QuEmb

Van Voorhis Group

Aug 18, 2025

CONTENTS

QuEmb is a robust framework designed to implement the Bootstrap Embedding (BE) method, efficiently treating electron correlation in molecules, surfaces, and solids. This repository contains the Python implementation of the BE methods, including periodic bootstrap embedding. The code leverages PySCF library for quantum chemistry calculations and utlizes Python's multiprocessing module to enable parallel computations in high-performance computing environments.

QuEmb includes two libraries: `quemb.molbe` and `quemb.kbe`. The `quemb.molbe` library implements BE for molecules and supramolecular complexes, while the `quemb.kbe` library is designed to handle periodic systems such as surfaces and solids using periodic BE.

CHAPTER

ONE

REFERENCES

1. OR Meitei, T Van Voorhis, Periodic bootstrap embedding, JCTC 19 3123 2023
2. OR Meitei, T Van Voorhis, Electron correlation in 2D periodic systems, arXiv:2308.06185
3. HZ Ye, HK Tran, T Van Voorhis, Bootstrap embedding for large molecular systems, JCTC 16 5035 2020

1.1 Installation

1.1.1 Prerequisites

- Python `3.10 <= version < 3.13`
- PySCF library
- Numpy
- Scipy
- libDMET (required for periodic BE)
- Wannier90 ##

## `Wannier90` code is optional and only necessary to use Wannier functions in periodic code.

The required dependencies, with the exception of the optional `Wannier90`, are automatically installed by `pip`.

1.1.2 Documentation

Option 1: `PDF version`

Option 2: Build the documentation locally.

```
cd docs
make html
```

or

```
cd docs
make latexpdf
```

1.1.3 Installation

One can just `pip install` directly from the Github repository

```
pip install git+https://https://github.com/troyvvgroup/quemb
```

Alternatively one can manually clone and install as in

```
git clone --recurse-submodules https://https://github.com/troyvvgroup/quemb
cd quemb
pip install .
```

1.1.4 Optional dependencies

If you want to use the ORCA backend for Hartree-Fock you need to install ORCA from here. This requires a registration and is free for academic use. In addition you need to install the python interface via:

```
pip install orca-pi
```

1.2 API reference

molbe	
kbe	
shared	

1.2.1 quemb.molbe

Modules

autofrag	
be_parallel	
chemfrag	This module implements the fragmentation of molecular and periodic systems based on chemical connectivity that uses the overlap of tabulated van der Waals radii.
eri_onthefly	
eri_sparse_DF	
fragment	
graphfrag	
helper	

continues on next page

Table 2 – continued from previous page

lo	
mbe	
mf_interfaces	
misc	
opt	
pfrag	
solver	
ube	Bootstrap Embedding Calculation with an Unrestricted Hartree-Fock Bath

quemb.molbe.autofrag

Functions

autogen(mol[, frozen_core, n_BE, ...])	Automatic molecular partitioning

quemb.molbe.autofrag.autogen

`quemb.molbe.autofrag.`**`autogen`**(*mol*, *frozen_core=True*, *n_BE=2*, *write_geom=False*, *iao_valence_basis=None*, *iao_valence_only=False*, *print_frags=True*)

Automatic molecular partitioning

Partitions a molecule into overlapping fragments as defined in BE atom-based fragmentations. It automatically detects branched chemical chains and ring systems and partitions accordingly. For efficiency, it only checks two atoms for connectivity (chemical bond) if they are within 3.5 Angstrom. This value is hardcoded as normdist. Two atoms are defined as bonded if they are within 1.8 Angstrom (1.2 for Hydrogen atom). This is also hardcoded as bond & hbond.

Parameters

- **mol** (*Mole*) – This is required for the options, 'autogen', and 'chain' as frag_type.
- **frozen_core** (*bool, optional*) – Whether to invoke frozen core approximation. Defaults to True.
- **n_BE** (*int, optional*) – Specifies the order of bootstrap calculation in the atom-based fragmentation, i.e. BE(n). Supported values are 1, 2, 3, and 4 Defaults to 2.
- **write_geom** (*bool, optional*) – Whether to write a 'fragment.xyz' file which contains all the fragments in Cartesian coordinates. Defaults to False.
- **iao_valence_basis** (*str, optional*) – Name of minimal basis set for IAO scheme. 'sto-3g' is sufficient for most cases. Defaults to None.
- **iao_valence_only** (*bool, optional*) – If True, all calculations will be performed in the valence basis in the IAO partitioning. This is an experimental feature. Defaults to False.
- **print_frags** (*bool, optional*) – Whether to print out the list of resulting fragments. Defaults to True.

Classes

AutogenArgs([iao_valence_only])	Additional arguments for autogen
FragPart(*, mol, frag_type, n_BE, ...)	Data structure to hold the result of BE fragmentations.

quemb.molbe.autofrag.AutogenArgs

`class quemb.molbe.autofrag.`**`AutogenArgs`**(*iao_valence_only=False*)

Additional arguments for autogen

Parameters

`iao_valence_only` (`bool`) – If this option is set to True, all calculation will be performed in the valence basis in the IAO partitioning. This is an experimental feature.

Attributes

`iao_valence_only:` `bool`

Methods

__init__([iao_valence_only])	Method generated by attrs for class AutogenArgs.

quemb.molbe.autofrag.AutogenArgs.__init__

`AutogenArgs.`**`__init__`**(*iao_valence_only=False*)

Method generated by attrs for class AutogenArgs.

quemb.molbe.autofrag.FragPart

`class quemb.molbe.autofrag.`**`FragPart`**(** (Keyword-only parameters separator (PEP 3102)), mol, frag_type, n_BE, AO_per_frag, AO_per_edge_per_frag, ref_frag_idx_per_edge_per_frag, relAO_per_edge_per_frag, relAO_in_ref_per_edge_per_frag, relAO_per_origin_per_frag, weight_and_relAO_per_center_per_frag, motifs_per_frag, origin_per_frag, H_per_motif, add_center_atom, frozen_core, iao_valence_basis, iao_valence_only*)

Data structure to hold the result of BE fragmentations.

Attributes

`mol:` `TypeVar(_T_chemsystem, Mole, Cell)`

The full molecule.

`frag_type:` `Literal['chemgen', 'graphgen', 'autogen']`

The algorithm used for fragmenting.

`n_BE:` `int`

The level of BE fragmentation, i.e. 1, 2, …

`AO_per_frag:` `list[list[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]`

This is a list over fragments and gives the global orbital indices of all atoms in the fragment. These are ordered by the atoms in the fragment.

When using IAOs this refers to the large/working basis.

`AO_per_edge_per_frag: list[list[list[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]`

The global orbital indices, including hydrogens, per edge per fragment.

When using IAOs this refers to the valence/small basis.

`ref_frag_idx_per_edge_per_frag: list[list[NewType(FragmentIdx, integer)]]`

Reference fragment index per edge: A list over fragments: list of indices of the fragments in which an edge of the fragment is actually a center. The edge will be matched against this center. For fragments A, B: the A'th element of `.center`, if the edge of A is the center of B, will be B.

`relAO_per_edge_per_frag: list[list[list[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]`

The relative orbital indices, including hydrogens, per edge per fragment. The index is relative to the own fragment.

When using IAOs this refers to the valence/small basis.

`relAO_in_ref_per_edge_per_frag: list[list[list[NewType(RelAOIdxInRef, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]`

The relative atomic orbital indices per edge per fragment. **Note** for this variable relative means that the AO indices are relative to the other fragment where the edge is a center.

When using IAOs this refers to the valence/small basis.

`relAO_per_origin_per_frag: list[list[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]`

of the motif list for each fragment, this is always a `list(range(0, n))`

When using IAOs this refers to the valence/small basis.

`weight_and_relAO_per_center_per_frag: list[tuple[float, list[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]`

The first element is a float, the second is the list The float weight makes only sense for democratic matching and is currently 1.0 everywhere anyway. We concentrate only on the second part, i.e. the list of indices. This is a list whose entries are sequences containing the relative orbital index of the center sites within a fragment. Relative is to the own fragment.

When using IAOs this refers to the large/working basis.

`motifs_per_frag: list[list[NewType(MotifIdx, NewType(AtomIdx, int))]]`

The motifs/heavy atoms in each fragment, in order. Each are labeled based on the global atom index. It is ordered by origin, centers, edges!

`origin_per_frag: list[NewType(OriginIdx, NewType(CenterIdx, NewType(MotifIdx, NewType(AtomIdx, int))))]`

The origin for each fragment. (Note that for conventional BE there is just one origin per fragment)

`H_per_motif: Sequence[list[NewType(AtomIdx, int)]]`

`add_center_atom: list[list[NewType(CenterIdx, NewType(MotifIdx, NewType(AtomIdx, int)))]]`

A list over fragments. For each fragment a list of centers that are not the origin of that fragment.

frozen_core: bool

iao_valence_basis: str | None

iao_valence_only: bool

If this option is set to True, all calculation will be performed in the valence basis in the IAO partitioning. This is an experimental feature.

n_frag: int

ncore: int | None

no_core_idx: list[int] | None

core_list: list[int] | None

Methods

`__init__`(*, mol, frag_type, n_BE, ...)	Method generated by attrs for class FragPart.
`all_centers_are_origins`()	
`reindex`(idx)	
`to_Frags`(I, eri_file[, unrestricted])	

quemb.molbe.autofrag.FragPart.__init__

FragPart.**__init__**(**, mol, frag_type, n_BE, AO_per_frag, AO_per_edge_per_frag, ref_frag_idx_per_edge_per_frag, relAO_per_edge_per_frag, relAO_in_ref_per_edge_per_frag, relAO_per_origin_per_frag, weight_and_relAO_per_center_per_frag, motifs_per_frag, origin_per_frag, H_per_motif, add_center_atom, frozen_core, iao_valence_basis, iao_valence_only*)

Method generated by attrs for class FragPart.

quemb.molbe.autofrag.FragPart.all_centers_are_origins

FragPart.**all_centers_are_origins**()

Return type
bool

quemb.molbe.autofrag.FragPart.reindex

FragPart.**reindex**(*idx*)

Return type
Self

quemb.molbe.autofrag.FragPart.to_Frags

FragPart.**to_Frags**(*I, eri_file, unrestricted=False*)

Return type
Frags

quemb.molbe.be_parallel

Functions

be_func_parallel(pot, Fobjs, Nocc, solver, ...)	Embarrassingly Parallel High-Level Computation
be_func_parallel_u(pot, Fobjs, solver, enuc)	Embarrassingly Parallel High-Level Computation
run_solver(h1, dm0, scratch_dir, dname, nao, ...)	Run a quantum chemistry solver to compute the reduced density matrices.
run_solver_u(fobj_a, fobj_b, solver, enuc, ...)	Run a quantum chemistry solver to compute the reduced density matrices.

quemb.molbe.be_parallel.be_func_parallel

quemb.molbe.be_parallel.**be_func_parallel**(*pot*, *Fobjs*, *Nocc*, *solver*, *enuc*, *scratch_dir*, *solver_args*, *nproc=1*, *ompnum=4*, *only_chem=False*, *relax_density=False*, *use_cumulant=True*, *eeval=False*, *return_vec=False*)

Embarrassingly Parallel High-Level Computation

Performs high-level bootstrap embedding (BE) computation for each fragment. Computes 1-RDMs and 2-RDMs for each fragment. It also computes error vectors in BE density match. For selected CI solvers, this function exposes thresholds used in selected CI calculations (hci_cutoff, ci_coeff_cutoff, select_cutoff).

Parameters

- **pot** (list[float] | None) – Potentials (local & global) that are added to the 1-electron Hamiltonian component. The last element in the list is the chemical potential.
- **Fobjs** (list[*Frags*] | list[*Frags*]) – Fragment definitions.
- **Nocc** (int) – Number of occupied orbitals for the full system.
- **solver** (str) – High-level solver in bootstrap embedding. Supported values are ‘MP2’, ‘CCSD’, ‘FCI’, ‘HCI’, ‘SHCI’, and ‘SCI’.
- **enuc** (float) – Nuclear component of the energy.
- **nproc** (int) – Total number of processors assigned for the optimization. Defaults to 1. When nproc > 1, Python multithreading is invoked.
- **ompnum** (int) – If nproc > 1, sets the number of cores for OpenMP parallelization. Defaults to 4.
- **only_chem** (bool) – Whether to perform chemical potential optimization only. Refer to bootstrap embedding literature. Defaults to False.
- **eeval** (bool) – Whether to evaluate energies. Defaults to False.
- **scratch_dir** (*WorkDir*) – Scratch directory root
- **use_cumulant** (bool) – Use cumulant energy expression. Defaults to True
- **return_vec** (bool) – Whether to return the error vector. Defaults to False.

Returns

Depending on the parameters, returns the error norm or a tuple containing the error norm, error vector, and the computed energy.

Return type

float or *tuple*

quemb.molbe.be_parallel.be_func_parallel_u

`quemb.molbe.be_parallel.`**`be_func_parallel_u`**(*pot*, *Fobjs*, *solver*, *enuc*, *hf_veff=None*, *nproc=1*, *ompnum=4*, *relax_density=False*, *use_cumulant=True*, *frozen=False*)

Embarrassingly Parallel High-Level Computation

Performs high-level unrestricted bootstrap embedding (UBE) computation for each fragment. Computes 1-RDMs and 2-RDMs for each fragment to return the energy. As such, this currently is equipped for one-shot U-CCSD BE.

Parameters

- **pot** (*list of float*) – Potentials (local & global) that are added to the 1-electron Hamiltonian component. The last element in the list is the chemical potential. Should always be 0, as this is still a one-shot only implementation
- **Fobjs** (*list of tuple of* FragPart) – Fragment definitions, alpha and beta components.
- **solver** (*str*) – High-level solver in bootstrap embedding. Supported value is 'UCCSD'.
- **enuc** (*float*) – Nuclear component of the energy.
- **hf_veff** (*tuple of ndarray, optional*) – Alpha and beta Hartree-Fock effective potential.
- **nproc** (*int, optional*) – Total number of processors assigned for the optimization. Defaults to 1. When nproc > 1, Python multithreading is invoked.
- **ompnum** (*int, optional*) – If nproc > 1, sets the number of cores for OpenMP parallelization. Defaults to 4.
- **use_cumulant** – Whether to use the cumulant energy expression, by default True.
- **frozen** (*bool, optional*) – Frozen core. Defaults to False

Returns

Returns the computed energy

Return type

float

quemb.molbe.be_parallel.run_solver

`quemb.molbe.be_parallel.`**`run_solver`**(*h1*, *dm0*, *scratch_dir*, *dname*, *nao*, *nocc*, *n_frag*, *weight_and_relAO_per_center*, *TA*, *h1_e*, *solver='CCSD'*, *eri_file='eri_file.h5'*, *veff=None*, *veff0=None*, *eeval=True*, *ret_vec=False*, *use_cumulant=True*, *relax_density=False*, *solver_args=None*)

Run a quantum chemistry solver to compute the reduced density matrices.

Parameters

- **h1** (`ndarray[tuple[int, ...], dtype[float64]]`) – One-electron Hamiltonian matrix.
- **dm0** (`ndarray[tuple[int, ...], dtype[float64]]`) – Initial guess for the density matrix.
- **scratch_dir** (*WorkDir*) – The scratch dir root.
- **dname** (`str`) – Directory name for storing intermediate files. Fragment files will be stored in `scratch_dir / dname`.

- `scratch_dir` – The scratch directory. Fragment files will be stored in `scratch_dir / dname`.
- **nao** (`int`) – Number of atomic orbitals.
- **nocc** (`int`) – Number of occupied orbitals.
- **n_frag** (`int`) – Number of fragment sites.
- **weight_and_relAO_per_center** (`list[tuple[float, list[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]`) – Scaling factor for the electronic energy **and** the relative AO indices per center per frag
- **TA** (`ndarray[tuple[int, ...], dtype[float64]]`) – Transformation matrix for embedding orbitals.
- **h1_e** (`ndarray[tuple[int, ...], dtype[float64]]`) – One-electron integral matrix.
- **solver** (`Literal['MP2', 'CCSD', 'FCI', 'HCI', 'SHCI', 'SCI', 'DMRG']`) – Solver to use for the calculation. Default is 'CCSD'.
- **eri_file** (`str`) – Filename for the electron repulsion integrals. Default is 'eri_file.h5'.
- **veff** (`ndarray[tuple[int, ...], dtype[float64]] | None`) – Veff matrix to be passed to energy, if non-cumulant energy.
- **veff0** (`ndarray[tuple[int, ...], dtype[float64]] | None`) – Veff0 matrix, passed to energy, the hf_veff in the fragment Schmidt space
- **use_cumulant** (`bool`) – If True, use the cumulant approximation for RDM2. Default is True.
- **eeval** (`bool`) – If True, evaluate the electronic energy. Default is True.
- **ret_vec** (`bool`) – If True, return vector with error and rdms. Default is True.
- **relax_density** (`bool`) – If True, use CCSD relaxed density. Default is False

Returns

Depending on the input parameters, returns the molecular orbital coefficients, one-particle and two-particle reduced density matrices, and optionally the fragment energy.

Return type

tuple

quemb.molbe.be_parallel.run_solver_u

`quemb.molbe.be_parallel.`**`run_solver_u`**(*fobj_a*, *fobj_b*, *solver*, *enuc*, *hf_veff*, *relax_density=False*, *frozen=False*, *use_cumulant=True*)

Run a quantum chemistry solver to compute the reduced density matrices.

Parameters

- **fobj_a** (*`Frags`*) – Alpha spin molbe.pfrag.Frags object
- **fobj_b** (*`Frags`*) – Beta spin molbe.pfrag.Frags object
- **solver** (*`str`*) – High-level solver in bootstrap embedding. Supported value is “UCCSD”
- **enuc** (*`float`*) – Nuclear component of the energy
- **hf_veff** (*`tuple of ndarray, optional`*) – Alpha and beta spin Hartree-Fock effective potentials.

- **relax_density** (*bool, optional*) – If True, uses relaxed density matrix for UCCSD, defaults to False.
- **frozen** (*bool, optional*) – If True, uses frozen core, defaults to False
- **use_cumulant** (*bool, optional*) – If True, uses the cumulant approximation for RDM2. Default is True.

Returns
As implemented, only returns the UCCSD fragment energy

Return type
float

quemb.molbe.chemfrag

This module implements the fragmentation of molecular and periodic systems based on chemical connectivity that uses the overlap of tabulated van der Waals radii.

There are three main classes:

- *BondConnectivity* **contains the connectivity data of a chemical system**
  and is fully independent of the BE fragmentation level or used basis sets. After construction the knowledge about motifs in the system are available, if hydrogen atoms are treated differently then the motifs are all non-hydrogen atoms, while if hydrogen atoms are treated equal then all atoms are motifs.
- *PurelyStructureFragmented* **is depending on the** *BondConnectivity*
  and performs the fragmentation depending on the BE fragmentation level, but is still independent of the used basis set. After construction this class knows about the assignment of origins, centers, and edges.
- *Fragmented* **is depending on the** *PurelyStructureFragmented*
  and assigns the AO indices to each fragment and is responsible for the book keeping of which AO index belongs to which center and edge.

Functions

chemgen(mol, n_BE, args, frozen_core, ...)	Fragment a molecule based on chemical connectivity.
restrict_keys(D, keys)	Restrict the keys of a dictionary to a subset.

quemb.molbe.chemfrag.chemgen

`quemb.molbe.chemfrag.`**`chemgen`**(*mol*, *n_BE*, *args*, *frozen_core*, *iao_valence_basis*)

Fragment a molecule based on chemical connectivity.

