

---

**C3**

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

**Jan 06, 2022**



# API DOCUMENTATION:

<b>1</b>	<b>Introduction to <math>C^3</math> Toolset</b>	<b>3</b>
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
<b>2</b>	<b>Parameter handling</b>	<b>5</b>
2.1	The opt_map . . . . .	6
2.2	Optimizer scaling . . . . .	8
2.3	Storing and reading . . . . .	8
<b>3</b>	<b>Setup of a two-qubit chip with <math>C^3</math></b>	<b>11</b>
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
<b>4</b>	<b>Dynamics</b>	<b>23</b>
<b>5</b>	<b>Open-loop optimal control</b>	<b>27</b>
<b>6</b>	<b>Entangling gate on two coupled qubits</b>	<b>31</b>
6.1	Imports . . . . .	31
6.2	Model components . . . . .	32
6.3	Control signals . . . . .	33
6.4	Gates-set and Parameter map . . . . .	35
6.5	Instructions . . . . .	38
6.6	The experiment . . . . .	39
6.7	Dynamics . . . . .	40
6.8	Visualisation with qiskit circuit . . . . .	44
6.9	Open-loop optimal control . . . . .	45
6.10	Results of the optimisation . . . . .	47
6.11	Optimizing the single qubit gate on Qubit 1 . . . . .	50

<b>7</b>	<b>Simulated calibration</b>	<b>53</b>
7.1	ORBIT - Single-length randomized benchmarking	53
7.2	Communication with the experiment	54
7.3	Optimization	55
7.4	Representation of the experiment within $C^3$	56
7.5	Algorithms	59
7.6	Analysis	61
<b>8</b>	<b>Model Learning</b>	<b>63</b>
8.1	Imports	63
8.1.1	The Dataset	63
8.1.2	The Model for Model Learning	66
8.2	Define Constants	66
8.3	Model	67
8.4	Generator	67
8.5	Gateset	68
8.6	Experiment	70
8.6.1	Optimizer	70
8.6.2	Model Learning	71
8.7	Result of Model Learning	71
8.7.1	Visualisation & Analysis of Results	71
8.8	Open, Clean-up and Convert Logfiles	72
8.9	Summary of Logs	72
8.10	Plotting	73
8.11	Qubit Anharmonicity	73
8.12	Qubit Frequency	74
8.13	Goal Function	75
<b>9</b>	<b>Sensitivity Analysis</b>	<b>77</b>
<b>10</b>	<b>Logs and current optimization status</b>	<b>81</b>
<b>11</b>	<b>C3 Simulator as a backend for Qiskit Experiments</b>	<b>83</b>
11.1	Define a basic Quantum circuit	83
11.2	Get the C3 Provider and Backend	83
11.3	Run a physical device simulation using C3	84
11.4	Run Simulation and verify results on Qiskit simulator	85
<b>12</b>	<b>API Documentation</b>	<b>87</b>
12.1	C3objs	87
12.2	Experiment module	88
12.3	Model module	91
12.4	Parameter map	93
12.5	Main module	95
12.6	Module contents	95
12.7	Subpackages	95
12.7.1	Generator package	95
12.7.1.1	Submodules	95
12.7.1.2	Devices module	95
12.7.1.3	Generator module	102
12.7.1.4	Module contents	103
12.7.2	Libraries package	103
12.7.2.1	Algorithms module	103
12.7.2.2	Chip module	107
12.7.2.3	Constants module	113

12.7.2.4	Envelopes module	113
12.7.2.5	Estimators module	115
12.7.2.6	Fidelities module	116
12.7.2.7	Hamiltonians module	120
12.7.2.8	Sampling module	122
12.7.2.9	Tasks module	123
12.7.2.10	Module contents	124
12.7.3	Optimizers	124
12.7.3.1	C1 - Optimal control	124
12.7.3.2	C2 - Calibration	125
12.7.3.3	C3 - Characterization	126
12.7.3.4	Optimizer module	127
12.7.3.5	Sensitivity analysis	129
12.7.3.6	Module contents	130
12.7.4	Signal package	130
12.7.4.1	Submodules	130
12.7.4.2	Gates module	130
12.7.4.3	Pulse module	131
12.7.4.4	Module contents	132
12.7.5	Utilities package	132
12.7.5.1	Qutip utilities module	132
12.7.5.2	Tensorflow utilities module	135
12.7.5.3	Log Reader utilities module	138
12.7.5.4	Miscellaneous utilities module	138
12.7.5.5	Module contents	139
12.7.6	Qiskit modules for C3	139
12.7.6.1	C3 Backend module	139
12.7.6.2	C3 Provider module	142
12.7.6.3	C3 Job module	143
12.7.6.4	C3 Exceptions module	143
12.7.6.5	C3 Gates module	143
12.7.6.6	C3 Backend Utilities module	144
12.7.6.7	Module contents	146
<b>13</b>	<b>Indices and tables</b>	<b>147</b>
	<b>Index</b>	<b>149</b>



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.*





## 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.

## 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 identifiers.

```
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 respectively.

```
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 and 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
    0.40000000059604645
  ]
}
```

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





## 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  $\omega$  and anharmonicity  $\delta$  :

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

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 bandwidth 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()
```



## 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)

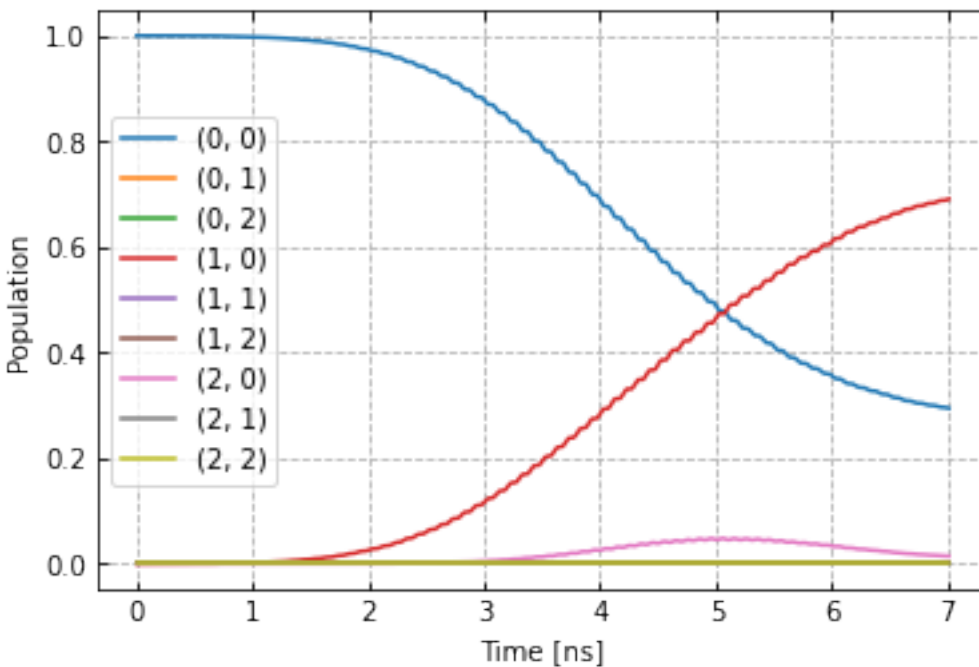
```

"""
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)
```

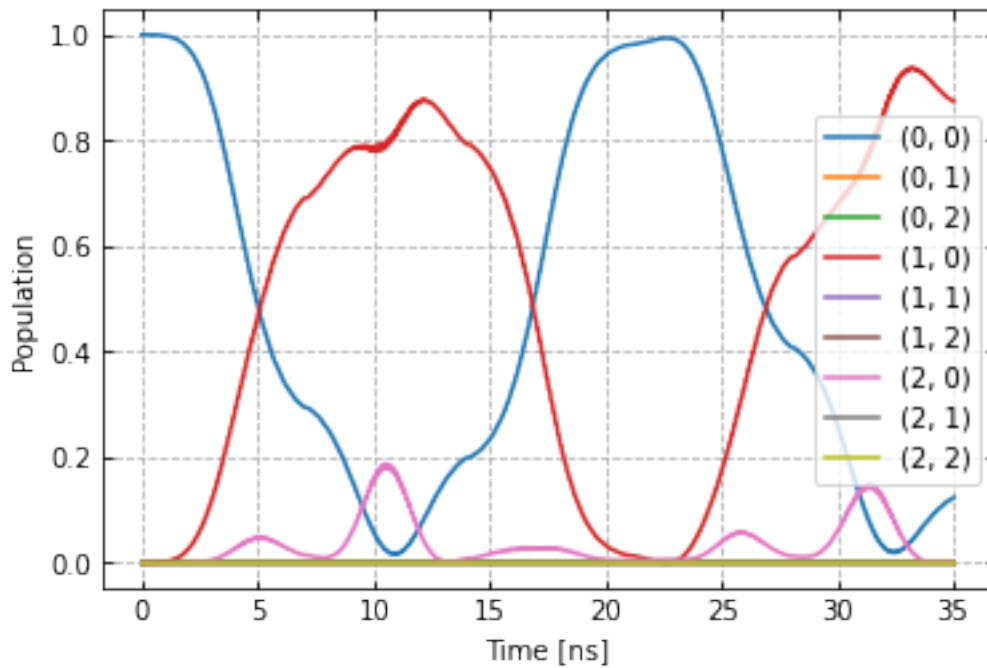


We can see an ill-defined un-optimized gate. The labels indicate qubit states in the product basis. Next we increase the number of repetitions of the same gate.

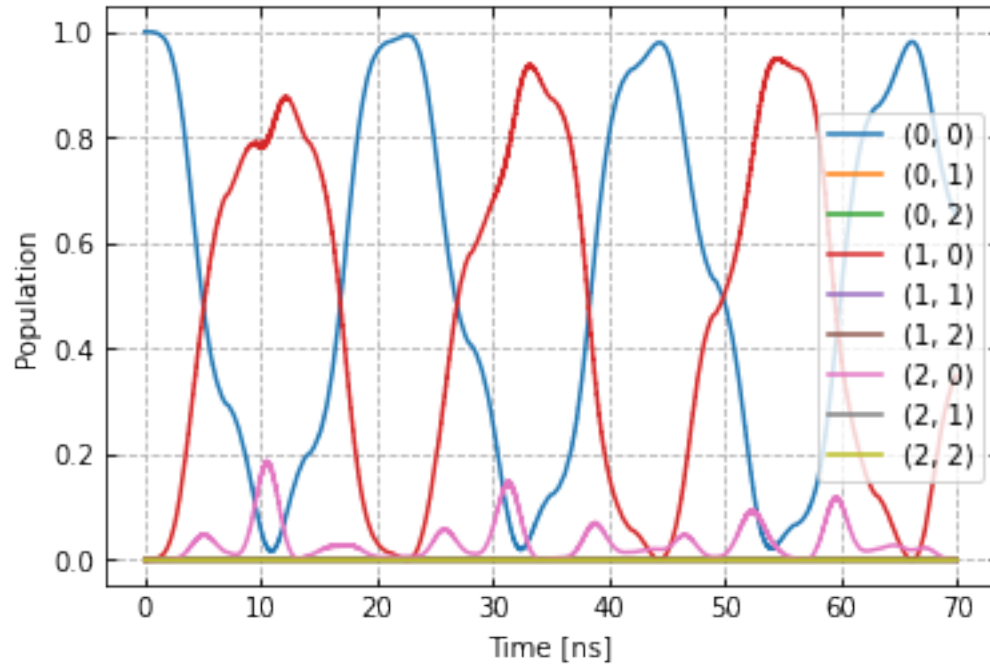
```
barely_a_seq * 10
```

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



## OPEN-LOOP OPTIMAL CONTROL

In order to improve the gate from the previous example *Setup of a two-qubit chip with C<sup>3</sup>*, 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  $|2, 0\rangle$  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 LBFQ-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()
```

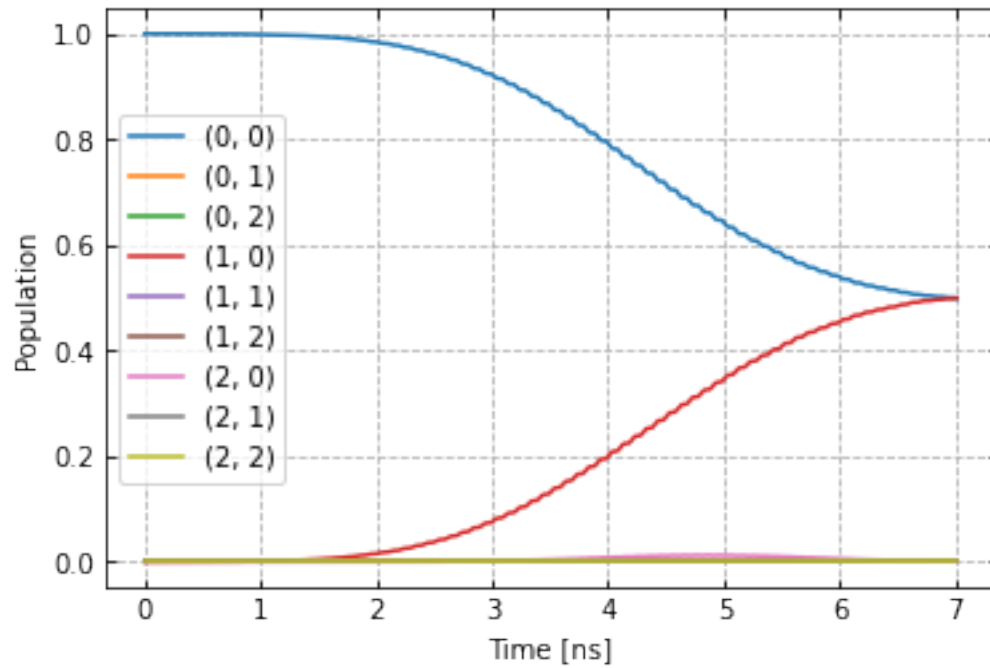
After a few steps we have improved the gate significantly, as we can check with

```
opt.current_best_goal
```

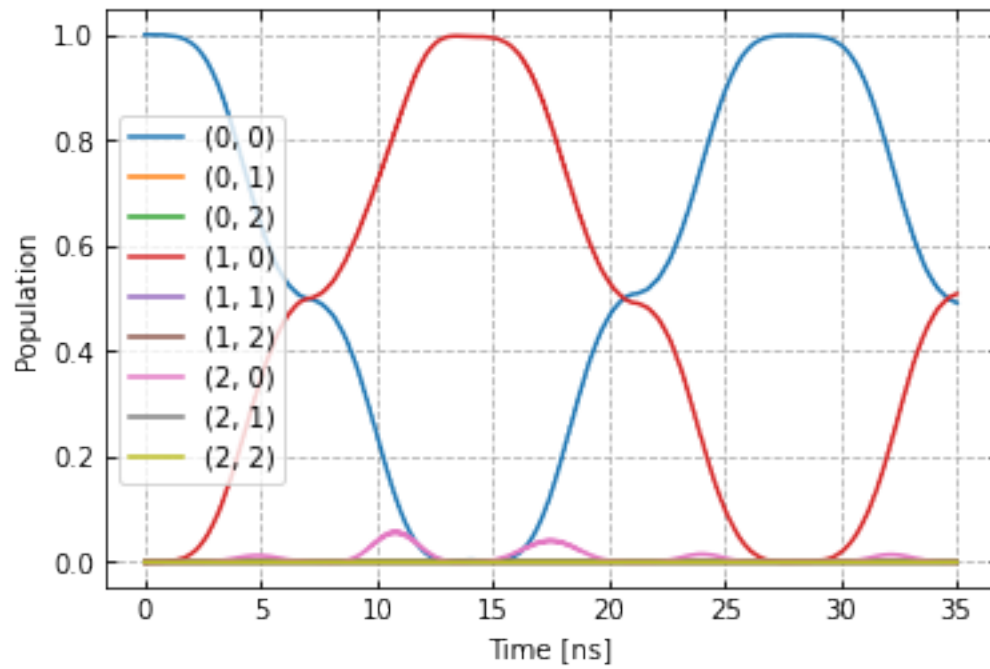
```
0.00063
```

And by looking at the same sequences as before.

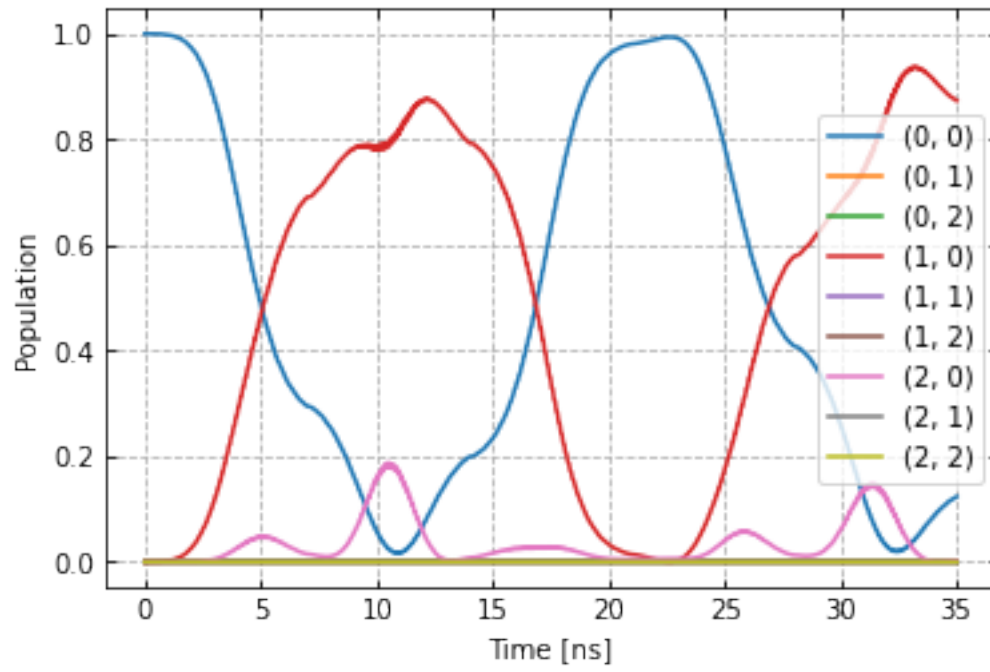
```
plot_dynamics(exp, init_state, barely_a_seq)
```



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



Compared to before the optimization.



## 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
from pprint import pprint

# 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

# Qiskit related modules
```

