

# Documentation - QuantumDeviceSimulation

This is the documentation for a simulation tool created as part of my Master Thesis. Ultimately, this module is built as a wrapper for [QuTiP](#) in an attempt to ease simulation of routine tasks in Superconducting Qubits.

The module is built up of three main parts: [Devices](#), [Simulation](#) and [Analysis](#) which are working together to build, simulate and analyze the desired superconducting system.

The complete content of this module:

---

## [Devices](#)

To built up the devices and quantum chips, we have all statics in the device part of the module. The main goals are to calculate and store:

- Hamiltonians from calibrated or device parameters
  - Decoherence operators
  - Interaction between different devices in a so called [Devices/System](#)
  - Different pulses which can be send in to the devices to interact with them
- 

## [Experiment](#)

When devices are designed, the time evolution can be calculated at different degrees of complexity. These simulation strategies are found in this part of the module. And currently support the following:

- Unitary Evolutions using the Schrodinger Equation
  - Lindblad Evolution which also take decoherence into account by evolving the total density matrix and collapse operators
    - This can be done by using the Lindblad Master Equation to do deterministically
    - Or by doing a Monte Carlo Style Experiment
- 

## [Analysis](#)

Lastly, we have a module to do common analysis of the simulation traces. This is still very much under development. The hope is, however, that it should take xarrays of the same type support in [OPX Control](#). Hopefully this will decrease the distance between simulations and experiment to hopefully integrate the two together.

## Simulation

The simulation is the module responsible for the time integration. Most of the functionality is collected in the `SimulationExperiment` - parent class which keeps track of sweeping, data storage, calculation of expectation values etc.. The `SimulationExperiment` is subclassed in order to provide a method for simulating. This allows us to simulate using different models like Unitary, Lindblad or Stochastic simulations.

The following subclasses are available:

- [Schrödinger Experiment](#) allows unitary evolution without loss. This is however one dimensional and can go very fast.
- [Lindblad Experiment](#) is a deterministic evolution using the Lindblad Master Equation to take care of decoherence and leakage due to interaction with the environment
- [Monte Carlo Experiment](#) takes care of decoherence and losses to the environment by applying them stochastically. This means that we can approximate the Lindblad solution by dialing up the number of trajectories.
- [Stochastic Master Equation](#) is used for simulating homodyne and heterodyne measurements of the system and how it behaves under continuous monitoring.

## Simulation Experiment Class

### Note

This class uses a `Dataclass` to store data from simulation and save it. In the future this should be changed such that it comes in the same way as from the 'OPX\_control' library which is used in the lab.

The `SimulationExperiment` call the overwritten `simulate` method to simulate a configuration. This now takes care of looping over the swept parameters defined in the [Systems](#) simulated. At the moment it supports sweeps over 1 or 2 parameters as well as the possibility to save the state or density matrix at each time in the simulation.

## Subclasses

### Note

Maybe move the subclasses to each its own file. It is probably nice with a guide how to use it and an example for each.

## Schrödinger Experiment

The simplest implemented experiment is the Schrödinger experiment which takes states and simply evolves them using the Schrödinger equation:

$$\frac{d}{dt}|\psi\rangle = -i\hbar\hat{H}|\psi\rangle$$

The Schrödinger equation is unitary evolutions and does not support decoherences, density matrices or measurements.

Since the Schrödinger equation is unitary, it is not necessary to keep track of a whole density matrix and is for this reason the fastest of the possible simulations.

To run an experiment it can be defined using:

```
experiment = SchoedingerExperiment(  
    system: System,  
    states: Iterable[qutip.Qobj],  
    times: Iterable[float],  
    expectation_operators: list[qutip.Qobj] = [],  
    store_states: bool = False,  
    store_measurements: bool = False,  
    only_store_final: bool = False,  
    save_path: str = None,  
)
```

And can then be run simply by:

```
results = experient.run()
```

The parameters of the `SchroedingerExperiment` are as follows:

Parameter	Function
system	The System that should be simulated
states	States or list of states to simulate
times	List of times for simulation
expectation_operators	List of operators for which an expectation value should be calculated
store_states	Whether the states should be stored
only_store_final	Whether only the final state should be considered in storing states and calculating expectation vaues
save_path	What path the final results should be saved to

## Lindblad Experiment

To consider losses and decoherences from the system, one should use the Lindblad Master equation to simulate the time evolution of a density matrix in contact with the environment. The master equation which is evolved is given by:

$$\dot{\rho}(t) = -idt[H, \rho(t)] + \sum_a \left( L_a \rho(t) L_a^\dagger - \frac{1}{2} L_a L_a^\dagger \rho(t) - \frac{1}{2} \rho(t) L_a L_a^\dagger \right)$$

where  $L_\alpha$  is dissipation operators.

