PyCCE

Release 0.6.8

Mykyta Onizhuk

Oct 26, 2021

GETTING STARTED

CHAPTER

ONE

THEORETICAL BACKGROUND

This document contains a brief list of the coupling parameters between the central and the bath spins used in **PyCCE**, a description of the qubit dephasing, and a summary of the cluster correlation expansion (CCE) method. You can find more details in the following references[1][2][3].

1.1 Hamiltonian

The **PyCCE** package allows one to simulate the dynamics of a central spin interacting with a spin bath through the following Hamiltonian:

$$\hat{H} = \hat{H}_S + \hat{H}_{SB} + \hat{H}_B$$

Where $\hat{H}_S$ is the Hamiltonian of the free central spin, $\hat{H}_{SB}$ denotes interactions between central spin and a spin belonging to the bath, and $\hat{H}_B$ are intrinsic bath spin interactions:

$$\begin{aligned}
\hat{H}_S &= \mathbf{S}\mathbf{D}\mathbf{S} + \mathbf{B}\gamma_S\mathbf{S} \\
\hat{H}_{SB} &= \sum_i \mathbf{S}\mathbf{A}_i\mathbf{I}_i \\
\hat{H}_B &= \sum_i \mathbf{I}_i\mathbf{P}_i\mathbf{I}_i + \mathbf{B}\gamma_i\mathbf{I}_i + \sum_{i>j} \mathbf{I}_i\mathbf{J}_{ij}\mathbf{I}_j
\end{aligned}$$

Where $\mathbf{S} = (\hat{S}_x, \hat{S}_y, \hat{S}_z)$ are the components of spin operators of the central spin, $\mathbf{I} = (\hat{I}_x, \hat{I}_y, \hat{I}_z)$ are the components of the bath spin operators, and $\mathbf{B} = (B_x, B_y, B_z)$ is an external applied magnetic field.

The interactions are described by the following tensors that are either required to be input by user or can be generated by the package itself (see *Hamiltonian Parameters Input* for details):

- $\mathbf{D}$ ($\mathbf{P}$) is the self-interaction tensor of the central spin (bath spin). For the electron spin, the tensor corresponds to the zero-field splitting (ZFS) tensor. For nuclear spins corresponds to the quadrupole interactions tensor.
- γ_i-spin describing the interaction of the spin and the external magnetic field B. We assume that for the bath spins, it is isotropic.
- $\mathbf{A}$ is the interaction tensor between central and bath spins. In the case of the nuclear spin bath, it corresponds to the hyperfine couplings.
- $\mathbf{J}$ is the interaction tensor between bath spins.

[1] Mykyta Onizhuk and Giulia Galli. "PyCCE: A Python Package for Cluster Correlation Expansion Simulations of Spin Qubit Dynamic". Adv. Theory Simul. 2021, 2100254, https://onlinelibrary.wiley.com/doi/10.1002/adts.202100254

[2] Wen Yang and Ren-Bao Liu. "Quantum many-body theory of qubit decoherence in a finite-size spin bath". Phys. Rev. B78, p. 085315, https://link.aps.org/doi/10.1103/PhysRevB.78.085315

[3] Mykyta Onizhuk et al. "Probing the Coherence of Solid-State Qubits at Avoided Crossings". PRX Quantum 2, p. 010311. https://link.aps.org/doi/10.1103/PRXQuantum.2.010311.

1.2 Qubit dephasing

Usually, two coherence times are measured to characterize the loss of a qubit coherence - T_1 and T_2. T_1 defines the timescale over which the qubit population is thermalized; T_2 describes a purely quantum phenomenon - the loss of the phase of the qubit's superposition state.

In the pure dephasing regime ($T_1 >> T_2$) the decoherence of the central spin is completely determined by the decay of the off diagonal element of the density matrix of the qubit.

Namely, if the qubit is initially prepared in the $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{i\phi}|1\rangle)$ state, the loss of the relative phase of the $|0\rangle$ and $|1\rangle$ levels is characterized by the coherence function:

$$\mathcal{L}(t) = \frac{\langle 1|\,\hat{\rho}_S(t)\,|0\rangle}{\langle 1|\,\hat{\rho}_S(0)\,|0\rangle} = \frac{\langle\hat{\sigma}_-(t)\rangle}{\langle\hat{\sigma}_-(0)\rangle}$$

Where $\hat{\rho}_S(t)$ is the density matrix of the central spin and $|0\rangle$ and $|1\rangle$ are qubit levels.

The cluster correlation expansion (CCE) method was first introduced in ref.[?]. The core idea of the CCE approach is that the spin bath-induced decoherence can be factorized into set of irreducible contributions from the bath spin clusters. Written in terms of the coherence function:

$$\mathcal{L}(t) = \prod_C \tilde{L}_C = \prod_i \tilde{L}_{\{i\}} \prod_{i,j} \tilde{L}_{\{ij\}} ...$$

Where each cluster contribution is defined recursively as:

$$\tilde{L}_C = \frac{L_C}{\prod_{C'} \tilde{L}_{C' \subset C}}$$

Where L_C is a coherence function of the qubit, interacting only with the bath spins in a given cluster C (with the cluster Hamiltonian $\hat{H}_C$), and $\tilde{L}_{C'}$ are contributions of C' subcluster of C.

For example, the contribution of the single spin i is equal to the coherence function of the bath with one isolated spin i:

$$\tilde{L}_i = L_i$$

The contribution of pair of spins i and j is equal to:

$$\tilde{L}_{ij} = \frac{L_{ij}}{\tilde{L}_i \tilde{L}_j}$$

and so on.

Maximum size of the cluster included into the expansion determines the order of CCE approximation. For example, in the CCE2 approximation, only contributions up to spin pairs are included, and in CCE3 - up to triplets of bath spins are included, etc.

The way the coherence function for each cluster is computed slightly varies between depending on whether the conventional or generalized CCE method is used.

1.2.1 Conventional CCE

In the original formulation of the CCE method, the total Hamiltonian of the system is reduced to the sum of two effective Hamiltonians, conditioned on the qubit levels of the central spin:

$$\hat{H} = 00 \otimes \hat{H}^{(0)} + 11 \otimes \hat{H}^{(1)}$$

Where $\hat{H}^{(\alpha)}$ is an effective Hamiltonian acting on the bath when the central qubit is in the α state ($\alpha = 0, 1$ is one of the two eigenstates of the $\hat{H}_S$ chosen as qubit levels).

Given an initial qubit state $\psi = \frac{1}{\sqrt{2}}(0 + e^{i\phi}1)$ and an initial state of the bath spin cluster C characterized by the density matrix $\hat{\rho}_C$, the coherence function of the qubit interacting with the cluster C is computed as:

$$L_C(t) = Tr[\hat{U}_C^{(0)}(t)\hat{\rho}_C\hat{U}_C^{(1)\dagger}(t)]$$

Where $\hat{U}_C^{(\alpha)}(t)$ is time propagator defined in terms of the effective Hamiltonian $\hat{H}_C^{(\alpha)}$ and the number of decoupling pulses. Note that $\hat{H}_C^{(\alpha)}$ here includes only degrees of freedom of the given cluster.

For free induction decay (FID) the time propagators are trivial:

$$\hat{U}_C^{(0)} = e^{-\frac{i}{\hbar}\hat{H}_C^{(0)}t};\ \hat{U}_C^{(1)} = e^{-\frac{i}{\hbar}\hat{H}_C^{(1)}t}$$

And for the generic decoupling sequence with N (even) decoupling pulses applied at $t_1, t_2 ... t_N$ we write:

$$\hat{U}^{(\alpha)}(t) = e^{-\frac{i}{\hbar}\hat{H}_C^{(\alpha)}(t_N - t_{N-1})}e^{-\frac{i}{\hbar}\hat{H}_C^{(\beta)}(t_{N-1} - t_{N-2})}...e^{-\frac{i}{\hbar}\hat{H}_C^{(\beta)}(t_2 - t_1)}e^{-\frac{i}{\hbar}\hat{H}_C^{(\alpha)}t_1}$$

Where $\alpha = 0, 1$ and $\beta = 1, 0$ accordingly (when $\alpha = 0$ one should take $\beta = 1$ and vice versa). $t = \sum_i t_i$ is the total evolution time. In sequences with odd number of pulses *N*, the leftmost propagator is the exponent of $\hat{H}_C^{(\beta)}$.

1.2.2 Generalized CCE

Instead of projecting the total Hamiltonian on the qubit levels, one may directly include the central spin degrees of freedom to each clusters. We refer to such formulation as gCCE.

In this case we write the cluster Hamiltonian as:

$$\hat{H}_C = \mathbf{SDS} + \mathbf{B}\gamma_S\mathbf{S} + \sum_{i\in C}\mathbf{SA}_i\mathbf{I}_i + \sum_{i\in C}\mathbf{I}_i\mathbf{P}_i\mathbf{I}_i + \mathbf{B}\gamma_i\mathbf{I}_i + \\ \sum_{i<j\in C}\mathbf{I}_i\mathbf{J}_{ij}\mathbf{I}_j + \sum_{a\notin C}\mathbf{SA}_a\langle\mathbf{I}_a\rangle + \sum_{i\in C,\ a\notin C}\mathbf{I}_i\mathbf{J}_{ia}\langle\mathbf{I}_a\rangle$$

And the coherence function of the cluster $L_C(t)$ is computed as:

$$L_C(t) = 0\hat{U}_C(t)\hat{\rho}_{C+S}\hat{U}_C^{\dagger}(t)1$$

Where $\hat{\rho}_{C+S} = \hat{\rho}_C \otimes \hat{\rho}_S$ is the combined initial density matrix of the bath spins' cluster and central spin.

Further details on the theoretical background are available in the references below.

CHAPTER

TWO

QUICK START

The generic workflow of the simulation includes first the generation of the spin bath in the material, and second carrying the CCE dynamics calculations for the qubit interacting with this spin bath.

2.1 Base Units

- All coupling constants are given in kHz.
- Timesteps are in millisecond (ms).
- Distances are in angstrom (A).
- Gyromagnetic ratios are given in $\mathrm{rad} \cdot \mathrm{ms}^{-1} \cdot \mathrm{G}^{-1}$.
- Quadrupole constants are given in barn ($10^{-28}\ \mathrm{m}^2$).
- Magnetic field is given in Gauss (G).

2.2 Simple Example

The simplest example includes the following steps:

1. Generate the `BathCell` object. Here we use the interface with `ase` which can effortlessly generate unit cells of many materials. As an example, we import the diamond structure.

```
import numpy as np
import pycce as pc
from ase.build import bulk

cell = pc.BathCell.from_ase(bulk('C', 'diamond', cubic=True))
```

2. Using the `BathCell` object, generate spin bath of the most common isotopes in the material. Here we generate the spin bath of size 200 Angstrom and remove one carbon, where the spin of interest is located, from the diamond crystal lattice.

```
atoms = cell.gen_supercell(200, remove=('C', [0, 0, 0]))
```

   This function returns the `BathArray` instance, which contains names of the bath spins in `'N'`, their coordinates in angstrom in `'xyz'`, empty arrays of hyperfine couplings in kHz in `'A'`, and quadrupole couplings in kHz in `'Q'` namefields. The hyperfine couplings will be generated by `Simulator` in the next step. For alternative ways to define hyperfine couplings see *Hamiltonian Parameters Input*.

3. Setup the `Simulator` using the generated spin bath. The first required argument is the total spin of the central spin, `r_bath`, `r_dipole` and `order` are convergence parameters (see the *Tutorials* for examples of convergence), `magnetic_field` is the external applied magnetic field along the z-axis, and `pulses` is the number of decoupling π pulses in Carr-Purcell-Meiboom-Gill (CPMG) sequence (0 - FID, 1 - Hahn-echo, 2 - Carr Purcell etc.).

```
calc = pc.Simulator(0.5, position=[0, 0, 0], bath=atoms, r_bath=40,
                    r_dipole=6, order=2, magnetic_field=500, pulses=1)
```

   The hyperfine couplings are automatically generated at this step assuming point dipole-dipole interactions between central spin and bath spins.

4. Compute the coherence function of the qubit using `.compute` method of the `Simulator` object with conventional CCE.

```
time_points = np.linspace(0, 2, 101)
coherence = calc.compute(time_points)
```

This function outputs Numpy array with the same shape as the `time_points` and contains the coherence function computed at each time step. By default `compute` method uses the conventional CCE to compute the coherence function.

More detailed examples of **PyCCE** usage are available in the tutorials.

CHAPTER

THREE

TUTORIALS

The examples below are available as Jupyter notebooks in the Github repository.

3.1 NV Center in Diamond

In this tutorial we will go over the main steps of running CCE calculations for the NV center in diamond with the **PyCCE** module. Those include:

- Generating the spin bath using the `pycce.BathCell` instance.
- Setting up properties of the `pycce.Simulator` instance.
- Running the calculations with the `Simulator.compute` function.

We will computethe Hahn-echo coherence function (with decoupling π-pulse applied) using the following available methods:

- Conventional CCE.
- Generalized CCE (gCCE).
- gCCE with Monte-Carlo bath sampling.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

import sys
import pycce as pc
import ase

from mpl_toolkits import mplot3d

seed = 8805
np.random.seed(seed)
np.set_printoptions(suppress=True, precision=5)
```

3.1.1 Generate nuclear spin bath

Building a supercell of nuclear spins from the `ase.Atoms` object..

Build BathCell

To generate cell it from `ase.atoms` object, use classmethod `BathCell.from_ase`.

[2]:
```
from ase.build import bulk

# Generate unitcell from ase
diamond = bulk('C', 'diamond', cubic=True)
diamond = pc.bath.BathCell.from_ase(diamond)
```

The following attributes are created with this initiallization:

- `.cell` is ndarray containing information of lattice vectors. Each **column** is a lattice vector in cartesian coordinates.
- `.atoms` is a dictionary with keys corresponding to the atom name, and each item is a list of the coordinates in cell coordinates.

[3]:
```
print('Cell\n', diamond.cell)
print('\nAtoms\n', diamond.atoms)
```

```
Cell
 [[3.57 0.   0.  ]
 [0.   3.57 0.  ]
 [0.   0.   3.57]]

Atoms
 defaultdict(<class 'list'>, {'C': [array([0., 0., 0.]), array([0.25, 0.25, 0.25]),␣
↪array([0. , 0.5, 0.5]), array([0.25, 0.75, 0.75]), array([0.5, 0. , 0.5]), array([0.75,
↪ 0.25, 0.75]), array([0.5, 0.5, 0. ]), array([0.75, 0.75, 0.25])]})
```

Populate BathCell with isotopes

The **PyCCE** package uses EasySpin database of the concentrations of all common stable isotopes with non-zero spin, however the user can proide custom concentrations.

Use function `BathCell.add_isotopes` to add one (or several) isotopes of the element. Each isotope is initiallized with `tuple` containing name of the isotope and its concentration.

Name of the isotope includes the number and element symbol, provided in the `atoms` object. As an output, the `BathCell.add_isotopes` method returns view on dictionary `BathCell.isotopes` which can be modified directly. Structure of the dictionary-like object:

```
{element_1: {isotope_1: concentration, isotope_2: concentration},
 element_2: {isotope_3: concentration ...}}
```

[4]:
```
# Add types of isotopes
diamond.add_isotopes(('13C', 0.011))
```

[4]:
```
defaultdict(dict, {'C': {'13C': 0.011}})
```

Isotopes may also be directly added to `BathCell.isotopes`. For example, below we are adding an isotope without the nuclear spin:

[5]:
```
diamond.isotopes['C']['14C'] = 0.001
```

Set z-direction of the bath (optional)

In the `Simulator` object everything is set in S_z basis. When the quantization axis of the defect does not allign with the (0, 0, 1) direction of the crystal axis, the user needs to define the axis.

If one wants to specify the complete rotation of cartesian axes, one can provide a rotation matrix to rotate the cartesian reference frame with respect to the cell coordinates by calling the `BathCell.rotate` method.

[6]:
```
# set z direction of the defect
diamond.zdir = [1, 1, 1]
```

Generate spin bath

To generate the spin bath, use the `BathCell.gen_supercell` method. First argument is the linear size of the supercell (minimum distance between any two faces of the supercell is equal to or larger than this parameter). Additional keyword arguments are `remove` and `add`.

`remove` takes a `tuple` or `list of tuples` as an argument. First element of each tuple is the name of the **atom** at that location, second element - coordinates in unit cell coordinates. If such atoms are found in the supercell, they are removed from it.

`add` takes a `tuple` or `list of tuples` as an argument. First element of each tuple is the name of the **isotope** at that location, second element - coordinates in unit cell coordinates. Each of the specified isotopes will be added in the final supercell at specified locations.

[7]:
```
# Add the defect. remove and add atoms at the positions (in cell coordinates)
atoms = diamond.gen_supercell(200, remove=[('C', [0., 0, 0]),
                                           ('C', [0.5, 0.5, 0.5])],
                              add=('14N', [0.5, 0.5, 0.5]),
                              seed=seed)
```

```
/home/onizhuk/midway/codes_development/pyCCE/pycce/bath/array.py:168: UserWarning: Spin␣
↪type for 14C was not provided and was not found in common isotopes.
  obj[n] = array[n]
```

Note, that because the `14C` isotope doesn't have a spin, **PyCCE** does not find it in common isotopes, and raises a warning. We have to provide `SpinType` for it separately, or define the properties as follows:

[8]:
```
atoms['14C'].gyro = 0
atoms['14C'].spin = 0
```

3.1.2 BathArray Structure

The bath spins are stored in the `BathArray` object - a subclass of `np.ndarray` with fixed datastructure:

- `N` field `dtype('<U16')` contains the names of bath spins.
- `xyz` field `dtype('<f8', (3,))` contains the positions of bath spins (in A).
- `A` field `dtype('<f8', (3, 3))` contains the hyperfine coupling of bath spins (in kHz).
- `Q` field `dtype('<f8', (3, 3))` contains the quadrupole tensor of bath spins (in kHz) (Relevant for spin >= 1).

All of the fields are accesible as attributes of `BathArray`. Additionally, the subarrays of the specific spins are accessible with their name as indicated above.

Upon generation of the array from the cell, the `Q` and `A` fields are empty. The Hyperfine couplings will be automatically computed by the `Simulator` object, however the quadrupole couplings must be set by the user.

The additional attributes allow one to access `SpinType` properties:

- `name` returns the spin name or array of spin names;
- `spin` returns the value of the spin or array of ones;
- `gyro` returns gyromagnetic ratios of the spins;
- `q` returns quadrupole constants of the spins;
- `detuning` returns detunings of the spins (See definition below).

For example, below we print out the attributes of the first two spins in the `BathArray`.

[9]:
```
print('Names\n', atoms[:2].N)
print('\nCoordinates\n', atoms[:2].xyz)
print('\nHyperfine tensors\n', atoms[:2].A)
print('\nQuadrupole tensors\n',atoms[:2].Q)
```

```
Names
 ['13C' '13C']

Coordinates
 [[-13.97678  -1.48178 -92.75132]
 [ 27.89939  42.17939 -45.86038]]

Hyperfine tensors
 [[[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]]

Quadrupole tensors
 [[[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]]
```

The properties of spin types (gyromagnetic ratio, quadrupole moment, etc) are stored in the `BathArray.types` attribute, which is an instance of `SpinDict` containing `SpinType` classes. For most known isotopes `SpinType` can be found in the `pycce.common_isotopes` dictionary, and is set by default (including electron spin-1/2, which is denoted by setting `N = e`). The user can add additional `SpinType` objects, by calling `BathArray.add_type` method or setting elements of `SpinDict` directly. For details of the first approach see documentation of `SpinDict.add_type` method.

The direct setting of types is rather simple. The user can set elements of `SpinDict` with `tuple`, containing:

- **(spin, gyromagnetic ratio, quadrupole moment *(optional)*, detuning *(optional)*,)**

OR

- **(isotope, spin, gyromagnetic ratio, quadrupole moment *(optional)*, detuning *(optional)*,)**

where:

- **isotope** (*str*) is the name of the given spin (same one as in `N` field of `BathArray`) to define new `SpinType` object. The key of `SpinDict` **has** to be the correct name of the spin ("isotope" field in the tuple).
- **spin** (*float*) is the total spin of the given bath spin.
- **gyromagnetic ratio** (*float*) is the gyromagnetic ratio of the given bath spin.
- **quadrupole moment** (*float*) is the quadrupole moment of the given bath spin. Relevant only when electric field gradient are used to generate quadrupole couplings for spins, stored in the `BathArray`, with `BathArray.from_efg` method.
- **detuning** (float) is an additional energy splitting for model spins, included as an extra $+\omega\hat{S}_z$ term in the Hamiltonian, where ω is the detuning.

Units of gyromagnetic ratio are rad / ms / G, quadrupole moments are given in barn, detunings are given in kHz.

[10]:
```
# Several ways to set SpinDict elements
atoms.types['14C'] = 0, 0, 0
atoms.types['Y'] = ('Y', 0, 0, 0)
atoms.types['A'] = pc.SpinType('A', 0, 0, 0)

print(atoms.types)
```

```
SpinDict(13C: (0.5, 6.7283), 14N: (1.0, 1.9338, 0.0204), 14C: (0.0, 0.0000), ...)
```

3.1.3 Simulator class

The parameters of the CCE simulator engine.

Main parameters to consider:

- `alpha` — first qubit state in S_z basis.
- `beta` — second qubit state in S_z basis.
- `position` — coordinates of the central spin.
- bath — spin bath in any specified format. Can be either:
- Instance of `BathArray` class;
- `ndarray` with `dtype([('N', np.unicode_, 16), ('xyz', np.float64, (3,))])` containing names of bath spins (same ones as stored in self.ntype) and positions of the spins in angstroms;
- The name of the `.xyz` text file containing 4 columns: name of the bath spin and xyz coordinates in A.
- `r_bath` — cutoff radius around the central spin for the bath.

- `order` — maximum size of the cluster.
- `r_dipole` — cutoff radius for the pairwise distance to consider two nuclear spins to be connected.
- `magnetic_field` — applied magnetic field. Can also be provided during the simulation run.
- `pulses` — number of pulses in Carr-Purcell-Meiboom-Gill (CPMG) sequence or the pulse sequence itself.

Additional parameters, necessary for the generalized CCE:

- `D` — Zero field splitting (ZFS) tensor of the central spin, or axial ZFS.
- `E` — if `D` is provided as axial ZFS, then `E` sets transverse ZFS.

For the full description see the documentation of the `Simulator` object.

First we setup a "mock" instance of `Simulator` to visualize the smaller part of the bath around the central spin.

[11]:
```
# Setting the runner engine
mock = pc.Simulator(spin=1, position=[0,0,0],
                    bath=atoms, r_bath=20,
                    r_dipole=6, order=3)
```

During the initiallization, depending on the provided keyword arguments several methods may be called:

- `Simulator.read_bath` is called if keyword `bath` is provided. It may take several additional arguments:
  - `r_bath` - cutoff distance from the qubit for the bath.
  - `skiprows` - if `bath` is provided as `.xyz` file, this argument tells how many rows to skip when reading the file.
  - `external_bath` - `BathArray` instance, which contains bath spins with pre defined hyperfines to be used.
  - `hyperfine` - defines the way to compute hyperfine couplings. If it is not given and `bath` doesn't contain any predefined hyperfines (`bath['A'].any() == False`) the point dipole approximation is used. Otherwise it can be an instance of `pc.Cube` object, or callable with signature `func(coord, gyro, central_gyro)`, where `coord` is an array of the bath spin coordinate, gyro is the gyromagnetic ratio of bath spin, central_gyro is the gyromagnetic ratio of the central bath spin.
  - `types` - instance of `SpinDict` or input to create one.
  - `error_range` - maximum allowed distance between positions in `bath` and `external_bath` for two spins to be considered the same.
  - `ext_r_bath` - cutoff distance from the qubit for the `external_bath`. Useful if `external_bath` has very assymetric shape and user wants to keep the precision level of the hyperfine at different distances consistent.
  - `imap` - instance of the `pc.InteractionMap` class, which contain tensor of bath spin interactions. If not provided, interactions between bath spins are assumed to be the same as one of point dipoles.

Generates `BathArray` object with hyperfine tensors to be used in the calculation.

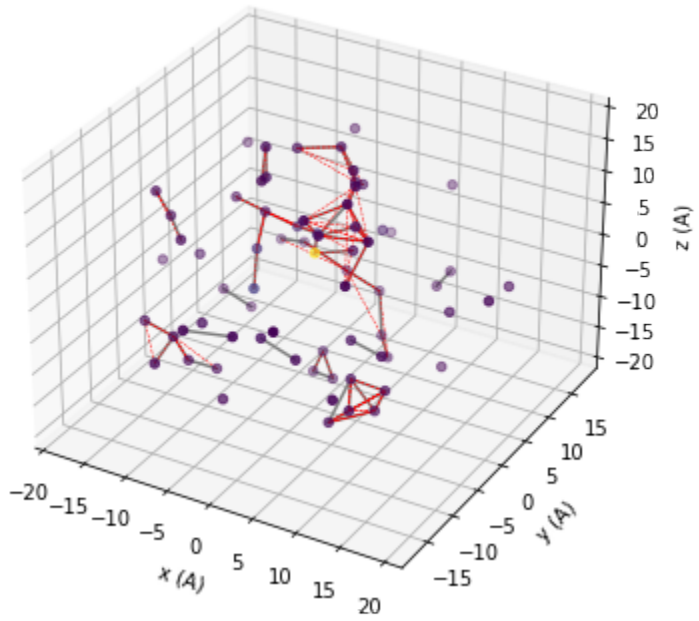
- `Simulator.generate_clusters` is called if `order` and `r_dipole` are provided. It produces `dict` object, which contains the indexes of the bath spins in the clusters.

We implemented the following procedure to determine the clusters:

Each bath spin i forms a cluster of one. Bath spins i and j form cluster of two if there is an edge between them (distance $d_{ij} \leq$ `r_dipole`). Bath spins i, j, and k form a cluster of three if enough edges connect them (e.g., there are two edges ij and jk) and so on. In general, we assume that spins $\{i..n\}$ form clusters if they form a connected graph. Only clusters up to the size indicated by the `order` parameter (equal to CCE order) are included.

We use `matplotlib` to visualize the spatial distribution of the spin bath. The grey lines show connected pairs of nuclear spins, red dashed lines show clusters of three. You can try to increase `r_dipole`, `r_bath` parameters, or increase `order` and visuallize.

[12]:
```
# add 3D axis
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(projection='3d')

# We want to visualize the smaller bath
data = mock.bath

# First plot the positions of the bath
colors = np.abs(data.A[:,2,2]/data.A[:,2,2].max())
ax.scatter3D(data.x, data.y, data.z, c=colors, cmap='viridis');
# Plot all pairs of nuclear spins, which are contained
# in the calc.clusters dictionary under they key 2
for c in mock.clusters[2]:
    ax.plot3D(data.x[c], data.y[c], data.z[c], color='grey')
# Plot all triplets of nuclear spins, which are contained
# in the calc.clusters dictionary under they key 3
for c in mock.clusters[3]:
    ax.plot3D(data.x[c], data.y[c], data.z[c], color='red', ls='--', lw=0.5)

ax.set(xlabel='x (A)', ylabel='y (A)', zlabel='z (A)');
```



Now we setup `Simulator` object for the actual simulation.

[13]:
```
# Parameters of CCE calculations engine

