QC Lab

Tempelaar Team

Apr 03, 2025

CONTENTS

QC Lab is a Python package designed for implementing and executing quantum-classical (QC) dynamics simulations. It offers an environment for developing physical models and QC algorithms which 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. Users that implement their own models and algorithms will have the opportunity to contribute them to QC Lab to form a growing library of quantum-classical dynamics tools.

QC Lab is developed and maintained by the Tempelaar Team in the Chemistry Department of Northwestern University in Evanston, Illinois, USA.

CHAPTER ONE

CAPABILITIES

1.1 Dynamics Algorithms

The following algorithms are implemented making use of the complex-classical coordinate formalism established in [1].

- Mean-field (Ehrenfest) dynamics [2]
- Fewest-switches surface hopping (FSSH) dynamics [3]

1.2 Model Systems

- Spin-boson model [4]
- Holstein lattice model [5]
- Fenna-Matthews-Olson (FMO) complex [6, 7]

CHAPTER

TWO

INSTALLING QC_LAB

This alpha release of QC Lab can be installed from source by downloading the repository and executing.:

```
pip install -e ./
```

from inside its topmost directory (where the *setup.py* file is located).

CHAPTER

THREE

USER GUIDE

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

3.1 User Guide

3.1.1 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.

The code in this page is implemented in the notebook *quickstart.ipynb* which can be found in the *examples* directory of the repository.

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 from *qc_lab.Simulation*. Each object has a set of default settings which can be accessed by calling *sim.default_settings*. Passing a dictionary to the simulation object when instantiating it will override the default settings.

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

Alternatively, you can directly modify the simulation settings by assigning new values to the settings 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*).

Note

QC Lab expects that the total time of the simulation (*tmax*) is an integer multiple of the output timestep (*dt_output*), which must also be an integer multiple of the propagation timestep (*dt*).

```
# change settings to customize simulation
sim.settings.num_trajs = 200
sim.settings.batch_size = 50
sim.settings.tmax = 30
sim.settings.dt = 0.01
sim.settings.dt_output = 0.1
```

Instantiating Model Object

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

```
sim.model = SpinBoson()
print('default model constants: ', sim.model.default_constants)
# default model constants:  {'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 which likewise has a set of default settings.

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

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([1, 0], 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_dict.keys())
# calculated quantities: dict_keys(['seed', 'dm_db', 'classical_energy', 'quantum_energy'])
```

Each of the calculated quantities is normalized with respect to the number of trajectories (note that this might depend on the type of algorithm used) and can be accessed through the *data.data_dict* attribute. The normlaization factor for the data is kept in *data.data_dict["norm_factor"]*.

```
norm_factor = data.data_dict['norm_factor']
classical_energy = data.data_dict['classical_energy']
quantum_energy = data.data_dict['quantum_energy']
populations = np.real(np.einsum('tii->ti', data.data_dict['dm_db']))
```

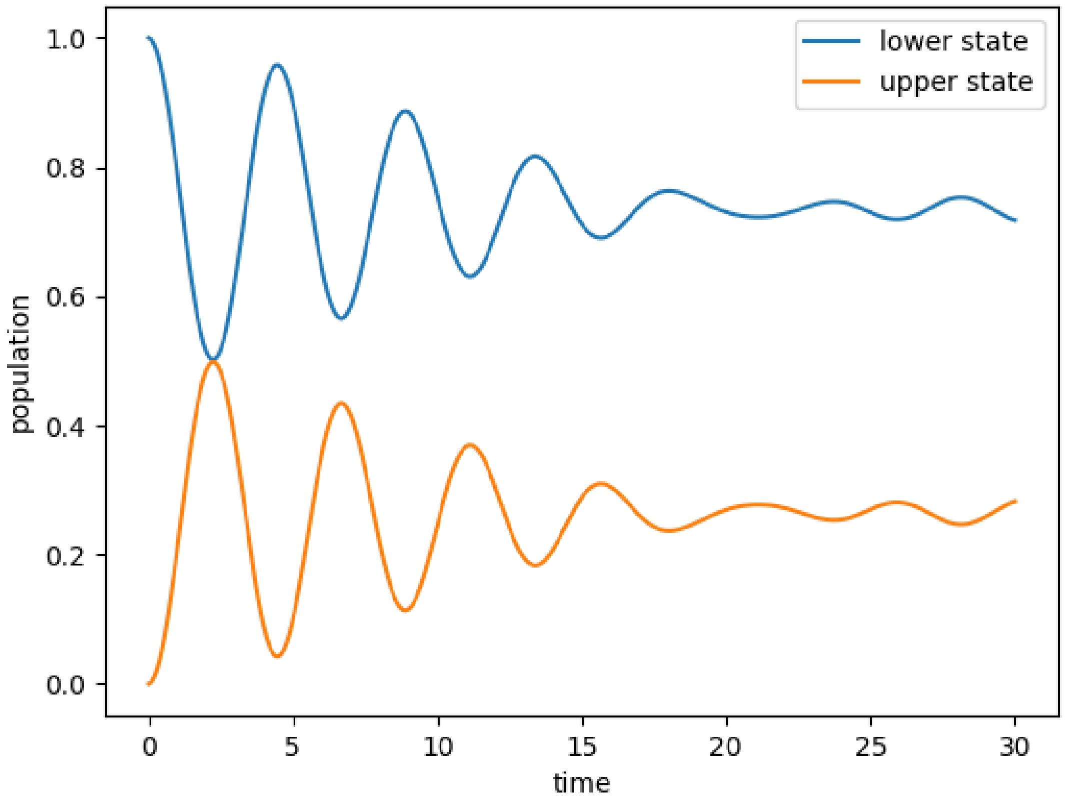
The time axis can be retrieved from the simulation object through its settings.

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

Plotting Results

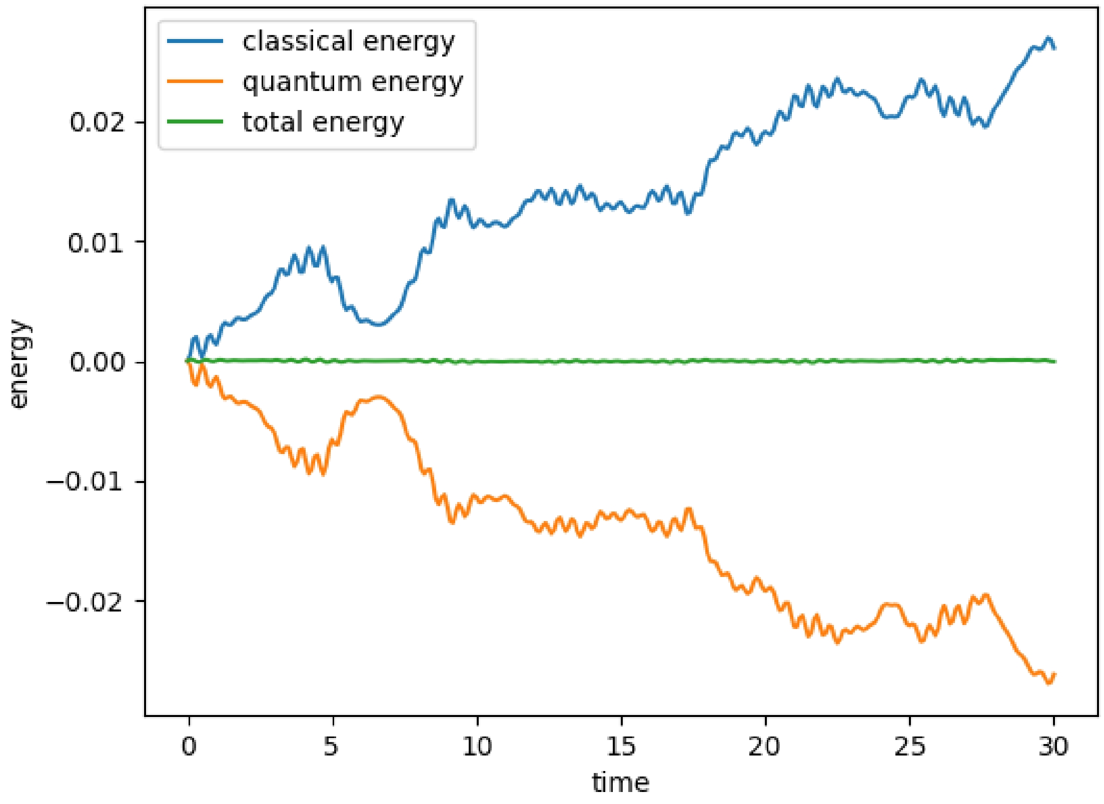
Finally, we can plot the results of the simulation like the population dynamics.

