QC Lab

Tempelaar Group

Mar 17, 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 qc_lab import Simulation # import simulation class
from qc_lab.models import SpinBoson # import model class
from qc_lab.algorithms import MeanField # import algorithm class
from qc_lab.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). Importantly, QC Lab expects that *num_trajs* is an integer multiple of *batch_size*. If not, it will use the lower integer multiple (which could be zero!).

```
# 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 = SpinBoson()
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 can be accessed as follows,

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

Running the Simulation

Finally, we run the simulation using the dynamics driver. Here, we are using the serial driver. QC Lab comes with several different types of parallel drivers which are discussed elsewhere.

```
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. In mean-field dynamics this is equivalent to the number of seeds.

```
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))
```

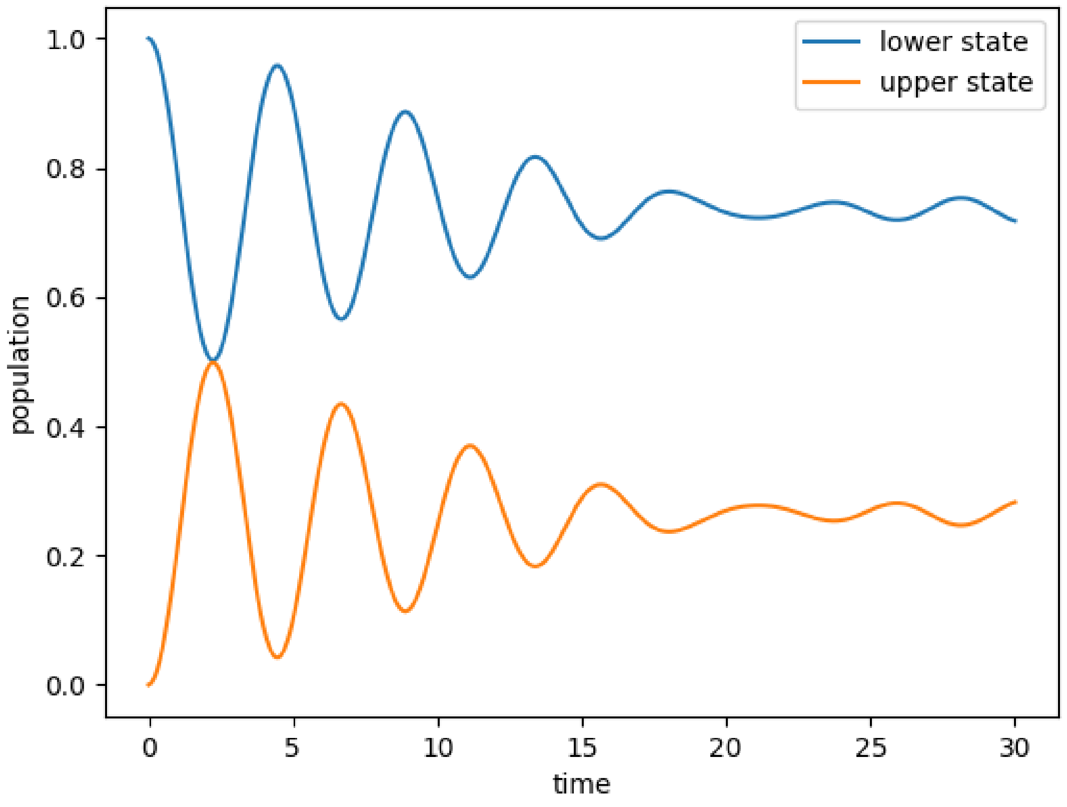
The time axis can be retrieved from the simulation object through its settings

```
time = sim.settings.tdat_output
```