Parameters

- **mol** (TypeVar(_T_chemsystem, Mole, Cell)) – Molecule or Cell to be fragmented.
- **n_BE** (int) – BE fragmentation level.
- **args** (*ChemGenArgs* | None) – Additional arguments for ChemGen fragmentation. These are passed on to *quemb.molbe.chemfrag.PurelyStructureFragmented.from_mole()* and documented there.
- **frozen_core** (bool) – Do we perform a frozen core calculation?
- **iao_valuence_basis** – The minimal basis used for the IAO definition.

- **swallow_replace** – If a fragment would be swallowed, it is instead replaced by the largest fragment that contains the smaller fragment. The definition of the origin is taken from the smaller fragment. This means, there will be no centers other than origins.

Return type
Fragmented

quemb.molbe.chemfrag.restrict_keys

`quemb.molbe.chemfrag.`**`restrict_keys`**(*D*, *keys*)

Restrict the keys of a dictionary to a subset.

The function has the interface declared in such a way that if the subset of keys is actually a subtype of the type of the keys, the type of the keys of the returned dictionary is narrowed down.

Return type
`Mapping`[`TypeVar`(`_T_Key`, bound= `Hashable`), `TypeVar`(`_T_Val`)]

Classes

BondConnectivity(bonds_atoms, motifs, ...[, ...])	Data structure to store the connectivity data of a molecule.
ChemGenArgs(*[, treat_H_different, ...])	Additional arguments for ChemGen fragmentation.
Fragmented(*, mol, conn_data, ...)	Contains the whole BE fragmentation information, including AO indices.
PurelyStructureFragmented(*, mol, ...)	Data structure to store the fragments of a molecule.

quemb.molbe.chemfrag.BondConnectivity

class `quemb.molbe.chemfrag.`**`BondConnectivity`**(*bonds_atoms*, *motifs*, *bonds_motifs*, *H_atoms*, *H_per_motif*, *atoms_per_motif*, *treat_H_different=True*)

Data structure to store the connectivity data of a molecule.

This collects all information that is independent of the chosen fragmentation scheme, i.e. BE1, BE2, etc., and is independent of the basis set, i.e. STO-3G, 6-31G, etc.

Attributes

`bonds_atoms: Final[Mapping[NewType(AtomIdx, int), OrderedSet[NewType(AtomIdx, int)]]]`

The connectivity graph of the molecule.

`motifs: Final[OrderedSet[NewType(MotifIdx, NewType(AtomIdx, int))]]`

The heavy atoms/motifs in the molecule. If hydrogens are not treated differently then every hydrogen is also a motif on its own.

`bonds_motifs: Final[Mapping[NewType(MotifIdx, NewType(AtomIdx, int)), OrderedSet[NewType(MotifIdx, NewType(AtomIdx, int))]]]`

`H_atoms: Final[OrderedSet[NewType(AtomIdx, int)]]`

The hydrogen atoms in the molecule. If hydrogens are not treated differently, then this is an empty set.

`H_per_motif: Final[Mapping[NewType(MotifIdx, NewType(AtomIdx, int)), OrderedSet[NewType(AtomIdx, int)]]]`

The hydrogen atoms per motif. If hydrogens are not treated differently, then the values of the dictionary are empty sets.

atoms_per_motif: Final[Mapping[NewType(MotifIdx, NewType(AtomIdx, int)), OrderedSet[NewType(AtomIdx, int)]]]

All atoms per motif. Lists the motif/heavy atom first.

treat_H_different: Final[bool]

Do we treat hydrogens differently?

Methods

__init__(bonds_atoms, motifs, bonds_motifs, ...)	Method generated by attrs for class BondConnectivity.
from_cartesian(m, *[, bonds_atoms, ...])	Create a *BondConnectivity* from a `chemcoord.Cartesian`.
from_cell(cell, *[, bonds_atoms, ...])	Create a *BondConnectivity* from a `pyscf.pbc.gto.cell.Cell`.
from_mole(mol, *[, bonds_atoms, vdW_radius, ...])	Create a *BondConnectivity* from a `pyscf.gto.mole.Mole`.
get_BE_fragment(i_center, n_BE)	Return the BE fragment around atom `i_center`.
get_all_BE_fragments(n_BE)	Return all BE-fragments

quemb.molbe.chemfrag.BondConnectivity.__init__

BondConnectivity.**__init__**(*bonds_atoms*, *motifs*, *bonds_motifs*, *H_atoms*, *H_per_motif*, *atoms_per_motif*, *treat_H_different=True*)

Method generated by attrs for class BondConnectivity.

quemb.molbe.chemfrag.BondConnectivity.from_cartesian

classmethod BondConnectivity.**from_cartesian**(*m*, ***, *bonds_atoms=None*, *vdW_radius=None*, *modify_atom_data=None*, *treat_H_different=True*)

Create a *BondConnectivity* from a `chemcoord.Cartesian`.

Parameters

- **m** (`Cartesian`) – The Cartesian object to extract the connectivity data from.
- **bonds_atoms** (`Mapping[int, set[int]] | None`) – Can be used to specify the connectivity graph of the molecule. Has exactly the same format as the output of `chemcoord.Cartesian.get_bonds()`, which is called internally if this argument is not specified. Allows it to manually change the connectivity by modifying the output of `chemcoord.Cartesian.get_bonds()`. The keyword is mutually exclusive with `vdW_radius`.
- **vdW_radius** (`int | float | floating | Callable[[int | float | floating], int | float | floating] | Mapping[str, int | float | floating] | None`) – If `bonds_atoms` is `None`, then the connectivity graph is determined by the van der Waals radius of the atoms. It is possible to pass:
  - a single number which is used as radius for all atoms,
  - a callable which is applied to all radii and can be used to e.g. scale via `lambda r: r * 1.1`,

  - a dictionary which maps the element symbol to the van der Waals radius, to change the radius of individual elements, e.g. `{"C": 1.5}`.

  The keyword is mutually exclusive with `bonds_atoms`.

- **modify_atom_data** (`Mapping[int, float] | None`) – To change the van der Waals radius of one or more specific atoms, pass a dictionary that looks like `{index1: 1.5}`.
- **treat_H_different** (`bool`) – If True, we treat hydrogen atoms differently from heavy atoms.

Return type
`Self`

quemb.molbe.chemfrag.BondConnectivity.from_cell

classmethod `BondConnectivity.from_cell`(*cell*, ***, *bonds_atoms=None*, *vdW_radius=None*, *modify_atom_data=None*, *treat_H_different=True*)

Create a *BondConnectivity* from a `pyscf.pbc.gto.cell.Cell`. This function considers the periodic boundary conditions by adding periodic copies of the cell to the molecule. The connectivity graph from the open boundary condition supercell is then used to determine the connectivity graph of the original periodic cell.

Parameters

- **cell** (`Cell`) – The `pyscf.pbc.gto.cell.Cell` to extract the connectivity data from.
- **bonds_atoms** (*Mapping[int, OrderedSet[int]]*) – Can be used to specify the connectivity graph of the molecule. Has exactly the same format as the output of `chemcoord.Cartesian.get_bonds()`, which is called internally if this argument is not specified. Allows it to manually change the connectivity by modifying the output of `chemcoord.Cartesian.get_bonds()`. The keyword is mutually exclusive with `vdW_radius`.
- **vdW_radius** (`int | float | floating | Callable[[int | float | floating], int | float | floating] | Mapping[str, int | float | floating] | None`) – If `bonds_atoms` is `None`, then the connectivity graph is determined by the van der Waals radius of the atoms. It is possible to pass:
  - a single number which is used as radius for all atoms,
  - a callable which is applied to all radii and can be used to e.g. scale via `lambda r: r * 1.1`,
  - a dictionary which maps the element symbol to the van der Waals radius, to change the radius of individual elements, e.g. `{"C": 1.5}`.

  The keyword is mutually exclusive with `bonds_atoms`.

- **modify_atom_data** (`Mapping[int, float] | None`) – To change the van der Waals radius of one or more specific atoms, pass a dictionary that looks like `{index1: 1.5}`.
- **treat_H_different** (`bool`) – If True, we treat hydrogen atoms differently from heavy atoms.

Return type
`Self`

quemb.molbe.chemfrag.BondConnectivity.from_mole

`classmethod BondConnectivity.from_mole(mol, *, bonds_atoms=None, vdW_radius=None, modify_atom_data=None, treat_H_different=True)`

Create a `BondConnectivity` from a `pyscf.gto.mole.Mole`.

Parameters

- **mol** (`Mole`) – The `pyscf.gto.mole.Mole` to extract the connectivity data from.
- **bonds_atoms** (*Mapping[int, OrderedSet[int]]*) – Can be used to specify the connectivity graph of the molecule. Has exactly the same format as the output of `chemcoord.Cartesian.get_bonds()`, which is called internally if this argument is not specified. Allows it to manually change the connectivity by modifying the output of `chemcoord.Cartesian.get_bonds()`. The keyword is mutually exclusive with `vdW_radius`.
- **vdW_radius** (`int` | `float` | `floating` | `Callable[[int | float | floating], int | float | floating]` | `Mapping[str, int | float | floating]` | `None`) – If `bonds_atoms` is `None`, then the connectivity graph is determined by the van der Waals radius of the atoms. It is possible to pass:
  - a single number which is used as radius for all atoms,
  - a callable which is applied to all radii and can be used to e.g. scale via `lambda r: r * 1.1`,
  - a dictionary which maps the element symbol to the van der Waals radius, to change the radius of individual elements, e.g. `{"C": 1.5}`.

  The keyword is mutually exclusive with `bonds_atoms`.
- **modify_atom_data** (`Mapping[int, float]` | `None`) – To change the van der Waals radius of one or more specific atoms, pass a dictionary that looks like `{index1: 1.5}`.
- **treat_H_different** (`bool`) – If True, we treat hydrogen atoms differently from heavy atoms.

Return type

`Self`

quemb.molbe.chemfrag.BondConnectivity.get_BE_fragment

`BondConnectivity.get_BE_fragment(i_center, n_BE)`

Return the BE fragment around atom `i_center`.

The BE fragment is the set of motifs (heavy atoms if hydrogens are different) that are reachable from the center atom within `(n_BE - 1)` bonds. This means that `n_BE == 1` returns only the center atom itself.

Parameters

- **i_center** (`NewType(MotifIdx, NewType(AtomIdx, int))`) – The index of the center atom.
- **n_BE** (`int`) – Defines the `(n_BE - 1)`-th coordination sphere to consider.

Return type

`OrderedSet[NewType(MotifIdx, NewType(AtomIdx, int))]`

quemb.molbe.chemfrag.BondConnectivity.get_all_BE_fragments

BondConnectivity.**get_all_BE_fragments**(*n_BE*)

Return all BE-fragments

Parameters
n_BE (int) – The coordination sphere to consider.

Returns
A dictionary mapping the center atom to the BE-fragment around it.

Return type
dict

quemb.molbe.chemfrag.ChemGenArgs

class quemb.molbe.chemfrag.**ChemGenArgs**(*, *treat_H_different=True*, *bonds_atoms=None*, *vdW_radius=None*, *modify_atom_data=None*, *swallow_replace=False*, *wrong_iao_indexing=False*)

Additional arguments for ChemGen fragmentation.

These are passed on to *quemb.molbe.chemfrag.PurelyStructureFragmented.from_mole()* and documented there.

Attributes

treat_H_different: Final[bool]

bonds_atoms: Mapping[int, set[int]] | None

vdW_radius: int | float | floating | Callable[[int | float | floating], int | float | floating] | Mapping[str, int | float | floating] | None

modify_atom_data: Mapping[int, float] | None

swallow_replace: bool

If a fragment would be swallowed, it is instead replaced by the largest fragment that contains the smaller fragment. The definition of the origin is taken from the smaller fragment. This means, there will be no centers other than origins.

Methods

__init__(*[, treat_H_different, ...])	Method generated by attrs for class ChemGenArgs.

quemb.molbe.chemfrag.ChemGenArgs.__init__

ChemGenArgs.**__init__**(*, *treat_H_different=True*, *bonds_atoms=None*, *vdW_radius=None*, *modify_atom_data=None*, *swallow_replace=False*, *wrong_iao_indexing=False*)

Method generated by attrs for class ChemGenArgs.

quemb.molbe.chemfrag.Fragmented

class quemb.molbe.chemfrag.**Fragmented**(*, *mol*, *conn_data*, *frag_structure*, *AO_per_atom*, *AO_per_frag*, *AO_per_motif*, *AO_per_edge_per_frag*, *relAO_per_motif_per_frag*, *relAO_per_edge_per_frag*, *relAO_per_center_per_frag*, *relAO_per_origin_per_frag*, *relAO_in_ref_per_edge_per_frag*, *frozen_core*, *iao_valence_mol*)

Contains the whole BE fragmentation information, including AO indices.

This takes into account the geometrical data and the used basis sets, hence it "knows" which AO index belongs to which atom and which fragment. It depends on *PurelyStructureFragmented* to store structural data, but contains more information.

Attributes

mol: TypeVar(_T_chemsystem, Mole, Cell)

The full molecule

conn_data: Final[*BondConnectivity*]

frag_structure: Final[*PurelyStructureFragmented*]

AO_per_atom: Final[Sequence[OrderedSet[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]

The atomic orbital indices per atom

AO_per_frag: Final[Sequence[OrderedSet[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]

The atomic orbital indices per fragment

AO_per_motif: Final[Mapping[NewType(MotifIdx, NewType(AtomIdx, int)), Mapping[NewType(AtomIdx, int), OrderedSet[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]]

The atomic orbital indices per motif

AO_per_edge_per_frag: Final[Sequence[Mapping[NewType(EdgeIdx, NewType(MotifIdx, NewType(AtomIdx, int))), Mapping[NewType(AtomIdx, int), OrderedSet[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]]]

The atomic orbital indices per edge per fragment. The AO index is global.

relAO_per_motif_per_frag: Final[Sequence[Mapping[NewType(MotifIdx, NewType(AtomIdx, int)), Mapping[NewType(AtomIdx, int), OrderedSet[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]]]

The relative atomic orbital indices per motif per fragment. Relative means that the AO indices are relative to the **own** fragment.

relAO_per_edge_per_frag: Final[Sequence[Mapping[NewType(EdgeIdx, NewType(MotifIdx, NewType(AtomIdx, int))), Mapping[NewType(AtomIdx, int), OrderedSet[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]]]

The relative atomic orbital indices per edge per fragment. Relative means that the AO indices are relative to the **own** fragment. This variable is a strict subset of *relAO_per_motif_per_frag*, in the sense that the motif indices, the keys in the Mapping, are restricted to the edges of the fragment.

relAO_per_center_per_frag: Final[Sequence[Mapping[NewType(CenterIdx, NewType(MotifIdx, NewType(AtomIdx, int))), Mapping[NewType(AtomIdx, int), OrderedSet[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]]]

The relative atomic orbital indices per edge per fragment. Relative means that the AO indices are relative to the **own** fragment. This variable is a subset of *relAO_per_motif_per_frag*, in the sense that the motif indices, the keys in the Mapping, are restricted to the centers of the fragment.

relAO_per_origin_per_frag: Final[Sequence[Mapping[NewType(OriginIdx, NewType(CenterIdx, NewType(MotifIdx, NewType(AtomIdx, int)))), Mapping[NewType(AtomIdx, int), OrderedSet[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]]]

The relative atomic orbital indices per origin per fragment. Relative means that the AO indices are relative to the **own** fragment. This variable is a subset of *relAO_per_center_per_frag*, in the sense that the motif indices, the keys in the Mapping, are restricted to the origins of the fragment. This variable was formerly known as `centerf_idx`.

relAO_in_ref_per_edge_per_frag: Final[Sequence[Mapping[NewType(EdgeIdx, NewType(MotifIdx, NewType(AtomIdx, int))), Mapping[NewType(AtomIdx, int), OrderedSet[NewType(RelAOIdxInRef, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]]]

The relative atomic orbital indices per edge per fragment. Relative means that the AO indices are relative to the **other** fragment where the edge is a center.

frozen_core: Final[bool]

Do we have frozen_core AO index offsets?

iao_valence_mol: TypeVar(_T_chemsystem, Mole, Cell) | None

The molecule with the valence/minimal basis, if we use IAO.

Methods

__init__(*, mol, conn_data, frag_structure, ...)	Method generated by attrs for class Fragmented.
from_frag_structure(mol, frag_structure, ...)	Construct a *Fragmented*
from_mole(mol, n_BE, *[, frozen_core, ...])	Construct a *Fragmented* from `pyscf.gto.mole.Mole`
get_FragPart([wrong_iao_indexing])	Match the output of *quemb.molbe.autofrag.autogen()*.

quemb.molbe.chemfrag.Fragmented.__init__

`Fragmented.`**__init__**(*, *mol*, *conn_data*, *frag_structure*, *AO_per_atom*, *AO_per_frag*, *AO_per_motif*, *AO_per_edge_per_frag*, *relAO_per_motif_per_frag*, *relAO_per_edge_per_frag*, *relAO_per_center_per_frag*, *relAO_per_origin_per_frag*, *relAO_in_ref_per_edge_per_frag*, *frozen_core*, *iao_valence_mol*)

Method generated by attrs for class Fragmented.

quemb.molbe.chemfrag.Fragmented.from_frag_structure

classmethod `Fragmented.`**from_frag_structure**(*mol*, *frag_structure*, *frozen_core*, *iao_valence_basis=None*)

Construct a *Fragmented*

Parameters

- **mol** (`TypeVar`(_T_chemsystem, `Mole`, `Cell`)) – The Molecule to extract the connectivity data from.
- **frag_structure** (*PurelyStructureFragmented*) – The fragmented structure to use.

Return type

`Self`

quemb.molbe.chemfrag.Fragmented.from_mole

classmethod Fragmented.**from_mole**(*mol*, *n_BE*, ***, *frozen_core=False*, *treat_H_different=True*, *bonds_atoms=None*, *vdW_radius=None*, *modify_atom_data=None*, *iao_valence_basis=None*, *autocratic_matching=True*, *swallow_replace=False*)

Construct a *Fragmented* **from** `pyscf.gto.mole.Mole` or `pyscf.pbc.gto.cell.Cell`.

Parameters

- **mol** (`TypeVar`(`_T_chemsystem`, `Mole`, `Cell`)) – The `pyscf.gto.mole.Mole` or `pyscf.pbc.gto.cell.Cell` to extract the connectivity data from.
- **n_BE** (`int`) – The BE fragmentation level.
- **treat_H_different** (`bool`) – If True, we treat hydrogen atoms differently from heavy atoms.
- **bonds_atoms** (`Mapping`[`int`, `set`[`int`]] | `None`) – Can be used to specify the connectivity graph of the molecule. Has exactly the same format as the output of `chemcoord.Cartesian.get_bonds()`, which is called internally if this argument is not specified. Allows it to manually change the connectivity by modifying the output of `chemcoord.Cartesian.get_bonds()`. The keyword is mutually exclusive with `vdW_radius`.
- **vdW_radius** (`int` | `float` | `floating` | `Callable`[[`int` | `float` | `floating`], `int` | `float` | `floating`] | `Mapping`[`str`, `int` | `float` | `floating`] | `None`) – If `bonds_atoms` is `None`, then the connectivity graph is determined by the van der Waals radius of the atoms. It is possible to pass:
  - a single number which is used as radius for all atoms,
  - a callable which is applied to all radii and can be used to e.g. scale via `lambda r: r * 1.1`,
  - a dictionary which maps the element symbol to the van der Waals radius, to change the radius of individual elements, e.g. `{"C": 1.5}`.

  The keyword is mutually exclusive with `bonds_atoms`.
- **modify_atom_data** (`Mapping`[`int`, `float`] | `None`) – To change the van der Waals radius of one or more specific atoms, pass a dictionary that looks like `{index1: 1.5}`.
- **autocratic_matching** (`bool`) – Assume autocratic matching for possibly shared centers. Will call *`PurelyStructureFragmented.get_autocratically_matched()`* upon construction. Look there for more details.
- **swallow_replace** (`bool`) – If a fragment would be swallowed, it is instead replaced by the largest fragment that contains the smaller fragment. The definition of the origin is taken from the smaller fragment. This means, there will be no centers other than origins.