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

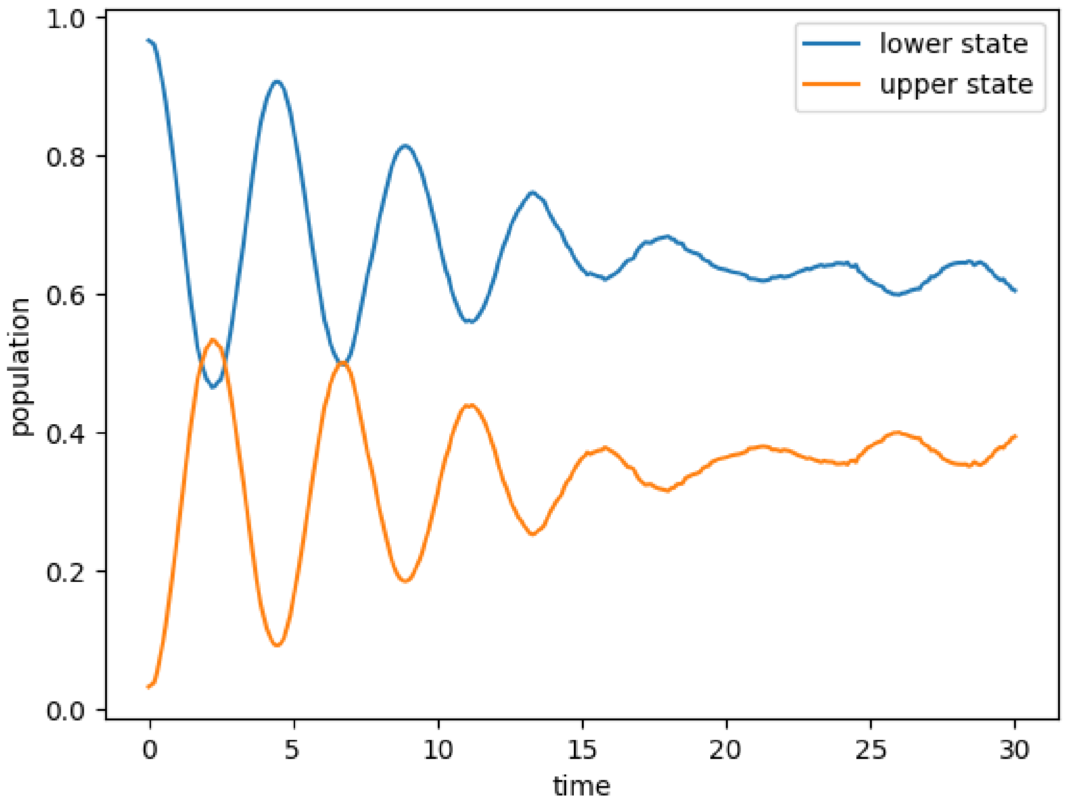
If you want to do a surface hopping calculation rather than a mean-field one, QC Lab makes it very easy to do so. Simply import the relevant Algorithm class and set *sim.algorithm* to it and rerun the calculation.

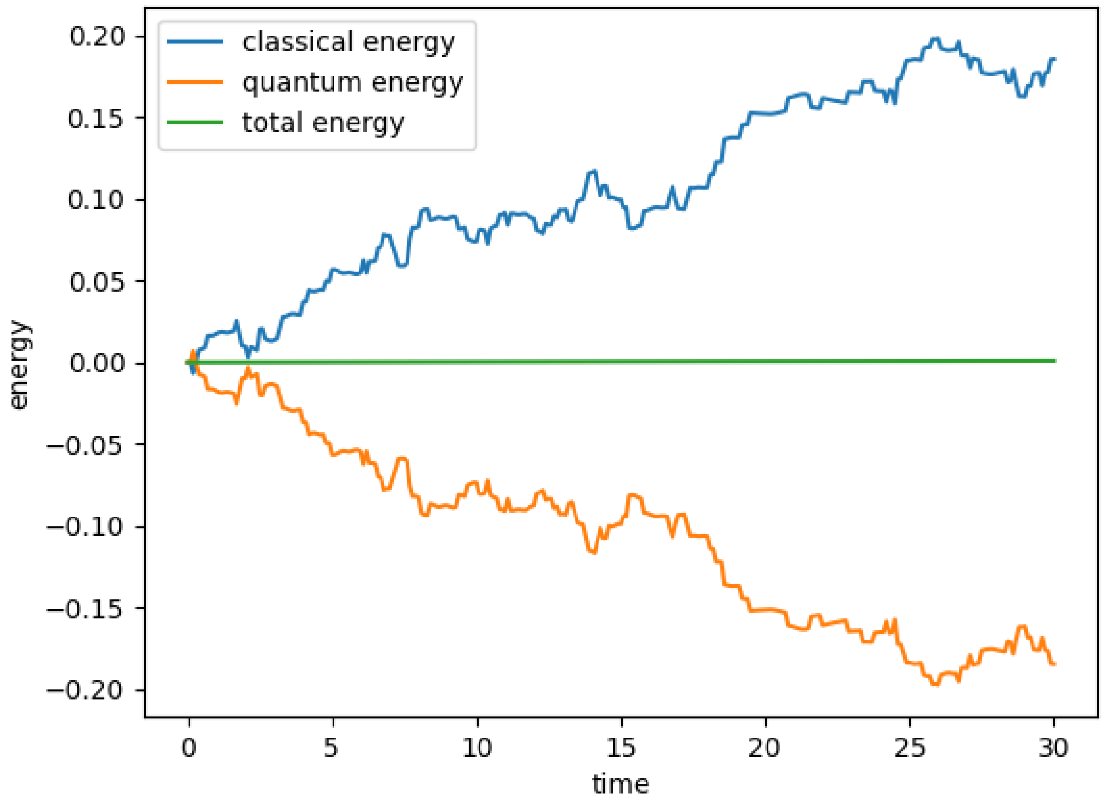
```
from qc_lab.algorithms import FewestSwitchesSurfaceHopping

sim.algorithm = FewestSwitchesSurfaceHopping()

data = serial_driver(sim)
```

The populations can be visualized in a similar way as before. Note that the simulation settings chosen here are solely for testing purposes. Publication quality simulations would require checking convergence of the number of trajectories and the timestep.


lower state
upper state
population
time
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25
30



Changing the Driver

You can likewise run the simulation using a parallel driver. Here we use the multiprocessing driver to split the trajectories over four tasks.

```
from qc_lab.dynamics import parallel_driver_multiprocessing

data = parallel_driver_multiprocessing(sim, num_tasks=4)
```

Units in QC Lab

QC Lab is written assuming all energies are in units of the thermal quantum ($k_\mathrm{B}T$). Units of time are then determined by assuming a value for the temperature defining the thermal quantum and calculating the equivalent timescales. For example, if we assume a standard temperature of $T = 298.15\,\mathrm{K}$ then the thermal quantum is $k_\mathrm{B}T = 25.7\,\mathrm{meV}$ and one unit of time is $\hbar/k_\mathrm{B}T = 25.6\,\mathrm{fs}$.