Plotting Results

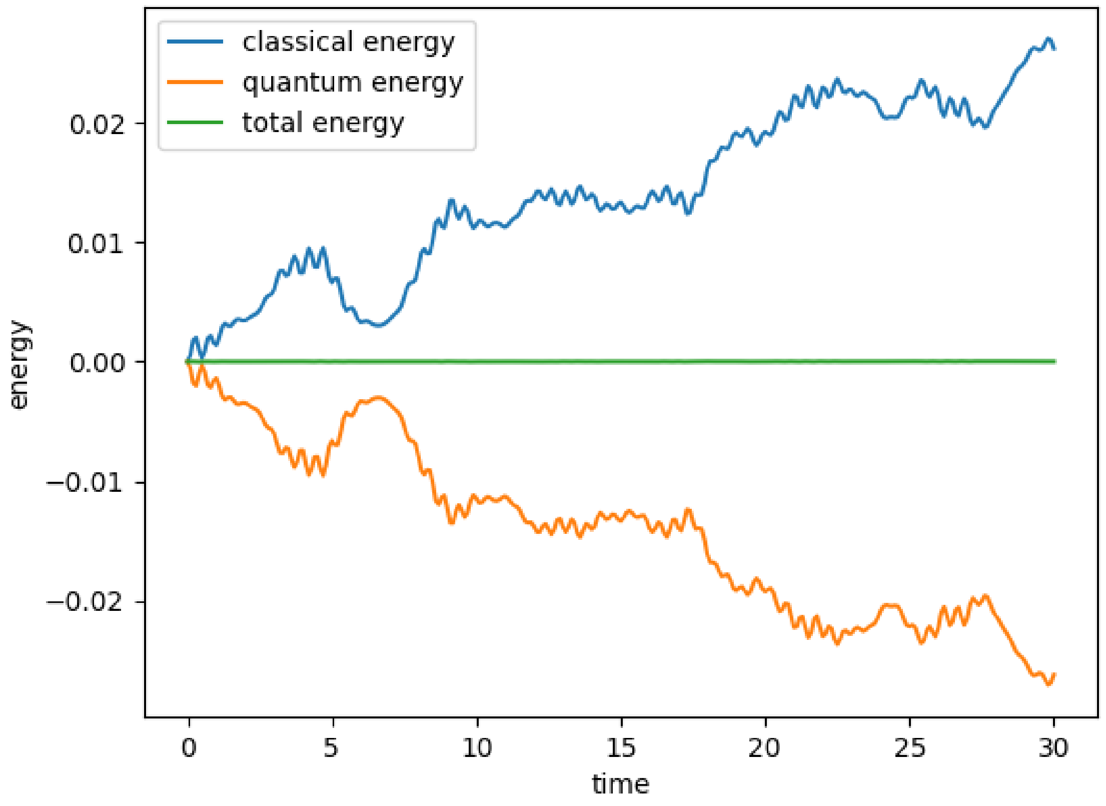
Finally, we can plot the results of the simulation like the population dynamics:

```
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()
```



Changing the Algorithm

If you want to do a surface hopping calculation rather than a mean-field one, QC Lab makes it very easy to do so. Simply import the relevant Algorithm class and set *sim.algorithm* to it and rerun the calculation:

```
from qc_lab.algorithms import FewestSwitchesSurfaceHopping

sim.algorithm = FewestSwitchesSurfaceHopping()

data = serial_driver(sim)
```

1.1.2 Dynamics Drivers

Serial Driver

Parallel Drivers

QC Lab implements parallelization using different parallel drivers. The multiprocessing driver is compatible with Jupyter notebooks and is useful for calculations on a single node. The MPI driver, however, can be used in clusters and is compatible with different schedulers like SLURM.

An important aspect of these drivers is how they interface with the simulation settings. In particular, when running a simulation with a parallel driver, each batch of trajectories is sent to its own task (parallel process). As a result, it is necessary that the total number of trajectories is an integer multiple of the number of tasks times the batch size.

Multiprocessing Driver

The *parallel_driver_multiprocessing* function in the *qclab.dynamics* module is used to run simulations in parallel using the *multiprocessing* library in Python. This driver is compatible with Jupyter notebooks and is useful for calculations on a single node.

