] = None*, *hilbert_dim: Optional[int] = None*, *freq: Optional[c3.c3objs.Quantity] = None*, *anhar: Optional[c3.c3objs.Quantity] = None*, *phi: Optional[c3.c3objs.Quantity] = None*, *phi_0: Optional[c3.c3objs.Quantity] = None*, *gamma: Optional[c3.c3objs.Quantity] = None*, *d: Optional[c3.c3objs.Quantity] = None*, *t1: Optional[c3.c3objs.Quantity] = None*, *t2star: Optional[c3.c3objs.Quantity] = None*, *temp: Optional[c3.c3objs.Quantity] = None*, *params=None*)
Bases: c3.libraries.chip.Transmon

energies_from_frequencies()

get_Hamiltonian(*signal: Optional[Union[dict, bool]] = None*, *transform: Optional[tensorflow.python.framework.ops.Tensor] = None*)
Compute the Hamiltonian. :param signal: dictionary with signals to be used a time dependend Hamiltonian. By default "values" key will be used.

If *true* value control hamiltonian will be returned, used for later combination of signal and hamiltonians.

Parameters **transform** – transform the hamiltonian, e.g. for expressing the hamiltonian in the expressed basis. Use this function if transform will be necessary and signal is given, in order to apply the *transform* only on single hamiltonians instead of all timeslices.

get_Hs(*ann_oper*)

get_prefactors(*sig*)

init_Hs(*ann_oper*)

c3.libraries.chip.**dev_reg_deco**(*func*)
Decorator for making registry of functions

12.7.2.3 Constants module

All physical constants used in other code.

12.7.2.4 Envelopes module

Library of envelope functions.

All functions assume the input of a time vector.

c3.libraries.envelopes.**cosine**(*t*, *params*)
Cosine-shaped envelope. Maximum value is 1, area is given by length.

Parameters **params** (*dict*) –

t_final [float] Total length of the Gaussian.

sigma: float Width of the Gaussian.

c3.libraries.envelopes.**cosine_flattop**(*t*, *params*)
Cosine-shaped envelope. Maximum value is 1, area is given by length.

Parameters **params** (*dict*) –

t_final [float] Total length of the Gaussian.

sigma: float Width of the Gaussian.

c3.libraries.envelopes.**delta_pulse**(*t*, *params*)
Pulse shape which gives an output only at a given time bin

c3.libraries.envelopes.**drag_der**(*t*, *params*)
Derivative of second order gaussian.

c3.libraries.envelopes.**drag_sigma**(*t*, *params*)
Second order gaussian.

c3.libraries.envelopes.**env_reg_deco**(*func*)
Decorator for making registry of functions

c3.libraries.envelopes.**flattop**(*t*, *params*)
Flattop gaussian with width of length risefall, modelled by error functions.

Parameters **params** (*dict*) –

t_up [float] Center of the ramp up.

t_down [float] Center of the ramp down.

risefall [float] Length of the ramps.

c3.libraries.envelopes.**flattop_cut**(*t*, *params*)
Flattop gaussian with width of length risefall, modelled by error functions.

Parameters **params** (*dict*) –

t_up [float] Center of the ramp up.

t_down [float] Center of the ramp down.

risefall [float] Length of the ramps.

c3.libraries.envelopes.**flattop_cut_center**(*t*, *params*)
Flattop gaussian with width of length risefall, modelled by error functions.

Parameters **params** (*dict*) –

t_up [float] Center of the ramp up.

t_down [float] Center of the ramp down.

risefall [float] Length of the ramps.

c3.libraries.envelopes.**flattop_risefall**(*t*, *params*)
Flattop gaussian with width of length risefall, modelled by error functions.

Parameters **params** (*dict*) –

t_final [float] Total length of pulse.

risefall [float] Length of the ramps. Position of ramps is so that the pulse starts with the start of the ramp-up and ends at the end of the ramp-down

c3.libraries.envelopes.**flattop_risefall_1ns**(*t*, *params*)
Flattop gaussian with fixed width of 1ns.

c3.libraries.envelopes.**flattop_variant**(*t*, *params*)
Flattop variant.

c3.libraries.envelopes.**fourier_cos**(*t*, *params*)
Fourier basis of the pulse constant pulse (cos).

Parameters **params** (*dict*) –

amps [list] Weights of the fourier components

freqs [list] Frequencies of the fourier components

c3.libraries.envelopes.**fourier_sin**(*t*, *params*)
Fourier basis of the pulse constant pulse (sin).

Parameters **params** (*dict*) –

amps [list] Weights of the fourier components

freqs [list] Frequencies of the fourier components

c3.libraries.envelopes.**gaussian_der**(*t*, *params*)
Derivative of the normalized gaussian (ifself not normalized).

c3.libraries.envelopes.**gaussian_der_nonorm**(*t*, *params*)
Derivative of the normalized gaussian (ifself not normalized).

c3.libraries.envelopes.**gaussian_nonorm**(*t*, *params*)
Non-normalized gaussian. Maximum value is 1, area is given by length.

Parameters **params** (*dict*) –

t_final [float] Total length of the Gaussian.

sigma: float Width of the Gaussian.

c3.libraries.envelopes.**gaussian_sigma**(*t*, *params*)
Normalized gaussian. Total area is 1, maximum is determined accordingly.

Parameters **params** (*dict*) –

t_final [float] Total length of the Gaussian.

sigma: float Width of the Gaussian.

c3.libraries.envelopes.**no_drive**(*t*, *params=None*)
Do nothing.

c3.libraries.envelopes.**pwc**(*t*, *params*)
Piecewise constant pulse.

c3.libraries.envelopes.**pwc_shape**(*t*, *params*)
Piecewise constant pulse while defining only a given number of samples, while interpolating linearly between those. :param t: :param params: t_bin_start/t_bin_end can be used to specify specific range. e.g. timepoints taken from awg.

c3.libraries.envelopes.**pwc_shape_plateau**(*t*, *params*)

c3.libraries.envelopes.**pwc_symmetric**(*t*, *params*)
symmetic PWC pulse This works only for inphase component

c3.libraries.envelopes.**rect**(*t*, *params=None*)
Rectangular pulse. Returns 1 at every time step.

c3.libraries.envelopes.**slepian_fourier**(*t*, *params*)

c3.libraries.envelopes.**trapezoid**(*t*, *params*)
Trapezoidal pulse. Width of linear slope.

Parameters **params** (*dict*) –

t_final [float] Total length of pulse.

risefall [float] Length of the slope

12.7.2.5 Estimators module

Collection of estimator functions, to compare two sets of (noisy) data.

c3.libraries.estimators.**dv_g_LL_prime**(*gs*, *dv_gs*, *weights*)

c3.libraries.estimators.**estimator_reg_deco**(*func*)
Decorator for making registry of functions

c3.libraries.estimators.**g_LL_prime**(*exp_values*, *sim_values*, *exp_stds*, *shots*)

c3.libraries.estimators.**g_LL_prime_combined**(*gs*, *weights*)

c3.libraries.estimators.**mean_dist**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Return the root mean squared of the differences.

c3.libraries.estimators.**mean_exp_stds_dist**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Return the mean of the distance in number of exp_stds away.

c3.libraries.estimators.**mean_sim_stds_dist**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Return the mean of the distance in number of exp_stds away.

c3.libraries.estimators.**median_dist**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Return the median of the differences.

c3.libraries.estimators.**neg_loglkh_binom**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Average likelihood of the experimental values with binomial distribution.

Return the likelihood of the experimental values given the simulated values, and given a binomial distribution function.

c3.libraries.estimators.**neg_loglkh_binom_norm**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Average likelihood of the exp values with normalised binomial distribution.

Return the likelihood of the experimental values given the simulated values, and given a binomial distribution function that is normalised to give probability 1 at the top of the distribution.

c3.libraries.estimators.**neg_loglkh_gauss**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Likelihood of the experimental values.

The distribution is assumed to be binomial (approximated by a gaussian).

c3.libraries.estimators.**neg_loglkh_gauss_norm**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Likelihood of the experimental values.

The distribution is assumed to be binomial (approximated by a gaussian) that is normalised to give probability 1 at the top of the distribution.

c3.libraries.estimators.**neg_loglkh_gauss_norm_sum**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Likelihood of the experimental values.

The distribution is assumed to be binomial (approximated by a gaussian) that is normalised to give probability 1 at the top of the distribution.

c3.libraries.estimators.**neg_loglkh_multinom**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Average likelihood of the experimental values with multinomial distribution.

Return the likelihood of the experimental values given the simulated values, and given a multinomial distribution function.

c3.libraries.estimators.**neg_loglkh_multinom_norm**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Average likelihood of the experimental values with multinomial distribution.

Return the likelihood of the experimental values given the simulated values, and given a multinomial distribution function that is normalised to give probability 1 at the top of the distribution.

c3.libraries.estimators.**rms_dist**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Return the root mean squared of the differences.

c3.libraries.estimators.**rms_exp_stds_dist**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Return the root mean squared of the differences measured in exp_stds.

c3.libraries.estimators.**rms_sim_stds_dist**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Return the root mean squared of the differences measured in exp_stds.

c3.libraries.estimators.**std_of_diffs**(*exp_values*, *sim_values*, *exp_stds*, *shots*)
Return the std of the distances.

12.7.2.6 Fidelities module

Library of fidelity functions.

c3.libraries.fidelities.**RB**(*propagators*, *min_length: int = 5*, *max_length: int = 500*, *num_lengths: int = 20*, *num_seqs: int = 30*, *logspace=False*, *lindbladian=False*, *padding=''*)

c3.libraries.fidelities.**average_infid**(*ideal: numpy.ndarray*, *actual: tensorflow.python.framework.ops.Tensor*, *index: List[int] = [0]*, *dims=[2]*) → tensorflow.python.framework.constant_op.constant
Average fidelity uses the Pauli basis to compare. Thus, perfect gates are always 2x2 (per qubit) and the actual unitary needs to be projected down.

Parameters

- **ideal** (*np.ndarray*) – Contains ideal unitary representations of the gate

- **actual** (*tf.Tensor*) – Contains actual unitary representations of the gate
- **index** (*List[int]*) – Index of the qubit(s) in the Hilbert space to be evaluated
- **dims** (*list*) – List of dimensions of qubits

c3.libraries.fidelities.**average_infid_seq**(*propagators: dict*, *instructions: dict*, *index*, *dims*, *n_eval=- 1*)
Average sequence fidelity over all gates in propagators.

Parameters

- **propagators** (*dict*) – Contains unitary representations of the gates, identified by a key.
- **index** (*int*) – Index of the qubit(s) in the Hilbert space to be evaluated
- **dims** (*list*) – List of dimensions of qubits
- **proj** (*boolean*) – Project to computational subspace

Returns Mean average fidelity

Return type tf.float64

c3.libraries.fidelities.**average_infid_set**(*propagators: dict*, *instructions: dict*, *index: List[int]*, *dims*, *n_eval=- 1*)
Mean average fidelity over all gates in propagators.

Parameters

- **propagators** (*dict*) – Contains unitary representations of the gates, identified by a key.
- **index** (*int*) – Index of the qubit(s) in the Hilbert space to be evaluated
- **dims** (*list*) – List of dimensions of qubits
- **proj** (*boolean*) – Project to computational subspace

Returns Mean average fidelity

Return type tf.float64

c3.libraries.fidelities.**epc_analytical**(*propagators: dict*, *index*, *dims*, *proj: bool*, *cliffords=False*)

c3.libraries.fidelities.**fid_reg_deco**(*func*)
Decorator for making registry of functions

c3.libraries.fidelities.**leakage_RB**(*propagators*, *min_length: int = 5*, *max_length: int = 500*, *num_lengths: int = 20*, *num_seqs: int = 30*, *logspace=False*, *lindbladian=False*)

c3.libraries.fidelities.**lindbladian_RB_left**(*propagators: dict*, *gate: str*, *index*, *dims*, *proj: bool = False*)

c3.libraries.fidelities.**lindbladian_RB_right**(*propagators: dict*, *gate: str*, *index*, *dims*, *proj: bool*)

c3.libraries.fidelities.**lindbladian_average_infid**(*ideal: numpy.ndarray*, *actual: tensorflow.python.framework.constant_op.constant*, *index=[0]*, *dims=[2]*) → tensorflow.python.framework.constant_op.constant
Average fidelity uses the Pauli basis to compare. Thus, perfect gates are always 2x2 (per qubit) and the actual unitary needs to be projected down.