The Lindblad Master Equation considers a linear equation for the density matrix and for this reason it scales heavy with the size of the Hilbert Space. For this reason, it is significantly slower than the Schrödinger equation for high dimensional problems.

To simulate the Lindblad Master equation the following experiment should be set up in exactly the same way as the `SchoedingerExperiment`, but will consider dissipation operators of the system:

```

experiment = LindbladExperiment(
    system: System,
    states: Iterable[qutip.Qobj],
    times: Iterable[float],
    expectation_operators: list[qutip.Qobj] = [],
    store_states: bool = False,
    store_measurements: bool = False,
    only_store_final: bool = False,
    save_path: str = None,
)

```

And can be run by:

```

experiment.run()

```

The parameters are also the same as `SchroedinerExperiment` :

Parameter	Function
system	The System that should be simulated
states	States or list of states to simulate
times	List of times for simulation
expectation_operators	List of operators for which an expectation value should be calculated
store_states	Whether the states should be stored
only_store_final	Whether only the final state should be considered in storing states and calculating expectation values
save_path	What path the final results should be saved to

## Monte Carlo Experiment

For larger dimensions it can be beneficial to add the collapse operators stochastically by using the Monte Carlo Simulation. This simulation will apply a collapse operator depending on the given rate and time step. By doing this, the problem is still one dimensional and can be repeated multiple times to approximate the Lindblad equation.

A Monte Carlo Experiment is setup using:

```

experiment = MonteCarloExperiment(
    system: System,
    states: Iterable[qutip.Qobj],
    times: Iterable[float],
    expectation_operators: list[qutip.Qobj] = [],
    store_states: bool = False,
    store_measurements: bool = False,
    only_store_final: bool = False,
    save_path: str = None,
    ntraj: int = 1,
    exp_val_method="average",
)

```

And is run by:

```
results = experiment.run()
```

The parameters of the `MonteCarloExperiment` are:

Parameter	Function
system	The System that should be simulated
states	States or list of states to simulate
times	List of times for simulation
expectation_operators	List of operators for which an expectation value should be calculated
store_states	Whether the states should be stored
only_store_final	Whether only the final state should be considered in storing states and calculating expectation values
save_path	What path the final results should be saved to
ntraj	How many times to repeat each subexperiment
exp_val_method	When set to "average" this will take the average of all trajectories otherwise expectation values for each trajectory is returned

## Stochastic Master Equation

The most complex experiment is evolving the stochastic master equation. In addition, to including decoherence and dissipation this also allows for measurement feedback and a measurement record.

The Stochastic Master Equation takes the form:

$$d\rho = -i[H, \rho]dt + \mathcal{D}[c]\rho dt + \mathcal{H}[c]\rho dW$$

Where the superoperators refer to the Lindblad dissipator:

$$\mathcal{D}[c]\rho(t) = c\rho(t)c^\dagger - \frac{1}{2}cc^\dagger\rho(t) - \frac{1}{2}\rho(t)cc^\dagger$$

And the stochastic part given by:

$$\mathcal{H}[c]\rho(t) = c\rho(t) + \rho(t)c - \langle c + c^\dagger \rangle \rho(t)$$

And  $dW$  is a stochastic variable of the Wiener process with variance  $dt$ .

Currently this is simulated either by using a homodyne or heterodyne setup of the collapse operator using the `method` keyword, depending on whether one or two quadratures should be measured.

When using `StochasticMasterEquation` the system parameter `system.stochastic_dissipators` will also be considered.