(continues on next page)

(continued from previous page)

```

from c3.qiskit import C3Provider
from c3.qiskit.c3_gates import RX90pGate
from qiskit import QuantumCircuit, Aer, execute
from qiskit.tools.visualization import plot_histogram

```

## 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"],

```

(continues on next page)

(continued from previous page)

```

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)

```

(continues on next page)

(continued from previous page)

```

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',
    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": lo,
        "AWG": awg,
        "DigitalToAnalog": dig_to_an,
        "Response": resp,
        "Mixer": mixer,
        "VoltsToHertz": v_to_hz
    },
    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"],
        }
    }
)

```



## 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_2Q = 45e-9
sideband = 50e6
gauss_params_2Q_1 = {
    'amp': Qty(value=0.8, min_val=0.2, max_val=3, unit="V"),
    't_final': Qty(value=t_final_2Q, min_val=0.5 * t_final_2Q, max_val=1.5 * t_final_2Q,
↪unit="s"),
    'sigma': Qty(value=t_final_2Q / 4, min_val=t_final_2Q / 8, max_val=t_final_2Q / 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_2Q_2 = {
    'amp': Qty(value=0.03, min_val=0.02, max_val=0.6, unit="V"),
    't_final': Qty(value=t_final_2Q, min_val=0.5 * t_final_2Q, max_val=1.5 * t_final_2Q,
↪unit="s"),
    'sigma': Qty(value=t_final_2Q / 4, min_val=t_final_2Q / 8, max_val=t_final_2Q / 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_env_2Q_1 = pulse.Envelope(
    name="gauss1",
    desc="Gaussian envelope on drive 1",
    params=gauss_params_2Q_1,
    shape=envelopes.gaussian_nonorm
)
gauss_env_2Q_2 = pulse.Envelope(
    name="gauss2",
    desc="Gaussian envelope on drive 2",
    params=gauss_params_2Q_2,
    shape=envelopes.gaussian_nonorm
)
```

We choose a single qubit gate time of 7ns and a gaussian envelope shape with a list of parameters.

```
t_final_1Q = 7e-9 # Time for single qubit gates
sideband = 50e6
gauss_params_single = {
    'amp': Qty(
        value=0.5,
        min_val=0.2,
```

(continues on next page)

(continued from previous page)

```

        max_val=0.6,
        unit="V"
    ),
    't_final': Qty(
        value=t_final_1Q,
        min_val=0.5 * t_final_1Q,
        max_val=1.5 * t_final_1Q,
        unit="s"
    ),
    'sigma': Qty(
        value=t_final_1Q / 4,
        min_val=t_final_1Q / 8,
        max_val=t_final_1Q / 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_env_1Q = 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 (used for the single qubit gates).

```

nodrive_env = pulse.Envelope(
    name="no_drive",
    params={
        't_final': Qty(
            value=t_final_1Q,
            min_val=0.5 * t_final_1Q,
            max_val=1.5 * t_final_1Q,
            unit="s"
        )
    }
)

```

(continues on next page)

(continued from previous page)

```

    )
    },
    shape=envelopes.no_drive
)

```

The carrier signal of each drive for the 2 Qubit gates is set to the resonance frequency of the target qubit.

```

lo_freq_q1 = freq_q1 + sideband
lo_freq_q2 = freq_q2 + sideband

carr_2Q_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_2Q_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')
    }
)

```

We specify the drive tones for the 1Q gates with an offset from the qubit frequencies. As is done in experiment, we will later adjust the resonance by modulating the envelope function.

```

carr_1Q_1 = pulse.Carrier(
    name="carrier",
    desc="Frequency of the local oscillator",
    params={
        'freq': Qty(
            value=lo_freq_q1,
            min_val=0.9 * lo_freq_q1 ,
            max_val=1.1 * lo_freq_q1 ,
            unit='Hz 2pi'
        ),
        'framechange': Qty(
            value=0.0,
            min_val= -np.pi,
            max_val= 3 * np.pi,
            unit='rad'
        )
    }
)

```

(continues on next page)

(continued from previous page)

```

carr_1Q_2 = pulse.Carrier(
    name="carrier",
    desc="Frequency of the local oscillator",
    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 controlled by qubit 1
cnot12 = gates.Instruction(
    name="cx", targets=[0, 1], t_start=0.0, t_end=t_final_2Q, channels=["d1", "d2"],
    ideal=np.array([
        [1,0,0,0],
        [0,1,0,0],
        [0,0,0,1],
        [0,0,1,0]
    ])
)
cnot12.add_component(gauss_env_2Q_1, "d1")
cnot12.add_component(carr_2Q_1, "d1")
cnot12.add_component(gauss_env_2Q_2, "d2")
cnot12.add_component(carr_2Q_2, "d2")
cnot12.comps["d1"]["carrier"].params["framechange"].set_value(
    (-sideband * t_final_2Q) * 2 * np.pi % (2 * np.pi)
)

```

We also add some typical single qubit gates to the instruction set.

```

rx90p_q1 = gates.Instruction(
    name="rx90p", targets=[0], t_start=0.0, t_end=t_final_1Q, channels=["d1", "d2"]
)
rx90p_q2 = gates.Instruction(
    name="rx90p", targets=[1], t_start=0.0, t_end=t_final_1Q, channels=["d1", "d2"]
)

```

(continues on next page)

(continued from previous page)

```

rx90p_q1.add_component(gauss_env_1Q, "d1")
rx90p_q1.add_component(carr_1Q_1, "d1")

rx90p_q2.add_component(gauss_env_1Q, "d2")
rx90p_q2.add_component(carr_1Q_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_1Q_2), "d2")
rx90p_q1.comps["d2"]["carrier"].params["framechange"].set_value(
    (-sideband * t_final_1Q) * 2 * np.pi % (2 * np.pi)
)

rx90p_q2.add_component(nodrive_env, "d1")
rx90p_q2.add_component(copy.deepcopy(carr_1Q_1), "d1")
rx90p_q2.comps["d1"]["carrier"].params["framechange"].set_value(
    (-sideband * t_final_1Q) * 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, rx90p_q1, rx90p_q2], 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()])
# print(unitaries[rx90p_q1.get_key()])

```

```

2022-01-01 20:55:15.191809: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.
    ↪cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
2022-01-01 20:55:15.193814: W tensorflow/core/platform/profile_utils/cpu_utils.cc:128]
    ↪Failed to get CPU frequency: 0 Hz

```

## 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]
 [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.
    """
    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')
```

(continues on next page)

(continued from previous page)

```

        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
    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
    -----

```

(continues on next page)

```

"""
# 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])

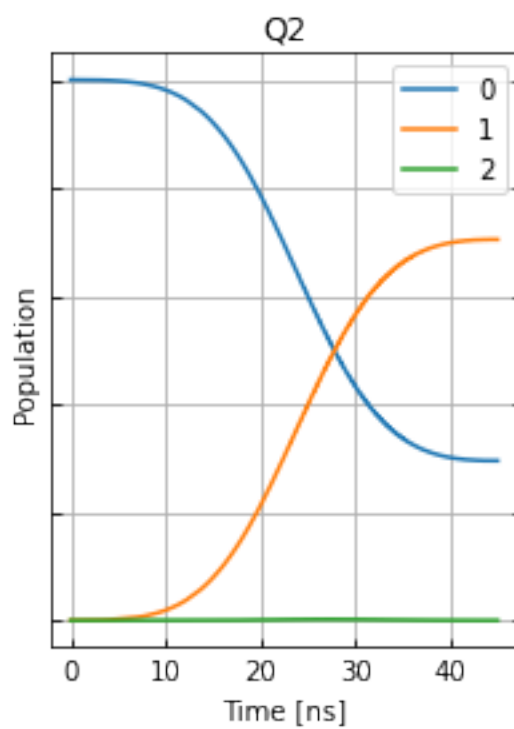
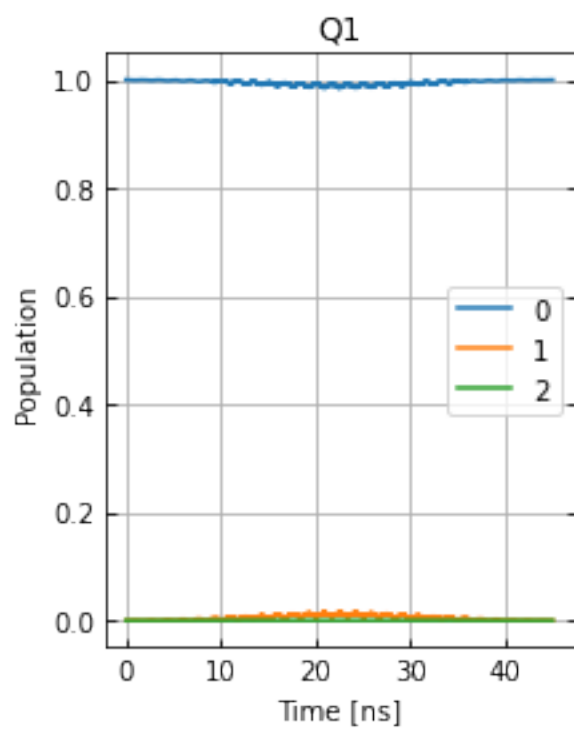
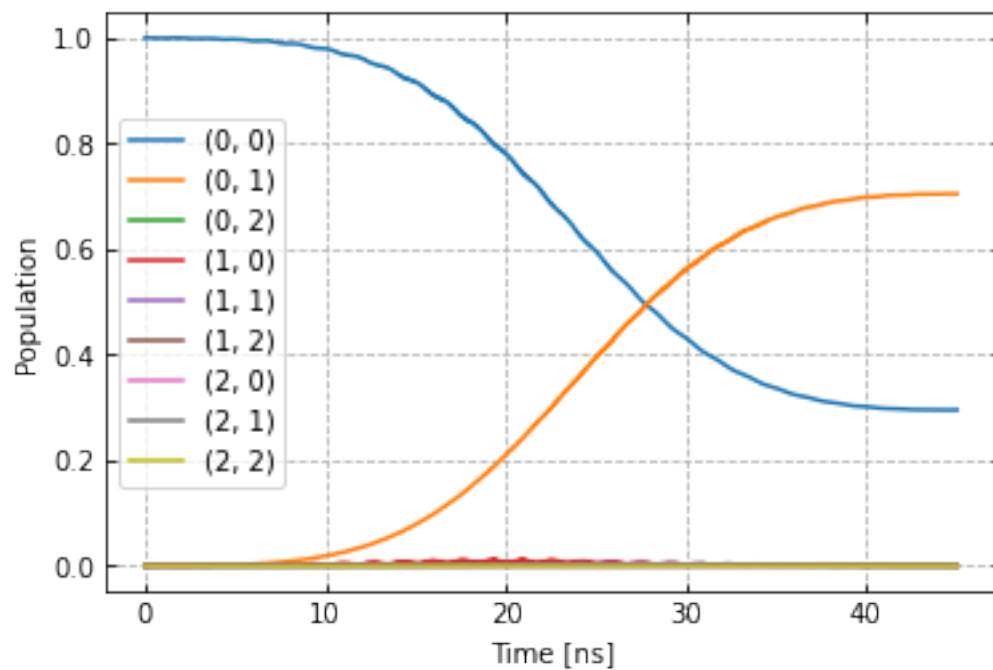
# create both subplots
titles = list(exp.pmap.model.subsystems.keys())
fig, axs = plt.subplots(1, len(splitted), sharey="all")
for idx, ax in enumerate(axs):
    ax.plot(ts / 1e-9, splitted[idx].T)
    ax.tick_params(direction="in", left=True, right=True, top=False, bottom=True)
    ax.set_xlabel("Time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

plt.tight_layout()
plt.show()

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

```





## 6.8 Visualisation with qiskit circuit

```
qc = QuantumCircuit(2, 2)
qc.append(RX90pGate(), [0])
qc.cx(0, 1)
qc.draw()
```

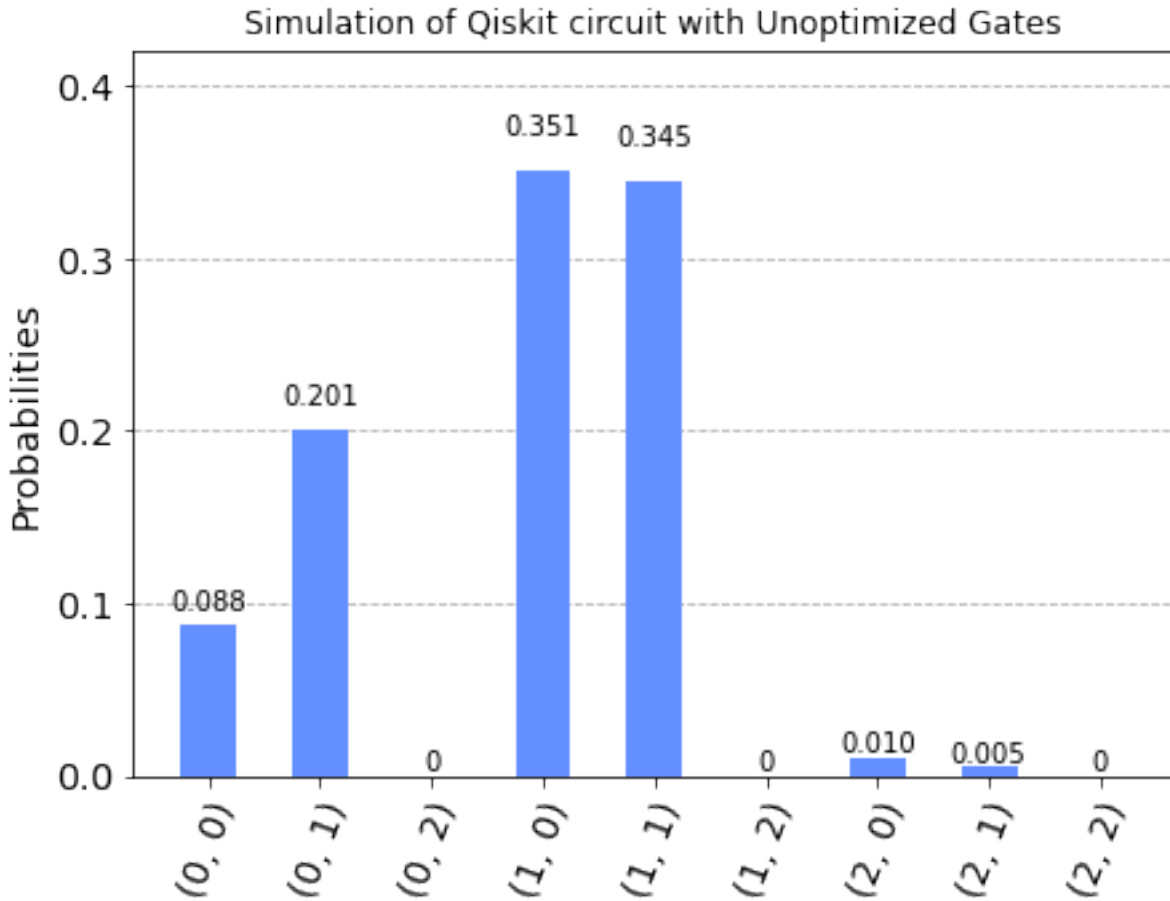
```
c3_provider = C3Provider()
c3_backend = c3_provider.get_backend("c3_qasm_physics_simulator")
c3_backend.set_c3_experiment(exp)
```

```
c3_job_unopt = c3_backend.run(qc)
result_unopt = c3_job_unopt.result()
res_pops_unopt = result_unopt.data()["state_pops"]
print("Result from unoptimized gates:")
pprint(res_pops_unopt)
```