Return type

`Self`

quemb.molbe.chemfrag.Fragmented.get_FragPart

Fragmented.**get_FragPart**(*wrong_iao_indexing=None*)

Match the output of *`quemb.molbe.autofrag.autogen()`*.

Return type
FragPart

quemb.molbe.chemfrag.PurelyStructureFragmented

`class quemb.molbe.chemfrag.`**`PurelyStructureFragmented`**`(`**, mol, motifs_per_frag, centers_per_frag, edges_per_frag, origin_per_frag, atoms_per_frag, ref_frag_idx_per_edge, conn_data, n_BE*`)`

Data structure to store the fragments of a molecule.

This takes into account only the connectivity data and the fragmentation scheme but is independent of the basis sets or the electronic structure.

Attributes

`mol: TypeVar(_T_chemsystem, Mole, Cell)`

The full molecule

`motifs_per_frag: Final[Sequence[OrderedSet[NewType(MotifIdx, NewType(AtomIdx, int))]]]`

The motifs per fragment. Note that the full set of motifs for a fragment is the union of all center motifs and edge motifs. The order is guaranteed to be first origin, centers, then edges and in each category the motif index is ascending.

`centers_per_frag: Final[Sequence[OrderedSet[NewType(CenterIdx, NewType(MotifIdx, NewType(AtomIdx, int)))]]]`

The centers per fragment. Note that the set of centers is the complement of the edges. The order is guaranteed to be ascending. Every motif is at least once a center, but a given motif can appear multiple times as center in different fragments. By using *`get_autocratically_matched()`* we can ensure that a given center appears exactly once as a center.

`edges_per_frag: Final[Sequence[OrderedSet[NewType(EdgeIdx, NewType(MotifIdx, NewType(AtomIdx, int)))]]]`

The edges per fragment. Note that the set of edges is the complement of the centers. The order is guaranteed to be ascending.

`origin_per_frag: Final[Sequence[OrderedSet[NewType(OriginIdx, NewType(CenterIdx, NewType(MotifIdx, NewType(AtomIdx, int))))]]]`

The origins per frag. Note that for “normal” BE calculations there is exacctly one origin per fragment, i.e. the *Sequence* has one element. The order is guaranteed to be ascending.

`atoms_per_frag: Final[Sequence[OrderedSet[NewType(AtomIdx, int)]]]`

The atom indices per fragment. it contains both motif **and** hydrogen indices. The order of the motifs is the same as in *`motifs_per_frag`*. The hydrogen atoms directly follow the motif to which they are attached, and are then ascendingly sorted. To given an example: if 1 and 4 are motif indices and hydrogen atoms 5, 6 are connected to 1, while hydrogen atoms 2, 3 are connected to 4, then the order is: `[1, 5, 6, 4, 2, 3]`.

`ref_frag_idx_per_edge: Final[Sequence[Mapping[NewType(EdgeIdx, NewType(MotifIdx, NewType(AtomIdx, int))), NewType(FragmentIdx, integer)]]]`

For each edge in a fragment it points to the index of the fragment where this fragment is a center, i.e. where this edge is correctly described and should be matched against. Variable was formerly known as *center*.

conn_data: Final[*BondConnectivity*]

Connectivity data of the molecule.

n_BE: Final[int]

Methods

__init__(*, mol, motifs_per_frag, ...)	Method generated by attrs for class PurelyStructure-Fragmented.
from_conn_data(mol, conn_data, n_BE, ...)	
from_mole(mol, n_BE, *[, treat_H_different, ...])	Construct a *PurelyStructureFragmented* from a `pyscf.gto.mole.Mole`.
get_autocratically_matched()	Ensure that no centers exist, that are shared among fragments.
get_string()	Get a long string representation of the fragments.
is_ordered()	Return if `self` is ordered.
shared_centers_exist()	Check if shared centers exist.
write_geom([prefix, dir])	Write the structures of the fragments to files.

quemb.molbe.chemfrag.PurelyStructureFragmented.__init__

PurelyStructureFragmented.**__init__**(*, *mol*, *motifs_per_frag*, *centers_per_frag*, *edges_per_frag*, *origin_per_frag*, *atoms_per_frag*, *ref_frag_idx_per_edge*, *conn_data*, *n_BE*)

Method generated by attrs for class PurelyStructureFragmented.

quemb.molbe.chemfrag.PurelyStructureFragmented.from_conn_data

classmethod PurelyStructureFragmented.**from_conn_data**(*mol*, *conn_data*, *n_BE*, *swallow_replace*)

Return type
`Self`

quemb.molbe.chemfrag.PurelyStructureFragmented.from_mole

classmethod PurelyStructureFragmented.**from_mole**(*mol*, *n_BE*, *, *treat_H_different=True*, *bonds_atoms=None*, *vdW_radius=None*, *modify_atom_data=None*, *autocratic_matching=True*, *swallow_replace=False*)

Construct a *PurelyStructureFragmented* from a `pyscf.gto.mole.Mole`.

Parameters

- **mol** (`TypeVar`(`_T_chemsystem`, `Mole`, `Cell`)) – The Molecule to extract the connectivity data from.
- **n_BE** (`int`) – The coordination sphere to consider.
- **treat_H_different** (`bool`) – If True, we treat hydrogen atoms differently from heavy atoms.

- **bonds_atoms** (Mapping[int, set[int]] | None) – Can be used to specify the connectivity graph of the molecule. Has exactly the same format as the output of `chemcoord.Cartesian.get_bonds()`, which is called internally if this argument is not specified. Allows it to manually change the connectivity by modifying the output of `chemcoord.Cartesian.get_bonds()`. The keyword is mutually exclusive with `vdW_radius`.
- **vdW_radius** (int | float | floating | Callable[[int | float | floating], int | float | floating] | Mapping[str, int | float | floating] | None) – If `bonds_atoms` is `None`, then the connectivity graph is determined by the van der Waals radius of the atoms. It is possible to pass:
  - a single number which is used as radius for all atoms,
  - a callable which is applied to all radii and can be used to e.g. scale via `lambda r: r * 1.1`,
  - a dictionary which maps the element symbol to the van der Waals radius, to change the radius of individual elements, e.g. `{"C": 1.5}`.

  The keyword is mutually exclusive with `bonds_atoms`.
- **modify_atom_data** (Mapping[int, float] | None) – To change the van der Waals radius of one or more specific atoms, pass a dictionary that looks like `{index1: 1.5}`.
- **autocratic_matching** (bool) – Assume autocratic matching for possibly shared centers. Will call *`get_autocratically_matched()`* upon construction. Look there for more details.
- **swallow_replace** (bool) – If a fragment would be swallowed, it is instead replaced by the largest fragment that contains the smaller fragment. The definition of the origin is taken from the smaller fragment. This means, there will be no centers other than origins.

Return type
`Self`

quemb.molbe.chemfrag.PurelyStructureFragmented.get_autocratically_matched

`PurelyStructureFragmented.`**`get_autocratically_matched`**`()`

Ensure that no centers exist, that are shared among fragments.

This is the same as using autocratic matching. If there is a motif, which appears as center in multiple fragments, we will choose a fragment whose origin is closest to the center. In this fragment it will stay a center, while the same motif will become an edge in the other fragments.

For example, if we have the following nested structure (part of a larger molecule that continues left and right)

```
--- 1 - 2 - 3 - 4 - 5 ---
            |
            6
            |
            7
```

and assume BE(3) fragmentation then the atom 6 appears as center in the BE(3)-fragments around atoms 2, 3, and 4. Since atom 6 is closest to atom 3, it will stay a center in the fragment around atom 3 and will be re-declared as edge in the fragments around 2 and 4.

Return type
`Self`

quemb.molbe.chemfrag.PurelyStructureFragmented.get_string

PurelyStructureFragmented.**get_string**()

Get a long string representation of the fragments.

One can also call `str(self)` to get a short string representation.

Return type
`str`

quemb.molbe.chemfrag.PurelyStructureFragmented.is_ordered

PurelyStructureFragmented.**is_ordered**()

Return if `self` is ordered.

Ordered in this context means, that first the origins, then centers, then edges appear in the motif.

Return type
`bool`

quemb.molbe.chemfrag.PurelyStructureFragmented.shared_centers_exist

PurelyStructureFragmented.**shared_centers_exist**()

Check if shared centers exist.

Using *get_autocratically_matched()* it is possible to re-declare shared centers as edges.

Return type
`bool`

quemb.molbe.chemfrag.PurelyStructureFragmented.write_geom

PurelyStructureFragmented.**write_geom**(*prefix='f'*, *dir=PosixPath('.')*)

Write the structures of the fragments to files.

Return type
`None`

quemb.molbe.eri_onthefly

Functions

integral_direct_DF(mf, Fobjs, file_eri[, ...])	Calculate AO density-fitted 3-center integrals on-the-fly and transform to Schmidt space for given fragment objects

quemb.molbe.eri_onthefly.integral_direct_DF

quemb.molbe.eri_onthefly.**integral_direct_DF**(*mf*, *Fobjs*, *file_eri*, *auxbasis=None*)

Calculate AO density-fitted 3-center integrals on-the-fly and transform to Schmidt space for given fragment objects

Parameters

- **mf** (*RHF*) – Mean-field object for the chemical system (typically BE.mf)

- **Fobjs** (*list of FragPart*) – List containing fragment objects (typically BE.Fobjs) The MO coefficients are taken from Frags.TA and the transformed ERIs are stored in Frags.dname as h5py datasets.
- **file_eri** (*File*) – HDF5 file object to store the transformed fragment ERIs
- **auxbasis** (*str, optional*) – Auxiliary basis used for density fitting. If not provided, use pyscf's default choice for the basis set used to construct mf object; by default None

quemb.molbe.eri_sparse_DF

Functions

account_for_symmetry(reachable)	Account for permutational symmetry and remove all q that are larger than p.
approx_S_abs(mol[, nroots])	Compute the approximated absolute overlap matrix.
contract_with_TA_1st(TA, int_mu_nu_P, ...)	
contract_with_TA_2nd_to_dense(TA, int_mu_i_P)	Contract the first dimension of `int_mu_i_P` with the first dimension of `TA`.
contract_with_TA_2nd_to_sym_dense(TA, int_mu_i_P)	Contract the first dimension of `int_mu_i_P` with the first dimension of `TA`.
contract_with_TA_2nd_to_sym_sparse(TA, ...)	Contract the first dimension of `int_mu_i_P` with the first dimension of `TA`.
conversions_AO_shell(mol)	Return dictionaries that for a shell index return the corresponding AO indices and for an AO index return the corresponding shell index.
find_screening_radius(mol[, auxmol, ...])	Return a dictionary with radii for each element in `mol` that can be used to screen the 2-electron integrals to be lower than threshold.
get_atom_per_AO(mol)	
get_atom_per_MO(atom_per_AO, TA[, epsilon])	
get_blocks(reachable)	Return the value of the border elements of contiguous blocks in the sequence X."
get_complement(reachable)	Return the orbitals that cannot be reached by an orbital after screening.
get_dense_integrals(ints_3c2e, df_coef)	Compute dense ERIs from sparse 3-center integrals and sparse DF coefficients.
get_orbs_per_atom(atom_per_orb)	
get_orbs_reachable_by_atom(orb_per_atom, ...)	
get_orbs_reachable_by_orb(atom_per_orb, ...)	Concatenate the `atom_per_orb` and `reachable_orb_per_atom` Such that it becomes a mapping `i_orb -> i_atom -> j_atom`
get_reachable(atoms_per_start_orb, ...)	Return the sorted orbitals that can by reached for each orbital after screening.
get_screened(mol, screening_radius)	
get_sparse_D_ints_and_coeffs(mol, auxmol[, ...])	Return the 3-center 2-electron integrals and fitting coefficients in a semi-sparse format for the AO basis

continues on next page

Table 15 – continued from previous page

get_sparse_P_mu_nu(mol, auxmol, exch_reachable)	Return the 3-center 2-electron integrals in a sparse format.
grid_S_abs(mol[, grid_level])	Calculates the overlap matrix $S_ij = \int \|phi_i(r)\|\|phi_j(r)\|dr$ using numerical integration on a DFT grid.
identify_contiguous_blocks(X)	Identify the indices of contiguous blocks in the sequence X.
to_numba_input(exch_reachable)	Convert the reachable orbitals to a list of numpy arrays.
transform_sparse_DF_integral_cpp(mf, Fobjs, ...)	
transform_sparse_DF_integral_cpp_gpu(mf, ...)	
transform_sparse_DF_integral_nb(mf, Fobjs, ...)	
traverse_nonzero(g[, unique])	Traverse the non-zero elements of a semi-sparse 3-index tensor.

quemb.molbe.eri_sparse_DF.account_for_symmetry

quemb.molbe.eri_sparse_DF.**account_for_symmetry**(*reachable*)

Account for permutational symmetry and remove all q that are larger than p.

Parameters

reachable (Mapping[TypeVar(_T_start, bound= integer), Collection[TypeVar(_T_target, bound= integer)]])

Return type

dict[TypeVar(_T_start, bound= integer), list[TypeVar(_T_target, bound= integer)]]

Example

```
>>> account_for_symmetry({0: [0, 1, 2], 1: [0, 1, 2], 2: [0, 1, 2]})
>>> {0: [0], 1: [0, 1], 2: [0, 1, 2]}
```

quemb.molbe.eri_sparse_DF.approx_S_abs

quemb.molbe.eri_sparse_DF.**approx_S_abs**(*mol*, *nroots=500*)

Compute the approximated absolute overlap matrix.

The calculation is only exact for uncontracted, cartesian basis functions. Since the absolute value is not a linear function, the value after contraction and/or transformation to spherical-harmonics is approximated via the RHS of the triangle inequality:

$$\int |\phi_i(\mathbf{r})| \, |\phi_j(\mathbf{r})| \, d\mathbf{r} \leq \sum_{\alpha,\beta} |c_{\alpha i}| \, |c_{\beta j}| \int |\chi_\alpha(\mathbf{r})| \, |\chi_\beta(\mathbf{r})| \, d\mathbf{r}$$

Parameters

- **mol** (Mole)
- **nroots** (int) – Number of roots for the Gauß-Hermite quadrature.

Return type

ndarray[tuple[int, ...], dtype[float64]]

quemb.molbe.eri_sparse_DF.contract_with_TA_1st

`quemb.molbe.eri_sparse_DF.`**`contract_with_TA_1st`**(*TA*, *int_mu_nu_P*, *AO_reachable_per_SchmidtMO*)

Return type
`SemiSparse3DTensor`

quemb.molbe.eri_sparse_DF.contract_with_TA_2nd_to_dense

`quemb.molbe.eri_sparse_DF.`**`contract_with_TA_2nd_to_dense`**(*TA*, *int_mu_i_P*)

Contract the first dimension of `int_mu_i_P` with the first dimension of `TA`. If the result is known to be symmetric use *`contract_with_TA_2nd_to_sym_dense()`* or *`contract_with_TA_2nd_to_sym_sparse()`* instead.

Can be used to e.g. compute contractions of mixed fragment-bath integrals.

$$(ia|P) = \sum_{\mu} T_{\mu,i}(\mu a|P)$$
$$(ai|P) = \sum_{\mu} T_{\mu,a}(\mu i|P)$$

Returns
A dense 3D tensor $(ia|P)$, which can have different lengths along the first two dimensions, e.g. fragment and bath orbitals. The last dimension is along the auxiliary basis.

Return type
Tensor3D

quemb.molbe.eri_sparse_DF.contract_with_TA_2nd_to_sym_dense

`quemb.molbe.eri_sparse_DF.`**`contract_with_TA_2nd_to_sym_dense`**(*TA*, *int_mu_i_P*)

Contract the first dimension of `int_mu_i_P` with the first dimension of `TA`. We assume the result to be symmetric in the first two dimensions.

If the result is known to be non-symmetric use *`contract_with_TA_2nd_to_dense()`* instead.

Can be used to e.g. compute contractions to purely fragment, or purely bath integrals.

$$(ij|P) = \sum_{\mu} T_{\mu,i}(\mu j|P)$$
$$(ab|P) = \sum_{\mu} T_{\mu,a}(\mu b|P)$$

Returns
A dense 3D tensor $(ij|P)$, symmetric in the first two (MO) dimensions. The last dimension is along the auxiliary basis.

Return type
Tensor3D

quemb.molbe.eri_sparse_DF.contract_with_TA_2nd_to_sym_sparse

`quemb.molbe.eri_sparse_DF.`**`contract_with_TA_2nd_to_sym_sparse`**(*TA*, *int_mu_i_P*, *i_reachable_by_j*)

Contract the first dimension of `int_mu_i_P` with the first dimension of `TA`. We assume the result to be symmetric in the first two dimensions.

If the result is known to be non-symmetric use *`contract_with_TA_2nd_to_dense()`* instead.

Can be used to e.g. compute contractions to purely fragment, or purely bath integrals.

$$(ij|P) = \sum_{\mu} T_{\mu,i}(\mu j|P)$$
$$(ab|P) = \sum_{\mu} T_{\mu,a}(\mu b|P)$$

Returns
A dense 3D tensor $(ij|P)$, symmetric in the first two (MO) dimensions. The last dimension is along the auxiliary basis.

Return type
Tensor3D

quemb.molbe.eri_sparse_DF.conversions_AO_shell

`quemb.molbe.eri_sparse_DF.`**`conversions_AO_shell`**`(`*mol*`)`

Return dictionaries that for a shell index return the corresponding AO indices and for an AO index return the corresponding shell index.

Parameters
`mol` (`Mole`) – The molecule.

Return type
`tuple[dict[NewType(ShellIdx, integer), list[NewType(AOIdx, NewType(OrbitalIdx, integer))]], dict[NewType(AOIdx, NewType(OrbitalIdx, integer)), NewType(ShellIdx, integer)]]`

quemb.molbe.eri_sparse_DF.find_screening_radius

`quemb.molbe.eri_sparse_DF.`**`find_screening_radius`**`(`*mol*, *auxmol=None*, ***, *threshold=1e-07*, *scale_factor=1.03*`)`

Return a dictionary with radii for each element in `mol` that can be used to screen the 2-electron integrals to be lower than threshold.

For a threshhold T and for all screened pairs of μ, ν the screening radius is defined in the following way: If `auxmol` is not given, the screening radius is calculated such that $(\mu\nu|\mu\nu) < T$. If `auxmol` is given, the screening radius is calculated such that $\sum_P |(\mu\nu|P)| < T$.

Parameters

- **`mol`** (`Mole`) – The molecule for which the screening radii are calculated.
- **`auxmol`** (`Mole` | `None`) – The molecule with the auxiliary basis.
- **`threshold`** (`float`) – The threshold for the integral values.
- **`scale_factor`** (`float`) – The scaling factor for the screening radius.

Return type
`dict[str, float]`

quemb.molbe.eri_sparse_DF.get_atom_per_AO

`quemb.molbe.eri_sparse_DF.`**`get_atom_per_AO`**`(`*mol*`)`

Return type
`dict[NewType(AOIdx, NewType(OrbitalIdx, integer)), set[NewType(AtomIdx, int)]]`

quemb.molbe.eri_sparse_DF.get_atom_per_MO

`quemb.molbe.eri_sparse_DF.`**`get_atom_per_MO`**(*atom_per_AO*, *TA*, *epsilon=1e-08*)

Return type
`dict`[`NewType`(`MOIdx`, `NewType`(`OrbitalIdx`, `integer`)), `set`[`NewType`(`AtomIdx`, `int`)]]

quemb.molbe.eri_sparse_DF.get_blocks

`quemb.molbe.eri_sparse_DF.`**`get_blocks`**(*reachable*)

Return the value of the border elements of contiguous blocks in the sequence X."