# Order of CCE aproximation
order = 2
# Bath cutoff radius
```

(continues on next page)

(continued from previous page)

```
r_bath = 40  # in A
# Cluster cutoff radius
r_dipole = 8  # in A
# position of central spin
position = [0, 0, 0]
# Qubit levels (in Sz basis)
alpha = [0, 0, 1]; beta = [0, 1, 0]
# ZFS Parametters of NV center in diamond
D = 2.88 * 1e6 # in kHz
E = 0 # in kHz
```

The code already knows the properties of the most common nuclear spins and of elecron spin (accessible under the name `'e'`), however the user can provide their own by providing `types` keyword argument of `Simulator.read_bath` method. The way to initiallize `SpinType` objects is the same as in `SpinDict` above.

[14]:
```
# The code already knows most exsiting isotopes.
#              Bath spin types
#              name    spin    gyro       quadrupole (for s>1/2)
spin_types = [('14N',  1,      1.9338,    20.44),
              ('13C',  1 / 2,  6.72828),
              ('29Si', 1 / 2, -5.3188),]
```

Setting the `Simulator` object

All of the kwargs can be provided at the moment of creation. If all of the kwargs are provided, several methods of the `Simulator` class are called:

- `Simulator.read_bath`;
- `Simulator.generate_clusters`.

The details are available in the `Simulator` methods description.

[15]:
```
# Setting the runner engine
calc = pc.Simulator(spin=1, position=position,
                    alpha=alpha, beta=beta,
                    types=spin_types, D=D, E=E,
                    bath=atoms, r_bath=r_bath,
                    r_dipole=r_dipole, order=order)
```

Taking advantage of subclassing `np.ndarray` we can change *in situ* the quadrupole tensor of the Nitrogen nuclear spin.

[16]:
```
nspin = calc.bath
# Set model quadrupole tensor at N atom
quad = np.asarray([[-2.5, 0, 0],
                   [0, -2.5, 0],
                   [0, 0,  5.0]]) * 1e3 * 2 * np.pi

nspin['Q'][nspin['N'] == '14N'] = quad
```

Note, that we need to apply the boolean mask **second** because of how structured arrays work.

Compute coherence function with conventional CCE

The general interface to compute any property with PyCCE is implemented through the `Simulator.compute` method. It takes two keyword arguments to determine which quantity to compute and how:

- `method` can take 'cce' or 'gcce' values, and determines which method to use - conventional or generalized CCE.
- `quantity` can take 'coherence' or 'noise' values, and determines which quantity to compute - coherence function or autocorrelation function of the noise.

Each of the methods can be performed with Monte Carlo bath state sampling (if `nbstates` keyword is non zero) and with interlaced averaging (If `interlaced` keyword is set to `True`).

In the first example we use the conventional CCE method without Monte Carlo bath state sampling. In the conventional CCE method the Hamiltonian is projected on the qubit levels, and the coherence is computed from the overlap of the bath evolution, entangled with two different qubit states.

The conventional CCE requires one argument:

- `timespace` — time points at which the coherence function is computed.

Additionally, one can provide the following arguments now, instead of when initiallizing `Simulator` object:

- `pulses` — number of pulses in CPMG sequence (0 - FID, 1 - HE etc., default 0) or explicit sequence of pulses as `Sequence` class instance.
- `mangetic_field` — array of the magnetic field, default (0, 0, 0).

[17]:
```
# Time points
time_space = np.linspace(0, 2, 201)  # in ms
# Number of pulses in CPMG seq (0 = FID, 1 = HE)
N = 1
# Mag. Field (Bx By Bz)
B = np.array([0, 0, 500])  # in G

l_conv = calc.compute(time_space, pulses=N, magnetic_field=B,
                      method='cce', quantity='coherence', as_delay=False)
```

[18]:
```
%%timeit
calc.compute(time_space, pulses=N, magnetic_field=B,
             method='cce', quantity='coherence', as_delay=False)
```

```
1.09 s ± 1.69 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Generalized CCE (gCCE)

In contrast to the conventional CCE method, in generalized CCE arpproach each cluster includes the central spin explicitly.

`Simulator` can take `pulses` argument as an actual pulse sequence with a `list` of `tuples` or `Sequence` class instance. In each tuple the first entry is the axis, the second entry - rotational angle.

For example:

```
[('x', np.pi), ('y', np.pi), ('x', np.pi), ('y', np.pi)]
```

will define XY-4 pulse sequence.

An integer number to define the number of pulses is also accepted as in the case of conventional CCE. If the integer is provided, the code assumes the CPMG sequence.

[19]:
```
# Hahn-echo pulse sequence
pulse_sequence = [('x', np.pi)]

# Calculate coherence function
l_generatilze = calc.compute(time_space, magnetic_field=B,
                             pulses=pulse_sequence,
                             method='gcce', quantity='coherence')
```

[20]:
```
%%timeit
calc.compute(time_space, magnetic_field=B,
             pulses=pulse_sequence, D=D, E=E,
             method='gcce',
             quantity='coherence')
```

```
3.31 s ± 2.32 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

gCCE with random sampling of bath states

Using this approach, one may carry out generalized CCE calculations for the set of random bath states. This functionality can be turned on by by setting the keyword argument `nbstates` to a number of bath states to sample over. Recommended number of bath states is above 100, but the convergence should be checked for each system. Note, that this computation is roughly `nbstates` times longer than an equivalent generalized CCE calculation, as it computes everything `nbstates` times.

For details see `help(calc.compute)`.

[21]:
```
# Number of random bath states to sample over
n_bath_states = 20

# Calculate coherence function
l_gcce = calc.compute(time_space, magnetic_field=B,
                      pulses=pulse_sequence,
                      nbstates=n_bath_states,
                      method='gcce', quantity='coherence', seed=seed)
```

[22]:
```
%%timeit
n_bath_states = 5
calc.compute(time_space, magnetic_field=B,
             pulses=pulse_sequence,
             nbstates=n_bath_states,
             method='gcce', quantity='coherence', seed=seed)
```

```
19.2 s ± 12.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Take a look at the results of three different methods, and check that they produce similar coherence decay. Note that the results obtained using gCCE with bath states sampling deviates from other ones (generalized and conventional CCE), as the chosen number of states (20) is not enough to converge.

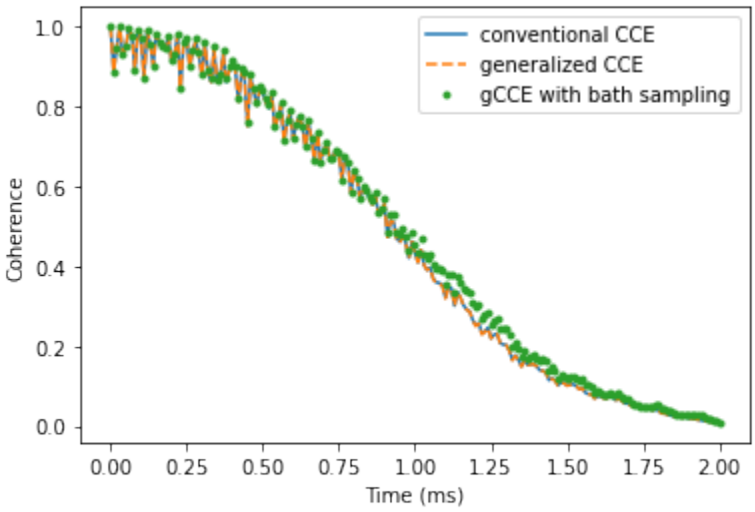
[23]:
```
plt.plot(time_space, l_conv.real,
         label='conventional CCE')
plt.plot(time_space, l_generatilze.real,
         label='generalized CCE', ls='--')
plt.plot(time_space, l_gcce.real,
```

(continues on next page)

(continued from previous page)

```
        label='gCCE with bath sampling', ls='', marker='.')
plt.legend()
plt.xlabel('Time (ms)')
plt.ylabel('Coherence')
```

[23]:
```
Text(0, 0.5, 'Coherence')
```



Convergence parameters

Having confirmed that all methods produce the same results, we check the convergence of the conventional CCE with respect to `order`, `r_bath`, `r_dipole` parameters of the `Simulator` object.

First, define all of the parameters.

[24]:
```
parameters = dict(
    order=2, # CCE order
    r_bath=40,  # Size of the bath in A
    r_dipole=8,  # Cutoff of pairwise clusters in A
    position=[0, 0, 0], # Position of central Spin
    alpha=[0, 0, 1],
    beta=[0, 1, 0],
    pulses = 1, # N pulses in CPMG sequence
    magnetic_field=[0,0,500]
) # Qubit levels)

time_space = np.linspace(0, 2, 201)  # Time points in ms
```

We can define a little helper function to streamline the process. Note that resetting the parameters automatically recomputes the properties of the bath.

[25]:
```
def runner(variable, values):
    invalue = parameters[variable]
    calc = pc.Simulator(spin=1, bath=atoms, **parameters)
    ls = []
```

(continues on next page)

(continued from previous page)

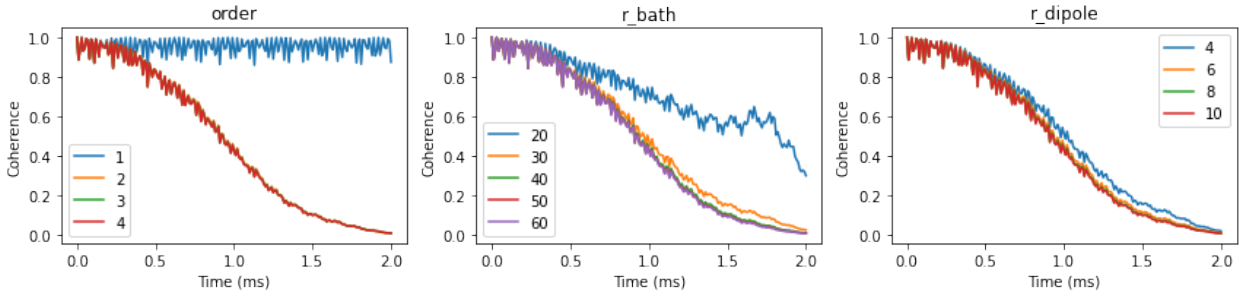
```
    for v in values:
        setattr(calc, variable, v)
        l = calc.compute(time_space, method='cce',
                         quantity='coherence')

        ls.append(l.real)

    parameters[variable] = invalue
    ls = pd.DataFrame(ls, columns=time_space, index=values).T
    return ls
```

Now we can compute the coherence function at different values of the parameters:

[26]:
```
orders = runner('order', [1, 2, 3, 4])
rbs = runner('r_bath', [20, 30, 40, 50, 60])
rds = runner('r_dipole', [4, 6, 8, 10])
```

We can visualize the convergence of the coherence function with respect to different parameters:

[27]:
```
fig, axes = plt.subplots(1, 3, figsize=(12, 3))
orders.plot(ax=axes[0], title='order')
rbs.plot(ax=axes[1], title='r_bath')
rds.plot(ax=axes[2], title='r_dipole')
for ax in axes:
    ax.set(xlabel='Time (ms)', ylabel='Coherence')
fig.tight_layout()
```



3.2 VV in SiC

An example of computing Free Induction Decay (FID) and Hahn-echo (HE) with hyperfine couplings from GIPAW for axial and basal divacancies.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import sys
import ase
import pandas as pd
import warnings
sys.path.append('/home/onizhuk/midway/codes_development/pyCCE')
```

(continues on next page)

(continued from previous page)

```
import pycce as pc

np.set_printoptions(suppress=True, precision=5)
warnings.simplefilter("ignore")

seed = 8805
```

3.2.1 Axial kk-VV

First we compute FID and HE for axial divacancy.

Build BathCell from the ground

One can set up an `BathCell` instance by providing the parameters of the unit cell, or `cell` argument as 3x3 tensor, where each column defines a, b, c unit cell vectors in cartesian coordinates.

In this tutorial we use the first approach.

[2]:
```
# Set up unit cell with (a, b, c, alpha, beta, gamma)
sic = pc.BathCell(3.073, 3.073, 10.053, 90, 90, 120, 'deg')
# z axis in cell coordinates
sic.zdir = [0, 0, 1]
```

Next, user has to define positions of atoms in the unit cell. It is done with `BathCell.add_atoms` function. It takes an unlimited number of arguments, each argument is a tuple. First element of the tuple is the name of the atom, second - list of xyz coordinates either in cell units (if keyword `type='cell'`, default value) or in Angstrom (if keyword `type='angstrom'`). Returns `BathCell.atoms` dictionary, which contains list of coordinates for each type of elements.

[3]:
```
# position of atoms
sic.add_atoms(('Si', [0.00000000, 0.00000000, 0.1880]),
              ('Si', [0.00000000, 0.00000000, 0.6880]),
              ('Si', [0.33333333, 0.66666667, 0.4380]),
              ('Si', [0.66666667, 0.33333333, 0.9380]),
              ('C', [0.00000000, 0.00000000, 0.0000]),
              ('C', [0.00000000, 0.00000000, 0.5000]),
              ('C', [0.33333333, 0.66666667, 0.2500]),
              ('C', [0.66666667, 0.33333333, 0.7500]));
```

Two types of isotopes present in SiC: ^{29}Si and ^{13}C. We add this information with the `BathCell.add_isotopes` function. The code knows most of the concentrations, so this step is actually unnecessary. If no isotopes is provided, the natural concentration of common magnetic isotopes is assumed.

[4]:
```
# isotopes
sic.add_isotopes(('29Si', 0.047), ('13C', 0.011))

# defect position in cell units
vsi_cell = [0, 0, 0.1880]
vc_cell = [0, 0, 0]

# Generate bath spin positions
```

(continues on next page)

(continued from previous page)

```
atoms = sic.gen_supercell(200, remove=[('Si', vsi_cell),
                                       ('C', vc_cell)],
                          seed=seed)
```

Read Quantum Espresso output

PyCCE provides a helper function `read_qe` in `pycce.io` module to read hyperfine couplings from quantum espresso output. `read_qe` takes from 1 to 3 positional arguments:

- `pwfile` name of the pw input/output file;
- `hyperfine` name of the gipaw output file containing hyperfine couplings;
- `efg` name of the gipaw output file containing electric field tensor calculations.

During its call, `read_qe` will read the cell matrix in pw file and apply it to the coordinates is necessary. However, usually we still need to rotate and translate the Quantum Espresso supercell to allign it with our `BathArray`. To do so we can provide additional keywords arguments `center` and `rotation_matrix`. `center` is the position of (0, 0, 0) point in coordinates of pw file, and `rotation_matrix` is rotation matrix which aligns z-direction of the GIPAW output. This matrix, acting on the (0, 0, 1) in Cartesian coordinates of GIPAW output should produce (a, b, c) vector, alligned with zdirection of the BathCell. Keyword argument `rm_style` shows whether `rotation_matrix` contains coordinates of new basis set as rows (`'row'`, common in physics) or columns (`'col'`, common in math).

[5]:
```
# Prepare rotation matrix to alling with z axis of generated atoms
# This matrix, acting on the [0, 0, 1] in Cartesian coordinates of GIPAW output
# Should produce [a, b, c] vector, alligned with zdirection of the BathCell
M = np.array([[0, 0, -1],
              [0, -1, 0],
              [-1, 0, 0]])

# Position of (0,0,0) point in cell coordinates
center = [0.6, 0.5, 0.5]
# Read GIPAW results
exatoms = pc.read_qe('axial/pw.in',
                     hyperfine='axial/gipaw.out',
                     center=center, rotation_matrix=M,
                     rm_style='col',
                     isotopes={'C':'13C', 'Si':'29Si'})
```

`pc.read_qe` produces instance of `BathArray`, with names of bath spins as the most common isotopes of the following elements (if keyword `isotopes` set to `None`) or from the mapping provided by the `isotopes` argument.

Set up CCE Simulator

In this example we set up a bare `Simulator` and add properties of the spin bath later.

[6]:
```
# Setting up CCE calculations
pos = sic.to_cartesian(vsi_cell)
CCE_order = 2
r_bath = 40
r_dipole = 8
B = np.array([0, 0, 500])
```

(continues on next page)

(continued from previous page)

```
calc = pc.Simulator(1, pos, alpha=[0, 0, 1], beta=[0, 1, 0], magnetic_field=B)
```

Function `Simulator.read_bath` can be called explicitly to initiallize spin bath. Additional keyword argument `external_bath` takes instance of `BathArray` with hyperfine couplings read from Quantum Espresso. The program then finds the spins with the same name at the same positions (within the range defined by `error_range` keyword argument) in the total bath and sets their hyperfine couplings.

Finally, we call `Simulator.generate_clusters` to find the bath spin clusters in the provided bath.

[7]:
```
calc.read_bath(atoms, r_bath, external_bath=exatoms);
calc.generate_clusters(CCE_order, r_dipole=r_dipole);
```

FID with DFT hyperfine couplings

We provide `pulses` argument directly to the compute function instead of during initiallization of the `Simulator` object.

[8]:
```
time_space = np.linspace(0, 0.01, 501)
N = 0

ldft = calc.compute(time_space, pulses=N, as_delay=False)
```

[9]:
```
len(calc.pulses)
```

[9]:
```
0
```

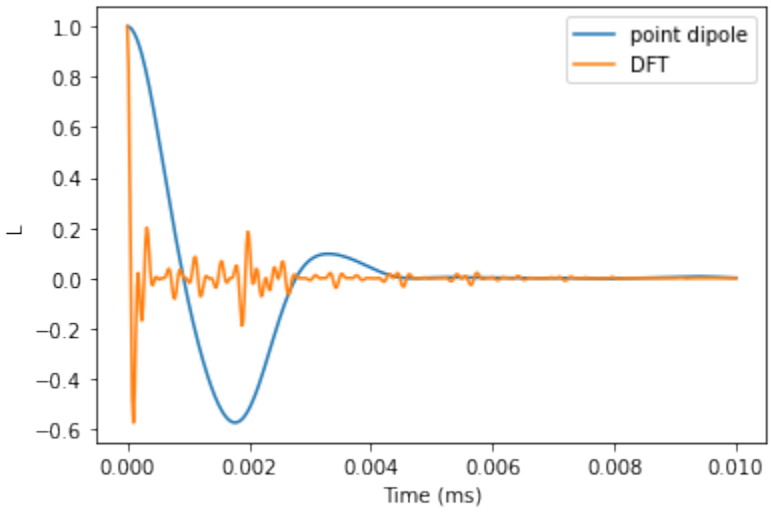
FID with hyperfine couplings from point dipole approximation

[10]:
```
pdcalc = pc.Simulator(1, pos, alpha=[0, 0, 1], beta=[0, 1, 0], magnetic_field=B,
                      bath=atoms, r_bath=r_bath, order=CCE_order, r_dipole=r_dipole)
lpd = pdcalc.compute(time_space, pulses=N, as_delay=False)
```

Plot the results and and verify that the predictions are significantly different.

[11]:
```
plt.plot(time_space, lpd.real, label='point dipole')
plt.plot(time_space, ldft.real, label='DFT')
plt.legend()
plt.xlabel('Time (ms)')
plt.ylabel('L')
```

[11]:
```
Text(0, 0.5, 'L')
```



Hahn-echo comparison

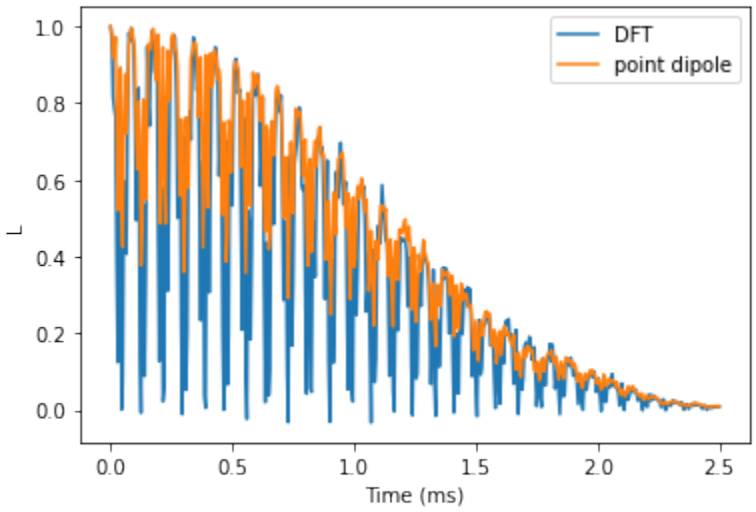
Now we compare the predictions for Hahn-echo signal with different hyperfine couplings.

[12]:
```
he_time = np.linspace(0, 2.5, 501)
B = np.array([0, 0, 500])
N = 1

he_ldft = calc.compute(he_time, magnetic_field=B, pulses=N, as_delay=False)
he_lpd = pdcalc.compute(he_time, magnetic_field=B, pulses=N, as_delay=False)
```

Plot the results and compare. We observe that electron spin echo modulations differ significantly, while the observed decay is about the same.

[13]:
```
plt.plot(he_time, he_ldft.real, label='DFT')
plt.plot(he_time, he_lpd.real, label='point dipole')

plt.legend()
plt.xlabel('Time (ms)')
plt.ylabel('L')
```

[13]:
```
Text(0, 0.5, 'L')
```



3.2.2 Basal kh-VV in SiC

The basal divacancy's Hamiltonian includes both D and E terms, which allows for mixing between +1 and -1 spin levels at zero field.

Thus, either the generalized CCE should be used, or additional perturbational Hamiltonian terms are to be added. Here we consider the generalized CCE framework.

First, prepare rotation matrix for DFT results. The same supercell was used to compute hyperfine couplings, however z-axis of the electron spin qubit is aligned with Si-C bond, therefore we will need to rotate the DFT supercell accordingly.

[14]:
```
# Coordinates of vacancies in cell coordinates (note that Vsi is not located in the␣
↪first unitcell)
vsi_cell = -np.array([1 / 3, 2 / 3, 0.0620])
vc_cell = np.array([0, 0, 0])

sic.zdir = [0, 0, 1]

# Rotation matrix for DFT supercell
R = pc.rotmatrix([0, 0, 1], sic.to_cartesian(vsi_cell - vc_cell))
```

Total spin bath can be initiallized by simply setting z direction of the `BathCell` object.

[15]:
```
sic.zdir = vsi_cell - vc_cell

# Generate bath spin positions
sic.add_isotopes(('29Si', 0.047), ('13C', 0.011))
atoms = sic.gen_supercell(200, remove=[('Si', vsi_cell),
                                       ('C', vc_cell)],
                          seed=seed)
```

Read DFT results with `read_qe` function. To rotate in the correct frame we need to apply both changes of basis consequently

[16]:
```
M = np.array([[0, 0, -1],
              [0, -1, 0],
              [-1, 0, 0]])

# Position of (0,0,0) point in cell coordinates
center = np.array([0.59401, 0.50000, 0.50000])

# Read GIPAW results
exatoms = pc.read_qe('basal/pw.in',
                     hyperfine='basal/gipaw.out',
                     center=center, rotation_matrix=(M.T @ R),
                     rm_style='col',
                     isotopes={'C':'13C', 'Si':'29Si'})
```

To check whether our rotations produced correct results we can find the indexes of the `BathArray` and DFT output with `pc.same_bath_indexes` function. It returns a tuple, containing the indexes of elements in the two `BathArray` instances with the same position and name. First element of the tuple - indexes of first argument, second - of the second. For that we generate supecell with `BathCell` class, containing 100% isotopes, and count the number of found indexes - iut should be equal to the size of DFT supercell.

[17]:
```
# isotopes
sic.add_isotopes(('29Si', 1), ('13C', 1))
allcell = sic.gen_supercell(50, remove=[('Si', vsi_cell),
                                        ('C', vc_cell)],
                            seed=seed)

indexes, ext_indexes = pc.same_bath_indexes(allcell, exatoms, 0.2, True)
print(f"There are {indexes.size} same elements."
      f" Size of the DFT supercell is {exatoms.size}")
```

```
There are 1438 same elements. Size of the DFT supercell is 1438
```

Setting up calculations

Now we can safely setup calculations of coherence function with DFT couplings. We will compare results with or without bath state sampling.

[18]:
```
D = 1.334 * 1e6
E = 0.0184 * 1e6
magnetic_field = 0

calc = pc.Simulator(1, pos, bath=atoms, external_bath=exatoms, D=D, E=E,
                    magnetic_field=magnetic_field,
                    r_bath=r_bath, order=CCE_order, r_dipole=r_dipole)
```

The code automatically picks up the two lowest eigenstates of the central spin hamiltonian as qubit states.

[19]:
```
print(calc)
```

```
Simulator for spin-1.
alpha: 0
beta: 1
gyromagnetic ratio: -17608.59705 kHz * rad / G
```

(continues on next page)

(continued from previous page)

```
zero field splitting:
array([[-426266.66667,       0.     ,       0.     ],
       [      0.     , -463066.66667,       0.     ],
       [      0.     ,       0.     ,  889333.33333]])
magnetic field:
array([0., 0., 0.])

Parameters of cluster expansion:
r_bath: 40
r_dipole: 8
order: 2

Bath consists of 761 spins.