The results of the `StochasticMasterEquation` will include `measurements`. Which are equal to the measurement record given by:

$$dr = \langle X \rangle dt + \frac{dW}{\sqrt{8k}}$$

The experiment is defined using:

```

experiment = StochasticMasterEquation(
    system: System,
    states: Iterable[qutip.Qobj],
    times: Iterable[float],
    expectation_operators: list[qutip.Qobj] = [],
    store_states: bool = False,
    store_measurements: bool = False,
    only_store_final: bool = False,
    save_path: str = None,
    ntraj: int = 1,
    exp_val_method="average",
    method: str = "heterodyne",
    store_measurements: bool = True,
    nsubsteps: int = 1,
)

```

And run like the others by:

```

results = experiment.run()

```

The parameters are:

Parameter	Function
system	The System that should be simulated
states	States or list of states to simulate
times	List of times for simulation
expectation_operators	List of operators for which an expectation value should be calculated
store_states	Whether the states should be stored
only_store_final	Whether only the final state should be considered in storing states and calculating expectation values
save_path	What path the final results should be saved to
ntraj	How many times to repeat each subexperiment
exp_val_method	When set to "average" this will take the average of all trajectories otherwise expectation values for each trajectory is returned
method	Set to "homodyne" or "heterodyne" depending on one or two quadrature measurement
store_measurements	whether to store the measurement record
nsubsteps	If there should be substeps in the simulation between returned points

## Devices

All the physical devices and pulses are written as children to the `Device`-class.

All devices are collected in three main categories:

- [Device](#) store the physical devices with Hamiltonian, decays and other parameters

- [Systems](#) are connecting physical devices from the [Device](#) class. This can be to combine multiple qubits, qubit and a resonator, or drive them with pulses.
- [Pulses](#) are different time dependent functions which can be coupled at the appropriate keyword in a [System](#).

## List of Devices:

A running list of devices, systems and pulses are found here:

- [Device](#)
  - [Simple Qubit](#)
  - [Transmon](#)
  - [Resonator](#)
- [Systems](#)
  - [Qubit System](#)
  - [Qubit Resonator System](#)
  - [Approximated Systems](Systems#Approximated Systems)
- [Pulses](#)
  - [GaussianPulse](#)
  - [SquareCosinePulse](#)
  - [CloakedPulse](#)

## Device Parent Class

The Device Parent Class:

```
Device(ABC):

    def set_operators(self) -> None:
        SHOULD BE OVERWRITTEN TO SET DEVICES OPERATORS GIVEN PARAMETERS
```

is an abstract class made to keep track of static and sweepable parameters. It has the following methods. When subclassed it should have a new version which creates a new init calling the parent and overwriting the `Device.set_operators` method.

The `init(self)` should define a `self.sweepable_parameters` a list with strings referring to the defined parameters which should have the ability to be swept in an [.md](#). Furthermore, it should also include `self.update_methods`

An example could be the following:

```
def NewDevice(Device):

    def __init__(self, ...):
        DEFINE NEW PARAMETERS HERE FOR THE CLASS
        self.sweepable_parameters = ["PARAM1", "PARAM2"]
        self.update_methods = [self.set_operators]
        super().__init__()
```

```
def set_operators(self):
    SET HAMILTONIAN AND OTHER OPERATORS HERE
    self.hamiltonian = qutip.number(self.levels) # For Example a harmonic oscillator
```

## Device

Device is part of the Devices submodule which contains the different children classes to the [Device Parent Class](Devices#Device Parent Class).

## Qubits

### Simple Qubit

The most basic device is the simple qubit. This is simply defined from a qubit frequency given in GHz which defines the energy gap between  $|0\rangle$  and  $|1\rangle$ , along with the anharmonicity. If the anharmonicity is None, the Qubit is a simple two-level system, but with it defined, there will be a third level  $|2\rangle$  with energy  $2f_{01} + \alpha$  where  $\alpha$  is the anharmonicity.

Furthermore, the qubit can be defined with a decay by defining a  $T_1 \neq 0$ .

```
SimpleQubit(
    frequency: float,
    anharmonicity: float = None,
    T1: float = 0.0,
):
```

The sweepable parameters of the SimpleQubit are:

Parameter	Use	Sweepable
frequency	The energy spacing between the 0 and 1 level in GHz	x
anharmonicity	The difference in energy splitting between 2-1 and 1-0 . (Given in Ghz)	x
T1	The characteristic time of qubit decay	x

And the update methods calculates the following operators/dissipators:

- Hamiltonian
- Charge Matrix
- Dissipators
  - Qubit Decay

## Transmon

The Transmon qubit defines an n-level anharmonic system from the physical parameters Transmon Device. The Hamiltonian and Charge Matrix are calculated numerically by diagonalizing the Hamiltonian in the charge basis of the charge matrix. (see [CircuitQ: an open-source toolbox for superconducting circuits](#))