Function Signature

```
qclab.dynamics.parallel_driver_multiprocessing(sim, seeds=None, data=None, num_
↪tasks=None)
```

Parameters

- **sim** (*Simulation*): The simulation object that contains the model, settings, and state.
- **seeds** (*array-like, optional*): An array of seed values for the random number generator. If not provided, seeds will be generated automatically.
- **data** (*Data, optional*): A Data object to store the results of the simulation. If not provided, a new Data object will be created.
- **num_tasks** (*int, optional*): The number of parallel tasks (processes) to use for the simulation. If not provided, the number of available CPU cores will be used.

Returns

- **data** (*Data*): A Data object containing the results of the simulation.

Example

Here is an example of how to use the *parallel_driver_multiprocessing* function to run a simulation in parallel assuming that the simulation object has been set up according to the quickstart guide.:

```
# Import the parallel driver
from qc_lab.dynamics import parallel_driver_multiprocessing

# Run the simulation using the parallel driver
data = parallel_driver_multiprocessing(sim, num_tasks=4)
```

Notes

- **The total number of trajectories must be an integer multiple of the number of tasks times the batch size. If not,**
  the driver will use the lower integer multiple (which could be zero!).
- This driver is suitable for use in Jupyter notebooks and single-node calculations.
- For cluster-based calculations, consider using the MPI driver.

References

- multiprocessing library

MPI Driver

The *parallel_driver_mpi* function in the *qclab.dynamics* module is used to run simulations in parallel using the *mpi4py* library. This driver is suitable for use in cluster environments and is compatible with different schedulers like SLURM. Unlike the multiprocessing driver, the MPI driver requires a script to be run using the *mpiexec* or *mpirun* command.

Function Signature

```
qclab.dynamics.parallel_driver_mpi(sim, seeds=None, data=None, num_tasks=None)
```

Parameters

- **sim** (*Simulation*): The simulation object that contains the model, settings, and state.
- **seeds** (*array-like, optional*): An array of seed values for the random number generator. If not provided, seeds will be generated automatically.
- **data** (*Data, optional*): A Data object to store the results of the simulation. If not provided, a new Data object will be created.
- **num_tasks** (*int, optional*): The number of parallel tasks (processes) to use for the simulation. If not provided, the number of available MPI processes will be used.

Returns

- **data** (*Data*): A Data object containing the results of the simulation.

Example

Here is an example of how to use the *parallel_driver_mpi* function to run a simulation in parallel. Suppose the following code is saved in a script called `mpi_example.py`and that the simulation object has been set up according to the quickstart guide.:

```python
# Import the parallel driver
from qc_lab.dynamics import parallel_driver_mpi
# Import the MPI module
from mpi4py import MPI

# initialize the sim object using the quickstart guide

# Run the simulation using the parallel driver
data = parallel_driver_mpi(sim, num_tasks=100)

# Determine the rank of the current process
rank = MPI.COMM_WORLD.Get_rank()
if rank == 0:
    # do something with the data only on the master process
    print(data)
```

The parallel execution can be started using the *mpiexec* or *mpirun* command where the number of tasks used in the execution should be the same as the one used in the call to *parallel_driver_mpi*. For example:

```
mpirun -n 100 python parallel_example.py
```

If using a scheduler like SLURM, the number of tasks can be specified in the job script. For example:

```bash
#!/bin/bash
#SBATCH -A # your allocation
#SBATCH -p # your partition
#SBATCH -N 2
#SBATCH --ntasks-per-node 50
#SBATCH --cpus-per-task 1
#SBATCH -t 01:00:00
#SBATCH --mem-per-cpu=1G

ulimit -c 0
ulimit -s unlimited

mpirun -n 100 python mpi_example.py
```

Notes

- The total number of trajectories must be an integer multiple of the number of tasks times the batch size.
- This driver is suitable for use in cluster environments and is compatible with different schedulers like SLURM.
- For single-node calculations, optionally consider using the multiprocessing driver.

References

- mpi4py library

1.1.3 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 Ω is the characteristic frequency and λ 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 (Ω)	Characteristic frequency	0.1
l_reorg (λ)	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*.

