# QC Lab

Tempelaar Group

Jan 03, 2025

# CONTENTS

**QC Lab** is a Python package designed for implementing and executing quantum-classical (QC) dynamics simulations. It offers a framework for developing both physical models and QC algorithms that enables algorithms and models to be combined arbitrarily. QC Lab comes with a variety of already implemented models and algorithms which we hope encourage new researchers to explore the field of quantum-classical dynamics. It also provides a framework for advanced users to implement their own models and algorithms which we hope will become part of a growing library of quantum-classical dynamics tools.

CHAPTER

# ONE

# USER GUIDE

A guide for using models and algorithms that are shipped with QC Lab.

## 1.1 User Guide

### 1.1.1 Quick Start Guide

QC Lab is organized into models and algorithms which are combined into a simulation object. The simulation object fully defines a quantum-classical dynamics simulation which is then carried out by a dynamics driver. This guide will walk you through the process of setting up a simulation object and running a simulation.

#### Importing Modules

First, we import the necessary modules:

```
import numpy as np
import matplotlib.pyplot as plt
from qclab import Simulation # import simulation class
from qclab.models import SpinBosonModel # import model class
from qclab.algorithms import MeanField # import algorithm class
from qclab.dynamics import serial_driver # import dynamics driver
```

#### Instantiating Simulation Object

Next, we instantiate a simulation object. Each object has a set of default parameters which can be accessed by calling *sim.default_parameters*. Passing a dictionary to the simulation object when instantiating it will override the default parameters.

```
sim = Simulation()
print('default simulation parameters: ', sim.default_parameters)
# default simulation parameters:  {'tmax': 10, 'dt': 0.01, 'dt_output': 0.1, 'num_trajs': 10,
↪'batch_size': 1}
```

Alternatively, you can directly modify the simulation parameters by assigning new values to the parameters attribute of the simulation object. Here we change the number of trajectories that the simulation will run, and how many trajectories are run at a time (the batch size). We also change the total time of each trajectory (tmax) and the timestep used for propagation (dt).

```
# change parameters to customize simulation
sim.parameters.num_trajs = 200
sim.parameters.batch_size = 20
sim.parameters.tmax = 30
sim.parameters.dt = 0.001
```

### Instantiating Model Object

Next, we instantiate a model object. Like the simulation object, it has a set of default parameters.

```
sim.model = SpinBosonModel()
print('default model parameters: ', sim.model.default_parameters)
# default model parameters:  {'temp': 1, 'V': 0.5, 'E': 0.5, 'A': 100, 'W': 0.1, 'l_reorg': 0.
↪005, 'boson_mass': 1}
```

### Instantiating Algorithm Object

Next, we instantiate an algorithm object.

```
sim.algorithm = MeanField()
print('default algorithm parameters: ', sim.algorithm.default_parameters)
# default algorithm parameters:  {}
```

### Setting Initial State

Before using the dynamics driver to run the simulation, it is necessary to provide the simulation with an initial state. This initial state is dependent on both the model and algorithm. For mean-field dynamics, we require a diabatic wavefunction called “wf_db”. Because we are using a spin-boson model, this wavefunction should have dimension 2.

The initial state is stored in *sim.state* which must be accessed with a particular “modify” function as follows,

```
sim.state.modify('wf_db', np.array([0, 1], dtype=np.complex128))
```

### Running the Simulation

Finally, we run the simulation using the dynamics driver.

```
data = serial_driver(sim)
```

### Analyzing Results

The data object returned by the dynamics driver contains the results of the simulation in a dictionary with keys corresponding to the names of the observables that were requested to be recorded during the simulation.

```
print('calculated quantities:', data.data_dic.keys())
# calculated quantities: dict_keys(['seed', 'dm_db', 'classical_energy', 'quantum_energy'])
```

Each of the calculated quantities must be normalized with respect to the number of trajectories,

```
num_trajs = len(data.data_dic['seed'])
classical_energy = data.data_dic['classical_energy'] / num_trajs
quantum_energy = data.data_dic['quantum_energy'] / num_trajs
populations = np.real(np.einsum('tii->ti', data.data_dic['dm_db'] / num_trajs))
```

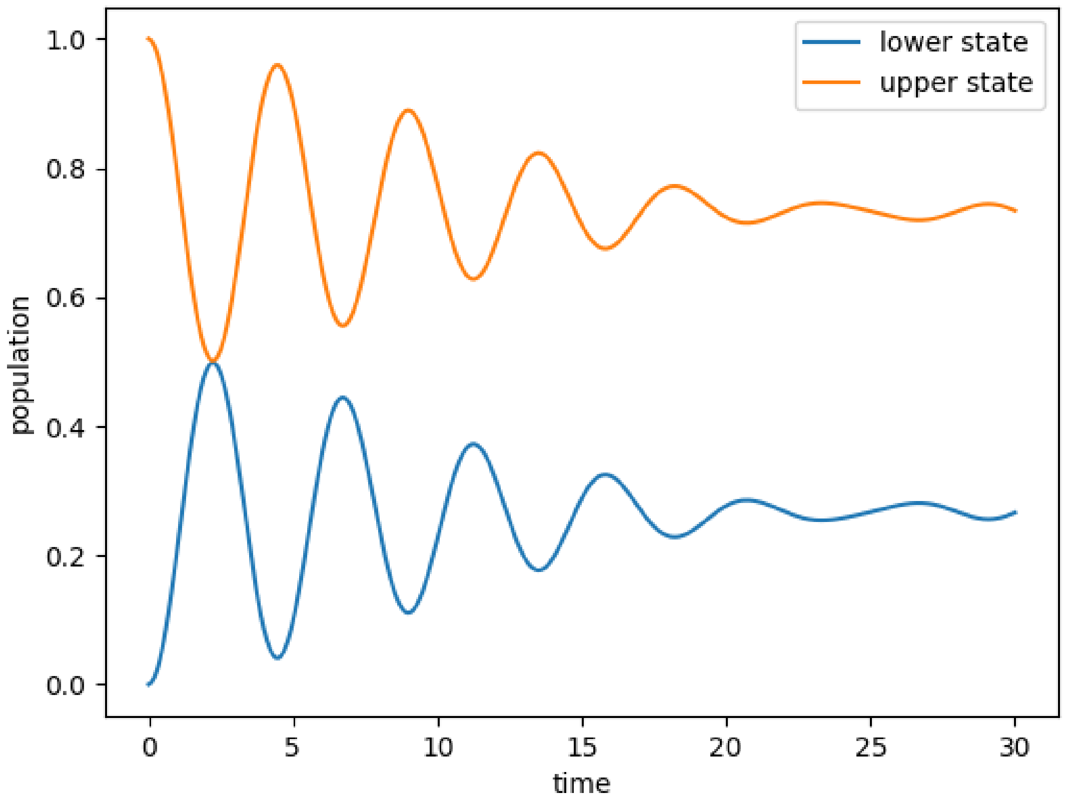
The time axis can be retrieved from the simulation object

```
time = sim.parameters.time
```