Parameters

- **ideal** (*np.ndarray*) – Contains ideal unitary representations of the gate
- **actual** (*tf.Tensor*) – Contains actual unitary representations of the gate
- **index** (*int*) – Index of the qubit(s) in the Hilbert space to be evaluated

- **dims** (*list*) – List of dimensions of qubits

c3.libraries.fidelities.**lindbladian_average_infid_set**(*propagators: dict*, *instructions: Dict[str, c3.signal.gates.Instruction]*, *index*, *dims*, *n_eval*)

Mean average fidelity over all gates in propagators.

Parameters

- **propagators** (*dict*) – Contains unitary representations of the gates, identified by a key.
- **index** (*int*) – Index of the qubit(s) in the Hilbert space to be evaluated
- **dims** (*list*) – List of dimensions of qubits
- **proj** (*boolean*) – Project to computational subspace

Returns Mean average fidelity

Return type tf.float64

c3.libraries.fidelities.**lindbladian_epc_analytical**(*propagators: dict*, *index*, *dims*, *proj: bool*, *cliffords=False*)

c3.libraries.fidelities.**lindbladian_population**(*propagators: dict*, *lvl: int*, *gate: str*)

c3.libraries.fidelities.**lindbladian_unitary_infid**(*ideal: numpy.ndarray*, *actual: tensorflow.python.framework.constant_op.constant*, *index=[0]*, *dims=[2]*) → tensorflow.python.framework.constant_op.constant

Variant of the unitary fidelity for the Lindbladian propagator.

Parameters

- **ideal** (*np.ndarray*) – Contains ideal unitary representations of the gate
- **actual** (*tf.Tensor*) – Contains actual unitary representations of the gate
- **index** (*List[int]*) – Index of the qubit(s) in the Hilbert space to be evaluated
- **dims** (*list*) – List of dimensions of qubits

Returns Overlap fidelity for the Lindblad propagator.

Return type tf.float

c3.libraries.fidelities.**lindbladian_unitary_infid_set**(*propagators: dict*, *instructions: Dict[str, c3.signal.gates.Instruction]*, *index*, *dims*, *n_eval*)

Variant of the mean unitary fidelity for the Lindbladian propagator.

Parameters

- **propagators** (*dict*) – Contains actual unitary representations of the gates, resulting from physical simulation
- **instructions** (*dict*) – Contains the perfect unitary representations of the gates, identified by a key.
- **index** (*List[int]*) – Index of the qubit(s) in the Hilbert space to be evaluated
- **dims** (*list*) – List of dimensions of qubits
- **n_eval** (*int*) – Number of evaluation

Returns Mean overlap fidelity for the Lindblad propagator for all gates in propagators.

Return type tf.float

c3.libraries.fidelities.**open_system_deco**(*func*)
Decorator for making registry of functions

c3.libraries.fidelities.**orbit_infid**(*propagators*, *RB_number: int = 30*, *RB_length: int = 20*, *lindbladian=False*, *shots: Optional[int] = None*, *seqs=None*, *noise=None*)

c3.libraries.fidelities.**population**(*propagators: dict*, *lvl: int*, *gate: str*)

c3.libraries.fidelities.**populations**(*state*, *lindbladian*)

c3.libraries.fidelities.**set_deco**(*func*)
Decorator for making registry of functions

c3.libraries.fidelities.**state_deco**(*func*)
Decorator for making registry of functions

c3.libraries.fidelities.**state_transfer_infid**(*ideal: numpy.ndarray*, *actual: tensorflow.python.framework.constant_op.constant*, *index*, *dims*, *psi_0*)
Single gate state transfer infidelity. The dimensions of psi_0 and ideal need to be compatible and index and dims need to project actual to these same dimensions.

Parameters

- **ideal** (*np.ndarray*) – Contains ideal unitary representations of the gate
- **actual** (*tf.Tensor*) – Contains actual unitary representations of the gate
- **index** (*int*) – Index of the qubit(s) in the Hilbert space to be evaluated
- **dims** (*list*) – List of dimensions of qubits
- **psi_0** (*tf.Tensor*) – Initial state

Returns State infidelity for the selected gate

Return type tf.float

c3.libraries.fidelities.**state_transfer_infid_set**(*propagators: dict*, *instructions: dict*, *index*, *dims*, *psi_0*, *n_eval=- 1*, *proj=True*)
Mean state transfer infidelity.

Parameters

- **propagators** (*dict*) – Contains unitary representations of the gates, identified by a key.
- **index** (*int*) – Index of the qubit(s) in the Hilbert space to be evaluated
- **dims** (*list*) – List of dimensions of qubits
- **psi_0** (*tf.Tensor*) – Initial state of the device
- **proj** (*boolean*) – Project to computational subspace

Returns State infidelity, averaged over the gates in propagators

Return type tf.float

c3.libraries.fidelities.**unitary_deco**(*func*)
Decorator for making registry of functions

c3.libraries.fidelities.**unitary_infid**(*ideal: numpy.ndarray*, *actual: tensorflow.python.framework.ops.Tensor*, *index: Optional[List[int]] = None*, *dims=None*) → tensorflow.python.framework.ops.Tensor

Unitary overlap between ideal and actually performed gate.

Parameters

- **ideal** (*np.ndarray*) – Ideal or goal unitary representation of the gate.
- **actual** (*np.ndarray*) – Actual, physical unitary representation of the gate.
- **index** (*List[int]*) – Index of the qubit(s) in the Hilbert space to be evaluated
- **gate** (*str*) – One of the keys of propagators, selects the gate to be evaluated
- **dims** (*list*) – List of dimensions of qubits

Returns Unitary fidelity.

Return type tf.float

c3.libraries.fidelities.**unitary_infid_set**(*propagators: dict*, *instructions: dict*, *index*, *dims*, *n_eval=- 1*)

Mean unitary overlap between ideal and actually performed gate for the gates in propagators.

Parameters

- **propagators** (*dict*) – Contains actual unitary representations of the gates, resulting from physical simulation
- **instructions** (*dict*) – Contains the perfect unitary representations of the gates, identified by a key.
- **index** (*List[int]*) – Index of the qubit(s) in the Hilbert space to be evaluated
- **dims** (*list*) – List of dimensions of qubits
- **n_eval** (*int*) – Number of evaluation

Returns Unitary fidelity.

Return type tf.float

12.7.2.7 Hamiltonians module

Library of Hamiltonian functions.

c3.libraries.hamiltonians.**duffing**(*a*)

Anharmonic part of the duffing oscillator.

Parameters **a** (*Tensor*) – Annihilator.

Returns Number operator.

Return type Tensor

c3.libraries.hamiltonians.**hamiltonian_reg_deco**(*func*)

Decorator for making registry of functions

c3.libraries.hamiltonians.**int_XX**(*anhs*)

Dipole type coupling.

Parameters **anhs** (*Tensor list*) – Annihilators.

Returns coupling

Return type Tensor

c3.libraries.hamiltonians.**int_YY**(*anhs*)
Dipole type coupling.

Parameters **anhs** (*Tensor list*) – Annihilators.

Returns coupling

Return type Tensor

c3.libraries.hamiltonians.**resonator**(*a*)
Harmonic oscillator hamiltonian given the annihilation operator.

Parameters **a** (*Tensor*) – Annihilator.

Returns Number operator.

Return type Tensor

c3.libraries.hamiltonians.**third_order**(*a*)

Parameters **a** (*Tensor*) – Annihilator.

Returns

- *Tensor* – Number operator.
- *return literally the Hamiltonian a_dag a a + a_dag a_dag a for the use in any Hamiltonian that uses more than*
- *just a resonator or Duffing part. A more general type of quantum element on a physical chip can have this type of interaction.*
- *One example is a three wave mixing element used in signal amplification called a Superconducting non-linear asymmetric inductive eLement*
- *(SNAIL in short). The code is a simple modification of the Duffing function and written in the same style.*

c3.libraries.hamiltonians.**x_drive**(*a*)
Semiclassical drive.

Parameters **a** (*Tensor*) – Annihilator.

Returns Number operator.

Return type Tensor

c3.libraries.hamiltonians.**y_drive**(*a*)
Semiclassical drive.

Parameters **a** (*Tensor*) – Annihilator.

Returns Number operator.

Return type Tensor

c3.libraries.hamiltonians.**z_drive**(*a*)
Semiclassical drive.

Parameters **a** (*Tensor*) – Annihilator.

Returns Number operator.

Return type Tensor

12.7.2.8 Sampling module

Functions to select samples from a dataset by various criteria.

c3.libraries.sampling.**all**(*learn_from*, *batch_size*)
Return all points.

Parameters

- **learn_from** (*list*) – List of data points
- **batch_size** (*int*) – Number of points to select

Returns All indeces.

Return type list

c3.libraries.sampling.**even**(*learn_from*, *batch_size*)
Select evenly distanced samples across the set.

Parameters

- **learn_from** (*list*) – List of data points
- **batch_size** (*int*) – Number of points to select

Returns Selected indices.

Return type list

c3.libraries.sampling.**even_fid**(*learn_from*, *batch_size*)
Select evenly among reached fidelities.

Parameters

- **learn_from** (*list*) – List of data points.
- **batch_size** (*int*) – Number of points to select.

Returns Selected indices.

Return type list

c3.libraries.sampling.**from_end**(*learn_from*, *batch_size*)
Select from the end.

Parameters

- **learn_from** (*list*) – List of data points
- **batch_size** (*int*) – Number of points to select

Returns Selected indices.

Return type list

c3.libraries.sampling.**from_start**(*learn_from*, *batch_size*)
Select from the beginning.

Parameters

- **learn_from** (*list*) – List of data points
- **batch_size** (*int*) – Number of points to select

Returns Selected indices.

Return type list

c3.libraries.sampling.**high_std**(*learn_from*, *batch_size*)
Select points that have a high ratio of standard deviation to mean. Sampling from ORBIT data, points with a high std have the most coherent error, thus might be suitable for model learning. This has yet to be confirmed beyond anecdotal observation.

Parameters

- **learn_from** (*list*) – List of data points.
- **batch_size** (*int*) – Number of points to select.

Returns Selected indices.

Return type list

c3.libraries.sampling.**random_sample**(*learn_from*, *batch_size*)
Select randomly.

Parameters

- **learn_from** (*list*) – List of data points.
- **batch_size** (*int*) – Number of points to select.

Returns Selected indices.

Return type list

c3.libraries.sampling.**sampling_reg_deco**(*func*)
Decorator for making registry of functions

12.7.2.9 Tasks module

class c3.libraries.tasks.**ConfusionMatrix**(*name: str = 'conf_matrix'*, *desc: str = ' '*, *comment: str = ' '*, *params=None*, ***confusion_rows*)
Bases: c3.libraries.tasks.Task

Allows for misclassificaiton of readout measurement.

confuse(*pops*)
Apply the confusion (or misclassification) matrix to populations.

Parameters **pops** (*list*) – Populations

Returns Populations after misclassification.

Return type list

class c3.libraries.tasks.**InitialiseGround**(*name: str = 'init_ground'*, *desc: str = ' '*, *comment: str = ' '*, *init_temp: Optional[c3.c3objs.Quantity] = None*, *params=None*)
Bases: c3.libraries.tasks.Task

Initialise the ground state with a given thermal distribution.

initialise(*drift_ham*, *lindbladian=False*, *init_temp=None*)
Prepare the initial state of the system. At the moment finite temperature requires open system dynamics.

Parameters

- **drift_ham** (*tf.Tensor*) – Drift Hamiltonian.
- **lindbladian** (*boolean*) – Whether to include open system dynamics. Required for Temperature > 0.

- **init_temp** (*Quantity*) – Temperature of the device.

Returns State or density vector

Return type tf.Tensor

class c3.libraries.tasks.**MeasurementRescale**(*name: str = 'meas_rescale'*, *desc: str = ' '*, *comment: str = ' '*, *meas_offset: Optional[c3.c3objs.Quantity] = None*, *meas_scale: Optional[c3.c3objs.Quantity] = None*, *params=None*)

Bases: c3.libraries.tasks.Task

Rescale the result of the measurements. This is usually done to account for preparation errors.

Parameters

- **meas_offset** (*Quantity*) – Offset added to the measured signal.
- **meas_scale** (*Quantity*) – Factor multiplied to the measured signal.

rescale(*pop1*)

Apply linear rescaling and offset to the readout value.

Parameters **pop1** (*tf.float64*) – Population in first excited state.

Returns Population after rescaling.

Return type tf.float64

class c3.libraries.tasks.**Task**(*name: str = ' '*, *desc: str = ' '*, *comment: str = ' '*, *params: Optional[dict] = None*)

Bases: c3.c3objs.C3obj

Task that is part of the measurement setup.

c3.libraries.tasks.**task_deco**(*cl*)

Decorator for task classes list.