Clusters include:
761  clusters of order 1.
1870  clusters of order 2.
```

We can use the `Simulator.eigenstates` function to generate qubit states.

[20]:
```
calc.eigenstates(D=D, E=E, alpha=-1, beta=0)
print(f'0 state: {calc.alpha}; 1 state: {calc.beta}')
```

```
0 state: [-0.70711+0.j  0.     +0.j -0.70711+0.j]; 1 state: [ 0.+0.j -1.+0.j  0.+0.j]
```

FID Decay

Now, use the generalized CCE to compute FID decay of the coherence function at different CCE orders.

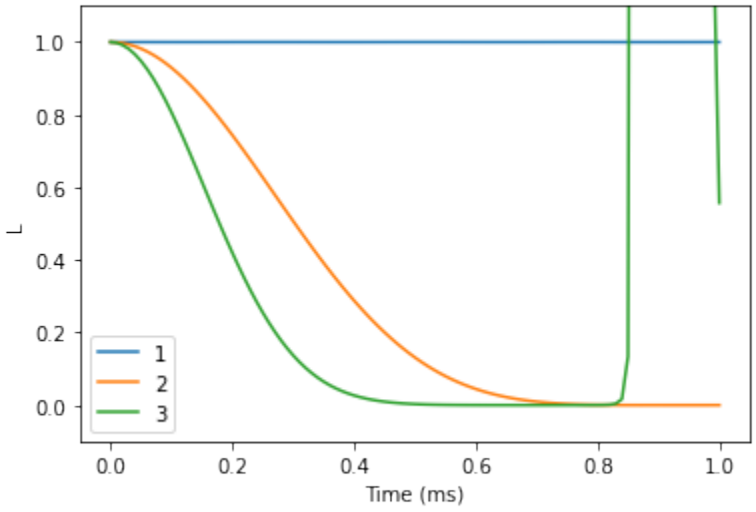
[21]:
```
N = 0 # Number of pulses
time_space = np.linspace(0, 1, 101) # Time points at which to compute

orders = [1, 2, 3]
lgen = []

r_bath = 30
r_dipole = 8

calc = pc.Simulator(1, pos, bath=atoms, external_bath=exatoms,
                    D=D, E=E, pulses=N, alpha=-1, beta=0,
                    r_bath=r_bath, r_dipole=r_dipole)
```

[22]:
```
for o in orders:
    calc.generate_clusters(o)
    l = calc.compute(time_space, method='gcce',
                     quantity='coherence', as_delay=False)

    lgen.append(np.abs(l))

lgen = pd.DataFrame(lgen, columns=time_space, index=orders).T
```

We see that the results do not converge, but rather start to diverge. Bath sampling (setting `nbstates` to some value) will help with that.

[23]:
```
lgen.plot()
plt.xlabel('Time (ms)')
plt.ylabel('L')
plt.ylim(-0.1,1.1)
```

[23]:
```
(-0.1, 1.1)
```



Note, that this approach is nbstates times more expensive than the gCCE, therefore the following calculation will take a couple of minutes.

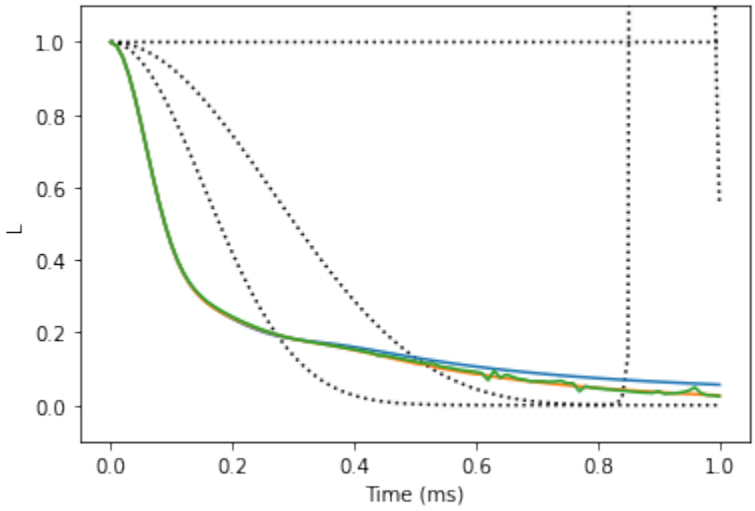
[24]:
```
orders = [1, 2, 3]
lgcce = []

r_bath = 30
r_dipole = 6

for o in orders:
    calc.generate_clusters(o)

    l = calc.compute(time_space, nbstates=30, seed=seed,
                     method='gcce',
                     quantity='coherence', as_delay=False)

    lgcce.append(np.abs(l))

lgcce = pd.DataFrame(lgcce, columns=time_space, index=orders).T
```

Compare the two results

The gCCE results are converged at 1st order. Note that we used only a small number of bath states (30), thus the calculations are not converged with respect to the number of bath states. Calculations with higher number of bath states (~100) will produce correct results.

[25]:
```
plt.plot(lgen, color='black', ls=':')
plt.plot(lgcce)

plt.xlabel('Time (ms)')
plt.ylabel('L')
plt.ylim(-0.1,1.1)
```

[25]:
```
(-0.1, 1.1)
```



Hahn-echo decay

Using the similar procedure to the one used for FID, we can compute the Hahn-echo decay.

[26]:
```
r_bath = 40
r_dipole = 8
order = 2
N = 1 # Number of pulses

calc = pc.Simulator(1, pos, bath=atoms, external_bath=exatoms,
                    pulses=N, D=D, E=E, alpha=-1, beta=0,
                    r_bath=r_bath, order=order, r_dipole=r_dipole)
```

[27]:
```
ts = np.linspace(0, 4, 101) # time points (in ms)
```

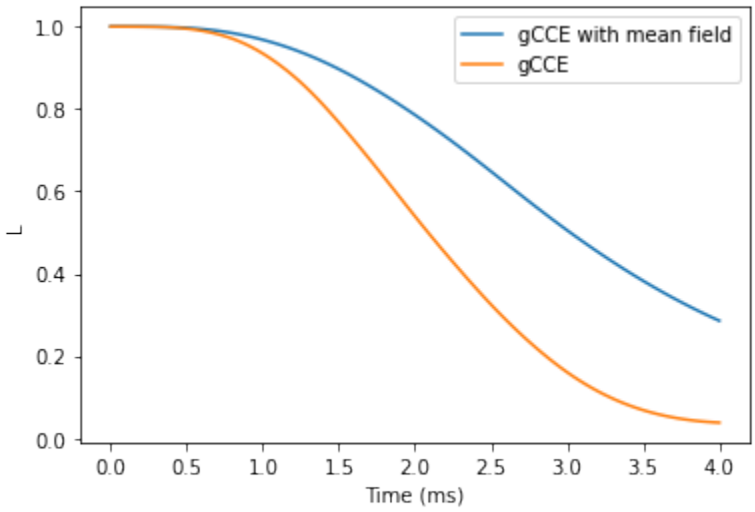
[28]:
```
helgen = calc.compute(ts,
                      method='gcce', quantity='coherence')
```

Note the number of nbstates leads to significantly increased time of the calculation. The interface to mpi implementation is provided with keyword `parallel_states`. However it requires `mpi4py` installed and a run on several cores.

[29]:
```
helgcce = calc.compute(time_space, nbstates=30, seed=seed,
                       method='gcce', quantity='coherence')
```

[30]:
```
plt.plot(ts, helgcce, label='gCCE with mean field')
plt.plot(ts, helgen, label='gCCE')

plt.legend()
plt.xlabel('Time (ms)')
plt.ylabel('L')
```

[30]:
```
Text(0, 0.5, 'L')
```



3.3 Shallow donor in Si

Example of more complicated simulations, in which we compare the coherence predicted with point-dipole hyperfine couplings and one obtained using the hyperfines from model wavefunction of the shallow donor in Si (P:Si).

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import sys
import ase

import pycce as pc

seed = 8800
np.set_printoptions(suppress=True, precision=5)
```

First, as always, generate spin bath with `BathCell` instance. To get parameters we use ase interface. It allows to conveniently read structure files of any type.

[2]:
```
# Generate unitcell from ase
from ase import io
s = io.read('si.cif')
s = pc.bath.BathCell.from_ase(s)
```

(continues on next page)

(continued from previous page)

```
# Add types of isotopes
s.add_isotopes(('29Si', 0.047))
# set z direction of the defect
s.zdir = [1, 1, 1]
# Generate supercell
atoms = s.gen_supercell(200, remove=[('Si', [0., 0., 0.])], seed=seed)
```

3.3.1 Calculations with point dipole hyperfine couplings

Here we compute Hahn-echo decay with point dipole hyperfine couplings. All of the parameters are converged, however it never hurts to check!

[3]:
```
# Parameters of CCE calculations engine

# Order of CCE aproximation
CCE_order = 2
# Bath cutoff radius
r_bath = 80  # in A
# Cluster cutoff radius
r_dipole = 10  # in A

# position of central spin
position = [0, 0, 0]
# Qubit levels (in Sz basis)
alpha = [0, 1]; beta = [1, 0]
# Mag. Field (Bx By Bz)
B = np.array([0, 0, 1000])  # in G
# Number of pulses in CPMG seq (0 = FID, 1 = HE etc)
pulses = 1

# Setting the runner engine
calc = pc.Simulator(spin=0.5, position=position, alpha=alpha, beta=beta,
                    bath=atoms, r_bath=r_bath, magnetic_field=B, pulses=pulses,
                    r_dipole=r_dipole, order=CCE_order)
```

[4]:
```
# Time points
time_space = np.linspace(0, 2, 201)  # in ms
```

For comparison, we compute both with generalized CCE and usual CCE coherence. Note a relatively large bath (`r_bath = 80`), so the calculations will take some time.

[5]:
```
l_cce = calc.compute(time_space, method='CCE')
l_gen = calc.compute(time_space, method='gCCE')
```

3.3.2 Hyperfine couplings of the shallow donor

We compute the hyperfine couplings of the shallow donnor, following the formulae by Rogerio de Sousa and S. Das Sarma (Phys Rev B 68, 115322 (2003)).

[6]:
```
# PHYSICAL REVIEW B 68, 115322 (2003)
n = 0.81
a = 25.09

def factor(x, y, z, n=0.81, a=25.09, b=14.43):
    top = np.exp(-np.sqrt(x**2/(n*b)**2 + (y**2 + z**2)/(n*a)**2))
    bottom = np.sqrt(np.pi * (n * a)**2 * (n * b) )

    return top / bottom

def contact_si(r, gamma_n, gamma_e=pc.ELECTRON_GYRO, a_lattice=5.43, nu=186, n=0.81,␣
↪a=25.09, b=14.43):
    k0 = 0.85 * 2 * np.pi / a_lattice
    pre = 8 / 9 * gamma_n * gamma_e * pc.HBAR * nu
    xpart = factor(r[0], r[1], r[2], n=n, a=a, b=b) * np.cos(k0 * r[0])
    ypart = factor(r[1], r[2], r[0], n=n, a=a, b=b) * np.cos(k0 * r[1])
    zpart = factor(r[2], r[0], r[1], n=n, a=a, b=b) * np.cos(k0 * r[2])
    return pre * (xpart + ypart + zpart) ** 2
```

We make a copy of the `BathArray` object, and set up their hyperfines according to the reference above.

[7]:
```
newatoms = atoms.copy()

# Generate hyperfine from point dipole
newatoms.from_point_dipole(position)

# Following PRB paper
newatoms['A'][newatoms.dist() < n*a] = 0
newatoms['A'] += np.eye(3)[np.newaxis,:,:] * contact_si(newatoms['xyz'].T, newatoms.
↪types['29Si'].gyro)[:,np.newaxis, np.newaxis]
```

Now we set up a `Simulator` object. Because hyperfines in `newatoms` are nonzero, they are **not** approximated as the ones of point dipole.

[8]:
```
calc = pc.Simulator(spin=0.5, position=position, alpha=alpha, beta=beta,
                    bath=newatoms, r_bath=r_bath, magnetic_field=B, pulses=pulses,
                    r_dipole=r_dipole, order=CCE_order)
```

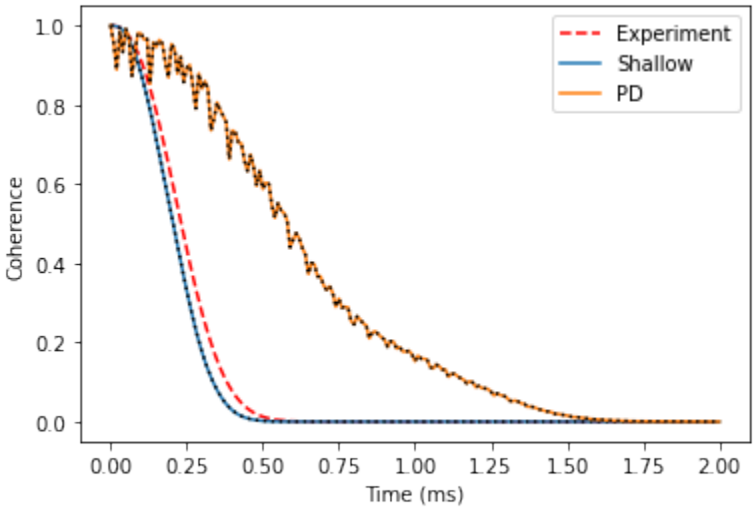
[9]:
```
shallow_l_cce = calc.compute(time_space, method='CCE')
shallow_l_gen = calc.compute(time_space, method='gCCE')
```

3.3.3 Compare the results

We find that the point dipole gives a poor agreement with the experimental data. Model wavefunction, on the countrary, produces great agreement with the experimental coherence time from work of Eisuke Abe et al. (Phys Rev B 82, 121201(R) (2010)).

[10]:
```
t2exp = 0.27 # Experimental T2 from PhysRevB.82.121201
decay = lambda t: np.exp(-(t/t2exp)**2.4)
plt.plot(time_space, decay(time_space), color='red', label='Experiment', ls='--')

plt.plot(time_space, shallow_l_cce.real, label='Shallow')
plt.plot(time_space, shallow_l_gen.real, ls=':', c='black')

plt.plot(time_space, l_cce.real, label='PD')
plt.plot(time_space, l_gen.real, ls=':', c='black')
plt.legend();
plt.xlabel('Time (ms)')
plt.ylabel('Coherence')
```

[10]:
```
Text(0, 0.5, 'Coherence')
```



Interesting to note - the decay depends significantly on the orientation of the magnetic field. You can check it yourself!

3.4 Correlation function

In this tutorial we will compute the coherence function of the NV Center in diamond and then reproduce it from the correlation function of the noise.

The correlation function $C(t)$ of the effective magnetic field (noise) along the z-axis can be defined as follows:

$C(t) = \langle \beta_z(t) \beta_z(0) \rangle$

With β_z given as:

$\beta_z(t) = U^\dagger(t) \left(\sum_{\{I\}} A_{zz} I_z \right) U(t)$

Where $U(t)$ is time propagator.

Within the CCE formalism, the correlation function is computed as:

$C(t) = \sum_{\{i\}} \tilde{C}_{\{i\}}(t) + \sum_{\{ij\}} \tilde{C}_{\{ij\}}(t) \ + \ ...$

With contributions computed as:

$\tilde{C}_\nu(t) = C_\nu(t) - \sum_{\nu' \subset \nu} \tilde{C}_{\nu'}(t)$

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

import sys

import pycce as pc
import ase

seed = 42055
np.set_printoptions(suppress=True, precision=5)
```

3.4.1 Generate nuclear spin bath

Building a `BathArray` of nuclear spins from the `ase.Atoms` object.

[2]:
```
from ase.build import bulk

# Generate unitcell from ase
diamond = bulk('C', 'diamond', cubic=True)
diamond = pc.bath.BathCell.from_ase(diamond)
# Add types of isotopes
diamond.add_isotopes(('13C', 0.011))
# set z direction of the defect
diamond.zdir = [1, 1, 1]
# Add the defect. remove and add atoms at the positions (in cell coordinates)
atoms = diamond.gen_supercell(200, remove=[('C', [0., 0, 0]),
                                           ('C', [0.5, 0.5, 0.5])],
                              add=('14N', [0.5, 0.5, 0.5]),
                              seed=seed)
```

Next, we define all of the parameters of the simulation. We are interested in the very specific regime, when all nearby nuclear spins are removed. To achieve this goal we define an `inner = 20` parameter, and remove all nuclear spins within this radius.

[3]:
```
position = np.array([0, 0, 0])
inner = 20
smallatoms = atoms[atoms.dist(position) >= inner]

parameters = dict(
    order=2, # CCE order
    r_bath=60,  # Size of the bath in A
    r_dipole=6,  # Cutoff of pairwise clusters in A
    position=position, # Position of central Spin
    alpha=[0, 0, 1], # 0 qubit state
    beta=[0, 1, 0], # 1 qubit state
    magnetic_field = 500, # magnetic field along z-axis
```

(continues on next page)

(continued from previous page)

```
    pulses=1 # N pulses in CPMG sequence
) # Qubit levels

ts = np.linspace(0, 2.5, 1001)  # Time points in ms
```

3.4.2 Coherence calculations

Next, we set up `Simulator` objects and check convergence with respect to the CCE order.

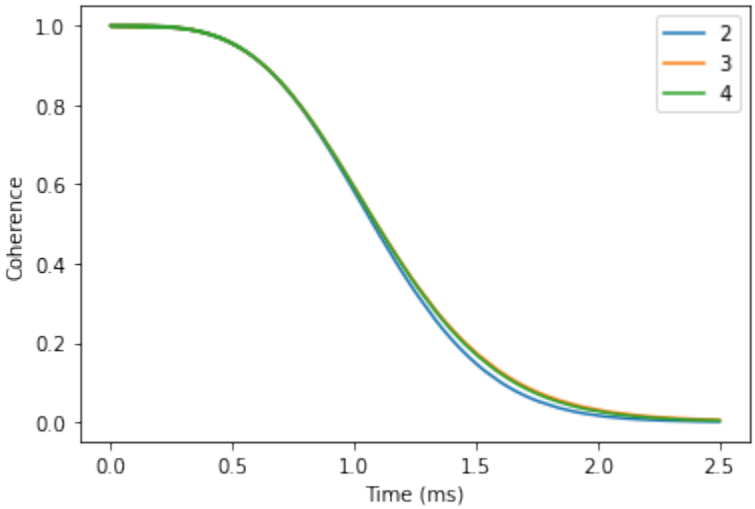
[4]:
```
calc = pc.Simulator(spin=1, bath=smallatoms, **parameters)
```

[5]:
```
orders = [2, 3, 4]
coh = {}
for o in orders:
    calc.generate_clusters(o)
    coh[o] = calc.compute(ts, method='cce', quantity='coherence')
coh = np.abs(pd.DataFrame(coh, index=ts))
coh.index.name = 'Time (ms)'
```

Visually verify the convergence.

[6]:
```
coh.plot()
plt.ylabel('Coherence')
```

[6]:
```
Text(0, 0.5, 'Coherence')
```



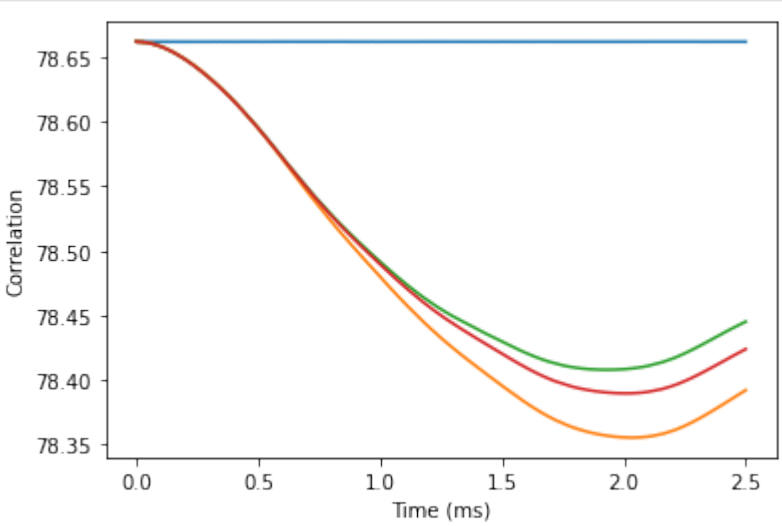
3.4.3 Noise calculations

To compute the correlation function of the noise, we call `Simulator.compute` method and specify `quantity = 'noise'`.

First we determine convergence of the correlation function with the CCE order.

[7]:
```
for o in [1, 2, 3, 4]:
    calc.generate_clusters(o)
    noise = calc.compute(ts, method='cce', quantity='noise')
    plt.plot(ts, noise.real, label=o)
plt.xlabel('Time (ms)')
plt.ylabel('Correlation')
```

[7]:
```
Text(0, 0.5, 'Correlation')
```



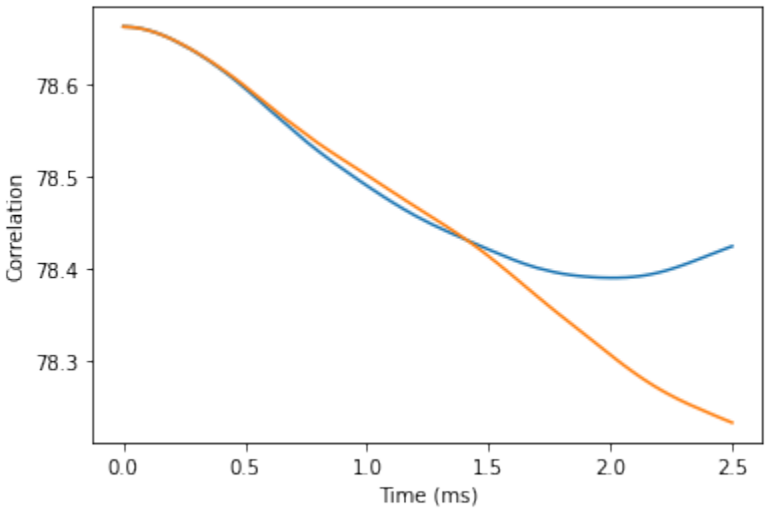
The difference between third and fourth order is fairly small, we will use the fourth order for the following calculations.

[8]:
```
calc.generate_clusters(4)

noise = calc.compute(ts, method='cce', quantity='noise')
genoise = calc.compute(ts, method='gcce', quantity='noise', nbstates=0)
```

Compare the results obtained with CCE and gCCE approaches. Note that they are slighlity different. However, as we will see it does not impact the predicted coherence.

[9]:
```
plt.plot(ts, noise.real, label='CCE')
plt.plot(ts, genoise.real, label='gCCE')
plt.xlabel('Time (ms)')
plt.ylabel('Correlation')
```

[9]:
```
Text(0, 0.5, 'Correlation')
```



Assuming that the noise is Gaussian, we can reproduce the coherence from the average phase squared $\langle\phi^2\rangle$, accumulated by the spin qubit:

$L(t) = e^{-\langle\phi^2(t)\rangle}$

The average phase is obtained from the autocorrelation function as:

$\langle\phi^2(t)\rangle = \int_0^t d\tau C(\tau)F(\tau)$

Where $F(\tau)$ is the correlation filter function (see Phys. Rev. A 86, 012314 (2012) for details).

PyCCE code already has implemented calculations of the phase in the `pycce.filter` module:

`pycce.filter.gaussian_phase` takes three positional arguments: - `timespace` - time points at which correlation function was computed; - `corr` - noise autocorrelation function; - `npulses` - number of pulses in CPMG sequence.

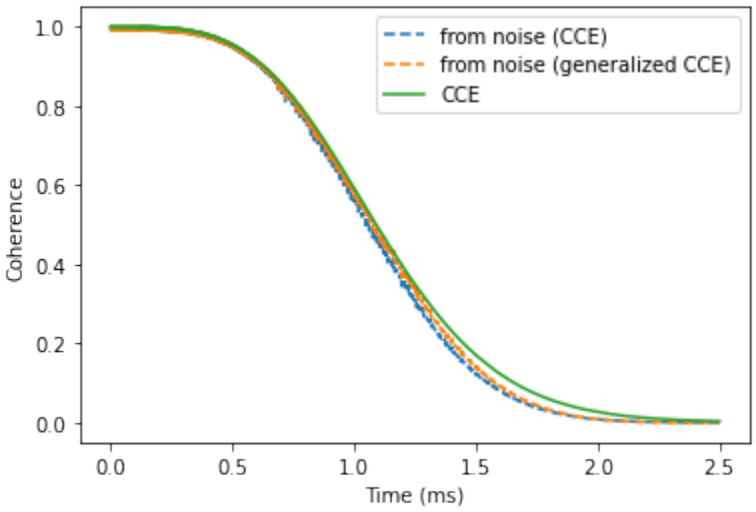
Here we compute the phase for the Hahn-echo experiment. Note that the implementation of `gaussian_phase` is not heavily optimized and can take a hot second.

[10]:
```
import pycce.filter

chis =  pycce.filter.gaussian_phase(ts, np.abs(noise), 1)
gchis = pycce.filter.gaussian_phase(ts, np.abs(genoise), 1)
```

Now compare results from direct calculations of the coherence function, and the one reconstructed from the noise autocorrelation:

[11]:
```
plt.plot(ts, np.exp(-chis), ls='--', label='from noise (CCE)')
plt.plot(ts, np.exp(-gchis), ls='--', marker='', label='from noise (generalized CCE)')
plt.plot(ts, coh[4], label='CCE')
plt.legend()
plt.xlabel('Time (ms)')
plt.ylabel('Coherence')
```

[11]:
```
Text(0, 0.5, 'Coherence')
```



The recommended order of the tutorials is from the top to bottom:

- *NV Center in Diamond* example goes through the *Quick Start* example in more details.
- *VV in SiC* tutorial explores the difference between generalized CCE with and without random bath state sampling. Also, in this example we introduce the way to work with DFT output of hyperfine tensors.
- *Shallow donor in Si* example shows the way to include the custom hyperfine couplings for more delocalized defects in semiconductors.
- *Correlation function* example explains the way to use autocorrelation function of the noise to predict the decay of the coherence of the NV center in diamond.

CHAPTER

FOUR

GENERATING THE SPIN BATH

4.1 Random bath

Documentation for the `pycce.random_bath` function, used to generate random bath.

random_bath(*names*, *size*, *number=1000*, *density=None*, *types=None*, *density_units='cm-3'*, *center=None*, *seed=None*)

Generate random bath containing spins with names provided with argument `name` in the box of size `size`. By default generates coordinates in range (-size/2; +size/2) but this behavior can be changed by providing `center` keyword.

Examples

Generate 2000 ^{13}C nuclear spins in the cubic box with the side of 100 angstrom:

```
>>> atoms = random_bath('13C', 100, number=2000, seed=10)
>>> print(atoms.size)
2000
>>> print(round(atoms.x.min()), round(atoms.x.max()))
-50.0 50.0
```

Generate electron spin bath with density $10^{17}\mathrm{cm}^{-3}$ in the cuboid box:

```
>>> electrons = random_bath('e', [1e3, 2e3, 3e3], density=1e17,
>>>                         density_units='cm-3', seed=10)
>>> print(electrons.size, round(electrons.x.min()), round(electrons.x.max()))
600 -494.0 500.0
>>> print(electrons.types)
SpinDict(e: (e, 0.5, -17608.59705))
```

Parameters

- **names** (*str or array-like with length n*) – Name of the bath spin or array with the names of the bath spins,
- **size** (*float or ndarray with shape (3,)*) – Size of the box. If float is given, assumes 3D cube with the edge = `size`. Otherwise the size specifies the dimensions of the box. Dimensionality is controlled by setting entries of the size array to 0.
- **number** (*int or array-like with length n*) – Number of the bath spins in the box or array with the numbers of the bath spins. Has to have the same length as the `name` array.

- **density** (*float or array-like with length n*) – Concentration of the bath spin or array with the concentrations. Has to have the same length as the `name` array.
- **types** (SpinDict) – Dictionary with SpinTypes or input to create one.
- **density_units** (*str*) – If number of spins provided as density, defines units. Values are accepted in the format `m`, or `m^x` or `m-x` where m is the length unit, x is dimensionality of the bath (e.g. x = 1 for 1D, 2 for 2D etc). If only `m` is provided the dimensions are inferred from `size` argument. Accepted length units:
  - `m` meters;
  - `cm` centimeters;
  - `a` angstroms.
- **center** (*ndarray with shape (3,)*) – Coordinates of the (0, 0, 0) point of the final coordinate system in the initial coordinates. Default is `size / 2` - center is in the middle of the box.

Returns Array of the bath spins with random positions.

Return type BathArray with shape (np.prod(number))

4.2 BathCell

Documentation for the `pycce.BathCell` - class for convenient generation of `BathArray` and the necessary helper functions.

class BathCell(*a=None*, *b=None*, *c=None*, *alpha=None*, *beta=None*, *gamma=None*, *angle='rad'*, *cell=None*)

Generator of the bath spins positions from the unit cell of the material.

Parameters

- **a** (*float*) – a parameter of the primitive cell.
- **b** (*float*) – b parameter of the primitive cell.
- **c** (*float*) – c parameter of the primitive cell.
- **alpha** (*float*) – α angle of the primitive cell.
- **beta** (*float*) – β angle of the primitive cell.
- **gamma** (*float*) – γ angle of the primitive cell.
- **angle** (*str*) – units of the α, β, γ angles. Can be either radians (`'rad'`), or degrees (`'deg'`).
- **cell** (*ndarray with shape (3, 3)*) – Parameters of the cell.

  `cell` is 3x3 matrix with columns of coordinates of crystallographic vectors in the cartesian reference frame. See `cell` attribute.

  If provided, overrides a, b, and c.

cell

Parameters of the cell. `cell` is 3x3 matrix with entries:

$$[[a_x\ b_x\ c_x]$$
$$[a_y\ b_y\ c_y]$$
$$[a_z\ b_z\ c_z]]$$

where a, b, c are crystallographic vectors and x, y, z are their coordinates in the cartesian reference frame.

Type ndarray with shape (3, 3)

atoms
Dictionary containing coordinates and occupancy of each lattice site:

```
{atom_1: [array([x1, y1, z1]), array([x2, y2, z2])],
 atom_2: [array([x3, y3, z3]), ...]}
```

Type dict

isotopes
Dictionary containing spin types and their concentration for each lattice site type:

```
{atom_1: {spin_1: concentration, spin_2: concentration},
 atom_2: {spin_3: concentration ...}}
```

where `atom_i` are lattice site types, and `spin_i` are spin types.

Type dict

property zdir
z-direction of the reference cartesian coordinate frame in cell coordinates.

Type ndarray

rotate(*rotation_matrix*)
Rotate the BathCell using the rotation matrix provided.

Parameters **rotation_matrix** (*ndarray with shape (3,)*) – Rotation matrix R which rotates the old basis of the cartesian reference frame to the new basis.

set_zdir(*direction*, *type='cell'*)
Set z-direction of the cell.

Parameters

- **direction** (*ndarray with shape (3,)*) – Direction of the z axis.
- **type** (*str*) – How coordinates in `direction` are stored. If `type="cell"`, assumes crystallographic coordinates. If `type="angstrom"` assumes that z direction is given in the cartresian reference frame.

add_atoms(**args*, *type='cell'*)
Add coordinates of the lattice sites to the unit cell.

Parameters

- ***args** (*tuple*) – List of tuples, each containing the type of atom N (*str*), and the xyz coordinates in the format (*float, float, float*): `(N, [x, y, z])`.
- **type** (*str*) – Type of coordinates. Can take values of `['cell', 'angstrom']`.

  If `type="cell"`, assumes crystallographic coordinates.

  If `type="angstrom"` assumes that coordinates are given in the cartresian reference frame.

Returns View of `cell.atoms` dictionary, where each key is the type of lattice site, and each value is the list of coordinates in crystallographic frame.

Return type dict

Examples

```
>>> cell = BathCell(10)
>>> cell.add_atoms(('C', [0, 0, 0]), ('C', [5, 5, 5]), type='angstrom')
>>> cell.add_atoms(('Si', [0, 0.5, 0.]), type='cell')
>>> print(cell.atoms)
{'C': [array([0., 0., 0.]), array([0.5, 0.5, 0.5])], 'Si': [array([0. , 0.5, 0.␣
↪])]}
```

add_isotopes(**args*)

Add spins that can populate each lattice site type.

Parameters ***args** (*tuple or list of tuples*) – Each tuple can have any of the following formats:

- Name of the lattice site *N* (*str*), name of the spin *X* (*str*), concentration *c* (*float*, in decimal): `(N, X, c)`.
- Isotope name *X and concentration `c*: `(X, c)`.

  In this case, the name of the isotope is given in the format `"{}{}".format(digits, atom_name)` where `digits` is any set of digits 0-9, `atom_name` is the name of the corresponding lattice site. Convenient when generating nuclear spin bath.

Returns

View of `cell.isotopes` dictionary which contains information about lattice site types, spin types, and their concentrations:

```
{atom_1: {spin_1: concentration, spin_2: concentration},
 atom_2: {spin_3: concentration ...}}
```

Return type dict

Examples

```
>>> cell = BathCell(10)
>>> cell.add_atoms(('C', [0, 0, 0]), ('C', [5, 5, 5]), type='angstrom')
>>> cell.add_isotopes(('C', 'X', 0.001), ('13C', 0.0107))
>>> print(cell.isotopes)
{'C': {'X': 0.001, '13C': 0.0107}}
```

gen_supercell(*size*, *add=None*, *remove=None*, *seed=None*)

Generate supercell populated with spins.

Note: If `isotopes` were not provided, assumes the natural concentration of nuclear spin isotopes for each lattice site type. However, if any isotope concentration is provided, then uses only user-defined ones.

Parameters

- **size** (*float*) – Approximate linear size of the supercell. The generated supercell will have minimal distance between opposite sides larger than this parameter.

- **add** (*tuple or list of tuples*) – Tuple or list of tuples containing common_isotopes to add as a defect. Each tuple contains name of the new isotope and its coordinates in the cell basis: `(isotope_name, x_cell, y_cell, z_cell)`.
- **remove** (*tuple or list of tuples*) – Tuple or list of tuples containing bath to remove in the defect. Each tuple contains name of the atom to remove and its coordinates in the cell basis: `(atom_name, x_cell, y_cell, z_cell)`.
- **seed** (*int*) – Seed for random number generator.

Note: While `add` takes the **spin** name as an argument, `remove` takes the lattice site name.

Returns Array of the spins in the given supercell.

Return type *BathArray*

`to_cartesian`(*coord*)

Transform coordinates from crystallographic basis to the cartesian reference frame.

Parameters **coord** (*ndarray with shape (3,) or (n, 3)*) – Coordinates in crystallographic basis or array of coordinates.

Returns Cartesian coordinates in angstrom.

Return type ndarray with shape (3,) or (n, 3)

`to_cell`(*coord*)

Transform coordinates from the cartesian coordinates of the reference frame to the cell coordinates.

Parameters **coord** (*ndarray with shape (3,) or (n, 3)*) – Cartesian coordinates in angstrom or array of coordinates.

Returns Coordinates in the cell basis.

Return type ndarray with shape (3,) or (n, 3)

`classmethod from_ase`(*atoms_object*)

Generate `BathCell` instance from `ase.Atoms` object of Atomic Simulations Environment (ASE) package.

Parameters **atoms_object** (*Atoms*) – Atoms object, used to generate new `BathCell` instance.

Returns New instance of the `BathCell` with atoms read from `ase.Atoms`.

Return type *BathCell*

`defect`(*cell*, *atoms*, *add=None*, *remove=None*)

Generate a defect in the given supercell.

The defect will be located in the unit cell, located roughly in the middle of the supercell, generated by `BathCell`, such that (0, 0, 0) of cartesian reference frame is located at (0, 0, 0) position of this unit cell.

Parameters

- **cell** (*ndarray with shape (3, 3)*) – parameters of the unit cell.
- **atoms** (`BathArray`) – Array of spins in the supercell.
- **add** (*tuple or list of tuples*) – Add spin type(s) to the supercell at specified positions to create point defect. Each tuple contains name of the new isotope and its coordinates in the cell basis: `(isotope_name, x_cell, y_cell, z_cell)`.

- **remove** (*tuple or list of tuples*) – Remove lattice site from the supercell at specified position to create point defect. Each tuple contains name of the atom to remove and its coordinates in the cell basis: `(atom_name, x_cell, y_cell, z_cell)`.

Returns Array of spins with the defect added.

Return type *BathArray*

4.3 BathArray

Documentation for the `pycce.BathArray` - central class, containing properties of the bath spins.

class BathArray(*shape=None*, *array=None*, *names=None*, *hyperfines=None*, *quadrupoles=None*, *types=None*, *imap=None*, *ca=None*, *sn=None*, *hf=None*, *q=None*, *efg=None*)

Subclass of `ndarray` containing information about the bath spins.

The subclass has fixed structured datatype:

```
_dtype_bath = np.dtype([('N', np.unicode_, 16),
                        ('xyz', np.float64, (3,)),
                        ('A', np.float64, (3, 3)),
                        ('Q', np.float64, (3, 3))])
```

Accessing different fields results in the `ndarray` view.

Each of the fields can be accessed as the attribute of the `BathArray` instance and modified accordingly. In addition to the name fields, the information of the bath spin types is stored in the `types` attribute. All of the items in `types` can be accessed as attributes of the BathArray itself.

Examples

Generate empty `BathArray` instance.

```
>>> ba = BathArray((3,))
>>> print(ba)
[('', [0., 0., 0.], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]], [[0., 0., 0.], [0.,␣
↪0., 0.], [0., 0., 0.]])
 ('', [0., 0., 0.], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]], [[0., 0., 0.], [0.,␣
↪0., 0.], [0., 0., 0.]])
 ('', [0., 0., 0.], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]], [[0., 0., 0.], [0.,␣
↪0., 0.], [0., 0., 0.]])]
```

Generate BathArray from the set of arrays:

```
>>> import numpy as np
>>> ca = np.random.random((3, 3))
>>> sn = ['1H', '2H', '3H']
>>> hf = np.random.random((3, 3, 3))
>>> ba = BathArray(ca=ca, hf=hf, sn=sn)
>>> print(ba.N, ba.types)
['1H' '2H' '3H'] SpinDict(1H: (1H, 0.5, 26.7519), 2H: (2H, 1, 4.1066, 0.00286), 3H:␣
↪(3H, 0.5, 28.535))
```

Warning: Due to how structured arrays work, if one uses a boolean array to access an subarray, and then access the name field, the initial array *will not change*.

Example:

```
>>> ba = BathArray((10,), sn='1H')
>>> print(ba.N)
['1H' '1H' '1H' '1H' '1H' '1H' '1H' '1H' '1H' '1H']
>>> bool_mask = np.arange(10) % 2 == 0
>>> ba[bool_mask]['N'] = 'e'
>>> print(ba.N)
['1H' '1H' '1H' '1H' '1H' '1H' '1H' '1H' '1H' '1H']
```

To achieve the desired result, one should first access the name field and only then apply the boolean mask:

```
>>> ba['N'][bool_mask] = 'e'
>>> print(ba.N)
['e' '1H' 'e' '1H' 'e' '1H' 'e' '1H' 'e' '1H']
```

Parameters

- **shape** (*tuple*) – Shape of the array.
- **array** (*array-like*) – Either an unstructured array with shape (n, 3) containing coordinates of bath spins as rows OR structured ndarray with the same fields as the datatype of the bath.
- **name** (*array-like*) – Array of the bath spin name.
- **hyperfines** (*array-like*) – Array of the hyperfine tensors with shape (n, 3, 3).
- **quadrupoles** (*array-like*) – Array of the quadrupole tensors with shape (n, 3, 3).
- **efg** (*array-like*) – Array of the electric field gradients with shape (n, 3, 3) for each bath spin. Used to compute Quadrupole tensors for spins >= 1. Requires the spin types either be found in `common_isotopes` or specified with `types` argument.
- **types** (SpinDict) – SpinDict or input to create one. Contains either SpinTypes of the bath spins or tuples which will initialize those. See `pycce.bath.SpinDict` documentation for details.
- **imap** (InteractionMap) – Instance of InteractionMap containing user defined interaction tensors between bath spins stored in the array.
- **ca** (*array-like*) – Shorthand notation for `array` argument.
- **sn** (*array-like*) – Shorthand notation for `name` argument.
- **hf** (*array-like*) – Shorthand notation for `hyperfines` argument.
- **q** (*array-like*) – Shorthand notation for `quadrupoles` argument.

sort(*axis=- 1*, *kind=None*, *order=None*)

Sort array in-place. Is implemented only when imap is None. Otherwise use `np.sort`.

property name

Array of the `name` attribute for each spin in the array from `types` dictionary.

Note: While the value of this attribute should be the same as the `N` field of the BathArray instance, `.name` *should not* be used for production as it creates a *new* array from `types` dictionary.

Type ndarray

property s
Array of the `spin` (spin value) attribute for each spin in the array from `types` dictionary.

Type ndarray

property dim
Array of the `dim` (dimensions of the spin) attribute for each spin in the array from `types` dictionary.

Type ndarray

property gyro
Array of the `gyro` (gyromagnetic ratio) attribute for each spin in the array from `types` dictionary.

Type ndarray

property q
Array of the `q` (quadrupole moment) attribute for each spin in the array from `types` dictionary.

Type ndarray

property detuning
Array of the `detuning` attribute for each spin in the array from `types` dictionary.

Type ndarray

property x
Array of x coordinates for each spin in the array (`bath['xyz'][:, 0]`).

Type ndarray

property y
Array of y coordinates for each spin in the array (`bath['xyz'][:, 1]`).

Type ndarray

property z
Array of z coordinates for each spin in the array (`bath['xyz'][:, 2]`).

Type ndarray

property N
Array of name for each spin in the array (`bath['N']`).

Type ndarray

property xyz
Array of coordinates for each spin in the array (`bath['xyz']`).

Type ndarray

property A
Array of hyperfine tensors for each spin in the array (`bath['A']`).

Type ndarray

property Q
Array of quadrupole tensors for each spin in the array (`bath['Q']`).

Type ndarray

add_type(**args*, ***kwargs*)

Add spin type to the `types` dictionary.

Parameters

- ***args** – Any number of positional inputs to create `SpinDict` entries. E.g. the tuples of form (name `str`, spin `float`, gyro `float`, q `float`).
- ****kwargs** – Any number of keyword inputs to create `SpinDict` entries. E.g. name = (spin, gyro, q).

For details and allowed inputs see `SpinDict` documentation.

Returns A view of `self.types` instance.

Return type *SpinDict*

add_interaction(*i*, *j*, *tensor*)

Add interactions tensor between bath spins with indexes `i` and `j`.

Note: If called from the subarray this method **does not** change the tensors of the total BathArray.

Parameters

- **i** (*int or ndarray (n,)*) – Index of the first spin in the pair or array of the indexes of the first spins in n pairs.
- **j** (*int or ndarray with shape (n,)*) – Index of the second spin in the pair or array of the indexes of the second spins in n pairs.
- **tensor** (*ndarray with shape (3,3) or (n, 3,3)*) – Interaction tensor between the spins i and j or array of tensors.

update(*ext_bath*, *error_range=0.2*, *ignore_isotopes=True*, *inplace=True*)

Update the properties of the spins in the array using data from other `BathArray` instance. For each spin in `ext_bath` check whether there is such spin in the array that has the same position within allowed error range given by `error_range` and has the same name. If such spins is found in the array, then it's coordinates, hyperfine tensor and quadrupole tensor are updated using the values of the spin in the `ext_bath` object.

If `ignore_isotopes` is true, then the name check ignores numbers in the name of the spins.

Parameters

- **ext_bath** (`BathArray`) – Array of the new spins.
- **error_range** (*float*) – +- distance in Angstrom within which two positions are considered to be the same. Default is 0.2 A.
- **ignore_isotopes** (*bool*) – True if ignore numbers in the name of the spins. Default True.
- **inplace** (*bool*) – True if changes parameters of the array in place. If False, returns copy of the array.

Returns updated BathArray instance.

Return type *BathArray*

from_point_dipole(*position*, *gyro_e=- 17608.59705*, *inplace=True*)
Generate hyperfine couplings, assuming that bath spins interaction with central spin is the same as the one between two magnetic point dipoles.

Parameters

- **position** (*ndarray with shape (3,)*) – position of the central spin
- **gyro_e** (*float or ndarray with shape (3,3)*) – gyromagnetic ratio of the central spin

  OR

  tensor corresponding to interaction between magnetic field and central spin.
- **inplace** (*bool*) – True if changes parameters of the array in place. If False, returns copy of the array.

Returns updated BathArray instance with changed hyperfine couplings.

Return type *BathArray*

from_cube(*cube*, *gyro_e=- 17608.59705*, *inplace=True*)
Generate hyperfine couplings, assuming that bath spins interaction with central spin can be approximated as a point dipole, interacting with given spin density distribution.

Parameters

- **cube** (Cube) – An instance of *Cube* object, which contains spatial distribution of spin density. For details see documentation of *Cube* class.
- **gyro_e** (*float*) – Gyromagnetic ratio of the central spin.
- **inplace** (*bool*) – True if changes parameters of the array in place. If False, returns copy of the array.

Returns Updated BathArray instance with changed hyperfine couplings.

Return type *BathArray*

from_func(*func*, *gyro_e=- 17608.59705*, *vectorized=False*, *inplace=True*)
Generate hyperfine couplings from user-defined function.

Parameters

- **func** (*func*) – Callable with signature:

```
func(coord, gyro, central_gyro)
```

  where `coord` is array of the bath spin coordinate, `gyro` is the gyromagnetic ratio of bath spin, `central_gyro` is the gyromagnetic ratio of the central bath spin.
- **gyro_e** (*float*) – gyromagnetic ratio of the central spin to be used in the function.
- **vectorized** (*bool*) – If True, assume that func takes arrays of all bath spin coordinates and array of gyromagnetic ratios as arguments.
- **inplace** (*bool*) – True if changes parameters of the array in place. If False, returns copy of the array.

Returns Updated BathArray instance with changed hyperfine couplings.

Return type *BathArray*

from_efg(*efg*, *inplace=True*)
Generate quadrupole splittings from electric field gradient tensors for spins >= 1.

Parameters

- **efg** (*array-like*) – Array of the electric field gradients for each bath spin. The data for spins-1/2 should be included but can be any value.
- **inplace** (*bool*) – True if changes parameters of the array in place. If False, returns copy of the array.

Returns Updated BathArray instance with changed quadrupole tensors.

Return type *BathArray*

dist(*position=None*)

Compute the distance of the bath spins from the given position.

Parameters **position** (*ndarray with shape (3,)*) – Cartesian coordinates of the position from which to compute the distance. Default is (0, 0, 0).

Returns Array of distances of each bath spin from the given position in angstrom.

Return type ndarray with shape (n,)

savetxt(*filename*, *fmt='%18.8f'*, *strip_isotopes=False*, ***kwargs*)

Save name of the isotopes and their coordinates to the txt file of xyz format.

Parameters

- **filename** (*str or file*) – Filename or file handle.
- **fmt** (*str*) – Format of the coordinate entry.
- **strip_isotopes** (*bool*) – True if remove numbers from the name of bath spins. Default False.
- ****kwargs** – Additional keywords of the `numpy.savetxt` function.

sort(*a*, **args*, ***kwargs*)

Return a indexes of an sorted array. Overrides `numpy.argsort` function.

same_bath_indexes(*barray_1*, *barray_2*, *error_range=0.2*, *ignore_isotopes=True*)

Find indexes of the same array elements in two `BathArray` instances.

Parameters

- **barray_1** (`BathArray`) – First array.
- **barray_2** (`BathArray`) – Second array.
- **error_range** (*float*) – If distance between positions in two arrays is smaller than `error_range` they are assumed to be the same.
- **ignore_isotopes** (*bool*) – True if ignore numbers in the name of the spins. Default True.

Returns

tuple containing:

- **ndarray**: Indexes of the elements in the first array found in the second.
- **ndarray**: Indexes of the elements in the second array found in the first.

Return type tuple

utilities.**rotmatrix**(*final_vector*)

Generate 3D rotation matrix which applied on initial vector will produce vector, aligned with final vector.

Examples

```
>>> R = rotmatrix([0,0,1], [1,1,1])
>>> R @ np.array([0,0,1])
array([0.577, 0.577, 0.577])
```

Parameters

- **initial_vector** (*ndarray with shape(3,)*) – Initial vector.
- **final_vector** (*ndarray with shape (3,)*) – Final vector.

Returns Rotation matrix.

Return type ndarray with shape (3, 3)

4.3.1 InteractionMap

class InteractionMap(*rows=None*, *columns=None*, *tensors=None*)

Dict-like object containing information about tensor interactions between two spins.

Each key is a tuple of two bath spin indexes.

Parameters

- **rows** (*array-like with shape (n,)*) – Indexes of the bath spins, appearing on the left in the pairwise interaction.
- **columns** (*array-like with shape (n,)*) – Indexes of the bath spins, appearing on the right in the pairwise interaction.
- **tensors** (*array-like with shape (n, 3, 3)*) – Tensors of pairwise interactions between two spins with the indexes in `rows` and `columns`.

mapping

Actual dictionary storing the data.

Type dict

property indexes

Array with the indexes of pairs of bath spins, for which the tensors are stored.

Type ndarray with shape (n, 2)

shift(*start*, *inplace=True*)

Add an offset `start` to the indexes. If `inplace` is False, returns the copy of InteractionMap.

Parameters

- **start** (*int*) – Offset in indexes.
- **inplace** (*bool*) – If True, makes changes inplace. Otherwise returns copy of the map.

Returns Map with shifted indexes.

Return type *InteractionMap*

keys() → a set-like object providing a view on D's keys

items() → a set-like object providing a view on D's items

subspace(*array*)

Get new InteractionMap with indexes readressed from array. Within the subspace indexes are renumbered.

Examples

The subspace of [3,4,7] indexes will contain InteractionMap only within [3,4,7] elements with new indexes [0, 1, 2].

```
>>> import numpy as np
>>> im = InteractionMap()
>>> im[0, 3] = np.eye(3)
>>> im[3, 7] = np.ones(3)
>>> for k in im: print(k, '\n', im[k],)
(0, 3)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
(3, 7)
 [[1. 1. 1.]
  [1. 1. 1.]
  [1. 1. 1.]]
>>> array = [3, 4, 7]
>>> sim = im.subspace(array)
>>> for k in sim: print(k, '\n', sim[k])
(0, 2)
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

Parameters **array** (*ndarray*) – Either bool array containing True for elements within the subspace or array of indexes presented in the subspace.

Returns The map for the subspace.

Return type *InteractionMap*

classmethod from_dict(*dictionary*, *presorted=False*)

Generate InteractionMap from the dictionary. :param dictionary: Dictionary with tensors. :type dictionary: dict :param presorted: If true, assumes that the keys in the dictionary were already presorted. :type presorted: bool

Returns New instance generated from the dictionary.

Return type *InteractionMap*

4.3.2 Cube

class Cube(*filename*)

Class to process the .cube datafiles with spin polarization.

Parameters **filename** (*str*) – Name of the .cube file.

comments

First two lines of the .cube file.

Type str

origin

Coordinates of the origin in angstrom.

Type ndarray with shape (3,)

voxel
Parameters of the voxel - unit of the 3D grid in angstrom.

Type ndarray with shape (3,3)

size
Size of the cube.

Type ndarray with shape (3,)

atoms
Array of atoms in the cube.

Type BathArray with shape (n)

data
Data stored in cube.

Type ndarray with shape (size[0], size[1], size[2]

grid
Coordinates of the points at which data is computed.

Type ndarray with shape (size[0], size[1], size[2], 3

integral
Data integrated over cube.

Type float

spin
integral / 2 - total spin.

Type float

transform(*rotmatrix=None*, *shift=None*)
Changes coordinates of the grid. DOES NOT ASSUME PERIODICITY.

Parameters

- **rotmatrix** (*ndarray with shape (3, 3)*) – Rotation matrix *R*:

$$R = [n_1^{(1)} n_1^{(2)} n_1^{(3)}] \\ [n_2^{(1)} n_2^{(2)} n_2^{(3)}] \\ [n_3^{(1)} n_3^{(2)} n_3^{(3)}]$$

  where $n_i^{(j)}$ corresponds to the coefficient of initial basis vector i for j new basis vector:

$$e'_j = n_1^{(j)} \vec{e}_1 + n_2^{(j)} \vec{e}_2 + n_3^{(j)} \vec{e}_3$$

  In other words, columns of *R* are coordinates of the new basis in the old basis.

  Given vector in initial basis v = [v1, v2, v3], vector in new basis is given as v' = R.T @ v.

- **shift** (*ndarray with shape (3,)*) – Shift in the origin of coordinates (in the old basis).

integrate(*position*, *gyro_n*, *gyro_e=- 17608.59705*, *spin=None*)
Integrate over polarization data, stored in Cube object, to obtain hyperfine dipolar-dipolar tensor.

Parameters

- **position** (*ndarray with shape (3,) or (n, 3)*) – Position of the bath spin at which to compute hyperfine tensor or array of positions.
- **gyro_n** (*float or ndarray with shape (n,)*) – Gyromagnetic ratio of the bath spin or array of the ratios.
- **gyro_e** (*float*) – Gyromagnetic ratio of central spin.
- **spin** (*float*) – Total spin of the central spin. If not given, taken from the integral of the polarization.

Returns Hyperfine tensor or array of hyperfine tensors.

Return type ndarray with shape (3, 3) or (n, 3, 3)

4.4 SpinDict and SpinType

Documentation for the `SpinDict` - dict-like class which describes the properties of the different types of the spins in the bath.

class SpinDict(**args*, ***kwargs*)

Wrapper class for dictionary tailored for containing properties of the spin types. Can take `np.void` or `BathArray` instances as keys. Every entry is instance of the `SpinType`.

Each entry of the `SpinDict` can be initianlized as follows:

- As a Tuple containing name (optional), spin, gyromagnetic ratio, quadrupole constant (optional) and detuning (optional).
- As a `SpinType` instance.

Examples

```
>>> types = SpinDict()
>>> types['1H'] = ('1H', 1 / 2, 26.7519)
>>> types['2H'] = 1, 4.1066, 0.00286
>>> types['3H'] = SpinType('3H', 1 / 2, 28.535, 0)
>>> print(types)
SpinDict({'1H': (1H, 0.5, 26.7519, 0.0), '2H': (2H, 1, 4.1066, 0.00286), '3H': (3H,␣
↪0.5, 28.535, 0)})
```

If `SpinType` of the given bath spin is not provided, when requested `SpinDict` will try to find information about the bath spins in the `common_isotopes`.

If found, adds an entry to the given `SpinDict` instance and returns it. Otherwise `KeyError` is raised.

To initiallize several `SpinType` entries one can use `add_types` method.

Parameters

- ***args** – Any numbers of arguments which could initialize `SpinType` instances.
- ****kwargs** – Any numbers of keyword arguments which could initialize `SpinType` instances. For details see `SpinDict.add_type` method.

add_type(**args*, ***kwargs*)

Add one or several spin types to the spin dictionary.