3.1.2 Dynamics Drivers

Drivers in QC Lab are functions that interface a Simulation object with the Dynamics core. Drivers are responsible for initializing the objects needed for the Dynamics core to operate and handle the assignment of random seeds and the grouping of simulations into batches.

Serial Driver

The *serial_driver* function in the *qc_lab.dynamics* module is used to run simulations in serial.

Function Signature

```
qc_lab.dynamics.serial_driver(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.

Returns

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

Example

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

```
# Import the serial driver
from qc_lab.dynamics import serial_driver

# Run the simulation using the parallel driver
data = serial_driver(sim)
```

Parallel Multiprocessing Driver

The *parallel_driver_multiprocessing* function in the *qc_lab.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

```
qc_lab.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

- 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 *qc_lab.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

```
qc_lab.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.:

```
# 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:

```
#!/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

- 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

3.1.3 Models

Models in QC Lab are classes that define the physical properties of the system, minimally through the quantum, classical, and quantum-classical Hamiltonians. Each model in QC Lab has a set of constants that can be accessed by the user when the model is instantiated.

Spin-Boson Model

We employ the same Hamiltonian and naming conventions as in Tempelaar & Reichman 2019.

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

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

$$\hat{H}_{\mathrm{q-c}} = \sigma_z \sum_{\alpha}^{A} g_\alpha q_\alpha$$

$$H_{\mathrm{c}} = \sum_{\alpha}^{A} \frac{p_\alpha^2}{2m} + \frac{1}{2} m\omega_\alpha^2 q_\alpha^2$$

where σ_z is the Pauli matrix, E is the diagonal energy, V is the off-diagonal coupling, and A is the number of bosons.

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.

$$P(\boldsymbol{p}, \boldsymbol{q}) \propto \exp\left(-\frac{H_{\mathrm{c}}(\boldsymbol{p}, \boldsymbol{q})}{k_{\mathrm{B}} T}\right)$$

Constants

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

Table 1: SpinBosonModel constants

Parameter (symbol)	Description	Default Value
kBT (kBT)	Thermal quantum	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

```
from qc_lab.models import SpinBoson
from qc_lab import Simulation
from qc_lab.algorithms import MeanField
from qc_lab.dynamics import serial_driver
import numpy as np

# instantiate a simulation
sim = Simulation()

# instantiate a model
sim.model = SpinBoson()

# instantiate an algorithm
sim.algorithm = MeanField()

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

# 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 and is described in detail in Krotz et al. 2021 .

The quantum Hamiltonian of the Holstein model is a nearest-neighbor tight-binding model

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

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

The quantum-classical Hamiltonian is the Holstein coupling with dimensionless electron-phonon coupling g and phonon frequency ω

$$\hat{H}_{\mathrm{q-c}} = g\sqrt{2m\omega^3}\sum_{i}^{N}\hat{c}_i^{\dagger}\hat{c}_i q_i$$

and the classical Hamiltonian is the harmonic oscillator

$$H_{\mathrm{c}} = \sum_{i}^{N}\frac{p_i^2}{2m} + \frac{1}{2}m\omega^2 q_i^2$$

with mass m.

The classical coordinates are sampled from a Boltzmann distribution.

$$P(\boldsymbol{p},\boldsymbol{q}) \propto \exp\left(-\frac{H_{\mathrm{c}}(\boldsymbol{p},\boldsymbol{q})}{k_{\mathrm{B}}T}\right)$$

Constants

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

Table 2: HolsteinLatticeModel constants

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
J (J)	Hopping energy	1
phonon_mass (m)	Phonon mass	1
periodic_boundary	Periodic boundary condition	*True*`

Example

```
from qc_lab.models import HolsteinLattice
from qc_lab import Simulation
from qc_lab.algorithms import MeanField
from qc_lab.dynamics import serial_driver
import numpy as np

# instantiate a simulation
sim = Simulation()

# instantiate a model
sim.model = HolsteinLattice()

# instantiate an algorithm
sim.algorithm = MeanField()

# define an initial diabatic wavefunction
sim.state.wf_db = np.zeros((sim.model.constants.num_quantum_states), dtype=complex)
sim.state.wf_db[0] = 1.0

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

Fenna-Matthews-Olson Model

The Fenna-Matthews-Olson (FMO) complex is a pigment-protein complex found in green sulfur bacteria. We implemented it in QC Lab as an 7 site model with Holstein-type coupling to local vibrational modes with couplings and frequencies sampled from a Debye spectral density according to Mulvihill et. al 2021.

$$\hat{H}_{\mathrm{q}} = \begin{pmatrix} 12410 & -87.7 & 5.5 & -5.9 & 6.7 & -13.7 & -9.9 \\ -87.7 & 12530 & 30.8 & 8.2 & 0.7 & 11.8 & 4.3 \\ 5.5 & 30.8 & 12210.0 & -53.5 & -2.2 & -9.6 & 6.0 \\ -5.9 & 8.2 & -53.5 & 12320 & -70.7 & -17.0 & -63.3 \\ 6.7 & 0.7 & -2.2 & -70.7 & 12480 & 81.1 & -1.3 \\ -13.7 & 11.8 & -9.6 & -17.0 & 81.1 & 12630 & 39.7 \\ -9.9 & 4.3 & 6.0 & -63.3 & -1.3 & 39.7 & 12440 \end{pmatrix}$$

where the matrix elements above are in units of wavenumbers. Note that the values below are in units of thermal quantum at 298.15K.

The quantum-classical and classical Hamiltonians are

$$\hat{H}_{\mathrm{q-c}} = \sum_{i}\sum_{j}^{A} \omega_j g_j c_i^{\dagger} c_i q_j^{(i)}$$

$$H_{\mathrm{c}} = \sum_{i}\sum_{j}^{A} \frac{p_j^{(i)2}}{2m} + \frac{1}{2} m\omega_j^2 q_j^{(i)2}$$

where $q_j^{(i)}$ is the j-th coordinate coupled to site i and $p_j^{(i)}$ is the corresponding momentum.

The couplings and frequencies are sampled from a Debye spectral density

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

$$g_j = \omega_j \sqrt{\frac{2\lambda}{A}}$$

where Ω is the characteristic frequency and λ is the reorganization energy.

The classical coordinates are sampled from a Boltzmann distribution.

$$P(\boldsymbol{p},\boldsymbol{q}) \propto \exp\left(-\frac{H_{\mathrm{c}}(\boldsymbol{p},\boldsymbol{q})}{k_{\mathrm{B}}T}\right)$$

Constants

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

Table 3: HolsteinLatticeModel constants

Parameter (symbol)	Description	Default Value
temp (T)	Temperature	1
mass (m)	Vibrational mass	1
A (A)	Number of bosons	200
W (Ω)	Characteristic frequency	$106.14\ \mathrm{cm}^{-1}$
l_reorg (λ)	Reorganization energy	$35\ \mathrm{cm}^{-1}$

Example

```
from qc_lab.models import FMOComplex
from qc_lab import Simulation
from qc_lab.algorithms import MeanField
from qc_lab.dynamics import serial_driver
import numpy as np

# instantiate a simulation
sim = Simulation()

# instantiate a model
sim.model = FMOComplex()
```

(continues on next page)

(continued from previous page)

```python

# instantiate an algorithm
sim.algorithm = MeanField()

# define an initial diabatic wavefunction
sim.state.wf_db = np.zeros((sim.model.constants.num_quantum_states), dtype=complex)
sim.state.wf_db[5] = 1.0

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

3.1.4 Algorithms

Algorithms in QC Lab are classes that can be paired with models to carry out quantum-classical dynamics simulations. Like models, each algorithm in QC Lab has a set of settings that allow users to control particular aspects of the algorithm.

