BoFiN-HEOM

Release 0.0.1

Oct 21, 2020

Contents

1 Introduction **3**

2 Installation **5**
- 2.1 Installing dependencies & setting up . . . 5
- 2.2 Installing the Python version . . . 6
- 2.3 Installing the C++ version . . . 6

3 Bosonic Environments **7**
- 3.1 Multiple environments . . . 9

4 Fermionic Environments **11**
- 4.1 Multiple environments . . . 13

5 Dynamics (ODE) solver **15**

6 Steady-state Solver **17**

7 References **19**

8 bofin **21**
- 8.1 bofin package . . . 21

9 Indices and tables **25**

Bibliography **27**

Python Module Index **29**

Index **31**

BoFiN-HEOM is a "Hierarchical Equations of Motion" solver for quantum systems coupled to Bosonic or Fermionic environments. The HEOM method was originally developed by Tanimura and Kubo [TK89] .

BoFiN-HEOM is designed to be as generic as possible. It relies on the QuTiP package, and is provided in two versions :

- **BoFiN** : Pure Python version of the HEOM solver. Has a `BosonicHEOMSolver` and `FermionicHEOMSolver` class. It can be found here .
- **BoFiN-fast** : Hybrid C++ - Python version, of the HEOM solver. Here the backend for RHS construction of the HEOM solver written in C++. It is otherwise completely identical (both in user interface and functionality) to the pure Python version. It can be found here .

It should be noted that the C++ version dramatically speeds up RHS construction, with respect to the Python version. We have noted more than 10x speedup using the C++ version for some hard Fermionic HEOM problems.

The provided example notebooks explain some common choices of environmental parameterization. See Installation Instructions for details on how to set up both versions.

CHAPTER 1

Introduction

The HEOM method was originally developed in the context of Physical Chemistry to ‘’exactly” solve a quantum system in contact with a bosonic environment, encapsulated in the Hamiltonian

$$H = H_s + \sum_k \omega_k a_k^\dagger a_k + \hat{Q} \sum_k g_k \left(a_k + a_k^\dagger \right).$$

As in other solutions to this problem, the properties of the bath are encapsulated by its temperature and its spectral density,

$$J(\omega) = \pi \sum_k g_k^2 \delta(\omega - \omega_k).$$

In the HEOM, for bosonic baths, we typically choose a Drude-Lorentz spectral density

$$J_D = \frac{2\lambda\gamma\omega}{(\gamma^2 + \omega^2)}$$

or an under-damped Brownian motion spectral density

$$J_U = \frac{\alpha^2 \Gamma \omega}{[(\omega_c^2 - \omega^2)^2 + \Gamma^2 \omega^2]}.$$

Other cases are usually approached with fitting.

Given the spectral density, the HEOM needs a decomposition of the bath correlation functions in terms of exponentials. In the Matsubara and Pade sections we describe how this is done with code examples, and how these are passed to the solver.

In addition to the Bosonic case we also provide a solver for Feriomic environments.

The two options are dealt with by two different classes, `BosonicHEOMSolver` and `FermionicHEOMSolver`.

Each have associated ODE solvers for dynamics and steady-state solutions, as required.

CHAPTER 2

Installation

We have developed two packaged versions of the HEOM solver :

- **BoFiN** : Pure Python version of the HEOM solver. Has a `BosonicHEOMSolver` and `FermionicHEOMSolver` class. It can be found here .
- **BoFiN-fast** : Hybrid C++ - Python version, of the HEOM solver. Here the backend for RHS construction of the HEOM solver written in C++. It is otherwise completely identical (both in user interface and functionality) to the pure Python version. It can be found here .

The following sections explain how to set up both versions, and common dependencies.

2.1 Installing dependencies & setting up

The core requirements are `numpy`, `scipy`, `cython` and `qutip`. For uniformity across platforms, we recommend using Conda environments to keep the setup clean, and to install dependencies painlessly (since `pip install` is known to have issues on Mac OS). Once you have Conda installed, make a fresh Python 3 environment called `bofin_env`, and then switch to it:

```
conda create -n bofin_env python=3.8
conda activate bofin_env
```

In your `bofin_env` environment, install requirements using:

```
conda install numpy scipy cython
conda install -c conda-forge qutip
```

Also, `matplotlib` is required for visualizations. This will ensure painless setup across Windows, Linux and Mac OS.