To define the Transmon call the following:



```
SimpleQubit(
    self,
    EC: float,
    EJ: float,
    n_cutoff: int = 20,
    ng: float = 0.0,
    levels: int = 3,
    T1: float = 0.0,
)
```

The parameters of the Transmon qubit are:

Parameter	Use	Sweepable
EC	Energy associated with capacitor in GHz	x
EJ	Energy associated with Josephson Junction in GHz	x
n_cutoff	Number of charges to include in calculations of hamiltntonian.	
ng	Charge offset in units of 2e	x
levels	Define how many energy levels of the Transmon should be considered.	
T1	The characteristic time of qubit decay process	x

And the update methods calculates the following operators/dissipators:

- Hamiltonian
- Charge Matrix
- Dissipators
  - Qubit Decay

## Resonator

The resonator is a defined as a quantum harmonic oscillator with energy levels  $(n + \frac{1}{2})2\pi f$  with  $f$  the frequency of the resonator. It further supports decay of the resonator given by the characteristic time  $\kappa$ .

To define a resonator use the following:

```
Resonator(
    frequency: float,
    levels=10,
    kappa: float = 0
)
```

The sweepable parameters of the Resonator are:

Parameter	Use	Sweepable
frequency	The energy spacing between levels in GHz	x
levels	The number of levels to consider	
kappa	The characteristic time of photon decay	x

Which updates the following operators and dissipators:

- Hamiltonian
- Coupling Operator
- Dissipators
  - Qubit Decay

## Systems

In this module, a **system** is made to simulate the interaction between different [devices](#) such that different interactions can be calculated. The system class also takes care of propagating updates directly to the devices which it is built of while also maintaining its own sweepable parameters.

### System Parent Class

The **System parent class** defines much of the logistics for the updating parameters in the overall system or in the device which it is made of.

The parent class takes the following abstract form:

```
class System(ABC):
    @abstractmethod
    def set_operators(self):
        WRITE THIS FUNCTION SUCH THAT IT UPDATES THE OPERATORS
        DEVICE OPERATORS ARE UPDATED BEFORE THIS FUNCTION IS CALLED

    @abstractmethod
    def get_states(self):
        THIS FUNCTION SHOULD BE USED TO GET THE BASIS STATES OF THE SYSTEM
```

In addition to the new methods, the new `init()` function should also define a `self.sweepableparameters` and a `self.update_methods`. *Maybe also even a `self.dimensions`?*

An example for defining a new system can be seen here:

```
class NewSystem(System):

    def __init__(self, qubit, PARAM1, PARAM2):
        CALCULATIONS HERE

        self.sweepable_parameters = ["PARAM1"]
        self.update_method = [self.update_operators, self.update_dissipators]

    def set_operators(self):
        CALCULATE HAMILTONIAN FOR THE ENTIRE SYSTEM HERE

    def set_dissipators(self):
        CALCULATE THE DISSIPATORS HERE

    def get_states(self, state_numbers):
        states = FIND THE STATES HERE
        return states

    ...
```

```
## Systems
```

Some simple systems are already defined **in** the module **and** are documented below. Some systems are approximation of these systems **and** will be found **in** the new section.

```
### QubitSystem
```

The simplest system connects a qubit to a pulse drive line. It can be defined by:

```
```python
QubitSystem(
    self,
    qubit: Device,
    qubit_pulse: Pulse = None,
)
```

To get a state, one can call the following code with state being the integer of the desired level.

```
state = QubitSystem.get_states(state: int)
```

A few simple methods are defined to get common expectation value operators.