A block is defined as a sequence of consecutive integers. Returns a list of tuples, where each tuple contains the value at the start and at the end of a block.

Parameters
X

Return type
`list`[`tuple`[`TypeVar`(`_T`, `int`, `integer`), `TypeVar`(`_T`, `int`, `integer`)]]

Example

```
>>> X = [1, 2, 3, 5, 6, 7, 9, 10]
>>> get_blocks(X) == [(1, 3), (5, 7), (9, 10)]
```

quemb.molbe.eri_sparse_DF.get_complement

`quemb.molbe.eri_sparse_DF.`**`get_complement`**(*reachable*)

Return the orbitals that cannot be reached by an orbital after screening.

Return type
`dict`[`TypeVar`(`_T_start`, bound= `integer`), `list`[`TypeVar`(`_T_target`, bound= `integer`)]]

quemb.molbe.eri_sparse_DF.get_dense_integrals

`quemb.molbe.eri_sparse_DF.`**`get_dense_integrals`**(*ints_3c2e*, *df_coef*)

Compute dense ERIs from sparse 3-center integrals and sparse DF coefficients.

We evaluate the integrals via

$$(\mu, \nu|\rho\sigma) = \sum_P (\mu\nu|P) C^P_{\rho\sigma}$$

Parameters

- **`ints_3c2e`** (*`SemiSparseSym3DTensor`*) – Sparse 3-center integrals in the form of a *`SemiSparseSym3DTensor`*. $(\mu\nu|P)$ is given by `ints_3c2e[mu, nu][P]`.
- **`df_coef`** (*`SemiSparseSym3DTensor`*) – DF coefficients in the form of a *`SemiSparseSym3DTensor`*. $C^P_{\mu\nu}$ is given by `df_coef[mu, nu][P]`.

Return type
`ndarray`[`tuple`[`int`, `...`], `dtype`[`float64`]]

quemb.molbe.eri_sparse_DF.get_orbs_per_atom

`quemb.molbe.eri_sparse_DF.`**`get_orbs_per_atom`**`(`*atom_per_orb*`)`

Return type

`dict[NewType(AtomIdx, int), set[TypeVar(_T_orb_idx, bound= NewType(OrbitalIdx, integer))]]`

quemb.molbe.eri_sparse_DF.get_orbs_reachable_by_atom

`quemb.molbe.eri_sparse_DF.`**`get_orbs_reachable_by_atom`**`(`*orb_per_atom*, *screened*`)`

Return type

`dict[NewType(AtomIdx, int), dict[NewType(AtomIdx, int), Set[TypeVar(_T_orb_idx, bound= NewType(OrbitalIdx, integer))]]]`

quemb.molbe.eri_sparse_DF.get_orbs_reachable_by_orb

`quemb.molbe.eri_sparse_DF.`**`get_orbs_reachable_by_orb`**`(`*atom_per_orb*, *reachable_orb_per_atom*`)`

Concatenate the `atom_per_orb` and `reachable_orb_per_atom` Such that it becomes a mapping `i_orb -> i_atom -> j_atom`

Return type

`dict[TypeVar(_T_start_orb, bound= NewType(OrbitalIdx, integer)), dict[NewType(AtomIdx, int), Mapping[NewType(AtomIdx, int), Set[TypeVar(_T_target_orb, bound= NewType(OrbitalIdx, integer))]]]]`

quemb.molbe.eri_sparse_DF.get_reachable

`quemb.molbe.eri_sparse_DF.`**`get_reachable`**`(`*atoms_per_start_orb*, *atoms_per_target_orb*, *screened_connection*`)`

Return the sorted orbitals that can by reached for each orbital after screening.

Parameters

- **`mol`** – The molecule.
- **`atoms_per_orb`** – The atoms per orbital. For AOs this is the atom the AO is centered on, i.e. a set containing only one element, but for delocalised MOs there can be more than one atom.
- **`screening_radius`** – The screening cutoff is given by the overlap of van der Waals radii. By default, all radii are set to 5 Å, i.e. the screening distance is 10 Å. Alternatively, a callable or a dictionary can be passed. The callable is called with the tabulated van der Waals radius of the atom as argument and can be used to scale it up. The dictionary can be used to define different van der Waals radii for different elements. Compare to the `modify_element_data` argument of `get_bonds()`.

Return type

`dict[TypeVar(_T_start_orb, bound= NewType(OrbitalIdx, integer)), list[TypeVar(_T_target_orb, bound= NewType(OrbitalIdx, integer))]]`

quemb.molbe.eri_sparse_DF.get_screened

`quemb.molbe.eri_sparse_DF.`**`get_screened`**`(`*mol*, *screening_radius*`)`

Return type

`dict[NewType(AtomIdx, int), set[NewType(AtomIdx, int)]]`

quemb.molbe.eri_sparse_DF.get_sparse_D_ints_and_coeffs

`quemb.molbe.eri_sparse_DF.`**`get_sparse_D_ints_and_coeffs`**(*mol*, *auxmol*, *screening_radius=None*)

Return the 3-center 2-electron integrals and fitting coefficients in a semi-sparse format for the AO basis

One can obtain `g[p, q, r, s] == ints_3c2e[p, q] @ df_coef[r, s]`.

The datastructures use sparsity in the `p, q` and in the `r, s` pairs but the dimension along the auxiliary basis is densely stored.

Parameters

- **`mol`** (`Mole`) – The molecule.
- **`auxmol`** (`Mole`) – The molecule with auxiliary basis functions.
- **`screening_radius`** (`int` | `float` | `floating` | `Callable`[[`int` | `float` | `floating`], `int` | `float` | `floating`] | `Mapping`[`str`, `int` | `float` | `floating`] | `None`) – The screening cutoff is given by the overlap of van der Waals radii. By default, the radii are determined by *`find_screening_radius()`*. Alternatively, a fixed radius, callable or a dictionary can be passed. The callable is called with the tabulated van der Waals radius of the atom as argument and can be used to scale it up. The dictionary can be used to define different van der Waals radii for different elements. Compare to the `modify_element_data` argument of `get_bonds()`.

Return type

`tuple`[*`SemiSparseSym3DTensor`*, *`SemiSparseSym3DTensor`*]

quemb.molbe.eri_sparse_DF.get_sparse_P_mu_nu

`quemb.molbe.eri_sparse_DF.`**`get_sparse_P_mu_nu`**(*mol*, *auxmol*, *exch_reachable*)

Return the 3-center 2-electron integrals in a sparse format.

Return type

`SemiSparseSym3DTensor`

quemb.molbe.eri_sparse_DF.grid_S_abs

`quemb.molbe.eri_sparse_DF.`**`grid_S_abs`**(*mol*, *grid_level=2*)

Calculates the overlap matrix $S_ij = \int |phi_i(r)||phi_j(r)|dr$ using numerical integration on a DFT grid.

Parameters

- **`mol`** (`Mole`)
- **`grid_level`** (`int`) – Directly passed on to pyscf grid generation.

Return type

`ndarray`[`tuple`[`int`, `...`], `dtype`[`float64`]]

quemb.molbe.eri_sparse_DF.identify_contiguous_blocks

`quemb.molbe.eri_sparse_DF.`**`identify_contiguous_blocks`**(*X*)

Identify the indices of contiguous blocks in the sequence X.

A block is defined as a sequence of consecutive integers. Returns a list of tuples, where each tuple contains the start and one-past-the-end indices of a block. This means that the returned tuples can be used in slicing operations.

Parameters
X (Sequence[TypeVar(_T, int, integer)])

Return type
list[tuple[int, int]]

Example

```
>>> X = [1, 2, 3, 5, 6, 7, 9, 10]
>>> blocks = identify_contiguous_blocks(X)
>>> assert blocks  == [(0, 3), (3, 6), (6, 8)]
>>> assert X[blocks[1][0] : blocks[1][1]] == [5, 6, 7]
```

quemb.molbe.eri_sparse_DF.to_numba_input

quemb.molbe.eri_sparse_DF.**to_numba_input**(*exch_reachable*)

Convert the reachable orbitals to a list of numpy arrays.

This contains the same information but is a far more efficient layout for numba. Ensures that the start orbs are contiguos and sorted and the target orbs are sorted (but not necessarily contiguos).

Return type
List[ndarray[tuple[int], dtype[TypeVar(_T_target_orb, bound= NewType(OrbitalIdx, integer))]]]

quemb.molbe.eri_sparse_DF.transform_sparse_DF_integral_cpp

quemb.molbe.eri_sparse_DF.**transform_sparse_DF_integral_cpp**(*mf*, *Fobjs*, *auxbasis*, *file_eri_handler*, *AO_coeff_epsilon*, *MO_coeff_epsilon*, *n_threads*)

Return type
None

quemb.molbe.eri_sparse_DF.transform_sparse_DF_integral_cpp_gpu

quemb.molbe.eri_sparse_DF.**transform_sparse_DF_integral_cpp_gpu**(*mf*, *Fobjs*, *auxbasis*, *file_eri_handler*, *AO_coeff_epsilon*, *MO_coeff_epsilon*, *n_threads*)

Return type
None

quemb.molbe.eri_sparse_DF.transform_sparse_DF_integral_nb

quemb.molbe.eri_sparse_DF.**transform_sparse_DF_integral_nb**(*mf*, *Fobjs*, *auxbasis*, *file_eri_handler*, *AO_coeff_epsilon*, *MO_coeff_epsilon*, *n_threads*)

Return type
None

quemb.molbe.eri_sparse_DF.traverse_nonzero

`quemb.molbe.eri_sparse_DF.`**`traverse_nonzero`**`(`*g*, *unique=True*`)`

Traverse the non-zero elements of a semi-sparse 3-index tensor.

Parameters

- **g** (*SemiSparseSym3DTensor*)
- **unique** (`bool`) – Whether to account for 2-fold permutational symmetry and only return `p >= q`.

Return type

`Iterator[tuple[NewType(OrbitalIdx, integer), NewType(OrbitalIdx, integer)]]`

Classes

MutableSemiSparse3DTensor(shape)	Specialised datastructure for semi-sparse 3-indexed tensors.
SemiSparse3DTensor(unique_dense_data, keys, ...)	Specialised datastructure for immutable and semi-sparse 3-indexed tensors.
SemiSparseSym3DTensor(unique_dense_data, ...)	Specialised datastructure for immutable, semi-sparse, and partially symmetric 3-indexed tensors.

quemb.molbe.eri_sparse_DF.MutableSemiSparse3DTensor

`class` `quemb.molbe.eri_sparse_DF.`**`MutableSemiSparse3DTensor`**`(`*shape*`)`

Specialised datastructure for semi-sparse 3-indexed tensors.

For a tensor, T_{ijk}, to be stored in this datastructure we assume

- sparsity along the i, j indices, i.e. $T_{ijk} = 0$ for many i, j
- dense storage along the k index

It can be used for example to store the partially contracted 3-center, 2-electron integrals $(\mu i|P)$, with AO μ, localised MO i, and auxiliary basis indices P. Semi-sparsely, because it is assumed that there are many exchange pairs μ, i which are zero, while the integral along the auxiliary basis P is stored densely as numpy array.

2-fold permutational symmetry for the μ, i pairs is not assumed.

Note that this class is mutable which makes it more flexible in practice, but also less performant for certain operations. If possible, it is recommended to use *SemiSparse3DTensor*.

Attributes

```
class_type = jitclass.MutableSemiSparse3DTensor#153f64500<shape:UniTuple(int64 x
3),naux:int64,_data:DictType[UniTuple(int64 x 2),array(float64, 1d,
A)]<iv=None>,MO_reachable_by_AO:ListType[instance.jitclass.
SortedIntSet#152282b10<_lookup:DictType[int64,bool]<iv=None>,
items:ListType[int64]>],AO_reachable_by_MO:ListType[instance.jitclass.
SortedIntSet#152282b10<_lookup:DictType[int64,bool]<iv=None>,
items:ListType[int64]>]>
```

Methods

`__init__`(shape)	
`to_dense`()	

quemb.molbe.eri_sparse_DF.MutableSemiSparse3DTensor.__init__

`MutableSemiSparse3DTensor.`**`__init__`**`(`*shape*`)`

quemb.molbe.eri_sparse_DF.MutableSemiSparse3DTensor.to_dense

`MutableSemiSparse3DTensor.`**`to_dense`**`()`

quemb.molbe.eri_sparse_DF.SemiSparse3DTensor

class `quemb.molbe.eri_sparse_DF.`**`SemiSparse3DTensor`**`(`*unique_dense_data*, *keys*, *shape*, *AO_reachable_by_MO_with_offsets*, *AO_reachable_by_MO*`)`

Specialised datastructure for immutable and semi-sparse 3-indexed tensors.

For a tensor, T_{ijk}, to be stored in this datastructure we assume

- sparsity along the i, j indices, i.e. $T_{ijk} = 0$ for many i, j
- dense storage along the k index

It can be used for example to store the partially contracted 3-center, 2-electron integrals $(\mu i|P)$, with AO μ, localised MO i, and auxiliary basis indices P. Semi-sparsely, because it is assumed that there are many exchange pairs μ, i which are zero, while the integral along the auxiliary basis P is stored densely as numpy array.

2-fold permutational symmetry for the μ, i pairs is not assumed.

Note that this class is immutable which enables to store the non-zero data in a dense manner, which has some performance benefits.

Attributes

`class_type = jitclass.SemiSparse3DTensor#153f653d0<_keys:array(int64, 1d, C),dense_data:array(float64, 2d, C),shape:UniTuple(int64 x 3),AO_reachable_by_MO_with_offsets:ListType[ListType[UniTuple(int64 x 2)]],AO_reachable_by_MO:ListType[array(int64, 1d, C)],_data:DictType[int64,array(float64, 1d, C)]<iv=None>,naux:int64>`

`shape:` `tuple[int, int, int]`

number of AOs, number of MOs, number of auxiliary functions.

`naux:` `int`

number of auxiliary functions

`AO_reachable_by_MO:` `list[ndarray[tuple[int], dtype[NewType(AOIdx, NewType(OrbitalIdx, integer))]]]`

For a given MO index `i` the `self.AO_reachable_by_MO[i]` returns all `mu` that are assumed unrelevant for $(\mu i|rs)$ after screening. Note that $(pi|P)$ might still be non-zero.

AO_reachable_by_MO_with_offsets: list[list[tuple[int, NewType(AOIdx, NewType(OrbitalIdx, integer))]]]

The following datastructures also return the offset to index the `dense_data` directly and enables very fast loops without having to compute the offset.

```python
for offset, mu in self.AO_reachable_by_MO[i]:
    self.dense_data[offset] == self[mu, i]  # True
```

dense_data: ndarray[tuple[int, ...], dtype[float64]]

Methods

__init__(unique_dense_data, keys, shape, ...)	
to_dense()	Convert to dense 3D tensor

quemb.molbe.eri_sparse_DF.SemiSparse3DTensor.__init__

SemiSparse3DTensor.**__init__**(*unique_dense_data*, *keys*, *shape*, *AO_reachable_by_MO_with_offsets*, *AO_reachable_by_MO*)

quemb.molbe.eri_sparse_DF.SemiSparse3DTensor.to_dense

SemiSparse3DTensor.**to_dense**()

Convert to dense 3D tensor

quemb.molbe.eri_sparse_DF.SemiSparseSym3DTensor

class quemb.molbe.eri_sparse_DF.**SemiSparseSym3DTensor**(*unique_dense_data*, *nao*, *naux*, *exch_reachable*)

Specialised datastructure for immutable, semi-sparse, and partially symmetric 3-indexed tensors.

For a tensor, T_{ijk}, to be stored in this datastructure we assume

- 2-fold permutational symmetry for the i, j indices, i.e. $T_{ijk} = T_{jik}$
- sparsity along the i, j indices, i.e. $T_{ijk} = 0$ for many i, j
- dense storage along the k index

It can be used for example to store the 3-center, 2-electron integrals $(\mu\nu|P)$, with AOs μ, ν and auxiliary basis indices P. Semi-sparsely, because it is assumed that there are many exchange pairs μ, ν which are zero, while the integral along the auxiliary basis P is stored densely as numpy array.

2-fold permutational symmetry for the μ, ν pairs is assumed, i.e.

$$(\mu\nu|P) == (\nu, \mu|P)$$

Note that this class is immutable which enables to store the unique (symmetry), non-zero data in a dense manner, which has some performance benefits.

Attributes

class_type = jitclass.SemiSparseSym3DTensor#153ab7200<_keys:array(int64, 1d, C),shape:UniTuple(int64 x 3),unique_dense_data:array(float64, 2d, C),exch_reachable:ListType[array(int64, 1d, C)],exch_reachable_with_offsets:ListType[ListType[UniTuple(int64 x 2)]],exch_reachable_unique:ListType[array(int64, 1d, C)],exch_reachable_unique_with_offsets:ListType[ListType[UniTuple(int64 x 2)]],_data:DictType[int64,array(float64, 1d, C)]<iv=None>,nao:int64,naux:int64>

unique_dense_data: ndarray[tuple[int, ...], dtype[float64]]

The non-zero data that also accounts for symmetry.

nao: int

number of AOs

naux: int

number of auxiliary functions

shape: tuple[int, int, int]

number of AOs, number of AOs, number of auxiliary functions.

exch_reachable: list[ndarray[tuple[int], dtype[NewType(OrbitalIdx, integer)]]]

For a given p the `exch_reachable[p]` returns all q that are assumed unrelevant for (p q | r s) after screening. Note that (p q | P) might still be non-zero.

exch_reachable_unique: list[ndarray[tuple[int], dtype[NewType(OrbitalIdx, integer)]]]

This is the same as `exch_reachable` except it is accounting for symmetry `p >= q`

exch_reachable_with_offsets: list[list[tuple[int, NewType(OrbitalIdx, integer)]]]

The following datastructures also return the offset to index the `unique_dense_data` directly and enables very fast loops without having to compute the offset.

```
for offset, q in self.exch_reachable_with_offsets[p]:
    self.unique_dense_data[offset] == self[p, q]  # True
```

exch_reachable_unique_with_offsets: list[list[tuple[int, NewType(OrbitalIdx, integer)]]]

The following datastructures also return the offset to index the `unique_dense_data` directly and enables very fast loops without having to compute the offset. It only returns q with `p >= q`

```
for offset, q in self.exch_reachable_unique_with_offsets[p]:
    self.unique_dense_data[offset] == self[p, q]  # True
```

Methods

__init__(unique_dense_data, nao, naux, ...)	
idx(b)	Return compound index
to_dense()	Convert to dense 3D tensor

quemb.molbe.eri_sparse_DF.SemiSparseSym3DTensor.__init__

`SemiSparseSym3DTensor.`**`__init__`**`(`*unique_dense_data*, *nao*, *naux*, *exch_reachable*`)`

quemb.molbe.eri_sparse_DF.SemiSparseSym3DTensor.idx

`SemiSparseSym3DTensor.`**`idx`**`(`*b*`)`

Return compound index

Return type
`int`

quemb.molbe.eri_sparse_DF.SemiSparseSym3DTensor.to_dense

`SemiSparseSym3DTensor.`**`to_dense`**`()`

Convert to dense 3D tensor

quemb.molbe.fragment

Functions

`fragmentate`(mol, *[, frag_type, ...])	Fragment/partitioning definition

quemb.molbe.fragment.fragmentate

`quemb.molbe.fragment.`**`fragmentate`**`(`*mol*, ***, *frag_type='autogen'*, *iao_valence_basis=None*, *print_frags=True*, *write_geom=False*, *n_BE=2*, *frag_prefix='f'*, *frozen_core=False*, *order_by_size=False*, *additional_args=None*`)`

Fragment/partitioning definition

Interfaces the fragmentation functions in MolBE. It defines edge & center for density matching and energy estimation. It also forms the base for IAO/PAO partitioning for a large basis set bootstrap calculation.