Mean-Field Dynamics

The *qc_lab.algorithms.MeanField* class implements the mean-field (Ehrenfest) dynamics algorithm according to Tully 1998.

Settings

The mean-field algorithm has no default settings.

Initial State

The mean-field algorithm requires an initial diabatic wavefunction called *wf_db* which is a complex NumPy array with dimension *sim.model.constants.num_quantum_states*. For example:

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

Output Variables

The following table lists the default output variables for the *MeanField* class.

Table 4: *MeanField* Output Variables

Variable name	Description
classical_energy	Energy in the classical subsystem
quantum_energy	Energy in the quantum subsystem
dm_db	Diabatic density matrix

Example

The following example demonstrates how to run a mean-field simulation for a spin-boson model using all default settings.

```python
import numpy as np
from qc_lab import Simulation # import simulation class
```

(continues on next page)

(continued from previous page)

```
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

sim = Simulation()
sim.model = SpinBoson()
sim.algorithm = MeanField()
sim.state.wf_db= np.array([1, 0], dtype=complex)
data = serial_driver(sim)
```

Fewest-Switches Surface Hopping

The *qc_lab.algorithms.FewestSwitchesSurfaceHopping* class implements Tully's Fewest-Switches Surface Hopping (FSSH) dynamics algorithm according to Hammes-Schiffer 1994.

Settings

Table 5: *FewestSwitchesSurfaceHopping* settings

Setting name (type)	Description	Default value
fssh_deterministic (bool)	If *True* the algorithm uses a deterministic representation of the initial state by propagating all possible initial active surfaces. If *False*, it samples the initial active surface according to the adiabatic populations.	*False*
gauge_fixing (int=0,1,2)	The level of gauge fixing to employ on the eigenvectors at each timestep. (0: adjust only the sign, 1: adjust the sign and phase using the overlap with the previous timestep, 2: adjust the sign and phase by calculating the derivative couplings.)	0

Initial State

The FSSH algorithm requires an initial diabatic wavefunction called *wf_db* which is a complex NumPy array with dimension *sim.model.constants.num_quantum_states*. For example:

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

Output Variables

The following table lists the default output variables for the *FewestSwitchesSurfaceHopping* class.

Table 6: *FewestSwitchesSurfaceHopping* Output Variables

Variable name	Description
classical_energy	Energy in the classical subsystem
quantum_energy	Energy in the quantum subsystem
dm_db	Diabatic density matrix

Example

The following example demonstrates how to run a mean-field simulation for a spin-boson model using all default settings.

```python
import numpy as np
from qc_lab import Simulation # import simulation class
from qc_lab.models import SpinBoson # import model class
from qc_lab.algorithms import FewestSwitchesSurfaceHopping # import algorithm class
from qc_lab.dynamics import serial_driver # import dynamics driver

sim = Simulation()
sim.model = SpinBoson()
sim.algorithm = FewestSwitchesSurfaceHopping()
sim.state.wf_db= np.array([1, 0], dtype=complex)
data = serial_driver(sim)
```

3.1.5 Data Object

The Data object in QC Lab is used to store the results of simulations. Data objects come with a variety of methods that are intended as quality-of-life features to make using QC Lab easier. Here we review those methods.

Assuming we have a data object or have initialized one as:

```python
from qc_lab import Data
data = Data()
```

Methods

`data.`**`add_data`**`(`*new_data*`)`

Adds the data from new_data to the data object. Joins the seeds together and sums any data with the same keys.

Parameters
`new_data` – An input data object.

`data.`**`save_as_h5`**`(`*filename*`)`

Saves the data in the data object as an HDF5 file.

Parameters
`filename` – A string providing the name of the file to save the data to.

`data.`**`load_from_h5`**`(`*filename*`)`

Loads data into the data object from an HDF5 file.

Parameters
`filename` – A string providing the name of the file to load the data from.

Returns
The Data object with the loaded data.

Attributes

`data.`**`data_dic`**

A dictionary containing the data stored in the Data object. By default, this contains the seeds (*data.data_dict[“seed”]*) used in the simulation. In a new data object, this list is empty.

3.1.6 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 following the formalism developed in Miyazaki 2024

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

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 qc_lab 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': 0.005, '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.

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

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

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

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 complex-valued coordinate weight (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 functions 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"))
    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 = self.constants.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).

```
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.spin_boson_coupling = 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.spin_boson_coupling)  # should be a list␣
↪of length 10
model.constants.A = 5
print('coupling strengths: ', model.constants.spin_boson_coupling)  # 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 2×2 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).

```python
import qc_lab.ingredients as ingredients

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

The rest of the model ingredients can likewise be written:

```python
@ingredients.vectorize_ingredient
def h_qc(self, constants, parameters, **kwargs):
    z = kwargs['z']
    g = constants.spin_boson_coupling
    m = constants.classical_coordinate_mass
    h = 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 + np.conj(z)))
    h_qc[1, 1] = -h_qc[0, 0]
    return h_qc

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

Now you have a working model class which you can instantiate and use following the instructions in the Quick tart Guide!

Note

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

Prior to implementing upgrades, a minimal model will be subject to a number of defaults which are discussed in the *Default Behavior* section.

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': 0.005, '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"))
        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 = self.constants.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.spin_boson_coupling = 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␣
```

(continues on next page)

(continued from previous page)

```
↪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 = constants.E
        V = constants.V
        return np.array([[E, V], [V, -E]], dtype=complex)

    @ingredients.vectorize_ingredient
    def h_qc(self, constants, parameters, **kwargs):
        z = kwargs['z']
        g = constants.spin_boson_coupling
        m = constants.classical_coordinate_mass
        h = 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 + np.conj(z)))
        h_qc[1, 1] = -h_qc[0, 0]
        return h_qc

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

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 = constants.E
```

(continues on next page)

(continued from previous page)

```
    V = 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")
    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.spin_boson_coupling
    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(
        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")
    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

Derivatives of the Hamiltonian with respect to each classical coordinate (refered to here as gradients) are automatically calculated in QC Lab using a finite difference method. This can cause significant computational overhead and can 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

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

which can be implemented in a vectorized fashion as:

```
def dh_c_dzc(self, constants, parameters, **kwargs):
    z = kwargs.get("z")
    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 / 2) * (
        ((w**2) / h) - h
    )
    b = (1 / 2) * (
        ((w**2) / h) + h
    )
    dh_c_dzc = b[..., :] * z + 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:

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

    recalculate = False
    if self.dh_qc_dzc_shape is not None:
        if self.dh_qc_dzc_shape[0] != batch_size:
            recalculate = True

    if (
        self.dh_qc_dzc_inds is None
        or self.dh_qc_dzc_mels is None
        or self.dh_qc_dzc_shape is None
        or recalculate
    ):

        m = constants.classical_coordinate_mass
        g = constants.spin_boson_coupling
        h = constants.classical_coordinate_weight
        dh_qc_dzc = np.zeros((batch_size, constants.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]
```

(continues on next page)

(continued from previous page)

```
        inds = np.where(dh_qc_dzc != 0)
        mels = dh_qc_dzc[inds]
        shape = np.shape(dh_qc_dzc)
        self.dh_qc_dzc_inds = inds
        self.dh_qc_dzc_mels = dh_qc_dzc[inds]
        self.dh_qc_dzc_shape = shape
    else:
        inds = self.dh_qc_dzc_inds
        mels = self.dh_qc_dzc_mels
        shape = self.dh_qc_dzc_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.

```
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 (i.e., 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 with a thermal quantum given by *kBT* 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:

```
def init_classical(model, constants, parameters, **kwargs):
    del model, parameters
    seed = kwargs.get("seed", None)
    kBT = constants.kBT
    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
```

(continues on next page)

(continued from previous page)

```
        )
        # Calculate the complex-valued classical coordinate.
        z = np.sqrt(h * m / 2) * (q + 1.0j * (p / (h * m)))
        out[s] = z
    return out
```

The full code for the *UpgradedSpinBoson* model is:

```
class UpgradedSpinBoson(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
        }
        self.dh_qc_dzc_inds = None
        self.dh_qc_dzc_mels = None
        self.dh_qc_dzc_shape = None
        self.linear_h_qc = True
        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"))
        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 = self.constants.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"))
```

(continues on next page)

(continued from previous page)

```
        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,
    ]

    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 = constants.E
        V = 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")
        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.spin_boson_coupling
        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(
            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")
        if kwargs.get("batch_size") is not None:
            batch_size = kwargs.get("batch_size")
            assert len(z) == batch_size
        else:
```

(continues on next page)

(continued from previous page)

```
            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

    def dh_c_dzc(self, constants, parameters, **kwargs):
        z = kwargs.get("z")
        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 / 2) * (
            ((w**2) / h) - h
        )
        b = (1 / 2) * (
            ((w**2) / h) + h
        )
        dh_c_dzc = b[..., :] * z + a[..., :] * np.conj(z)
        return dh_c_dzc

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

        recalculate = False
        if self.dh_qc_dzc_shape is not None:
            if self.dh_qc_dzc_shape[0] != batch_size:
                recalculate = True

        if (
            self.dh_qc_dzc_inds is None
            or self.dh_qc_dzc_mels is None
            or self.dh_qc_dzc_shape is None
            or recalculate
        ):

            m = constants.classical_coordinate_mass
            g = constants.spin_boson_coupling
            h = constants.classical_coordinate_weight
            dh_qc_dzc = np.zeros((batch_size, constants.A, 2, 2), dtype=complex)
```

(continues on next page)

(continued from previous page)

```
            dh_qc_dzc[:, :, 0, 0] = (g * np.sqrt(1 / (2 * m * h)))[..., :]
            dh_qc_dzc[:, :, 1, 1] = -dh_qc_dzc[..., :, 0, 0]
            inds = np.where(dh_qc_dzc != 0)
            mels = dh_qc_dzc[inds]
            shape = np.shape(dh_qc_dzc)
            self.dh_qc_dzc_inds = inds
            self.dh_qc_dzc_mels = dh_qc_dzc[inds]
            self.dh_qc_dzc_shape = shape
        else:
            inds = self.dh_qc_dzc_inds
            mels = self.dh_qc_dzc_mels
            shape = self.dh_qc_dzc_shape
        return inds, mels, shape

    def init_classical(model, constants, parameters, **kwargs):
        del model, parameters
        seed = kwargs.get("seed", None)
        kBT = constants.kBT
        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-valued classical coordinate.
            z = np.sqrt(h * m / 2) * (q + 1.0j * (p / (h * m)))
            out[s] = z
        return out
```

Note

The upgraded model still relies on numerical hopping for FSSH simulations. Below we will see how to use the analytic hopping ingredient for the harmonic oscillator.

Using Built-in Ingredients

QC Lab comes with a number of built-in ingredients that can be used to construct a model rather than writing ingredients from scratch like above. These ingredients can be imported from *qc_lab.ingredients* and are documented in the *Ingredients* section.

For the present example, we can avoid writing our own optimized ingredients and simply add the available built-in ingredients to the model.

First let's load the quantum Hamiltonian as the built-in two-level system Hamiltonian:

```
import qc_lab.ingredients as ingredients
model = MinimalSpinBoson

model.h_q = ingredients.two_level_system_h_q
```

We will also need a new initialization function to interface with the constants used by the built-in ingredient:

```
def initialize_constants_h_q(model):
    """
    Initialize the constants for the quantum Hamiltonian.
    """
    model.constants.two_level_system_a = model.constants.get(
        "E", model.default_constants.get("E")
    )
    model.constants.two_level_system_b = -model.constants.get(
        "E", model.default_constants.get("E")
    )
    model.constants.two_level_system_c = model.constants.get(
        "V", model.default_constants.get("V")
    )
    model.constants.two_level_system_d = 0
```

Let's then add the ingredient and initialization function to the model class:

```
model.h_q = ingredients.two_level_system_h_q
model.initialize_constants_h_q = initialize_constants_h_q
# also update the list of initialization functions
model.swap_initialization_function(model, 'initialize_constants_h_q', initialize_
↪constants_h_q)
```

here, we used the *swap_initialization_function* method to update the list of initialization functions. This is a convenience function that will swap out the initialization function with a given name. In this case, we are swapping out the *initialize_constants_h_q* function with the new one we just defined.

Next we can load the classical Hamiltonian as the built-in harmonic oscillator Hamiltonian and update the initialization function like before:

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

model.h_c = ingredients.harmonic_oscillator_h_c
model.initialize_constants_h_c = initialize_constants_h_c
model.swap_initialization_function(model, 'initialize_constants_h_c', initialize_
↪constants_h_c)
```

We can also load analytic gradients for the classical Hamiltonian (which relies on the same constants has the classical Hamiltonian).

```
model.dh_c_dzc = ingredients.harmonic_oscillator_dh_c_dzc
```

Next we can load the quantum-classical Hamiltonian and its gradient. We will use the built-in ingredient *diagonal_linear_h_qc* which is a generic quantum-classical Hamiltonian that linearly couples classical coordinates to the diagonal of the Hamiltonian. As a result we must specify a set of couplings to ensure that each coordinate is coupled to the correct entry of the diagonal.

Then we can load in the built-in classical initialization ingredient which samples the Boltzmann distribution for the harmonic oscillator Hamiltonian.

```
model.init_classical = ingredients.harmonic_oscillator_init_classical
```

Next, we can load an ingredient that executes the hopping procedure of the FSSH algorithm according to a harmonic oscillator. This will improve the performance of the FSSH algorithm.

```
model.hop_function = ingredients.harmonic_oscillator_boltzmann_init_classical
```

Lastly, we can add a flag to the model class that enables the RK4 solver in QC Lab to avoid recalculating gradients of the quantum-classical Hamiltonian (which is a constant if the quantum-classical Hamiltonian is linear in z).

```
model.linear_h_qc = True
```

The resulting model class is now fully upgraded and can be used to simulate the spin-boson model with significantly improved performance.