No measurements **in** circuit "circuit-0", classical register will remain **all** zeros.

```
Result from unoptimized gates:
{'(0, 0)': 0.08793249061599799,
 '(0, 1)': 0.20140214999016118,
 '(0, 2)': 1.7916216253388795e-05,
 '(1, 0)': 0.3507070201094552,
 '(1, 1)': 0.34492529411056444,
 '(1, 2)': 5.054006496570894e-06,
 '(2, 0)': 0.009940433069486916,
 '(2, 1)': 0.005069321214083548,
 '(2, 2)': 3.20667334658816e-07}
```

```
plot_histogram(res_pops_unopt, title='Simulation of Qiskit circuit with Unoptimized Gates
→')
```



## 6.9 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")],
]
```

(continues on next page)

(continued from previous page)

```
parameter_map.set_opt_map(gateset_opt_map)

parameter_map.print_parameters()
```

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

Start the optimisation

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

```
C3:STATUS:Saving as: /var/folders/04/np4lgk2d7sq6w0dnp758sgp80000gn/T/tmpyryftkry4/c3logs/
↳ cnot12/2022_01_01_T_20_55_19/open_loop.c3log
```

The final parameters and the fidelity are

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

```
cx[0, 1]-d1-gauss1-amp           : 2.359 V
cx[0, 1]-d1-gauss1-freq_offset   : -53.252 MHz 2pi
cx[0, 1]-d1-gauss1-xy_angle     : 587.818 mrad
```

(continues on next page)

(continued from previous page)

```

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

```

```
0.0055213431696764514
```

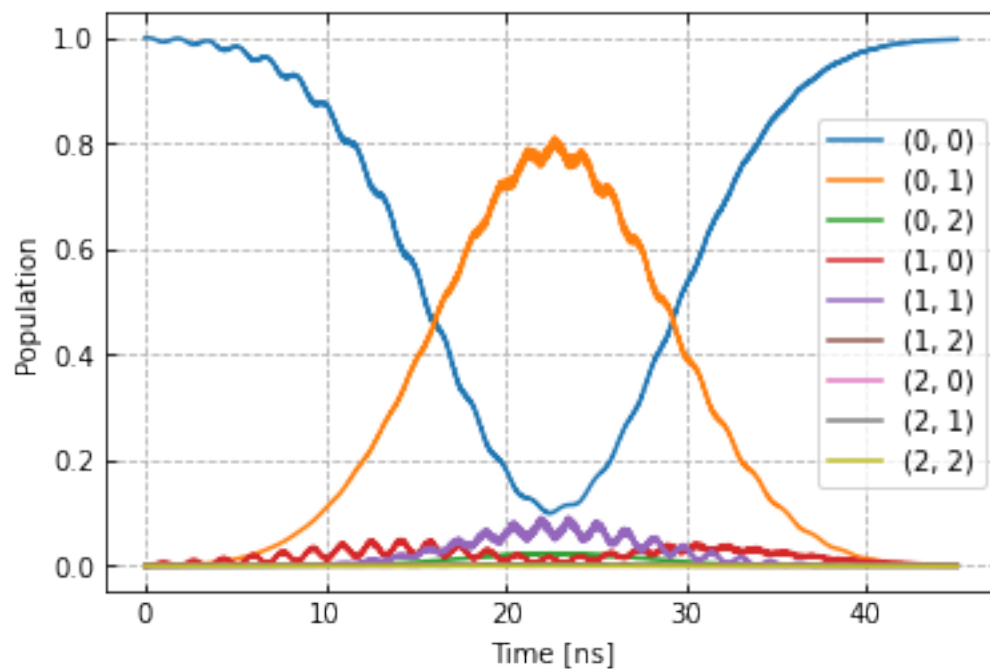
## 6.10 Results of the optimisation

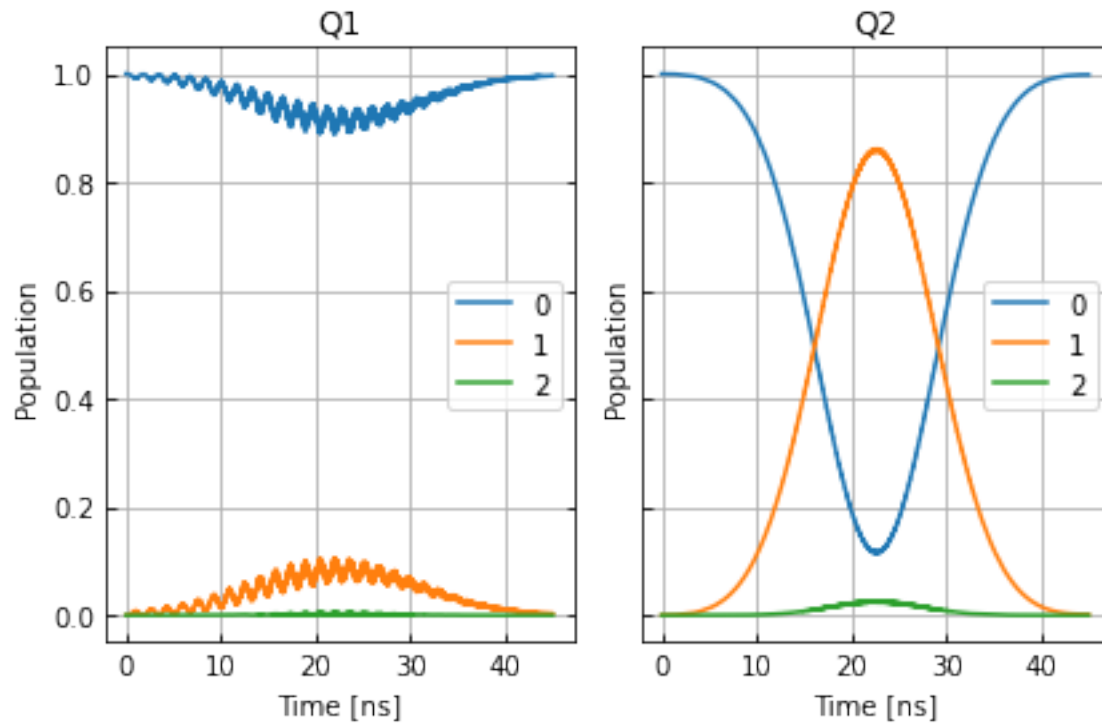
Plotting the dynamics with the same initial state:

```

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

```



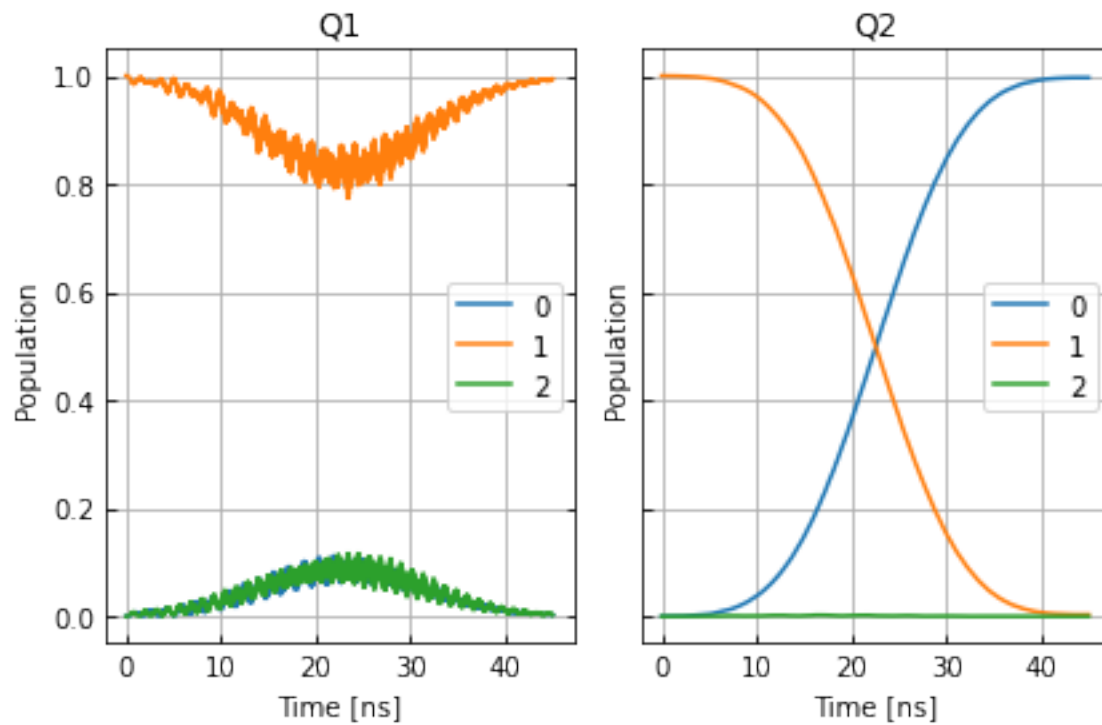
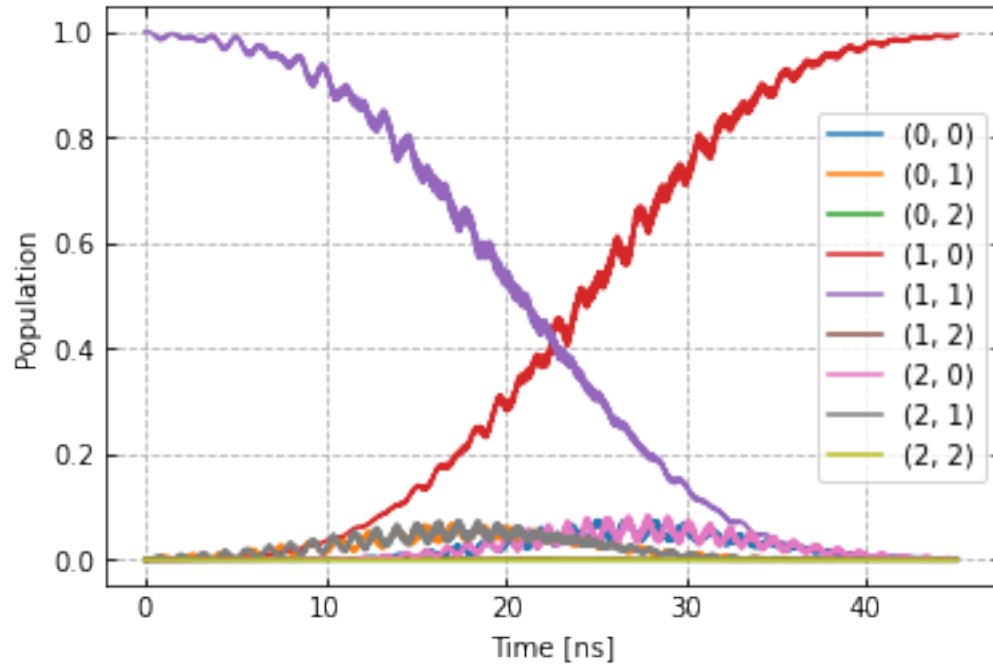


Now we plot the dynamics for the control in the excited state.

```
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.

## 6.11 Optimizing the single qubit gate on Qubit 1

```

opt_gates = [rx90p_q1.get_key()]
gateset_opt_map=[
    [
        (rx90p_q1.get_key(), "d1", "gauss", "amp"),
    ],
    [
        (rx90p_q1.get_key(), "d1", "gauss", "freq_offset"),
    ],
    [
        (rx90p_q1.get_key(), "d1", "gauss", "xy_angle"),
    ],
    [
        (rx90p_q1.get_key(), "d1", "gauss", "delta"),
    ],
    [
        (rx90p_q1.get_key(), "d1", "carrier", "framechange"),
    ]
]
parameter_map.set_opt_map(gateset_opt_map)
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
rx90p[0]-d1-carrier-framechange : 0.000 rad

```

```

opt_1Q = OptimalControl(
    dir_path=log_dir,
    fid_func=fidelities.unitary_infid_set,
    fid_subspace=["Q1", "Q2"],
    pmap=parameter_map,
    algorithm=algorithms.lbfgs,
    options={
        "maxfun": 25
    },
    run_name="rx90p_q1"
)

```

```

exp.set_opt_gates(opt_gates)
opt_1Q.set_exp(exp)
opt_1Q.optimize_controls()

```

```

C3:STATUS:Saving as: /var/folders/04/np4lgk2d7sq6w0dpn758sgp80000gn/T/tmpyftkry4/c3logs/
↪rx90p_q1/2022_01_01_T_20_58_09/open_loop.c3log

```

```

parameter_map.print_parameters()
print(opt_1Q.current_best_goal)

```



```

rx90p[0]-d1-gauss-amp           : 390.140 mV
rx90p[0]-d1-gauss-freq_offset   : -52.986 MHz 2pi
rx90p[0]-d1-gauss-xy_angle     : -195.771 mrad
rx90p[0]-d1-gauss-delta        : -964.376 m
rx90p[0]-d1-carrier-framechange : -282.109 mrad

```

```
0.00575481536619904
```

Before running the qiskit simulation, we must call `set_opt_gates()` to ensure propagators are calculated for all the required gates

```
exp.set_opt_gates([rx90p_q1.get_key(), cnot12.get_key()])
```

```

c3_job_opt = c3_backend.run(qc)
result_opt = c3_job_opt.result()
res_pops_opt = result_opt.data()["state_pops"]
print("Result from gates:")
pprint(res_pops_opt)