2.2 Installing the Python version

Clone the BoFiN repository given here using `git clone`.

Once you have the dependencies installed, from the parent repository folder, run the following command:

```
pip3 install -e .
```

This will install the pure Python version of the Bosonic HEOM Solver.

2.3 Installing the C++ version

Clone the BoFiN-fast repository given here using `git clone`.

Once you have the dependencies installed, from the parent repository folder, run the following commands:

```
python3 setup.py build_ext --inplace
pip3 install -e .
```

This installs the hybrid Python - C++ version of the HEOM solvers. These are identical in usage and functionality to the Python solvers.

CHAPTER 3

Bosonic Environments

The basic class object used to construct the problem is imported in the following way (alongside QuTiP, with which we define system Hamiltonian and coupling operators)

```
from qutip import *
from bofin.heom import BosonicHEOMSolver
```

If one is using the C++ BoFiN_fast package, the import is instead

```
from qutip import *
from bofinfast.heom import BosonicHEOMSolver
```

Apart from this difference in import, and some additional features in the solvers in the C++ variant, the functionality that follows applies to both libraries.

One defines a particular problem instance in the following way:

```
Solver = BosonicHEOMSolver(Hsys, Q, ckAR, ckAI, vkAR, vkAI, NC, options=options)
```

The parameters accepted by the solver are : - `Hsys` : the system Hamiltonian in quantum object form - `Q` a coupling operator (or list of coupling operators) that couple the system to the environment - `ckAR` and `vkAR` : respectively the coefficients and frequencies of the real parts of the correlation functions - `ckAI` and `vkAI` : respectively the coefficients and frequencies of the imaginary parts of the correlation functions - `NC` : the truncation parameter of the hierarchy - `options` is a standard QuTiP `ODEoptions` object, which is used by the ODE solver.