12.7.2.10 Module contents

12.7.3 Optimizers

12.7.3.1 C1 - Optimal control

Object that deals with the open loop optimal control.

class c3.optimizers.optimalcontrol.**OptimalControl**(*fid_func*, *fid_subspace*, *pmap*, *dir_path=None*, *callback_fids=None*, *algorithm=None*, *initial_point: str = ''*, *store_unitaries=False*, *options={}*, *run_name=None*, *interactive=True*, *include_model=False*, *logger=None*, *fid_func_kwargs={}*)

Bases: c3.optimizers.optimizer.Optimizer

Object that deals with the open loop optimal control.

Parameters

- **dir_path** (*str*) – Filepath to save results
- **fid_func** (*callable*) – infidelity function to be minimized
- **fid_subspace** (*list*) – Indeces identifying the subspace to be compared

- **pmap** (*ParameterMap*) – Identifiers for the parameter vector
- **callback_fids** (*list of callable*) – Additional fidelity function to be evaluated and stored for reference
- **algorithm** (*callable*) – From the algorithm library Save plots of control signals
- **store_unitaries** (*boolean*) – Store propagators as text and pickle
- **options** (*dict*) – Options to be passed to the algorithm
- **run_name** (*str*) – User specified name for the run, will be used as root folder
- **fid_func_kwargs** (*dict*) – Additional kwargs to be passed to the main fidelity function.

goal_run(*current_params: tensorflow.python.framework.ops.Tensor*) → tf.float64
Evaluate the goal function for current parameters.

Parameters **current_params** (*tf.Tensor*) – Vector representing the current parameter values.

Returns Value of the goal function

Return type tf.float64

load_model_parameters(*adjust_exp: str*) → None

log_setup() → None
Create the folders to store data.

optimize_controls(*setup_log: bool = True*) → None
Apply a search algorithm to your gateset given a fidelity function.

set_callback_fids(*callback_fids*) → None

set_fid_func(*fid_func*) → None

12.7.3.2 C2 - Calibration

Object that deals with the closed loop optimal control.

class c3.optimizers.calibration.**Calibration**(*eval_func*, *pmap*, *algorithm*, *dir_path=None*, *exp_type=None*, *exp_right=None*, *options={}*, *run_name=None*)

Bases: c3.optimizers.optimizer.Optimizer

Object that deals with the closed loop optimal control.

Parameters

- **dir_path** (*str*) – Filepath to save results
- **eval_func** (*callable*) – infidelity function to be minimized
- **pmap** (*ParameterMap*) – Identifiers for the parameter vector
- **algorithm** (*callable*) – From the algorithm library
- **options** (*dict*) – Options to be passed to the algorithm
- **run_name** (*str*) – User specified name for the run, will be used as root folder

goal_run(*current_params*)
Evaluate the goal function for current parameters.

Parameters **current_params** (*tf.Tensor*) – Vector representing the current parameter values.

Returns Value of the goal function

Return type tf.float64

log_pickle(*params*, *seqs*, *results*, *results_std*, *shots*)
Save a pickled version of the performed experiment, suitable for model learning.

Parameters

- **params** (*tf.Tensor*) – Vector of parameter values
- **seqs** (*list*) – Strings identifying the performed instructions
- **results** (*list*) – Values of the goal function
- **results_std** (*list*) – Standard deviation of the results, in the case of noisy data
- **shots** (*list*) – Number of repetitions used in averaging noisy data

log_setup() → None
Create the folders to store data.

Parameters

- **dir_path** (*str*) – Filepath
- **run_name** (*str*) – User specified name for the run

optimize_controls() → None
Apply a search algorithm to your gateset given a fidelity function.

set_eval_func(*eval_func*, *exp_type*)
Setter for the eval function.

Parameters **eval_func** (*callable*) – Function to be evaluated

12.7.3.3 C3 - Characterization

Object that deals with the model learning.

class c3.optimizers.modellearning.**ModelLearning**(*sampling*, *batch_sizes*, *pmap*, *datafiles*, *dir_path=None*, *estimator=None*, *seqs_per_point=None*, *state_labels=None*, *callback_foms=[]*, *algorithm=None*, *run_name=None*, *options={}*)

Bases: c3.optimizers.optimizer.Optimizer

Object that deals with the model learning.

Parameters

- **dir_path** (*str*) – Filepath to save results
- **sampling** (*str*) – Sampling method from the sampling library
- **batch_sizes** (*list*) – Number of points to select from each dataset
- **seqs_per_point** (*int*) – Number of sequences that use the same parameter set
- **pmap** (*ParameterMap*) – Identifiers for the parameter vector
- **state_labels** (*list*) – Identifiers for the qubit subspaces
- **callback_foms** (*list*) – Figures of merit to additionally compute and store
- **algorithm** (*callable*) – From the algorithm library
- **run_name** (*str*) – User specified name for the run, will be used as root folder

- **options** (*dict*) – Options to be passed to the algorithm

confirm() → None
Compute the validation set, i.e. the value of the goal function on all points of the dataset that were not used for learning.

goal_run(*current_params: tensorflow.python.framework.constant_op.constant*) → tf.float64
Evaluate the figure of merit for the current model parameters.

Parameters **current_params** (*tf.Tensor*) – Current model parameters

Returns Figure of merit

Return type tf.float64

goal_run_with_grad(*current_params*)
Same as goal_run but with gradient. Very resource intensive. Unoptimized at the moment.

learn_model() → None
Performs the model learning by minimizing the figure of merit.

log_setup() → None
Create the folders to store data.

Parameters

- **dir_path** (*str*) – Filepath
- **run_name** (*str*) – User specified name for the run

read_data(*datafiles: Dict[str, str]*) → None
Open data files and read in experiment results.

Parameters **datafiles** (*dict*) – List of paths for files that contain learning data.

select_from_data(*batch_size*) → List[int]
Select a subset of each dataset to compute the goal function on.

Parameters **batch_size** (*int*) – Number of points to select

Returns Indeces of the selected data points.

Return type list

12.7.3.4 Optimizer module

Optimizer object, where the optimal control is done.

class c3.optimizers.optimizer.**BaseLogger**
Bases: object

log_parameters(*evaluation*, *optim_status*)

start_log(*opt*, *logdir*)

class c3.optimizers.optimizer.**BestPointLogger**
Bases: *c3.optimizers.optimizer.BaseLogger*

class c3.optimizers.optimizer.**Optimizer**(*pmap: c3.parametermap.ParameterMap*, *initial_point: str = ''*, *algorithm: Optional[Callable] = None*, *store_unitaries: bool = False*, *logger: Optional[List] = None*)
Bases: object

General optimizer class from which specific classes are inherited.

Parameters

- **algorithm** (*callable*) – From the algorithm library
- **store_unitaries** (*boolean*) – Store propagators as text and pickle
- **logger** (*List*) – Logging classes

end_log() → None
Finish the log by recording current time and total runtime.

fct_to_min(*input_parameters: Union[numpy.ndarray, tensorflow.python.framework.constant_op.constant]*) → Union[numpy.ndarray, tensorflow.python.framework.constant_op.constant]
Wrapper for the goal function.

Parameters **input_parameters** (*[np.array, tf.constant]*) – Vector of parameters in the optimizer friendly way.

Returns Value of the goal function. Float if input is np.array else tf.constant

Return type [np.ndarray, tf.constant]

fct_to_min_autograd(*x*)

Wrapper for the goal function, including evaluation and storage of the gradient.

Parameters

x [np.array] Vector of parameters in the optimizer friendly way.

float Value of the goal function.

goal_run(*current_params: Union[numpy.ndarray, tensorflow.python.framework.constant_op.constant]*) → Union[numpy.ndarray, tensorflow.python.framework.constant_op.constant]
Placeholder for the goal function. To be implemented by inherited classes.

goal_run_with_grad(*current_params*)

load_best(*init_point*) → None
Load a previous parameter point to start the optimization from. Legacy wrapper. Method moved to Parametermap.

Parameters **init_point** (*str*) – File location of the initial point

log_best_unitary() → None
Save the best unitary in the log.

log_parameters() → None
Log the current status. Write parameters to log. Update the current best parameters. Call plotting functions as set up.

lookup_gradient(*x*)
Return the stored gradient for a given parameter set.

Parameters **x** (*np.array*) – Parameter set.

Returns Value of the gradient.

Return type np.array

replace_logdir(*new_logdir*)
Specify a new filepath to store the log.

Parameters **new_logdir** –

set_algorithm(*algorithm: Optional[Callable]*) → None

set_created_by(*config*) → None
Store the config file location used to created this optimizer.

set_exp(*exp: c3.experiment.Experiment*) → None

start_log() → None
Initialize the log with current time.

class c3.optimizers.optimizer.**TensorBoardLogger**
Bases: *c3.optimizers.optimizer.BaseLogger*

log_parameters(*evaluation*, *optim_status*)

set_logdir(*logdir*)

start_log(*opt*, *logdir*)

write_params(*params*, *step=0*)

12.7.3.5 Sensitivity analysis

Module for Sensitivity Analysis. This allows the sweeping of the goal function in a given range of parameters to ascertain whether the dataset being used is sensitive to changes in the parameters of interest

class c3.optimizers.sensitivity.**Sensitivity**(*sampling: str*, *batch_sizes: Dict[str, int]*, *pmap: c3.parametermap.ParameterMap*, *datafiles: Dict[str, str]*, *state_labels: Dict[str, List[Any]]*, *sweep_map: List[List[Tuple[str]]]*, *sweep_bounds: List[List[int]]*, *algorithm: str*, *estimator: Optional[str] = None*, *estimator_list: Optional[List[str]] = None*, *dir_path: Optional[str] = None*, *run_name: Optional[str] = None*, *options={}*)
Bases: c3.optimizers.modellearning.ModelLearning

Class for Sensitivity Analysis, subclassed from Model Learning

Parameters

- **sampling** (*str*) – Name of the sampling method from library
- **batch_sizes** (*Dict[str, int]*) – Number of points to select from the dataset
- **pmap** (*ParameterMap*) – Model parameter map
- **datafiles** (*Dict[str, str]*) – The datafiles for each of the learning datasets
- **state_labels** (*Dict[str, List[Any]]*) – The labels for the excited states of the system
- **sweep_map** (*List[List[List[str]]]*) – Map of variables to be swept in exp_opt_map format
- **sweep_bounds** (*List[List[int]]*) – List of upper and lower bounds for each sweeping variable
- **algorithm** (*str*) – Name of the sweeping algorithm from the library
- **estimator** (*str, optional*) – Name of estimator method from library, by default None
- **estimator_list** (*List[str], optional*) – List of different estimators to be used, by default None
- **dir_path** (*str, optional*) – Path to save sensitivity logs, by default None

- **run_name** (*str, optional*) – Name of the experiment run, by default None
- **options** (*dict, optional*) – Options for the sweeping algorithm, by default {}

Raises **NotImplementedError** – When trying to set the estimator or estimator_list

sensitivity()
Run the sensitivity analysis.

12.7.3.6 Module contents

12.7.4 Signal package

12.7.4.1 Submodules

12.7.4.2 Gates module

class c3.signal.gates.**Instruction**(*name: str = ' '*, *targets: Optional[list] = None*, *params: Optional[dict] = None*, *ideal: Optional[numpy.array] = None*, *channels: List[str] = []*, *t_start: Optional[float] = None*, *t_end: Optional[float] = None*)

Bases: object

Collection of components making up the control signal for a line.

Parameters

- **t_start** (*np.float64*) – Start of the signal.
- **t_end** (*np.float64*) – End of the signal.
- **channels** (*list*) – List of channel names (strings)

comps
Nested dictionary with lines and components as keys

Type dict

Example

comps = {

'channel_1' [{] 'envelope1': envelope1, 'envelope2': envelope2, 'carrier': carrier }

}

add_component(*comp: c3.c3objs.C3obj*, *chan: str*, *options=None*, *name=None*) → None
Add one component, e.g. an envelope, local oscillator, to a channel.

Parameters

- **comp** (*C3obj*) – Component to be added.
- **chan** (*str*) – Identifier for the target channel
- **options** (*dict*) –

Options for this component, available keys are

delay: Quantity Delay execution of this component by a certain time

trigger_comp: Tuple[str] Tuple of (chan, name) of component acting as trigger. Delay time will be counted beginning with end of trigger

t_final_cut: Quantity Length of component, signal will be cut after this time. Also used for the trigger. If not given this invokation from components *t_final* will be attempted.

drag: bool Use drag correction for this component.