Parameters

- **`frag_type`** (`Literal['chemgen', 'graphgen', 'autogen']`) – Name of fragmentation function. ‘chemgen’, ‘autogen’, and ‘graphgen’ are supported. Defaults to ‘autogen’.
- **`n_BE`** (*`int, optional`*) – Specifies the order of bootstrap calculation in the atom-based fragmentation, i.e. BE(n). For a simple linear system A-B-C-D, BE(1) only has fragments [A], [B], [C], [D] BE(2) has [A, B, C], [B, C, D] ben …
- **`mol`** (`Mole`) – This is required for the following `frag_type` options: `"chemgen"`, `"graphgen"`, `"autogen"`
- **`iao_valence_basis`** (`str` | `None`) – Name of minimal basis set for IAO scheme. ‘sto-3g’ suffice for most cases.
- **`frozen_core`** (`bool`) – Whether to invoke frozen core approximation. This is set to False by default
- **`print_frags`** (`bool`) – Whether to print out list of resulting fragments. True by default
- **`write_geom`** (`bool`) – Whether to write ‘fragment.xyz’ file which contains all the fragments in cartesian coordinates.
- **`frag_prefix`** (`str`) – Prefix to be appended to the fragment datanames. Useful for managing fragment scratch directories.

- **order_by_size** (bool) – Order the fragments by descending size. This can be beneficial for better load-balancing.
- **additional_args** (*AutogenArgs* | *ChemGenArgs* | *GraphGenArgs* | None) – Additional arguments for different fragmentation functions.

Return type
FragPart

quemb.molbe.graphfrag

Functions

graphgen(mol[, n_BE, frozen_core, ...])	Generate fragments via adjacency graph.

quemb.molbe.graphfrag.graphgen

quemb.molbe.graphfrag.**graphgen**(*mol*, *n_BE=2*, *frozen_core=True*, *remove_nonunique_frags=True*, *frag_prefix='f'*, *connectivity='euclidean'*, *iao_valence_basis=None*, *cutoff=0.0*, *export_graph_to=None*, *print_frags=True*)

Generate fragments via adjacency graph.

Generalizes the BEn fragmentation scheme to arbitrary fragment sizes using a graph theoretic heuristic. In brief: atoms are assigned to nodes in an adjacency graph and edges are weighted by some distance metric. For a given fragment center site, Dijkstra's algorithm is used to find the shortest path from that center to its neighbors. The number of nodes visited on that shortest path determines the degree of separation of the corresponding neighbor. I.e., all atoms whose shortest paths from the center site visit at most 1 node must be direct neighbors to the center site, which gives BE2-type fragments; all atoms whose shortest paths visit at most 2 nodes must then be second-order neighbors, hence BE3; and so on. NOTE: Currently does not support periodic calculations or IAOs.

Parameters

- **mol** (Mole) – The gto.Mole object.
- **n_BE** (str | int) – The order of nearest neighbors (with respect to the center atom) included in a fragment. Supports all 'n' in 'BEn'. Defaults to 2.
- **frozen_core** (bool) – Whether to exclude core AO indices from the fragmentation process. True by default.
- **remove_nonunique_frags** (bool) – Whether to remove fragments which are strict subsets of another fragment in the system. Defaults to True.
- **frag_prefix** (str) – Prefix to be appended to the fragment datanames. Useful for managing fragment scratch directories. Defaults to "f".
- **connectivity** (Literal['euclidean']) – Keyword string specifying the distance metric to be used for edge weights in the fragment adjacency graph. Currently supports "euclidean" (which uses the square of the distance between atoms in real space to determine connectivity within a fragment.) Defaults to "euclidean".
- **cutoff** (float) – Atoms with an edge weight beyond *cutoff* will be excluded from the *shortest_path* calculation. When set to 0.0, *cutoff* will be determined dynamically based on the magnitude of *n_BE*. Defaults to 0.0. NOTE: For very large systems a smaller *cutoff* often significantly improves runtime, but can sometimes affect the fragmentation pattern if set *too* small.

- **`export_graph_to`** (`Path` | `None`) – If not *None*, specifies the path to which the fragment connectivity graph will be saved. Defaults to None.
- **`print_frags`** (`bool`) – Whether to print simplified string representations of fragment connectivity graphs. Defaults to True.

Return type

`FragPart`

Classes

`GraphGenArgs`(*[, connectivity, cutoff, ...])	Graphgen specific arguments.
`GraphGenUtility`()	Utility functions for handling graphs in *graphgen()*.

quemb.molbe.graphfrag.GraphGenArgs

`class quemb.molbe.graphfrag.`**`GraphGenArgs`**`(`*, *connectivity='euclidean'*, *cutoff=0.0*, *remove_nonunique_frags=True*`)`

Graphgen specific arguments.

Parameters

- **`connectivity`** – Keyword string specifying the distance metric to be used for edge weights in the fragment adjacency graph. Currently supports "euclidean" (which uses the square of the distance between atoms in real space to determine connectivity within a fragment.)
- **`cutoff`** – Atoms with an edge weight beyond *cutoff* will be excluded from the *shortest_path* calculation. This is crucial when handling very large systems, where computing the shortest paths from all to all becomes non-trivial. Defaults to 20.0.
- **`remove_nonunique_frags`** – Whether to remove fragments which are strict subsets of another fragment in the system. True by default.

Attributes

`connectivity: Final[Literal['euclidean']]`

`cutoff: Final[float]`

`remove_nonunique_frags: Final[bool]`

Methods

`__init__`(*[, connectivity, cutoff, ...])	Method generated by attrs for class GraphGenArgs.

quemb.molbe.graphfrag.GraphGenArgs.__init__

`GraphGenArgs.`**`__init__`**`(`*, *connectivity='euclidean'*, *cutoff=0.0*, *remove_nonunique_frags=True*`)`

Method generated by attrs for class GraphGenArgs.

quemb.molbe.graphfrag.GraphGenUtility

class quemb.molbe.graphfrag.**GraphGenUtility**

Utility functions for handling graphs in *graphgen()*.

Methods

__init__()	Method generated by attrs for class GraphGenUtility.
export_graph(edge_list, adx_map, ...[, ...])	Export a visual representation of a fragment-based adjacency graph to a PNG.
get_subgraphs(motifs_per_frag, edge_list, ...)	Construct labeled subgraphs for fragments using motif, edge, and label data.

quemb.molbe.graphfrag.GraphGenUtility.__init__

GraphGenUtility.**__init__**()

Method generated by attrs for class GraphGenUtility.

quemb.molbe.graphfrag.GraphGenUtility.export_graph

static GraphGenUtility.**export_graph**(*edge_list*, *adx_map*, *adjacency_graph*, *origin_per_frag*, *dnames*, *outdir=None*, *outname='AdjGraph'*, *cmap='cubehelix'*, *node_position='coordinates'*)

Export a visual representation of a fragment-based adjacency graph to a PNG.

This function draws a directed network graph using a provided adjacency structure, with fragments color-coded and displayed using either coordinate-based or spring-layout node positioning, and saves the resulting image to the specified output directory.

Parameters

- **edge_list** (list[Sequence]) – A list of edge groupings, each corresponding to a fragment's set of directed edges.
- **adx_map** (dict) – Mapping from node indices to node metadata (must include 'coord' and 'label' keys).
- **adjacency_graph** (Graph) – The NetworkX graph containing node and edge connectivity.
- **origin_per_frag** (list[Sequence[int]]) – A list of node index groups, each corresponding to the origin nodes of a fragment.
- **dnames** (list[str]) – Names for each fragment, used in the legend.
- **outdir** (Path | None) – If specified as *Path*: export .png to *outdir*. If *None*: no .png is exported, but (fix, ax) are still returned.
- **outname** (str) – Filename (without extension) for the output image. Default is "AdjGraph".
- **cmap** (str) – Matplotlib colormap name used to color-code fragments. Default is "cubehelix".
- **node_position** (str) – Node positioning strategy: "coordinates" for z-shifted spatial coordinates, or "spring" for force-directed layout. Default is "coordinates".

Returns
(fix, ax) – Returns the corresponding *pyplot* objects.

Return type
tuple[*object*, *object*]

Notes

- Assumes that *adx_map* contains a *"coord"* field for all nodes: *[x, y, z]* coordinates when using *"coordinates"* positioning.
- Colors are assigned per fragment based on the chosen colormap.
- Nodes are drawn in two layers to produce a bordered effect.
- Arcs between nodes are radially offset to distinguish overlapping edges.

quemb.molbe.graphfrag.GraphGenUtility.get_subgraphs

`static GraphGenUtility.get_subgraphs(`*motifs_per_frag*, *edge_list*, *origin_per_frag*, *adx_map*, *fdx=None*, *options={}*`)`

Construct labeled subgraphs for fragments using motif, edge, and label data.

If a fragment index (*fdx*) is provided, returns a dictionary containing the subgraph for that single fragment. Otherwise, returns a dictionary of subgraphs for all fragments, keyed by their respective fragment indices (*fdx*).

Parameters

- **`motifs_per_frag`** (`list[Sequence[int]]`) – A list where each item is a sequence of node indices (motifs) per fragment.
- **`edge_list`** (`list[Sequence]`) – A list of edge groupings, each corresponding to a fragment's set of edges.
- **`origin_per_frag`** (`list[Sequence[int]]`) – A list where each item is a sequence of node indices marking origin nodes within each fragment.
- **`adx_map`** (`dict`) – A mapping from node index to metadata, where each value must contain a *"label"* field for node annotation.
- **`fdx`** (`int` | `None`) – Index of a specific fragment to extract. If None (default), subgraphs for all fragments are returned.
- **`options`** (`dict`) – Optional keyword arguments passed to the *nx.Graph* constructor for each subgraph.

Returns
subgraph_dict – a dictionary mapping fragment indices to *networkx.graph* objects. If *fdx* is given, the dictionary will contain only one entry for that fragment.

Return type
dict

Notes

- Origin nodes are highlighted by enclosing their labels in square brackets.
- Returned graphs include labeled nodes and edges per fragment.
- The structure and labels are derived from *adx_map* and *origin_per_frag*.

quemb.molbe.helper

Functions

are_equal(m1, m2)	
get_core(mol)	Calculate the number of cores for each atom in the molecule.
get_eri(i_frag, Nao[, symm, ignore_symm, ...])	Retrieve and optionally restore electron repulsion integrals (ERI) from an HDF5 file.
get_frag_energy(mo_coeffs, nsocc, n_frag, ...)	Compute the fragment energy.
get_frag_energy_u(mo_coeffs, nsocc, n_frag, ...)	Compute the fragment energy for unrestricted calculations
get_scf0bj(h1, Eri, nocc[, dm0])	Initialize and run a restricted Hartree-Fock (RHF) calculation.
get_veff(eri_, dm, S, TA, hf_veff)	Calculate the effective HF potential (Veff) for a given density matrix and electron repulsion integrals.

quemb.molbe.helper.are_equal

quemb.molbe.helper.**are_equal**(*m1*, *m2*)

Return type
bool

quemb.molbe.helper.get_core

quemb.molbe.helper.**get_core**(*mol*)

Calculate the number of cores for each atom in the molecule.

Parameters
mol (Mole | Cell) – Molecule or cell object from PySCF.

Returns
(Ncore, idx, corelist)

Return type
tuple

quemb.molbe.helper.get_eri

quemb.molbe.helper.**get_eri**(*i_frag*, *Nao*, *symm=8*, *ignore_symm=False*, *eri_file='eri_file.h5'*)

Retrieve and optionally restore electron repulsion integrals (ERI) from an HDF5 file.

This function reads the ERI for a given fragment from an HDF5 file, and optionally restores the symmetry of the ERI.

Parameters

- **i_frag** (*str*) – Fragment identifier used to locate the ERI data in the HDF5 file.
- **Nao** (*int*) – Number of atomic orbitals.
- **symm** (*int, optional*) – Symmetry of the ERI. Defaults to 8.
- **ignore_symm** (*bool, optional*) – If True, the symmetry step is skipped. Defaults to False.

- **eri_file** (*str, optional*) – Filename of the HDF5 file containing the electron repulsion integrals. Defaults to 'eri_file.h5'.

Returns
Electron repulsion integrals, possibly restored with symmetry.

Return type
ndarray

quemb.molbe.helper.get_frag_energy

`quemb.molbe.helper.get_frag_energy`(*mo_coeffs*, *nsocc*, *n_frag*, *weight_and_relAO_per_center*, *TA*, *h1*, *rdm1*, *rdm2s*, *dname*, *veff0=None*, *veff=None*, *use_cumulant=True*, *eri_file='eri_file.h5'*)

Compute the fragment energy.

This function calculates the energy contribution of a fragment within a larger molecular system using the provided molecular orbital coefficients, density matrices, and effective potentials.

Parameters

- **mo_coeffs** (*ndarray*) – Molecular orbital coefficients.
- **nsocc** (*int*) – Number of occupied orbitals.
- **n_frag** (*int*) – Number of fragment sites.
- **weight_and_relAO_per_center** – List containing energy scaling factors and indices.
- **TA** (*ndarray*) – Transformation matrix.
- **h1** (*ndarray*) – One-electron Hamiltonian.
- **rdm1** (*ndarray*) – One-particle density matrix.
- **rdm2s** (*ndarray*) – Two-particle density matrix.
- **dname** (*str*) – Dataset name in the HDF5 file.
- **veff0** (*ndarray*) – veff0 matrix, the original hf_veff in the fragment Schmidt space
- **veff** (*ndarray*) – veff for non-cumulant energy expression
- **use_cumulant** (*bool*) – Whether to return cumulant energy, by default True
- **eri_file** (*str, optional*) – Filename of the HDF5 file containing the electron repulsion integrals. Defaults to 'eri_file.h5'.

Returns
List containing the energy contributions: [e1_tmp, e2_tmp, ec_tmp].

Return type
list

quemb.molbe.helper.get_frag_energy_u

`quemb.molbe.helper.get_frag_energy_u`(*mo_coeffs*, *nsocc*, *n_frag*, *weight_and_relAO_per_center*, *TA*, *h1*, *hf_veff*, *rdm1*, *rdm2s*, *dname*, *eri_file='eri_file.h5'*, *gcores=None*, *frozen=False*, *veff0=None*)

Compute the fragment energy for unrestricted calculations

This function calculates the energy contribution of a fragment within a larger molecular system using the provided molecular orbital coefficients, density matrices, and effective potentials.

Parameters

- **mo_coeffs** (*tuple of ndarray*) – Molecular orbital coefficients.
- **nsocc** (*tuple of int*) – Number of occupied orbitals.
- **n_frag** (*tuple of int*) – Number of fragment sites.
- **weight_and_relAO_per_center** – List containing energy scaling factors and indices.
- **TA** (*tuple of ndarray*) – Transformation matrix.
- **h1** (*tuple of ndarray*) – One-electron Hamiltonian.
- **hf_veff** (*tuple of ndarray*) – Hartree-Fock effective potential.
- **rdm1** (*tuple of ndarray*) – One-particle density matrix.
- **rdm2s** (*tuple of ndarray*) – Two-particle density matrix.
- **dname** (*list*) – Dataset name in the HDF5 file.
- **eri_file** (*str, optional*) – Filename of the HDF5 file containing the electron repulsion integrals. Defaults to 'eri_file.h5'.
- **gcores**
- **frozen** (*bool, optional*) – Indicate frozen core. Default is False

Returns
List containing the energy contributions: [e1_tmp, e2_tmp, ec_tmp].

Return type
list

quemb.molbe.helper.get_scfObj

`quemb.molbe.helper.`**`get_scfObj`**(*h1*, *Eri*, *nocc*, *dm0=None*)

Initialize and run a restricted Hartree-Fock (RHF) calculation.

This function sets up an SCF (Self-Consistent Field) object using the provided one-electron Hamiltonian, electron repulsion integrals, and number of occupied orbitals. It then runs the SCF procedure, optionally using an initial density matrix.

Parameters

- **h1** (*ndarray*) – One-electron Hamiltonian matrix.
- **Eri** (*ndarray*) – Electron repulsion integrals.
- **nocc** (*int*) – Number of occupied orbitals.
- **dm0** (*ndarray, optional*) – Initial density matrix. If not provided, the SCF calculation will start from scratch. Defaults to None.

Returns
mf_ – The SCF object after running the Hartree-Fock calculation.

Return type
RHF

quemb.molbe.helper.get_veff

quemb.molbe.helper.**get_veff**(*eri_*, *dm*, *S*, *TA*, *hf_veff*)

Calculate the effective HF potential (Veff) for a given density matrix and electron repulsion integrals.

This function computes the effective potential by transforming the density matrix, computing the Coulomb (J) and exchange (K) integrals.

Parameters

- **eri** (*ndarray*) – Electron repulsion integrals.
- **dm** (*ndarray*) – Density matrix. 2D array.
- **S** (*ndarray*) – Overlap matrix.
- **TA** (*ndarray*) – Transformation matrix.
- **hf_veff** (*ndarray*) – Hartree-Fock effective potential for the full system.

Returns

Effective HF potential in the embedding basis.