```
# An operator for finding the number operator
QubitSystem.qubit_state_operator()

# Or the occupation for a specific state
QubitSystem.qubit_state_occupation_operator(state: int = 1)
```

## QubitResonatorSystem

The QubitResonatorSystem is made for combining one [Device > Qubits](#) class element with a [Device > Resonator](#) along with [pulses](#) each.

The QubitResonatorSystem is called with the following syntax:

```
QubitResonatorSystem(
    qubit: Device,
    resonator: Device,
    coupling_strength: float,
    resonator_pulse: Pulse = None,
    qubit_pulse: Pulse = None,
)
```

The qubit and resonator are connected with the  $g \hat{n} \otimes (a + a^\dagger)$  where  $g$  is the coupling strength,  $\hat{n}$  is the charge matrix of the qubit and  $a$  and  $a^\dagger$  are the lowering and raising operators of the resonator.

Initial states are found as  $|\text{qubit state}\rangle \otimes |\text{resonator state}\rangle$  calling:

```
QubitResonatorSystem.get_states(qubit_states: int = 0, resonator_states: int = 0)
```

And the following operators can be found to calculate common expectation values:

```
# The photon number operator by tracing out the qubit
QubitResonatorSystem.photon_number_operator()
```

```
# The qubit number operator is found:
QubitResonatorSystem.qubit_state_operator()

# The occupation operator for a specific qubit state can be found
QubitResonatorSystem.qubit_state_occupation_operator(state: int = 1)

# And the I and Q operator for measuring the quadratures of resonator can be found as
QubitResonatorSystem.resonator_I()
QubitResonatorSystem.resonator_Q()
```

## Approximated Systems

As some system very complex to simulate. For this reason a few approximations are made and implemented in order to get simpler simulations.

## DispersiveQubitResonatorSystem

The dispersive qubit resonator system uses the Jaynes Cummings model and by going into the interaction picture of a square resonator drive one can completely eliminate the time-dependence in the Hamiltonian. Furthermore, the interaction term is reduced to the linear version where the interaction term is in the eigenstates.

### Note

Not correctly implemented yet. This version does not include qubit drives

Since this system does not have any pulse the duration of the pulse the effective pulse will be always on. The duration of the pulse is equal to the simulation time.

One define such a system by:

```
DispersiveQubitResonatorSystem(
    qubit: Device,
    resonator: Device,
    drive_frequency: float,
    drive_amplitude: float,
    coupling_strength: float
)
```

Where the drive\_frequency, drive\_amplitude are the parameters that would go into [Pulses > SquareCosinePulse](#).

States and methods for operators are the same as the ones found in the [QubitResonatorSystem](#):

#todo

- ☐ Reimplement
  - ☐ Any way to keep the qubit pulse?
  - ☐ Maybe as subclass to the `QubitResonatorSystem`

# Pulses

#todo

- ☐ Create a Pulse Sequence, which simply adds multiple pulses in parallel or series

## Pulse Parent Class

At the moment the Pulse Parent class mostly serves a typing help. It just has single abstract method inheriting most functionality from the [Device Parent Class](Devices#Device Parent Class). A Pulse class takes the structure:

```
class Pulse(Device):  
  
    @abstractmethod  
    def set_pulse(self):  
        A FUNCTION THAT DEFINES A PULSE AS  
        self.pulse: callable(t, args) -> float
```

### Note

With `np.piecewise` it should be possible to write this in a vectorized form. This could hopefully help with performance.

## Square Cosine Pulse

The simplest pulse is the cosine pulse with a simple rectangular envelope. It is defined using the following:

```
SquareCosinePulse(  
    frequency: float,  
    amplitude: float,  
    start_time: float = 0,  
    duration: float = None,  
    phase: float = 0,  
)
```

Parameter	Function	Sweepable
frequency	Set the frequency of the pulse	x
amplitude	The amplitude	x
start_time	When the pulse starts	x
duration	How long it lasts	x
phase	A phase to give to the oscillating term	x

## Gaussian Pulse

The simplest pulse is the square cosine pulse. It has the following arguments:

```
GaussianPulse(
    frequency: float,
    amplitude: float,
    sigma: float,
    start_time=0,
    duration=0,
    phase=0,
    drag_alpha=0,
):
```

Where the parameters are given by the following:

Parameter	Function	Sweepable
frequency	Set the frequency of the pulse	x
amplitude	The amplitude	x
sigma	The width of the pulse given as standard deviation of the gaussian envelope	x
start_time	When the pulse starts	x
duration	How long it lasts	x
phase	A phase to give to the oscillating term	x
drag_alpha	if DRAG should be applied to the pulse, this $\alpha_{DRAG} \neq 0$	x

## Analysis

### Implementations

What do I still need to implement in the analysis library?

- ☐ Clean up in the code

### Documentation

What do I need to document

- ☐ Overview → how is the submodule built?
- ☐ Automatic Plotter when sweeping
- ☐ Q Function Methods for plotting density matrices