The full code for upgrading the *MinimalSpinBoson* using built-in ingredients is:

```
model = MinimalSpinBoson

model.h_q = ingredients.two_level_system_h_q

def initialize_constants_h_q(model):
    """
    Initialize the constants for the quantum Hamiltonian.
    """
    model.constants.two_level_system_a = model.constants.get(
        "E", model.default_constants.get("E")
    )
    model.constants.two_level_system_b = -model.constants.get(
        "E", model.default_constants.get("E")
    )
    model.constants.two_level_system_c = model.constants.get(
        "V", model.default_constants.get("V")
    )
    model.constants.two_level_system_d = 0

model.initialize_constants_h_q = initialize_constants_h_q
# also update the list of initialization functions
model.swap_initialization_function(model, 'initialize_constants_h_q', initialize_
↪constants_h_q)

def initialize_constants_h_c(model):
    """
    Initialize the constants for the classical Hamiltonian.
```

(continues on next page)

(continued from previous page)

```
    """
    w = model.constants.get("w", model.default_constants.get("w"))
    model.constants.harmonic_oscillator_frequency = w

model.h_c = ingredients.harmonic_oscillator_h_c
model.initialize_constants_h_c = initialize_constants_h_c
model.swap_initialization_function(model, 'initialize_constants_h_c', initialize_
↪constants_h_c)

model.dh_c_dzc = ingredients.harmonic_oscillator_dh_c_dzc

def initialize_constants_h_qc(model):
    """
    Initialize the constants for the quantum-classical coupling Hamiltonian.
    """
    num_bosons = model.constants.get("A", model.default_constants.get("A"))
    l_reorg = model.constants.get("l_reorg", model.default_constants.get("l_reorg"))
    m = model.constants.get("boson_mass", model.default_constants.get("boson_mass"))
    h = (
        model.constants.classical_coordinate_weight
    )
    w = model.constants.w
    model.constants.diagonal_linear_coupling = np.zeros((2, num_bosons))
    model.constants.diagonal_linear_coupling[0] = (
        w * np.sqrt(2 * l_reorg / num_bosons) * (1 / np.sqrt(2 * m * h))
    )
    model.constants.diagonal_linear_coupling[1] = (
        -w * np.sqrt(2 * l_reorg / num_bosons) * (1 / np.sqrt(2 * m * h))
    )

model.h_qc = ingredients.diagonal_linear_h_qc
model.dh_qc_dzc = ingredients.diagonal_linear_dh_qc_dzc
model.dh_qc_dzc_inds = None
model.dh_qc_dzc_mels = None
model.dh_qc_dzc_shape = None
model.swap_initialization_function(model, 'initialize_constants_h_qc', initialize_
↪constants_h_qc)

model.init_classical = ingredients.harmonic_oscillator_boltzmann_init_classical

model.hop_function = ingredients.harmonic_oscillator_hop_function

model.linear_h_qc = True
```

3.1.7 Default Behavior

Default Simulation Settings

By default QC Lab uses the following settings in the simulation object. These settings can be adjusted by changing the values in the *sim* object.

```
sim = Simulation()
sim.settings.var = val # Can change the value of a setting like this

# or by passing the setting directly to the simulation object.
sim = Simulation({'var': val})
```

Table 7: Default Simulation Settings

Variable	Description	Default Value
num_trajs	The total number of trajectories to run.	10
batch_size	The (maximum) number of trajectories to run simultaneously.	1
tmax	The total time of each trajectory.	10
dt	The timestep used for executing the update recipe (the dynamics propagation).	0.01
dt_output	The timestep used for executing the output recipe (the calculation of observables).	0.1

Note

QC Lab expects that the total time of the simulation is an integer multiple of the output timestep *dt_output*, which must also be an integer multiple of the propagation timestep *dt*.

Default Model Attributes

For minimal models where only the Hamiltonian of the system is defined in the Model class, QC Lab employs numerical methods to carry out particular steps in the dynamics algorithms. This page describes those default actions and also the constants that can be used to manipulate them. Because they are formally treated as model ingredients they have the same ingredient format discussed in the model development guide.

All of the constants below can be set by adjusting their value in the *model.constants* object, for example:

```
sim.model.constants.default_value = # fix the value of the constant 'default_value'
```

Initialization of classical coordinates

`sim.model.`**`init_classical`**(*model*, *constants*, *parameters*, *seed=seed*)

Parameters

seed (*np.ndarray((batch_size), dtype=int)*) – List of seeds used to initialize random numbers.

Returns

Initial complex-valued classical coordinate.

Return type

np.ndarray((batch_size, sim.model.constants.num_classical_coordinates), dtype=complex)

By default, QC Lab uses a Markov-chain Monte Carlo implementation of the Metropolis-Hastings algorithm to sample a Boltzmann distribution corresponding to *sim.model.h_c* at the thermal quantum *sim.model.constants.kBT*. We encourage caution and further validation before using it on arbitrary classical potentials as fine-tuning of the algorithm parameters may be required to obtain reliable results.

The implementation utilizes a single random walker in *sim.model.constants.num_classical_coordinates* dimensions or *sim.model.constants.num_classical_coordinates* walkers each in one dimension (depending on if *mcmc_h_c_separable==True*) and evolves the walkers from the initial point *mcmc_init_z* by sampling a Gaussian distribution with a standard deviation *mcmc_std* for *mcmc_burn_in_size* steps. It then evolves the walkers another *mcmc_sample_size* steps to collect a distribution of initial coordinates from which the required number of initial conditions are drawn uniformly. At a minimum, one should ensure that *mcmc_sample_size* is large enough to ensure a full exploration of the phase-space.

Variable name	Description	Default value
mcmc_burn_in_size	Number of burn-in steps.	10000
mcmc_sample_size	Number of samples to collect from which initial conditions are drawn. To ensure a full exploration of the phase-space this should be as large as practical.	100000
mcmc_h_c_separable	A boolean indicating if the classical Hamiltonian is separable into independent terms for each coordinate. If True each coordinate will be independently sampled improving the performance of the algorithm. If False the sampling will occur in the full dimensional space.	True
mcmc_init_z	The initial coordinate that the random walker is initialized at.	A point in the interval (0,1) for both real and imaginary parts in each coordinate. (This is deterministically chosen for reproducability).
mcmc_std	The standard deviation of the Gaussian used to generate the random walk.	1

Classical Hamiltonian gradients

`sim.model.`**`dh_c_dzc`**(*model*, *constants*, *parameters*, *z=z*)

Parameters
: **z** (*np.ndarray((batch_size, sim.model.constants.num_classical_coordinates), dtype=complex)*) – complex-valued classical coordinate.

Returns
: Gradient of the classical Hamiltonian.

Return type
: np.ndarray((batch_size, sim.model.constants.num_classical_coordinates), dtype=complex)

QC Lab utilizes a finite difference method to calculate the gradient of the classical Hamiltonian.

Variable name	Description	Default value
dh_qc_dzc_finite_differences_delta	Finite difference that each coordinate is varied by.	1e-6

Quantum-classical Hamiltonian gradients

`sim.model.`**`dh_c_dzc`**(*model*, *constants*, *parameters*, *z=z*)

Parameters
: **z** (*np.ndarray((batch_size, sim.model.constants.num_classical_coordinates), dtype=complex)*) – complex-valued classical coordinate.

Returns
Indices of nonzero values

Return type
np.ndarray((# of nonzero values, 4), dtype=int)

Returns
Values

Return type
np.ndarray((# of nonzero values), dtype=complex)

Returns
Shape of dense gradient: (batch_size, sim.model.constants.num_classical_coordinates, sim.model.constants.num_quantum_states, sim.model.constants.num_quantum_states)

Return type
Tuple

QC Lab utilizes a finite difference method to calculate the gradient of the quantum-classical Hamiltonian. Unlike that of the classical Hamiltonian, however, the output is in a sparse format.

Variable name	Description	Default value
dh_qc_dzc_finite_differences_delta	finite difference that each coordinate is varied by.	1e-6

Surface Hopping Switching Algorithm

`sim.model.`**`hop_function`**(*model*, *constants*, *parameters*, *z=z*, *delta_z=delta_z*, *ev_diff=ev_diff*)

Parameters

- **z** (*np.ndarray(sim.model.constants.num_classical_coordinates, dtype=complex)*) – Complex-valued classical coordinate (in a single trajectory).
- **delta_z** (*np.ndarray(sim.model.constants.num_classical_coordinates, dtype=complex)*) – Rescaling direction.
- **ev_diff** (*float*) – Energy difference between final and initial surface (final - initial).

Returns
Rescaled coordinate.