Return type

ndarray

quemb.molbe.lo

Functions

get_iao(Co, S12, S1, S2, mol, iao_valence_basis)	Gets symmetrically orthogonalized IAO coefficient matrix from system MOs Derived from G.
get_loc(mol, C[, method, pop_method, init_guess])	Import, initialize, and call localization procedure *method* for C from *PySCF*
get_pao(Ciao, S1, S12, mol, iao_valence_basis)	Get (symmetrically though often canonically) orthogonalized PAOs from given (localized) IAOs Defined in detail in J.
get_xovlp(mol[, basis])	Gets overlap matrix between the two bases and in secondary basis.
remove_core_mo(Clo, Ccore, S[, thr])	Remove core molecular orbitals from localized Clo

quemb.molbe.lo.get_iao

quemb.molbe.lo.**get_iao**(*Co*, *S12*, *S1*, *S2*, *mol*, *iao_valence_basis*, *iao_loc_method='lowdin'*)

Gets symmetrically orthogonalized IAO coefficient matrix from system MOs Derived from G. Knizia: J. Chem. Theory Comput. 2013, 9, 11, 4834–4843 Note: same function as *get_iao_from_s12* in frankenstein

Parameters

- **Co** (ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]) – occupied MO coefficient matrix with core
- **S12** (ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]) – ovlp between working (large) basis and valence (minimal) basis can be thought of as working basis in valence basis
- **S1** (ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]) – AO ovlp matrix, in working (large) basis

- **S2** (ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]) – AO ovlp matrix, in valence (minimal) basis
- **mol** (Mole) – mol object
- **iao_valence_basis** (str) – (minimal-like) basis used for valence orbitals
- **iao_loc_method** (Literal['lowdin', 'boys', 'PM', 'ER']) – Localization method for the IAOs and PAOs. If symmetric orthogonalization is used, the overlap matrices between the valence (minimal) and working (large) basis are determined by separating S1 (working) by AO labels. If other localization methods are used, these matrices are calculated in full Default is lowdin

Returns
(symmetrically orthogonalized)

Return type
Ciao *quemb.shared.typing.Matrix*

quemb.molbe.lo.get_loc

`quemb.molbe.lo.`**`get_loc`**`(`*mol*, *C*, *method='ER'*, *pop_method=None*, *init_guess='atomic'*`)`

Import, initialize, and call localization procedure *method* for C from *PySCF*

Parameters

- **mol** (Mole) – mol object
- **C** (ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]) – MO coefficients
- **method** (Literal['cholesky', 'ER', 'PM', 'boys']) – Localization method. Options include: EDMINSTON-RUEDENBERG, ER; PIPEK-MIZEY, PM; FOSTER-BOYS, boys; cholesky;
- **pop_method** (str | None) – Method for calculating orbital population, by default 'meta-lowdin' See pyscf.lo for more details and options. This is only used for Pipek-Mezey localization
- **init_guess** (ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]] | str | None) – Initial guess for localization optimization. Default is *atomic*, See pyscf.lo for more details and options

Returns
mlo – Localized mol object

Return type
quemb.shared.typing.Matrix

quemb.molbe.lo.get_pao

`quemb.molbe.lo.`**`get_pao`**`(`*Ciao*, *S1*, *S12*, *mol*, *iao_valence_basis*, *iao_loc_method='lowdin'*`)`

Get (symmetrically though often canonically) orthogonalized PAOs from given (localized) IAOs Defined in detail in J. Chem. Theory Comput. 2024, 20, 24, 10912–10921

Parameters

- **Ciao** (ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]) – the orthogonalized IAO coefficient matrix (output of :func:get_iao)
- **S1** (ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]) – ao ovlp matrix in working (large) basis

- **S12** (ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]) – ovlp between working (large) basis and valence (minimal) basis
- **mol** (Mole) – mol object
- **iao_valence_basis** (str) – (minimal-like) basis used for valence orbitals
- **iao_loc_method** (Literal['lowdin', 'boys', 'PM', 'ER']) – Localization method for the IAOs and PAOs. If symmetric orthogonalization is used, the overlap matrices between the valence (minimal) and working (large) basis are determined by separating S1 (working) by AO labels. If other localization methods are used, these matrices are calculated in full Default is lowdin

Returns
Cpao – (orthogonalized)

Return type
quemb.shared.typing.Matrix

quemb.molbe.lo.get_xovlp

quemb.molbe.lo.**get_xovlp**(*mol*, *basis='sto-3g'*)

Gets overlap matrix between the two bases and in secondary basis. Used for IAOs: returns the overlap between valence (minimal) and working (large) bases and overlap in the minimal basis

Parameters

- **mol** (Mole) – mol object to get working (large) and valence (minimal) basis
- **basis** (str) – the IAO valence (minimal-like) basis, Knizia recommended 'minao'

Return type
tuple[ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]], ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]]

Returns

- **S12** (*numpy.ndarray*) – Overlap of two basis sets
- **S22** (*numpy.ndarray*) – Overlap in new basis set

quemb.molbe.lo.remove_core_mo

quemb.molbe.lo.**remove_core_mo**(*Clo*, *Ccore*, *S*, *thr=0.5*)

Remove core molecular orbitals from localized Clo

Return type
ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]

quemb.molbe.mbe

Module Attributes

IntTransforms	Literal type describing allowed transformation strategies.

quemb.molbe.mbe.IntTransforms

quemb.molbe.mbe.**IntTransforms**

Literal type describing allowed transformation strategies.

alias of `Literal`[‘in-core’, ‘out-core-DF’, ‘int-direct-DF’, ‘sparse-DF-cpp’, ‘sparse-DF-nb’, ‘sparse-DF-cpp-gpu’, ‘sparse-DF-nb-gpu’]

Functions

initialize_pot(n_frag, relAO_per_edge)	Initialize the potential array for bootstrap embedding.

quemb.molbe.mbe.initialize_pot

quemb.molbe.mbe.**initialize_pot**(*n_frag*, *relAO_per_edge*)

Initialize the potential array for bootstrap embedding.

This function initializes a potential array for a given number of fragments (`n_frag`) and their corresponding edge indices (`relAO_per_edge`). The potential array is initialized with zeros for each pair of edge site indices within each fragment, followed by an additional zero for the global chemical potential.

Parameters

- **n_frag** (*int*) – Number of fragments.
- **relAO_per_edge** (*list of list of list of int*) – List of edge indices for each fragment. Each element is a list of lists, where each sublist contains the indices of edge sites for a particular fragment.

Returns

Initialized potential array with zeros.

Return type

list of *float*

Classes

BE(mf, fobj, *[, eri_file, lo_method, ...])	Class for handling bootstrap embedding (BE) calculations.
storeBE(Nocc, hf_veff, hcore, S, C, hf_dm, ...)	

quemb.molbe.mbe.BE

class quemb.molbe.mbe.**BE**(*mf*, *fobj*, *, *eri_file='eri_file.h5'*, *lo_method='lowdin'*, *iao_loc_method='lowdin'*, *lo_bath_post_schmidt=None*, *pop_method=None*, *restart=False*, *restart_file='storebe.pk'*, *nproc=1*, *ompnum=4*, *thr_bath=1e-10*, *scratch_dir=None*, *int_transform='in-core'*, *auxbasis=None*, *MO_coeff_epsilon=1e-05*, *AO_coeff_epsilon=1e-10*)

Class for handling bootstrap embedding (BE) calculations.

This class encapsulates the functionalities required for performing bootstrap embedding calculations, including setting up the BE environment, initializing fragments, performing SCF calculations, and evaluating energies.

mf

PySCF mean-field object.

Type

SCF

fobj

Fragment object containing sites, centers, edges, and indices.

Type

FragPart

eri_file

Path to the file storing two-electron integrals.

Type

str

lo_method

Method for orbital localization, default is "lowdin".

Methods

`__init__`(mf, fobj, *[, eri_file, lo_method, ...])	Constructor for BE object.
`compute_energy_full`([approx_cumulant, ...])	Compute the total energy using rdms in the full basis.
`get_be_error_jacobian`([jac_solver])	
`initialize`(eri_, *, restart, int_transform)	Initialize the Bootstrap Embedding calculation.
`localize`(lo_method, fobj[, iao_loc_method, ...])	Molecular orbital localization
`oneshot`([solver, use_cumulant, nproc, ...])	Perform a one-shot bootstrap embedding calculation.
`optimize`([solver, method, only_chem, ...])	BE optimization function
`print_ini`()	Print initialization banner for the MOLBE calculation.
`rdm1_fullbasis`([return_ao, only_rdm1, ...])	Compute the one- and two-particle reduced density matrices (RDM1 and RDM2).
`read_heff`([heff_file])	Read the effective Hamiltonian from a file.
`save`([save_file])	Save intermediate results for restart.
`update_fock`([heff])	Update the Fock matrix for each fragment with the effective Hamiltonian.
`write_heff`([heff_file])	Write the effective Hamiltonian to a file.

quemb.molbe.mbe.BE.__init__

BE.**__init__**(*mf*, *fobj*, *, *eri_file='eri_file.h5'*, *lo_method='lowdin'*, *iao_loc_method='lowdin'*, *lo_bath_post_schmidt=None*, *pop_method=None*, *restart=False*, *restart_file='storebe.pk'*, *nproc=1*, *ompnum=4*, *thr_bath=1e-10*, *scratch_dir=None*, *int_transform='in-core'*, *auxbasis=None*, *MO_coeff_epsilon=1e-05*, *AO_coeff_epsilon=1e-10*)

Constructor for BE object.

Parameters

- **mf** (`SCF`) – PySCF mean-field object.
- **fobj** (`FragPart[Mole]`) – Fragment object containing sites, centers, edges, and indices.
- **eri_file** (`str` | `PathLike`) – Path to the file storing two-electron integrals.

- **lo_method** (Literal['lowdin', 'boys', 'ER', 'PM', 'IAO']) – Method for orbital localization, by default "lowdin".
- **iao_loc_method** (Literal['lowdin', 'boys', 'PM', 'ER']) – Method for IAO localization, by default "lowdin"
- **lo_method_bath_post_schmidt** – If not None, then perform a localization of the bath orbitals **after** the Schmidt decomposition.
- **pop_method** (str | None) – Method for calculating orbital population, by default 'meta-lowdin' See pyscf.lo for more details and options
- **restart** (bool) – Whether to restart from a previous calculation, by default False.
- **restart_file** (str | PathLike) – Path to the file storing restart information, by default 'storebe.pk'.
- **nproc** (int) – Number of processors for parallel calculations, by default 1. If set to >1, threaded parallel computation is invoked.
- **ompnum** (int) – Number of OpenMP threads, by default 4.
- **thr_bath** (*float*,) – Threshold for bath orbitals in Schmidt decomposition
- **scratch_dir** (*WorkDir* | None) – Scratch directory.
- **int_transform** (Literal['in-core', 'out-core-DF', 'int-direct-DF', 'sparse-DF-cpp', 'sparse-DF-nb', 'sparse-DF-cpp-gpu', 'sparse-DF-nb-gpu']) – The possible integral transformations.
  - "in-core" (default): Use a dense representation of integrals in memory without density fitting (DF) and transform in-memory.
  - "out-core-DF": Use a dense, DF representation of integrals, the DF integrals $(\mu, \nu|P)$ are stored on disc.
  - "int-direct-DF": Use a dense, DF representation of integrals, the required DF integrals $(\mu, \nu|P)$ are computed and fitted on-demand for each fragment.
  - "sparse-DF-cpp": Use a sparse, DF representation of integrals, and avoid recomputation of elements that are shared across fragments. Uses a C++ implementation for performance heavy code.
  - "sparse-DF-nb": Use a sparse, DF representation of integrals, and avoid recomputation of elements that are shared across fragments. Uses a numba implementation for performance heavy code.
  - "sparse-DF-cpp-gpu": Use a sparse, DF representation of integrals, and avoid recomputation of elements that are shared across fragments. Uses a C++ + CUDDA implementation for performance heavy code, only available when compiled with CUDABlas.
  - "sparse-DF-nb-gpu": Use a sparse, DF representation of integrals, and avoid recomputation of elements that are shared across fragments. Uses a numba implementation + cupy for performance heavy code. Only available if cupy is installed.
- **auxbasis** (str | None) – Auxiliary basis for density fitting, by default None (uses default auxiliary basis defined in PySCF). Only relevant for `int_transform in {"int-direct-DF", "sparse-DF"}`.
- **MO_coeff_epsilon** (float) – The cutoff value of the absolute overlap $\int |\phi_i||\varphi_\mu|$ when a MO coefficient i and an AO coefficient μ are considered to be connected for sparsity screening. Smaller value means less screening.

- **AO_coeff_epsilon** (float) – The cutoff value of the absolute overlap $\int |\varphi_\mu||\varphi_\nu|$ when two AO coefficient μ, ν are considered to be connected for sparsity screening. Here the absolute overlap matrix is used. Smaller value means less screening.

quemb.molbe.mbe.BE.compute_energy_full

`BE.compute_energy_full`(*approx_cumulant=False*, *use_full_rdm=False*, *return_rdm=True*)

Compute the total energy using rdms in the full basis.

Parameters

- **approx_cumulant** (*bool, optional*) – If True, use an approximate cumulant for the energy computation. Default is False.
- **use_full_rdm** (*bool, optional*) – If True, use the full two-particle RDM for energy computation. Default is False.
- **return_rdm** (*bool, optional*) – If True, return the computed reduced density matrices (RDMs). Default is True.

Returns

If `return_rdm` is True, returns a tuple containing the one-particle and two-particle reduced density matrices (RDM1 and RDM2). Otherwise, returns None.

Return type

tuple of *ndarray* or *None*

Notes

This function computes the total energy in the full basis, with options to use approximate or true cumulants, and to return the reduced density matrices (RDMs). The energy components are printed as part of the function's output.

quemb.molbe.mbe.BE.get_be_error_jacobian

`BE.get_be_error_jacobian`(*jac_solver='HF'*)

Return type

`ndarray[tuple[int, ...], dtype[floating]]`

quemb.molbe.mbe.BE.initialize

`BE.initialize`(*eri_*, ***, *restart*, *int_transform*)

Initialize the Bootstrap Embedding calculation.

Parameters

- **eri** (*ndarray*) – Electron repulsion integrals.
- **restart** (*bool, optional*) – Whether to restart from a previous calculation, by default False.
- **int_transfrom** – Which integral transformation to perform.

Return type

`None`

quemb.molbe.mbe.BE.localize

`BE.localize`(*lo_method*, *fobj*, *iao_loc_method='lowdin'*, *iao_valence_only=False*, *pop_method=None*, *init_guess=None*, *hstack=False*, *save=True*)

Molecular orbital localization

Performs molecular orbital localization computations. For large basis, IAO is recommended augmented with PAO orbitals.

NOTE: For molecular systems, with frozen core, the core and valence are localized TOGETHER. This is not the case for periodic systems.

Parameters

- **lo_method** (`Literal['lowdin', 'boys', 'ER', 'PM', 'IAO']`) – Method for orbital localization. Supports "lowdin" (Löwdin or symmetric orthogonalization), "boys" (Foster-Boys), "PM" (Pipek-Mezey", and "ER" (Edmiston-Rudenberg). By default "lowdin"
- **fobj** (*FragPart*)
- **iao_loc_method** (`Literal['lowdin', 'boys', 'PM', 'ER']`) – Name of localization method in quantum chemistry for the IAOs and PAOs. Options include "lowdin", "boys", 'PM', 'ER' (as documented in PySCF). Default is "lowdin". If not using lowdin, we suggest using 'PM', as it is more robust than 'boys' localization and less expensive than 'ER'
- **iao_valence_only** (*bool*) – If this option is set to True, all calculation will be performed in the valence basis in the IAO partitioning. Default is False. This is an experimental feature: the returned energy is not accurate

Return type

`None` | `ndarray[tuple[int, ...], dtype[float64]]`

quemb.molbe.mbe.BE.oneshot

`BE.oneshot`(*solver='CCSD'*, *use_cumulant=True*, *nproc=1*, *ompnum=4*, *solver_args=None*)

Perform a one-shot bootstrap embedding calculation.

Parameters

- **solver** (`Literal['MP2', 'CCSD', 'FCI', 'HCI', 'SHCI', 'SCI', 'DMRG']`) – High-level quantum chemistry method, by default 'CCSD'. 'CCSD', 'FCI', and variants of selected CI are supported.
- **use_cumulant** (`bool`) – Whether to use the cumulant energy expression, by default True.
- **nproc** (`int`) – Number of processors for parallel calculations, by default 1. If set to >1, multi-threaded parallel computation is invoked.
- **ompnum** (`int`) – Number of OpenMP threads, by default 4.

Return type

`None`

quemb.molbe.mbe.BE.optimize

`BE.optimize`(*solver='CCSD'*, *method='QN'*, *only_chem=False*, *use_cumulant=True*, *conv_tol=1e-06*, *relax_density=False*, *jac_solver='HF'*, *nproc=1*, *ompnum=4*, *max_iter=500*, *trust_region=False*, *solver_args=None*)

BE optimization function

Interfaces BEOPT to perform bootstrap embedding optimization.

Parameters

- **solver** (Literal['MP2', 'CCSD', 'FCI', 'HCI', 'SHCI', 'SCI', 'DMRG']) – High-level solver for the fragment, by default "CCSD"
- **method** (str) – Optimization method, by default 'QN'
- **only_chem** (bool) – If true, density matching is not performed – only global chemical potential is optimized, by default False
- **use_cumulant** (bool) – Whether to use the cumulant energy expression, by default True.
- **conv_tol** (float) – Convergence tolerance, by default 1.e-6
- **relax_density** (bool) – Whether to use relaxed or unrelaxed densities, by default False This option is for using CCSD as solver. Relaxed density here uses Lambda amplitudes, whereas unrelaxed density only uses T amplitudes. c.f. See http://classic.chem.msu.su/cgi-bin/ceilidh.exe/gran/gamess/forum/?C34df668afbHW-7216-1405+00.htm for the distinction between the two
- **max_iter** (int) – Maximum number of optimization steps, by default 500
- **nproc** (int) – Total number of processors assigned for the optimization. Defaults to 1. When nproc > 1, Python multithreading is invoked.
- **ompnum** (int) – If nproc > 1, ompnum sets the number of cores for OpenMP parallelization. Defaults to 4
- **jac_solver** (Literal['HF', 'MP2', 'CCSD']) – Method to form Jacobian used in optimization routine, by default HF. Options include HF, MP2, CCSD
- **trust_region** (bool) – Use trust-region based QN optimization, by default False

Return type

None

quemb.molbe.mbe.BE.print_ini

`BE.`**`print_ini`**`()`

Print initialization banner for the MOLBE calculation.

quemb.molbe.mbe.BE.rdm1_fullbasis

`BE.`**`rdm1_fullbasis`**`(`*return_ao=True*, *only_rdm1=False*, *only_rdm2=False*, *return_lo=False*, *return_RDM2=True*, *print_energy=False*`)`

Compute the one- and two-particle reduced density matrices (RDM1 and RDM2).

Parameters

- **return_ao** (*bool*, *optional*) – Whether to return the RDMs in the AO basis. Default is True.
- **only_rdm1** (*bool*, *optional*) – Whether to compute only the RDM1. Default is False.
- **only_rdm2** (*bool*, *optional*) – Whether to compute only the RDM2. Default is False.
- **return_lo** (*bool*, *optional*) – Whether to return the RDMs in the localized orbital (LO) basis. Default is False.
- **return_RDM2** (*bool*, *optional*) – Whether to return the two-particle RDM (RDM2). Default is True.

- **print_energy** (*bool, optional*) – Whether to print the energy contributions. Default is False.

Returns

- **rdm1AO** (*numpy.ndarray*) – The one-particle RDM in the AO basis.
- **rdm2AO** (*numpy.ndarray*) – The two-particle RDM in the AO basis (if return_RDM2 is True).
- **rdm1LO** (*numpy.ndarray*) – The one-particle RDM in the LO basis (if return_lo is True).
- **rdm2LO** (*numpy.ndarray*) – The two-particle RDM in the LO basis (if return_lo and return_RDM2 are True).
- **rdm1MO** (*numpy.ndarray*) – The one-particle RDM in the molecular orbital (MO) basis (if return_ao is False).
- **rdm2MO** (*numpy.ndarray*) – The two-particle RDM in the MO basis (if return_ao is False and return_RDM2 is True).

quemb.molbe.mbe.BE.read_heff

BE.**read_heff**(*heff_file='bepotfile.h5'*)

Read the effective Hamiltonian from a file.

Parameters

heff_file (*str, optional*) – Path to the file storing effective Hamiltonian, by default 'bepotfile.h5'.

quemb.molbe.mbe.BE.save

BE.**save**(*save_file='storebe.pk'*)

Save intermediate results for restart.

Parameters

save_file (str | PathLike) – Path to the file storing restart information, by default 'storebe.pk'.

Return type

None

quemb.molbe.mbe.BE.update_fock

BE.**update_fock**(*heff=None*)

Update the Fock matrix for each fragment with the effective Hamiltonian.

Parameters

heff (*list of ndarray, optional*) – List of effective Hamiltonian matrices for each fragment, by default None.

Return type

None

quemb.molbe.mbe.BE.write_heff

BE.**write_heff**(*heff_file='bepotfile.h5'*)

Write the effective Hamiltonian to a file.

Parameters
: **heff_file** (*str, optional*) – Path to the file to store effective Hamiltonian, by default 'bepotfile.h5'.

Return type
: None

quemb.molbe.mbe.storeBE

class quemb.molbe.mbe.**storeBE**(*Nocc, hf_veff, hcore, S, C, hf_dm, hf_etot, W, lmo_coeff, enuc, ek, E_core, C_core, P_core, core_veff*)

Attributes

Nocc: int

hf_veff: ndarray[tuple[int, ...], dtype[floating]]

hcore: ndarray[tuple[int, ...], dtype[floating]]

S: ndarray[tuple[int, ...], dtype[floating]]

C: ndarray[tuple[int, ...], dtype[floating]]

hf_dm: ndarray[tuple[int, ...], dtype[floating]]

hf_etot: float

W: ndarray[tuple[int, ...], dtype[floating]]

lmo_coeff: ndarray[tuple[int, ...], dtype[floating]]

enuc: float

ek: float

E_core: float

C_core: ndarray[tuple[int, ...], dtype[floating]]

P_core: ndarray[tuple[int, ...], dtype[floating]]

core_veff: ndarray[tuple[int, ...], dtype[floating]]

Methods

__init__(Nocc, hf_veff, hcore, S, C, hf_dm, ...)	Method generated by attrs for class storeBE.

quemb.molbe.mbe.storeBE.__init__

storeBE.**__init__**(*Nocc, hf_veff, hcore, S, C, hf_dm, hf_etot, W, lmo_coeff, enuc, ek, E_core, C_core, P_core, core_veff*)

Method generated by attrs for class storeBE.

quemb.molbe.mf_interfaces

Modules

main
orca_interface
pyscf_interface

quemb.molbe.mf_interfaces.main

Functions

dump_scf(mf, chkfile)	Store a PySCF RHF object to an HDF5 file.
get_mf(mol, *[, work_dir, backend, ...])	Compute the mean-field (SCF) object for a given molecule using the selected backend.
load_scf(chkfile)	Recreate a PySCF Mole and RHF object from an HDF5 file.

quemb.molbe.mf_interfaces.main.dump_scf

`quemb.molbe.mf_interfaces.main.`**`dump_scf`**`(`*mf*, *chkfile*`)`

Store a PySCF RHF object to an HDF5 file.