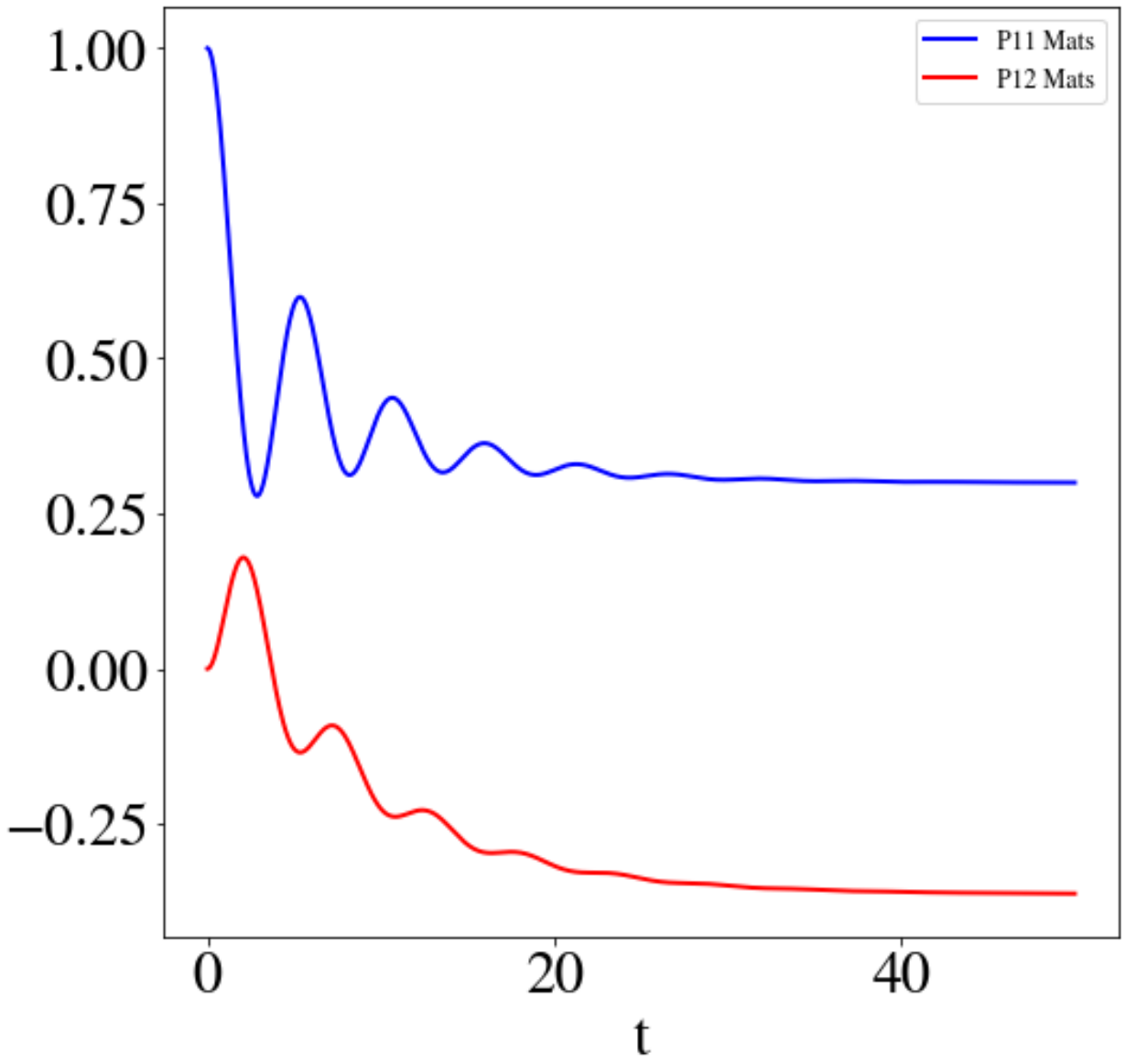
Thus an example solution to a single spin coupled to a Drude-Lorentz spectral density with Matsubara decomposition is (taken from example notebook 1a):

```
%pylab inline
from qutip import *
from bofin.heom import BosonicHEOMSolver

def cot(x):
    return 1./np.tan(x)
```

(continues on next page)

(continued from previous page)

```
    # Defining the system Hamiltonian
eps = .5      # Energy of the 2-level system.
Del = 1.0     # Tunnelling term
Hsys = 0.5 * eps * sigmaz() + 0.5 * Del* sigmax()

    # Initial state of the system.
rho0 = basis(2,0) * basis(2,0).dag()

    # System-bath coupling (Drude-Lorentz spectral density)
Q = sigmaz() # coupling operator

tlist = np.linspace(0, 50, 1000)

    #Bath properties:
gamma = .5 # cut off frequency
lam = .1 # coupling strength
T = 0.5
beta = 1./T

    #HEOM parameters
NC = 5 # cut off parameter for the bath
Nk = 2 # number of Matsubara terms
ckAR = [ lam * gamma * (cot(gamma / (2 * T)))]
ckAR.extend([(4 * lam * gamma * T *  2 * np.pi * k * T / (( 2 * np.pi * k * T)**2 -␣
↪gamma**2)) for k in range(1,Nk+1)])
vkAR = [gamma]
vkAR.extend([2 * np.pi * k * T for k in range(1,Nk+1)])
ckAI = [lam * gamma * (-1.0)]
vkAI = [gamma]
NR = len(ckAR)
NI = len(ckAI)
Q2 = [Q for kk in range(NR+NI)]


options = Options(nsteps=15000, store_states=True, rtol=1e-14, atol=1e-14)
HEOMMats = BosonicHEOMSolver(Hsys, Q2, ckAR, ckAI, vkAR, vkAI, NC, options=options)

    #Run ODE solver
resultMats = HEOMMats.run(rho0, tlist)

# Define some operators with which we will measure the system
# Populations
P11p=basis(2,0) * basis(2,0).dag()
P22p=basis(2,1) * basis(2,1).dag()
# 1,2 element of density matrix  - corresonding to coherence
P12p=basis(2,0) * basis(2,1).dag()
# Calculate expectation values in the bases
P11exp = expect(resultMats.states, P11p)
P22exp = expect(resultMats.states, P22p)
P12exp = expect(resultMats.states, P12p)

# Plot the results
fig, axes = plt.subplots(1, 1, sharex=True, figsize=(8,8))
axes.plot(tlist, np.real(P11exp), 'b', linewidth=2, label="P11 Mats")
axes.plot(tlist, np.real(P12exp), 'r', linewidth=2, label="P12 Mats")
axes.set_xlabel(r't', fontsize=28)
axes.legend(loc=0, fontsize=12)
```