### Plotting Results

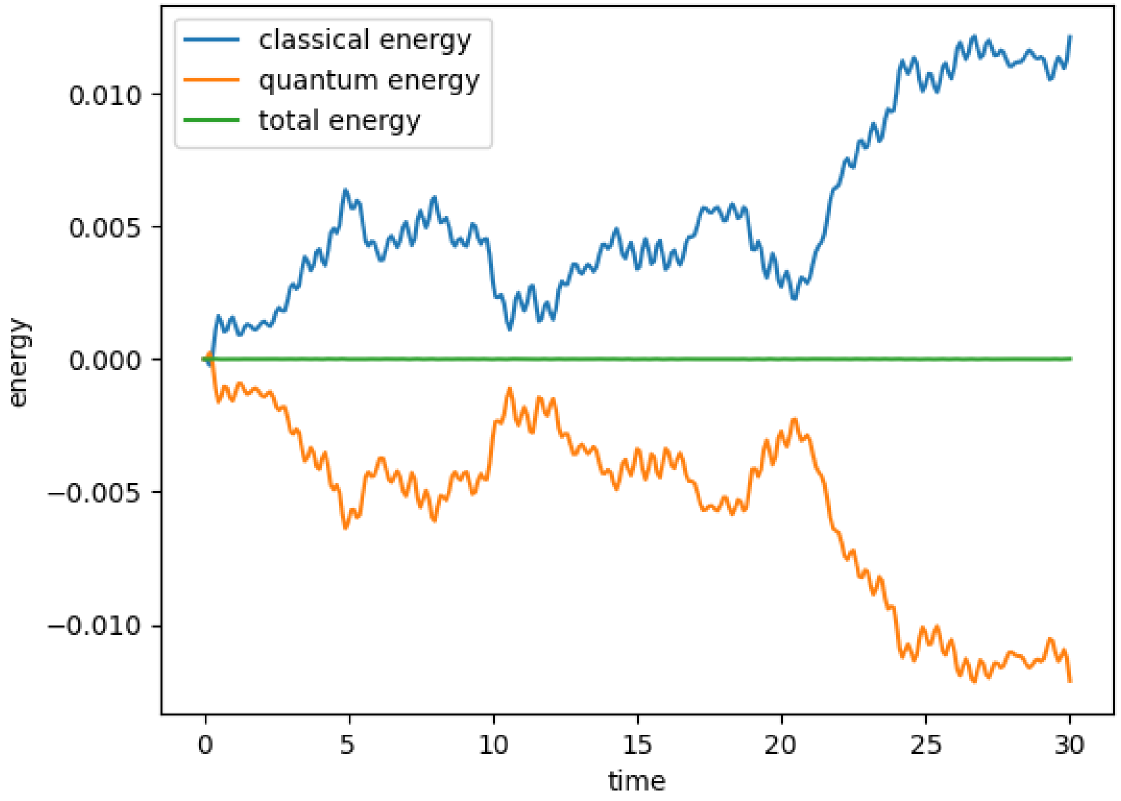
Finally, we can plot the results of the simulation like the population dynamics:

```
plt.plot(time, populations[:, 0], label='lower state')
plt.plot(time, populations[:, 1], label='upper state')
plt.xlabel('time')
plt.ylabel('population')
plt.legend()
plt.show()
```



We can verify that the total energy of the simulation was conserved by inspecting the change in energy of quantum and classical subsystems over time.

```
plt.plot(time, classical_energy - classical_energy[0], label='classical energy')
plt.plot(time, quantum_energy - quantum_energy[0], label='quantum energy')
plt.plot(time, classical_energy + quantum_energy - classical_energy[0] - quantum_
↪energy[0], label='total energy')
plt.xlabel('time')
plt.ylabel('energy')
plt.legend()
plt.show()
```



### 1.1.2 Models

Models in QC Lab are classes that define the physical properties of the system. Each model in QC Lab has a set of parameters that can be accessed by the user when the model is instantiated. The contents and structure of a model class are described in the Developer Guide.

The models currently available in QC Lab can be imported from the models module:

```
from qclab.models.spin_boson import SpinBosonModel
```

After which they can be instantiated with a set of parameters:

```
model = SpinBosonModel(parameters=dict(temp=1, V=0.5, E=0.5, A=100, W=0.1, l_reorg=0.005,
↪ boson_mass=1))
```

See the subsequent pages for more information on the models available in QC Lab including their default parameters.

## Spin-Boson Model

The quantum-classical Hamiltonian of the spin-boson model is:

$$\hat{H}_{\mathrm{q}} = \begin{pmatrix} -E & V \\ V & E \end{pmatrix}$$

$$\hat{H}_{\mathrm{q-c}} = \sigma_z \sum_{\alpha}^{A} \frac{g_\alpha}{\sqrt{2mh_\alpha}} \left(z^*_\alpha + z_\alpha\right)$$

$$H_{\mathrm{c}} = \sum_{\alpha}^{A} \omega_\alpha z^*_\alpha z_\alpha$$

The couplings and frequencies are sampled from a Debye spectral density:

$$\omega_\alpha = \Omega \tan\left(\frac{\alpha - 1/2}{2A}\pi\right)$$

$$g_\alpha = \omega_\alpha \sqrt{\frac{2\lambda}{A}}$$

Where $\Omega$ is the characteristic frequency and $\lambda$ is the reorganization energy.

The classical coordinates are sampled from a Boltzmann distribution:

$$P(z) \propto \exp\left(-\frac{H_{\mathrm{c}}(\boldsymbol{z})}{T}\right)$$

and by convention we assume that $\hbar = 1$, $k_B = 1$, and $\omega_\alpha = h_\alpha$.