Return type

`None`

quemb.molbe.mf_interfaces.main.get_mf

`quemb.molbe.mf_interfaces.main.`**`get_mf`**`(`*mol*, ***, *work_dir=None*, *backend='pyscf'*, *additional_args=None*`)`

Compute the mean-field (SCF) object for a given molecule using the selected backend.

Supports multiple SCF backends, including PySCF and ORCA, with optional RIJCOSX acceleration for large systems. The ORCA runs are isolated in a working directory, which can be provided or inferred from the environment.

Parameters

- **mol** (`Mole`) – The molecule to perform the SCF calculation on.
- **work_dir** (*`WorkDir`* | `None`) – Working directory for external backend calculations (e.g., ORCA). If None, a directory is created based on the environment.
- **backend** (`Literal['pyscf', 'orca']`) – The SCF backend to use: “pyscf”, “orca”, or “orca-RIJCOSX”.

Note

Using any of the ORCA options requires `orca` (version >= 6.1) in your path and the ORCA python interface (OPI) to be installed.

- **additional_args** (*OrcaArgs* | None) – Pass on additional arguments to the respective backends.

Return type
RHF

Returns
The resulting mean-field (RHF) object from the selected backend.

quemb.molbe.mf_interfaces.main.load_scf

quemb.molbe.mf_interfaces.main.**load_scf**(*chkfile*)

Recreate a PySCF Mole and RHF object from an HDF5 file.

Return type
tuple[Mole, RHF]

quemb.molbe.mf_interfaces.orca_interface

Functions

get_mf_orca(mol, work_dir, n_procs, ...)	
get_orca_basis(mol)	

quemb.molbe.mf_interfaces.orca_interface.get_mf_orca

quemb.molbe.mf_interfaces.orca_interface.**get_mf_orca**(*mol*, *work_dir*, *n_procs*, *simple_keywords*, *blocks*)

Return type
RHF

quemb.molbe.mf_interfaces.orca_interface.get_orca_basis

quemb.molbe.mf_interfaces.orca_interface.**get_orca_basis**(*mol*)

Return type
SimpleKeyword

Classes

OrcaArgs(*[, n_procs])	Use to pass information to ORCA.

quemb.molbe.mf_interfaces.orca_interface.OrcaArgs

class quemb.molbe.mf_interfaces.orca_interface.**OrcaArgs**(**, n_procs=1*, *simple_keywords*, *blocks*)

Use to pass information to ORCA. Follows the ORCA python interface. (https://www.faccts.de/docs/opi/1.0/docs/index.html)

You can use the *get_orca_basis()* function to translate a pyscf basis label to ORCA A "normal" Hartree Fock calculation can be invoked via

```
>>> OrcaArgs(
>>>     simple_keywords=[],
>>>     blocks=[BlockBasis(basis=get_orca_basis(mol))],
>>> )
```

While a RIJK calculation with parallelisation would be:

```
>>> from opi.input.blocks.block_basis import BlockBasis
>>> from opi.input.simple_keywords import Approximation, SimpleKeyword
>>> OrcaArgs(
>>>     n_procs=4,
>>>     simple_keywords=[Approximation.RIJK],
>>>     blocks=[
>>>         BlockBasis(basis=get_orca_basis(mol), auxjk=SimpleKeyword("def2/jk"))
>>>     ],
>>> ),
```

Attributes

n_procs: Final[int]

simple_keywords: Final[Sequence[SimpleKeyword]]

blocks: Final[Sequence[Block]]

Methods

__init__(*[, n_procs])	Method generated by attrs for class OrcaArgs.

quemb.molbe.mf_interfaces.orca_interface.OrcaArgs.__init__

OrcaArgs.**__init__**(*, *n_procs=1*, *simple_keywords*, *blocks*)

Method generated by attrs for class OrcaArgs.

quemb.molbe.mf_interfaces.pyscf_interface

Functions

create_mf(mol, mo_coeff, mo_energy, mo_occ, ...)	Create a pyscf mean-field object from data **without** running a calculation
get_mf_pyscf(mol)	Run an RHF calculation in pyscf

quemb.molbe.mf_interfaces.pyscf_interface.create_mf

quemb.molbe.mf_interfaces.pyscf_interface.**create_mf**(*mol*, *mo_coeff*, *mo_energy*, *mo_occ*, *e_tot*)

Create a pyscf mean-field object from data **without** running a calculation

Return type

RHF

quemb.molbe.mf_interfaces.pyscf_interface.get_mf_pyscf

`quemb.molbe.mf_interfaces.pyscf_interface.`**`get_mf_pyscf`**`(`*mol*`)`

Run an RHF calculation in pyscf

Return type

`RHF`

quemb.molbe.misc

Functions

be2fcidump(be_obj, fcidump_prefix, basis)	Construct FCIDUMP file for each fragment in a given BE object Assumes molecular, restricted BE calculation
be2puffin(xyzfile, basis[, hcore, ...])	Front-facing API bridge tailored for SCINE Puffin
libint2pyscf(xyzfile, hcore, basis[, ...])	Build a pyscf Mole and RHF/UHF object using the given xyz file and core Hamiltonian (in libint standard format) c.f.
print_energy_cumulant(ecorr, e_V_Kapprox, ...)	
print_energy_noncumulant(be_tot, e1, ec, e2, ...)	
ube2fcidump(be_obj, fcidump_prefix, basis)	Construct FCIDUMP file for each fragment in a given BE object Assumes molecular, restricted BE calculation

quemb.molbe.misc.be2fcidump

`quemb.molbe.misc.`**`be2fcidump`**`(`*be_obj*, *fcidump_prefix*, *basis*`)`

Construct FCIDUMP file for each fragment in a given BE object Assumes molecular, restricted BE calculation

Parameters

- **`be_obj`** (`BE`) – BE object
- **`fcidump_prefix`** (*`str`*) – Prefix for path & filename to the output fcidump files Each file is named [fcidump_prefix]_f0, . . .
- **`basis`** (*`str`*) – ‘embedding’ to get the integrals in the embedding basis ‘fragment_mo’ to get the integrals in the fragment MO basis

quemb.molbe.misc.be2puffin

`quemb.molbe.misc.`**`be2puffin`**`(`*xyzfile*, *basis*, *hcore=None*, *libint_inp=False*, *pts_and_charges=None*, *jk=None*, *use_df=False*, *charge=0*, *spin=0*, *nproc=1*, *ompnum=1*, *n_BE=1*, *df_aux_basis=None*, *frozen_core=True*, *localization_method='lowdin'*, *unrestricted=False*, *from_chk=False*, *checkfile=None*, *ecp=None*, *frag_type='chemgen'*`)`

Front-facing API bridge tailored for SCINE Puffin

Returns the CCSD oneshot energies - QM/MM notes: Using QM/MM alongside big basis sets, especially with a frozen core, can cause localization and numerical stability problems. Use with caution. Additional work to this end on localization, frozen core, ECPs, and QM/MM in this capacity is ongoing. - If running unrestricted QM/MM calculations, with ECPs, in a large basis set, do not freeze the core. Using an ECP for heavy atoms improves the localization numerics, but this is not yet compatible with frozen core on the rest of the atoms.

Parameters

- **xyzfile** (*str*) – Path to the xyz file
- **basis** (*str*) – Name of the basis set
- **hcore** (*ndarray*) – Two-dimensional array of the core Hamiltonian
- **libint_inp** (*bool*) – True for hcore provided in Libint format. Else, hcore input is in PySCF format Default is False, i.e., hcore input is in PySCF format
- **pts_and_charges** (*tuple of ndarray*) – QM/MM (points, charges). Use pyscf's QM/MM instead of starting Hamiltonian
- **jk** (*ndarray*) – Coulomb and Exchange matrices (pyscf will calculate this if not given)
- **use_df** (*bool, optional*) – If true, use density-fitting to evaluate the two-electron integrals
- **charge** (*int, optional*) – Total charge of the system
- **spin** (*int, optional*) – Total spin of the system, pyscf definition
- **nproc** (*int, optional*)
- **ompnum** (*int, optional*) – Set number of processors and ompnum for the jobs
- **frozen_core** (*bool, optional*) – Whether frozen core approximation is used or not, by default True
- **localization_method** (*str, optional*) – For now, lowdin is best supported for all cases. IAOs to be expanded By default 'lowdin'
- **unrestricted** (*bool, optional*) – Unrestricted vs restricted HF and CCSD, by default False
- **from_chk** (*bool, optional*) – Run calculation from converged RHF/UHF checkpoint. By default False
- **checkfile** (*str, optional*) – if not None: - if from_chk: specify the checkfile to run the embedding calculation - if not from_chk: specify where to save the checkfile By default None
- **ecp** (*str, optional*) – specify the ECP for any atoms, accompanying the basis set syntax; for example `{'Na': 'bfd-pp', 'Ru': 'bfd-pp'}` By default None

quemb.molbe.misc.libint2pyscf

quemb.molbe.misc.**libint2pyscf**(*xyzfile*, *hcore*, *basis*, *hcore_skiprows=1*, *use_df=False*, *unrestricted=False*, *spin=0*, *charge=0*)

Build a pyscf Mole and RHF/UHF object using the given xyz file and core Hamiltonian (in libint standard format) c.f. In libint standard format, the basis sets appear in the order atom# n l m 0 1 0 0 1s 0 2 0 0 2s 0 2 1 -1 2py 0 2 1 0 2pz 0 2 1 1 2px … In pyscf, the basis sets appear in the order atom # n l m 0 1 0 0 1s 0 2 0 0 2s 0 2 1 1 2px 0 2 1 -1 2py 0 2 1 0 2pz … For higher angular momentum, both use [-l, -l+1, …, l-1, l] ordering.

Parameters

- **xyzfile** (*str*) – Path to the xyz file
- **hcore** (*str*) – Path to the core Hamiltonian
- **basis** (*str*) – Name of the basis set
- **hcore_skiprows** (*int, optional*) – # of first rows to skip from the core Hamiltonian file, by default 1

- **use_df** (*bool, optional*) – If true, use density-fitting to evaluate the two-electron integrals
- **unrestricted** (*bool, optional*) – If true, use UHF bath
- **spin** (*int, optional*) – 2S, Difference between the number of alpha and beta electrons
- **charge** (*int, optional*) – Total charge of the system

Return type

(*Mole*, *RHF*, or *UHF*)

quemb.molbe.misc.print_energy_cumulant

quemb.molbe.misc.**print_energy_cumulant**(*ecorr*, *e_V_Kapprox*, *e_F_dg*, *e_hf*)

quemb.molbe.misc.print_energy_noncumulant

quemb.molbe.misc.**print_energy_noncumulant**(*be_tot*, *e1*, *ec*, *e2*, *e_hf*, *e_nuc*)

quemb.molbe.misc.ube2fcidump

quemb.molbe.misc.**ube2fcidump**(*be_obj*, *fcidump_prefix*, *basis*)

Construct FCIDUMP file for each fragment in a given BE object Assumes molecular, restricted BE calculation

Parameters

- **be_obj** (BE) – BE object
- **fcidump_prefix** (*str*) – Prefix for path & filename to the output fcidump files Each file is named [fcidump_prefix]_f0, …
- **basis** (*str*) – 'embedding' to get the integrals in the embedding basis 'fragment_mo' to get the integrals in the fragment MO basis

quemb.molbe.opt

Classes

BEOPT(pot, Fobjs, Nocc, enuc, scratch_dir[, ...])	Perform BE optimization.

quemb.molbe.opt.BEOPT

class quemb.molbe.opt.**BEOPT**(*pot*, *Fobjs*, *Nocc*, *enuc*, *scratch_dir*, *solver='CCSD'*, *nproc=1*, *ompnum=4*, *only_chem=False*, *use_cumulant=True*, *max_space=500*, *conv_tol=1e-06*, *relax_density=False*, *ebe_hf=0.0*, *iter=0*, *err=0.0*, *Ebe=NOTHING*, *solver_args=None*)

Perform BE optimization.

Implements optimization algorithms for bootstrap optimizations, namely, chemical potential optimization and density matching. The main technique used in the optimization is a Quasi-Newton method. It interface to external (adapted version) module originally written by Hong-Zhou Ye.

Parameters

- **pot** (list[float]) – List of initial BE potentials. The last element is for the global chemical potential.

- **Fobjs** (list[*Frags*] | list[*Frags*]) – Fragment object
- **Nocc** (int) – No. of occupied orbitals for the full system.
- **enuc** (float) – Nuclear component of the energy.
- **scratch_dir** (*WorkDir*) – Scratch directory
- **solver** (Literal['MP2', 'CCSD', 'FCI', 'HCI', 'SHCI', 'SCI', 'DMRG']) – High-level solver in bootstrap embedding. 'MP2', 'CCSD', 'FCI' are supported. Selected CI versions, 'HCI', 'SHCI', & 'SCI' are also supported. Defaults to 'CCSD'
- **only_chem** (bool) – Whether to perform chemical potential optimization only. Refer to bootstrap embedding literatures.
- **nproc** (int) – Total number of processors assigned for the optimization. Defaults to 1. When nproc > 1, Python multithreading is invoked.
- **ompnum** (int) – If nproc > 1, ompnum sets the number of cores for OpenMP parallelization. Defaults to 4
- **max_space** (int) – Maximum number of bootstrap optimizaiton steps, after which the optimization is called converged.
- **conv_tol** (float) – Convergence criteria for optimization. Defaults to 1e-6
- **ebe_hf** (float) – Hartree-Fock energy. Defaults to 0.0

Attributes

pot: list[float]

Fobjs: list[*Frags*] | list[*Frags*]

Nocc: int

enuc: float

scratch_dir: *WorkDir*

solver: Literal['MP2', 'CCSD', 'FCI', 'HCI', 'SHCI', 'SCI', 'DMRG']

nproc: int

ompnum: int

only_chem: bool

use_cumulant: bool

max_space: int

conv_tol: float

relax_density: bool

ebe_hf: float

iter: int

err: float

Ebe: `ndarray[tuple[int, ...], dtype[float64]]`

solver_args: *`UserSolverArgs`* `| None`

Methods

`__init__`(pot, Fobjs, Nocc, enuc, scratch_dir)	Method generated by attrs for class BEOPT.
`objfunc`(xk)	Computes error vectors, RMS error, and BE energies.
`optimize`(method[, J0, trust_region])	Main kernel to perform BE optimization

quemb.molbe.opt.BEOPT.__init__

`BEOPT.`**`__init__`**(*pot*, *Fobjs*, *Nocc*, *enuc*, *scratch_dir*, *solver='CCSD'*, *nproc=1*, *ompnum=4*, *only_chem=False*, *use_cumulant=True*, *max_space=500*, *conv_tol=1e-06*, *relax_density=False*, *ebe_hf=0.0*, *iter=0*, *err=0.0*, *Ebe=NOTHING*, *solver_args=None*)

Method generated by attrs for class BEOPT.

quemb.molbe.opt.BEOPT.objfunc

`BEOPT.`**`objfunc`**(*xk*)

Computes error vectors, RMS error, and BE energies.

If nproc (set in initialization) > 1, a multithreaded function is called to perform high-level computations.

Parameters

`xk` (`list[float]`) – Current potentials in the BE optimization.

Returns

Error vectors.

Return type

list

quemb.molbe.opt.BEOPT.optimize

`BEOPT.`**`optimize`**(*method*, *J0=None*, *trust_region=False*)

Main kernel to perform BE optimization

Parameters

- **`method`** (*`str`*) – High-level quantum chemistry method.
- **`J0`** (*`list of list of float, optional`*) – Initial Jacobian
- **`trust_region`** (*`bool, optional`*) – Use trust-region based QN optimization, by default False

quemb.molbe.pfrag

Functions

`schmidt_decomposition`(mo_coeff, nocc, AO_in_frag)	Perform Schmidt decomposition on the molecular orbital coefficients.

continues on next page

Table 41 – continued from previous page

union_of_frag_MOs_and_index(Fobjs, S[, epsilon])	Get the union of all fragment MOs as one Matrix and the respective indices for each fragment to refer to the global fragment MO matrix.

quemb.molbe.pfrag.schmidt_decomposition

quemb.molbe.pfrag.**schmidt_decomposition**(*mo_coeff*, *nocc*, *AO_in_frag*, *thr_bath=1e-10*, *cinv=None*, *rdm=None*, *norb=None*)

Perform Schmidt decomposition on the molecular orbital coefficients.

This function decomposes the molecular orbitals into fragment and environment parts using the Schmidt decomposition method. It computes the transformation matrix (TA) which includes both the fragment orbitals and the entangled bath.

Parameters

- **mo_coeff** (ndarray[tuple[int, ...], dtype[float64]]) – Molecular orbital coefficients.
- **nocc** (int) – Number of occupied orbitals.
- **Frag_sites** (*list of int*) – List of fragment sites (indices).
- **thr_bath** (float) – Threshold for bath orbitals in Schmidt decomposition
- **cinv** (ndarray[tuple[int, ...], dtype[float64]] | None) – Inverse of the transformation matrix. Defaults to None.
- **rdm** (ndarray[tuple[int, ...], dtype[float64]] | None) – Reduced density matrix. If not provided, it will be computed from the molecular orbitals. Defaults to None.
- **norb** (int | None) – Specifies number of bath orbitals. Used for UBE to make alpha and beta spaces the same size. Defaults to None

Returns

TA, norbs_frag, norbs_bath

Transformation matrix (TA) including both fragment and entangled bath orbitals.

Return type

tuple

quemb.molbe.pfrag.union_of_frag_MOs_and_index

quemb.molbe.pfrag.**union_of_frag_MOs_and_index**(*Fobjs*, *S*, *epsilon=1e-10*)

Get the union of all fragment MOs as one Matrix and the respective indices for each fragment to refer to the global fragment MO matrix.

This allows to reuse information such as integrals for the fragment MOs.

Parameters

- **Fobjs** (Sequence[*Frags*]) – A sequence of Frags.
- **S** (ndarray[tuple[int, ...], dtype[float64]]) – The AO overlap matrix.
- **epsilon** (float) – Cutoff to consider overlap values to be zero or one.

Return type

tuple[ndarray[tuple[int, ...], dtype[float64]], list[ndarray[tuple[int], dtype[int64]]]]

Classes

Frags(AO_in_frag, ifrag, AO_per_edge, ...[, ...])	Class for handling fragments in bootstrap embedding.

quemb.molbe.pfrag.Frags

class quemb.molbe.pfrag.**Frags**(*AO_in_frag*, *ifrag*, *AO_per_edge*, *ref_frag_idx_per_edge*, *relAO_per_edge*, *relAO_in_ref_per_edge*, *weight_and_relAO_per_center*, *relAO_per_origin*, *eri_file='eri_file.h5'*, *unrestricted=False*)

Class for handling fragments in bootstrap embedding.

This class contains various functionalities required for managing and manipulating fragments for BE calculations.