3.1 Multiple environments

The above example describes a single environment parameterized by the lists of coefficients and frequencies in the correlation functions.

For multiple environments, the list of coupling operators and bath properties must all be extended in a particular way. Note this functionality differs in the case of the Fermionic solver.

For the Bosonic solver, for `N` baths, each `ckAR`, `vkAR`, `ckAI`, and `vkAI` are extended `N` times with the appropriate number of terms of that bath.

On the other hand, the list of coupling operators is defined in such a way that the terms corresponding to the real cooefficients are **given first**, and the imaginary terms after. Thus if each bath has N_k coefficients, the list of coupling operators is of length $N_k \times (N_R + N_I)$.

This is best illustrated by the example in example notebook 2. In that case each bath is identical, and there are seven baths, each with a unique coupling operator defined by a projector onto a single state:

```
ckAR = [pref * lam * gamma * (cot(gamma / (2 * T))) + 0.j]
ckAR.extend([(pref * 4 * lam * gamma * T *  2 * np.pi * k * T / (( 2 * np.pi * k *␣
↪T)**2 - gamma**2))+0.j for k in range(1,Nk+1)])
vkAR = [gamma+0.j]
vkAR.extend([2 * np.pi * k * T + 0.j for k in range(1,Nk+1)])
ckAI = [pref * lam * gamma * (-1.0) + 0.j]
vkAI = [gamma+0.j]

NR = len(ckAR)
NI = len(ckAI)
Q2 = []
ckAR2 = []
ckAI2 = []
vkAR2 = []
vkAI2 = []
for m in range(7):
    Q2.extend([ basis(7,m)*basis(7,m).dag() for kk in range(NR)])
    ckAR2.extend(ckAR)
    vkAR2.extend(vkAR)

for m in range(7):
    Q2.extend([ basis(7,m)*basis(7,m).dag() for kk in range(NI)])
    ckAI2.extend(ckAI)
    vkAI2.extend(vkAI)
```

CHAPTER 4

Fermionic Environments

The basic class object used to construct the problem is imported in the following way (alongside QuTiP, with which we define system Hamiltonian and coupling operators), for the pure Python BoFiN version

```
from qutip import *
from bofin.heom import FermionicHEOMSolver
```

If one is using the C++ BoFiN-fast package, the import is instead

```
from qutip import *
from bofinfast.heom import FermionicHEOMSolver
```

Apart from this difference in import, and some additional features in the solvers in the C++ variant, the functionality that follows applies to both libraries.

One defines a particular problem instance in the following way:

```
Solver = FermionicHEOMSolver(Hsys, Q,  eta_list, gamma_list, NC,o ptions=options)
```

The parameters accepted by the solver are :

- `Hsys` : system Hamiltonian in quantum object form
- `Q` : a list of coupling operators (minimum two) that couple the system to the environment
- `eta_list` and `gamma_list` : the coefficients and frequencies of the correlation functions of the environment
- `NC` : the truncation parameter of the hierarchy
- `options` : standard QuTiP `ODEoptions` object, which is used by the ODE solver.

Note that for the Fermionic case, unlike the Bosonic case, we **don't** explicitly split the real and imaginary parts of the correlation functions. Also, for the Fermionic case, a single environment has a minimum of two coupling operators (due to the fundemental difference in how the environment interacts with the system for fermionic environments).