### Parameters

The following table lists all of the parameters required by the *SpinBosonModel* class:

Table 1: SpinBosonModel Parameters

| Parameter (symbol) | Description | Default Value |
|---|---|---|
| *temp* $(T)$ | Temperature | 1 |
| *V* $(V)$ | Off-diagonal coupling | 0.5 |
| *E* $(E)$ | Diagonal energy | 0.5 |
| *A* $(A)$ | Number of bosons | 100 |
| *W* $(\Omega)$ | Characteristic frequency | 0.1 |
| *l_reorg* $(\lambda)$ | Reorganization energy | 0.005 |
| *boson_mass* $(m)$ | Mass of the bosons | 1 |

### Example

```python
from qclab.models import SpinBosonModel
from qclab import Simulation
from qclab.algorithms import MeanField
from qclab.dynamics import serial_driver
import numpy as np

# instantiate a simulation
sim = Simulation()

# instantiate a model
sim.model = SpinBosonModel()

# instantiate an algorithm
sim.algorithm = MeanField()

# define an initial diabatic wavefunction
sim.state.modify('wf_db',np.array([1, 0], dtype=np.complex128))

# run the simulation
data = serial_driver(sim)
```

### Holstein Lattice Model

The Holstein Lattice Model is a nearest-neighbor tight-binding model combined with an idealized optical phonon that interacts via a Holstein coupling. The current implementation accommodates a single electronic particle. The quantum-classical Hamiltonian of the Holstein model is:

$$\hat{H}_{\mathrm{q}} = -t\sum_{\langle i,j\rangle}^{N} \hat{c}_i^{\dagger}\hat{c}_j$$

where $\langle i, j\rangle$ denotes nearest-neighbor sites with or without periodic boundaries determined by the parameter *periodic_boundary=True*.

$$\hat{H}_{\mathrm{q-c}} = g\omega\sum_{i}^{N} \hat{c}_i^{\dagger}\hat{c}_i \frac{1}{\sqrt{2mh_i}}\left(z_i^* + z_i\right)$$

$$H_{\mathrm{c}} = \omega\sum_{i}^{N} z_i^* z_i$$

Here, $g$ is the dimensionless electron-phonon coupling, $\omega$ is the phonon frequency, $m$ is the phonon mass, and $h_i$ is the complex-valued coordinate parameter which we take to be $h_i = \omega$.

The classical coordinates are sampled from a Boltzmann distribution:

$$P(z) \propto \exp\left(-\frac{H_{\mathrm{c}}(\boldsymbol{z})}{T}\right)$$

and by convention we assume that $\hbar = 1$, $k_B = 1$.

### Parameters

The following table lists all of the parameters required by the *HolsteinLatticeModel* class:

Table 2: HolsteinLatticeModel Parameters

| Parameter (symbol) | Description | Default Value |
|---|---|---|
| *temp* ($T$) | Temperature | 1 |
| *g* ($g$) | Dimensionless electron-phonon coupling | 0.5 |
| *w* ($\omega$) | Phonon frequency | 0.5 |
| *N* ($N$) | Number of sites | 10 |
| *t* ($t$) | Hopping energy | 1 |
| *phonon_mass* ($m$) | Phonon mass | 1 |
| *periodic_boundary* | Periodic boundary condition | True |

### Example

```python
from qclab.models import HolsteinLatticeModel
from qclab import Simulation
from qclab.algorithms import MeanField
from qclab.dynamics import serial_driver
import numpy as np

# instantiate a simulation
sim = Simulation()

# instantiate a model
sim.model = HolsteinLatticeModel()

# instantiate an algorithm
sim.algorithm = MeanField()

# define an initial diabatic wavefunction
wf_db_0 = np.zeros((sim.model.parameters.N), dtype=np.complex128)
wf_db_0[0] = 1.0 + 0.0j
sim.state.modify('wf_db',wf_db_0)

# run the simulation
data = serial_driver(sim)
```

## 1.1.3 Algorithms

Algorithms in QC Lab are classes that can be paired with models to carry out a quantum-classical dynamics simulation. Like models, each algorithm in QC Lab has a set of parameters that allow users to control particular attributes of the algorithm. The contents and structure of an algorithm class are described in the Developer Guide.

The algorithms currently available in QC Lab can be imported from the algorithms module:

```python
from qclab.algorithms import MeanField
```

After which they can be instantiated with a set of parameters:

```
algorithm = MeanField(parameters=None)
```

Here there are no parameters passed to the *MeanField* algorithm.

### Mean-Field Dynamics

The *MeanField* class implements the mean-field dynamics algorithm. This class is part of the *qclab* library and is used to simulate quantum systems using mean-field theory.

### Class Definition

**class** qclab.algorithms.mean_field.**MeanField**(*parameters=None*)

Bases: `Algorithm`

Mean-field dynamics algorithm class.

The algorithm class has a set of parameters that define the algorithm Some of these parameters depends on the model i.e. num_branches is always the same as the number of quantum states in the model for deterministic surface hopping methods.

**initialization_recipe = [<function MeanField.<lambda>>, <function MeanField.<lambda>>]**

**output_recipe = [<function MeanField.<lambda>>, <function MeanField.<lambda>>, <function MeanField.<lambda>>]**

**output_variables = ['dm_db', 'classical_energy', 'quantum_energy']**

**update_recipe = [<function MeanField.<lambda>>, <function MeanField.<lambda>>, <function MeanField.<lambda>>]**

### Initialization

The *MeanField* class is initialized with a set of parameters. These parameters can be provided as a dictionary. If no parameters are provided, default parameters are used.

```
from qclab.algorithms import MeanField

parameters = {
    'param1': value1,
    'param2': value2,
    # ... other parameters ...
}

mean_field = MeanField(parameters)
```

## Recipes

The *MeanField* class uses three main recipes for its operations:

1. **Initialization Recipe**: This recipe initializes the simulation state.
2. **Update Recipe**: This recipe updates the simulation state at each time step.
3. **Output Recipe**: This recipe computes the output variables from the simulation state.