- **t_end** (*float*) – End of this component. None will use the full instruction. If t_end is None and t_start is given a length will be inherited from the instruction.

as_openqasm() → dict

asdict() → dict

auto_adjust_t_end(*buffer=0*)

from_dict(*cfg*, *name=None*)

get_awg_signal(*chan*, *ts*, *options=None*)

get_full_gate_length()

get_ideal_gate(*dims*, *index=None*)

get_key() → str

get_optimizable_parameters()

get_timings(*chan*, *name*, *minimal_time=False*)

quick_setup(*chan*, *qubit_freq*, *gate_time*, *v2hz=1*, *sideband=None*) → None
Initialize this instruction with a default envelope and carrier.

12.7.4.3 Pulse module

class c3.signal.pulse.**Carrier**(*name: str*, *desc: str = ' '*, *comment: str = ' '*, *params: dict = {}*)
Bases: c3.c3objs.C3obj

Represents the carrier of a pulse.

write_config(*filepath: str*) → None
Write dictionary to a HJSON file.

class c3.signal.pulse.**Envelope**(*name: str*, *desc: str = ' '*, *comment: str = ' '*, *params: dict = {}*, *shape: Optional[Union[Callable, str]] = None*, *drag=False*)
Bases: c3.c3objs.C3obj

Represents the envelopes shaping a pulse.

Parameters **shape** (*Callable*) – function evaluating the shape in time

asdict() → dict

get_shape_values(*ts*, *t_before=None*)
Return the value of the shape function at the specified times.

Parameters

- **ts** (*tf.Tensor*) – Vector of time samples.
- **t_before** (*tf.float64*) – Offset the beginning of the shape by this time.

write_config(*filepath: str*) → None
Write dictionary to a HJSON file.

class c3.signal.pulse.**EnvelopeNetZero**(*name: str*, *desc: str = ' '*, *comment: str = ' '*, *params: dict = {}*, *shape: Optional[function] = None*, *drag: bool = False*)

Bases: c3.signal.pulse.Envelope

Represents the envelopes shaping a pulse.

Parameters

- **shape** (*function*) – function evaluating the shape in time
- **params** (*dict*) – Parameters of the envelope Note: t_final

get_shape_values(*ts*, *t_before=None*)

Return the value of the shape function at the specified times.

Parameters

- **ts** (*tf.Tensor*) – Vector of time samples.
- **t_before** (*tf.float64*) – Offset the beginning of the shape by this time.

c3.signal.pulse.**comp_reg_deco**(*func*)

Decorator for making registry of functions

12.7.4.4 Module contents

12.7.5 Utilities package

12.7.5.1 Qutip utilities module

Useful functions to get basis vectors and matrices of the right size.

c3.utils.qt_utils.**T1_sequence**(*length*, *target*)

Generate a gate sequence to measure relaxation time in a two-qubit chip.

Parameters

- **length** (*int*) – Number of Identity gates.
- **target** (*int*) – Which qubit is measured.

Returns Relaxation sequence.

Return type list

c3.utils.qt_utils.**basis**(*lvls: int*, *pop_lvl: int*) → numpy.array

Construct a basis state vector.

Parameters

- **lvls** (*int*) – Dimension of the state.
- **pop_lvl** (*int*) – The populated entry.

Returns A normalized state vector with one populated entry.

Return type np.array

c3.utils.qt_utils.**expand_dims**(*op*, *dim*)

pad operator with zeros to be of dimension dim Attention! Not related to the TensorFlow function

c3.utils.qt_utils.**get_basis_matrices**(*dim*)

Basis matrices with single ones of the matrices with given dimensions.

c3.utils.qt_utils.**hilbert_space_kron**(*op*, *indx*, *dims*)
Extend an operator op to the full product hilbert space given by dimensions in dims.

Parameters

- **op** (*np.array*) – Operator to be extended.
- **indx** (*int*) – Position of which subspace to extend.
- **dims** (*list*) – New dimensions of the subspace.

Returns Extended operator.

Return type np.array

c3.utils.qt_utils.**insert_mat_kron**(*dims*, *target_ids*, *matrix*) → numpy.ndarray
Insert matrix at given indices. All other spaces are filled with zeros.

Parameters

- **dims** (*dimensions of each qubit subspace*) –
- **target_ids** (*qubits to apply matrix to*) –
- **matrix** (*matrix to be applied*) –

Returns

Return type composed matrix

c3.utils.qt_utils.**inverseC**(*sequence*)
Find the clifford to end a sequence s.t. it returns identity.

c3.utils.qt_utils.**kron_ids**(*dims*, *indices*, *matrices*)
Kronecker product of matrices at specified indices with identities everywhere else.

c3.utils.qt_utils.**np_kron_n**(*mat_list*)
Apply Kronecker product to a list of matrices.

c3.utils.qt_utils.**pad_matrix**(*matrix*, *dim*, *padding*)
Fills matrix dimensions with zeros or identity.

c3.utils.qt_utils.**pauli_basis**(*dims=[2]*)
Qutip implementation of the Pauli basis.

Parameters **dims** (*list*) – List of dimensions of each subspace.

Returns A square matrix containing the Pauli basis of the product space

Return type np.array

c3.utils.qt_utils.**perfect_cliffords**(*lvls: List[int]*, *proj: str = 'fulluni'*, *num_gates: int = 1*)
Legacy function to compute the clifford gates.

c3.utils.qt_utils.**perfect_parametric_gate**(*paulis_str*, *ang*, *dims*)
Construct an ideal parametric gate.

Parameters

- **paulis_str** (*str*) –

  Names for the Pauli matrices that identify the rotation axis. Example:

  - "X" for a single-qubit rotation about the X axis
  - "Z:X" for an entangling rotation about Z on the first and X on the second qubit

- **ang** (*float*) – Angle of the rotation

- **dims** (*list*) – Dimensions of the subspaces.

Returns Ideal gate.

Return type np.array

c3.utils.qt_utils.**perfect_single_q_parametric_gate**(*pauli_str*, *target*, *ang*, *dims*)
Construct an ideal parametric gate.

Parameters

- **paulis_str** (*str*) –

  Name for the Pauli matrices that identify the rotation axis. Example:

  - "X" for a single-qubit rotation about the X axis

- **ang** (*float*) – Angle of the rotation
- **dims** (*list*) – Dimensions of the subspaces.

Returns Ideal gate.

Return type np.array

c3.utils.qt_utils.**projector**(*dims*, *indices*, *outdims=None*)
Computes the projector to cut down a matrix to the computational space. The subspaces indicated in indeces will be projected to the lowest two states, the rest is projected onto the lowest state. If outdims is defined projection will be performed to those states.

c3.utils.qt_utils.**ramsey_echo_sequence**(*length*, *target*)
Generate a gate sequence to measure dephasing time in a two-qubit chip including a flip in the middle. This echo reduce effects detrimental to the dephasing measurement.

Parameters

- **length** (*int*) – Number of Identity gates. Should be even.
- **target** (*str*) – Which qubit is measured. Options: "left" or "right"

Returns Dephasing sequence.

Return type list

c3.utils.qt_utils.**ramsey_sequence**(*length*, *target*)
Generate a gate sequence to measure dephasing time in a two-qubit chip.

Parameters

- **length** (*int*) – Number of Identity gates.
- **target** (*str*) – Which qubit is measured. Options: "left" or "right"

Returns Dephasing sequence.

Return type list

c3.utils.qt_utils.**rotation**(*phase: float*, *xyz: numpy.array*) → numpy.array
General Rotation using Euler's formula.

Parameters

- **phase** (*np.float*) – Rotation angle.
- **xyz** (*np.array*) – Normal vector of the rotation axis.

Returns Unitary matrix

Return type np.array

c3.utils.qt_utils.**single_length_RB**(*RB_number: int*, *RB_length: int*, *target: int = 0*) → List[List[str]]
Given a length and number of repetitions it compiles Randomized Benchmarking sequences.

Parameters

- **RB_number** (*int*) – The number of sequences to construct.
- **RB_length** (*int*) – The number of Cliffords in each individual sequence.
- **target** (*int*) – Index of the target qubit

Returns List of RB sequences.

Return type list

c3.utils.qt_utils.**two_qubit_gate_tomography**(*gate*)
Sequences to generate tomography for evaluating a two qubit gate.

c3.utils.qt_utils.**xy_basis**(*lvls: int*, *vect: str*)
Construct basis states on the X, Y and Z axis.

Parameters

- **lvls** (*int*) – Dimensions of the Hilbert space.
- **vect** (*str*) – Identifier of the state. Options:

  'zp', 'zm', 'xp', 'xm', 'yp', 'ym'

Returns A state on one of the axis of the Bloch sphere.

Return type np.array

12.7.5.2 Tensorflow utilities module

Various utility functions to speed up tensorflow coding.

c3.utils.tf_utils.**Id_like**(*A*)
Identity of the same size as A.

c3.utils.tf_utils.**get_tf_log_level**()
Display the current tensorflow log level of the system.

c3.utils.tf_utils.**set_tf_log_level**(*lvl*)
Set tensorflows system log level.

REMARK: it seems like the 'TF_CPP_MIN_LOG_LEVEL' variable expects a string. the input of this function seems to work with both string and/or integer, as casting string to string does nothing. feels hacked? but I guess it's just python...

c3.utils.tf_utils.**super_to_choi**(*A*)
Convert a super operator to choi representation.

c3.utils.tf_utils.**tf_abs**(*x*)
Rewritten so that is has a gradient.

c3.utils.tf_utils.**tf_abs_squared**(*x*)
Rewritten so that is has a gradient.

c3.utils.tf_utils.**tf_ave**(*x: list*)
Take average of a list of values in tensorflow.

c3.utils.tf_utils.**tf_average_fidelity**(*A*, *B*, *lvls=None*)
A very useful but badly named fidelity measure.

c3.utils.tf_utils.**tf_choi_to_chi**(*U*, *dims=None*)
Convert the choi representation of a process to chi representation.

c3.utils.tf_utils.**tf_convolve**(*sig: tensorflow.python.framework.ops.Tensor*, *resp: tensorflow.python.framework.ops.Tensor*)
Compute the convolution with a time response.

Parameters

- **sig** (*tf.Tensor*) – Signal which will be convoluted, shape: [N]
- **resp** (*tf.Tensor*) – Response function to be convoluted with signal, shape: [M]

Returns convoluted signal of shape [N]

Return type tf.Tensor

c3.utils.tf_utils.**tf_diff**(*l*)
Running difference of the input list l. Equivalent to np.diff, except it returns the same shape by adding a 0 in the last entry.

c3.utils.tf_utils.**tf_dm_to_vec**(*dm*)
Convert a density matrix into a density vector.

c3.utils.tf_utils.**tf_dmdm_fid**(*rho*, *sigma*)
Trace fidelity between two density matrices.

c3.utils.tf_utils.**tf_dmket_fid**(*rho*, *psi*)
Fidelity between a state vector and a density matrix.

c3.utils.tf_utils.**tf_ketket_fid**(*psi1*, *psi2*)
Overlap of two state vectors.

c3.utils.tf_utils.**tf_kron**(*A*, *B*)
Kronecker product of 2 matrices. Can be applied with batch dimmensions.

c3.utils.tf_utils.**tf_limit_gpu_memory**(*memory_limit*)
Set a limit for the GPU memory.

c3.utils.tf_utils.**tf_list_avail_devices**()
List available devices.

Function for displaying all available devices for tf_setuptensorflow operations on the local machine.

TODO: Refine output of this function. But without further knowledge about what information is needed, best practise is to output all information available.

c3.utils.tf_utils.**tf_log10**(*x*)
Tensorflow had no logarithm with base 10. This is ours.

c3.utils.tf_utils.**tf_log_level_info**()
Display the information about different log levels in tensorflow.

c3.utils.tf_utils.**tf_matmul_left**(*dUs: tensorflow.python.framework.ops.Tensor*)

Parameters **dUs** – tf.Tensor Tensorlist of shape (N, n,m) with number N matrices of size nxm

Multiplies a list of matrices from the left.

c3.utils.tf_utils.**tf_matmul_n**(*tensor_list*)
Multiply a list of tensors as binary tree.

EXPERIMENTAL

c3.utils.tf_utils.**tf_matmul_right**(*dUs*)

Parameters **dUs** – tf.Tensor Tensorlist of shape (N, n,m) with number N matrices of size nxm

Multiplies a list of matrices from the right.

c3.utils.tf_utils.**tf_measure_operator**(*M*, *rho*)
Expectation value of a quantum operator by tracing with a density matrix.

Parameters

- **M** (*tf.tensor*) – A quantum operator.
- **rho** (*tf.tensor*) – A density matrix.

Returns Expectation value.