A simple example of how this code works, taken from example notebook 4b, is:

```
%pylab inline
from qutip import *
from bofin.heom import FermionicHEOMSolver
def deltafun(j,k):
    if j==k:
        return 1.
    else:
        return 0.

    # Defining the system Hamiltonian
#Single fermion.
d1 = destroy(2)
#Site energy
e1 = 1.
H0 = e1*d1.dag()*d1
#There are two leads, but we seperate the interaction into two terms, labelled with␣
↪\sigma=\pm
#such that there are 4 interaction operators (See paper)
Qops = [d1.dag(),d1,d1.dag(),d1]

        #Bath properties:
Gamma = 0.01  #coupling strength
W=1. #cut-off
T = 0.025851991 #temperature
beta = 1./T

theta = 2. #Bias
mu_l = theta/2.
mu_r = -theta/2.
        #HEOM parameters
#Pade decompositon: construct correlation parameters

tlist = np.linspace(0,10,200)
lmax =10
Alpha =np.zeros((2*lmax,2*lmax))
for j in range(2*lmax):
    for k in range(2*lmax):
        Alpha[j][k] = (deltafun(j,k+1)+deltafun(j,k-1))/sqrt((2*(j+1)-1)*(2*(k+1)-1))
eigvalsA=eigvalsh(Alpha)
eps = []
for val in  eigvalsA[0:lmax]:
    eps.append(-2/val)

AlphaP =np.zeros((2*lmax-1,2*lmax-1))
for j in range(2*lmax-1):
    for k in range(2*lmax-1):
        AlphaP[j][k] = (deltafun(j,k+1)+deltafun(j,k-1))/sqrt((2*(j+1)+1)*(2*(k+1)+1))

eigvalsAP=eigvalsh(AlphaP)
chi = []
for val in  eigvalsAP[0:lmax-1]:
    #print(-2/val)
    chi.append(-2/val)
eta_list = [0.5*lmax*(2*(lmax + 1) - 1)*(
  np.prod([chi[k]**2 - eps[j]**2 for k in range(lmax - 1)])/
    np.prod([eps[k]**2 - eps[j]**2 +deltafun(j,k) for k in range(lmax)]))
          for j in range(lmax)]
```

(continues on next page)

(continued from previous page)

```
kappa = [0]+eta_list
epsilon = [0]+eps

def f_approx(x):
    f = 0.5
    for l in range(1,lmax+1):
        f= f - 2*kappa[l]*x/(x**2+epsilon[l]**2)
    return f

def C(tlist,sigma,mu):
    eta_list = []
    gamma_list  =[]


    eta_0 = 0.5*Gamma*W*f_approx(1.0j*beta*W)

    gamma_0 = W - sigma*1.0j*mu
    eta_list.append(eta_0)
    gamma_list.append(gamma_0)
    if lmax>0:
        for l in range(1,lmax+1):
            eta_list.append(-1.0j*(kappa[l]/beta)*Gamma*W**2/(-(epsilon[l]**2/
↪beta**2)+W**2))
            gamma_list.append(epsilon[l]/beta - sigma*1.0j*mu)
    c_tot = []
    for t in tlist:
        c_tot.append(sum([eta_list[l]*exp(-gamma_list[l]*t) for l in range(lmax+1)]))
    return c_tot, eta_list, gamma_list


cppL,etapL,gampL = C(tlist,1.0,mu_l)
cpmL,etamL,gammL = C(tlist,-1.0,mu_l)
cppR,etapR,gampR = C(tlist,1.0,mu_r)
cpmR,etamR,gammR = C(tlist,-1.0,mu_r)

Kk=lmax+1
Ncc = 2  #For a single impurity we converge with Ncc = 2
#Note here that the functionality differs from the bosonic case. Here we send lists␣
↪of lists, were each sub-list
#refers to one of the two coupling terms for each bath (the notation here refers to␣
↪eta|sigma|L/R)

eta_list = [etapR,etamR,etapL,etamL]
gamma_list = [gampR,gammR,gampL,gammL]
options = Options(nsteps=15000, store_states=True, rtol=1e-14, atol=1e-14)
resultHEOM2 = FermionicHEOMSolver(H0, Qops,  eta_list, gamma_list, Ncc,
↪options=options)
```

4.1 Multiple environments

In dealing with multiple environments the Fermionic solver operates in a slightly different way to the Bosonic case, as already shown in the above example. Each bath is specified by coupling to two system operators (which are related by hermitian conjugation), and the parameters for the bath coefficients associated with each of the those operators are defined in a list in the corresponding possition in `eta_list` and `gamma_list`.