### Initialization Recipe

```
initialization_recipe = [
    lambda sim, state: tasks.initialize_z_coord(sim=sim, state=state, seed=state.seed),
    lambda sim, state: tasks.update_h_quantum_vectorized(sim=sim, state=state, z_
↪coord=state.z_coord),
]
```

### Update Recipe

```
update_recipe = [
    lambda sim, state: tasks.update_h_quantum_vectorized(sim=sim, state=state, z_
↪coord=state.z_coord),
    lambda sim, state: tasks.update_z_coord_rk4_vectorized(sim=sim, state=state, z_
↪coord=state.z_coord,
                                                           output_name='z_coord',␣
↪wf=state.wf_db,
                                                           update_quantum_classical_
↪forces_bool=False),
    lambda sim, state: tasks.update_wf_db_rk4_vectorized(sim=sim, state=state),
]
```

### Output Recipe

```
output_recipe = [
    lambda sim, state: tasks.update_dm_db_mf_vectorized(sim=sim, state=state),
    lambda sim, state: tasks.update_quantum_energy_mf_vectorized(sim=sim, state=state,␣
↪wf=state.wf_db),
    lambda sim, state: tasks.update_classical_energy_vectorized(sim=sim, state=state, z_
↪coord=state.z_coord),
]
```

### Output Variables

The *MeanField* class computes the following output variables:

- *dm_db*: The density matrix database.
- *classical_energy*: The classical energy of the system.
- *quantum_energy*: The quantum energy of the system.

```
output_variables = [
    'dm_db',
    'classical_energy',
    'quantum_energy',
]
```

## Fewest-Switches Surface Hopping

The *FewestSwitchesSurfaceHopping* class implements the fewest-switches surface hopping algorithm. This class is part of the *qclab* library and is used to simulate quantum systems using surface hopping methods.

### Class Definition

**class** qclab.algorithms.fewest_switches_surface_hopping.**FewestSwitchesSurfaceHopping**(*parameters=None*)

Bases: `Algorithm`

**initialization_recipe = [<function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>]**

**output_recipe = [<function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>, <function FewestSwitchesSurfaceHopping.<lambda>>]**

**output_variables = ['quantum_energy', 'classical_energy', 'dm_db']**

```
update_recipe = [<function FewestSwitchesSurfaceHopping.<lambda>>, <function
FewestSwitchesSurfaceHopping.<lambda>>, <function
FewestSwitchesSurfaceHopping.<lambda>>, <function
FewestSwitchesSurfaceHopping.<lambda>>, <function
FewestSwitchesSurfaceHopping.<lambda>>, <function
FewestSwitchesSurfaceHopping.<lambda>>, <function
FewestSwitchesSurfaceHopping.<lambda>>, <function
FewestSwitchesSurfaceHopping.<lambda>>, <function
FewestSwitchesSurfaceHopping.<lambda>>, <function
FewestSwitchesSurfaceHopping.<lambda>>]
```

### Initialization

The *FewestSwitchesSurfaceHopping* class is initialized with a set of parameters. These parameters can be provided as a dictionary. If no parameters are provided, default parameters are used.

```python
from qclab.algorithms import FewestSwitchesSurfaceHopping

parameters = {
    'fssh_deterministic': False,
    'num_branches': 2,
    'gauge_fixing': 2,
    # ... other parameters ...
}

fssh = FewestSwitchesSurfaceHopping(parameters)
```

### Recipes

The *FewestSwitchesSurfaceHopping* class uses three main recipes for its operations:

1. **Initialization Recipe**: This recipe initializes the simulation state.
2. **Update Recipe**: This recipe updates the simulation state at each time step.
3. **Output Recipe**: This recipe computes the output variables from the simulation state.

### Initialization Recipe

```python
initialization_recipe = [
    lambda sim, state: tasks.initialize_z_coord(sim=sim, state=state, seed=state.seed),
    lambda sim, state: tasks.broadcast_var_to_branch_vectorized(sim=sim, state=state,␣
↪val=state.z_coord,
                                                                name='z_coord_branch'),
    lambda sim, state: tasks.broadcast_var_to_branch_vectorized(sim=sim, state=state,␣
↪val=state.wf_db,
                                                                name='wf_db_branch'),
    lambda sim, state: tasks.update_h_quantum_vectorized(sim=sim, state=state, z_
↪coord=state.z_coord_branch),
    lambda sim, state: tasks.diagonalize_matrix_vectorized(sim=sim, state=state,␣
↪matrix=state.h_quantum,
```

(continues on next page)

(continued from previous page)

```
                                                            eigvals_name='eigvals',␣
↪eigvecs_name='eigvecs'),
    lambda sim, state: tasks.gauge_fix_eigs_vectorized(sim=sim, state=state,␣
↪eigvals=state.eigvals,
                                                        eigvecs=state.eigvecs, eigvecs_
↪previous=state.eigvecs,
                                                        output_eigvecs_name='eigvecs', z_
↪coord=state.z_coord_branch,
                                                        gauge_fixing=2),
    lambda sim, state: tasks.copy_value_vectorized(sim=sim, state=state, val=state.
↪eigvecs,
                                                    name='eigvecs_previous'),
    lambda sim, state: tasks.copy_value_vectorized(sim=sim, state=state, val=state.
↪eigvals,
                                                    name='eigvals_previous'),
    lambda sim, state: tasks.basis_transform_vec_vectorized(sim=sim, state=state, input_
↪vec=state.wf_db_branch,
                                                             basis=np.einsum('...ij->...ji
↪', state.eigvecs).conj(),
                                                             output_name='wf_adb_branch'),
    lambda sim, state: tasks.initialize_random_values_fssh(sim=sim, state=state),
    lambda sim, state: tasks.initialize_active_surface(sim=sim, state=state),
    lambda sim, state: tasks.initialize_dm_adb_0_fssh_vectorized(sim=sim, state=state),
    lambda sim, state: tasks.update_act_surf_wf_vectorized(sim=sim, state=state),
    lambda sim, state: tasks.update_quantum_energy_fssh_vectorized(sim=sim, state=state),
    lambda sim, state: tasks.initialize_timestep_index(sim=sim, state=state),
]
```

### Update Recipe

```
update_recipe = [
    lambda sim, state: tasks.copy_value_vectorized(sim=sim, state=state, val=state.
↪eigvecs,
                                                    name='eigvecs_previous'),
    lambda sim, state: tasks.copy_value_vectorized(sim=sim, state=state, val=state.
↪eigvals,
                                                    name='eigvals_previous'),
    lambda sim, state: tasks.update_z_coord_rk4_vectorized(sim=sim, state=state,␣
↪wf=state.act_surf_wf,
                                                            z_coord=state.z_coord_branch,
                                                            output_name='z_coord_branch',
                                                            update_quantum_classical_
↪forces_bool=False),
    lambda sim, state: tasks.update_wf_db_eigs_vectorized(sim=sim, state=state, wf_
↪db=state.wf_db_branch,
                                                           eigvals=state.eigvals,␣
↪eigvecs=state.eigvecs,
                                                           adb_name='wf_adb_branch',␣
↪output_name='wf_db_branch'),
    lambda sim, state: tasks.update_h_quantum_vectorized(sim=sim, state=state, z_
```