Parameters

- ***args** – Any numbers of arguments which could initialize `SpinType` instances. Accepted arguments:
  - Tuple containing name, spin, gyromagnetic ratio, quadrupole constant (optional) and detuning (optional).
  - `SpinType` instance.

  Can also initialize one instance of `SpinType` if each argument corresponds to each positional argument necessary to initiallize.
- ****kwargs** – Any numbers of keyword arguments which could initialize `SpinType` instances. Usefull as an alternative for updating the dictionary. for each keyword argument adds an entry to the `SpinDict` with the same name as keyword.

Examples

```
>>> types = SpinDict()
>>> types.add_type('1H', 1 / 2, 26.7519)
>>> types.add_type(('1H_det', 1 / 2, 26.7519, 10), ('2H', 1, 4.1066, 0.00286),
>>>                SpinType('3H', 1 / 2, 28.535, 0), e=(1 / 2, 6.7283, 0))
>>> print(types)
SpinDict(1H: (1H, 0.5, 26.7519), 1H_det: (1H_det, 0.5, 26.7519, 10),
2H: (2H, 1, 4.1066, 0.00286), 3H: (3H, 0.5, 28.535), e: (e, 0.5, 6.7283))
```

class SpinType(*name*, *s=0.0*, *gyro=0.0*, *q=0.0*, *detuning=0.0*)

Class which contains properties of each spin type in the bath.

Parameters

- **name** (*str*) – Name of the bath spin.
- **s** (*float*) – Total spin of the bath spin.

  Default 0.
- **gyro** (*float*) – Gyromagnetic ratio in rad * kHz / G.

  Default 0.
- **q** (*float*) – Quadrupole moment in barn (for s > 1/2).

  Default 0.
- **detuning** (*float*) – Energy detuning from the zeeman splitting in kHz, included as an extra $+\omega\hat{S}_z$ term in the Hamiltonian, where ω is the detuning.

  Default 0.

name

Name of the bath spin.

Type str

s

Total spin of the bath spin.

Type float

dim

Spin dimensionality = 2s + 1.

Type int

`gyro`
Gyromagnetic ratio in rad/(ms * G).

Type float

`q`
Quadrupole moment in barn (for s > 1/2).

Type float

`detuning`
Energy detuning from the zeeman splitting in kHz.

Type float

`common_isotopes = SpinDict(1H: (0.5, 26.7522), 2H: (1.0, 4.1066, 0.0029), 3He: (0.5, -20.3789), ...)`
An instance of the `SpinDict` dictionary, containing properties for the most of the common isotopes with nonzero spin. The isotope is considered common if it is stable and has nonzero concentration in nature.

Type *SpinDict*

`common_concentrations = {element ('H', 'He',...) : { isotope ('1H', '2H', ..) : concentration}}`
Nested dict containing natural concentrations of the stable nuclear isotopes.

Type dict

CHAPTER

FIVE

RUNNING THE SIMULATIONS

5.1 Setting up the Simulator Object

Documentation for the `pycce.Simulator` - main class for conducting CCE Simulations.

class Simulator(*spin*, *position=None*, *alpha=None*, *beta=None*, *gyro=- 17608.59705*, *magnetic_field=None*, *D=0.0*, *E=0.0*, *r_dipole=None*, *order=None*, *bath=None*, *pulses=None*, *as_delay=False*, *n_clusters=None*, ***bath_kw*)

The main class for CCE calculations.

The typical usage includes:

1. Read array of the bath spins. This is done with `Simulator.read_bath` method which accepts either reading from .xyz file or from the `BathArray` instance with defined positions and names of the bath spins. In the process, the subset of the array within the distance of `r_dipole` from the central spin is taken and for this subset the Hyperfine couplings can be generated.

   If no `hyperfine` keyword is provided and there are some hyperfine couplings already, then no changes are done to the hyperfine tensors. If `hyperfine='pd'`, the hyperfine couplings are computed assuming point dipole approximation. For all accepted arguments, see `Simulator.read_bath`.

2. Generate set of clusters with `Simulator.generate_clusters`, determined by the maximum connectivity radius `r_dipole` and the maximum size of the cluster `order`.

3. Compute the desired property with `Simulator.compute` method.

Note: Directly setting up the attribute values will rerun `Simulator.read_bath` and/or `Simulator.generate_clusters` to reflect updated value of the given attribute.

E.g. If `Simulator.r_bath` is set to some new value after initialization, then `Simulator.read_bath` and `Simulator.generate_clusters` are called with the increased bath.

First two steps are usually done during the initialization of the `Simulator` object by providing the necessary arguments.

Notes

Depending on the number of provided arguments, in the initialization process will call the following methods to setup the calculation engine.

- If `bath` is provided, `Simulator.read_bath` is called with additional keywords in `**bath_kw`.
- If both `r_dipole` and `order` are provided and `bath` is not None, the `Simulator.generate_clusters` is called.

See the corresponding method documentation for details.

Examples:

```
>>> atoms = random_bath('13C', 100, number=2000, seed=10)
>>> calc = Simulator(1, bath=atoms, r_bath=40, r_dipole=6,
>>>                  order=2, D=2.88 * 2 * np.pi * 1e6,
>>>                  magnetic_field=500, pulses=1)
>>> print(calc)
Simulator for spin-1.
alpha: [0.+0.j 1.+0.j 0.+0.j]
beta: [0.+0.j 0.+0.j 1.+0.j]
gyromagnetic ratio: -17608.59705 kHz * rad / G
zero field splitting:
array([[-6031857.895,        0.   ,        0.   ],
       [       0.   , -6031857.895,        0.   ],
       [       0.   ,        0.   , 12063715.79 ]])
magnetic field:
array([  0.,   0., 500.])

Parameters of cluster expansion:
r_bath: 40
r_dipole: 6
order: 2

Bath consists of 549 spins.

Clusters include:
549  clusters of order 1.
457  clusters of order 2.
```

Parameters

- **spin** (*float*) – Total spin of the central spin.
- **position** (*ndarray*) – Cartesian coordinates in Angstrom of the central spin. Default (0., 0., 0.).
- **alpha** (*float or ndarray with shape (2s+1,)*) – 0 state of the qubit in S_z basis or the index of eigenstate to be used as one.

  Default: state with $m_s = +s$ where m_s is the z-projection of the spin and s is the total spin if no information of central spin Hamiltonian is provided. Otherwise lowest energy eigenstate of the central spin Hamiltonian.
- **beta** (*float or ndarray with shape (2s+1,)*) – 1 state of the qubit in S_z basis or the index of the eigenstate to be used as one.

Default: state with $m_s = +s - 1$ where m_s is the z-projection of the spin and s is the total spin if no information of central spin Hamiltonian is provided. Otherwise second lowest energy eigenstate of the central spin Hamiltonian.

- **gyro** (*float or ndarray with shape (3,3)*) – Gyromagnetic ratio of central spin in rad / ms / G.

  OR

  Tensor describing central spin interactions with the magnetic field.

  Default -17608.597050 kHz * rad / G - gyromagnetic ratio of the free electron spin.

- **D** (*float or ndarray with shape (3, 3)*) – D (longitudinal splitting) parameter of central spin in ZFS tensor of central spin in kHz.

  OR

  Total ZFS tensor. Default 0.

- **E** (*float*) – E (transverse splitting) parameter of central spin in ZFS tensor of central spin in kHz. Default 0. Ignored if `D` is None or tensor.

- **bath** (*ndarray or str*) – First positional argument of the `Simulator.read_bath` method.

  Either:

  - Instance of BathArray class;
  - ndarray with `dtype([('N', np.unicode_, 16), ('xyz', np.float64, (3, ))])` containing names of bath spins (same ones as stored in self.ntype) and positions of the spins in angstroms;
  - the name of the xyz text file containing 4 cols: name of the bath spin and xyz coordinates in A.

- **r_dipole** (*float*) – Maximum connectivity distance between two bath spins.

- **order** (*int*) – Maximum size of the cluster to be considered in CCE expansion.

- **n_clusters** (*dict*) – Dictionary which contain maximum number of clusters of the given size. Has the form `n_clusters = {order: number}`, where `order` is the size of the cluster, `number` is the maximum number of clusters with this size.

  If provided, sort the clusters by the strength of cluster interaction, equal to the lowest pairwise interaction in the cluster. Then the strongest `number` of clusters is taken.

- **pulses** (*list or int or* Sequence) – Number of pulses in CPMG sequence or list with pulses.

- ****bath_kw** – Additional keyword arguments for the `Simulator.read_bath` method.

position
Position of the central spin in Cartesian coordinates.

Type ndarray with shape (3,)

spin
Value of the central spin s.

Type float

`gyro`
Gyromagnetic ratio of central spin in rad / ms / G.

OR

Tensor describing central spin interactions with the magnetic field.

Default -17608.597050 kHz * rad / G - gyromagnetic ratio of the free electron spin.

Type gyro (float or ndarray with shape (3,3))

`zfs`
Zero field splitting tensor of the central spin

Type ndarray with shape (3,3)

`clusters`
Dictionary containing information about cluster structure of the bath.

Each keys n correspond to the size of the cluster. Each `Simulator.clusters[n]` contains `ndarray` of shape (m, n), where m is the number of clusters of given size, n is the size of the cluster. Each row of this array contains indexes of the bath spins included in the given cluster. Generated during `.generate_clusters` call.

Type dict

`as_delay`
True if time points are delay between pulses (for equispaced pulses), False if time points are total time. Ignored if `pulses` contains the time delays.

Type bool

`state`
Innitial state of the qubit in gCCE simulations. Assumed to be $1/\sqrt{2}(0 + 1$ unless provided during `Simulator.compute` call.

Type ndarray

`interlaced`
True if use hybrid CCE approach - for each cluster sample over states of the supercluster.

Type bool

`seed`
Seed for random number generator, used in random bath states sampling.

Type int

`nbstates`
Number or random bath states to sample over.

Type int

`fixstates`
If not None, shows which bath states to fix in random bath states.

Each key is the index of bath spin, value - fixed $\hat{S}_z$ projection of the mixed state of nuclear spin.

Type dict

`masked`
True if mask numerically unstable points (with coherence > 1) in the averaging over bath states.

Note: It is up to user to check whether the possible instability is due to numerical error or unphysical assumptions of the calculations.

Type bool

`second_order`
True if add second order perturbation theory correction to the cluster Hamiltonian in conventional CCE. Relevant only for conventional CCE calculations.

Type bool

`level_confidence`
Maximum fidelity of the qubit state to be considered eigenstate of the central spin Hamiltonian when `second_order` set to True.

Type float

`projected_bath_state`
Array with z-projections of the bath spins states.

Type ndarray with shape (n,)

`bath_state`
Array of bath states.

Type bath_state (ndarray)

`timespace`
Time points at which compute the desired property.

Type timespace (ndarray with shape (n,))

`property alpha`
0 qubit state of the central spin in Sz basis **OR** index of the energy state to be considered as one.

Type ndarray or int

`property beta`
1 qubit state of the central spin in Sz basis **OR** index of the energy state to be considered as one.

Type ndarray or int

`property magnetic_field`
Array containing external magnetic field as (Bx, By, Bz). Default (0, 0, 0).

Type ndarray

`property order`
Maximum size of the cluster.

Type int

`property n_clusters`
Dictionary which contain maximum number of clusters of the given size. Has the form `n_clusters = {order: number}`, where `order` is the size of the cluster, `number` is the maximum number of clusters with this size.

If provided, sorts the clusters by the strength of cluster interaction, equal to the lowest pairwise interaction in the cluster. Then the strongest `number` of clusters is taken.

Type dict

property r_dipole
Maximum connectivity distance.

Type float

property pulses
List-like object, containing the sequence of the instantaneous ideal control pulses.

Each item is Pulse object, containing the following attributes:

- **axis** (*str*): Axis of rotation of the central spin. Can be 'x', 'y', or 'z'.
- **angle** (*float or str*): Angle of rotation of central spin. Can be provided in rad, or as a string, containing fraction of pi: 'pi', 'pi/2', '2*pi' etc. Default is None.
- **delay** (*float or ndarray*): Delay before the pulse or array of delays with the same shape as time points.
- **bath_names** (*str or array-like of str*): Name or array of names of bath spin types, impacted by the bath pulse.
- **bath_axes** (*str or array-like of str*): Axis of rotation or array of axes of the bath spins. If bath_names is provided, but bath_axes and bath_angles are not, assumes the same axis and angle as the one of the central spin.
- **bath_angles** (*float or str or array-like*): Angle of rotation or array of axes of rotations of the bath spins.

If delay is not provided in **all** pulses, assumes even delay of CPMG sequence.

If only **some** delays are provided, assumes 0 delay in the pulses without delay provided.

Type *Sequence*

set_zfs(*D=None*, *E=0*)
Set Zero Field Splitting of the central spin from longitudinal ZFS *D* and transverse ZFS *E*.

Parameters

- **D** (*float or ndarray with shape (3, 3)*) – D (longitudinal splitting) parameter of central spin in ZFS tensor of central spin in kHz.

  OR

  Total ZFS tensor. Default 0.
- **E** (*float*) – E (transverse splitting) parameter of central spin in ZFS tensor of central spin in kHz. Default 0. Ignored if D is None or tensor.

set_magnetic_field(*magnetic_field=None*)
Set magnetic field from either value of the magnetic field along z-direction or full magnetic field vector.

Parameters **magnetic_field** (*float or array-like*) – Magnetic field along z-axis.

OR

Array containing external magnetic field as (Bx, By, Bz). Default (0, 0, 0).

set_states(*alpha=None*, *beta=None*)
Set 0 and 1 Qubit states of the Simulator object.

Parameters

- **alpha** (*int or ndarray with shape (2s+1,)*) – 0 state of the qubit in S_z basis or the index of eigenstate to be used as one.

Default: Lowest energy eigenstate of the central spin Hamiltonian. Otherwise state with $m_s = +s$ where m_s is the z-projection of the spin and s is the total spin if no information of central spin Hamiltonian is provided.

- **beta** (*int or ndarray with shape (2s+1,)*) – 1 state of the qubit in S_z basis or the index of the eigenstate to be used as one.

  Default: Second lowest energy eigenstate of the central spin Hamiltonian. Otherwise state with $m_s = +s - 1$ where m_s is the z-projection of the spin and s is the total spin if no information of central spin Hamiltonian is provided.

eigenstates(*alpha=None*, *beta=None*, *magnetic_field=None*, *D=None*, *E=0*, *return_eigen=True*)

Compute eigenstates of the central spin Hamiltonian.

If `alpha` is provided, set alpha state as eigenstate. Similarly, if `beta` is provided, set beta state as eigenstate

Parameters

- **alpha** (*int*) – Index of the state to be considered as 0 (alpha) qubit state in order of increasing energy (0 - lowest energy).
- **beta** (*int*) – Index of the state to be considered as 1 (beta) qubit state.
- **magnetic_field** (*ndarray with shape (3,)*) – Array containing external magnetic field as (Sx, By, Bz).
- **D** (*float or ndarray with shape (3, 3)*) – D (longitudinal splitting) parameter of central spin in kHz *OR* total ZFS tensor.
- **E** (*float*) – E (transverse splitting) parameter of central spin in kHz. Ignored if `D` is None or tensor.
- **return_eigen** (*bool*) – If true, returns eigenvalues and eigenvectors of the central spin Hamiltonian.

Returns

tuple containing:

- **ndarray with shape (2s+1,)**: Array with eigenvalues of the central spin Hamiltonian.
- **ndarray with shape (2s+1, 2s+1)**: Array with eigenvectors of the central spin Hamiltonian. Each column of the array is eigenvector.

Return type tuple

5.2 Reading the Bath

Documentation for the `Simulator.read_bath` and `Simulator.generate_clusters` method. These methods are called automatically on the initialization of the `Simulator` object if the necessary keywords are provided. Otherwise they can also be called by themselves to update the properties of the spin bath in `Simulator` object.

`Simulator.`**read_bath**(*bath=None*, *r_bath=None*, *skiprows=1*, *external_bath=None*, *hyperfine=None*, *types=None*, *error_range=None*, *ext_r_bath=None*, *imap=None*)

Read spin bath from the file or from the `BathArray`.

Parameters

- **bath** (*ndarray, BathArray or str*) – Either:
  - Instance of BathArray class;

  - ndarray with `dtype([('N', np.unicode_, 16), ('xyz', np.float64, (3, ))])` containing names of bath spins (same ones as stored in self.ntype) and positions of the spins in angstroms;
  - the name of the xyz text file containing 4 cols: name of the bath spin and xyz coordinates in A.

- **r_bath** (*float*) – Cutoff size of the spin bath.
- **skiprows** (*int, optional*) – If `bath` is name of the file, this argument gives number of rows to skip while reading the .xyz file (default 1).
- **external_bath** (BathArray, *optional*) – BathArray containing spins read from DFT output (see `pycce.io`).
- **hyperfine** (*str, func, or Cube instance, optional*) – This argument tells the code how to generate hyperfine couplings.

  If (`hyperfine = None` and all A in provided bath are 0) or (`hyperfine = 'pd'`), use point dipole approximation.

  Otherwise can be an instance of `Cube` object, or callable with signature: `func(coord, gyro, central_gyro)`, where coord is array of the bath spin coordinate, gyro is the gyromagnetic ratio of bath spin, central_gyro is the gyromagnetic ratio of the central bath spin.
- **types** (SpinDict) – SpinDict or input to create one. Contains either SpinTypes of the bath spins or tuples which will initialize those.

  See `pycce.bath.SpinDict` documentation for details.
- **error_range** (*float, optional*) – Maximum distance between positions in bath and external bath to consider two positions the same (default 0.2).
- **ext_r_bath** (*float, optional*) – Maximum distance from the central spins of the bath spins for which to use the DFT positions.
- **imap** (InteractionMap) – Instance of `InteractionMap` class, containing interaction tensors for bath spins. Each key of the `InteractionMap` is a tuple with indexes of two bath spins. The value is the 3x3 tensor describing the interaction between two spins in a format:

$$I^i J I^j = I^i_x J_{xx} I^j_x + I^i_x J_{xy} I^j_y ...$$

Note: For each bath spin pair without interaction tensor in `imap`, coupling is approximated assuming magnetic point dipole–dipole interaction. If `imap = None` all interactions between bath spins are approximated in this way. Then interaction tensor between spins *i* and *j* is computed as:

$$\mathbf{J}_{ij} = -\gamma_i \gamma_j \frac{\hbar^2}{4\pi\mu_0} \left[\frac{3\vec{r}_{ij} \otimes \vec{r}_{ij} - |r_{ij}|^2 I}{|r_{ij}|^5} \right]$$

Where γ_i is gyromagnetic ratio of *i* spin, I is 3x3 identity matrix, and $\vec{r}_{ij}$ is distance between two spins.

Returns The view of `Simulator.bath` attribute, generated by the method.

Return type *BathArray*

`Simulator.`**generate_clusters**(*order=None*, *r_dipole=None*, *r_inner=0*, *strong=False*, *ignore=None*, *n_clusters=None*)

Generate set of clusters used in CCE calculations.

The clusters are generated from the following procedure:

- Each bath spin i forms a cluster of one.
- Bath spins i and j form cluster of two if there is an edge between them (distance $d_{ij} \leq$ `r_dipole`).
- Bath spins i, j, and j form a cluster of three if enough edges connect them (e.g., there are two edges ij and jk)
- And so on.

In general, we assume that spins $\{i..n\}$ form clusters if they form a connected graph. Only clusters up to the size imposed by the `order` parameter (equal to CCE order) are included.

Parameters

- **order** (*int*) – Maximum size of the cluster.
- **r_dipole** (*float*) – Maximum connectivity distance.
- **r_inner** (*float*) – Minimum connectivity distance.
- **strong** (*bool*) – True - generate only clusters with "strong" connectivity (all nodes should be interconnected). Default False.
- **ignore** (*list or str, optional*) – If not None, includes the names of bath spins which are ignored in the cluster generation.
- **n_clusters** (*dict*) – Dictionary which contain maximum number of clusters of the given size. Has the form `n_clusters = {order: number}`, where `order` is the size of the cluster, `number` is the maximum number of clusters with this size.

  If provided, sort the clusters by the strength of cluster interaction, equal to the lowest pairwise interaction in the cluster. Then the strongest `number` of clusters is taken.

Returns

View of `Simulator.clusters`. `Simulator.clusters` is a dictionary with keys corresponding to size of the cluster.

I.e. `Simulator.clusters[n]` contains `ndarray` of shape (m, n), where m is the number of clusters of given size, n is the size of the cluster. Each row contains indexes of the bath spins included in the given cluster.

Return type dict

5.3 Calculate Properties with Simulator

Documentation for the `Simulator.compute` method - the interface to run calculations with **PyCCE**.

`Simulator.compute`(*timespace*, *quantity='coherence'*, *method='cce'*, ***kwargs*)

General function for computing properties with CCE.

The dynamics are simulated using the Hamiltonian:

$$\hat{H}_S = \mathbf{SDS} + \mathbf{B}\gamma_S\mathbf{S}$$
$$\hat{H}_{SB} = \sum_i \mathbf{S}\mathbf{A}_i\mathbf{I}_i$$
$$\hat{H}_B = \sum_i \mathbf{I}_i\mathbf{P}_i\mathbf{I}_i + \mathbf{B}\gamma_i\mathbf{I}_i + \sum_{i>j} \mathbf{I}_i\mathbf{J}_{ij}\mathbf{I}_j$$

Here $\hat{H}_S$ is the central spin Hamiltonian with Zero Field splitting tensor $\mathbf{D}$ and gyromagnetic ratio tensor $\gamma_\mathbf{S} = \mu_S\mathbf{g}$ are read from `Simulator.zfs` and `Simulator.gyro` respectively.

The $\hat{H}_{SB}$ is the Hamiltonian describing interactions between central spin and the bath. The hyperfine coupling tensors $\mathbf{A}_i$ are read from the `BathArray` stored in `Simulator.bath['A']`. They can be generated using point dipole approximation or provided by the user (see `Simulator.read_bath` for details).

The $\hat{H}_B$ is the Hamiltonian describing interactions between the bath spins. The self interaction tensors $\mathbf{P}_i$ are read from the `BathArray` stored in `Simulator.bath['Q']` and have to be provided by the user.

The gyromagnetic ratios γ_i are read from the `BathArray.gyros` attribuite, which is generated from the properties of the types of bath spins, stored in `BathArray.types`. They can either be provided by user or read from the `pycce.common_isotopes` object.

The interaction tensors $\mathbf{J}_{ij}$ are assumed from point dipole approximation or can be provided in `BathArray.imap` attrubite.

Note: The `compute` method takes two keyword arguments to determine which quantity to compute and how:

- *method* can take 'cce' or 'gcce' values, and determines which method to use - conventional or generalized CCE.
- *quantity* can take 'coherence' or 'noise' values, and determines which quantity to compute - coherence function or autocorrelation function of the noise.

Each of the methods can be performed with monte carlo bath state sampling (if `nbstates` keyword is non zero) and with interlaced averaging (If `interlaced` keyword is set to `True`).

Examples

First set up Simulator object using random bath of 1000 13C nuclear spins.

```
>>> import pycce as pc
>>> import numpy as np
>>> atoms = pc.random_bath('13C', 100, number=2000, seed=10) # Random spin bath
>>> calc = pc.Simulator(1, bath=atoms, r_bath=40, r_dipole=6,
>>>                     order=2, D=2.88 * 1e6, # D of NV in GHz -> kHz
>>>                     magnetic_field=500, pulses=1)
>>> ts = np.linspace(0, 2, 101) # timesteps
```

We set magnetic field to 500 G along z-axis and chose 1 decoupling pulse (Hahn-echo) in this example. The zero field splitting is set to the one of NV center in diamond.

Run conventional CCE calculation at time points `timespace` to obtain coherence without second order effects:

```
>>> calc.compute(ts)
```

This will call `Simulator.cce_coherence` method with default keyword values.

Compute the coherence conventional CCE coherence with second order interactions between bath spins:

```
>>> calc.compute(ts, second_order=True)
```

Compute the coherence with conventional CCE with bath state sampling (over 10 states):

```
>>> calc.compute(ts, nbstates=10)
```

Compute the coherence with generalized CCE:

```
>>> calc.compute(ts, method='gcce')
```

This will call `Simulator.gcce_dm` method with default keyword values and obtain off diagonal element as $0\hat{\rho}_C 1$, where $\hat{\rho}_C$ is the density matrix of the qubit.

Compute the coherence with generalized CCE with bath state sampling (over 10 states):

```
>>> calc.compute(ts, method='gcce', nbstates=10)
```

Parameters

- **timespace** (*ndarray with shape (n,)*) – Time points at which compute the desired property.
- **quantity** (*str*) – Which quantity to compute. Case insensitive.

  Possible values:

  – 'coherence': compute coherence function.

  – 'noise': compute noise autocorrelation function.
- **method** (*str*) – Which implementation of CCE to use. Case insensitive.

  Possible values:

  – 'cce': conventional CCE, where interactions are mapped on 2 level pseudospin.

  – 'gcce': Generalized CCE where central spin is included in each cluster.
- **magnetic_field** (*ndarray with shape (3,)*) – Magnetic field vector of form (Bx, By, Bz).

  Default is **None**. Overrides `Simulator.magnetic_field` if provided.
- **D** (*float or ndarray with shape (3,3)*) – D (longitudinal splitting) parameter of central spin in ZFS tensor of central spin in kHz.

  OR

  Total ZFS tensor.

  Default is **None**. Overrides``Simulator.zfs`` if provided.
- **E** (*float*) – E (transverse splitting) parameter of central spin in ZFS tensor of central spin in kHz. Ignored if `D` is None or tensor.

  Default is 0.
- **pulses** (*list or int or* Sequence) – Number of pulses in CPMG sequence.

  OR

  Sequence of the instantaneous ideal control pulses. It can be provided as an instance of `Sequence` class (See documentation for pycce.Sequence).

  `pulses` can be provided as a list with tuples or dictionaries, each tuple or dictionary is used to initialize `Pulse` class instance.

  For example, for only central spin pulses the `pulses` argument can be provided as a list of tuples, containing:

  1. axis the rotation is about;
  2. angle of rotation;

3. (optional) Time before the pulse. Can be as fixed, as well as varied. If varied, it should be provided as an array with the same length as `timespace`.

E.g. for Hahn-Echo the `pulses` can be defined as `[('x', np.pi)]` or `[('x', np.pi, timespace / 2)]`.

Note: If delay is not provided in **all** pulses, assumes even delay of CPMG sequence. If only **some** delays are provided, assumes `delay = 0` in the pulses without delay.

Then total experiment is assumed to be:

tau – pulse – 2tau – pulse – … – 2tau – pulse – tau

Where tau is the delay between pulses.

The sum of delays at each time point should be less or equal to the total time of the experiment at the same time point, provided in `timespace` argument.

Warning: In conventional CCE calculations, only pi pulses on the central spin are allowed.

In the calculations of noise autocorrelation this parameter is ignored.