Typically these must be ordered in the above shown way, such that, for the first environment, `Qops[0]` is the operator associated with the correlation function $\sigma = +$, while `Qops[1]` is associated with $\sigma = -$.

This continues for each environment, with a corresponding set of two operators in `Qops`, and corresponding lists of `etap*` and `etam*` in `eta_list`.

CHAPTER 5

Dynamics (ODE) solver

Once either a Bosonic or Fermionic problem has been defined, one can solve the dynamics for a particular initial condition. This was already shown in the Bosonic class example, but in more detail, given a defined `BosonicHEOMSolver` instance, one can call:

```
HEOMMats = BosonicHEOMSolver(Hsys, Q2, ckAR, ckAI, vkAR, vkAI, NC, options=options)
#Run ODE solver
resultMats = HEOMMats.run(rho0, tlist)
```

By default this takes only two parameters: - `rho0`: a quantum object state or density matrix defining in the initial condition for the system - `tlist`: a set of timesteps for which to return results

The object returned is a standard QuTiP `results` object. Most importantly, this contains the system state at each time-step in `resultmMats.states`.

Note that at this time this auxiliary density operators are **not** returned directly to the user. This will be modified in the near future to be an option (currently they are returned by default in the `steadystate()` solver).

> The BoFiN-fast C++ solver contains some additional options to use Intel MKL parallelization, that can help speed-up the ODE solution.

CHAPTER 6

Steady-state Solver

Once a (Bosonic or Fermionic) problem has been defined, one can solve for the steady-state dynamics. By default this is done by direct SPLU decomposition (combined with the standard imposition of normalization on the system density operator).

Typical usage is:

```
HEOMMats = BosonicHEOMSolver(Hsys, Q2, ckAR, ckAI, vkAR, vkAI, NC, options=options)
#get steady state
rho_ss,full_ss=resultHEOM.steady_state()
```

`rho_ss` is a standard QuTiP quantum object for the system state alone. `full_ss` includes that and every auxiliary density operator encoding the environment, but in `numpy array` format (not converted to quantum objects). To determine which ADO is which, one can use the `enr_state_dictionaries()` built into QuTiP. This is also used to generate the ordering of all the HEOM operators in our code. An example of how this is done is presented in example notebook 4b, where it is used to get the electronic current from the ADOs.

We will supplement this with further examples for the Bosonic case in the future.

Additional options for `steady_state()` include `use_mkl = True` which will use MKL parallelization for the problem, if available. BoFiN_fast also includes an `approx=True` option for use of an iterative `lgmres` approach.

CHAPTER 7

References

CHAPTER 8

bofin

8.1 bofin package

8.1.1 Submodules

8.1.2 bofin.heom module

This module provides exact solvers for a system-bath setup using the hierarchy equations of motion (HEOM).

class bofin.heom.**BosonicHEOMSolver**(*H_sys*, *coup_op*, *ckAR*, *ckAI*, *vkAR*, *vkAI*, *N_cut*, *options=None*)

Bases: object

This is a class for solvers that use the HEOM method for calculating the dynamics evolution. There are many references for this. A good introduction, and perhaps closest to the notation used here is: DOI:10.1103/PhysRevLett.104.250401 A more canonical reference, with full derivation is: DOI: 10.1103/PhysRevA.41.6676 The method can compute open system dynamics without using any Markovian or rotating wave approximation (RWA) for systems where the bath correlations can be approximated to a sum of complex exponentials. The method builds a matrix of linked differential equations, which are then solved used the same ODE solvers as other qutip solvers (e.g. mesolve)/

H_sys
System Hamiltonian Or Liouvillian Or QobjEvo Or list of Hamiltonians with time dependence

Format for input (if list): [time_independent_part, [H1, time_dep_function1], [H2, time_dep_function2]]