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

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

Here, g is the dimensionless electron-phonon coupling, ω 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 (ω)	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()

```

(continues on next page)

(continued from previous page)

```
# 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)
```

1.1.4 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:

```
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

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(sim=sim, state=state, seed=state.seed),
    lambda sim, state: tasks.update_h_quantum_vectorized(sim=sim, state=state, z=state.
↪z),
]
```

Update Recipe

```
update_recipe = [
    lambda sim, state: tasks.update_h_quantum_vectorized(sim=sim, state=state, z=state.
↪z),
    lambda sim, state: tasks.update_z_rk4_vectorized(sim=sim, state=state, z=state.z,
                                                      output_name='z', 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=state.z),
]
```

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

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.

```
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

```
initialization_recipe = [
    lambda sim, state: tasks.initialize_z(sim=sim, state=state, seed=state.seed),
    lambda sim, state: tasks.broadcast_var_to_branch_vectorized(sim=sim, state=state,␣
↪val=state.z,
                                                               name='z_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=state.z_
↪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,
                                                      output_eigvecs_name='eigvecs',␣
↪z=state.z_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'),
```

(continues on next page)

(continued from previous page)

```
    lambda sim, state: tasks.update_z_rk4_vectorized(sim=sim, state=state, wf=state.act_
↪surf_wf,
                                                          z=state.z_branch,
                                                          output_name='z_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=state.z_
↪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=state.z_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=state.z_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 as 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 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_{c} is the classical Hamiltonian. $\boldsymbol{z}$ is a complex-valued classical coordinate that defines 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 model constants and any input constants into the set of constants needed for QC Lab and all of the model ingredients to run.

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

# create a minimal spin-boson model subclass
class MinimalSpinBoson(Model):
    def __init__(self, constants=None):
        if constants is None:
            constants = {}
        self.default_constants = {
            '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_constants, constants)
```

In the above example, the *__init__* method takes an optional *constants* dictionary which is added to the *default_constants* dictionary by *super().__init__*. The *default_constants* dictionary contains the default input constants for the model. These constants are independent from the internal constants 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 σ_z is the Pauli-z matrix ($\sigma_z = |0\rangle\langle 0| - |1\rangle\langle 1|$), g_α is the coupling strength, m is the boson mass, h_α 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 Ω is the characteristic frequency and λ 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 ω_α and g_α in the input constants, we can instead specify the characteristic frequency Ω and the reorganization energy λ. We can then use an internal function to generate the remaining constants needed by the model and any constants needed by QC Lab.

Initialization functions

This is accomplished by specifying a list of function called *initialization_functions* as an attribute of the model class. These functions will be executed by the Model superclass when the model is instantiated or whenever a constant is changed. Because it is a list of functions, it is executed from start to finish, so the order of the functions can sometimes be important. The first function we will specify initializes the model constants and make use of the *get* method of the Constants object (*self.constants*) to obtain the input constants. We use the get method of the *default_constants* dictionary (*self.default_constants*) to obtain the default constants in the event that the input constant has not been specified.

Here, we initialize the constants needed by QC Lab which are the number of classical coordinates (*sim.model.constants.num_classical_coordinates*), the number of quantum states (*sim.model.constants.num_quantum_states*), the classical coordinate weight (*sim.model.constants.classical_coordinate_weight*), and the classical coordinate mass (*sim.model.constants.classical_coordinate_mass*). Because the classical Hamiltonian is a harmonic oscillator, we set the classical coordinate weight to the oscillator frequencies (*sim.model.constant.w*) even though these frequencies are not strictly speaking a constant needed by QC Lab (they would otherwise be specified in the initialization function for the classical Hamiltonian).

```python
def initialize_constants_model(self):
    num_bosons = self.constants.get("A", self.default_constants.get("A"))
    char_freq = self.constants.get("W", self.default_constants.get("W"))
    w = self.constants.get("w", self.default_constants.get("w"))
    boson_mass = self.constants.get(
        "boson_mass", self.default_constants.get("boson_mass")
    )
    self.constants.w = char_freq * np.tan(
        ((np.arange(num_bosons) + 1) - 0.5) * np.pi / (2 * num_bosons)
    )
    # The following constants are required by QC Lab
    self.constants.num_classical_coordinates = num_bosons
    self.constants.num_quantum_states = 2
    self.constants.classical_coordinate_weight = w
    self.constants.classical_coordinate_mass = boson_mass * np.ones(num_bosons)
```