(continues on next page)

(continued from previous page)

```
↪coord=state.z_coord_branch),
    lambda sim, state: tasks.diagonalize_matrix_vectorized(sim=sim, state=state,␣
↪matrix=state.h_quantum,
                                                              eigvals_name='eigvals',␣
↪eigvecs_name='eigvecs'),
    lambda sim, state: tasks.gauge_fix_eigs_vectorized(sim=sim, state=state,␣
↪eigvals=state.eigvals,
                                                          eigvecs=state.eigvecs,
                                                          eigvecs_previous=state.eigvecs_
↪previous,
                                                          output_eigvecs_name='eigvecs', z_
↪coord=state.z_coord_branch,
                                                          gauge_fixing=sim.algorithm.
↪parameters.gauge_fixing),
    lambda sim, state: tasks.update_active_surface_fssh(sim=sim, state=state),
    lambda sim, state: tasks.update_act_surf_wf_vectorized(sim=sim, state=state),
    lambda sim, state: tasks.update_timestep_index(sim=sim, state=state),
]
```

### Output Recipe

```
output_recipe = [
    lambda sim, state: tasks.update_dm_db_fssh_vectorized(sim=sim, state=state),
    lambda sim, state: tasks.update_quantum_energy_fssh_vectorized(sim=sim, state=state),
    lambda sim, state: tasks.update_classical_energy_fssh_vectorized(sim=sim,␣
↪state=state,
                                                                       z_coord=state.z_
↪coord_branch),
]
```

### Output Variables

The *FewestSwitchesSurfaceHopping* class computes the following output variables:

- *quantum_energy*: The quantum energy of the system.
- *classical_energy*: The classical energy of the system.
- *dm_db*: The density matrix database.

```
output_variables = [
    'quantum_energy',
    'classical_energy',
    'dm_db',
]
```

CHAPTER

TWO

# DEVELOPER GUIDE

A guide for developing or modifying models and algorithms in QC Lab.

## 2.1 Developer Guide

### 2.1.1 Model Development

This page will guide you in the construction of a new Model Class for use in QC Lab.

The model class describes a physical model which is described by a set of functions referred to as "ingredients". QC Lab is designed to accommodate a minimal model that consists of only a quantum-classical Hamiltonian. However, by incorporating additional ingredients, such as analytic gradients, the performance of QC Lab can be greatly improved. We will first describe the construction of a minimal model class, and then discuss how to incorporate additional ingredients.

**Table of Contents**

#### Minimal Model Class

A physical model in QC Lab is assumed to consist of a quantum-classical Hamiltonian of the form:

$$\hat{H}(\boldsymbol{z}) = \hat{H}_{\mathrm{q}} + \hat{H}_{\mathrm{q-c}}(\boldsymbol{z}) + H_{\mathrm{c}}(\boldsymbol{z})$$

where $\hat{H}_{\mathrm{q}}$ is the quantum Hamiltonian, $\hat{H}_{\mathrm{q-c}}$ is the quantum-classical coupling Hamiltonian, and $H_{\mathrm{c}}$ is the classical Hamiltonian. $\boldsymbol{z}$ is a complex-valued classical coordinate that describes the state of the classical degrees of freedom.

Before describing how the model ingredients should be structured in the model class, we will first describe the *__init__* method of the model class which is responsible for initializing the default parameters and any input parameters into the set of parameters needed for QC Lab to run.

```
from qclab import Model  # import the model class

# create a minimal spin-boson model subclass
class MinimalSpinBoson(Model):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, parameters=None):
        if parameters is None:
            parameters = {}
        self.default_parameters = {
            'temp': 1, 'V': 0.5, 'E': 0.5, 'A': 100, 'W': 0.1,
            'l_reorg': 0.02 / 4, 'boson_mass': 1
        }
        super().__init__(self.default_parameters, parameters)
```

In the above example, the *__init__* method takes an optional *parameters* dictionary which is added to the *default_parameters* dictionary by *super().__init__*. The *default_parameters* dictionary contains the default input parameters for the model. These parameters are independent from the internal parameters required by QC Lab to function and are instead drawn from the analytic formulation of the spin-boson model.

$$\hat{H}_{\mathrm{q}} = \begin{pmatrix} -E & V \\ V & E \end{pmatrix}$$

$$\hat{H}_{\mathrm{q-c}} = \sigma_z \sum_{\alpha}^{A} \frac{g_\alpha}{\sqrt{2mh_\alpha}} (z_\alpha^* + z_\alpha)$$

$$H_{\mathrm{c}} = \sum_{\alpha}^{A} \omega_\alpha z_\alpha^* z_\alpha$$

Here $\sigma_z$ is the Pauli-z matrix ($\sigma_z = |0\rangle\langle 0| - |1\rangle\langle 1|$), $g_\alpha$ is the coupling strength, $m$ is the boson mass, $h_\alpha$ is the z-coordinate parameter (which here we may take to correspond to the frequencies: $h_\alpha = \omega_\alpha$), and $A$ is the number of bosons. We sample the frequencies and coupling strengths from a Debye spectral density which is discretized to obtain

$$\omega_\alpha = \Omega \tan\left(\frac{\alpha - 1/2}{2A}\pi\right)$$

$$g_\alpha = \omega_\alpha \sqrt{\frac{2\lambda}{A}}$$

Where $\Omega$ is the characteristic frequency and $\lambda$ is the reorganization energy.

In a spin-boson model, the number of bosons $A$ can be quite large (e.g. 100). Rather than specifying every value of $\omega_\alpha$ and $g_\alpha$ in the input parameters, we can instead specify the characteristic frequency $\Omega$ and the reorganization energy $\lambda$. We can then use an internal function to generate the remaining parameters needed by the model and any parameters needed by QC Lab.

This is accomplished by specifying a function called *update_model_parameters* as a method of the model class.

```
def update_model_parameters(self):
    self.parameters.w = self.parameters.W * np.tan(((np.arange(self.parameters.A) + 1) -␣
↪0.5) * np.pi / (2 * self.parameters.A))
    self.parameters.g = self.parameters.w * np.sqrt(2 * self.parameters.l_reorg / self.
↪parameters.A)

    ### additional parameters required by QC Lab
    self.parameters.pq_weight = self.parameters.w
    self.parameters.num_classical_coordinates = self.parameters.A
    self.parameters.mass = np.ones(self.parameters.A) * self.parameters.boson_mass
```

Here, we obtain the frequencies and coupling strengths from the Debye spectral density. We then specify the parameters needed by QC Lab. Namely the complex-valued coordinate parameter $h_\alpha$ is denoted as "pq_weight", the number of classical coordinates is denoted as "num_classical_coordinates", and the mass is denoted as "mass".