Type Qobj or list

coup_op [Qobj or list] Operator describing the coupling between system and bath. Could also be a list of operators, which needs to be the same length as ck's and vk's.

ckAR, ckAI, vkAR, vkAI [lists] Lists containing coefficients for fitting spectral density correlation

N_cut [int] Cutoff parameter for the bath

options [qutip.solver.Options] Generic solver options. If set to None the default options will be used

boson_grad_n(*he_n*)
Get the gradient term for the hierarchy ADM at level n

boson_grad_next(*he_n*, *k*, *next_he*)
Get the next gradient

boson_grad_prev(*he_n*, *k*, *prev_he*)
Get the previous gradient

boson_rhs()
Make the RHS for bosonic case

configure(*H_sys*, *coup_op*, *ckAR*, *ckAI*, *vkAR*, *vkAI*, *N_cut*, *options=None*)
Configure the solver using the passed parameters The parameters are described in the class attributes, unless there is some specific behaviour

Parameters **options** (`qutip.solver.Options`) – Generic solver options. If set to None the default options will be used

populate(*heidx_list*)
Given a Hierarchy index list, populate the graph of next and previous elements

process_input(*H_sys*, *coup_op*, *ckAR*, *ckAI*, *vkAR*, *vkAI*, *N_cut*, *options=None*)
Type-checks provided input Merges same gammas

reset()
Reset any attributes to default values

run(*rho0*, *tlist*)
Function to solve for an open quantum system using the HEOM model.

Parameters

- **rho0** (*Qobj*) – Initial state (density matrix) of the system.
- **tlist** (*list*) – Time over which system evolves.

Returns **results** – Object storing all results from the simulation.

Return type `qutip.solver.Result`

steady_state(*max_iter_refine=100*, *use_mkl=False*, *weighted_matching=False*)
Computes steady state dynamics

max_iter_refine [Int] Parameter for the mkl LU solver. If pardiso errors are returned this should be increased.

use_mkl [Boolean] Optional override default use of mkl if mkl is installed.

weighted_matching [Boolean] Setting this true may increase run time, but reduce stability (pardisio may not converge).

class bofin.heom.**FermionicHEOMSolver**(*H_sys*, *coup_op*, *ck*, *vk*, *N_cut*, *options=None*)
Bases: `object`

Same as BosonicHEOMSolver, but with Fermionic baths.

H_sys
System Hamiltonian Or Liouvillian Or QobjEvo Or list of Hamiltonians with time dependence

Format for input (if list): [time_independent_part, [H1, time_dep_function1], [H2, time_dep_function2]]

Type Qobj or list

coup_op
Operator describing the coupling between system and bath. Could also be a list of operators, which needs to be the same length as ck's and vk's.

Type Qobj or list

ck, vk
Lists containing spectral density correlation

Type lists

N_cut
Cutoff parameter for the bath

Type int

options
Generic solver options. If set to None the default options will be used

Type `qutip.solver.Options`

configure(*H_sys*, *coup_op*, *ck*, *vk*, *N_cut*, *options=None*)
Configure the solver using the passed parameters The parameters are described in the class attributes, unless there is some specific behaviour

Parameters **options** (`qutip.solver.Options`) – Generic solver options. If set to None the default options will be used

fermion_grad_n(*he_n*)
Get the gradient term for the hierarchy ADM at level n

fermion_grad_next(*he_n*, *k*, *next_he*, *idx*)
Get next gradient

fermion_grad_prev(*he_n*, *k*, *prev_he*, *idx*)
Get next gradient

fermion_rhs()
Make the RHS for fermionic case

populate(*heidx_list*)
Given a Hierarchy index list, populate the graph of next and previous elements

process_input(*H_sys*, *coup_op*, *ck*, *vk*, *N_cut*, *options=None*)
Type-checks provided input Merges same gammas

reset()
Reset any attributes to default values

run(*rho0*, *tlist*)
Function to solve for an open quantum system using the HEOM model.

Parameters

- **rho0** (*Qobj*) – Initial state (density matrix) of the system.
- **tlist** (*list*) – Time over which system evolves.

Returns **results** – Object storing all results from the simulation.