Return type
np.ndarray(sim.model.constants.num_classical_coordinates, dtype=complex)

Returns
True or False depending on if a hop happened.

Return type
Bool.

QC Lab implements a numerical method to find the scalar factor (gamma) required to rescale classical coordinates in the surface hopping algorithm. It works by constructing a uniform grid with *numerical_fssh_hop_num_points* points from negative to positive and determines the point at which energy is conserved the closest. It then recenters the grid at that point and reduces the range by 0.5 and once again searches for the point at which energy is conserved the closest. It repeats that step for *numerical_fssh_hop_max_iter* iterations or until the energy difference is less than *numerical_fssh_hop_threshold*. If the energy it reaches is less than the threshold then the hop is accepted, if it is greater then the hop is rejected.

Variable name	Description	Default value
numerical_fssh_hop_gamma_range	Interval from minus to positive over which gamma is initially sampled.	5
numerical_fssh_hop_num_points	The number of points on the grid used to sample gamma.	10
numerical_fssh_hop_threshold	The threshold used to determine if a hop is conserving energy at a given gamma.	1e-6
numerical_fssh_hop_max_iter	The maximum number of iterations before a search for gamma is halted.	20

3.1.8 Algorithm Development

In this guide, we will discuss how to make in-place modifications to Algorithms. Algorithm development is a more advanced topic and requires a good understanding of the underlying steps of the algorithm so we will not go into detail (in this guide) about how to develop a new algorithm from scratch. Instead, we will focus on making changes to existing algorithms.

Before we proceed, let's discuss the structure of an algorithm in QC Lab. An algorithm in QC Lab is a Python class that inherits from the *Algorithm* class in the *qc_lab.algorithm* module. The built-in algorithms are found in the *qc_lab.algorithms* module and presently contain *qc_lab.algorithms.MeanField* and *qc_lab.algorithms.FewestSwitchesSurfaceHopping*.

We can start by importing the *MeanField* algorithm from the *qc_lab.algorithms* module:

```python
from qc_lab.algorithms import MeanField
```

Each algorithm consists of three lists of functions which are referred to as "recipes", the functions themselves are referred to as "tasks".

```python
# the initialization recipe
print(MeanField.initialization_recipe)

# the update recipe
print(MeanField.update_recipe)

# the output recipe
print(MeanField.output_recipe)
```

As the name implies, the *initialization_recipe* initializes all the variables required for the algorithm. The *update_recipe* updates the variables at each time step, and the *output_recipe* is used to output the results of the algorithm.

In addition to the recipes, a list of variable names is needed to specify which variables the algorithm will store in the Data object.

```python
# the variables that the algorithm will store
print(MeanField.output_variables)
```

Adding output obvservables

To add an additional variable to the output of an algorithm we must define a task that calculates the variable and add it to the output recipe.

Linear response functions

For example, let's calculate a linear response function that can be used to calculate the absorption spectrum of a system. Mathematically we will calculate

$$R(t) = \langle \psi(0) | \psi(t) \rangle$$

where $|\psi(t)\rangle$ is the diabatic wavefunction at time t.

```
def update_response_function(sim, parameters, state, **kwargs):
    # First get the diabatic wavefunction.
    wf_db = state.wf_db
    # If we are at the first timestep we can store the diabatic wavefunction in the␣
↪parameters object
    if sim.t_ind == 0:
        parameters.wf_db_0 = np.copy(wf_db)
    # Next calculate the response function and store it in the state object.
    state.response_function = np.sum(np.conj(parameters.wf_db_0) * wf_db, axis=-1)
    return parameters, state
```

Next we can add this task to the output recipe.

```
MeanField.output_recipe.append(update_response_function)
```

Finally we can add the relevant variable name to the output_variables list.

```
MeanField.output_variables.append('response_function')
```

We can then run a simulation and calculate the corresponding spectral function,

```
from qc_lab import Simulation
from qc_lab.dynamics import parallel_driver_multiprocessing
from qc_lab.models import SpinBoson

# instantiate a simulation
sim = Simulation()
print('default simulation settings: ', sim.default_settings)

# change settings to customize simulation
sim.settings.num_trajs = 1000
sim.settings.batch_size = 250
sim.settings.tmax = 50
sim.settings.dt = 0.01

# instantiate a model
sim.model = SpinBoson({'l_reorg': 0.2})
print('default model constants: ', sim.model.default_constants) # print out default␣
↪constants

# instantiate an algorithm
sim.algorithm = MeanField()
print('default algorithm settings: ', sim.algorithm.default_settings) # print out␣
↪default settings
```

(continues on next page)

(continued from previous page)

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

# run the simulation
data = parallel_driver_multiprocessing(sim, num_tasks=4)

# plot the data.
print('calculated quantities:', data.data_dic.keys())
response_function = data.data_dict['response_function']
time = sim.settings.tdat_output
plt.plot(time, np.real(response_function), label='R(t)')
plt.xlabel('time')
plt.ylabel('response function')
plt.legend()
plt.show()

plt.plot(np.real(np.roll(np.fft.fft(response_function), len(time)//2)))
plt.xlabel('freq')
plt.ylabel('absorbance')
plt.show()
```

Adiabatic populations

Next, let's calculate the adiabatic populations of the system as is sometimes done in scattering problems. Obviously these populations will only have a well-defined meaning in regimes with no nonadiabatic coupling.

```
def update_adiabatic_populations(sim, parameters, state, **kwargs):
    # First get the Hamiltonian and calculate its eigenvalues and eigenvectors.
    H = state.h_quantum # this is the quantum plus quantum-classical Hamiltonian.
    # Next obtain its eigenvalues and eigenvectors.
    evals, evecs = np.linalg.eigh(H)
    # Now calculate the adiabatic wavefunction.
    wf_adb = np.einsum('tji,tj->ti', np.conj(evecs), state.wf_db)
    # Finally calculate the populations (note that we do not sum over the batch, this is␣
↪done internally by QC Lab).
    pops_adb = np.abs(wf_adb)**2
    # Store the populations in the state object.
    state.pops_adb = pops_adb
    return parameters, state
```

Next we can add this task to the output recipe.

```
MeanField.output_recipe.append(update_adiabatic_populations)
```

Finally we can add the relevant variable name to the output_variables list.

```
MeanField.output_variables.append('pops_adb')
```

We can then run a simulation and plot the populations. Note that since the spin-boson model is always in a coupling regime these populations will not have a well-defined meaning.

```
from qc_lab import Simulation
from qc_lab.dynamics import serial_driver
from qc_lab.models import SpinBoson

# Instantiate a simulation.
sim = Simulation()
print('default simulation settings: ', sim.default_settings)

# Change settings to customize simulation.
sim.settings.num_trajs = 100
sim.settings.batch_size = 100
sim.settings.tmax = 25
sim.settings.dt = 0.01

# Instantiate a model.
sim.model = SpinBoson()
print('default model constants: ', sim.model.default_constants) # print out default␣
↪constants

# Instantiate an algorithm.
sim.algorithm = MeanField()
print('default algorithm settings: ', sim.algorithm.default_settings) # print out␣
↪default settings