Now you can check that the *update_model_parameters* is functioning properly by changing one of the input parameters (A for example) and then checking that the coupling strengths are updated appropriately:

```
model = MinimalSpinBoson()
model.parameters.A = 10
print('coupling strengths: ', model.parameters.g)  # should be a list of length 10
model.parameters.A = 5
print('coupling strengths: ', model.parameters.g)  # should be a list of length 5
```

Now we can add the minimal set of ingredients to the model class. The ingredients are the quantum Hamiltonian, the quantum-classical coupling Hamiltonian, and the classical Hamiltonian. The ingredients in a model class take a standard form which is required by QC Lab. Any argument (other than the model class itself) is passed as a keyword argument to the ingredient.

```
def h_q(self, **kwargs):
    E = self.parameters.E
    V = self.parameters.V
    return np.array([[-E, V], [V, E]], dtype=complex)

def h_qc(self, **kwargs):
    z_coord = kwargs['z_coord']
    g = self.parameters.g
    m = self.parameters.mass
    h = self.parameters.pq_weight
    h_qc = np.zeros((2, 2), dtype=complex)
    h_qc[0, 0] = np.sum((g * np.sqrt(1 / (2 * m * h))) * (z_coord + np.conj(z_coord)))
    h_qc[1, 1] = -h_qc[0, 0]
    return h_qc

def h_c(self, **kwargs):
    z_coord = kwargs['z_coord']
    w = self.parameters.w
    return np.sum(w * np.conj(z_coord) * z_coord)
```

Now you have a working model class which you can instantiate and use following the instructions in the *Quick Start Guide* section.

## Upgrading the Model Class

By default QC Lab assumes that a model's initial z coordinate is sampled from a Boltzmann distribution at temperature "temp" and that the classical Hamiltonian is harmonic with frequencies given by "pq_weight" and mass given by "mass". If this is not the case, the most prudent modification to make to the minimal model class is to specify how the classical coordinates are to be initialized.

This is accomplished by defining a method called *init_classical* which has the following form:

```
def init_classical(self, **kwargs):
    seed = kwargs['seed']
    np.random.seed(seed)  # initialize a random seed for any randomness in this function
    z_coord = # here we sample some distribution to obtain a complex array with length␣
↪sim.model.parameters.num_classical_coordinates
    return z_coord
```

The "seed" argument is passed to the method by QC Lab and is used to initialize a random seed for any randomness in the method. The method should return a complex array of length "sim.model.parameters.num_classical_coordinates".

While including *init_classical* ensures that the physical results of the model are correct, it does not change the performance of the minimal model.

The next recommended upgrade to the minimal model is to include analytic gradients for the classical and quantum-classical Hamiltonians with respect to the conjugate z coordinate. By default, QC Lab uses finite difference gradients which can be slow and inaccurate.

The gradient of the quantum-classical Hamiltonian is a complex-valued numpy array with the shape (num_classical_coordinates, num_state, num_states) where num_states is the dimension of the quantum Hilbert space. This structure appears naturally from the analytic form of the gradient. The $(\alpha, i, j)$-th element of this array is given by

$$\left\langle i \left| \frac{\partial \hat{H}_{\mathrm{q-c}}}{\partial z_{\alpha}^{*}} \right| j \right\rangle = (-1)^{i} \frac{g_{\alpha}}{\sqrt{2mh_{\alpha}}} \delta_{ij}.$$

When implemented this is:

```python
def dh_qc_dzc(self, **kwargs):
    g = self.parameters.g
    m = self.parameters.mass
    h = self.parameters.pq_weight
    dh_qc_dzc = np.zeros((self.parameters.A, 2, 2), dtype=complex)
    dh_qc_dzc[:, 0, 0] = g * np.sqrt(1 / (2 * m * h))
    dh_qc_dzc[:, 1, 1] = -dh_qc_dzc[:, 0, 0]
    return dh_qc_dzc
```

We can likewise implement a gradient for the classical Hamiltonian which is a complex-valued numpy array of shape (num_classical_coordinates). For the spin-boson model the classical Hamiltonian is harmonic and so has the form,

$$\frac{\partial H_{\mathrm{c}}}{\partial z_{\alpha}^{*}} = \omega_{\alpha} z_{\alpha}$$

which can be implemented as:

```python
def dh_c_dzc(self, **kwargs):
    w = self.parameters.w
    z_coord = kwargs['z_coord']
    dh_c_dzc = w * z_coord + 0.0j
    return dh_c_dzc
```

A more convenient way to incorporate these ingredients is to use the built-in set of ingredients available to QC Lab. For example, a model that has a classical Hamiltonian that is harmonic where the frequencies are given by “pq_weight” and the mass is given by “mass” can use the function `qclab.ingredients.harmonic_oscillator_dh_c_dzc` to generate the harmonic oscillator Hamiltonian and its gradient.

The next recommended upgrade is to include vectorized ingredients. Vectorized ingredients are ingredients that can be computed for a batch of trajectories simultaneously. If implemented making use of broadcasting and vectorized numpy functions, vectorized ingredients can greatly improve the performance of QC Lab.

As an example, let us consider a simulation where the z-coordinate comes as a vector with the shape (batch_size, num_classical_coordinates). A vectorized version of the classical Hamiltonian would accept the vectorized z-coordinate and return a vector of shape (batch_size) where each element is the energy of the classical coordinates in that batch. That general principle can be applied to any ingredient, where the vectorized form of an ingredient should output an array with shape (…, np.shape(output)) where np.shape(output) is the shape of the output of the non-vectorized ingredient and … are the additional dimensions of the z-coordinate (e.g. batch_size).

The vectorized form of the classical Hamiltonian for the spin-boson model is:

```python
def h_c_vectorized(model, **kwargs):
    z_coord = kwargs['z_coord']
    h_c = np.sum(model.parameters.pq_weight[..., :] * np.conjugate(z_coord) * z_coord,␣
↪axis=-1)
    return h_c
```

Importantly, the vectorized ingredient has the same name as the non-vectorized ingredient with "_vectorized" appended to the end.

Like dh_c_dzc, there are vectorized ingredients already built into QC Lab. For a full list of the available ingredients see the *Ingredients* section.