```

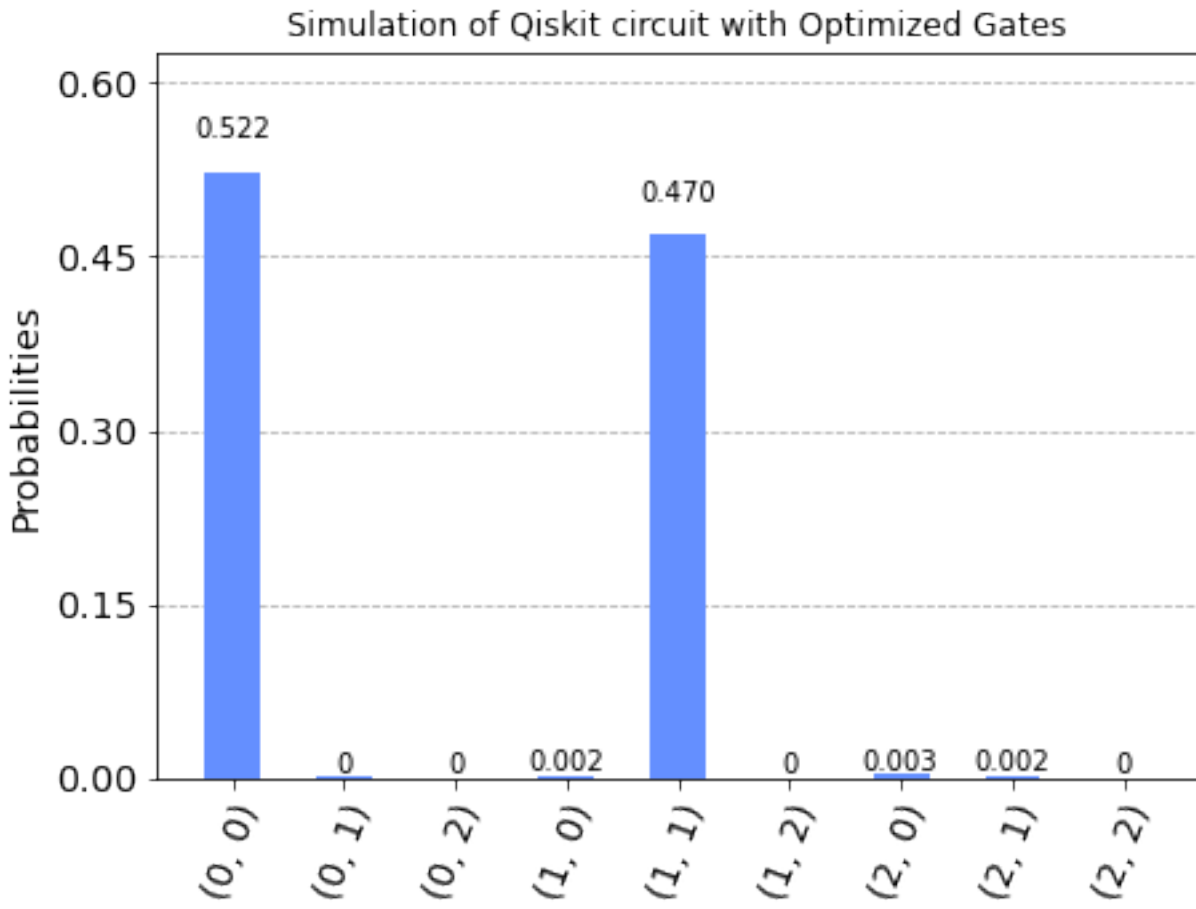
No measurements in circuit "circuit-0", classical register will remain all zeros.

```

Result from gates:
{'(0, 0)': 0.522074672738806,
 '(0, 1)': 0.0009262330305873641,
 '(0, 2)': 5.58398828418534e-07,
 '(1, 0)': 0.002148790053836785,
 '(1, 1)': 0.4695772823691831,
 '(1, 2)': 3.241481428134574e-05,
 '(2, 0)': 0.0030096031488172107,
 '(2, 1)': 0.0022302669196215767,
 '(2, 2)': 1.7852604795947343e-07}

```

```
plot_histogram(res_pops_opt, title='Simulation of Qiskit circuit with Optimized Gates')
```



## 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 managed 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 is 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)
        pops, _ = 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 wish 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  $\phi_{xy}$  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 specify a virtual AWG within  $C^3$  and transmit pixilated pulse shapes directly to the physical AWG.

## 7.5 Algorithms

As an optimization algorithm, 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  $\{|1\rangle, |2\rangle\}$ , 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/tmpicnbliz/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	2.074359374665050e-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 cyclor
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'] = cyclor(
    'linestyle', ["-", "--"])
)
rcParams['text.usetex'] = True
rcParams['font.size'] = 16
rcParams['font.family'] = 'serif'

logfile = opt.logdir + "calibration.log"
with open(logfile, "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)

```

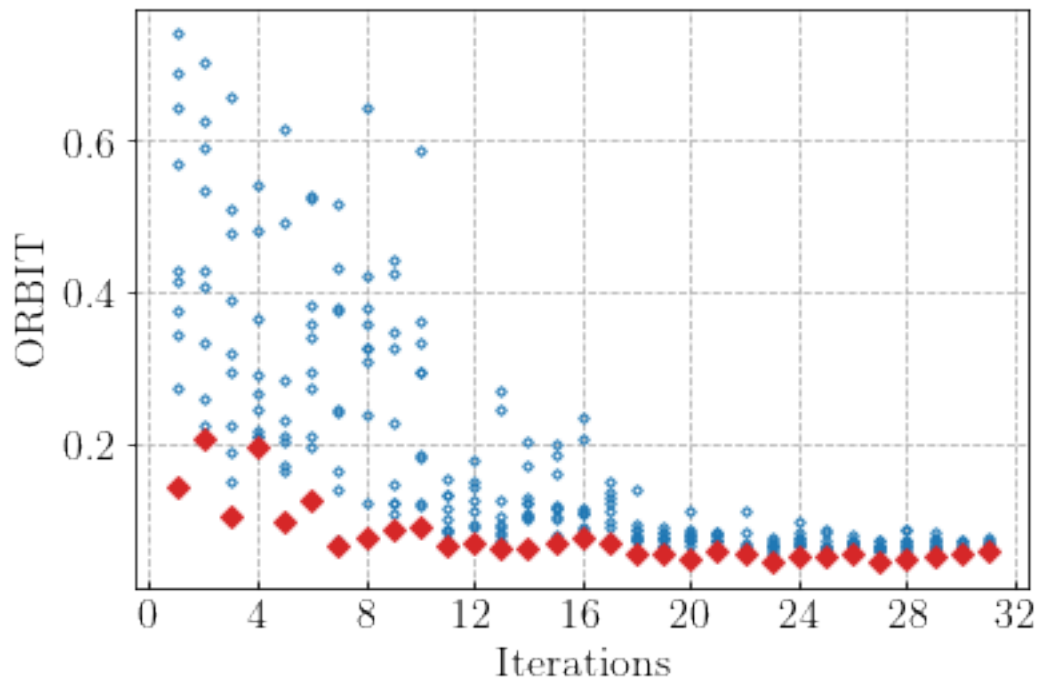
        if 'goal' in point.keys():
            goal_function[batch].append(point['goal'])

# 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'
)

```



## 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], [0.01230463327369004]),
 ([0.096], [0.009315793041926168]),
 ([0.368], [0.015250442616527561]),
 ([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
avg_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_l=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"
```

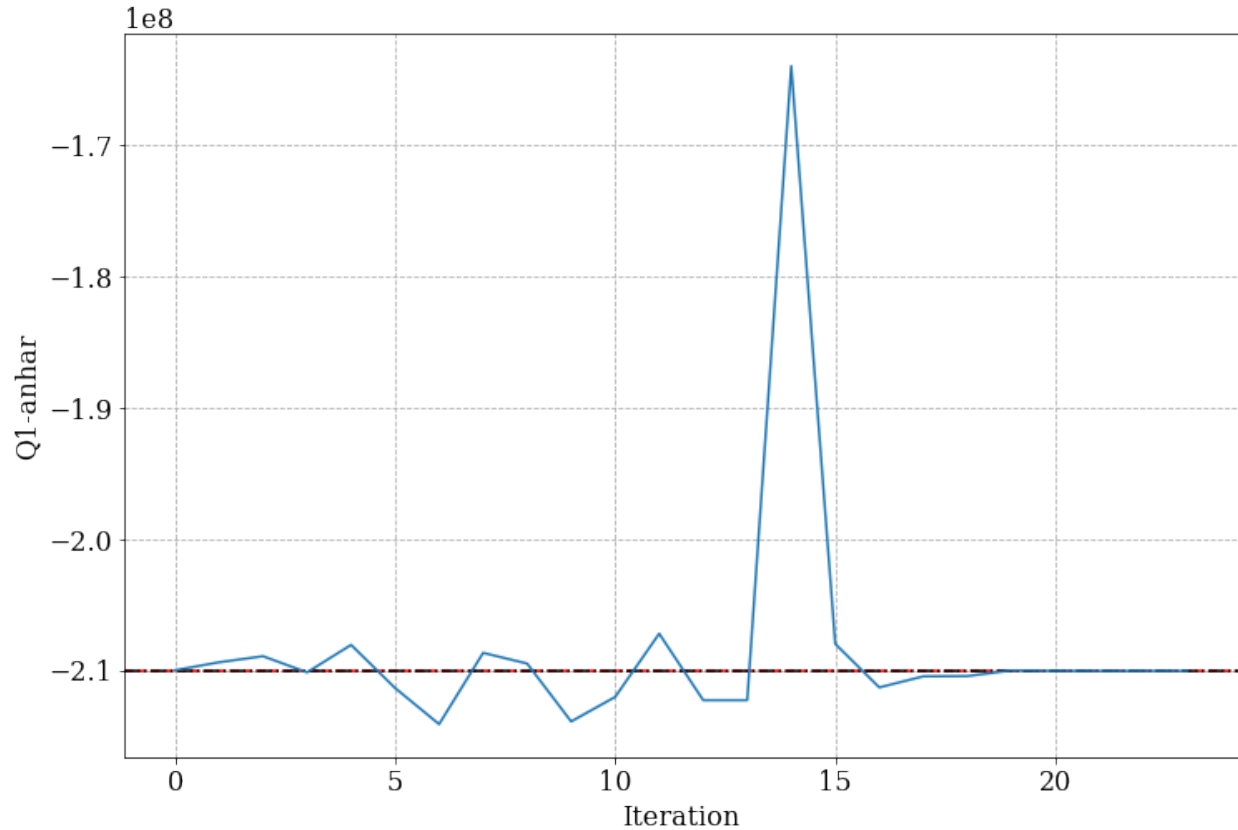
In the plots below, the blue line shows the progress of the parameter optimization while the black and the red lines indicate the converged and true value respectively

## 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

```

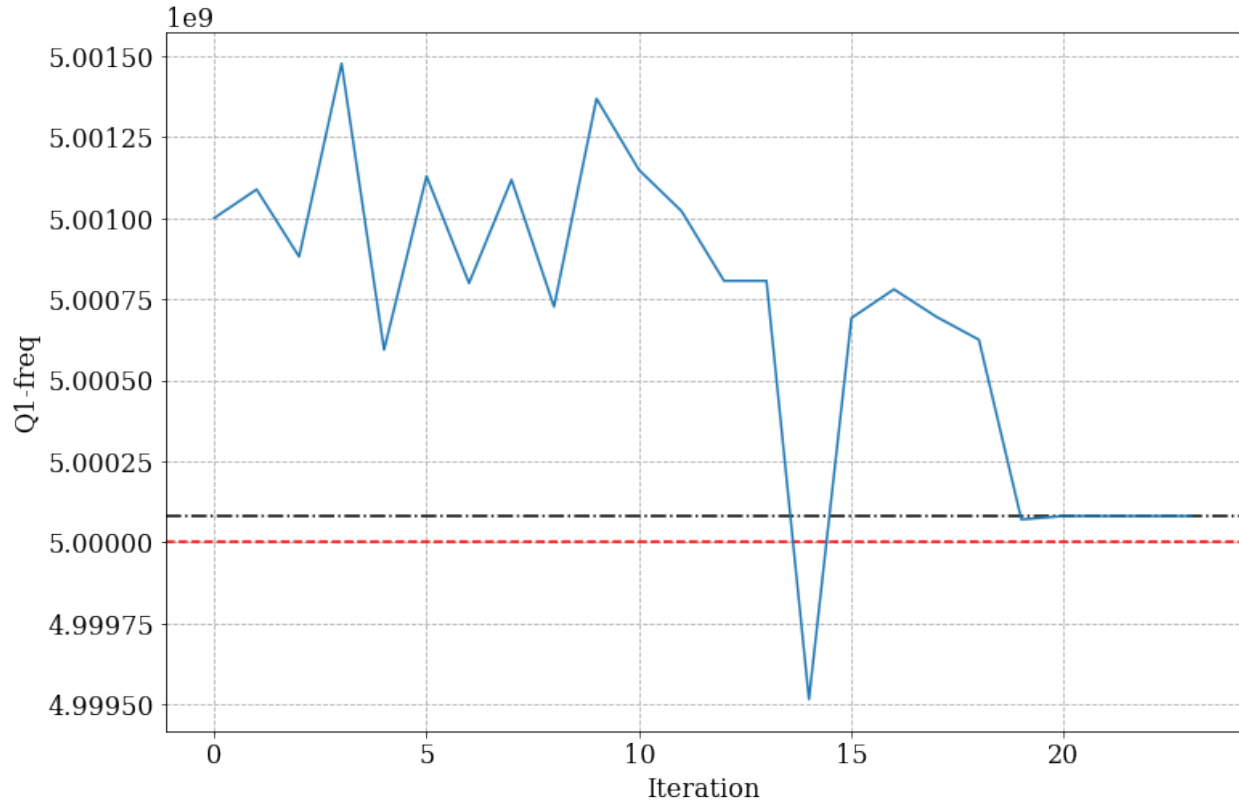
plot_item = "Q1-freq"
true_value = 5e9

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

```

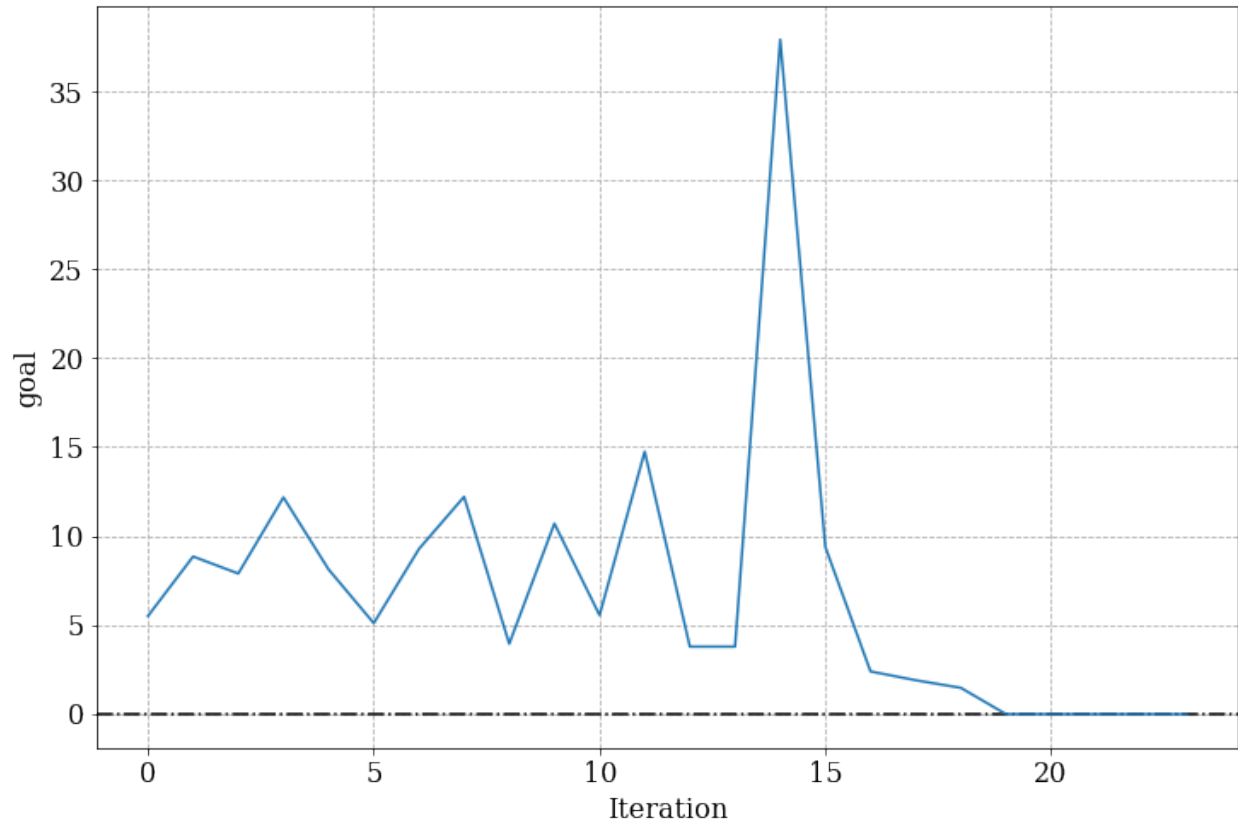
plot_item = "goal"

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>]
```