# Define an initial diabatic wavefunction.
sim.state.wf_db = np.array([1, 0], dtype=complex)

# Run the simulation.
data = serial_driver(sim)

# Plot the data.
print('calculated quantities:', data.data_dic.keys())
classical_energy = data.data_dict['classical_energy']
quantum_energy = data.data_dict['quantum_energy']
populations = np.real(np.einsum('tii->ti', data.data_dict['dm_db']))
adiabatic_populations = np.real(data.data_dict['pops_adb'])
time = sim.settings.tdat_output
plt.plot(time, adiabatic_populations[:,0], label='adiabatic state 0')
plt.plot(time, adiabatic_populations[:,1], label='adiabatic state 1')
plt.xlabel('time')
plt.ylabel('population')
plt.legend()
plt.show()
```

Note

In the above code we chose to modify the MeanField class itself rather than an instance of it. This can lead to troublesome behavior in a Jupyter notebook where the class will not be reloaded if the cell is rerun. Restarting the kernel will fix this issue. Otherwise one can modify an instance of the class by creating a new instance and modifying it.

Modifying algorithm behavior

In the same way that we could modify the output recipe, it is possible to modify the initialization and update recipes in the same way. We will not go into detail on how to do this here but the process is the same as for the output recipe (except there is no output variable in those cases).

CHAPTER

FOUR

SOFTWARE REFERENCE

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

4.1 Software Reference

4.1.1 Ingredients

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

```python
def ingredient_name(model, constants, parameters, **kwargs):
    # Calculate var.
    var = None
    return var
```

For consistency we include all of the arguments even if the ingredient does not use them. An ingredient that generates the quantum Hamiltonian for a two-level system might look like this:

```python
def two_level_system_h_q(model, constants, parameters, **kwargs):
    """
    Quantum Hamiltonian for a two-level system.

    Required Constants:
        - two_level_system_a: Energy of the first level.
        - two_level_system_b: Energy of the second level.
        - two_level_system_c: Real part of the coupling between levels.
        - two_level_system_d: Imaginary part of the coupling between levels.

    Keyword Arguments:
        - batch_size: (Optional) Number of batches for vectorized computation.
    """
    del model
    if kwargs.get("batch_size") is not None:
        batch_size = kwargs.get("batch_size")
    else:
        batch_size = len(parameters.seed)
    h_q = np.zeros((batch_size, 2, 2), dtype=complex)
    h_q[:, 0, 0] = constants.two_level_system_a
    h_q[:, 1, 1] = constants.two_level_system_b
    h_q[:, 0, 1] = constants.two_level_system_c + 1j * constants.two_level_system_d
    h_q[:, 1, 0] = constants.two_level_system_c - 1j * constants.two_level_system_d
    return h_q
```

When incorporated directly into the model class (for instance when writing a model class from scratch) one should replace *model* with *self*. See the Model Development section of the User Guide 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.

Quantum Hamiltonian

This file contains ingredient functions for use in Model classes.

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

Quantum Hamiltonian for a nearest-neighbor lattice.

Required Constants:

- *num_quantum_states*: Number of quantum states (sites).
- *nearest_neighbor_lattice_hopping_energy*: Hopping energy between sites.
- *nearest_neighbor_lattice_periodic_boundary*: Boolean indicating periodic boundary conditions.

Keyword Arguments:

- *batch_size*: (Optional) Number of batches for vectorized computation.

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

Quantum Hamiltonian for a two-level system.

Required Constants:

- *two_level_system_a*: Energy of the first level.
- *two_level_system_b*: Energy of the second level.
- *two_level_system_c*: Real part of the coupling between levels.
- *two_level_system_d*: Imaginary part of the coupling between levels.

Keyword Arguments:

- *batch_size*: (Optional) Number of batches for vectorized computation.

Quantum-Classical Hamiltonian

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

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

Gradient of the diagonal linear quantum-classical coupling Hamiltonian.

Required Constants:

- *num_quantum_states*: Number of quantum states (sites).
- *num_classical_coordinates*: Number of classical coordinates
- *diagonal_linear_coupling*: Array of coupling constants (num_sites, num_classical_coordinates).

Keyword Arguments:

- *z*: complex-valued classical coordinates.
- *batch_size*: (Optional) Number of batches for vectorized computation.

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

Diagonal linear quantum-classical coupling Hamiltonian.

Diagonal elements are given by

$H_{ii} = \sum_j \gamma_{ij}(z_j + z_j^*)$

Required Constants:

- *num_quantum_states*: Number of quantum states (sites).
- *num_classical_coordinates*: Number of classical coordinates.
- *diagonal_linear_coupling*: Array of coupling constants (num_sites, num_classical_coordinates).

Keyword Arguments:

- *z*: complex-valued classical coordinates.
- *batch_size*: (Optional) Number of batches for vectorized computation.

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*`)`

Derivative of the classical harmonic oscillator Hamiltonian with respect to the *z* coordinate.

Required Constants:

- *classical_coordinate_weight*: Array of weights for classical coordinates.
- *harmonic_oscillator_frequency*: Array of harmonic oscillator frequencies.

Keyword Arguments:

- *z*: complex-valued classical coordinates.
- *batch_size*: (Optional) Number of batches for vectorized computation.

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

Harmonic oscillator classical Hamiltonian function.

Required Constants:

- *classical_coordinate_weight*: Array of weights for classical coordinates.
- *harmonic_oscillator_frequency*: Array of harmonic oscillator frequencies.
- *classical_coordinate_mass*: Array of masses for classical coordinates.

Keyword Arguments:

- *z*: complex-valued classical coordinates.
- *batch_size*: (Optional) Number of batches for vectorized computation.

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.

Required Constants:

- *kBT*: Thermal quantum.
- *classical_coordinate_weight*: Array of weights for classical coordinates.
- *harmonic_oscillator_frequency*: Array of harmonic oscillator frequencies.
- *classical_coordinate_mass*: Array of masses for classical coordinates.

Keyword Arguments:

- *seed*: Array of random seeds for initialization.

`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.

Required Constants:

- *kBT*: Thermal quantum.
- *classical_coordinate_weight*: Array of weights for classical coordinates.
- *harmonic_oscillator_frequency*: Array of harmonic oscillator frequencies.
- *classical_coordinate_mass*: Array of masses for classical coordinates.

Keyword Arguments:

- *seed*: Array of random seeds for initialization.

CHAPTER

FIVE

BIBLIOGRAPHY

1. Miyazaki, K.; Krotz, A.; Tempelaar, R. Mixed Quantum-Classical Dynamics under Arbitrary Unitary Basis Transformations. J. Chem. Theory Comput. 2024, 20 (15), 6500-6509. https://doi.org/10.1021/acs.jctc.4c00555
2. Tully, J. C. Mixed Quantum–Classical Dynamics. Faraday Discuss. 1998, 110 (0), 407–419. https://doi.org/10.1039/A801824C.
3. Hammes-Schiffer, S.; Tully, J. C. Proton Transfer in Solution: Molecular Dynamics with Quantum Transitions. J. Chem. Phys. 1994, 101 (6), 4657–4667. https://doi.org/10.1063/1.467455.
4. Tempelaar, R.; Reichman, D. R. Generalization of Fewest-Switches Surface Hopping for Coherences. The Journal of Chemical Physics 2018, 148 (10), 102309. https://doi.org/10.1063/1.5000843.
5. Krotz, A.; Provazza, J.; Tempelaar, R. A Reciprocal-Space Formulation of Mixed Quantum–Classical Dynamics. J. Chem. Phys. 2021, 154 (22), 224101. https://doi.org/10.1063/5.0053177.
6. Fenna, R. E. & Matthews, B. W. Chlorophyll arrangement in a bacteriochlorophyll protein from Chlorobium limicola. Nature 258, 573–577 (1975). https://doi.org/10.1038/258573a0.
7. Mulvihill, E.; Lenn, K. M.; Gao, X.; Schubert, A.; Dunietz, B. D.; Geva, E. Simulating Energy Transfer Dynamics in the Fenna–Matthews–Olson Complex via the Modified Generalized Quantum Master Equation. The Journal of Chemical Physics 2021, 154 (20), 204109. https://doi.org/10.1063/5.0051101.

PYTHON MODULE INDEX

INDEX