Default is **None**. Overrides``Simulator.pulses`` if provided.

- **alpha** (*int or ndarray with shape (2s+1,)*) – 0 state of the qubit in S_z basis or the index of eigenstate to be used as one.

  Default is **None**. Overrides``Simulator.alpha`` if provided.

- **beta** (*int or ndarray with shape (2s+1,)*) – 1 state of the qubit in S_z basis or the index of the eigenstate to be used as one.

  Default is **None**. Overrides``Simulator.beta`` if provided.

- **as_delay** (*bool*) – True if time points are delay between pulses (for equispaced pulses), False if time points are total time. Ignored in gCCE if `pulses` contains the time fractions. Conventional CCE calculations do not support custom time fractions.

  Default is **False**.

- **interlaced** (*bool*) – True if use hybrid CCE approach - for each cluster sample over states of the supercluster.

  Default is **False**.

- **state** (*ndarray with shape (2s+1,)*) – Initial state of the central spin, used in gCCE and noise autocorrelation calculations.

  Defaults to $\frac{1}{N}(0+1)$ if not set.

- **bath_state** (*array-like*) – List of bath spin states. If `len(shape) == 1`, contains I_z projections of I_z eigenstates. Otherwise, contains array of initial density matrices of bath spins.

  Default is **None**. If not set, the code assumes completely random spin bath (density matrix of each nuclear spin is proportional to identity, $\hat{\mathbb{1}}/N$).

- **nbstates** (*int*) – Number or random bath states to sample over.

  If provided, sampling of random states is carried and `bath_states` values are ignored.

Default is 0.

- **seed** (*int*) – Seed for random number generator, used in random bath states sampling.

  Default is **None**.

- **masked** (*bool*) – True if mask numerically unstable points (with coherence > 1) in the averaging over bath states.

  Note: It is up to user to check whether the possible instability is due to numerical error or unphysical assumptions of the calculations.

  Default is **True** for coherence calculations, **False** for noise calculations.

- **parallel_states** (*bool*) – True if to use MPI to parallelize the calculations of density matrix equally over present mpi processes for random bath state sampling calculations.

  Compared to `parallel` keyword, when this argument is True each process is given a fraction of random bath states. This makes the implementation faster. Works best when the number of bath states is divisible by the number of processes, `nbstates % size == 0`.

  Default is **False**.

- **fixstates** (*dict*) – If not None, shows which bath states to fix in random bath states. Each key is the index of bath spin, value - fixed $\hat{S}_z$ projection of the mixed state of nuclear spin.

  Default is **None**.

- **second_order** (*bool*) – True if add second order perturbation theory correction to the cluster Hamiltonian in conventional CCE. Relevant only for conventional CCE calculations.

  If set to True sets the qubit states as eigenstates of central spin Hamiltonian from the following procedure. If qubit states are provided as vectors in S_z basis, for each qubit state compute the fidelity of the qubit state and all eigenstates of the central spin and chose the one with fidelity higher than `level_confidence`. If such state is not found, raises an error.

  Warning: Second order corrections are not implemented as mean field.

  I.e., setting `second_order=True` and `nbstates != 0` leads to the calculation, when mean field effect is accounted only from dipolar interactions within the bath.

  Default is **False**.

- **level_confidence** (*float*) – Maximum fidelity of the qubit state to be considered eigenstate of the central spin Hamiltonian.

  Default is 0.95.

- **direct** (*bool*) – True if use direct approach (requires way more memory but might be more numerically stable). False if use memory efficient approach.

  Default is **False**.

- **parallel** (*bool*) – True if parallelize calculation of cluster contributions over different mpi processes.

  Default is **False**.

- **Returns** – ndarray: Computed property.

5.4 Pulse sequences

Documentation of the `Pulse` and `Sequence` classes, used in definition of the complicated pulse sequences.

class Pulse(*axis=None*, *angle=None*, *delay=None*, *bath_names=None*, *bath_axes=None*, *bath_angles=None*)

Class containing properties of each control pulse, applied to the system.

Parameters

- **axis** (*str*) – Axis of rotation of the central spin. Can be 'x', 'y', or 'z'. Default is None.
- **angle** (*float or str*) – Angle of rotation of central spin. Can be provided in rad, or as a string, containing fraction of pi: `'pi'`, `'pi/2'`, `'2*pi'` etc. Default is None.
- **delay** (*float or ndarray*) – Delay before the pulse or array of delays with the same shape as time points. Default is None.
- **bath_names** (*str or array-like of str*) – Name or array of names of bath spin types, impacted by the bath pulse. Default is None.
- **bath_axes** (*str or array-like of str*) – Axis of rotation or array of axes of the bath spins. Default is None. If `bath_names` is provided, but `bath_axes` and `bath_angles` are not, assumes the same axis and angle as the one of the central spin
- **bath_angles** (*float or str or array-like*) – Angle of rotation or array of axes of rotations of the bath spins.

Examples

```
>>> Pulse('x', 'pi')
Pulse((x, 3.14))
>>> Pulse('x', 'pi', bath_names=['13C', '14C'])
Pulse((x, 3.14), {13C: (x, 3.14), 14C: (x, 3.14)})
>>> import numpy as np
>>> Pulse('x', 'pi', delay=np.linspace(0, 1, 5), bath_names=['13C', '14C'], bath_
↪axes='x', bath_angles='pi/2')
Pulse((x, 3.14), {13C: (x, 1.57), 14C: (x, 1.57)}, t = [0.   0.25 0.5  0.75 1.  ])
```

axis

Axis of rotation of the central spin

Type str

angle

Angle of rotation of central spin in rad.

Type float

flip

True if the angle == pi.

Type bool

bath_names

Array of names of bath spin types, impacted by the bath pulse.

Type ndarray

bath_axes

Array of axes of rotation of the bath spins.

Type ndarray

bath_angles
Array of angles of rotation of the bath spins.

Type ndarray

rotation
Matrix representation of the pulse for the given cluster. Generated by `Sequence` object.

Type ndarray

property delay
Delay before the pulse or array of delays with the same shape as time points.

Type float or ndarray

class Sequence(*t=None*)
List-like object, which contains the sequence of the pulses.

Each element is a `Pulse` instance, which can be generated from either the tuple with positional arguments or from the dictionary, or set manually.

If delay is not provided in **all** pulses in the sequence, assume equispaced pulse sequence:

t - pulse - 2t - pulse - 2t - … - pulse - t

If only **some** delays are provided, assumes 0 delay in the pulses without delay provided.

Examples

```
>>> import numpy as np
>>> Sequence([('x', np.pi, 0),
>>>           {'axis': 'y', 'angle': 'pi', 'delay': np.linspace(0, 1, 3), 'bath_
↪names': '13C'},
>>>           Pulse('x', 'pi', 1)])
[Pulse((x, 3.14), t = 0), Pulse((y, 3.14), {13C: (y, 3.14)}, t = [0.  0.5 1. ]),␣
↪Pulse((x, 3.14), t = 1)]
```

small_sigma
Dictionary with Pauli matrices of the central spin

Type dict

delays
List with delays before each pulse or None if equispaced. Generated by `.generate_pulses` method.

Type list or None

rotations
List with matrix representations of the rotation from each pulse. Generated by `.generate_pulses` method.

Type list

append(*item*)
S.append(value) – append value to the end of the sequence

set_central_spin(*alpha*, *beta*)
Set Pauli matrices of the central spin.

Parameters

- **alpha** (*ndarray*) – 0 state of the qubit in S_z basis.
- **beta** (*ndarray*) – 1 state of the qubit in S_z basis.

generate_pulses(*dimensions=None*, *bath=None*, *vectors=None*, *central_spin=True*)

Generate list of matrix representations of the rotations, induced by the sequence of the pulses.

The rotations are stored in the `.rotation` attribute of the each `Pulse` object and in `Sequence.rotations`.

Parameters

- **dimensions** (*ndarray with shape (N,)*) – Array of spin dimensions in the system.
- **bath** (*BathArray with shape (n,)*) – Array of bath spins in the system.
- **vectors** (*ndarray with shape (N, 3, prod(dimensions), prod(dimensions))*) – Array with vectors of spin matrices for each spin in the system.
- **central_spin** (*bool*) – True if generate the rotations including central spin rotations. Default is True.

Returns

tuple containing:

- **list** or **None**: List with delays before each pulse or None if equispaced.
- **list**: List with matrix representations of the rotation from each pulse.

Return type tuple

bath_rotation(*vectors*, *axis*, *angle*)

Generate rotation of the bath spins with given spin vectors.

Parameters

- **vectors** (*ndarray with shape (n, 3, x, x)*) – Array of *n* bath spin vectors.
- **axis** (*str*) – Axis of rotation.
- **angle** (*float*) – Angle of rotation.

Returns Matrix representation of the spin rotation.

Return type ndarray with shape (x, x)

CHAPTER

SIX

HAMILTONIAN PARAMETERS INPUT

The total Hamiltonian of the system is set as:

$$\hat{H} = \hat{H}_S + \hat{H}_{SB} + \hat{H}_B$$

with

$$\hat{H}_S = \mathbf{SDS} + \mathbf{B}\gamma_S\mathbf{S}$$
$$\hat{H}_{SB} = \sum_i \mathbf{S}\mathbf{A}_i\mathbf{I}_i$$
$$\hat{H}_B = \sum_i \mathbf{I}_i\mathbf{P}_i\mathbf{I}_i + \mathbf{B}\gamma_i\mathbf{I}_i + \sum_{i>j} \mathbf{I}_i\mathbf{J}_{ij}\mathbf{I}_j$$

Where $\hat{H}_S$ is the Hamiltonian of the free central spin, $\hat{H}_{SB}$ denotes interactions between central spin and bath spin, nd $\hat{H}_B$ are intrinsic bath spin interactions:

- $\mathbf{D}$ ($\mathbf{P}$) is the self interaction tensor of the central spin (bath spin). For the electron spin, corresponds to the Zero field splitting (ZFS) tensor. For nuclear spins corresponds to the quadrupole interactions tensor.
- γ_i is the magnetic field interaction tensor of the i-spin describing the interaction of the spin and the external magnetic field.
- $\mathbf{A}$ is the interaction tensor between central and bath spins. In the case of nuclear spin bath, corresponds to the hyperfine couplings.
- $\mathbf{J}$ is the interaction tensor between bath spins.

Each of this terms can be defined within **PyCCE** framework as following.

In general, central spin properties are stored in the `Simulator` instance, bath properties are stored in the `BathArray` instance.

6.1 Central Spin Hamiltonian

The central spin Hamiltonian is provided as attributes of the `Simulator` object:

- $\mathbf{D}$ is set with `Simulator.set_zfs` method or during the initialization of the `Simulator` object either from observables *D* and *E* of the zero field splitting **OR** directly as tensor for the interaction $\mathbf{SDS}$ in kHz. By default is zero.

  Examples:

```
>>> c = Simulator(1)
>>> print(c.zfs)
[[ 0.  0.  0.]
 [ 0. -0.  0.]
 [ 0.  0.  0.]]
>>> c.set_zfs(D=1e6)
>>> print(c.zfs)
[[-333333.333       0.          0.   ]
 [      0.    -333333.333       0.   ]
 [      0.          0.     666666.667]]
```

- γ_S, the tensor describing the interaction of the spin and the external magnetic field in units of gyromagnetic ratio $\mathrm{rad}\cdot\mathrm{kHz}\cdot\mathrm{G}^{-1}$. By default is equal to the gyromagnetic ratio of the free electron spin, $-17609\,\mathrm{rad}\cdot\mathrm{ms}^{-1}\cdot\mathrm{G}^{-1}$.

  For the electron spin, it is proportional to g-tensor $\mathbf{g}$ as:

  $$\gamma_S = \mathbf{g}\frac{\mu_B}{\hbar},$$

  where μ_B is Bohr magneton.

  For the nuclear central spin, it is proportional to the chemical shift tensor σ and gyromagnetic ratio γ as:

  $$\gamma_S = \gamma(1-\sigma)$$

  Examples:

```
>>> c = Simulator(1)
>>> print(c.gyro)
-17608.59705
```

Note: While all other coupling parameters are given in the units of frequency, the gyromagnetic ratio (and therefore tensors coupling magnetic field with the spin) are conventionally given in the units of **angular** frequency and differ by 2π.

The magnetic field is set with with `Simulator.set_magnetic_field` method or during the initialization of the `Simulator` object in G.

6.2 Spin-Bath Hamiltonian

The interactions between central spin and bath spins and are provided in the `['A']` namefield of the `BathArray` object in kHz.

Interaction tensors can be either:

- Directly provided by setting the values of `bath['A']` in kHz for each bath spin.
- Approximated from magnetic point dipole–dipole interactions by calling `BathArray.from_point_dipole` method. Then the tensors are computed as:

  $$\mathbf{A}_j = -\gamma_S\gamma_j\frac{\hbar^2}{4\pi\mu_0}\left[\frac{3\vec{r_j}\otimes\vec{r_j} - |r_{ij}|^2 I}{|r_j|^5}\right]$$

  Where γ_j is gyromagnetic ratio of j spin, $\vec{r_j}$ is position of the bath spin, and I is 3x3 identity matrix. The default option when reading the bath by `Simulator` object.

- Approximated from the spin density distribution of the central spin by calling `BathArray.from_cube` method.

  Examples:

```
>>> bath = random_bath('13C', size=100, number=5, seed=1)
>>> print(bath)
[('13C', [  1.182,  45.046, -35.584], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]],␣
↪[[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [ 44.865, -18.817,  -7.667], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]],␣
↪[[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [ 32.77 ,  -9.08 ,   4.959], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]],␣
↪[[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [-47.244,  25.351,   3.814], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]],␣
↪[[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [-17.027,  28.843, -19.681], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]],␣
↪[[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])]
>>> bath['A'] = 1
>>> print(bath)
[('13C', [  1.182,  45.046, -35.584], [[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]],␣
↪[[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [ 44.865, -18.817,  -7.667], [[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]],␣
↪[[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [ 32.77 ,  -9.08 ,   4.959], [[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]],␣
↪[[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [-47.244,  25.351,   3.814], [[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]],␣
↪[[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [-17.027,  28.843, -19.681], [[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]],␣
↪[[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])]
>>> bath.from_point_dipole([0, 0, 0])
>>> print(bath)
[('13C', [  1.182,  45.046, -35.584], [[-0.659,  0.032, -0.025], [ 0.032,  0.559, -
↪0.963], [-0.025, -0.963,  0.1  ]], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [ 44.865, -18.817,  -7.667], [[ 1.558, -1.092, -0.445], [-1.092, -0.588, ␣
↪0.187], [-0.445,  0.187, -0.97 ]], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [ 32.77 ,  -9.08 ,   4.959], [[ 5.32 , -2.327,  1.271], [-2.327, -2.434, -
↪0.352], [ 1.271, -0.352, -2.886]], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [-47.244,  25.351,   3.814], [[ 1.06 , -1.   , -0.151], [-1.   , -0.268, ␣
↪0.081], [-0.151,  0.081, -0.792]], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
 ('13C', [-17.027,  28.843, -19.681], [[-0.903, -2.081,  1.42 ], [-2.081,  1.393, -
↪2.405], [ 1.42 , -2.405, -0.49 ]], [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])]
```

6.3 Bath Hamiltonian

The self interaction tensors of the bath spins ae stored in the `['Q']` namefield of the `BathArray` object. By default they are set to 0. They can be either:

- Directly provided by setting the values of `bath['Q']` in kHz for each bath spin.
- Computed from the electric field gradient (EFG) tensors at each bath spin position, using `BathArray.from_efg` method.

The gyromagnetic ratio γ_j of each bath spin type is stored in the `BathArray.types`.

The couplings between bath spins are assumed to follow point dipole-dipole interactions as:

$$\mathbf{P}_{ij} = -\gamma_i\gamma_j\frac{\hbar^2}{4\pi\mu_0}\left[\frac{3\vec{r_{ij}}\otimes\vec{r_i}j - |r_{ij}|^2 I}{|r_{ij}|^5}\right]$$

Where γ_i is gyromagnetic ratio of *i* tensor, I is 3x3 identity matrix, and $\vec{r_{ij}}$ is distance between two vectors.

However, user can define the interaction tensors for specific bath spin pairs stored in the `BathArray` instance. This can be achieved by:

- Calling `BathArray.add_interaction` method of the `BathArray` instance.
- Providing `InteractionsMap` instance as `imap` keyword to the `Simulator.read_bath`.

Examples:

```
>>> import numpy as np
>>> bath = random_bath('13C', size=100, number=5, seed=1)
>>> print(bath.types)
SpinDict(13C: (13C, 0.5, 6.7283))
>>> test_tensor = np.random.random((3, 3))
>>> bath.add_interaction(0, 1, (test_tensor + test_tensor.T) / 2)
>>> print(bath.imap[0, 1])
[[0.786 0.53  0.404]
 [0.53  0.821 0.366]
 [0.404 0.366 0.655]]
>>> print(bath.imap[0, 1])
[[0.786 0.53  0.404]
 [0.53  0.821 0.366]
 [0.404 0.366 0.655]]
```

CHAPTER

SEVEN

ELECTRONIC STRUCTURE OUTPUT

Each of the interfaces includes the function that should be used to read electronic structure calculations output into `BathArray` instance.

7.1 Quantum Espresso interface

read_qe(*pwfile*, *hyperfine=None*, *efg=None*, *s=1*, *pwtype=None*, *types=None*, *isotopes=None*, *center=None*, *center_type=None*, *rotation_matrix=None*, *rm_style='col'*, *find_isotopes=True*)

Function to read PW/GIPAW output from Quantum Espresso into BathArray.

Changes the names of the atoms to the most abundant isotopes if `find_isotopes` set to True. If that is not the desired outcome, user can define which isotopes to use using keyword isotopes. If `find_isotopes` is False, then keep the original names even when `isotopes` argument is provided.

Parameters

- **pwfile** (*str*) – Name of PW input or output file. If the file doesn't have proper extension, parameter pw_type should indicate the type.
- **hyperfine** (*str*) – name of the GIPAW hyperfine output.
- **efg** (*str*) – Name of the gipaw electric field tensor output.
- **s** (*float*) – Spin of the central spin. Default 1.
- **pwtype** (*str*) – Type of the `pwfile`. if not listed, will be inferred from extension of pwfile.
- **types** (SpinDict *or list of tuples*) – SpinDict containing SpinTypes of isotopes or input to make one.
- **isotopes** (*dict*) – Optional. Dictionary with entries: {"element" : "isotope"}, where "element" is the name of the element in DFT output, "isotope" is the name of the isotope.
- **center** (*ndarray of shape (3,)*) – Position of (0, 0, 0) point in input coordinates.
- **center_type** (*str*) – Type of the coordinates provided in center argument. Possible value include: 'bohr', 'angstrom', 'crystal', 'alat'. Default assumes the same as in PW file.
- **rotation_matrix** (*ndarray of shape (3,3)*) – Rotation matrix to rotate basis. For details see utilities.transform.
- **rm_style** (*str*) – Indicates how rotation matrix should be interpreted. Can take values "col" or "row". Default "col"
- **find_isotopes** (*bool*) – If true, sets isotopes instead of names of the atoms.

Returns BathArray containing atoms with hyperfine couplings and quadrupole tensors from QE output.

Return type *BathArray*

7.2 ORCA interface

read_orca(*fname*, *isotopes=None*, *types=None*, *center=None*, *find_isotopes=True*, *rotation_matrix=None*, *rm_style='col'*)

Function to read ORCA output containing the hyperfines couplings and EFG tensors.

if `find_isotopes` is set to True changes the names of the atoms to the most abundant isotopes. If that is not the desired outcome, user can define which isotopes to use using keyword isotopes.

Parameters

- **fname** (*str*) – file name of the ORCA output.
- **isotopes** (*dict*) – Optional. Dictionary with entries:

```
{"element" : "isotope"}
```

  where "element" is the name of the element in DFT output, "isotope" is the name of the isotope.

- **types** (SpinDict *or list of tuples*) – SpinDict containing SpinTypes of isotopes or input to make one.
- **center** (*ndarray of shape (3,)*) – position of (0, 0, 0) point in the DFT coordinates.
- **rotation_matrix** (*ndarray of shape (3,3)*) – Rotation matrix to rotate basis. For details see utilities.transform.
- **rm_style** (*str*) – Indicates how rotation matrix should be interpreted. Can take values "col" or "row". Default "col"
- **find_isotopes** (*bool*) – If true, sets isotopes instead of names of the atoms.

Returns Array of bath spins with hyperfine couplings and quadrupole tensors from Orca output.

Return type *BathArray*

CHAPTER

EIGHT

CCE CALCULATORS

Documentation for the calculator objects called by `Simulator` object.

8.1 Base class

class RunObject(*timespace*, *clusters*, *bath*, *magnetic_field*, *alpha*, *beta*, *state*, *spin*, *zfs*, *gyro*, *nbstates=None*, *bath_state=None*, *seed=None*, *masked=True*, *fixstates=None*, *projected_bath_state=None*, *parallel=False*, *direct=False*, *parallel_states=False*, ***kwargs*)

Abstract class of the CCE simulation runner.

Implements cluster correlation expansion, interlaced averaging, and sampling over random bath states. Requires definition of the following methods, from which the kernel will be automatically created:

- `.generate_hamiltonian(self)` method which, using the attributes of the `self` object, computes cluster hamiltonian stored in `self.cluster_hamiltonian`.
- `.compute_result(self)` method which, using the attributes of the `self`, computes the resulting quantity for the given cluster.

Alternatively, user can define the kernel manually. Then the following methods have to be overridden:

- `.kernel(self, cluster, *args, **kwargs)` method which takes indexes of the bath spins in the given cluster as a first positional argument. This method is required for usual CCE runs.
- `.interlaced_kernel(self, cluster, supercluster, *args, **kwargs)` method which takes indexes of the bath spins in the given cluster as a first positional argument, indexes of the supercluster as a second positional argument. This method is required for interlaced CCE runs.

Parameters

- **timespace** (*ndarray with shape (t,)*) – Time delay values at which to compute propagators.
- **clusters** (*dict*) – Clusters included in different CCE orders of structure `{int order: ndarray([[i,j],[i,j]])}`.
- **bath** (*BathArray with shape (n,)*) – Array of *n* bath spins.
- **magnetic_field** (*ndarray*) – Magnetic field of type `magnetic_field = np.array([Bx, By, Bz])`.
- **alpha** (*int or ndarray with shape (2s+1,)*) – 0 state of the qubit in S_z basis or the index of eigenstate to be used as one.
- **beta** (*int or ndarray with shape (2s+1,)*) – 1 state of the qubit in S_z basis or the index of the eigenstate to be used as one.

- **state** (*ndarray with shape (2s+1,)*) – Initial state of the central spin, used in gCCE and noise autocorrelation calculations. Defaults to $\frac{1}{N}(0+1)$ if not set **OR** if alpha and beta are provided as indexes.
- **spin** (*float*) – Value of the central spin.
- **zfs** (*ndarray with shape (3,3)*) – Zero Field Splitting tensor of the central spin.
- **gyro** (*float or ndarray with shape (3, 3)*) – Gyromagnetic ratio of the central spin

  OR

  tensor corresponding to interaction between magnetic field and central spin.
- **nbstates** (*int*) – Number of random bath states to sample over in bath state sampling runs.
- **bath_state** (*ndarray*) – Array of bath states in any accepted format.
- **seed** (*int*) – Seed for the random number generator in bath states sampling.
- **masked** (*bool*) – True if mask numerically unstable points (with result > result[0]) in the sampling over bath states False if not. Default True.
- **fixstates** (*dict*) – If provided, contains indexes of bath spins with fixed pure state for sampling runs and interlaced runs.

  Each key is the index of bath spin, value - fixed $\hat{I}_z$ projection of the **pure** $\hat{I}_z$ eigenstate of bath spin.
- **projected_bath_state** (*ndarray with shape (n,)*) – Array with z-projections of the bath spins states. Overridden in runs with random bath state sampling.
- **parallel** (*bool*) – True if parallelize calculation of cluster contributions over different mpi processes. Default False.
- **direct** (*bool*) – True if use direct approach in run (requires way more memory but might be more numerically stable). False if use memory efficient approach. Default False.
- **parallel_states** (*bool*) – True if use MPI to parallelize the calculations of density matrix for each random bath state.
- ****kwargs** – Additional keyword arguments to be set as the attributes of the given object.

result_operator(*b*, */*)

Operator which will combine the result of expansion,.

Default: `operator.imul`.

contribution_operator(*b*, */*)

Operator which will combine multiple contributions of the same cluster in the optimized approach.

Default: `operator.ipow`.

removal_operator(*b*, */*)

Operator which will remove subcluster contribution from the given cluster contribution. First argument cluster contribution, second - subcluster contribution.

Defalut: `operator.itruediv`.

addition_operator(*axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*, *initial=<no value>*, *where=<no value>*)

Group operation which will combine contributions from the different clusters into one contribution in the direct approach.

Default: `numpy.prod`.

nbstates
Number of random bath states to sample over in bath state sampling runs.

Type int

timespace
Time points at which result will be computed.

Type ndarray with shape (t,)

clusters
Clusters included in different CCE orders of structure `{int order: ndarray([[i,j],[i,j]])}`.

Type dict

bath
Array of *n* bath spins.

Type BathArray with shape (n,)

spin
Value of the central spin

Type float

zfs
Zero Field Splitting tensor of the central spin.

Type ndarray with shape (3, 3)

gyro
float or ndarray with shape (3, 3): Gyromagnetic ratio of the central spin

OR

tensor corresponding to interaction between magnetic field and central spin.

bath_state
Array of bath states in any accepted format.

Type ndarray

projected_bath_state
Array with z-projections of the bath spins states. Overridden in runs with random bath state sampling.

Type ndarray with shape (n,)

magnetic_field
Magnetic field of type `magnetic_field = np.array([Bx, By, Bz])`.

Type ndarray

initial_alpha
0 state of the qubit in S_z basis or the index of eigenstate to be used as one.

Type ndarray

initial_beta
1 state of the qubit in S_z basis or the index of eigenstate to be used as one.

Type ndarray

alpha
0 state of the qubit in S_z basis. If initially provided as index, generated as a state during the run.

Type ndarray

beta
1 state of the qubit in S_z basis. If initially provided as index, generated as a state during the run.

Type ndarray

state
Initial state of the central spin, used in gCCE and noise autocorrelation calculations.

Defaults to $\frac{1}{N}(0+1)$ if not set **OR** if alpha and beta are provided as indexes.

Type ndarray

parallel
True if parallelize calculation of cluster contributions over different mpi processes. Default False.

Type bool

parallel_states
True if use MPI to parallelize the calculations of density matrix for each random bath state.

Type bool

direct
True if use direct approach in run (requires way more memory but might be more numerically stable). False if use memory efficient approach. Default False.

Type bool

seed
Seed for the random number generator in bath states sampling.

Type int

masked
True if mask numerically unstable points (with result > result[0]) in the sampling over bath states False if not. Default True.