Next we define a function which initializes the constants needed by the classical Hamiltonian, quantum Hamiltonian, and quantum-classical Hamiltonian. Be aware that the constants we define in the functions are dictated by the requirements of the ingredients (these are defined in the *Ingredients* section).

```python
def initialize_constants_h_c(self):
    """
    Initialize the constants for the classical Hamiltonian.
    """
    w = self.constants.get("w", self.default_constants.get("w"))
    self.constants.harmonic_oscillator_frequency = w


def initialize_constants_h_qc(self):
    """
    Initialize the constants for the quantum-classical coupling Hamiltonian.
    """
    num_bosons = self.constants.get("A", self.default_constants.get("A"))
    w = self.constants.get("w", self.default_constants.get("w"))
    l_reorg = self.constants.get("l_reorg", self.default_constants.get("l_reorg"))
    self.constants.g = w * np.sqrt(2 * l_reorg / num_bosons)

def initialize_constants_h_q(self):
    """
    Initialize the constants for the quantum Hamiltonian. None are required in this case.
    """
```

These are all placed into the *initialization_functions* list in the model class.

```
initialization_functions = [
    initialize_constants_model,
    initialize_constants_h_c,
    initialize_constants_h_qc,
    initialize_constants_h_q,
]
```

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

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

Ingredients

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.

A generic ingredients has as arguments the model class itself, the constants object containing time independent quantities (stored in sim.model.constants), and the parameters object which contain potentially time-dependent quantities (stored in sim.model.parameters). The ingredients can also take additional keyword arguments which are passed to the ingredient when it is called. The ingredients return the result of the calculation directly. Typically, users will never call ingredients as they are internal functions used by QC Lab to define the model.

As an example we will use the quantum Hamiltonian. Importantly, QC Lab is a vectorized code capable of calculating multiple quantum-classical trajectories simultaneously. As a result, the ingredients must also be vectorized, meaning that they accept as input quantities with an additional dimension corresponding to the number of trajectories (this is taken to be the first dimension as a convention). The quantum Hamiltonian is a 2x2 matrix and so the vectorized quantum Hamiltonian is a 3D array with shape (len(parameters.seed), 2, 2) where the number of trajectories is given by the number of seeds in the parameters object.

Rather than writing a vectorized ingredient (which will be discussed later) we can invoke a decorator (*ingredients.vectorize*) which will automatically vectorize the ingredient at the cost of some performance (it is strongly recommended to write vectorized ingredients as a first pass for performance optimization).

```
import qc_lab.ingredients as ingredients

@ingredients.vectorize_ingredient
def h_q(model, constants, parameters, **kwargs):
    """
    Calculates the quantum Hamiltonian
    """
    E = self.constants.E
    V = self.constants.V
    return np.array([[-E, V], [V, E]], dtype=complex)
```

The rest of the model ingredients can likewise be written:

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

@ingredients.vectorize_ingredient
def h_qc(self, constants, parameters, **kwargs):
    z_coord = kwargs['z_coord']
    g = self.constants.g
    m = self.constants.mass
    h = self.constants.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

@ingredients.vectorize_ingredient
def h_c(self, constants, parameters, **kwargs):
    z_coord = kwargs['z_coord']
    w = self.constants.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 Quickstart Guide!

Note

Please be aware that the performance is going to be significantly worse than what can be achieved by implementing the upgrades below.