The vectorized quantum-classical interaction is implemented as:

```python
def h_qc_vectorized(self, **kwargs):
    z_coord = kwargs['z_coord']
    g = self.parameters.g
    m = self.parameters.mass
    h = self.parameters.pq_weight
    h_qc = np.zeros((*np.shape(z_coord)[:-1], 2, 2), dtype=complex)
    h_qc[..., 0, 0] = np.sum((g * np.sqrt(1 / (2 * m * h)))[..., :] * (z_coord + np.
↪conj(z_coord)), axis=-1)
    h_qc[..., 1, 1] = -h_qc[..., 0, 0]
    return h_qc
```

and its gradient is implemented as:

```python
def dh_qc_dzc_vectorized(self, **kwargs):
    g = self.parameters.g
    m = self.parameters.mass
    h = self.parameters.pq_weight
    dh_qc_dzc = np.zeros((*np.shape(kwargs['z_coord'])[:-1], self.parameters.A, 2, 2),␣
↪dtype=complex)
    dh_qc_dzc[..., :, 0, 0] = (g * np.sqrt(1 / (2 * m * h)))[..., :]
    dh_qc_dzc[..., :, 1, 1] = -dh_qc_dzc[..., :, 0, 0]
    return dh_qc_dzc
```

When vectorized ingredients are present, QC Lab no longer uses the non-vectorized ingredients. This means that the non-vectorized ingredients can be omitted from the model class. The fully optimized spin-boson model class is then:

```python
from qclab import Model, ingredients
class SpinBosonModel(Model):
    def __init__(self, parameters=None):
        if parameters is None:
            parameters = {}
        self.default_parameters = {
            'temp': 1, 'V': 0.5, 'E': 0.5, 'A': 100, 'W': 0.1,
            'l_reorg': 0.02 / 4, 'boson_mass': 1
        }
        super().__init__(self.default_parameters, parameters)

    def update_model_parameters(self):
        self.parameters.w = self.parameters.W * np.tan(((np.arange(self.parameters.A) +␣
↪1) - 0.5) * np.pi / (2 * self.parameters.A))
```

(continues on next page)

(continued from previous page)

```
        self.parameters.g = self.parameters.w * np.sqrt(2 * self.parameters.l_reorg /␣
↪self.parameters.A)

        ### additional parameters required by QC Lab
        self.parameters.pq_weight = self.parameters.w
        self.parameters.num_classical_coordinates = self.parameters.A
        self.parameters.mass = np.ones(self.parameters.A) * self.parameters.boson_mass

        ### additional parameters for built-in ingredients
        self.parameters.two_level_system_a = self.parameters.E  # Diagonal energy of␣
↪state 0
        self.parameters.two_level_system_b = -self.parameters.E  # Diagonal energy of␣
↪state 1
        self.parameters.two_level_system_c = self.parameters.V  # Real part of the off-
↪diagonal coupling
        self.parameters.two_level_system_d = 0  # Imaginary part of the off-diagonal␣
↪coupling

    def h_qc_vectorized(self, **kwargs):
        z_coord = kwargs['z_coord']
        g = self.parameters.g
        m = self.parameters.mass
        h = self.parameters.pq_weight
        h_qc = np.zeros((*np.shape(z_coord)[:-1], 2, 2), dtype=complex)
        h_qc[..., 0, 0] = np.sum((g * np.sqrt(1 / (2 * m * h)))[..., :] * (z_coord + np.
↪conj(z_coord)), axis=-1)
        h_qc[..., 1, 1] = -h_qc[..., 0, 0]
        return h_qc

    def dh_qc_dzc_vectorized(self, **kwargs):
        g = self.parameters.g
        m = self.parameters.mass
        h = self.parameters.pq_weight
        dh_qc_dzc = np.zeros((*np.shape(kwargs['z_coord'])[:-1], self.parameters.A, 2,␣
↪2), dtype=complex)
        dh_qc_dzc[..., :, 0, 0] = (g * np.sqrt(1 / (2 * m * h)))[..., :]
        dh_qc_dzc[..., :, 1, 1] = -dh_qc_dzc[..., :, 0, 0]
        return dh_qc_dzc

    # Assigning functions from ingredients module
    init_classical = ingredients.harmonic_oscillator_boltzmann_init_classical
    h_c_vectorized = ingredients.harmonic_oscillator_h_c_vectorized
    h_q_vectorized = ingredients.two_level_system_h_q_vectorized
    dh_c_dzc_vectorized = ingredients.harmonic_oscillator_dh_c_dzc_vectorized
```

### 2.1.2 Algorithm Development

CHAPTER

THREE

# SOFTWARE REFERENCE

A reference guide for QC Lab, documenting all tasks and ingredients available in the software.

## 3.1 Software Reference

### 3.1.1 Ingredients

Ingredients are are methods associated with model classes. A generic ingredient has the form:

```
def label_var_options(model, **kwargs):
    return var
```

Here, label is a descriptor for the ingredient, var is the name of a variable used by QC Lab, and options is used to specify additional classifications of the ingredient. Examples for var include the quantum Hamiltonian *h_q*, the classical Hamiltonian *h_c*, and the quantum-classical Hamiltonian *h_qc*. An ingredient that generates the quantum-classical Hamiltonian for the spin-boson model might look like this:

```
def spin_boson_h_qc(model, **kwargs):
    z_coord = kwargs['z_coord']
    g = model.parameters.g
    m = model.parameters.mass
    h = model.parameters.pq_weight
    h_qc = np.zeros((2, 2), dtype=complex)
    h_qc[0, 0] = np.sum((g * np.sqrt(1 / (2 * m * h))) * (z_coord + np.conj(z_coord)))
    h_qc[1, 1] = -h_qc[0, 0]
    return h_qc
```

The “options” part of the ingredient name is a string that is used to specify additional options for the ingredient. For example, QC Lab treates vectorized and non-vectorized ingredients differently. The options string for a vectorized ingredient is “vectorized”.

```
def spin_boson_h_qc_vectorized(model, **kwargs):
    z_coord = kwargs['z_coord']
    g = model.parameters.g
    m = model.parameters.mass
    h = model.parameters.pq_weight
    h_qc = np.zeros((*np.shape(z_coord)[:-1], 2, 2), dtype=complex)
    h_qc[..., 0, 0] = np.sum((g * np.sqrt(1 / (2 * m * h)))[..., :] * (z_coord + np.
↪conj(z_coord)), axis=-1)
```

(continues on next page)

(continued from previous page)

```
    h_qc[..., 1, 1] = -h_qc[..., 0, 0]
    return h_qc
```

When incorporated directly into the model class one should replace *model* with *self* and the name of the method should be *model.var*. See the model_class section for a detailed example.