Type bool

fixstates
If provided, contains indexes of bath spins with fixed pure state for sampling runs and interlaced runs.

Each key is the index of bath spin, value - fixed $\hat{I}_z$ projection of the **pure** $\hat{I}_z$ eigenstate of bath spin.

Type dict

hamiltonian
central spin hamiltonian.

Type ndarray

energies
Eigen energies of the central spin hamiltonian.

Type ndarray

eigenvectors
Eigen states of the central spin hamiltonian.

Type ndarray

cluster
Array of the bath spins inside the given cluster.

Type *BathArray*

states
Array of the states of bath spins inside the given cluster.

Type ndarray

others
Array of the bath spins outside the given cluster.

Type *BathArray*

other_states
Array of the z-projections of the states of bath spins outside the given cluster.

Type ndarray

cluster_hamiltonian
Full hamiltonian of the given cluster. In conventional CCE, tuple with two projected hamiltonians.

Type *Hamiltonian* or tuple

result
Result of the calculation.

Type ndarray

preprocess()
Method which will be called before cluster-expanded run.

postprocess()
Method which will be called after cluster-expanded run.

kernel(*cluster*, **args*, ***kwargs*)
Central kernel that will be called in the cluster-expanded calculations

Parameters

- **cluster** (*ndarray*) – Indexes of the bath spins in the given cluster.
- ***args** – Positional arguments of the kernel.
- ****kwargs** – Keyword arguments of the kernel.

Returns Results of the calculations.

Return type ndarray

run(**args*, ***kwargs*)
Method that runs cluster-expanded single calculation.

Parameters

- ***args** – Positional arguments of the kernel.
- ****kwargs** – Keyword arguments of the kernel.

Returns Results of the calculations.

Return type ndarray

sampling_run(**args*, ***kwargs*)
Method that runs bath sampling calculations.

Parameters

- ***args** – Positional arguments of the kernel.
- ****kwargs** – Keyword arguments of the kernel.

Returns Results of the calculations.

Return type ndarray

`interlaced_kernel`(*cluster*, *supercluster*, **args*, ***kwargs*)
Central kernel that will be called in the cluster-expanded calculations with interlaced averaging of bath spin states.

Parameters

- **cluster** (*ndarray*) – Indexes of the bath spins in the given cluster.
- **supercluster** (*ndarray*) – Indexes of the bath spins in the supercluster of the given cluster. Supercluster is the union of all clusters in `.clusters` attribute, for which given cluster is a subset.
- ***args** – Positional arguments of the kernel.
- ****kwargs** – Keyword arguments of the kernel.

Returns Results of the calculations.

Return type ndarray

`interlaced_run`(**args*, ***kwargs*)
Method that runs cluster-expanded single calculation with interlaced averaging of bath spin states.

Parameters

- ***args** – Positional arguments of the interlaced kernel.
- ****kwargs** – Keyword arguments of the interlaced kernel.

Returns Results of the calculations.

Return type ndarray

`sampling_interlaced_run`(**args*, ***kwargs*)
Method that runs bath sampling calculations with interlaced averaging of bath spin states.

Parameters

- ***args** – Positional arguments of the interlaced kernel.
- ****kwargs** – Keyword arguments of the interlaced kernel.

Returns Results of the calculations.

Return type ndarray

`classmethod from_simulator`(*sim*, ***kwargs*)
Class method to generate `RunObject` from the properties of `Simulator` object.

Parameters

- **sim** (`Simulator`) – Object, whose properties will be used to initialize `RunObject` instance.
- ****kwargs** – Additional keyword arguments that will replace ones, recovered from the `Simulator` object.

Returns New instance of `RunObject` class.

Return type *RunObject*

`generate_supercluser_states`(*self*, *supercluster*)
Helper function to generate all possible pure states of the given supercluster.

Parameters

- **self** (RunObject) – Instance of the RunObject class, used in the calculation.
- **supercluster** (*ndarray with shape (n,)*) – Indexes of the bath spins in the supercluster.

Yields *ndarray with shape (n,)* – Pure state of the given supercluster.

8.2 Conventional CCE

propagators(*timespace*, *H0*, *H1*, *pulses*, *as_delay=False*)

Function to compute propagators U0 and U1 in conventional CCE.

Parameters

- **timespace** (*ndarray with shape (t,)*) – Time delay values at which to compute propagators.
- **H0** (*ndarray with shape (n, n)*) – Hamiltonian projected on alpha qubit state.
- **H1** (*ndarray with shape (n, n)*) – Hamiltonian projected on beta qubit state.
- **pulses** (*int or* Sequence) – Sequence of pulses.
- **as_delay** (*bool*) – True if time points are delay between pulses. False if time points are total time.

Returns

tuple containing:

- **ndarray with shape (t, n, n)**: Matrix representation of the propagator conditioned on the alpha qubit state for each time point.
- **ndarray with shape (t, n, n)**: Matrix representation of the propagator conditioned on the beta qubit state for each time point.

Return type tuple

compute_coherence(*H0*, *H1*, *timespace*, *N*, *as_delay=False*, *states=None*)

Function to compute cluster coherence function in conventional CCE.

Parameters

- **H0** (*ndarray*) – Hamiltonian projected on alpha qubit state.
- **H1** (*ndarray*) – Hamiltonian projected on beta qubit state.
- **timespace** (*ndarray*) – Time points at which to compute coherence function.
- **N** (*int*) – Number of pulses in CPMG.
- **as_delay** (*bool*) – True if time points are delay between pulses, False if time points are total time.
- **states** (*ndarray*) – ndarray of bath states in any accepted format.

Returns Coherence function of the central spin.

Return type ndarray

class CCE(**args*, *pulses=0*, *as_delay=False*, *second_order=False*, *level_confidence=0.95*, ***kwargs*)
Class for running conventional CCE simulations.

Note: Subclass of the `RunObject` abstract class.

Parameters

- ***args** – Positional arguments of the `RunObject`.
- **pulses** (*int or Sequence*) – number of pulses in CPMG sequence or instance of Sequence object. For now, only CPMG sequences are supported in conventional CCE simulations.
- **as_delay** (*bool*) – True if time points are delay between pulses, False if time points are total time.
- **second_order** (*bool*) – True if add second order perturbation theory correction to the cluster Hamiltonian. If set to True sets the qubit states as eigenstates of central spin Hamiltonian from the following procedure. If qubit states are provided as vectors in S_z basis, for each qubit state compute the fidelity of the qubit state and all eigenstates of the central spin and chose the one with fidelity higher than `level_confidence`. If such state is not found, raises an error.
- **level_confidence** (*float*) – Maximum fidelity of the qubit state to be considered eigenstate of the central spin Hamiltonian. Default 0.95.
- ****kwargs** – Keyword arguments of the `RunObject`.

initial_pulses
Input pulses

Type int or *Sequence*

pulses
If input Sequence contains only pi pulses at even delay, stores number of pulses. Otherwise stores full `Sequence`.

Type int or *Sequence*

as_delay
True if time points are delay between pulses, False if time points are total time.

Type bool

second_order
True if add second order perturbation theory correction to the cluster hamiltonian.

Type bool

level_confidence
Maximum fidelity of the qubit state to be considered eigenstate of the central spin hamiltonian.

Type float

energy_alpha
Eigen energy of the alpha state in the central spin Hamiltonian.

Type float

energy_beta
Eigen energy of the beta state in the central spin Hamiltonian.

Type float

`energies`
All eigen energies of the central spin Hamiltonian.

Type ndarray with shape (2s+1,)

`projections_alpha_all`
Array of vectors with spin operator matrix elements of type $[0\hat{S}_x i, 0\hat{S}_y i, 0\hat{S}_z i]$, where 0 is the alpha qubit state, i are all eigenstates of the central spin hamiltonian.

Type ndarray with shape (2s+1, 3)

`projections_beta_all`
Array of vectors with spin operator matrix elements of type $[1\hat{S}_x i, 1\hat{S}_y i, 1\hat{S}_z i]$, where 1 is the beta qubit state, i are all eigenstates of the central spin hamiltonian.

Type ndarray with shape (2s+1, 3)

`projections_alpha`
Vector with spin operator matrix elements of type $[0\hat{S}_x 0, 0\hat{S}_y 0, 0\hat{S}_z 0]$, where 0 is the alpha qubit state

Type ndarray with shape (3,)

`projections_beta`
Vectors with spin operator matrix elements of type $[1\hat{S}_x 1, 1\hat{S}_y 1, 1\hat{S}_z 1]$, where 1 is the beta qubit state.

Type ndarray with shape (3,)

`use_pulses`
True if use full `Sequence`. False if use only number of pulses.

Type bool

`preprocess()`
Method which will be called before cluster-expanded run.

`postprocess()`
Method which will be called after cluster-expanded run.

`generate_hamiltonian()`
Using the attributes of the `self` object, compute the two projected cluster hamiltonians.

Returns

Tuple containing:

- **Hamiltonian**: Cluster hamiltonian when qubit in the alpha state.
- **Hamiltonian**: Cluster hamiltonian when qubit in the alpha state.

Return type tuple

`compute_result()`
Using the attributes of the `self` object, compute the coherence function as overlap in the bath evolution.

Returns Computed coherence.

Return type ndarray

8.3 Generalized CCE

propagator(*timespace*, *hamiltonian*, *pulses=None*, *as_delay=False*)
Function to compute time propagator U.

Parameters

- **timespace** (*ndarray with shape (t,)*) – Time points at which to compute propagators.
- **hamiltonian** (*ndarray with shape (n, n)*) – Matrix representation of the cluster hamiltonian.
- **pulses** (Sequence) – Pulses as an instance of Sequence class with rotations already generated. Default is None.
- **as_delay** (*bool*) – True if time points are delay between pulses, False if time points are total time. Default is False.

Returns Array of propagators, evaluated at each time point in timespace.

Return type ndarray with shape (t, n, n)

compute_dm(*dm0*, *H*, *timespace*, *pulse_sequence=None*, *as_delay=False*, *states=None*)
Function to compute density matrix of the central spin, given Hamiltonian H.

Parameters

- **dm0** (*ndarray*) – Initial density matrix of central spin.
- **H** (*ndarray*) – Cluster Hamiltonian.
- **timespace** (*ndarray*) – Time points at which to compute density matrix.
- **pulse_sequence** (Sequence) – Sequence of pulses.
- **as_delay** (*bool*) – True if time points are delay between pulses, False if time points are total time.
- **states** (*ndarray*) – ndarray of bath states in any accepted format.

Returns Array of density matrices evaluated at all time points in timespace.

Return type ndarray

full_dm(*dm0*, *H*, *timespace*, *pulse_sequence=None*, *as_delay=False*)
A function to compute density matrix of the cluster, using hamiltonian H from the initial density matrix of the cluster.

Parameters

- **dm0** (*ndarray*) – Initial density matrix of the cluster
- **H** (*ndarray*) – Cluster Hamiltonian
- **timespace** (*ndarray*) – Time points at which to compute coherence function.
- **pulse_sequence** (Sequence) – Sequence of pulses.
- **as_delay** (*bool*) – True if time points are delay between pulses, False if time points are total time. Ignored if delays are provided in the pulse_sequence.

Returns Array of density matrices of the cluster, evaluated at the time points from timespace.

Return type ndarray

generate_dm0(*dm0*, *dimensions*, *states=None*)
A function to generate initial density matrix or statevector of the cluster. :param dm0: Initial density matrix of the central spin. :type dm0: ndarray :param dimensions: ndarray of bath spin dimensions. Last entry - electron spin dimensions. :type dimensions: ndarray :param states: ndarray of bath states in any accepted format. :type states: ndarray

Returns Initial density matrix of the cluster **OR** statevector if dm0 is vector and `states` are provided as list of pure states.

Return type ndarray

generate_pure_initial_state(*state0*, *dimensions*, *states*)
A function to generate initial state vector of the cluster with central spin.

Parameters

- **state0** (*ndarray*) – Initial state of the central spin.
- **dimensions** (*ndarray*) – ndarray of bath spin dimensions. Last entry - electron spin dimensions.
- **states** (*ndarray*) – ndarray of bath states in any accepted format.

Returns Initial state vector of the cluster.

Return type ndarray

gen_density_matrix(*states=None*, *dimensions=None*)
Generate density matrix from the ndarray of states.

Parameters

- **states** (*ndarray*) – Array of bath spin states. If None, assume completely random state. Can have the following forms:
  - array of the $\hat{I}_z$ projections for each spin. Assumes that each bath spin is in the pure eigenstate of $\hat{I}_z$.
  - array of the diagonal elements of the density matrix for each spin. Assumes mixed state and the density matrix for each bath spin is diagonal in $\hat{I}_z$ basis.
  - array of the density matrices of the bath spins.
- **dimensions** (*ndarray*) – array of bath spin dimensions. Last entry - electron spin dimensions.

Returns Density matrix of the system.

Return type ndarray

class gCCE(**args*, *as_delay=False*, *pulses=None*, ***kwargs*)
Class for running generalized CCE simulations.

Note: Subclass of the `RunObject` abstract class.

Parameters

- ***args** – Positional arguments of the `RunObject`.
- **pulses** (Sequence) – Sequence object, containing series of pulses, applied to the system.
- **as_delay** (*bool*) – True if time points are delay between pulses, False if time points are total time.

- ****kwargs** – Keyword arguments of the `RunObject`.

pulses
Sequence object, containing series of pulses, applied to the system.

Type *Sequence*

as_delay
True if time points are delay between pulses, False if time points are total time.

Type bool

dm0
Initial density matrix of the central spin.

Type ndarray with shape (2s+1, 2s+1)

normalization
Coherence at time 0.

Type float

zero_cluster
Coherence computed for the isolated central spin.

Type ndarray with shape (n,)

preprocess()
Method which will be called before cluster-expanded run.

postprocess()
Method which will be called after cluster-expanded run.

generate_hamiltonian()
Using the attributes of the `self` object, compute the cluster hamiltonian including the central spin.

Returns Cluster hamiltonian.

Return type *Hamiltonian*

compute_result()
Using the attributes of the `self` object, compute the coherence function of the central spin.

Returns Computed coherence.

Return type ndarray

8.4 Noise Autocorrelation

correlation_it_j0(*operator_i*, *operator_j*, *dm0_expanded*, *U*)
Function to compute correlation function of the operator i at time t and operator j at time 0

Parameters

- **operator_i** (*ndarray with shape (n, n)*) – Matrix representation of operator i.
- **operator_j** (*ndarray with shape (n, n)*) – Matrix representation of operator j.
- **dm0_expanded** (*ndarray with shape (n, n)*) – Initial density matrix of the cluster.
- **U** (*ndarray with shape (t, n, n)*) – Time evolution propagator, evaluated over t time points.

Returns Autocorrelation of the z-noise at each time point.

Return type ndarray with shape (t,)

compute_correlations(*nspin*, *dm0_expanded*, *U*, *central_spin=None*)

Function to compute correlations for the given cluster, given time propagator U.

Parameters

- **nspin** (BathArray) – BathArray of the given cluster of bath spins.
- **dm0_expanded** (*ndarray with shape (n, n)*) – Initial density matrix of the cluster.
- **U** (*ndarray with shape (t, n, n)*) – Time evolution propagator, evaluated over t time points.
- **central_spin** (*float*) – Value of the central spin.

Returns correlation of the Overhauser field, induced by the given cluster at each time point.

Return type ndarray with shape (t,)

class gCCENoise(**args*, ***kwargs*)

Class for running generalized CCE simulations of the noise autocorrelation function.

Note: Subclass of the RunObject abstract class.

Parameters

- ***args** – Positional arguments of the RunObject.
- ****kwargs** – Keyword arguments of the RunObject.

result_operator(*b*, */*)

Overridden operator which will combine the result of expansion: operator.iadd.

contribution_operator(*b*, */*)

Overridden operator which will combine multiple contributions of the same cluster in the optimized approach: operator.imul.

removal_operator(*b*, */*)

Overridden operator which remove subcluster contribution from the given cluster contribution: operator.isub.

addition_operator(*axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*, *initial=<no value>*, *where=<no value>*)

Overridden group operation which will combine contributions from the different clusters into one contribution in the direct approach: numpy.sum.

preprocess()

Method which will be called before cluster-expanded run.

postprocess()

Method which will be called after cluster-expanded run.

generate_hamiltonian()

Using the attributes of the self object, compute the cluster hamiltonian including the central spin.

Returns Cluster hamiltonian.

Return type *Hamiltonian*

compute_result()
Using the attributes of the self object, compute autocorrelation function of the noise from bath spins in the given cluster.

Returns Computed autocorrelation function.

Return type ndarray

class CCENoise(**args*, ***kwargs*)
Class for running conventional CCE simulations of the noise autocorrelation function.

Note: Subclass of the RunObject abstract class.

Warning: In general, for calculations of the autocorrelation function, better results are achieved with generalized CCE, which accounts for the evolution of the entangled state of the central spin.

Second order couplings between nuclear spins are not implemented.

Parameters

- ***args** – Positional arguments of the RunObject.
- ****kwargs** – Keyword arguments of the RunObject.

result_operator(*b*, /)
Overridden operator which will combine the result of expansion: operator.iadd.

contribution_operator(*b*, /)
Overridden operator which will combine multiple contributions of the same cluster in the optimized approach: operator.imul.

removal_operator(*b*, /)
Overridden operator which remove subcluster contribution from the given cluster contribution: operator.isub.

addition_operator(*axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*, *initial=<no value>*, *where=<no value>*)
Overridden group operation which will combine contributions from the different clusters into one contribution in the direct approach: numpy.sum.

preprocess()
Method which will be called before cluster-expanded run.

postprocess()
Method which will be called after cluster-expanded run.

generate_hamiltonian()
Using the attributes of the self object, compute the projected cluster hamiltonian, averaged for two qubit states.

Returns Cluster hamiltonian.

Return type *Hamiltonian*

compute_result()
Using the attributes of the self object, compute autocorrelation function of the noise from bath spins in the given cluster.

Returns Computed autocorrelation function.

Return type ndarray

8.5 Cluster-correlation Expansion Decorators

The way we find cluster in the code.

generate_clusters(*bath*, *r_dipole*, *order*, *r_inner=0*, *ignore=None*, *strong=False*, *nclusters=None*)
Generate clusters for the bath spins.

Parameters

- **bath** (BathArray) – Array of bath spins.
- **r_dipole** (*float*) – Maximum connectivity distance.
- **order** (*int*) – Maximum size of the clusters to find.
- **r_inner** (*float*) – Minimum connectivity distance.
- **ignore** (*list or str, optional*) – If not None, includes the names of bath spins which are ignored in the cluster generation.
- **strong** (*bool*) – Whether to find only completely interconnected clusters (default False).
- **nclusters** (*dict*) – Dictionary which contain maximum number of clusters of the given size. Has the form `n_clusters = {order: number}`, where `order` is the size of the cluster, `number` is the maximum number of clusters with this size.

  If provided, sorts the clusters by the strength of cluster interaction, equal to the lowest pairwise interaction in the cluster. Then the strongest `number` of clusters is taken.

Returns Dictionary with keys corresponding to size of the cluster, and value corresponds to ndarray of shape (matrix, N). Here matrix is the number of clusters of given size, N is the size of the cluster. Each row contains indexes of the bath spins included in the given cluster.

Return type dict

make_graph(*bath*, *r_dipole*, *r_inner=0*, *ignore=None*, *max_size=5000*)
Make a connectivity matrix for bath spins.

Parameters

- **bath** (BathArray) – Array of bath spins.
- **r_dipole** (*float*) – Maximum connectivity distance.
- **r_inner** (*float*) – Minimum connectivity distance.
- **ignore** (*list or str, optional*) – If not None, includes the names of bath spins which are ignored in the cluster generation.
- **max_size** (*int*) – Maximum size of the bath before less optimal (but less memory intensive) approach is used.

Returns Connectivity matrix.

Return type crs_matrix

connected_components(*csgraph*, *directed=False*, *connection='weak'*, *return_labels=True*)
Find connected components using `scipy.sparse.csgraph`. See documentation of `scipy.sparse.csgraph.connected_components`

find_subclusters(*maximum_order*, *graph*, *labels*, *n_components*, *strong=False*)
Find subclusters from connectivity matrix.

Parameters

- **maximum_order** (*int*) – Maximum size of the clusters to find.
- **graph** (*csr_matrix*) – Connectivity matrix.
- **labels** (*ndarray with shape (n,)*) – Array of labels of the connected components.
- **n_components** (*int*) – The number of connected components n.
- **strong** (*bool*) – Whether to find only completely interconnected clusters (default False).

Returns Dictionary with keys corresponding to size of the cluster, and value corresponds to ndarray of shape (matrix, N). Here matrix is the number of clusters of given size, N is the size of the cluster. Each row contains indexes of the bath spins included in the given cluster.

Return type dict

combine_clusters(*cs1*, *cs2*)

Combine two dictionaries with clusters.

Parameters

- **cs1** (*dict*) – First cluster dictionary with keys corresponding to size of the cluster, and value corresponds to ndarray of shape (matrix, N).
- **cs2** (*dict*) – Second cluster dictionary with the same structure.

Returns Combined dictionary with unique clusters from both dictionaries.

Return type dict

expand_clusters(*sc*)

Expand dict so each new cluster will include all possible additions of one more bath spin. This increases maximum size of the cluster by one.

Parameters **sc** (*dict*) – Initial clusters dictionary.

Returns Dictionary with expanded clusters.

Return type dict

find_valid_subclusters(*graph*, *maximum_order*, *nclusters=None*, *bath=None*, *strong=False*)

Find subclusters from connectivity matrix.

Parameters

- **maximum_order** (*int*) – Maximum size of the clusters to find.
- **graph** (*csr_matrix*) – Connectivity matrix.
- **nclusters** (*dict*) – Dictionary which contain maximum number of clusters of the given size.
- **bath** (BathArray) – Array of bath spins.
- **strong** (*bool*) – Whether to find only completely interconnected clusters (default False).

Returns Dictionary with keys corresponding to size of the cluster, and value corresponds to ndarray of shape (matrix, N). Here matrix is the number of clusters of given size, N is the size of the cluster. Each row contains indexes of the bath spins included in the given cluster.

Return type dict

General decorators that are used to expand kernel of the RunObject class or subclasses to the whole bath *via* CCE.

This module contains information about the way the cluster expansion is implemented in the package.

cluster_expansion_decorator(*_func=None*, ***, *result_operator=<built-in function imul>*, *contribution_operator=<built-in function ipow>*, *removal_operator=<built-in function itruediv>*, *addition_operator=<function prod>*)

Decorator for creating cluster correlation expansion of the method of `RunObject` class.

Parameters

- **_func** (*func*) – Function to expand.
- **result_operator** (*func*) – Operator which will combine the result of expansion (default: operator.imul).
- **contribution_operator** (*func*) – Operator which will combine multiple contributions of the same cluster (default: operator.ipow) in the optimized approach.
- **result_operator** – Operator which will combine the result of expansion (default: operator.imul).
- **removal_operator** (*func*) – Operator which will remove subcluster contribution from the given cluster contribution. First argument cluster contribution, second - subcluster contribution (default: operator.itruediv).
- **addition_operator** (*func*) – Group operation which will combine contributions from the different clusters into one contribution (default: np.prod).

Returns Expanded function.

Return type func

optimized_approach(*function*, *self*, **arg*, *result_operator=<built-in function imul>*, *contribution_operator=<built-in function ipow>*, ***kwarg*)

Optimized approach to compute cluster correlation expansion.

Parameters

- **function** (*func*) – Function to expand.
- **self** (RunObject) – Object whose method is expanded.
- ***arg** – list of positional arguments of the expanded function.
- **result_operator** (*func*) – Operator which will combine the result of expansion (default: operator.imul).
- **contribution_operator** (*func*) – Operator which will combine multiple contributions of the same cluster (default: operator.ipow).
- ****kwarg** – Dictionary containing all keyword arguments of the expanded function.

Returns Expanded function.

Return type func

direct_approach(*function*, *self*, **arg*, *result_operator=<built-in function imul>*, *removal_operator=<built-in function itruediv>*, *addition_operator=<function prod>*, ***kwarg*)

Direct approach to compute cluster correlation expansion.

Parameters

- **function** (*func*) – Function to expand.
- **self** (RunObject) – Object whose method is expanded.
- **result_operator** (*func*) – Operator which will combine the result of expansion (default: operator.imul).

- **removal_operator** (*func*) – Operator which will remove subcluster contribution from the given cluster contribution. First argument cluster contribution, second - subcluster contribution (default: operator.itruediv).
- **addition_operator** (*func*) – Group operation which will combine contributions from the different clusters into one contribution (default: np.prod).
- ****kwarg** – Dictionary containing all keyword arguments of the expanded function.

Returns Expanded method.

Return type func

interlaced_decorator(*_func=None*, ***, *result_operator=<built-in function imul>*, *contribution_operator=<built-in function ipow>*)

Decorator for creating interlaced cluster correlation expansion of the method of `RunObject` class.

Parameters

- **_func** (*func*) – Function to expand.
- **result_operator** (*func*) – Operator which will combine the result of expansion (default: operator.imul).
- **contribution_operator** (*func*) – Operator which will combine multiple contributions of the same cluster (default: operator.ipow) in the optimized approach.

Returns Expanded method.

Return type func

Decorators that are used to perform bath state sampling over the kernel of `RunObject`.

generate_bath_state(*bath*, *nbstates*, *seed=None*, *fixstates=None*, *parallel=False*)

Generator of the random *pure* $\hat{I}_z$ bath eigenstates.

Parameters

- **bath** (BathArray) – Array of bath spins.
- **nbstates** (*int*) – Number of random bath states to generate.
- **seed** (*int*) – Optional. Seed for RNG.
- **fixstates** (*dict*) – Optional. dict of which bath states to fix. Each key is the index of bath spin, value - fixed $\hat{I}_z$ projection of the mixed state of nuclear spin.

Yields *ndarray* – Array of `shape = len(bath)` containing z-projections of the bath spins states.

monte_carlo_method_decorator(*func*)

Decorator to sample over random bath states given function.

CHAPTER

NINE

HAMILTONIAN FUNCTIONS

9.1 Base Class

class Hamiltonian(*dimensions*, *vectors=None*)

Class containing properties of the Hamiltonian.

Essentially wrapper for ndarray with additional attributes of `dimensions` and `spins`.

Usual methods (e.g. `__setitem__` or `__getitem__`) access the `data` attribute.

Note: Algebraic operations with Hamiltonian will return ndarray instance.

Parameters **dimensions** (*array-like*) – array of the dimensions for each spin in the Hilbert space of the Hamiltonian.

dimensions

array of the dimensions for each spin in the Hilbert space of the Hamiltonian.

Type ndarray

spins

array of the spins, spanning the Hilbert space of the Hamiltonian.

Type ndarray

vectors

list with spin vectors of form `[[Ix, Iy, Iz], [Ix, Iy, Iz], ...]`.

Type list

data

matrix representation of the Hamiltonian.