The full minimal model looks like this:

```
class MinimalSpinBoson(Model):
    def __init__(self, constants=None):
        if constants is None:
            constants = {}
        self.default_constants = {
            '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_constants, constants)

    def initialize_constants_model(self):
        num_bosons = self.constants.get("A", self.default_constants.get("A"))
        char_freq = self.constants.get("W", self.default_constants.get("W"))
        w = self.constants.get("w", self.default_constants.get("w"))
        boson_mass = self.constants.get(
            "boson_mass", self.default_constants.get("boson_mass")
        )
        self.constants.w = char_freq * np.tan(
            ((np.arange(num_bosons) + 1) - 0.5) * np.pi / (2 * num_bosons)
```

(continues on next page)

(continued from previous page)

```
        )
        # The following constants are required by QC Lab.
        self.constants.num_classical_coordinates = num_bosons
        self.constants.num_quantum_states = 2
        self.constants.classical_coordinate_weight = w
        self.constants.classical_coordinate_mass = boson_mass * np.ones(num_bosons)

    def initialize_constants_h_c(self):
        """
        Initialize the constants for the classical Hamiltonian.
        """
        w = self.constants.get("w", self.default_constants.get("w"))
        self.constants.harmonic_oscillator_frequency = w


    def initialize_constants_h_qc(self):
        """
        Initialize the constants for the quantum-classical coupling Hamiltonian.
        """
        num_bosons = self.constants.get("A", self.default_constants.get("A"))
        w = self.constants.get("w", self.default_constants.get("w"))
        l_reorg = self.constants.get("l_reorg", self.default_constants.get("l_reorg"))
        self.constants.g = w * np.sqrt(2 * l_reorg / num_bosons)

    def initialize_constants_h_q(self):
        """
        Initialize the constants for the quantum Hamiltonian. None are required in this␣
↪case.
        """

    initialization_functions = [
        initialize_constants_model,
        initialize_constants_h_c,
        initialize_constants_h_qc,
        initialize_constants_h_q,
    ]

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

    @ingredients.vectorize_ingredient
    def h_qc(self, constants, parameters, **kwargs):
        z_coord = kwargs['z_coord']
        g = self.constants.g
        m = self.constants.classical_coordinate_mass
        h = self.constants.classical_coordinate_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)))
```

(continues on next page)

(continued from previous page)

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

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

Upgrading the Model Class

Vectorized Ingredients

The first upgrade we recommend 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.

Here we show vectorized versions of the ingredients used in the minimal model. Since they are vectorized, they do not need to use the *@ingredients.vectorize_ingredient* decorator. An important feature of vectorized ingredients is how they determine the number of trajectories being calculated. In ingredients that depend on the classical coordinate this is done by comparing the shape of the first index of the classical coordinate to the provided *batch_size* parameter. In others where the classical coordinate is not provided, the *batch_size* is compared to the number of seeds in the simulation.

```
def h_q(self, constants, parameters, **kwargs):
    if kwargs.get("batch_size") is not None:
        batch_size = kwargs.get("batch_size")
    else:
        batch_size = len(parameters.seed)
    E = self.constants.E
    V = self.constants.V
    h_q = np.zeros((batch_size, 2, 2), dtype=complex)
    h_q[:, 0, 0] = -E
    h_q[:, 1, 1] = E
    h_q[:, 0, 1] = V
    h_q[:, 1, 0] = V
    return h_q


def h_qc(self, constants, parameters, **kwargs):
    z = kwargs.get("z_coord")
    if kwargs.get("batch_size") is not None:
        batch_size = kwargs.get("batch_size")
        assert len(z) == batch_size
    else:
        batch_size = len(z)

    g = constants.g
    m = constants.classical_coordinate_mass
    h = constants.classical_coordinate_weight
    h_qc = np.zeros((batch_size, 2, 2), dtype=complex)
    h_qc[:, 0, 0] = np.sum(
```

(continues on next page)

(continued from previous page)

```
        g * np.sqrt(1 / (2 * m * h))[np.newaxis, :] * (z + np.conj(z)), axis=-1
    )
    h_qc[:, 1, 1] = -h_qc[:, 0, 0]
    return h_qc

def h_c(self, constants, parameters, **kwargs):
    z = kwargs.get("z_coord")
    if kwargs.get("batch_size") is not None:
        batch_size = kwargs.get("batch_size")
        assert len(z) == batch_size
    else:
        batch_size = len(z)

    h = constants.classical_coordinate_weight[np.newaxis, :]
    w = constants.harmonic_oscillator_frequency[np.newaxis, :]
    m = constants.classical_coordinate_mass[np.newaxis, :]
    q = np.sqrt(2 / (m * h)) * np.real(z)
    p = np.sqrt(2 * m * h) * np.imag(z)
    h_c = np.sum((1 / 2) * (((p**2) / m) + m * (w**2) * (q**2)), axis=-1)
    return h_c
```