## 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]
```

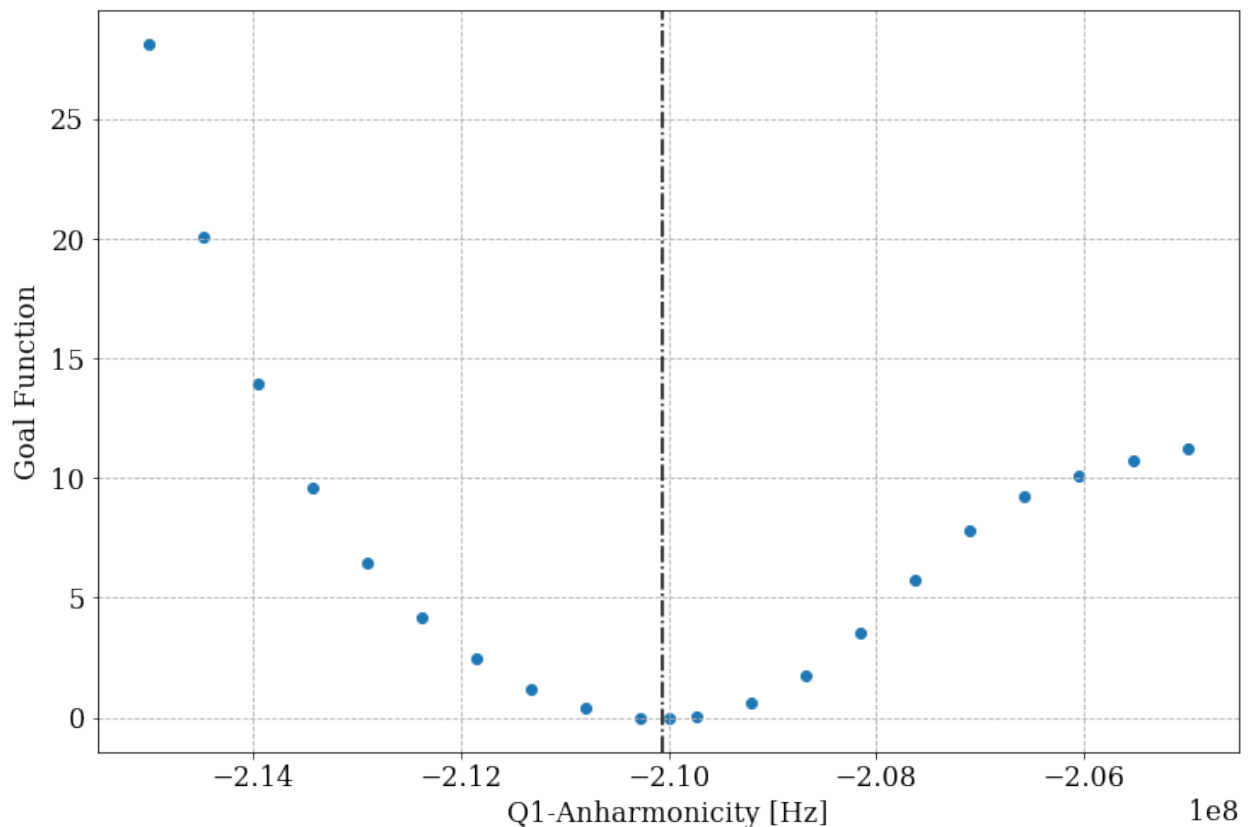
```
logfile = os.path.join(LOGDIR, "sensitivity.log")
with open(logfile, "r") as f:
    log = f.readlines()
```

```
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]
```

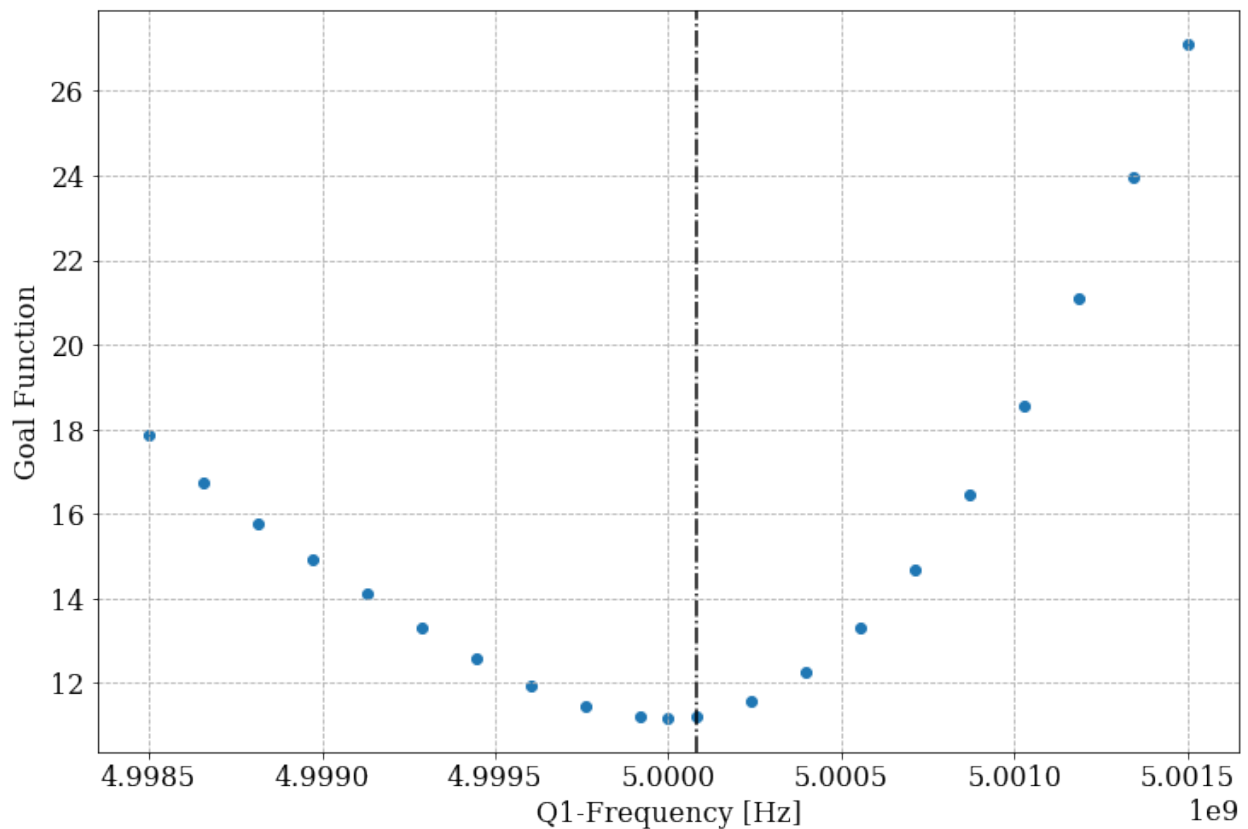
```
logfile = os.path.join(LOGDIR, "sensitivity.log")
with open(logfile, "r") as f:
    log = f.readlines()
```

```
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>
```





## 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





## 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 matplotlib
```

```
from pprint import pprint
import numpy as np
from c3.qiskit import C3Provider
from c3.qiskit.c3_gates import RX90pGate
from qiskit import transpile, execute, QuantumCircuit, Aer
from qiskit.tools.visualization import plot_histogram
```

### 11.1 Define a basic Quantum circuit

```
qc = QuantumCircuit(3, 3)
qc.append(RX90pGate(), [0])
qc.append(RX90pGate(), [1])
```

```
<qiskit.circuit.instructionset.InstructionSet at 0x17b189980>
```

```
qc.draw()
```

### 11.2 Get the C3 Provider and Backend

```
c3_provider = C3Provider()
c3_backend = c3_provider.get_backend("c3_qasm_physics_simulator")
```

```
config = c3_backend.configuration()

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

```
Name: c3_qasm_physics_simulator
Version: 0.1
Max Qubits: 10
OpenPulse Support: False
Basis Gates: ['cx', 'rx']
```

## 11.3 Run a physical device simulation using C3

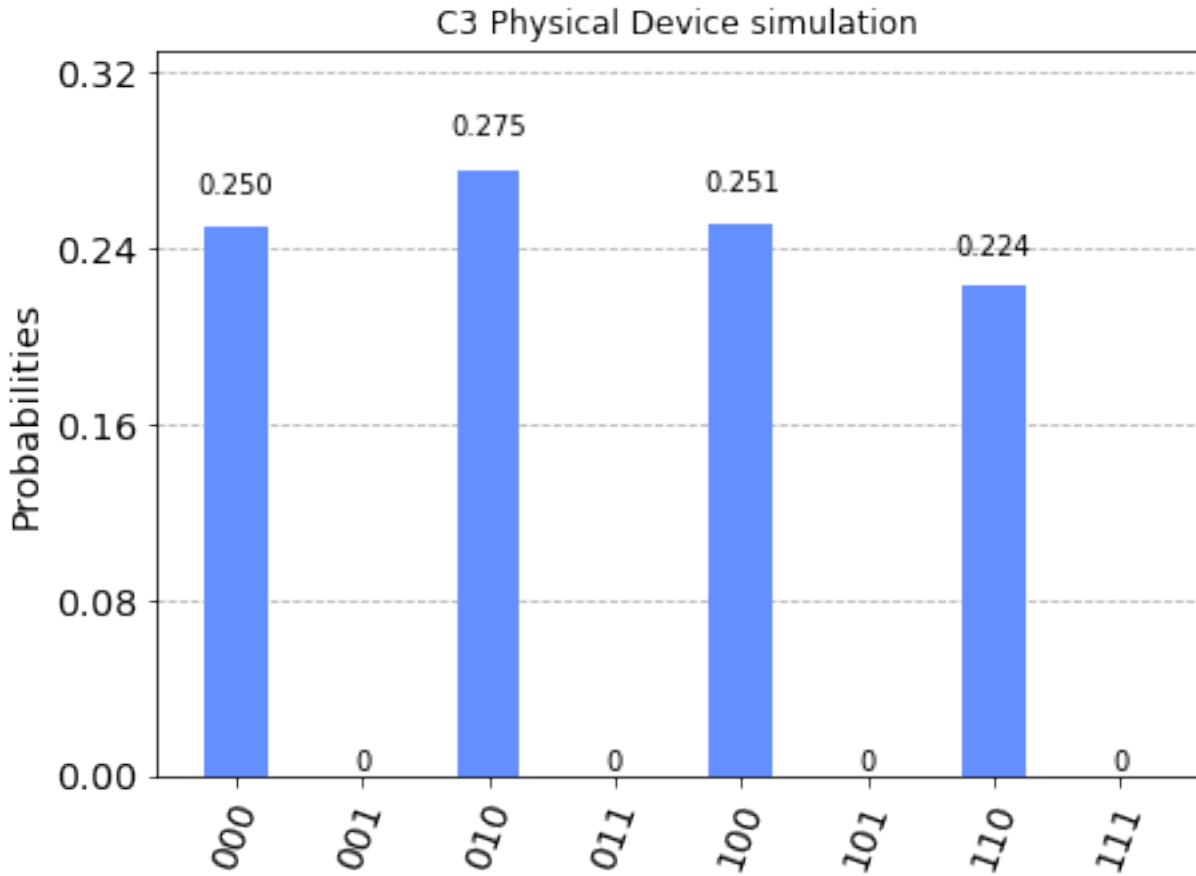
```
c3_backend.set_device_config("qiskit.cfg")
c3_job = c3_backend.run(qc)
result = c3_job.result()
```

```
No measurements in circuit "circuit-0", classical register will remain all zeros.
2022-01-01 03:30:00.931206: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.
↳cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
2022-01-01 03:30:00.933640: W tensorflow/core/platform/profile_utils/cpu_utils.cc:128]
↳Failed to get CPU frequency: 0 Hz
```

```
res_counts = result.get_counts()
pprint(res_counts)
```

```
{'000': 0.2501927838288728,
 '001': 1.6223933410984962e-29,
 '010': 0.27496041223138323,
 '011': 1.1175740719685343e-31,
 '100': 0.25116573990906244,
 '101': 2.4732633272223437e-33,
 '110': 0.22368106403066923,
 '111': 6.525386280486658e-35}
```

```
plot_histogram(res_counts, title='C3 Physical Device simulation')
```



As we can see above, the c3 simulator correctly calculates the populations while accounting for non-optimal pulses and device limitations.

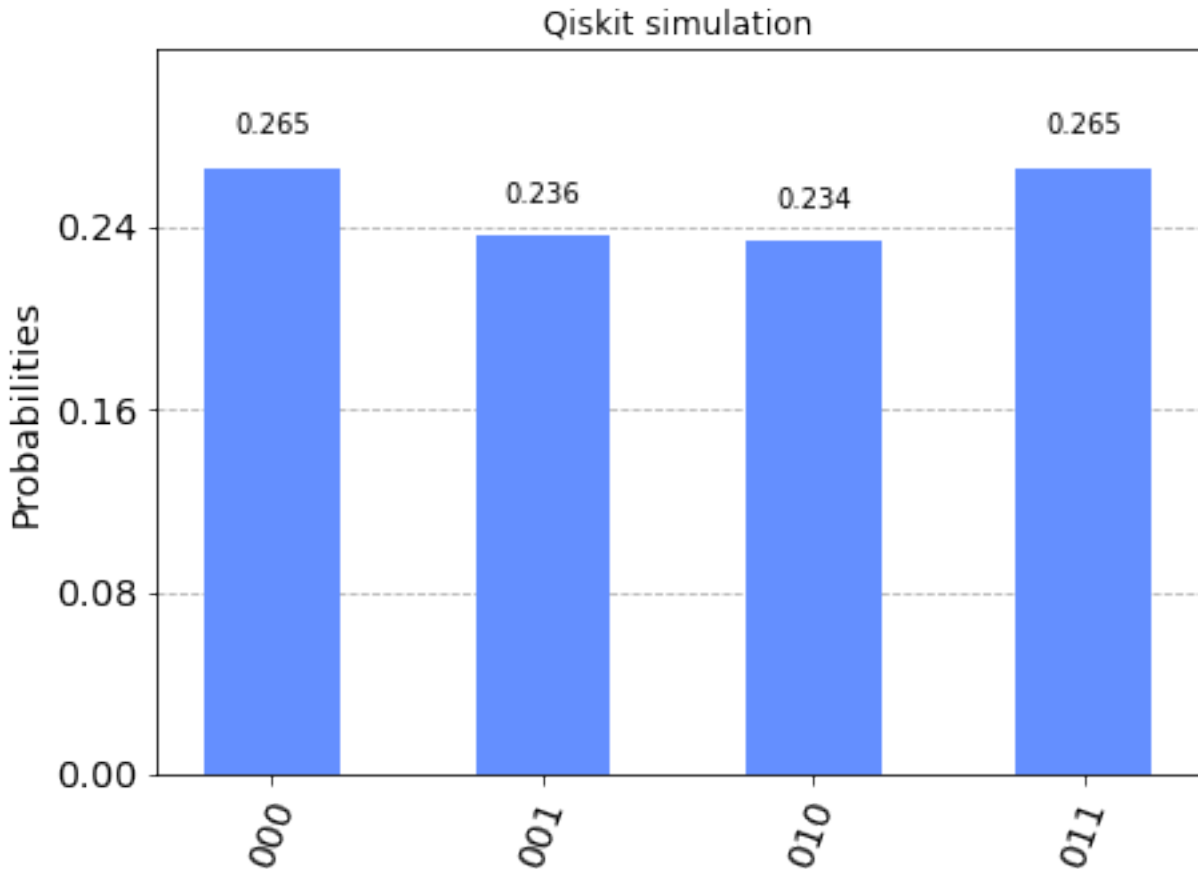
## 11.4 Run Simulation and verify results on Qiskit simulator

Qiskit uses little-endian bit ordering while most Quantum Computing literature uses big-endian. This is reflected in the reversed ordering of qubit labels here.

Ref: [Basis Vector Ordering in Qiskit](#)

```
qiskit_simulator = Aer.get_backend('qasm_simulator')
qc.measure([0, 1, 2], [0, 1, 2])
qiskit_result = execute(qc, qiskit_simulator, shots=1000).result()
counts = qiskit_result.get_counts(qc)
plot_histogram(counts, title='Qiskit simulation')
```