Type ndarray

9.2 Total Hamiltonian

hamiltonian_wrapper(*_func=None*, ***, *projected=False*)

Wrapper with the general structure for the cluster Hamiltonian. Adds several additional arguments to the wrapped function.

Additional parameters:

- **central_spin** (*float*) – value of the central spin.

Parameters

- **_func** (*func*) – Function to be wrapped.
- **projected** (*bool*) – True if return two projected Hamiltonians, False if remove single not projected one.

Returns Wrapped function.

Return type func

projected_hamiltonian(*bath*, *vectors*, *projections_alpha*, *projections_beta*, *mfield*, *others=None*, *other_states=None*, *energy_alpha=None*, *energy_beta=None*, *energies=None*, *projections_alpha_all=None*, *projections_beta_all=None*)

Compute projected hamiltonian on state and beta qubit states. Wrapped function so the actual call does not follow the one above!

Parameters

- **bath** (BathArray) – array of all bath spins in the cluster.
- **projections_alpha** (*ndarray with shape (3,)*) – Projections of the central spin level alpha $[\hat{S}_x, \hat{S}_y, \hat{S}_z]$.
- **projections_beta** (*ndarray with shape (3,)*) – Projections of the central spin level beta. $[\hat{S}_x, \hat{S}_y, \hat{S}_z]$
- **mfield** (*ndarray with shape (3,)*) – Magnetic field of type `mfield = np.array([Bx, By, Bz])`.
- **others** (*BathArray with shape (m,)*) – array of all bath spins outside the cluster
- **other_states** (*ndarray with shape (m,) or (m, 3)*) – Array of Iz projections for each bath spin outside of the given cluster.
- **energy_alpha** (*float*) – Energy of the alpha state
- **energy_beta** (*float*) – Energy of the beta state
- **energies** (*ndarray with shape (2s-1,)*) – Array of energies of all states of the central spin.
- **projections_alpha_all** (*ndarray with shape (2s-1, 3)*) – Array of vectors of the central spin matrix elements of form:

$$[\alpha\hat{S}_x j, \alpha\hat{S}_y j, \alpha\hat{S}_z j],$$

where α is the alpha qubit state, and æ are all states.

- **projections_beta_all** (*ndarray with shape (2s-1, 3)*) – Array of vectors of the central spin matrix elements of form:

$$[\beta\hat{S}_x j, \beta\hat{S}_y j, \beta\hat{S}_z j],$$

where β is the beta qubit state, and æ are all states.

Returns

tuple containing:

- **Hamiltonian**: Hamiltonian of the given cluster, conditioned on the alpha qubit state.
- **Hamiltonian**: Hamiltonian of the given cluster, conditioned on the beta qubit state.

Return type tuple

total_hamiltonian(*bath*, *vectors*, *mfield*, *zfs=None*, *others=None*, *other_states=None*, *central_gyro=-17608.59705*)

Compute total Hamiltonian for the given cluster including mean field effect of all bath spins. Wrapped function so the actual call does not follow the one above!

Parameters

- **bath** (BathArray) – array of all bath spins.
- **mfield** (*ndarray with shape (3,)*) – Magnetic field of type `mfield = np.array([Bx, By, Bz])`.
- **others** (*BathArray with shape (m,)*) – array of all bath spins outside the cluster
- **other_states** (*ndarray with shape (m,) or (m, 3)*) – Array of Iz projections for each bath spin outside of the given cluster.
- **zfs** (*ndarray with shape (3,3)*) – Zero Field Splitting tensor of the central spin.
- **central_gyro** (*float or ndarray with shape (3,3)*) – gyromagnetic ratio of the central spin OR tensor corresponding to interaction between magnetic field and central spin.
- **central_spin** (*float*) – value of the central spin.

Returns hamiltonian of the given cluster, including central spin.

Return type *Hamiltonian*

9.3 Separate Terms

Documentation for the functions used to generate spin Hamiltonian for each cluster.

expanded_single(*ivec*, *gyro*, *mfield*, *self_tensor*, *detuning=0.0*)

Function to compute the single bath spin term.

Parameters

- **ivec** (*ndarray with shape (3, n, n)*) – Spin vector of the bath spin in the full Hilbert space of the cluster.
- **gyro** (*float or ndarray with shape (3, 3)*) –
- **mfield** (*ndarray wtih shape (3,)* – Magnetic field of type `mfield = np.array([Bx, By, Bz])`.
- **self_tensor** (*ndarray with shape (3, 3)*) – tensor of self-interaction of type IPI where I is bath spin.
- **detuning** (*float*) – Additional term of d*Iz allowing to simulate different energy splittings of bath spins.

Returns Single bath spin term.

Return type ndarray with shape (n, n)

dipole_dipole(*coord_1*, *coord_2*, *g1*, *g2*, *ivec_1*, *ivec_2*)
Compute dipole_dipole interactions between two bath spins.

Parameters

- **coord_1** (*ndarray with shape (3,)*) – Coordinates of the first spin.
- **coord_2** (*ndarray with shape (3,)*) – Coordinates of the second spin.
- **g1** (*float*) – Gyromagnetic ratio of the first spin.
- **g2** (*float*) – Gyromagnetic ratio of the second spin.
- **ivec_1** (*ndarray with shape (3, n, n)*) – Spin vector of the first spin in the full Hilbert space of the cluster.
- **ivec_2** (*ndarray with shape (3, n, n)*) – Spin vector of the second spin in the full Hilbert space of the cluster.

Returns Dipole-dipole interactions.

Return type ndarray with shape (n, n)

bath_interactions(*nspin*, *ivectors*)
Compute interactions between bath spins.

Parameters

- **nspin** (BathArray) – Array of the bath spins in the given cluster.
- **ivectors** (*array-like*) – array of expanded spin vectors, each with shape (3, n, n).

Returns All intrabath interactions of bath spins in the cluster.

Return type ndarray with shape (n, n)

bath_mediated(*nspin*, *ivectors*, *energy_state*, *energies*, *projections*)
Compute all hyperfine-mediated interactions between bath spins.

Parameters

- **nspin** (BathArray) – Array of the bath spins in the given cluster.
- **ivectors** (*array-like*) – array of expanded spin vectors, each with shape (3,n,n).
- **energy_state** (*float*) – Energy of the qubit state on which the interaction is conditioned.
- **energies** (*ndarray with shape (2s-1,)*) – Array of energies of all states of the central spin.
- **projections** (*ndarray with shape (2s-1, 3)*) – Array of vectors of the central spin matrix elements of form:

$$[i\hat{S}_x j, i\hat{S}_y j, i\hat{S}_z j],$$

where i are different states of the central spin.

Returns Hyperfine-mediated interactions.

Return type ndarray with shape (n, n)

conditional_hyperfine(*hyperfine_tensor*, *ivec*, *projections*)
Compute conditional hyperfine Hamiltonian.

Parameters

- **hyperfine_tensor** (*ndarray with shape (3, 3)*) – Tensor of hyperfine interactions of the bath spin.
- **ivec** (*ndarray with shape (3, n, n)*) – Spin vector of the bath spin in the full Hilbert space of the cluster.
- **projections** (*ndarray with shape (3,)*) – Array of vectors of the central spin matrix elements of form:

$$[i\hat{S}_x j, i\hat{S}_y j, i\hat{S}_z j],$$

where j are different states of the central spin. If $i = j$, produces the usual conditioned hyperfine interactions and just equal to projections of $\hat{S}_z$ of the central spin state $[\hat{S}_x, \hat{S}_y, \hat{S}_z]$.

If $i \neq j$, gives second order perturbation.

Returns Conditional hyperfine interaction.

Return type ndarray with shape (n, n)

hyperfine(*hyperfine_tensor*, *svec*, *ivec*)

Compute hyperfine interactions between central spin and bath spin.

Parameters

- **hyperfine_tensor** (*ndarray with shape (3, 3)*) – Tensor of hyperfine interactions of the bath spin.
- **svec** (*ndarray with shape (3, n, n)*) – Spin vector of the central spin in the full Hilbert space of the cluster.
- **ivec** (*ndarray with shape (3, n, n)*) – Spin vector of the bath spin in the full Hilbert space of the cluster.

Returns Hyperfine interaction.

Return type ndarray with shape (n, n)

self_central(*svec*, *mfield*, *zfs=None*, *gyro=- 17608.59705*)

Function to compute the central spin term in the Hamiltonian.

Parameters

- **svec** (*ndarray with shape (3, n, n)*) – Spin vector of the central spin in the full Hilbert space of the cluster.
- **mfield** (*ndarray wtih shape (3,)* – Magnetic field of type `mfield = np.array([Bx, By, Bz])`.
- **zfs** (*ndarray with shape (3, 3)*) – Zero Field Splitting tensor of the central spin.
- **gyro** (*float or ndarray with shape (3,3)*) – gyromagnetic ratio of the central spin OR tensor corresponding to interaction between magnetic field and central spin.

Returns Central spin term.

Return type ndarray with shape (n, n)

overhauser_central(*svec*, *others_hyperfines*, *others_state*)

Compute Overhauser field term on the central spin from all other spins, not included in the cluster.

Parameters

- **svec** (*ndarray with shape (3, n, n)*) – Spin vector of the central spin in the full Hilbert space of the cluster.

- **others_hyperfines** (*ndarray with shape (m, 3, 3)*) – Array of hyperfine tensors for all bath spins not included in the cluster.
- **others_state** (*ndarray with shape (m,) or (m, 3)*) – Array of Iz projections for each bath spin outside of the given cluster.

Returns Central spin Overhauser term.

Return type ndarray with shape (n, n)

overhauser_bath(*ivec*, *position*, *gyro*, *other_gyros*, *others_position*, *others_state*)

Compute Overhauser field term on the bath spin in the cluster from all other spins, not included in the cluster.

Parameters

- **ivec** (*ndarray with shape (3, n, n)*) – Spin vector of the bath spin in the full Hilbert space of the cluster.
- **position** (*ndarray with shape (3,)*) – Position of the bath spin.
- **gyro** (*float*) – Gyromagnetic ratio of the bath spin.
- **other_gyros** (*ndarray with shape (m,)*) – Array of the gyromagnetic ratios of the bath spins, not included in the cluster.
- **others_position** (*ndarray with shape (m, 3)*) – Array of the positions of the bath spins, not included in the cluster.
- **others_state** (*ndarray with shape (m,) or (m, 3)*) – Array of Iz projections for each bath spin outside of the given cluster.

Returns Bath spin Overhauser term.

Return type ndarray with shape (n, n)

eta_hamiltonian(*nspin*, *central_spin*, *alpha*, *beta*, *eta*)

EXPERIMENTAL. Compute hamiltonian with eta-term - gradually turn off or turn on the secular interactions for alpha and beta qubit states.

Parameters

- **nspin** (BathArray) – Array of the bath spins in the given cluster.
- **central_spin** (*float*) – central spin.
- **alpha** (*ndarray with shape (2s+1,)*) – Vector representation of the alpha qubit state in Sz basis.
- **beta** (*ndarray with shape (2s+1,)*) – Vector representation of the beta qubit state in Sz basis.
- **eta** (*float*) – Value of dimensionless parameter eta (from 0 to 1).

Returns Eta term.

Return type ndarray with shape (n, n)

CHAPTER

TEN

UTILITY FUNCTIONS

Here are the various functions used throughout the **PyCCE** code. There is no real structure in this section.

Module with helper functions to obtain CPMG coherence from the noise autocorrelation function.

`filterfunc`(*ts*, *tau*, *npulses*)

Time-domain filter function for the given CPMG sequence.

Parameters

- **ts** (*ndarray with shape (n,)*) – Time points at which filter function will be computed.
- **tau** (*float*) – Delay between pulses.
- **npulses** (*int*) – Number of pulses in CPMG sequence.

Returns Filter function for the given CPMG sequence

Return type ndarray with shape (n,)

`gaussian_phase`(*timespace*, *corr*, *npulses*, *units='khz'*)

Compute average random phase squared assuming Gaussian noise.

Parameters

- **timespace** (*ndarray with shape (n,)*) – Time points at which correlation function was computed.
- **corr** (*ndarray with shape (n,)*) – Noise autocorrelation function.
- **npulses** (*int*) – Number of pulses in CPMG sequence.
- **units** (*str*) – If units contain frequency or angular frequency ('rad' in `units`).

Returns Random phase accumulated by the qubit.

Return type ndarray with shape (n,)

`rotmatrix`(*initial_vector*, *final_vector*)

Generate 3D rotation matrix which applied on initial vector will produce vector, aligned with final vector.

Examples

```
>>> R = rotmatrix([0,0,1], [1,1,1])
>>> R @ np.array([0,0,1])
array([0.577, 0.577, 0.577])
```

Parameters

- **initial_vector** (*ndarray with shape(3,)*) – Initial vector.
- **final_vector** (*ndarray with shape (3,)*) – Final vector.

Returns Rotation matrix.

Return type ndarray with shape (3, 3)

expand(*matrix*, *i*, *dim*)

Expand matrix M from it's own dimensions to the total Hilbert space.

Parameters

- **matrix** (*ndarray with shape (dim[i], dim[i])*) – Inital matrix.
- **i** (*int*) – Index of the spin dimensions in `dim` parameter.
- **dim** (*ndarray*) – Array pf dimensions of all spins present in the cluster.

Returns Expanded matrix.

Return type ndarray with shape (prod(dim), prod(dim))

dimensions_spinvectors(*nspin*, *central_spin=None*)

Generate two arrays, containing dimensions of the spins in the cluster and the vectors with spin matrices.

Parameters

- **nspin** (*BathArray with shape (n,)*) – Array of the n spins within cluster.
- **central_spin** (*float, optional*) – If provided, include dimensions of the central spin with the total spin s.

Returns

tuple containing:

- **ndarray with shape (n,)**: Array with dimensions for each spin.
- **list**: List with vectors of spin matrices for each spin in the cluster (Including central spin if `central_spin` is not None). Each with shape (3, N, N) where `N = prod(dimensions)`.

Return type tuple

spinvec(*s*, *j*, *dimensions*)

Generate spin vector for the particle, containing 3 spin matrices in the total basis of the system.

Parameters

- **s** (*float*) – Spin of the particle.
- **j** (*j*) – Particle index in `dimensions` array.
- **dimensions** (*ndarray*) – Array with dimensions of all spins in the cluster.

Returns Vector of spin matrices for the given spin in the cluster.

Return type ndarray with shape (3, prod(dimensions), prod(dimensions))

generate_projections(*state_a*, *state_b=None*)

Generate vector with the spin projections of the given spin states:

$$[a\hat{S}_x b, a\hat{S}_y b, a\hat{S}_z b],$$

where a and b are the given spin states.

Parameters

- **state_a** (*ndarray*) – state a of the central spin in $\hat{S}_z$ basis.
- **state_b** (*ndarray*) – state b of the central spin in $\hat{S}_z$ basis.

Returns $[\hat{S}_x, \hat{S}_y, \hat{S}_z]$ projections.

Return type ndarray with shape (3,)

zfs_tensor(*D*, *E=0*)

Generate (3, 3) ZFS tensor from observable parameters D and E.

Parameters

- **D** (*float or ndarray with shape (3, 3)*) – Longitudinal splitting (D) in ZFS **OR** total ZFS tensor.
- **E** (*float*) – Transverse splitting (E) in ZFS.

Returns Total ZFS tensor.

Return type ndarray with shape (3, 3)

project_bath_states(*states*)

Generate projections of bath states on S_z axis from any type of states input. :param states: Array of bath spin states. :type states: array-like

Returns Array of S_z projections of the bath states

Return type ndarray

partial_inner_product(*avec*, *total*, *dimensions*, *index=- 1*)

Returns partial inner product $b = a\psi$, where a provided by `avec` contains degrees of freedom to be "traced out" and ψ provided by `total` is the total statevector.

Parameters

- **avec** (*ndarray with shape (a,)*) –
- **total** (*ndarray with shape (a*b,)*) –
- **dimensions** (*ndarray with shape (n,)*) –
- **()** (*index*) –

Returns:

class SpinMatrix(*s*)

Class containing the spin matrices in Sz basis.

Parameters **s** (*float*) – Total spin.

class MatrixDict(**spins*)

Class for storing the SpinMatrtix objects.

keys() → a set-like object providing a view on D's keys

CHAPTER

ELEVEN

ES INTERFACE

Each of the interfaces uses subclass of the `DFTCoordinates` class to parse the output.

Note: The interfaces are in beta stage. Please let us know if you encounter any errors.

11.1 Quantum Espresso

class PWCoordinates(*filename*, *pwtype=None*, *to_angstrom=False*)

Coordinates of the system from the PW data of Quantum Espresso. Subclass of the DFTCoordinates.

With initiallization reads either output or input of PW module of QE.

Parameters

- **filename** (*str*) – name of the PW input or output.
- **pwfile** (*str*) – Name of PW input or output file. If the file doesn't have proper extension, parameter pw_type should indicate the type.
- **pwtype** (*str*) – Type of the coord_f. if not listed, will be inferred from extension of pwfile.
- **to_angstrom** (*bool*) – True if automatically convert the units of `cell` and `coordinates` to Angstrom.

parse_output(*filename*, *to_angstrom=False*)

Method to read coordinates of atoms from PW output into the PWCoordinates instance.

Parameters

- **filename** (*str*) – the name of the output file.
- **to_angstrom** (*bool*) – True if automatically convert the units of `cell` and `coordinates` to Angstrom.

Returns None

parse_input(*filename*, *to_angstrom=False*)

Method to read coordinates of atoms from PW input into the PWCoordinates instance.

Parameters

- **filename** (*str*) – the name of the output file.
- **to_angstrom** (*bool*) – True if automatically convert the units of `cell` and `coordinates` to Angstrom.

cell_from_system(*sdict*)
Function to obtain cell from namelist SYSTEM read from PW input.

Parameters **sdict** (*dict*) – Dictinary generated from namelist SYSTEM of PW input.

Returns

Cell is 3x3 matrix with entries:

```
[[a_x b_x c_x]
 [a_y b_y c_y]
 [a_z b_z c_z]],
```

where a, b, c are crystallographic vectors, and x, y, z are their coordinates in the cartesian reference frame.

Return type ndarray with shape (3,3)

celldms_from_abc(*ibrav*, *abc_list*)
Obtain celldms from ibrav value and a, b, c, cosab, cosac, cosbc parameters.

Using ibrav value and abc parameters from PW input generate celldm array, necessary to construct cell parameters. For details about abc and ibrav values see PW input documentation.

Parameters

- **ibrav** (*int*) – ibrav parameter of PW input.
- **abc_list** (*list*) – List, of 6 parameters: a, b, c, cosab, cosac, cosbc

Returns list of 6 values, from which cell can be generated.

Return type celldm (list)

read_gipaw_tensors(*lines*, *keyword=None*, *start=None*, *conversion=1*)
Helper function to read GIPAW tensors from the list of lines.

Parameters

- **lines** (*list of str*) – List of strings contraining lines from the file. Output of open(file).readlines().
- **keyword** (*str*) – Keyword in the line which indicates the beginning of the tensor data block.
- **start** (*int*) – Index of the line which indicates the beginning of the tensor data block.
- **conversion** (*float*) – Conversion factor from GIPAW units to the ones, used in this package.

Returns Array of tensors.

Return type ndarray with shape (n, 3, 3)

read_hyperfine(*filename*, *spin=1*)
Function to read hyperfine couplings from GIPAW output.

Parameters

- **filename** (*str*) – Name of the GIPAW hyperfine output.
- **spin** (*float*) – Spin of the central spin. Default 1.

Returns

Tuple containing:

- *ndarray with shape (n,)*: Array of Fermi contact terms.

- *ndarray with shape (n, 3,3)*: Array of spin dipolar hyperfine tensors.

Return type tuple

read_efg(*filename*)
Function to read electric field gradient tensors from GIPAW output.

Parameters **filename** (*str*) – Name of the GIPAW EFG-containing output.

Returns Array of EFG tensors.

Return type ndarray with shape (n, 3,3)

read_qe_namelists(*input_string*)
Read Fortran-like namelists from the large string.

Parameters **input_string** (*str*) – String representation of the QE input file.

Returns Dictionary, containing dicts for each namelist found in the input string.

Return type dict

get_ctype(*lin*)
Get coordinates type from the line of QE input/output.

Parameters **str** – Line from QE input/output containing string with coordinates type.

Returns type of the coordinates.

Return type str

11.2 ORCA

class ORCACoordinates(*orca_output*)
Coordinates of the system from the ORCA output. Subclass of the DFTCoordinates.

With initialization reads output of the ORCA.

Parameters **orca_output** (*str or list of str*) – either name of the output file or list of lines read from that file.

alat
The lattice parameter in angstrom.

Type float

cell
cell is 3x3 matrix with entries:

$$[[a_x\ b_x\ c_x]$$
$$[a_y\ b_y\ c_y]$$
$$[a_z\ b_z\ c_z]]$$

where a, b, c are crystallographic vectors, and x, y, z are their coordinates in the cartesian reference frame.

Type ndarray with shape (3, 3)

coordinates
array with the coordinates of atoms in the cell.

Type ndarray with shape (n, 3)

names
array with the names of atoms in the cell.

Type ndarray with shape (n,)

cell_units
Units of cell coordinates: ‘bohr’, ‘angstrom’, ‘alat’.

Type str

coordinates_units
Units of atom coordinates: ‘crystal’, ‘bohr’, ‘angstrom’, ‘alat’.

Type str

read_output(*orca_output*)
Method to read coordinates of atoms from ORCA output into the ORCACoordinates instance.

Parameters **orca_output** (*str or list of str*) – either name of the output file or list of lines read from that file.

11.3 Base class

class DFTCoordinates
Abstract class of a container of the DFT output coordinates.

alat
The lattice parameter in angstrom.

Type float

cell
cell is 3x3 matrix with entries:

$$[[a_x\ b_x\ c_x]$$
$$[a_y\ b_y\ c_y]$$
$$[a_z\ b_z\ c_z]]$$

where a, b, c are crystallographic vectors and x, y, z are their coordinates in the cartesian reference frame.

Type ndarray with shape (3, 3)

coordinates
Array with the coordinates of atoms in the cell.

Type ndarray with shape (n, 3)

names
Array with the names of atoms in the cell.

Type ndarray with shape (n,)

cell_units
Units of cell coordinates: ‘bohr’, ‘angstrom’, ‘alat’.

Type str

coordinates_units
Units of atom coordinates: ‘crystal’, ‘bohr’, ‘angstrom’, ‘alat’.

Type str

to_angstrom(*inplace=False*)
Method to transform cell and coordinates units to angstroms.

Parameters **inplace** (*bool*) – if True changes attributes inplace. Otherwise returns copy.

Returns Instance of the subclass with units of coordinates and cell of Angstroms.

Return type *DFTCoordinates* or subclass

get_angstrom(*coordinate*, *units*)
Change given coordinates to angstrom.

Parameters

- **coordinates** (*ndarray with shape (n, 3) or (3,)*) – Coordinates to change.
- **units** (*str*) – Initial units of the coordinates.

Returns Coordinates in angstrom.

Return type ndarray (n, 3)

change_to_angstrom(*coordinates*, *units*, *alat=None*, *cell=None*)
Change coordinates to angstrom.

Parameters

- **coordinates** (*ndarray with shape (n, 3) or (3,)*) – Coordinates to change.
- **units** (*str*) – Initial units of the coordinates.
- **alat** (*float*) – The lattice parameter in angstrom.
- **cell** (*ndarray with shape (3,3)*) – cell is 3x3 matrix with entries:

$$[[a_x\ b_x\ c_x]$$
$$[a_y\ b_y\ c_y]$$
$$[a_z\ b_z\ c_z]]$$

where a, b, c are crystallographic vectors, and x, y, z are their coordinates in the cartesian reference frame.

Returns Coordinates in angstrom.

Return type ndarray with shape (n, 3)

fortran_value(*value*)
Get value from Fortran-type variable.

Parameters **value** (*str*) – Value read from Fortran-type input.

Returns value in Python format.

Return type value (bool, str, float)

yield_index(*word*, *lines*, *start=0*, *case_sensitive=False*)
Generator which yields indexes of the lines containing specific word.

Parameters

- **word** (*str*) – Word to find in the line.
- **lines** (*list of str*) – List of strings contraining lines from the file. Output of open(file).readlines().
- **start** (*int*) – First index from which to start search.
- **case_sensitive** (*bool*) – If True looks for the exact match. Otherwise the search is case insensitive.

Yields *i (int)* – Index of the line containing word.

find_first_index(*word*, *lines*, *start=0*, *case_sensitive=False*)
Function to find first appearance of the index in the list of lines.

Parameters

- **word** (*str*) – Word to find in the line.
- **lines** (*list of str*) – List of strings contraining lines from the file. Output of open(file).readlines().
- **start** (*int*) – First index from which to start search.
- **case_sensitive** (*bool*) – If True looks for the exact match. Otherwise the search is case insensitive.

Returns Index of the first line from the start containing word.

Return type i (int)

set_isotopes(*array*, *isotopes=None*, *inplace=True*, *spin_types=None*)
Function to set the most common isotopes for the array containing DFT output. If some other isotope is specified, the A tensors are scaled accordingly.

Parameters

- **array** (BathArray) – Array with DFT spins.
- **isotopes** (*dict*) – Dictionary with chosen isotopes.
- **inplace** (*bool*) – True if change the array inplace.
- **spin_types** (SpinDict) – If provided, allows for custom defined SpinType instances.

Returns Array with DFT spins with correct isotopes.

Return type array (*BathArray*)

PyCCE is an open source Python library to simulate the dynamics of a spin qubit interacting with a spin bath using the cluster-correlation expansion (CCE) method.

CHAPTER

TWELVE

INSTALLATION

The recommended way to install **PyCCE** is to use **pip**:

```
$ pip install pycce
```

Otherwise you can install **PyCCE** directly using the source code. First copy the repository to the desired folder:

```
$ git clone https://github.com/foxfixfax/pycce.git
```

Then, execute **pip** in the folder containing **setup.py**:

```
$ pip install .
```

or run the python install command:

```
$ python setup.py install
```

CHAPTER

THIRTEEN

REQUIREMENTS

The following modules are required to run **PyCCE**.

- Python (version >= 3.6).
- NumPy (version >= 1.16).
- SciPy.
- Numba (version >= 0.50).
- Atomic Simulation Environment (ASE).
- Pandas.

PyCCE inherently supports parallelization with the **mpi4py** package, which requires the installation of MPI. However, for serial implementation the **mpi4py** is not required.

CHAPTER

FOURTEEN

HOW TO CITE

If you make use of **PyCCE** in a scientific publication, please cite the following paper:

> Mykyta Onizhuk and Giulia Galli. "PyCCE: A Python Package for Cluster Correlation Expansion Simulations of Spin Qubit Dynamic" Adv. Theory Simul. 2021, 2100254 https://onlinelibrary.wiley.com/doi/10.1002/adts.202100254

PYTHON MODULE INDEX

p

INDEX

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

X

Y

Z