Return type `qutip.solver.Result`

steady_state(*max_iter_refine=100*, *use_mkl=False*, *weighted_matching=False*)
Computes steady state dynamics

max_iter_refine [Int] Parameter for the mkl LU solver. If pardiso errors are returned this should be increased.

use_mkl [Boolean] Optional override default use of mkl if mkl is installed.

weighted_matching [Boolean] Setting this true may increase run time, but reduce stability (pardisio may not converge).

bofin.heom.**add_at_idx**(*seq*, *k*, *val*)
Add (subtract) a value in the tuple at position k

bofin.heom.**nexthe**(*current_he*, *k*, *ncut*)
Calculate the next heirarchy index for the current index *n*.

bofin.heom.**prevhe**(*current_he*, *k*, *ncut*)
Calculate the previous heirarchy index for the current index *n*.

8.1.3 Module contents

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[LACN19] Neill Lambert, Shahnawaz Ahmed, Mauro Cirio, and Franco Nori. Virtual excitations in the ultra-strongly-coupled spin-boson model: physical results from unphysical modes. *arXiv preprint arXiv:1903.05892*, 2019.

[Tan20] Yoshitaka Tanimura. Numerically "exact" approach to open quantum dynamics: the hierarchical equations of motion (heom). *The Journal of Chemical Physics*, 153(2):020901, 2020. URL: https://doi.org/10.1063/5.0011599, doi:10.1063/5.0011599.

[TK89] Yoshitaka Tanimura and Ryogo Kubo. Time evolution of a quantum system in contact with a nearly gaussian-markoffian noise bath. *J. Phys. Soc. Jpn.*, 58(1):101–114, 1989. doi:10.1143/jpsj.58.101.

Python Module Index

b

bofin, 24
bofin.heom, 21

Index

A

add_at_idx() (*in module bofin.heom*), 24

B

bofin (*module*), 24
bofin.heom (*module*), 21
boson_grad_n() (*bofin.heom.BosonicHEOMSolver method*), 22
boson_grad_next() (*bofin.heom.BosonicHEOMSolver method*), 22
boson_grad_prev() (*bofin.heom.BosonicHEOMSolver method*), 22
boson_rhs() (*bofin.heom.BosonicHEOMSolver method*), 22
BosonicHEOMSolver (*class in bofin.heom*), 21

C

configure() (*bofin.heom.BosonicHEOMSolver method*), 22
configure() (*bofin.heom.FermionicHEOMSolver method*), 23
coup_op (*bofin.heom.FermionicHEOMSolver attribute*), 22

F

fermion_grad_n() (*bofin.heom.FermionicHEOMSolver method*), 23
fermion_grad_next() (*bofin.heom.FermionicHEOMSolver method*), 23
fermion_grad_prev() (*bofin.heom.FermionicHEOMSolver method*), 23
fermion_rhs() (*bofin.heom.FermionicHEOMSolver method*), 23
FermionicHEOMSolver (*class in bofin.heom*), 22

H

H_sys (*bofin.heom.BosonicHEOMSolver attribute*), 21
H_sys (*bofin.heom.FermionicHEOMSolver attribute*), 22

N

N_cut (*bofin.heom.FermionicHEOMSolver attribute*), 23
nexthe() (*in module bofin.heom*), 24

O

options (*bofin.heom.FermionicHEOMSolver attribute*), 23

P

populate() (*bofin.heom.BosonicHEOMSolver method*), 22
populate() (*bofin.heom.FermionicHEOMSolver method*), 23
prevhe() (*in module bofin.heom*), 24
process_input() (*bofin.heom.BosonicHEOMSolver method*), 22
process_input() (*bofin.heom.FermionicHEOMSolver method*), 23

R

reset() (*bofin.heom.BosonicHEOMSolver method*), 22
reset() (*bofin.heom.FermionicHEOMSolver method*), 23
run() (*bofin.heom.BosonicHEOMSolver method*), 22
run() (*bofin.heom.FermionicHEOMSolver method*), 23

S

steady_state() (*bofin.heom.BosonicHEOMSolver method*), 22
steady_state() (*bofin.heom.FermionicHEOMSolver method*), 23