```
/opt/homebrew/Caskroom/miniforge/base/envs/c3-dev/lib/python3.8/site-packages/numpy/
↳ linalg/linalg.py:2159: RuntimeWarning: divide by zero encountered in det
  r = _umath_linalg.det(a, signature=signature)
/opt/homebrew/Caskroom/miniforge/base/envs/c3-dev/lib/python3.8/site-packages/numpy/
↳ linalg/linalg.py:2159: RuntimeWarning: invalid value encountered in det
  r = _umath_linalg.det(a, signature=signature)
```



## 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

$\text{scale} (\text{value} + 1) / 2 + \text{offset}$

#### 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/latest/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*, *psi\_init*: Optional[*tensorflow.python.framework.ops.Tensor*] = None)

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.
- **psi\_init** (*tf.Tensor*) – A tensor containing the initial statevector

**Returns** A list of populations

**Return type** list

**evaluate\_qasm**(*sequences*, *psi\_init*: Optional[*tensorflow.python.framework.ops.Tensor*] = None)

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.
- **psi\_init** (*tf.Tensor*) – A tensor containing the initial statevector

**Returns** A list of populations

**Return type** list

**expect\_oper**(*state*, *linbladlan*, *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 density 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\_indices**(*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 function 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*) → NoneSet 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 function 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*) → NoneWrite 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 optimization 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 component 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 bandwidth.

**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 bandwidth.

**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 explicitly

**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 quantum simulators, C<sup>3</sup> 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 interchangeably. 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](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](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](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](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\_opsers: 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  $w_{01} - w_{12}$
- **t1** (*Quantity*) –  $t_1$ , the time decay of the qubit due to dissipation
- **t2star** (*Quantity*) –  $t_{2star}$ , 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 known 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  $w_{01} - w_{12}$  :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 tunable 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 rise/fall, modelled by error functions.

**Parameters** *params* (*dict*) –

**t\_up** [float] Center of the ramp up.

**t\_down** [float] Center of the ramp down.

**rise/fall** [float] Length of the ramps.

`c3.libraries.envelopes.flattop_cut`(*t, params*)  
Flattop gaussian with width of length rise/fall, 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)`

symmetric 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.

**rise\_fall** [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_loglikh_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, *lindblad*=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 indices.

**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 misclassification 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*) – Indices 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** Indexes 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 invocation 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 *indices* 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 tensorflow's 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 integer 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 it has a gradient.

`c3.utils.tf_utils.tf_abs_squared(x)`  
Rewritten so that it 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 dimensions.

`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  $n \times m$

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 multiplication

`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 occurred

**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 occurred

**class** `c3.qiskit.c3_backend.C3QasmSimulator`(*configuration, provider=None, \*\*fields*)  
 Bases: `qiskit.providers.backend.BackendV1, abc.ABC`

An Abstract 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

**generate\_shot\_readout()**

Generate shot style readout from population

**Returns** List of shots for each output state

**Return type** List[int]

**get\_labels()**(*format: str = 'qiskit'*) → List[str]

Return state labels for the system

**Parameters** **format** (*str, optional*) – How to format the state labels, by default “qiskit”

**Returns**

A list of state labels in hex if qiskit format and decimal if c3 format