Return type tf.tensor

c3.utils.tf_utils.**tf_project_to_comp**(*A*, *dims*, *index=None*, *to_super=False*)
Project an operator onto the computational subspace.

c3.utils.tf_utils.**tf_setup**()

c3.utils.tf_utils.**tf_spost**(*A*)
Superoperator on the right of matrix A.

c3.utils.tf_utils.**tf_spre**(*A*)
Superoperator on the left of matrix A.

c3.utils.tf_utils.**tf_state_to_dm**(*psi_ket*)
Make a state vector into a density matrix.

c3.utils.tf_utils.**tf_super**(*A*)
Superoperator from both sides of matrix A.

c3.utils.tf_utils.**tf_super_to_fid**(*err*, *lvls*)
Return average fidelity of a process.

c3.utils.tf_utils.**tf_superoper_average_fidelity**(*A*, *B*, *lvls=None*)
A very useful but badly named fidelity measure.

c3.utils.tf_utils.**tf_superoper_unitary_overlap**(*A*, *B*, *lvls=None*)

c3.utils.tf_utils.**tf_unitary_overlap**(*A: tensorflow.python.framework.ops.Tensor*, *B: tensorflow.python.framework.ops.Tensor*, *lvls: Optional[tensorflow.python.framework.ops.Tensor] = None*) → tensorflow.python.framework.ops.Tensor
Unitary overlap between two matrices.

Parameters

- **A** (*tf.Tensor*) – Unitary A
- **B** (*tf.Tensor*) – Unitary B
- **lvls** (*tf.Tensor, optional*) – Levels, by default None

Returns Overlap between the two unitaries

Return type tf.Tensor

Raises

- **TypeError** – For errors during cast

- **ValueError** – For errors during matrix multiplicaton

c3.utils.tf_utils.**tf_vec_to_dm**(*vec*)
Convert a density vector to a density matrix.

12.7.5.3 Log Reader utilities module

c3.utils.log_reader.**show_table**(*log: Dict[str, Any]*, *console: rich.console.Console*) → None
Generate a rich table from an optimization status and display it on the console.

Parameters

- **log** (*Dict*) – Dictionary read from a json log file containing a c3-toolset optimization status.
- **console** (*Console*) – Rich console for output.

12.7.5.4 Miscellaneous utilities module

Miscellaneous, general utilities.

c3.utils.utils.**ask_yn**() → bool
Ask for y/n user decision in the command line.

c3.utils.utils.**deprecated**(*message: str*)
Decorator for deprecating functions

Parameters **message** (*str*) – Message to display along with DeprecationWarning

Examples

Add a `@deprecated("message")` decorator to the function:

```
@deprecated("Using standard width. Better use gaussian_sigma.")
def gaussian(t, params):
    ...
```

c3.utils.utils.**eng_num**(*val: float*) → Tuple[float, str]
Convert a number to engineering notation by returning number and prefix.

c3.utils.utils.**flatten**(*lis: List*, *ltypes=(<class 'list'>, <class 'tuple'>)*) → List
Flatten lists of arbitrary lengths https://rightfootin.blogspot.com/2006/09/more-on-python-flatten.html

Parameters

- **lis** (*List*) – The iterable to flatten
- **ltypes** (*tuple, optional*) – Possibly the datatype of the iterable, by default (list, tuple)

Returns Flattened list

Return type List

c3.utils.utils.**log_setup**(*data_path: Optional[str] = None*, *run_name: str = 'run'*) → str
Make sure the file path to save data exists. Create an appropriately named folder with date and time. Also creates a symlink "recent" to the folder.

Parameters

- **data_path** (*str*) – File path of where to store any data.

- **run_name** (*str*) – User specified name for the run.

Returns The file path to store new data.

Return type str

c3.utils.utils.**num3str**(*val: float*, *use_prefix: bool = True*) → str
Convert a number to a human readable string in engineering notation.

c3.utils.utils.**replace_symlink**(*path: str*, *alias: str*) → None
Create a symbolic link.

12.7.5.5 Module contents

12.7.6 Qiskit modules for C3

12.7.6.1 C3 Backend module

class c3.qiskit.c3_backend.**C3QasmPerfectSimulator**(*configuration=None*, *provider=None*, ***fields*)
Bases: c3.qiskit.c3_backend.C3QasmSimulator

A C3-based perfect gates simulator for Qiskit

Parameters **C3QasmSimulator** (*c3.qiskit.c3_backend.C3QasmSimulator*) – Inherits the C3QasmSimulator and implements a perfect gate simulator

DEFAULT_OPTIONS = {'initial_statevector': None, 'memory': False, 'shots': 1024}

MAX_QUBITS_MEMORY = 20

run_experiment(*experiment: qiskit.qobj.qasm_qobj.QasmQobjExperiment*) → Dict[str, Any]
Run an experiment (circuit) and return a single experiment result

Parameters **experiment** (*QasmQobjExperiment*) – experiment from qobj experiments list

Returns

A result dictionary which looks something like:

```
{
"name": name of this experiment (obtained from qobj.experiment␣
↪header)
"seed": random seed used for simulation
"shots": number of shots used in the simulation
"data":
    {
    "counts": {'0x9': 5, ...},
    "memory": ['0x9', '0xF', '0x1D', ..., '0x9']
    },
"status": status string for the simulation
"success": boolean
"time_taken": simulation time of this single experiment
}
```

Return type Dict[str, Any]

Raises **C3QiskitError** – If an error occured

class c3.qiskit.c3_backend.**C3QasmPhysicsSimulator**(*configuration=None*, *provider=None*, ***fields*)
Bases: c3.qiskit.c3_backend.C3QasmSimulator

A C3-based perfect gates simulator for Qiskit

Parameters **C3QasmSimulator** (*c3.qiskit.c3_backend.C3QasmSimulator*) – Inherits the C3QasmSimulator and implements a physics based simulator

DEFAULT_OPTIONS = {'initial_statevector': None, 'memory': False, 'shots': 1024}

MAX_QUBITS_MEMORY = 10

run_experiment(*experiment: qiskit.qobj.qasm_qobj.QasmQobjExperiment*) → Dict[str, Any]
Run an experiment (circuit) and return a single experiment result

Parameters **experiment** (*QasmQobjExperiment*) – experiment from qobj experiments list

Returns

A result dictionary which looks something like:

```
{
"name": name of this experiment (obtained from qobj.experiment␣
↪header)
"seed": random seed used for simulation
"shots": number of shots used in the simulation
"data":
    {
    "counts": {'0x9: 5, ...},
    "memory": ['0x9', '0xF', '0x1D', ..., '0x9']
    },
"status": status string for the simulation
"success": boolean
"time_taken": simulation time of this single experiment
}
```

Return type Dict[str, Any]

Raises **C3QiskitError** – If an error occured

class c3.qiskit.c3_backend.**C3QasmSimulator**(*configuration*, *provider=None*, ***fields*)
Bases: qiskit.providers.backend.BackendV1, abc.ABC

An Abtract Base Class for C3 Qasm Simulators for Qiskit. This class CAN NOT be instantiated directly. Classes derived from this must compulsorily implement

```
def __init__(self, configuration=None, provider=None, **fields):

def _default_options(cls) -> None:

def run_experiment(self, experiment: QasmQobjExperiment) -> Dict[str, Any]:
```

Parameters

- **Backend** (*qiskit.providers.BackendV1*) – The C3QasmSimulator is derived from BackendV1
- **ABC** (*abc.ABC*) – Helper class for defining Abstract classes using ABCMeta

disable_flip_labels() → None
Disable flipping of labels State Labels are flipped before returning results to match Qiskit style qubit indexing convention This function allows disabling of the flip

get_labels() → List[str]
Return state labels for the system

Returns

A list of state labels in hex format

```
labels = ['0x1', ...]
```

Return type List[str]

run(*qobj: qiskit.qobj.Qobj*, ***backend_options*) → c3.qiskit.c3_job.C3Job
Parse and run a Qobj

Parameters

- **qobj** (*Qobj*) – The Qobj payload for the experiment
- **backend_options** (*dict*) – backend options

Returns An instance of the C3Job (derived from JobV1) with the result

Return type C3Job

Raises **QiskitError** – Support for Pulse Jobs is not implemented

Notes

backend_options: Is a dict of options for the backend. It may contain

- "initial_statevector": vector_like

The "initial_statevector" option specifies a custom initial statevector for the simulator to be used instead of the all zero state. This size of this vector must be correct for the number of qubits in all experiments in the qobj.

Example:

```
backend_options = {
    "initial_statevector": np.array([1, 0, 0, 1j]) / np.sqrt(2),
}
```

abstract run_experiment(*experiment: qiskit.qobj.qasm_qobj.QasmQobjExperiment*) → Dict[str, Any]

set_device_config(*config_file: str*) → None
Set the path to the config for the device

Parameters **config_file** (*str*) – path to hjson file storing the configuration for all device parameters for simulation

12.7.6.2 C3 Provider module

class c3.qiskit.c3_provider.**C3Provider**
Bases: qiskit.providers.provider.ProviderV1

Provider for C3 Qiskit backends

Parameters **ProviderV1** (*ProviderV1*) – Derived from ProviderV1 from qiskit.providers.provider

backends(*name=None*, *filters=None*, ***kwargs*)
Return a list of backends matching the name

Parameters

- **name** (*str, optional*) – name of the backend, by default None
- **filters** (*callable, optional*) – Filtering conditions, as callable, by default None

Returns A list of backend instances matching the condition

Return type list[BackendV1]

12.7.6.3 C3 Job module

class c3.qiskit.c3_job.**C3Job**(*backend*, *job_id*, *result*)
Bases: qiskit.providers.job.JobV1

C3Job class

Parameters **Job** (*JobV1*) – Inherits JobV1 from qiskit.providers

result() → qiskit.result.result.Result
Return the result of the job

Returns Result of the job simulation

Return type qiskit.Result

status() → qiskit.providers.jobstatus.JobStatus
Return job status

Returns Status of the job

Return type qiskit.providers.JobStatus

submit() → None
Submit a job to the simulator

12.7.6.4 C3 Exceptions module

Exception for errors raised by Basic Aer.

exception c3.qiskit.c3_exceptions.**C3QiskitError**(**message*)
Bases: qiskit.exceptions.QiskitError

Base class for errors raised by C3 Qiskit Simulator.

12.7.6.5 C3 Backend Utilities module

Convenience Module for creating different c3_backend

c3.qiskit.c3_backend_utils.**flip_labels**(*counts: Dict[str, int]*) → Dict[str, int]
Flip C3 qubit labels to match Qiskit qubit indexing

Parameters **counts** (*Dict[str, int]*) – OpenQasm 2.0 result counts with original C3 style qubit indices

Returns OpenQasm 2.0 result counts with Qiskit style labels

Return type Dict[str, int]

Note: Basis vector ordering in Qiskit

Qiskit uses a slightly different ordering of the qubits compared to what is seen in Physics textbooks. In qiskit, the qubits are represented from the most significant bit (MSB) on the left to the least significant bit (LSB) on the right (big-endian). This is similar to bitstring representation on classical computers, and enables easy conversion from bitstrings to integers after measurements are performed.