Below we list all of the ingredients available in the current version of QC Lab and group ingredients by the attribute of the model that they pertain to.

## Current Ingredients

### Quantum Hamiltonian

This file contains ingredient functions for use in Model classes.

`qclab.ingredients.`**`nearest_neighbor_lattice_h_q_vectorized`**`(`*model*, ***kwargs*`)`

Calculate the vectorized quantum Hamiltonian for a nearest-neighbor lattice.

**Model Ingredient:**

- model.h_q_vectorized

**Model parameters:**

- model.parameters.nearest_neighbor_lattice_h_q_num_sites (int): Number of sites.
- model.parameters.nearest_neighbor_lattice_h_q_hopping_energy (complex): Hopping energy.
- model.parameters.nearest_neighbor_lattice_h_q_periodic_boundary (bool): Periodic boundary condition.

**Related functions:**

- `nearest_neighbor_lattice_h_q()`

`qclab.ingredients.`**`two_level_system_h_q_vectorized`**`(`*model*, ***kwargs*`)`

Calculate the vectorized quantum Hamiltonian for a two-level system.

**Model Ingredient:**

- model.h_q_vectorized

**Model parameters:**

- model.parameters.two_level_system_a (float): <0|H|0>
- model.parameters.two_level_system_b (float): <1|H|1>
- model.parameters.two_level_system_c (float): Re(<0|H|1>)
- model.parameters.two_level_system_d (float): Im(<0|H|1>)

**Related functions:**

- `two_level_system_h_q()`

### Quantum-Classical Interaction

Ingredients that generate quantum-classical interaction terms. This file contains ingredient functions for use in Model classes.

`qclab.ingredients.`**`holstein_lattice_dh_qc_dzc_vectorized`**`(`*model*, ***kwargs*`)`

**Calculate the vectorized derivative of the quantum-classical Hamiltonian with**
respect to the z-coordinates.

**Model Ingredient:**

- model.dh_qc_dzc_vectorized

**Required keyword arguments:**

- z_coord (np.ndarray): The z-coordinates.

**Model parameters:**

- model.parameters.holstein_lattice_h_qc_num_sites (int): Number of sites.
- model.parameters.holstein_lattice_h_qc_oscillator_frequency (float): Oscillator frequency.
- model.parameters.holstein_lattice_h_qc_dimensionless_coupling (float): Dimensionless coupling.

**Related functions:**

- `holstein_lattice_dh_qc_dzc()`

`qclab.ingredients.`**`holstein_lattice_h_qc_vectorized`**`(`*model*, ***kwargs*`)`

Calculate the vectorized quantum-classical Hamiltonian for a Holstein lattice.

**Model Ingredient:**

- model.h_qc_vectorized

**Required keyword arguments:**

- z_coord (np.ndarray): The z-coordinates.

**Model parameters:**

- model.parameters.holstein_lattice_h_qc_num_sites (int): Number of sites.
- model.parameters.holstein_lattice_h_qc_oscillator_frequency (float): Oscillator frequency.
- model.parameters.holstein_lattice_h_qc_dimensionless_coupling (float): Dimensionless coupling.

**Related functions:**

- `holstein_lattice_h_qc()`

### Classical Hamiltonian

Ingredients that generate classical Hamiltonians. This file contains ingredient functions for use in Model classes.

`qclab.ingredients.`**`harmonic_oscillator_dh_c_dzc_vectorized`**`(`*model*, ***kwargs*`)`

**Calculate the vectorized derivative of the classical Hamiltonian**
with respect to the z-coordinates.

**Model Ingredient:**

- model.dh_c_dzc_vectorized

**Required keyword arguments:**

- z_coord (np.ndarray): The z-coordinates.

**Model parameters:**

- model.parameters.pq_weight (np.ndarray): The weight parameters.

**Related functions:**

- `harmonic_oscillator_dh_c_dzc()`

`qclab.ingredients.`**`harmonic_oscillator_h_c_vectorized`**`(`*model*, ***kwargs*`)`

Harmonic oscillator classical Hamiltonian function.

**Model Ingredient:**

- model.h_c_vectorized

**Required keyword arguments:**

- z_coord (np.ndarray): The z-coordinates.

**Model parameters:**

- model.parameters.pq_weight (np.ndarray): The weight parameters.

**Related functions:**

- `harmonic_oscillator_h_c()`

`qclab.ingredients.`**`harmonic_oscillator_hop`**`(`*model*, ***kwargs*`)`

Perform a hopping operation for the harmonic oscillator.

**Model Ingredient:**

- model.hop

**Required keyword arguments:**

- z_coord (np.ndarray): The z-coordinates.
- delta_z_coord (np.ndarray): The change in z-coordinates.
- ev_diff (float): The energy difference.

**Model parameters:**

- model.parameters.pq_weight (np.ndarray): The weight parameters.

**Related functions:**

- *`harmonic_oscillator_boltzmann_init_classical()`*
- *`harmonic_oscillator_wigner_init_classical()`*

### Classical Initialization

Ingredients that initialize the complex-valued classical coordinates. This file contains ingredient functions for use in Model classes.

qclab.ingredients.**harmonic_oscillator_boltzmann_init_classical**(*model*, ***kwargs*)

Initialize classical coordinates according to Boltzmann statistics.

**Model Ingredient:**

- model.init_classical

**Required keyword arguments:**

- seed (int): The random seed.

**Model parameters:**

- model.parameters.temp (float): Temperature.
- model.parameters.mass (float): Mass.
- model.parameters.pq_weight (np.ndarray): The weight parameters.
- model.parameters.num_classical_coordinates (int): Number of classical coordinates.

**Related functions:**

- *harmonic_oscillator_wigner_init_classical()*

qclab.ingredients.**harmonic_oscillator_wigner_init_classical**(*model*, ***kwargs*)

**Initialize classical coordinates according to the Wigner distribution**
of the ground state of a harmonic oscillator.

**Model Ingredient:**

- model.init_classical

**Required model.parameters attributes:**

- pq_weight (float): The pq weight parameter.
- mass (float): The mass of the harmonic oscillator.
- temp (float): The temperature.
- num_classical_coordinates (int): The number of classical coordinates.

**Related functions:**

- *harmonic_oscillator_boltzmann_init_classical()*

### Deprecated and Non-Vectorized Ingredients

Please see the source code.

# PYTHON MODULE INDEX

# INDEX