```
labels = ['0x1', ...]
labels = ['(0, 0)', '(0, 1)', '(0, 2)', ...]
```

**Return type** List[str]

**Raises** **C3QiskitError** – When an supported format is passed

**locate\_measurements()**(*instructions\_list: List[Dict]*) → List[int]

Locate the indices of measurement operations in circuit

**Parameters** **instructions\_list** (*List [Dict]*) – Instructions List in Qasm style

**Returns** The indices where measurement operations occur

**Return type** List[int]

**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]

**sanitize\_instructions** (*instructions: qiskit.circuit.instruction.Instruction*) → Tuple[List[Any], List[Any]]  
Convert from qiskit instruction object and Sanitize instructions by removing unsupported operations

**Parameters instructions** (*Instruction*) – qasm as Qiskit Instruction object

**Returns**

Sanitized instruction list

Qasm style list of instruction represented as dicts

**Return type** Tuple[List[Any], List[Any]]

**Raises UserWarning** – Warns user about unsupported operations in circuit

**set\_c3\_experiment** (*exp: c3.experiment.Experiment*) → None  
Set user-provided c3 experiment object for backend

**Parameters exp** (*Experiment*) – C3 experiment object

**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 Gates module

Library for interoperability of c3 gates with qiskit

**class** `c3.qiskit.c3_gates.CR90Gate(label: Optional[str] = None)`  
 Bases: `qiskit.extensions.unitary.UnitaryGate`  
 Cross Resonance 90 degree gate

**Warning:** This is not equivalent to the RZX( $\pi/2$ ) gate in qiskit

**class** `c3.qiskit.c3_gates.CRGate(label: Optional[str] = None)`  
 Bases: `qiskit.extensions.unitary.UnitaryGate`  
 Cross Resonance Gate

**Warning:** This is not equivalent to the RZX( $\pi/2$ ) gate in qiskit

**class** `c3.qiskit.c3_gates.CRXpGate`  
 Bases: `qiskit.circuit.library.standard_gates.rx.CRXGate`

**class** `c3.qiskit.c3_gates.RX90mGate`(*label: Optional[str] = None*)  
Bases: `qiskit.circuit.library.standard_gates.rx.RXGate`

90 degree rotation around X axis in the negative direction

**class** `c3.qiskit.c3_gates.RX90pGate`(*label: Optional[str] = None*)  
Bases: `qiskit.circuit.library.standard_gates.rx.RXGate`

90 degree rotation around X axis in the positive direction

**class** `c3.qiskit.c3_gates.RXpGate`(*label: Optional[str] = None*)  
Bases: `qiskit.circuit.library.standard_gates.rx.RXGate`

180 degree rotation around X axis

**class** `c3.qiskit.c3_gates.RY90mGate`(*label: Optional[str] = None*)  
Bases: `qiskit.circuit.library.standard_gates.ry.RYGate`

90 degree rotation around Y axis in the negative direction

**class** `c3.qiskit.c3_gates.RY90pGate`(*label: Optional[str] = None*)  
Bases: `qiskit.circuit.library.standard_gates.ry.RYGate`

90 degree rotation around Y axis in the positive direction

**class** `c3.qiskit.c3_gates.RYpGate`(*label: Optional[str] = None*)  
Bases: `qiskit.circuit.library.standard_gates.ry.RYGate`

180 degree rotation around Y axis

**class** `c3.qiskit.c3_gates.RZ90mGate`(*label: Optional[str] = None*)  
Bases: `qiskit.circuit.library.standard_gates.rz.RZGate`

90 degree rotation around Z axis in the negative direction

**class** `c3.qiskit.c3_gates.RZ90pGate`(*label: Optional[str] = None*)  
Bases: `qiskit.circuit.library.standard_gates.rz.RZGate`

90 degree rotation around Z axis in the positive direction

**class** `c3.qiskit.c3_gates.RZpGate`(*label: Optional[str] = None*)  
Bases: `qiskit.circuit.library.standard_gates.rz.RZGate`

180 degree rotation around Z axis

### 12.7.6.6 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](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

**12.7.6.7 Module contents**



## INDICES AND TABLES

- [genindex](#)
- [modindex](#)
- [search](#)



## A

adaptive\_scan() (in module *c3.libraries.algorithms*),  
 103  
 add() (*c3.c3objs.Quantity* method), 87  
 add\_component() (*c3.signal.gates.Instruction* method),  
 130  
 Additive\_Noise (class in *c3.generator.devices*), 96  
 algo\_reg\_deco() (in module *c3.libraries.algorithms*),  
 103  
 all() (in module *c3.libraries.sampling*), 122  
 as\_openqasm() (*c3.signal.gates.Instruction* method),  
 131  
 asdict() (*c3.c3objs.C3obj* method), 87  
 asdict() (*c3.c3objs.Quantity* method), 87  
 asdict() (*c3.experiment.Experiment* method), 88  
 asdict() (*c3.generator.devices.AWG* method), 95  
 asdict() (*c3.generator.devices.Device* method), 97  
 asdict() (*c3.generator.generator.Generator* method),  
 102  
 asdict() (*c3.libraries.chip.LineComponent* method),  
 109  
 asdict() (*c3.libraries.chip.PhysicalComponent*  
 method), 109  
 asdict() (*c3.model.Model* method), 91  
 asdict() (*c3.parametermap.ParameterMap* method), 93  
 asdict() (*c3.signal.gates.Instruction* method), 131  
 asdict() (*c3.signal.pulse.Envelope* method), 131  
 ask\_yn() (in module *c3.utils.utils*), 138  
 auto\_adjust\_t\_end() (*c3.signal.gates.Instruction*  
 method), 131  
 average\_infid() (in module *c3.libraries.fidelities*), 116  
 average\_infid\_seq() (in module  
*c3.libraries.fidelities*), 117  
 average\_infid\_set() (in module  
*c3.libraries.fidelities*), 117  
 AWG (class in *c3.generator.devices*), 95

## B

backends() (*c3.qiskit.c3\_provider.C3Provider* method),  
 142  
 BaseLogger (class in *c3.optimizers.optimizer*), 127  
 basis() (in module *c3.utils.qt\_utils*), 132

BestPointLogger (class in *c3.optimizers.optimizer*),  
 127  
 blowup\_excitations() (*c3.model.Model* method), 91

## C

c3  
   module, 95  
 c3.c3objs  
   module, 87  
 c3.experiment  
   module, 88  
 c3.generator  
   module, 103  
   c3.generator.devices  
     module, 95  
   c3.generator.generator  
     module, 102  
 c3.libraries  
   module, 124  
   c3.libraries.algorithms  
     module, 103  
   c3.libraries.chip  
     module, 107  
   c3.libraries.constants  
     module, 113  
   c3.libraries.envelopes  
     module, 113  
   c3.libraries.estimators  
     module, 115  
   c3.libraries.fidelities  
     module, 116  
   c3.libraries.hamiltonians  
     module, 120  
   c3.libraries.sampling  
     module, 122  
   c3.libraries.tasks  
     module, 123  
 c3.main  
   module, 95  
 c3.model  
   module, 91  
 c3.optimizers

- module, 130
  - c3.optimizers.calibration
    - module, 125
  - c3.optimizers.modellearning
    - module, 126
  - c3.optimizers.optimalcontrol
    - module, 124
  - c3.optimizers.optimizer
    - module, 127
  - c3.optimizers.sensitivity
    - module, 129
  - c3.parametermap
    - module, 93
  - c3.qiskit
    - module, 146
  - c3.qiskit.c3\_backend
    - module, 139
  - c3.qiskit.c3\_backend\_utils
    - module, 144
  - c3.qiskit.c3\_exceptions
    - module, 143
  - c3.qiskit.c3\_gates
    - module, 143
  - c3.qiskit.c3\_job
    - module, 143
  - c3.qiskit.c3\_provider
    - module, 142
  - c3.signal
    - module, 132
  - c3.signal.gates
    - module, 130
  - c3.signal.pulse
    - module, 131
  - c3.utils
    - module, 139
  - c3.utils.log\_reader
    - module, 138
  - c3.utils.qt\_utils
    - module, 132
  - c3.utils.tf\_utils
    - module, 135
  - c3.utils.utils
    - module, 138
  - C3Job (class in c3.qiskit.c3\_job), 143
  - C3obj (class in c3.c3objs), 87
  - C3Provider (class in c3.qiskit.c3\_provider), 142
  - C3QasmPerfectSimulator (class in c3.qiskit.c3\_backend), 139
  - C3QasmPhysicsSimulator (class in c3.qiskit.c3\_backend), 139
  - C3QasmSimulator (class in c3.qiskit.c3\_backend), 140
  - C3QiskitError, 143
  - calc\_slice\_num() (c3.generator.devices.Device method), 97
  - Calibration (class in c3.optimizers.calibration), 125
  - Carrier (class in c3.signal.pulse), 131
  - check\_limits() (c3.parametermap.ParameterMap method), 93
  - cma\_pre\_lbfgs() (in module c3.libraries.algorithms), 103
  - cmaes() (in module c3.libraries.algorithms), 103
  - comp\_reg\_deco() (in module c3.signal.pulse), 132
  - comps (c3.signal.gates.Instruction attribute), 130
  - compute\_propagators() (c3.experiment.Experiment method), 89
  - compute\_states() (c3.experiment.Experiment method), 89
  - confirm() (c3.optimizers.modellearning.ModelLearning method), 127
  - confuse() (c3.libraries.tasks.ConfusionMatrix method), 123
  - ConfusionMatrix (class in c3.libraries.tasks), 123
  - convolve() (c3.generator.devices.HighpassFilter method), 99
  - convolve() (c3.generator.devices.Response method), 101
  - cosine() (in module c3.libraries.envelopes), 113
  - cosine\_flattop() (in module c3.libraries.envelopes), 113
  - cosm() (c3.libraries.chip.CShuntFluxQubitCos method), 108
  - Coupling (class in c3.libraries.chip), 108
  - CR90Gate (class in c3.qiskit.c3\_gates), 143
  - create\_IQC() (c3.generator.devices.AWG method), 95
  - create\_IQ\_pwc() (c3.generator.devices.AWG method), 96
  - create\_ts() (c3.generator.devices.Device method), 97
  - CRGate (class in c3.qiskit.c3\_gates), 143
  - Crosstalk (class in c3.generator.devices), 96
  - CRXpGate (class in c3.qiskit.c3\_gates), 143
  - CShuntFluxQubit (class in c3.libraries.chip), 107
  - CShuntFluxQubitCos (class in c3.libraries.chip), 107
  - cut\_excitations() (c3.model.Model method), 91
- ## D
- DC\_Noise (class in c3.generator.devices), 97
  - DC\_Offset (class in c3.generator.devices), 97
  - DEFAULT\_OPTIONS (c3.qiskit.c3\_backend.C3QasmPerfectSimulator attribute), 139
  - DEFAULT\_OPTIONS (c3.qiskit.c3\_backend.C3QasmPhysicsSimulator attribute), 140
  - delta\_pulse() (in module c3.libraries.envelopes), 113
  - deprecated() (in module c3.utils.utils), 138
  - dev\_reg\_deco() (in module c3.generator.devices), 102
  - dev\_reg\_deco() (in module c3.libraries.chip), 113
  - Device (class in c3.generator.devices), 97
  - DigitalToAnalog (class in c3.generator.devices), 98

- disable\_flip\_labels() (*c3.qiskit.c3\_backend.C3QasmSimulator method*), 140  
 drag\_der() (*in module c3.libraries.envelopes*), 113  
 drag\_sigma() (*in module c3.libraries.envelopes*), 113  
 Drive (*class in c3.libraries.chip*), 108  
 duffing() (*in module c3.libraries.hamiltonians*), 120  
 dv\_g\_LL\_prime() (*in module c3.libraries.estimators*), 115
- ## E
- enable\_drag() (*c3.generator.devices.AWG method*), 96  
 enable\_drag\_2() (*c3.generator.devices.AWG method*), 96  
 enable\_pwc() (*c3.generator.devices.AWG method*), 96  
 enable\_qasm() (*c3.experiment.Experiment method*), 89  
 end\_log() (*c3.optimizers.optimizer.Optimizer method*), 128  
 energies\_from\_frequencies() (*c3.libraries.chip.TransmonExpanded method*), 112  
 eng\_num() (*in module c3.utils.utils*), 138  
 env\_reg\_deco() (*in module c3.libraries.envelopes*), 113  
 Envelope (*class in c3.signal.pulse*), 131  
 EnvelopeNetZero (*class in c3.signal.pulse*), 131  
 epc\_analytical() (*in module c3.libraries.fidelities*), 117  
 estimator\_reg\_deco() (*in module c3.libraries.estimators*), 115  
 evaluate\_legacy() (*c3.experiment.Experiment method*), 89  
 evaluate\_qasm() (*c3.experiment.Experiment method*), 89  
 even() (*in module c3.libraries.sampling*), 122  
 even\_fid() (*in module c3.libraries.sampling*), 122  
 Example (*c3.signal.gates.Instruction attribute*), 130  
 expand\_dims() (*in module c3.utils.qt\_utils*), 132  
 expect\_oper() (*c3.experiment.Experiment method*), 89  
 Experiment (*class in c3.experiment*), 88  
 ExponentialIIR (*class in c3.generator.devices*), 98
- ## F
- fct\_to\_min() (*c3.optimizers.optimizer.Optimizer method*), 128  
 fct\_to\_min\_autograd() (*c3.optimizers.optimizer.Optimizer method*), 128  
 fid\_reg\_deco() (*in module c3.libraries.fidelities*), 117  
 Filter (*class in c3.generator.devices*), 98  
 flatten() (*in module c3.utils.utils*), 138  
 flattop() (*in module c3.libraries.envelopes*), 113  
 flattop\_cut() (*in module c3.libraries.envelopes*), 113  
 flattop\_cut\_center() (*in module c3.libraries.envelopes*), 114  
 flattop\_risefall() (*in module c3.libraries.envelopes*), 114  
 flattop\_risefall\_1ns() (*in module c3.libraries.envelopes*), 114  
 flattop\_variant() (*in module c3.libraries.envelopes*), 114  
 flip\_labels() (*in module c3.qiskit.c3\_backend\_utils*), 144  
 Fluxonium (*class in c3.libraries.chip*), 109  
 FluxTuning (*class in c3.generator.devices*), 98  
 FluxTuningLinear (*class in c3.generator.devices*), 99  
 fourier\_cos() (*in module c3.libraries.envelopes*), 114  
 fourier\_sin() (*in module c3.libraries.envelopes*), 114  
 frequency() (*c3.generator.devices.FluxTuningLinear method*), 99  
 from\_dict() (*c3.experiment.Experiment method*), 89  
 from\_dict() (*c3.signal.gates.Instruction method*), 131  
 from\_end() (*in module c3.libraries.sampling*), 122  
 from\_start() (*in module c3.libraries.sampling*), 122  
 fromdict() (*c3.generator.generator.Generator method*), 102  
 fromdict() (*c3.model.Model method*), 91  
 fromdict() (*c3.parametermap.ParameterMap method*), 93
- ## G
- g\_LL\_prime() (*in module c3.libraries.estimators*), 115  
 g\_LL\_prime\_combined() (*in module c3.libraries.estimators*), 115  
 gaussian\_der() (*in module c3.libraries.envelopes*), 114  
 gaussian\_der\_nonorm() (*in module c3.libraries.envelopes*), 114  
 gaussian\_nonorm() (*in module c3.libraries.envelopes*), 114  
 gaussian\_sigma() (*in module c3.libraries.envelopes*), 114  
 gcmaes() (*in module c3.libraries.algorithms*), 104  
 generate\_shot\_readout() (*c3.qiskit.c3\_backend.C3QasmSimulator method*), 141  
 generate\_signals() (*c3.generator.generator.Generator method*), 102  
 Generator (*class in c3.generator.generator*), 102  
 get\_anhar() (*c3.libraries.chip.Transmon method*), 112  
 get\_anharmonicity() (*c3.libraries.chip.CShuntFluxQubit method*), 107  
 get\_average\_amp() (*c3.generator.devices.AWG method*), 96  
 get\_awg\_signal() (*c3.signal.gates.Instruction method*), 131  
 get\_basis\_matrices() (*in module c3.utils.qt\_utils*), 132

- `get_dephasing_channel()` (*c3.model.Model* method), 92  
`get_factor()` (*c3.generator.devices.FluxTuning* method), 99  
`get_factor()` (*c3.libraries.chip.Transmon* method), 112  
`get_Frame_Rotation()` (*c3.model.Model* method), 91  
`get_freq()` (*c3.generator.devices.FluxTuning* method), 99  
`get_freq()` (*c3.libraries.chip.CShuntFluxQubit* method), 107  
`get_freq()` (*c3.libraries.chip.CShuntFluxQubitCos* method), 108  
`get_freq()` (*c3.libraries.chip.Transmon* method), 112  
`get_frequency()` (*c3.libraries.chip.CShuntFluxQubit* method), 107  
`get_full_gate_length()` (*c3.signal.gates.Instruction* method), 131  
`get_full_params()` (*c3.parametermap.ParameterMap* method), 93  
`get_ground_state()` (*c3.model.Model* method), 92  
`get_Hamiltonian()` (*c3.libraries.chip.Coupling* method), 108  
`get_Hamiltonian()` (*c3.libraries.chip.CShuntFluxQubit* method), 107  
`get_Hamiltonian()` (*c3.libraries.chip.CShuntFluxQubitCos* method), 108  
`get_Hamiltonian()` (*c3.libraries.chip.Drive* method), 108  
`get_Hamiltonian()` (*c3.libraries.chip.PhysicalComponent* method), 109  
`get_Hamiltonian()` (*c3.libraries.chip.Qubit* method), 110  
`get_Hamiltonian()` (*c3.libraries.chip.Resonator* method), 110  
`get_Hamiltonian()` (*c3.libraries.chip.SNAIL* method), 111  
`get_Hamiltonian()` (*c3.libraries.chip.Transmon* method), 111  
`get_Hamiltonian()` (*c3.libraries.chip.TransmonExpanded* method), 112  
`get_Hamiltonian()` (*c3.model.Model* method), 91  
`get_Hamiltonians()` (*c3.model.Model* method), 92  
`get_Hs()` (*c3.libraries.chip.TransmonExpanded* method), 112  
`get_I()` (*c3.generator.devices.AWG* method), 96  
`get_ideal_gate()` (*c3.signal.gates.Instruction* method), 131  
`get_init_ground_state()` (in module *c3.qiskit.c3\_backend\_utils*), 145  
`get_key()` (*c3.signal.gates.Instruction* method), 131  
`get_key_from_scaled_index()` (*c3.parametermap.ParameterMap* method), 93  
`get_labels()` (*c3.qiskit.c3\_backend.C3QasmSimulator* method), 141  
`get_limits()` (*c3.c3objs.Quantity* method), 87  
`get_Lindbladian()` (*c3.libraries.chip.Qubit* method), 110  
`get_Lindbladian()` (*c3.libraries.chip.Resonator* method), 110  
`get_Lindbladian()` (*c3.libraries.chip.Transmon* method), 112  
`get_Lindbladians()` (*c3.model.Model* method), 92  
`get_minimum_phi_var()` (*c3.libraries.chip.CShuntFluxQubit* method), 107  
`get_n_variable()` (*c3.libraries.chip.CShuntFluxQubitCos* method), 108  
`get_noise()` (*c3.generator.devices.Additive\_Noise* method), 96  
`get_noise()` (*c3.generator.devices.DC\_Noise* method), 97  
`get_noise()` (*c3.generator.devices.Pink\_Noise* method), 100  
`get_not_opt_params()` (*c3.parametermap.ParameterMap* method), 93  
`get_opt_limits()` (*c3.parametermap.ParameterMap* method), 93  
`get_opt_map()` (*c3.parametermap.ParameterMap* method), 93  
`get_opt_units()` (*c3.parametermap.ParameterMap* method), 93  
`get_opt_value()` (*c3.c3objs.Quantity* method), 88  
`get_optimizable_parameters()` (*c3.signal.gates.Instruction* method), 131  
`get_parameter()` (*c3.parametermap.ParameterMap* method), 93  
`get_parameter_dict()` (*c3.parametermap.ParameterMap* method), 93  
`get_parameters()` (*c3.parametermap.ParameterMap* method), 94  
`get_parameters_scaled()` (*c3.parametermap.ParameterMap* method), 94  
`get_perfect_gates()` (*c3.experiment.Experiment* method), 89  
`get_phase_variable()` (*c3.libraries.chip.CShuntFluxQubitCos* method), 108  
`get_potential_function()` (*c3.libraries.chip.CShuntFluxQubit* method), 107  
`get_potential_function()` (*c3.libraries.chip.Fluxonium* method), 109  
`get_prefactors()` (*c3.libraries.chip.TransmonExpanded* method), 112

- get\_Q() (*c3.generator.devices.AWG method*), 96  
 get\_qubit\_freqs() (*c3.model.Model method*), 92  
 get\_sequence() (in module *c3.qiskit.c3\_backend\_utils*), 145  
 get\_shape\_values() (*c3.signal.pulse.Envelope method*), 131  
 get\_shape\_values() (*c3.signal.pulse.EnvelopeNetZero method*), 132  
 get\_sparse\_Hamiltonian() (*c3.model.Model method*), 92  
 get\_sparse\_Hamiltonians() (*c3.model.Model method*), 92  
 get\_state\_indeces() (*c3.model.Model method*), 92  
 get\_state\_index() (*c3.model.Model method*), 92  
 get\_tf\_log\_level() (in module *c3.utils.tf\_utils*), 135  
 get\_third\_order\_prefactor() (*c3.libraries.chip.CShuntFluxQubit method*), 107  
 get\_timings() (*c3.signal.gates.Instruction method*), 131  
 get\_transformed\_hamiltonians() (*c3.libraries.chip.PhysicalComponent method*), 109  
 get\_value() (*c3.c3objs.Quantity method*), 88  
 get\_VZ() (*c3.experiment.Experiment method*), 89  
 goal\_run() (*c3.optimizers.calibration.Calibration method*), 125  
 goal\_run() (*c3.optimizers.modellearning.ModelLearning method*), 127  
 goal\_run() (*c3.optimizers.optimalcontrol.OptimalControl method*), 125  
 goal\_run() (*c3.optimizers.optimizer.Optimizer method*), 128  
 goal\_run\_with\_grad() (*c3.optimizers.modellearning.ModelLearning method*), 127  
 goal\_run\_with\_grad() (*c3.optimizers.optimizer.Optimizer method*), 128  
 grid2D() (in module *c3.libraries.algorithms*), 104
- ## H
- hamiltonian\_reg\_deco() (in module *c3.libraries.hamiltonians*), 120  
 high\_std() (in module *c3.libraries.sampling*), 122  