Analytic Gradients

By Default, QC Lab calculates gradients numerically with finite differences. This can in many cases be avoided by providing ingredients that return the gradients based on analytic formulas. The gradient of the classical Hamiltonian in the spin-boson model is given by

$$\frac{\partial H_c}{\partial z^*_\alpha} = \frac{1}{2}\left(\frac{\omega^2_\alpha}{h_\alpha} + h_\alpha\right) z_\alpha + \frac{1}{2}\left(\frac{\omega^2_\alpha}{h_\alpha} - h_\alpha\right) z^*_\alpha$$

which can be implemented in a vectorized fashion as:

```
def dh_c_dzc(self, constants, parameters, **kwargs):
    z = kwargs.get("z_coord")
    if kwargs.get("batch_size") is not None:
        batch_size = kwargs.get("batch_size")
        assert len(z) == batch_size
    else:
        batch_size = len(z)
    h = constants.classical_coordinate_weight
    w = constants.harmonic_oscillator_frequency
    a = (1 / 4) * (
        ((w**2) / h) - h
    )
    b = (1 / 4) * (
        ((w**2) / h) + h
    )
    dh_c_dzc = 2 * b[..., :] * z + 2 *a[..., :] * np.conj(z)
    return dh_c_dzc
```

Likewise we can construct an ingredient to generate the gradient of the quantum-classical Hamiltonian with respect to the conjugate z coordinate. In many cases this requires the calculation of a sparse tensor and so QC Lab assumes that it is in terms of indices, nonzero elements, and a shape.

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

Which can be implemented as:

```python
def dh_qc_dzc(self, constants, parameters, **kwargs):
    z = kwargs["z"]
    # Determine how many trajectories are being calculated.
    if kwargs.get("batch_size") is not None:
        batch_size = kwargs.get("batch_size")
    else:
        batch_size = len(parameters.seed)
    # Determine if we need to update the matrix elements.
    recalculate = False
    if model.dh_qc_dzc_shape is not None:
        if model.dh_qc_dzc_shape[0] != batch_size:
            recalculate = True
    if (
        model.dh_qc_dzc_inds is None
        or model.dh_qc_dzc_mels is None
        or model.dh_qc_dzc_shape is None
        or recalculate
    ):
        return model.dh_qc_dzc_inds, model.dh_qc_dzc_mels, model.dh_qc_dzc_shape
    # If we need to update the matrix elements, do so.
    num_sites = constants.num_quantum_states
    w = constants.holstein_coupling_oscillator_frequency
    g = constants.holstein_coupling_dimensionless_coupling
    h = constants.classical_coordinate_weight
    dh_qc_dzc = np.zeros((batch_size, num_sites, num_sites, num_sites), dtype=complex)
    np.einsum("tiii->ti", dh_qc_dzc, optimize="greedy")[...] = (g * w * np.sqrt(w / h))[
        ..., :
    ] * (np.ones_like(z, dtype=complex))
    inds = np.where(dh_qc_dzc != 0)
    mels = dh_qc_dzc[inds]
    shape = np.shape(dh_qc_dzc)
    model.dh_qc_dzc_inds = inds
    model.dh_qc_dzc_mels = dh_qc_dzc[inds]
    model.dh_qc_dzc_shape = shape
    return inds, mels, shape
```

An important feature of the above implementation is that it checks if the gradient has already been calculated, this is convenient because the gradient is a constant and so does not need to be recalculated every time the ingredient is called. As a consequence, however, we need to initialize the gradient to None in the model class.

```python
def __init__(self, constants=None):
    # Include initialization of the model as done above.
    self.dh_qc_dzc_inds = None
    self.dh_qc_dzc_mels = None
    self.dh_qc_dzc_shape = None
```