More details: https://qiskit.org/documentation/tutorials/circuits/3_summary_of_quantum_operations.html#Basis-vector-ordering-in-Qiskit

---

c3.qiskit.c3_backend_utils.**get_init_ground_state**(*n_qubits: int*, *n_levels: int*) → tensorflow.python.framework.ops.Tensor
Return a perfect ground state

Parameters

- **n_qubits** (*int*) – Number of qubits in the system
- **n_levels** (*int*) – Number of levels for each qubit

Returns Tensor array of ground state shape(m^n, 1), dtype=complex128 m = no of qubit levels n = no of qubits

Return type tf.Tensor

c3.qiskit.c3_backend_utils.**get_sequence**(*instructions: List*) → List[str]
Return a sequence of c3 gates from Qasm instructions

Parameters **instructions** (*List[dict]*) – Instructions from the qasm experiment, for example:

```
instructions: [
        {"name": "u1", "qubits": [0], "params": [0.4]},
        {"name": "u2", "qubits": [0], "params": [0.4,0.2]},
        {"name": "u3", "qubits": [0], "params": [0.4,0.2,-0.3]},
        {"name": "snapshot", "label": "snapstate1", "snapshot_type":
↪"statevector"},
        {"name": "cx", "qubits": [0,1]},
        {"name": "barrier", "qubits": [0]},
        {"name": "measure", "qubits": [0], "register": [1], "memory":␣
↪[0]},
        {"name": "u2", "qubits": [0], "params": [0.4,0.2], "conditional
↪": 2}
    ]
```

Returns

List of gates, for example:

```
sequence = ["rx90p[1]", "cr90[0,1]", "rx90p[0]"]
```

Return type List[str]

c3.qiskit.c3_backend_utils.**make_gate_str**(*instruction: dict*, *gate_name: str*) → str

Make C3 style gate name string

Parameters

- **instruction** (*Dict[str, Any]*) – A dict in OpenQasm instruction format

```
{"name": "rx", "qubits": [0], "params": [1.57]}
```

- **gate_name** (*str*) – C3 style gate names

Returns

C3 style gate name + qubit string

```
{"name": "rx", "qubits": [0], "params": [1.57]} -> rx90p[0]
```

Return type str

12.7.6.6 Module contents

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

adaptive_scan() (*in module c3.libraries.algorithms*), 95
add() (*c3.c3objs.Quantity method*), 79
add_component() (*c3.signal.gates.Instruction method*), 122
Additive_Noise (*class in c3.generator.devices*), 88
algo_reg_deco() (*in module c3.libraries.algorithms*), 95
all() (*in module c3.libraries.sampling*), 114
as_openqasm() (*c3.signal.gates.Instruction method*), 123
asdict() (*c3.c3objs.C3obj method*), 79
asdict() (*c3.c3objs.Quantity method*), 79
asdict() (*c3.experiment.Experiment method*), 80
asdict() (*c3.generator.devices.AWG method*), 87
asdict() (*c3.generator.devices.Device method*), 89
asdict() (*c3.generator.generator.Generator method*), 94
asdict() (*c3.libraries.chip.LineComponent method*), 101
asdict() (*c3.libraries.chip.PhysicalComponent method*), 101
asdict() (*c3.model.Model method*), 83
asdict() (*c3.parametermap.ParameterMap method*), 85
asdict() (*c3.signal.gates.Instruction method*), 123
asdict() (*c3.signal.pulse.Envelope method*), 123
ask_yn() (*in module c3.utils.utils*), 130
auto_adjust_t_end() (*c3.signal.gates.Instruction method*), 123
average_infid() (*in module c3.libraries.fidelities*), 108
average_infid_seq() (*in module c3.libraries.fidelities*), 109
average_infid_set() (*in module c3.libraries.fidelities*), 109
AWG (*class in c3.generator.devices*), 87

B

backends() (*c3.qiskit.c3_provider.C3Provider method*), 134
BaseLogger (*class in c3.optimizers.optimizer*), 119
basis() (*in module c3.utils.qt_utils*), 124
BestPointLogger (*class in c3.optimizers.optimizer*), 119
blowup_excitations() (*c3.model.Model method*), 83

C

c3
    module, 87
c3.c3objs
    module, 79
c3.experiment
    module, 80
c3.generator
    module, 95
c3.generator.devices
    module, 87
c3.generator.generator
    module, 94
c3.libraries
    module, 116
c3.libraries.algorithms
    module, 95
c3.libraries.chip
    module, 99
c3.libraries.constants
    module, 105
c3.libraries.envelopes
    module, 105
c3.libraries.estimators
    module, 107
c3.libraries.fidelities
    module, 108
c3.libraries.hamiltonians
    module, 112
c3.libraries.sampling
    module, 114
c3.libraries.tasks
    module, 115
c3.main
    module, 87
c3.model
    module, 83
c3.optimizers

module, 122
c3.optimizers.calibration
module, 117
c3.optimizers.modellearning
module, 118
c3.optimizers.optimalcontrol
module, 116
c3.optimizers.optimizer
module, 119
c3.optimizers.sensitivity
module, 121
c3.parametermap
module, 85
c3.qiskit
module, 136
c3.qiskit.c3_backend
module, 131
c3.qiskit.c3_backend_utils
module, 135
c3.qiskit.c3_exceptions
module, 134
c3.qiskit.c3_job
module, 134
c3.qiskit.c3_provider
module, 134
c3.signal
module, 124
c3.signal.gates
module, 122
c3.signal.pulse
module, 123
c3.utils
module, 131
c3.utils.log_reader
module, 130
c3.utils.qt_utils
module, 124
c3.utils.tf_utils
module, 127
c3.utils.utils
module, 130
C3Job (*class in c3.qiskit.c3_job*), 134
C3obj (*class in c3.c3objs*), 79
C3Provider (*class in c3.qiskit.c3_provider*), 134
C3QasmPerfectSimulator (*class in c3.qiskit.c3_backend*), 131
C3QasmPhysicsSimulator (*class in c3.qiskit.c3_backend*), 131
C3QasmSimulator (*class in c3.qiskit.c3_backend*), 132
C3QiskitError, 134
calc_slice_num() (*c3.generator.devices.Device method*), 89
Calibration (*class in c3.optimizers.calibration*), 117
Carrier (*class in c3.signal.pulse*), 123
check_limits() (*c3.parametermap.ParameterMap method*), 85
cma_pre_lbfgs() (*in module c3.libraries.algorithms*), 95
cmaes() (*in module c3.libraries.algorithms*), 95
comp_reg_deco() (*in module c3.signal.pulse*), 124
comps (*c3.signal.gates.Instruction attribute*), 122
compute_propagators() (*c3.experiment.Experiment method*), 81
compute_states() (*c3.experiment.Experiment method*), 81
confirm() (*c3.optimizers.modellearning.ModelLearning method*), 119
confuse() (*c3.libraries.tasks.ConfusionMatrix method*), 115
ConfusionMatrix (*class in c3.libraries.tasks*), 115
convolve() (*c3.generator.devices.HighpassFilter method*), 91
convolve() (*c3.generator.devices.Response method*), 93
cosine() (*in module c3.libraries.envelopes*), 105
cosine_flattop() (*in module c3.libraries.envelopes*), 105
cosm() (*c3.libraries.chip.CShuntFluxQubitCos method*), 100
Coupling (*class in c3.libraries.chip*), 100
create_IQ() (*c3.generator.devices.AWG method*), 87
create_IQ_pwc() (*c3.generator.devices.AWG method*), 87
create_ts() (*c3.generator.devices.Device method*), 89
Crosstalk (*class in c3.generator.devices*), 88
CShuntFluxQubit (*class in c3.libraries.chip*), 99
CShuntFluxQubitCos (*class in c3.libraries.chip*), 99
cut_excitations() (*c3.model.Model method*), 83

D

DC_Noise (*class in c3.generator.devices*), 89
DC_Offset (*class in c3.generator.devices*), 89
DEFAULT_OPTIONS (*c3.qiskit.c3_backend.C3QasmPerfectSimulator attribute*), 131
DEFAULT_OPTIONS (*c3.qiskit.c3_backend.C3QasmPhysicsSimulator attribute*), 132
delta_pulse() (*in module c3.libraries.envelopes*), 105
deprecated() (*in module c3.utils.utils*), 130
dev_reg_deco() (*in module c3.generator.devices*), 94
dev_reg_deco() (*in module c3.libraries.chip*), 105
Device (*class in c3.generator.devices*), 89
DigitalToAnalog (*class in c3.generator.devices*), 90
disable_flip_labels() (*c3.qiskit.c3_backend.C3QasmSimulator method*), 132
drag_der() (*in module c3.libraries.envelopes*), 105
drag_sigma() (*in module c3.libraries.envelopes*), 105
Drive (*class in c3.libraries.chip*), 100
duffing() (*in module c3.libraries.hamiltonians*), 112

dv_g_LL_prime() (*in module c3.libraries.estimators*), 107

E

enable_drag() (*c3.generator.devices.AWG method*), 88
enable_drag_2() (*c3.generator.devices.AWG method*), 88
enable_pwc() (*c3.generator.devices.AWG method*), 88
enable_qasm() (*c3.experiment.Experiment method*), 81
end_log() (*c3.optimizers.optimizer.Optimizer method*), 120
energies_from_frequencies() (*c3.libraries.chip.TransmonExpanded method*), 104
eng_num() (*in module c3.utils.utils*), 130
env_reg_deco() (*in module c3.libraries.envelopes*), 105
Envelope (*class in c3.signal.pulse*), 123
EnvelopeNetZero (*class in c3.signal.pulse*), 123
epc_analytical() (*in module c3.libraries.fidelities*), 109
estimator_reg_deco() (*in module c3.libraries.estimators*), 107
evaluate_legacy() (*c3.experiment.Experiment method*), 81
evaluate_qasm() (*c3.experiment.Experiment method*), 81
even() (*in module c3.libraries.sampling*), 114
even_fid() (*in module c3.libraries.sampling*), 114
Example (*c3.signal.gates.Instruction attribute*), 122
expand_dims() (*in module c3.utils.qt_utils*), 124
expect_oper() (*c3.experiment.Experiment method*), 81
Experiment (*class in c3.experiment*), 80
ExponentialIIR (*class in c3.generator.devices*), 90

F

fct_to_min() (*c3.optimizers.optimizer.Optimizer method*), 120
fct_to_min_autograd() (*c3.optimizers.optimizer.Optimizer method*), 120
fid_reg_deco() (*in module c3.libraries.fidelities*), 109
Filter (*class in c3.generator.devices*), 90
flatten() (*in module c3.utils.utils*), 130
flattop() (*in module c3.libraries.envelopes*), 105
flattop_cut() (*in module c3.libraries.envelopes*), 105
flattop_cut_center() (*in module c3.libraries.envelopes*), 106
flattop_risefall() (*in module c3.libraries.envelopes*), 106
flattop_risefall_1ns() (*in module c3.libraries.envelopes*), 106
flattop_variant() (*in module c3.libraries.envelopes*), 106
flip_labels() (*in module c3.qiskit.c3_backend_utils*), 135
Fluxonium (*class in c3.libraries.chip*), 101
FluxTuning (*class in c3.generator.devices*), 90
FluxTuningLinear (*class in c3.generator.devices*), 91
fourier_cos() (*in module c3.libraries.envelopes*), 106
fourier_sin() (*in module c3.libraries.envelopes*), 106
frequency() (*c3.generator.devices.FluxTuningLinear method*), 91
from_dict() (*c3.experiment.Experiment method*), 81
from_dict() (*c3.signal.gates.Instruction method*), 123
from_end() (*in module c3.libraries.sampling*), 114
from_start() (*in module c3.libraries.sampling*), 114
fromdict() (*c3.generator.generator.Generator method*), 94
fromdict() (*c3.model.Model method*), 83
fromdict() (*c3.parametermap.ParameterMap method*), 85

G

g_LL_prime() (*in module c3.libraries.estimators*), 107
g_LL_prime_combined() (*in module c3.libraries.estimators*), 107
gaussian_der() (*in module c3.libraries.envelopes*), 106
gaussian_der_nonorm() (*in module c3.libraries.envelopes*), 106
gaussian_nonorm() (*in module c3.libraries.envelopes*), 106
gaussian_sigma() (*in module c3.libraries.envelopes*), 106
gcmaes() (*in module c3.libraries.algorithms*), 96
generate_signals() (*c3.generator.generator.Generator method*), 94
Generator (*class in c3.generator.generator*), 94
get_anhar() (*c3.libraries.chip.Transmon method*), 104
get_anharmonicity() (*c3.libraries.chip.CShuntFluxQubit method*), 99
get_average_amp() (*c3.generator.devices.AWG method*), 88
get_awg_signal() (*c3.signal.gates.Instruction method*), 123
get_basis_matrices() (*in module c3.utils.qt_utils*), 124
get_dephasing_channel() (*c3.model.Model method*), 84
get_factor() (*c3.generator.devices.FluxTuning method*), 91
get_factor() (*c3.libraries.chip.Transmon method*), 104
get_Frame_Rotation() (*c3.model.Model method*), 83
get_freq() (*c3.generator.devices.FluxTuning method*), 91
get_freq() (*c3.libraries.chip.CShuntFluxQubit method*), 99