 HighpassExponential (class in *c3.generator.devices*), 99  
 HighpassFilter (class in *c3.generator.devices*), 99  
 hilbert\_space\_kron() (in module *c3.utils.qt\_utils*), 132  
 hjson\_decode() (in module *c3.c3objs*), 88  
 hjson\_encode() (in module *c3.c3objs*), 88
- ## I
- Id\_like() (in module *c3.utils.tf\_utils*), 135  
 init\_exponentiated\_vars() (*c3.libraries.chip.CShuntFluxQubitCos method*), 108  
 init\_Hs() (*c3.libraries.chip.Coupling method*), 108  
 init\_Hs() (*c3.libraries.chip.CShuntFluxQubit method*), 107  
 init\_Hs() (*c3.libraries.chip.CShuntFluxQubitCos method*), 108  
 init\_Hs() (*c3.libraries.chip.Drive method*), 109  
 init\_Hs() (*c3.libraries.chip.Qubit method*), 110  
 init\_Hs() (*c3.libraries.chip.Resonator method*), 110  
 init\_Hs() (*c3.libraries.chip.SNAIL method*), 111  
 init\_Hs() (*c3.libraries.chip.Transmon method*), 112  
 init\_Hs() (*c3.libraries.chip.TransmonExpanded method*), 113  
 init\_Ls() (*c3.libraries.chip.Qubit method*), 110  
 init\_Ls() (*c3.libraries.chip.Resonator method*), 110  
 init\_Ls() (*c3.libraries.chip.Transmon method*), 112  
 initialise() (*c3.libraries.tasks.InitialiseGround method*), 123  
 InitialiseGround (class in *c3.libraries.tasks*), 123  
 insert\_mat\_kron() (in module *c3.utils.qt\_utils*), 133  
 Instruction (class in *c3.signal.gates*), 130  
 int\_XX() (in module *c3.libraries.hamiltonians*), 120  
 int\_YY() (in module *c3.libraries.hamiltonians*), 121  
 inverseC() (in module *c3.utils.qt\_utils*), 133
- ## J
- jsonify\_list() (in module *c3.c3objs*), 88
- ## K
- kron\_ids() (in module *c3.utils.qt\_utils*), 133
- ## L
- lbfgs() (in module *c3.libraries.algorithms*), 104  
 lbfgs\_grad\_free() (in module *c3.libraries.algorithms*), 104  
 leakage\_RB() (in module *c3.libraries.fidelities*), 117  
 learn\_model() (*c3.optimizers.modellearning.ModelLearning method*), 127  
 lindbladian\_average\_infid() (in module *c3.libraries.fidelities*), 117  
 lindbladian\_average\_infid\_set() (in module *c3.libraries.fidelities*), 118  
 lindbladian\_epc\_analytical() (in module *c3.libraries.fidelities*), 118  
 lindbladian\_population() (in module *c3.libraries.fidelities*), 118  
 lindbladian\_RB\_left() (in module *c3.libraries.fidelities*), 117

- lindbladian\_RB\_right() (in module *c3.libraries.fidelities*), 117  
 lindbladian\_unitary\_infid() (in module *c3.libraries.fidelities*), 118  
 lindbladian\_unitary\_infid\_set() (in module *c3.libraries.fidelities*), 118  
 LineComponent (class in *c3.libraries.chip*), 109  
 list\_parameters() (*c3.model.Model* method), 92  
 L0 (class in *c3.generator.devices*), 100  
 load\_best() (*c3.optimizers.optimizer.Optimizer* method), 128  
 load\_model\_parameters() (*c3.optimizers.optimalcontrol.OptimalControl* method), 125  
 load\_quick\_setup() (*c3.experiment.Experiment* method), 89  
 load\_values() (*c3.parametermap.ParameterMap* method), 94  
 locate\_measurements() (*c3.qiskit.c3\_backend.C3QasmSimulator* method), 141  
 log\_best\_unitary() (*c3.optimizers.optimizer.Optimizer* method), 128  
 log\_parameters() (*c3.optimizers.optimizer.BaseLogger* method), 127  
 log\_parameters() (*c3.optimizers.optimizer.Optimizer* method), 128  
 log\_parameters() (*c3.optimizers.optimizer.TensorBoardLogger* method), 129  
 log\_pickle() (*c3.optimizers.calibration.Calibration* method), 126  
 log\_setup() (*c3.optimizers.calibration.Calibration* method), 126  
 log\_setup() (*c3.optimizers.modellearning.ModelLearning* method), 127  
 log\_setup() (*c3.optimizers.optimalcontrol.OptimalControl* method), 125  
 log\_setup() (in module *c3.utils.utils*), 138  
 log\_shapes() (*c3.generator.devices.AWG* method), 96  
 LONoise (class in *c3.generator.devices*), 100  
 lookup\_gate() (*c3.experiment.Experiment* method), 90  
 lookup\_gradient() (*c3.optimizers.optimizer.Optimizer* method), 128
- ## M
- make\_gate\_str() (in module *c3.qiskit.c3\_backend\_utils*), 145  
 MAX\_QUBITS\_MEMORY (*c3.qiskit.c3\_backend.C3QasmPerfectSimulator* attribute), 139  
 MAX\_QUBITS\_MEMORY (*c3.qiskit.c3\_backend.C3QasmPhysicsSimulator* attribute), 140
- ## N
- mean\_dist() (in module *c3.libraries.estimators*), 115  
 mean\_exp\_stds\_dist() (in module *c3.libraries.estimators*), 115  
 mean\_sim\_stds\_dist() (in module *c3.libraries.estimators*), 115  
 MeasurementRescale (class in *c3.libraries.tasks*), 124  
 median\_dist() (in module *c3.libraries.estimators*), 115  
 Mixer (class in *c3.generator.devices*), 100  
 Model (class in *c3.model*), 91  
 ModelLearning (class in *c3.optimizers.modellearning*), 126  
 module  
   *c3*, 95  
   *c3.c3objs*, 87  
   *c3.experiment*, 88  
   *c3.generator*, 103  
   *c3.generator.devices*, 95  
   *c3.generator.generator*, 102  
   *c3.libraries*, 124  
   *c3.libraries.algorithms*, 103  
   *c3.libraries.chip*, 107  
   *c3.libraries.constants*, 113  
   *c3.libraries.envelopes*, 113  
   *c3.libraries.estimators*, 115  
   *c3.libraries.fidelities*, 116  
   *c3.libraries.hamiltonians*, 120  
   *c3.libraries.sampling*, 122  
   *c3.libraries.tasks*, 123  
   *c3.main*, 95  
   *c3.model*, 91  
   *c3.optimizers*, 130  
   *c3.optimizers.calibration*, 125  
   *c3.optimizers.modellearning*, 126  
   *c3.optimizers.optimalcontrol*, 124  
   *c3.optimizers.optimizer*, 127  
   *c3.optimizers.sensitivity*, 129  
   *c3.parametermap*, 93  
   *c3.qiskit*, 146  
   *c3.qiskit.c3\_backend*, 139  
   *c3.qiskit.c3\_backend\_utils*, 144  
   *c3.qiskit.c3\_exceptions*, 143  
   *c3.qiskit.c3\_gates*, 143  
   *c3.qiskit.c3\_job*, 143  
   *c3.qiskit.c3\_provider*, 142  
   *c3.signal*, 132  
   *c3.signal.gates*, 130  
   *c3.signal.pulse*, 131  
   *c3.utils*, 139  
   *c3.utils.log\_reader*, 138  
   *c3.utils.qt\_utils*, 132  
   *c3.utils.tf\_utils*, 135  
   *c3.utils.utils*, 138



- neg\_loglkh\_binom\_norm() (in module *c3.libraries.estimators*), 115  
 neg\_loglkh\_gauss() (in module *c3.libraries.estimators*), 116  
 neg\_loglkh\_gauss\_norm() (in module *c3.libraries.estimators*), 116  
 neg\_loglkh\_gauss\_norm\_sum() (in module *c3.libraries.estimators*), 116  
 neg\_loglkh\_multinom() (in module *c3.libraries.estimators*), 116  
 neg\_loglkh\_multinom\_norm() (in module *c3.libraries.estimators*), 116  
 no\_drive() (in module *c3.libraries.envelopes*), 115  
 np\_kron\_n() (in module *c3.utils.qt\_utils*), 133  
 num3str() (in module *c3.utils.utils*), 139  
 numpy() (*c3.c3objs.Quantity* method), 88
- ## O
- oneplusone() (in module *c3.libraries.algorithms*), 105  
 open\_system\_deco() (in module *c3.libraries.fidelities*), 119  
 OptimalControl (class in *c3.optimizers.optimalcontrol*), 124  
 optimize\_controls() (*c3.optimizers.calibration.Calibration* method), 126  
 optimize\_controls() (*c3.optimizers.optimalcontrol.OptimalControl* method), 125  
 Optimizer (class in *c3.optimizers.optimizer*), 127  
 orbit\_infid() (in module *c3.libraries.fidelities*), 119
- ## P
- pad\_matrix() (in module *c3.utils.qt\_utils*), 133  
 ParameterMap (class in *c3.parametermap*), 93  
 pauli\_basis() (in module *c3.utils.qt\_utils*), 133  
 perfect\_cliffords() (in module *c3.utils.qt\_utils*), 133  
 perfect\_parametric\_gate() (in module *c3.utils.qt\_utils*), 133  
 perfect\_single\_q\_parametric\_gate() (in module *c3.utils.qt\_utils*), 134  
 PhysicalComponent (class in *c3.libraries.chip*), 109  
 Pink\_Noise (class in *c3.generator.devices*), 100  
 population() (in module *c3.libraries.fidelities*), 119  
 populations() (*c3.experiment.Experiment* method), 90  
 populations() (in module *c3.libraries.fidelities*), 119  
 print\_parameters() (*c3.parametermap.ParameterMap* method), 94  
 process() (*c3.experiment.Experiment* method), 90  
 process() (*c3.generator.devices.Additive\_Noise* method), 96  
 process() (*c3.generator.devices.Crosstalk* method), 97  
 process() (*c3.generator.devices.DC\_Offset* method), 97  
 process() (*c3.generator.devices.DigitalToAnalog* method), 98  
 process() (*c3.generator.devices.Filter* method), 98  
 process() (*c3.generator.devices.FluxTuning* method), 99  
 process() (*c3.generator.devices.HighpassFilter* method), 100  
 process() (*c3.generator.devices.LO* method), 100  
 process() (*c3.generator.devices.LONoise* method), 100  
 process() (*c3.generator.devices.Mixer* method), 100  
 process() (*c3.generator.devices.Response* method), 101  
 process() (*c3.generator.devices.ResponseFFT* method), 101  
 process() (*c3.generator.devices.StepFuncFilter* method), 101  
 process() (*c3.generator.devices.VoltsToHertz* method), 102  
 projector() (in module *c3.utils.qt\_utils*), 134  
 pwc() (in module *c3.libraries.envelopes*), 115  
 pwc\_shape() (in module *c3.libraries.envelopes*), 115  
 pwc\_shape\_plateau() (in module *c3.libraries.envelopes*), 115  
 pwc\_symmetric() (in module *c3.libraries.envelopes*), 115
- ## Q
- Quantity (class in *c3.c3objs*), 87  
 Qubit (class in *c3.libraries.chip*), 109  
 quick\_setup() (*c3.experiment.Experiment* method), 90  
 quick\_setup() (*c3.signal.gates.Instruction* method), 131
- ## R
- ramsey\_echo\_sequence() (in module *c3.utils.qt\_utils*), 134  
 ramsey\_sequence() (in module *c3.utils.qt\_utils*), 134  
 random\_sample() (in module *c3.libraries.sampling*), 123  
 RB() (in module *c3.libraries.fidelities*), 116  
 read\_config() (*c3.experiment.Experiment* method), 90  
 read\_config() (*c3.generator.generator.Generator* method), 102  
 read\_config() (*c3.model.Model* method), 92  
 read\_config() (*c3.parametermap.ParameterMap* method), 94  
 read\_data() (*c3.optimizers.modellearning.ModelLearning* method), 127  
 Readout (class in *c3.generator.devices*), 100  
 readout() (*c3.generator.devices.Readout* method), 100  
 rect() (in module *c3.libraries.envelopes*), 115  
 replace\_logdir() (*c3.optimizers.optimizer.Optimizer* method), 128  
 replace\_symlink() (in module *c3.utils.utils*), 139

- rescale() (*c3.libraries.tasks.MeasurementRescale method*), 124
- Resonator (*class in c3.libraries.chip*), 110
- resonator() (*in module c3.libraries.hamiltonians*), 121
- Response (*class in c3.generator.devices*), 101
- ResponseFFT (*class in c3.generator.devices*), 101
- result() (*c3.qiskit.c3\_job.C3Job method*), 143
- rms\_dist() (*in module c3.libraries.estimators*), 116
- rms\_exp\_stds\_dist() (*in module c3.libraries.estimators*), 116
- rms\_sim\_stds\_dist() (*in module c3.libraries.estimators*), 116
- rotation() (*in module c3.utils.qt\_utils*), 134
- run() (*c3.qiskit.c3\_backend.C3QasmSimulator method*), 141
- run\_cfg() (*in module c3.main*), 95
- run\_experiment() (*c3.qiskit.c3\_backend.C3QasmPerfectSimulator method*), 125
- run\_experiment() (*c3.qiskit.c3\_backend.C3QasmPhysicsSimulator method*), 140
- run\_experiment() (*c3.qiskit.c3\_backend.C3QasmSimulator method*), 142
- RX90mGate (*class in c3.qiskit.c3\_gates*), 143
- RX90pGate (*class in c3.qiskit.c3\_gates*), 144
- RXpGate (*class in c3.qiskit.c3\_gates*), 144
- RY90mGate (*class in c3.qiskit.c3\_gates*), 144
- RY90pGate (*class in c3.qiskit.c3\_gates*), 144
- RYpGate (*class in c3.qiskit.c3\_gates*), 144
- RZ90mGate (*class in c3.qiskit.c3\_gates*), 144
- RZ90pGate (*class in c3.qiskit.c3\_gates*), 144
- RZpGate (*class in c3.qiskit.c3\_gates*), 144
- ## S
- sampling\_reg\_deco() (*in module c3.libraries.sampling*), 123
- sanitize\_instructions() (*c3.qiskit.c3\_backend.C3QasmSimulator method*), 142
- select\_from\_data() (*c3.optimizers.modellearning.ModelLearning method*), 127
- Sensitivity (*class in c3.optimizers.sensitivity*), 129
- sensitivity() (*c3.optimizers.sensitivity.Sensitivity method*), 130
- set\_algorithm() (*c3.optimizers.optimizer.Optimizer method*), 128
- set\_c3\_experiment() (*c3.qiskit.c3\_backend.C3QasmSimulator method*), 142
- set\_callback\_fids() (*c3.optimizers.optimalcontrol.OptimalControl method*), 125
- set\_components() (*c3.model.Model method*), 92
- set\_created\_by() (*c3.experiment.Experiment method*), 90
- set\_created\_by() (*c3.optimizers.optimizer.Optimizer method*), 129
- set\_deco() (*in module c3.libraries.fidelities*), 119
- set\_dephasing\_strength() (*c3.model.Model method*), 92
- set\_device\_config() (*c3.qiskit.c3\_backend.C3QasmSimulator method*), 142
- set\_dressed() (*c3.model.Model method*), 92
- set\_enable\_store\_unitaries() (*c3.experiment.Experiment method*), 90
- set\_eval\_func() (*c3.optimizers.calibration.Calibration method*), 126
- set\_exp() (*c3.optimizers.optimizer.Optimizer method*), 129
- set\_fid\_func() (*c3.optimizers.optimalcontrol.OptimalControl method*), 92
- set\_FR() (*c3.model.Model method*), 92
- set\_limits() (*c3.c3objs.Quantity method*), 88
- set\_lindbladian() (*c3.model.Model method*), 92
- set\_logdir() (*c3.optimizers.optimizer.TensorBoardLogger method*), 129
- set\_max\_excitations() (*c3.model.Model method*), 92
- set\_opt\_gates() (*c3.experiment.Experiment method*), 90
- set\_opt\_gates\_seq() (*c3.experiment.Experiment method*), 90
- set\_opt\_map() (*c3.parametermap.ParameterMap method*), 94
- set\_opt\_value() (*c3.c3objs.Quantity method*), 88
- set\_parameters() (*c3.parametermap.ParameterMap method*), 94
- set\_parameters\_scaled() (*c3.parametermap.ParameterMap method*), 94
- set\_prop\_method() (*c3.experiment.Experiment method*), 90
- set\_subspace\_index() (*c3.libraries.chip.PhysicalComponent method*), 109
- set\_tasks() (*c3.model.Model method*), 92
- set\_tf\_log\_level() (*in module c3.utils.tf\_utils*), 135
- set\_value() (*c3.c3objs.Quantity method*), 88
- show\_table() (*in module c3.utils.log\_reader*), 138
- single\_eval() (*in module c3.libraries.algorithms*), 105
- single\_length\_RB() (*in module c3.utils.qt\_utils*), 134
- SkinEffectResponse (*class in c3.generator.devices*), 101
- slepian\_fourier() (*in module c3.libraries.envelopes*), 115
- SNAIL (*class in c3.libraries.chip*), 111
- start\_log() (*c3.optimizers.optimizer.BaseLogger method*), 127
- start\_log() (*c3.optimizers.optimizer.Optimizer method*), 127

- method), 129
- start\_log() (*c3.optimizers.optimizer.TensorBoardLogger* method), 129
- state\_deco() (*in module c3.libraries.fidelities*), 119
- state\_transfer\_infid() (*in module c3.libraries.fidelities*), 119
- state\_transfer\_infid\_set() (*in module c3.libraries.fidelities*), 119
- status() (*c3.qiskit.c3\_job.C3Job* method), 143
- std\_of\_diffs() (*in module c3.libraries.estimators*), 116
- step\_response\_function() (*c3.generator.devices.ExponentialIIR* method), 98
- step\_response\_function() (*c3.generator.devices.HighpassExponential* method), 99
- step\_response\_function() (*c3.generator.devices.SkinEffectResponse* method), 101
- step\_response\_function() (*c3.generator.devices.StepFuncFilter* method), 101
- StepFuncFilter (*class in c3.generator.devices*), 101
- store\_Udict() (*c3.experiment.Experiment* method), 91
- store\_values() (*c3.parametermap.ParameterMap* method), 94
- str\_parameters() (*c3.parametermap.ParameterMap* method), 94
- submit() (*c3.qiskit.c3\_job.C3Job* method), 143
- subtract() (*c3.c3objs.Quantity* method), 88
- super\_to\_choi() (*in module c3.utils.tf\_utils*), 135
- sweep() (*in module c3.libraries.algorithms*), 105
- ## T
- T1\_sequence() (*in module c3.utils.qt\_utils*), 132
- Task (*class in c3.libraries.tasks*), 124
- task\_deco() (*in module c3.libraries.tasks*), 124
- TensorBoardLogger (*class in c3.optimizers.optimizer*), 129
- tf\_abs() (*in module c3.utils.tf\_utils*), 135
- tf\_abs\_squared() (*in module c3.utils.tf\_utils*), 135
- tf\_adadelta() (*in module c3.libraries.algorithms*), 105
- tf\_adam() (*in module c3.libraries.algorithms*), 106
- tf\_ave() (*in module c3.utils.tf\_utils*), 135
- tf\_average\_fidelity() (*in module c3.utils.tf\_utils*), 135
- tf\_choi\_to\_chi() (*in module c3.utils.tf\_utils*), 135
- tf\_convolve() (*in module c3.utils.tf\_utils*), 136
- tf\_diff() (*in module c3.utils.tf\_utils*), 136
- tf\_dm\_to\_vec() (*in module c3.utils.tf\_utils*), 136
- tf\_dm\_dm\_fid() (*in module c3.utils.tf\_utils*), 136
- tf\_dm\_ket\_fid() (*in module c3.utils.tf\_utils*), 136
- tf\_ket\_ket\_fid() (*in module c3.utils.tf\_utils*), 136
- tf\_kron() (*in module c3.utils.tf\_utils*), 136
- tf\_limit\_gpu\_memory() (*in module c3.utils.tf\_utils*), 136
- tf\_list\_avail\_devices() (*in module c3.utils.tf\_utils*), 136
- tf\_log10() (*in module c3.utils.tf\_utils*), 136
- tf\_log\_level\_info() (*in module c3.utils.tf\_utils*), 136
- tf\_matmul\_left() (*in module c3.utils.tf\_utils*), 136
- tf\_matmul\_n() (*in module c3.utils.tf\_utils*), 136
- tf\_matmul\_right() (*in module c3.utils.tf\_utils*), 136
- tf\_measure\_operator() (*in module c3.utils.tf\_utils*), 137
- tf\_project\_to\_comp() (*in module c3.utils.tf\_utils*), 137
- tf\_rmsprop() (*in module c3.libraries.algorithms*), 106
- tf\_setup() (*in module c3.utils.tf\_utils*), 137
- tf\_sgd() (*in module c3.libraries.algorithms*), 106
- tf\_spost() (*in module c3.utils.tf\_utils*), 137
- tf\_spre() (*in module c3.utils.tf\_utils*), 137
- tf\_state\_to\_dm() (*in module c3.utils.tf\_utils*), 137
- tf\_super() (*in module c3.utils.tf\_utils*), 137
- tf\_super\_to\_fid() (*in module c3.utils.tf\_utils*), 137
- tf\_superoper\_average\_fidelity() (*in module c3.utils.tf\_utils*), 137
- tf\_superoper\_unitary\_overlap() (*in module c3.utils.tf\_utils*), 137
- tf\_unitary\_overlap() (*in module c3.utils.tf\_utils*), 137
- tf\_vec\_to\_dm() (*in module c3.utils.tf\_utils*), 138
- third\_order() (*in module c3.libraries.hamiltonians*), 121
- Transmon (*class in c3.libraries.chip*), 111
- TransmonExpanded (*class in c3.libraries.chip*), 112
- trapezoid() (*in module c3.libraries.envelopes*), 115
- two\_qubit\_gate\_tomography() (*in module c3.utils.qt\_utils*), 135
- ## U
- unitary\_deco() (*in module c3.libraries.fidelities*), 119
- unitary\_infid() (*in module c3.libraries.fidelities*), 119
- unitary\_infid\_set() (*in module c3.libraries.fidelities*), 120
- update\_dressed() (*c3.model.Model* method), 93
- update\_drift\_eigen() (*c3.model.Model* method), 93
- update\_Hamiltonians() (*c3.model.Model* method), 92
- update\_Lindbladians() (*c3.model.Model* method), 92
- update\_model() (*c3.model.Model* method), 93
- update\_parameters() (*c3.parametermap.ParameterMap* method), 95
- ## V
- VoltsToHertz (*class in c3.generator.devices*), 102

## W

- `write_config()` (*c3.experiment.Experiment* method), 91
- `write_config()` (*c3.generator.devices.Device* method), 98
- `write_config()` (*c3.generator.generator.Generator* method), 103
- `write_config()` (*c3.model.Model* method), 93
- `write_config()` (*c3.parametermap.ParameterMap* method), 95
- `write_config()` (*c3.signal.pulse.Carrier* method), 131
- `write_config()` (*c3.signal.pulse.Envelope* method), 131
- `write_params()` (*c3.optimizers.optimizer.TensorBoardLogger* method), 129

## X

- `x_drive()` (in module *c3.libraries.hamiltonians*), 121
- `xy_basis()` (in module *c3.utils.qt\_utils*), 135

## Y

- `y_drive()` (in module *c3.libraries.hamiltonians*), 121

## Z

- `z_drive()` (in module *c3.libraries.hamiltonians*), 121