Note that a flag can be included to prevent the RK4 solver in QC Lab from recalculating the quantum-classical forces (ie the expectation value of *dh_qc_dzc*): *sim.model.linear_h_qc = True*

Classical Initialization

By default QC Lab assumes that a model's initial z coordinate is sampled from a Boltzmann distribution at temperature "temp" and attempts to sample a Boltzmann distribution given the classical Hamiltonian. This is in practice making a number of assumptions, notably that all the z coordinates are uncoupled from one another in the classical Hamiltonian.

This is accomplished by defining an ingredient called *init_classical* which has the following form:

```python
def init_classical(model, constants, parameters, **kwargs):
del model, parameters
seed = kwargs.get("seed", None)
kbt = constants.temp
h = constants.classical_coordinate_weight
w = constants.harmonic_oscillator_frequency
m = constants.classical_coordinate_mass
out = np.zeros((len(seed), constants.num_classical_coordinates), dtype=complex)
for s, seed_value in enumerate(seed):
    np.random.seed(seed_value)
    # Calculate the standard deviations for q and p.
    std_q = np.sqrt(kbt / (m * (w**2)))
    std_p = np.sqrt(m * kbt)
    # Generate random q and p values.
    q = np.random.normal(
        loc=0, scale=std_q, size=constants.num_classical_coordinates
    )
    p = np.random.normal(
        loc=0, scale=std_p, size=constants.num_classical_coordinates
    )
    # Calculate the complex classical coordinate.
    z = np.sqrt(h * m / 2) * (q + 1.0j * (p / (h * m)))
    out[s] = z
return out
```

The "seed" argument is passed to the ingredient by QC Lab and is used to initialize any random numbers. The method should return a complex array of length "sim.model.constants.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 (α, 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
```

(continues on next page)

(continued from previous page)

```
    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,

$$\frac{\partial H_c}{\partial z^*_\alpha} = \omega_\alpha z_\alpha$$

which can be implemented as:

```
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:

```
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:

```
def h_qc_vectorized(self, **kwargs):
    z_coord = kwargs['z_coord']
    g = self.parameters.g
    m = self.parameters.mass
```

(continues on next page)

(continued from previous page)

```
    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:

```
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:

```
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))
        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␣
```

(continues on next page)

(continued from previous page)

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

`qc_lab.ingredients.`**`nearest_neighbor_lattice_h_q`**(*model*, *constants*, *parameters*, ***kwargs*)

Calculate the quantum Hamiltonian for a nearest-neighbor lattice.

`qc_lab.ingredients.`**`two_level_system_h_q`**(*model*, *constants*, *parameters*, ***kwargs*)

Calculate the quantum Hamiltonian for a two-level system.

Quantum-Classical Interaction

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

Classical Hamiltonian

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

`qc_lab.ingredients.`**`harmonic_oscillator_dh_c_dzc`**(*model*, *constants*, *parameters*, ***kwargs*)

Calculate the derivative of the classical harmonic oscillator Hamiltonian with respect to the z coordinate.

`qc_lab.ingredients.`**`harmonic_oscillator_h_c`**(*model*, *constants*, *parameters*, ***kwargs*)

Harmonic oscillator classical Hamiltonian function.

`qc_lab.ingredients.`**`harmonic_oscillator_hop`**(*model*, *constants*, *parameters*, ***kwargs*)

Perform a hopping operation for the harmonic oscillator.

Classical Initialization

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

`qc_lab.ingredients.`**`harmonic_oscillator_boltzmann_init_classical`**(*model*, *constants*, *parameters*, ***kwargs*)

Initialize classical coordinates according to Boltzmann statistics for the Harmonic oscillator.

`qc_lab.ingredients.`**`harmonic_oscillator_wigner_init_classical`**(*model*, *constants*, *parameters*, ***kwargs*)

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

Deprecated and Non-Vectorized Ingredients

Please see the source code.

PYTHON MODULE INDEX

INDEX