get_freq() (*c3.libraries.chip.CShuntFluxQubitCos method*), 100
get_freq() (*c3.libraries.chip.Transmon method*), 104
get_frequency() (*c3.libraries.chip.CShuntFluxQubit method*), 99
get_full_gate_length() (*c3.signal.gates.Instruction method*), 123
get_full_params() (*c3.parametermap.ParameterMap method*), 85
get_ground_state() (*c3.model.Model method*), 84
get_Hamiltonian() (*c3.libraries.chip.Coupling method*), 100
get_Hamiltonian() (*c3.libraries.chip.CShuntFluxQubit method*), 99
get_Hamiltonian() (*c3.libraries.chip.CShuntFluxQubitCos method*), 100
get_Hamiltonian() (*c3.libraries.chip.Drive method*), 100
get_Hamiltonian() (*c3.libraries.chip.PhysicalComponent method*), 101
get_Hamiltonian() (*c3.libraries.chip.Qubit method*), 102
get_Hamiltonian() (*c3.libraries.chip.Resonator method*), 102
get_Hamiltonian() (*c3.libraries.chip.SNAIL method*), 103
get_Hamiltonian() (*c3.libraries.chip.Transmon method*), 103
get_Hamiltonian() (*c3.libraries.chip.TransmonExpanded method*), 104
get_Hamiltonian() (*c3.model.Model method*), 83
get_Hamiltonians() (*c3.model.Model method*), 83
get_Hs() (*c3.libraries.chip.TransmonExpanded method*), 104
get_I() (*c3.generator.devices.AWG method*), 88
get_ideal_gate() (*c3.signal.gates.Instruction method*), 123
get_init_ground_state() (*in module c3.qiskit.c3_backend_utils*), 135
get_key() (*c3.signal.gates.Instruction method*), 123
get_key_from_scaled_index() (*c3.parametermap.ParameterMap method*), 85
get_labels() (*c3.qiskit.c3_backend.C3QasmSimulator method*), 133
get_limits() (*c3.c3objs.Quantity method*), 79
get_Lindbladian() (*c3.libraries.chip.Qubit method*), 102
get_Lindbladian() (*c3.libraries.chip.Resonator method*), 102
get_Lindbladian() (*c3.libraries.chip.Transmon method*), 104
get_Lindbladians() (*c3.model.Model method*), 83
get_minimum_phi_var() (*c3.libraries.chip.CShuntFluxQubit method*), 99
get_n_variable() (*c3.libraries.chip.CShuntFluxQubitCos method*), 100
get_noise() (*c3.generator.devices.Additive_Noise method*), 88
get_noise() (*c3.generator.devices.DC_Noise method*), 89
get_noise() (*c3.generator.devices.Pink_Noise method*), 92
get_not_opt_params() (*c3.parametermap.ParameterMap method*), 85
get_opt_limits() (*c3.parametermap.ParameterMap method*), 85
get_opt_map() (*c3.parametermap.ParameterMap method*), 85
get_opt_units() (*c3.parametermap.ParameterMap method*), 85
get_opt_value() (*c3.c3objs.Quantity method*), 80
get_optimizable_parameters() (*c3.signal.gates.Instruction method*), 123
get_parameter() (*c3.parametermap.ParameterMap method*), 85
get_parameter_dict() (*c3.parametermap.ParameterMap method*), 85
get_parameters() (*c3.parametermap.ParameterMap method*), 85
get_parameters_scaled() (*c3.parametermap.ParameterMap method*), 86
get_perfect_gates() (*c3.experiment.Experiment method*), 81
get_phase_variable() (*c3.libraries.chip.CShuntFluxQubitCos method*), 100
get_potential_function() (*c3.libraries.chip.CShuntFluxQubit method*), 99
get_potential_function() (*c3.libraries.chip.Fluxonium method*), 101
get_prefactors() (*c3.libraries.chip.TransmonExpanded method*), 104
get_Q() (*c3.generator.devices.AWG method*), 88
get_qubit_freqs() (*c3.model.Model method*), 84
get_sequence() (*in module c3.qiskit.c3_backend_utils*), 135
get_shape_values() (*c3.signal.pulse.Envelope method*), 123
get_shape_values() (*c3.signal.pulse.EnvelopeNetZero method*), 124
get_sparse_Hamiltonian() (*c3.model.Model method*), 84

get_sparse_Hamiltonians() (*c3.model.Model method*), 84
get_state_indeces() (*c3.model.Model method*), 84
get_state_index() (*c3.model.Model method*), 84
get_tf_log_level() (*in module c3.utils.tf_utils*), 127
get_third_order_prefactor() (*c3.libraries.chip.CShuntFluxQubit method*), 99
get_timings() (*c3.signal.gates.Instruction method*), 123
get_transformed_hamiltonians() (*c3.libraries.chip.PhysicalComponent method*), 101
get_value() (*c3.c3objs.Quantity method*), 80
get_VZ() (*c3.experiment.Experiment method*), 81
goal_run() (*c3.optimizers.calibration.Calibration method*), 117
goal_run() (*c3.optimizers.modellearning.ModelLearning method*), 119
goal_run() (*c3.optimizers.optimalcontrol.OptimalControl method*), 117
goal_run() (*c3.optimizers.optimizer.Optimizer method*), 120
goal_run_with_grad() (*c3.optimizers.modellearning.ModelLearning method*), 119
goal_run_with_grad() (*c3.optimizers.optimizer.Optimizer method*), 120
grid2D() (*in module c3.libraries.algorithms*), 96

H

hamiltonian_reg_deco() (*in module c3.libraries.hamiltonians*), 112
high_std() (*in module c3.libraries.sampling*), 114
HighpassExponential (*class in c3.generator.devices*), 91
HighpassFilter (*class in c3.generator.devices*), 91
hilbert_space_kron() (*in module c3.utils.qt_utils*), 124
hjson_decode() (*in module c3.c3objs*), 80
hjson_encode() (*in module c3.c3objs*), 80

I

Id_like() (*in module c3.utils.tf_utils*), 127
init_exponentiated_vars() (*c3.libraries.chip.CShuntFluxQubitCos method*), 100
init_Hs() (*c3.libraries.chip.Coupling method*), 100
init_Hs() (*c3.libraries.chip.CShuntFluxQubit method*), 99
init_Hs() (*c3.libraries.chip.CShuntFluxQubitCos method*), 100
init_Hs() (*c3.libraries.chip.Drive method*), 101
init_Hs() (*c3.libraries.chip.Qubit method*), 102
init_Hs() (*c3.libraries.chip.Resonator method*), 102
init_Hs() (*c3.libraries.chip.SNAIL method*), 103
init_Hs() (*c3.libraries.chip.Transmon method*), 104
init_Hs() (*c3.libraries.chip.TransmonExpanded method*), 105
init_Ls() (*c3.libraries.chip.Qubit method*), 102
init_Ls() (*c3.libraries.chip.Resonator method*), 102
init_Ls() (*c3.libraries.chip.Transmon method*), 104
initialise() (*c3.libraries.tasks.InitialiseGround method*), 115
InitialiseGround (*class in c3.libraries.tasks*), 115
insert_mat_kron() (*in module c3.utils.qt_utils*), 125
Instruction (*class in c3.signal.gates*), 122
int_XX() (*in module c3.libraries.hamiltonians*), 112
int_YY() (*in module c3.libraries.hamiltonians*), 113
inverseC() (*in module c3.utils.qt_utils*), 125

J

jsonify_list() (*in module c3.c3objs*), 80

K

kron_ids() (*in module c3.utils.qt_utils*), 125

L

lbfgs() (*in module c3.libraries.algorithms*), 96
lbfgs_grad_free() (*in module c3.libraries.algorithms*), 96
leakage_RB() (*in module c3.libraries.fidelities*), 109
learn_model() (*c3.optimizers.modellearning.ModelLearning method*), 119
lindbladian_average_infid() (*in module c3.libraries.fidelities*), 109
lindbladian_average_infid_set() (*in module c3.libraries.fidelities*), 110
lindbladian_epc_analytical() (*in module c3.libraries.fidelities*), 110
lindbladian_population() (*in module c3.libraries.fidelities*), 110
lindbladian_RB_left() (*in module c3.libraries.fidelities*), 109
lindbladian_RB_right() (*in module c3.libraries.fidelities*), 109
lindbladian_unitary_infid() (*in module c3.libraries.fidelities*), 110
lindbladian_unitary_infid_set() (*in module c3.libraries.fidelities*), 110
LineComponent (*class in c3.libraries.chip*), 101
list_parameters() (*c3.model.Model method*), 84
LO (*class in c3.generator.devices*), 92
load_best() (*c3.optimizers.optimizer.Optimizer method*), 120

load_model_parameters() (*c3.optimizers.optimalcontrol.OptimalControl method*), 117
load_quick_setup() (*c3.experiment.Experiment method*), 81
load_values() (*c3.parametermap.ParameterMap method*), 86
log_best_unitary() (*c3.optimizers.optimizer.Optimizer method*), 120
log_parameters() (*c3.optimizers.optimizer.BaseLogger method*), 119
log_parameters() (*c3.optimizers.optimizer.Optimizer method*), 120
log_parameters() (*c3.optimizers.optimizer.TensorBoardLogger method*), 121
log_pickle() (*c3.optimizers.calibration.Calibration method*), 118
log_setup() (*c3.optimizers.calibration.Calibration method*), 118
log_setup() (*c3.optimizers.modellearning.ModelLearning method*), 119
log_setup() (*c3.optimizers.optimalcontrol.OptimalControl method*), 117
log_setup() (*in module c3.utils.utils*), 130
log_shapes() (*c3.generator.devices.AWG method*), 88
LONoise (*class in c3.generator.devices*), 92
lookup_gate() (*c3.experiment.Experiment method*), 81
lookup_gradient() (*c3.optimizers.optimizer.Optimizer method*), 120

M

make_gate_str() (*in module c3.qiskit.c3_backend_utils*), 136
MAX_QUBITS_MEMORY (*c3.qiskit.c3_backend.C3QasmPerfectSimulator attribute*), 131
MAX_QUBITS_MEMORY (*c3.qiskit.c3_backend.C3QasmPhysicsSimulator attribute*), 132
mean_dist() (*in module c3.libraries.estimators*), 107
mean_exp_stds_dist() (*in module c3.libraries.estimators*), 107
mean_sim_stds_dist() (*in module c3.libraries.estimators*), 107
MeasurementRescale (*class in c3.libraries.tasks*), 116
median_dist() (*in module c3.libraries.estimators*), 107
Mixer (*class in c3.generator.devices*), 92
Model (*class in c3.model*), 83
ModelLearning (*class in c3.optimizers.modellearning*), 118
module
    c3, 87
    c3.c3objs, 79
    c3.experiment, 80
    c3.generator, 95
    c3.generator.devices, 87
    c3.generator.generator, 94
    c3.libraries, 116
    c3.libraries.algorithms, 95
    c3.libraries.chip, 99
    c3.libraries.constants, 105
    c3.libraries.envelopes, 105
    c3.libraries.estimators, 107
    c3.libraries.fidelities, 108
    c3.libraries.hamiltonians, 112
    c3.libraries.sampling, 114
    c3.libraries.tasks, 115
    c3.main, 87
    c3.model, 83
    c3.optimizers, 122
    c3.optimizers.calibration, 117
    c3.optimizers.modellearning, 118
    c3.optimizers.optimalcontrol, 116
    c3.optimizers.optimizer, 119
    c3.optimizers.sensitivity, 121
    c3.parametermap, 85
    c3.qiskit, 136
    c3.qiskit.c3_backend, 131
    c3.qiskit.c3_backend_utils, 135
    c3.qiskit.c3_exceptions, 134
    c3.qiskit.c3_job, 134
    c3.qiskit.c3_provider, 134
    c3.signal, 124
    c3.signal.gates, 122
    c3.signal.pulse, 123
    c3.utils, 131
    c3.utils.log_reader, 130
    c3.utils.qt_utils, 124
    c3.utils.tf_utils, 127
    c3.utils.utils, 130

N

neg_loglkh_binom() (*in module c3.libraries.estimators*), 107
neg_loglkh_binom_norm() (*in module c3.libraries.estimators*), 107
neg_loglkh_gauss() (*in module c3.libraries.estimators*), 108
neg_loglkh_gauss_norm() (*in module c3.libraries.estimators*), 108
neg_loglkh_gauss_norm_sum() (*in module c3.libraries.estimators*), 108
neg_loglkh_multinom() (*in module c3.libraries.estimators*), 108
neg_loglkh_multinom_norm() (*in module c3.libraries.estimators*), 108
no_drive() (*in module c3.libraries.envelopes*), 107
np_kron_n() (*in module c3.utils.qt_utils*), 125
num3str() (*in module c3.utils.utils*), 131
numpy() (*c3.c3objs.Quantity method*), 80

O

oneplusone() (*in module c3.libraries.algorithms*), 97
open_system_deco() (*in module c3.libraries.fidelities*), 111
OptimalControl (*class in c3.optimizers.optimalcontrol*), 116
optimize_controls() (*c3.optimizers.calibration.Calibration method*), 118
optimize_controls() (*c3.optimizers.optimalcontrol.OptimalControl method*), 117
Optimizer (*class in c3.optimizers.optimizer*), 119
orbit_infid() (*in module c3.libraries.fidelities*), 111

P

pad_matrix() (*in module c3.utils.qt_utils*), 125
ParameterMap (*class in c3.parametermap*), 85
pauli_basis() (*in module c3.utils.qt_utils*), 125
perfect_cliffords() (*in module c3.utils.qt_utils*), 125
perfect_parametric_gate() (*in module c3.utils.qt_utils*), 125
perfect_single_q_parametric_gate() (*in module c3.utils.qt_utils*), 126
PhysicalComponent (*class in c3.libraries.chip*), 101
Pink_Noise (*class in c3.generator.devices*), 92
population() (*in module c3.libraries.fidelities*), 111
populations() (*c3.experiment.Experiment method*), 81
populations() (*in module c3.libraries.fidelities*), 111
print_parameters() (*c3.parametermap.ParameterMap method*), 86
process() (*c3.experiment.Experiment method*), 82
process() (*c3.generator.devices.Additive_Noise method*), 88
process() (*c3.generator.devices.Crosstalk method*), 89
process() (*c3.generator.devices.DC_Offset method*), 89
process() (*c3.generator.devices.DigitalToAnalog method*), 90
process() (*c3.generator.devices.Filter method*), 90
process() (*c3.generator.devices.FluxTuning method*), 91
process() (*c3.generator.devices.HighpassFilter method*), 92
process() (*c3.generator.devices.LO method*), 92
process() (*c3.generator.devices.LONoise method*), 92
process() (*c3.generator.devices.Mixer method*), 92
process() (*c3.generator.devices.Response method*), 93
process() (*c3.generator.devices.ResponseFFT method*), 93
process() (*c3.generator.devices.StepFuncFilter method*), 93
process() (*c3.generator.devices.VoltsToHertz method*), 94
projector() (*in module c3.utils.qt_utils*), 126
pwc() (*in module c3.libraries.envelopes*), 107
pwc_shape() (*in module c3.libraries.envelopes*), 107
pwc_shape_plateau() (*in module c3.libraries.envelopes*), 107
pwc_symmetric() (*in module c3.libraries.envelopes*), 107

Q

Quantity (*class in c3.c3objs*), 79
Qubit (*class in c3.libraries.chip*), 101
quick_setup() (*c3.experiment.Experiment method*), 82
quick_setup() (*c3.signal.gates.Instruction method*), 123

R

ramsey_echo_sequence() (*in module c3.utils.qt_utils*), 126
ramsey_sequence() (*in module c3.utils.qt_utils*), 126
random_sample() (*in module c3.libraries.sampling*), 115
RB() (*in module c3.libraries.fidelities*), 108
read_config() (*c3.experiment.Experiment method*), 82
read_config() (*c3.generator.generator.Generator method*), 94
read_config() (*c3.model.Model method*), 84
read_config() (*c3.parametermap.ParameterMap method*), 86
read_data() (*c3.optimizers.modellearning.ModelLearning method*), 119
Readout (*class in c3.generator.devices*), 92
readout() (*c3.generator.devices.Readout method*), 92
rect() (*in module c3.libraries.envelopes*), 107
replace_logdir() (*c3.optimizers.optimizer.Optimizer method*), 120
replace_symlink() (*in module c3.utils.utils*), 131
rescale() (*c3.libraries.tasks.MeasurementRescale method*), 116
Resonator (*class in c3.libraries.chip*), 102
resonator() (*in module c3.libraries.hamiltonians*), 113
Response (*class in c3.generator.devices*), 93
ResponseFFT (*class in c3.generator.devices*), 93
result() (*c3.qiskit.c3_job.C3Job method*), 134
rms_dist() (*in module c3.libraries.estimators*), 108
rms_exp_stds_dist() (*in module c3.libraries.estimators*), 108
rms_sim_stds_dist() (*in module c3.libraries.estimators*), 108
rotation() (*in module c3.utils.qt_utils*), 126
run() (*c3.qiskit.c3_backend.C3QasmSimulator method*), 133
run_cfg() (*in module c3.main*), 87
run_experiment() (*c3.qiskit.c3_backend.C3QasmPerfectSimulator method*), 131

run_experiment() (*c3.qiskit.c3_backend.C3QasmPhysicsSimulator method*), 132
run_experiment() (*c3.qiskit.c3_backend.C3QasmSimulator method*), 133

S

sampling_reg_deco() (*in module c3.libraries.sampling*), 115
select_from_data() (*c3.optimizers.modellearning.ModelLearning method*), 119
Sensitivity (*class in c3.optimizers.sensitivity*), 121
sensitivity() (*c3.optimizers.sensitivity.Sensitivity method*), 122
set_algorithm() (*c3.optimizers.optimizer.Optimizer method*), 120
set_callback_fids() (*c3.optimizers.optimalcontrol.OptimalControl method*), 117
set_components() (*c3.model.Model method*), 84
set_created_by() (*c3.experiment.Experiment method*), 82
set_created_by() (*c3.optimizers.optimizer.Optimizer method*), 121
set_deco() (*in module c3.libraries.fidelities*), 111
set_dephasing_strength() (*c3.model.Model method*), 84
set_device_config() (*c3.qiskit.c3_backend.C3QasmSimulator method*), 133
set_dressed() (*c3.model.Model method*), 84
set_enable_store_unitaries() (*c3.experiment.Experiment method*), 82
set_eval_func() (*c3.optimizers.calibration.Calibration method*), 118
set_exp() (*c3.optimizers.optimizer.Optimizer method*), 121
set_fid_func() (*c3.optimizers.optimalcontrol.OptimalControl method*), 117
set_FR() (*c3.model.Model method*), 84
set_limits() (*c3.c3objs.Quantity method*), 80
set_lindbladian() (*c3.model.Model method*), 84
set_logdir() (*c3.optimizers.optimizer.TensorBoardLogger method*), 121
set_max_excitations() (*c3.model.Model method*), 84
set_opt_gates() (*c3.experiment.Experiment method*), 82
set_opt_gates_seq() (*c3.experiment.Experiment method*), 82
set_opt_map() (*c3.parametermap.ParameterMap method*), 86
set_opt_value() (*c3.c3objs.Quantity method*), 80
set_parameters() (*c3.parametermap.ParameterMap method*), 86
set_parameters_scaled() (*c3.parametermap.ParameterMap method*), 86
set_prop_method() (*c3.experiment.Experiment method*), 82
set_subspace_index() (*c3.libraries.chip.PhysicalComponent method*), 101
set_tasks() (*c3.model.Model method*), 84
set_tf_log_level() (*in module c3.utils.tf_utils*), 127
set_value() (*c3.c3objs.Quantity method*), 80
show_table() (*in module c3.utils.log_reader*), 130
single_eval() (*in module c3.libraries.algorithms*), 97
single_length_RB() (*in module c3.utils.qt_utils*), 126
SkinEffectResponse (*class in c3.generator.devices*), 93
slepian_fourier() (*in module c3.libraries.envelopes*), 107
SNAIL (*class in c3.libraries.chip*), 103
start_log() (*c3.optimizers.optimizer.BaseLogger method*), 119
start_log() (*c3.optimizers.optimizer.Optimizer method*), 121
start_log() (*c3.optimizers.optimizer.TensorBoardLogger method*), 121
state_deco() (*in module c3.libraries.fidelities*), 111
state_transfer_infid() (*in module c3.libraries.fidelities*), 111
state_transfer_infid_set() (*in module c3.libraries.fidelities*), 111
status() (*c3.qiskit.c3_job.C3Job method*), 134
std_of_diffs() (*in module c3.libraries.estimators*), 108
step_response_function() (*c3.generator.devices.ExponentialIIR method*), 90
step_response_function() (*c3.generator.devices.HighpassExponential method*), 91
step_response_function() (*c3.generator.devices.SkinEffectResponse method*), 93
step_response_function() (*c3.generator.devices.StepFuncFilter method*), 93
StepFuncFilter (*class in c3.generator.devices*), 93
store_Udict() (*c3.experiment.Experiment method*), 82
store_values() (*c3.parametermap.ParameterMap method*), 86
str_parameters() (*c3.parametermap.ParameterMap method*), 86
submit() (*c3.qiskit.c3_job.C3Job method*), 134
subtract() (*c3.c3objs.Quantity method*), 80
super_to_choi() (*in module c3.utils.tf_utils*), 127

sweep() (*in module c3.libraries.algorithms*), 97

T

T1_sequence() (*in module c3.utils.qt_utils*), 124
Task (*class in c3.libraries.tasks*), 116
task_deco() (*in module c3.libraries.tasks*), 116
TensorBoardLogger (*class in c3.optimizers.optimizer*), 121
tf_abs() (*in module c3.utils.tf_utils*), 127
tf_abs_squared() (*in module c3.utils.tf_utils*), 127
tf_adadelta() (*in module c3.libraries.algorithms*), 97
tf_adam() (*in module c3.libraries.algorithms*), 98
tf_ave() (*in module c3.utils.tf_utils*), 127
tf_average_fidelity() (*in module c3.utils.tf_utils*), 127
tf_choi_to_chi() (*in module c3.utils.tf_utils*), 127
tf_convolve() (*in module c3.utils.tf_utils*), 128
tf_diff() (*in module c3.utils.tf_utils*), 128
tf_dm_to_vec() (*in module c3.utils.tf_utils*), 128
tf_dmdm_fid() (*in module c3.utils.tf_utils*), 128
tf_dmket_fid() (*in module c3.utils.tf_utils*), 128
tf_ketket_fid() (*in module c3.utils.tf_utils*), 128
tf_kron() (*in module c3.utils.tf_utils*), 128
tf_limit_gpu_memory() (*in module c3.utils.tf_utils*), 128
tf_list_avail_devices() (*in module c3.utils.tf_utils*), 128
tf_log10() (*in module c3.utils.tf_utils*), 128
tf_log_level_info() (*in module c3.utils.tf_utils*), 128
tf_matmul_left() (*in module c3.utils.tf_utils*), 128
tf_matmul_n() (*in module c3.utils.tf_utils*), 128
tf_matmul_right() (*in module c3.utils.tf_utils*), 128
tf_measure_operator() (*in module c3.utils.tf_utils*), 129
tf_project_to_comp() (*in module c3.utils.tf_utils*), 129
tf_rmsprop() (*in module c3.libraries.algorithms*), 98
tf_setup() (*in module c3.utils.tf_utils*), 129
tf_sgd() (*in module c3.libraries.algorithms*), 98
tf_spost() (*in module c3.utils.tf_utils*), 129
tf_spre() (*in module c3.utils.tf_utils*), 129
tf_state_to_dm() (*in module c3.utils.tf_utils*), 129
tf_super() (*in module c3.utils.tf_utils*), 129
tf_super_to_fid() (*in module c3.utils.tf_utils*), 129
tf_superoper_average_fidelity() (*in module c3.utils.tf_utils*), 129
tf_superoper_unitary_overlap() (*in module c3.utils.tf_utils*), 129
tf_unitary_overlap() (*in module c3.utils.tf_utils*), 129
tf_vec_to_dm() (*in module c3.utils.tf_utils*), 130
third_order() (*in module c3.libraries.hamiltonians*), 113
Transmon (*class in c3.libraries.chip*), 103
TransmonExpanded (*class in c3.libraries.chip*), 104
trapezoid() (*in module c3.libraries.envelopes*), 107
two_qubit_gate_tomography() (*in module c3.utils.qt_utils*), 127

U

unitary_deco() (*in module c3.libraries.fidelities*), 111
unitary_infid() (*in module c3.libraries.fidelities*), 111
unitary_infid_set() (*in module c3.libraries.fidelities*), 112
update_dressed() (*c3.model.Model method*), 84
update_drift_eigen() (*c3.model.Model method*), 84
update_Hamiltonians() (*c3.model.Model method*), 84
update_Lindbladians() (*c3.model.Model method*), 84
update_model() (*c3.model.Model method*), 85
update_parameters() (*c3.parametermap.ParameterMap method*), 86

V

VoltsToHertz (*class in c3.generator.devices*), 94

W

write_config() (*c3.experiment.Experiment method*), 83
write_config() (*c3.generator.devices.Device method*), 90
write_config() (*c3.generator.generator.Generator method*), 95
write_config() (*c3.model.Model method*), 85
write_config() (*c3.parametermap.ParameterMap method*), 86
write_config() (*c3.signal.pulse.Carrier method*), 123
write_config() (*c3.signal.pulse.Envelope method*), 123
write_params() (*c3.optimizers.optimizer.TensorBoardLogger method*), 121

X

x_drive() (*in module c3.libraries.hamiltonians*), 113
xy_basis() (*in module c3.utils.qt_utils*), 127

Y

y_drive() (*in module c3.libraries.hamiltonians*), 113

Z

z_drive() (*in module c3.libraries.hamiltonians*), 113