Methods

__init__(AO_in_frag, ifrag, AO_per_edge, ...)	Constructor function for `Frags` class.
cons_fock(hf_veff, S, dm[, eri_])	Construct the Fock matrix for the fragment.
get_nsocc(S, C, nocc[, ncore])	Get the number of occupied orbitals for the fragment.
scf([heff, fs, eri, dm0, unrestricted, spin_ind])	Perform self-consistent field (SCF) calculation for the fragment.
sd(lao, lmo, nocc, thr_bath[, norb])	Perform Schmidt decomposition for the fragment.
set_udim(cout)	
update_ebe_hf([rdm_hf, mo_coeffs, eri, ...])	
update_heff(u[, cout, only_chem])	Update the effective Hamiltonian for the fragment.

quemb.molbe.pfrag.Frags.__init__

Frags.**__init__**(*AO_in_frag*, *ifrag*, *AO_per_edge*, *ref_frag_idx_per_edge*, *relAO_per_edge*, *relAO_in_ref_per_edge*, *weight_and_relAO_per_center*, *relAO_per_origin*, *eri_file='eri_file.h5'*, *unrestricted=False*)

Constructor function for `Frags` class.

Parameters

- **AO_in_frag** (Sequence[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]) – list of AOs in the fragment (i.e. `BE.AO_per_frag[i]` or `FragPart.AO_per_frag[i]`)
- **ifrag** (int) – fragment index ($\in [0, \text{BE.n\_frag} - 1]$)
- **AO_per_edge** (Sequence[Sequence[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]) – list of lists of edge site AOs for each atom in the fragment. Read more detailed description in *quemb.molbe.autofrag.FragPart*.
- **ref_frag_idx_per_edge** (Sequence[NewType(FragmentIdx, integer)]) – list of fragment indices where edge site AOs are center site. Read more detailed description in *quemb.molbe.autofrag.FragPart*.
- **relAO_per_edge** (Sequence[Sequence[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]) – list of lists of indices for edge site AOs within the fragment. Read more detailed description in *quemb.molbe.autofrag.FragPart*.

- **relAO_in_ref_per_edge** (Sequence[Sequence[NewType(RelAOIdxInRef, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]) – list of lists of indices within the fragment specified in `center` that points to the edge site AOs. Read more detailed description in *quemb.molbe.autofrag.FragPart*.
- **weight_and_relAO_per_center** (tuple[float, Sequence[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]) – weight used for energy contributions and the indices. Read more detailed description in *quemb.molbe.autofrag.FragPart*.
- **relAO_per_origin** (Sequence[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]) – indices of the origin site atoms in the fragment Read more detailed description in *quemb.molbe.autofrag.FragPart*.
- **eri_file** (str | PathLike) – two-electron integrals stored as h5py file, by default 'eri_file.h5'
- **unrestricted** (bool) – unrestricted calculation, by default False

quemb.molbe.pfrag.Frags.cons_fock

Frags.**cons_fock**(*hf_veff*, *S*, *dm*, *eri_=None*)

Construct the Fock matrix for the fragment.

Parameters

- **hf_veff** (*ndarray*) – Hartree-Fock effective potential.
- **S** (*ndarray*) – Overlap matrix.
- **dm** (*ndarray*) – Density matrix.
- **eri** (*ndarray, optional*) – Electron repulsion integrals, by default None.

quemb.molbe.pfrag.Frags.get_nsocc

Frags.**get_nsocc**(*S*, *C*, *nocc*, *ncore=0*)

Get the number of occupied orbitals for the fragment.

Parameters

- **S** (*ndarray*) – Overlap matrix.
- **C** (*ndarray*) – Molecular orbital coefficients.
- **nocc** (*int*) – Number of occupied orbitals.
- **ncore** (*int, optional*) – Number of core orbitals, by default 0.

Returns

Projected density matrix.

Return type

ndarray

quemb.molbe.pfrag.Frags.scf

Frags.**scf**(*heff=None*, *fs=False*, *eri=None*, *dm0=None*, *unrestricted=False*, *spin_ind=None*)

Perform self-consistent field (SCF) calculation for the fragment.

Parameters

- **heff** (*ndarray, optional*) – Effective Hamiltonian, by default None.
- **fs** (*bool, optional*) – Flag for full SCF, by default False.
- **eri** (*ndarray, optional*) – Electron repulsion integrals, by default None.
- **dm0** (*ndarray, optional*) – Initial density matrix, by default None.
- **unrestricted** (*bool, optional*) – Specify if unrestricted calculation, by default False
- **spin_ind** (*int, optional*) – Alpha (0) or beta (1) spin for unrestricted calculation, by default None

quemb.molbe.pfrag.Frags.sd

`Frags.sd`(*lao*, *lmo*, *nocc*, *thr_bath*, *norb=None*)

Perform Schmidt decomposition for the fragment.

Parameters

- **lao** (*ndarray*) – Orthogonalized AOs
- **lmo** (*ndarray*) – Local molecular orbital coefficients.
- **nocc** (*int*) – Number of occupied orbitals.
- **thr_bath** (*float,*) – Threshold for bath orbitals in Schmidt decomposition
- **norb** (*int, optional*) – Specify number of bath orbitals. Used for UBE, where different number of alpha and beta orbitals Default is None, allowing orbitals to be chosen by threshold

Return type

None

quemb.molbe.pfrag.Frags.set_udim

`Frags.set_udim`(*cout*)

quemb.molbe.pfrag.Frags.update_ebe_hf

`Frags.update_ebe_hf`(*rdm_hf=None*, *mo_coeffs=None*, *eri=None*, *return_e=False*, *unrestricted=False*, *spin_ind=None*)

quemb.molbe.pfrag.Frags.update_heff

`Frags.update_heff`(*u*, *cout=None*, *only_chem=False*)

Update the effective Hamiltonian for the fragment.

quemb.molbe.solver

Functions

be_func(pot, Fobjs, Nocc, solver, enuc, ...)	Perform bootstrap embedding calculations for each fragment.
be_func_u(pot, Fobjs, solver, enuc[, ...])	Perform bootstrap embedding calculations for each fragment with UCCSD.

continues on next page

Table 44 – continued from previous page

solve_block2(mf, nocc, frag_scratch, ...)	DMRG fragment solver using the pyscf.dmrgscf wrapper.
solve_ccsd(mf[, frozen, mo_coeff, relax, ...])	Solve the CCSD (Coupled Cluster with Single and Double excitations) equations.
solve_error(Fobjs, Nocc[, only_chem])	Compute the error for self-consistent fragment density matrix matching.
solve_mp2(mf[, frozen, mo_coeff, mo_occ, ...])	Perform an MP2 (2nd order Moller-Plesset perturbation theory) calculation.
solve_uccsd(mf, eris_inp[, frozen, relax, ...])	Solve the U-CCSD (Unrestricted Coupled Cluster with Single and Double excitations) equations.

quemb.molbe.solver.be_func

`quemb.molbe.solver.`**`be_func`**(*pot*, *Fobjs*, *Nocc*, *solver*, *enuc*, *solver_args*, *scratch_dir*, *only_chem=False*, *eeval=False*, *relax_density=False*, *return_vec=False*, *use_cumulant=True*)

Perform bootstrap embedding calculations for each fragment.

This function computes the energy and/or error for each fragment in a molecular system using various quantum chemistry solvers.

Parameters

- **pot** (`list[float]` | `None`) – List of potentials.
- **Fobjs** (*list of* `FragPart`) – List of fragment objects.
- **Nocc** (`int`) – Number of occupied orbitals.
- **solver** (`Literal['MP2', 'CCSD', 'FCI', 'HCI', 'SHCI', 'SCI', 'DMRG']`) – Quantum chemistry solver to use ('MP2', 'CCSD', 'FCI', 'SCI'). TODO 'HCI', 'SHCI'
- **enuc** (`float`) – Nuclear energy.
- **only_chem** (`bool`) – Whether to only optimize the chemical potential. Defaults to False.
- **eeval** (`bool`) – Whether to evaluate the energy. Defaults to False.
- **relax_density** (`bool`) – Whether to use the relaxed CCSD density matrix. Defaults to False.
- **return_vec** (`bool`) – Whether to return the error vector. Defaults to False.
- **use_cumulant** (`bool`) – Whether to use the cumulant-based energy expression. Defaults to True.
- **eeval** – Whether to evaluate the energy. Defaults to False.
- **return_vec** – Whether to return the error vector. Defaults to False.

Returns

Depending on the options, it returns the norm of the error vector, the energy, or a combination of these values.

Return type

float or *tuple*

quemb.molbe.solver.be_func_u

`quemb.molbe.solver.be_func_u`(*pot*, *Fobjs*, *solver*, *enuc*, *hf_veff=None*, *eeval=False*, *relax_density=False*, *use_cumulant=True*, *frozen=False*)

Perform bootstrap embedding calculations for each fragment with UCCSD.

This function computes the energy and/or error for each fragment in a molecular system using various quantum chemistry solvers.

Parameters

- **pot** (*list*) – List of potentials.
- **Fobjs** (*list*) – zip list of `quemb.molbe.autofrag.FragPart`, alpha and beta List of fragment objects. Each element is a tuple with the alpha and beta components
- **solver** (`Literal['UCCSD']`) – Quantum chemistry solver to use ('UCCSD').
- **enuc** (*float*) – Nuclear energy.
- **hf_veff** (*tuple of ndarray, optional*) – Hartree-Fock effective potential. Defaults to None.
- **eeval** (*bool, optional*) – Whether to evaluate the energy. Defaults to False.
- **relax_density** (*bool, optional*) – Whether to relax the density. Defaults to False.
- **return_vec** (*bool, optional*) – Whether to return the error vector. Defaults to False.
- **ebe_hf** (*float, optional*) – Hartree-Fock energy. Defaults to 0.
- **use_cumulant** (*bool, optional*) – Whether to use the cumulant-based energy expression. Defaults to True.
- **frozen** (*bool, optional*) – Frozen core. Defaults to False

Returns

Depending on the options, it returns the norm of the error vector, the energy, or a combination of these values.

Return type

float or *tuple*

quemb.molbe.solver.solve_block2

`quemb.molbe.solver.solve_block2`(*mf*, *nocc*, *frag_scratch*, *DMRG_args*, *use_cumulant*)

DMRG fragment solver using the pyscf.dmrgscf wrapper.

Parameters

- **mf** (`RHF`) – Mean field object or similar following the data signature of the pyscf.RHF class.
- **nocc** (`int`) – Number of occupied MOs in the fragment, used for constructing the fragment 1- and 2-RDMs.
- **frag_scratch** (`WorkDir`) – Fragment-level DMRG scratch directory.
- **use_cumulant** (`bool`) – Use the cumulant energy expression.

Returns

rdm1: numpy.ndarray
1-Particle reduced density matrix for fragment.

rdm2: numpy.ndarray
2-Particle reduced density matrix for fragment.

quemb.molbe.solver.solve_ccsd

`quemb.molbe.solver.`**`solve_ccsd`**(*mf*, *frozen=None*, *mo_coeff=None*, *relax=False*, *use_cumulant=True*, *rdm_return=False*, *rdm2_return=False*, *mo_occ=None*, *mo_energy=None*, *verbose=0*)

Solve the CCSD (Coupled Cluster with Single and Double excitations) equations.

This function sets up and solves the CCSD equations using the provided mean-field object. It can return the CCSD amplitudes (t1, t2), the one- and two-particle density matrices, and the CCSD object.

Parameters

- **mf** (*RHF*) – Mean-field object from PySCF.
- **frozen** (*list or int, optional*) – List of frozen orbitals or number of frozen core orbitals. Defaults to None.
- **mo_coeff** (*ndarray, optional*) – Molecular orbital coefficients. Defaults to None.
- **relax** (*bool, optional*) – Whether to use relaxed density matrices. Defaults to False.
- **use_cumulant** (*bool, optional*) – Whether to use cumulant-based energy expression. When using the cumulant, the one-particle density matrix is not included in the two-particle density matrix calculation (with_dm1 = False). Defaults to True.
- **rdm_return** (*bool, optional*) – Whether to return the one-particle density matrix. Defaults to False.
- **rdm2_return** (*bool, optional*) – Whether to return the two-particle density matrix. Defaults to False.
- **mo_occ** (*ndarray, optional*) – Molecular orbital occupations. Defaults to None.
- **mo_energy** (*ndarray, optional*) – Molecular orbital energies. Defaults to None.
- **verbose** (*int, optional*) – Verbosity level. Defaults to 0.

Returns

- t1 (numpy.ndarray): Single excitation amplitudes.
- t2 (numpy.ndarray): Double excitation amplitudes.
- **rdm1a (numpy.ndarray, optional): One-particle density matrix**
  (if rdm_return is True).
- **rdm2s (numpy.ndarray, optional): Two-particle density matrix**
  (if rdm2_return is True and rdm_return is True).
- **mycc (pyscf.cc.ccsd.CCSD, optional): CCSD object**
  (if rdm_return is True and rdm2_return is False).

Return type
tuple

quemb.molbe.solver.solve_error

`quemb.molbe.solver.solve_error`(*Fobjs*, *Nocc*, *only_chem=False*)

Compute the error for self-consistent fragment density matrix matching.

This function calculates the error in the one-particle density matrix for a given fragment, matching the density matrix elements of the edges and centers. It returns the norm of the error vector and the error vector itself.

Parameters

- **Fobjs** (*list of FragPart*) – List of fragment objects.
- **Nocc** (*int*) – Number of occupied orbitals.

Returns

- *float* – Norm of the error vector.
- *numpy.ndarray* – Error vector.

quemb.molbe.solver.solve_mp2

`quemb.molbe.solver.solve_mp2`(*mf*, *frozen=None*, *mo_coeff=None*, *mo_occ=None*, *mo_energy=None*)

Perform an MP2 (2nd order Moller-Plesset perturbation theory) calculation.

This function sets up and runs an MP2 calculation using the provided mean-field object. It returns the MP2 object after the calculation.

Parameters

- **mf** (RHF) – Mean-field object from PySCF.
- **frozen** (int | list[int] | None) – List of frozen orbitals or number of frozen core orbitals. Defaults to None.
- **mo_coeff** (ndarray[tuple[int, ...], dtype[floating]] | None) – Molecular orbital coefficients. Defaults to None.
- **mo_occ** (ndarray[tuple[int], dtype[floating]] | None) – Molecular orbital occupations. Defaults to None.
- **mo_energy** (ndarray[tuple[int], dtype[floating]] | None) – Molecular orbital energies. Defaults to None.

Return type

RMP2

Returns

The MP2 object after running the calculation.

quemb.molbe.solver.solve_uccsd

`quemb.molbe.solver.solve_uccsd`(*mf*, *eris_inp*, *frozen=None*, *relax=False*, *use_cumulant=True*, *rdm_return=False*, *rdm2_return=False*, *verbose=0*)

Solve the U-CCSD (Unrestricted Coupled Cluster with Single and Double excitations) equations.

This function sets up and solves the UCCSD equations using the provided mean-field object. It can return the one- and two-particle density matrices and the UCCSD object.

Parameters

- **mf** (*UHF*) – Mean-field object from PySCF. Constructed with make_uhf_obj

- **eris_inp** – Custom fragment ERIs object
- **frozen** (*list or int, optional*) – List of frozen orbitals or number of frozen core orbitals. Defaults to None.
- **relax** (*bool, optional*) – Whether to use relaxed density matrices. Defaults to False.
- **use_cumulant** (*bool, optional*) – Whether to use cumulant-based energy expression. Defaults to True.
- **rdm_return** (*bool, optional*) – Whether to return the one-particle density matrix. Defaults to False.
- **rdm2_return** (*bool, optional*) – Whether to return the two-particle density matrix. Defaults to False.
- **verbose** (*int, optional*) – Verbosity level. Defaults to 0.

Returns

- ucc (pyscf.cc.ccsd.UCCSD): UCCSD object
- **rdm1 (tuple, numpy.ndarray, optional): One-particle density matrix**
  (if rdm_return is True).
- **rdm2 (tuple, numpy.ndarray, optional): Two-particle density matrix**
  (if rdm2_return is True and rdm_return is True).

Return type

tuple

Classes

DMRG_ArgsUser([norb, nelec, startM, maxM, ...])	
SHCI_ArgsUser([hci_cutoff, hci_pt, ...])	
UserSolverArgs()	

quemb.molbe.solver.DMRG_ArgsUser

class quemb.molbe.solver.**DMRG_ArgsUser**(*norb=None, nelec=None, startM=25, maxM=500, max_iter=60, max_mem=100, max_noise=0.001, min_tol=1e-08, twodot_to_onedot=50, root=0, block_extra_keyword=NOTHING, schedule_kwargs=NOTHING, force_cleanup=False*)

Parameters

- **max_mem** (Final[int]) – Maximum memory in GB.
- **root** (Final[int]) – Number of roots to solve for.
- **startM** (Final[int]) – Starting MPS bond dimension - where the sweep schedule begins.
- **maxM** (Final[int]) – Maximum MPS bond dimension - where the sweep schedule terminates.
- **max_iter** (Final[int]) – Maximum number of sweeps.

- **twodot_to_onedot** (Final[int]) – Sweep index at which to transition to one-dot DMRG algorithm. All sweeps prior to this will use the two-dot algorithm.
- **block_extra_keyword** (Final[list[str]]) – Other keywords to be passed to block2. See: https://block2.readthedocs.io/en/latest/user/keywords.html
- **schedule_kwargs** (dict[str, list[int] | list[float]]) – Dictionary containing DMRG scheduling parameters to be passed to block2.

  e.g. The default schedule used here would be equivalent to the following:

  ```
  schedule_kwargs = {
      'scheduleSweeps': [0, 10, 20, 30, 40, 50],
      'scheduleMaxMs': [25, 50, 100, 200, 500, 500],
      'scheduleTols': [1e-5,1e-5, 1e-6, 1e-6, 1e-8, 1e-8],
      'scheduleNoises': [0.01, 0.01, 0.001, 0.001, 1e-4, 0.0],
  }
  ```

Attributes

norb: Final[int | None]

Becomes mf.mo_coeff.shape[1] by default

nelec: Final[int | None]

Becomes mf.mo_coeff.shape[1] by default

startM: Final[int]

maxM: Final[int]

max_iter: Final[int]

max_mem: Final[int]

max_noise: Final[float]

min_tol: Final[float]

twodot_to_onedot: Final[int]

root: Final[int]

block_extra_keyword: Final[list[str]]

schedule_kwargs: dict[str, list[int] | list[float]]

force_cleanup: Final[bool]

Methods

__init__([norb, nelec, startM, maxM, ...])	Method generated by attrs for class DMRG_ArgsUser.

quemb.molbe.solver.DMRG_ArgsUser.__init__

`DMRG_ArgsUser.__init__`(*norb=None*, *nelec=None*, *startM=25*, *maxM=500*, *max_iter=60*, *max_mem=100*, *max_noise=0.001*, *min_tol=1e-08*, *twodot_to_onedot=50*, *root=0*, *block_extra_keyword=NOTHING*, *schedule_kwargs=NOTHING*, *force_cleanup=False*)

Method generated by attrs for class DMRG_ArgsUser.

quemb.molbe.solver.SHCI_ArgsUser

`class quemb.molbe.solver.SHCI_ArgsUser`(*hci_cutoff=0.001*, *hci_pt=False*, *return_frag_data=False*)

Attributes

`hci_cutoff: Final[float]`

`hci_pt: Final[bool]`

`return_frag_data: Final[bool]`

Methods

__init__([hci_cutoff, hci_pt, return_frag_data])	Method generated by attrs for class SHCI_ArgsUser.

quemb.molbe.solver.SHCI_ArgsUser.__init__

`SHCI_ArgsUser.__init__`(*hci_cutoff=0.001*, *hci_pt=False*, *return_frag_data=False*)

Method generated by attrs for class SHCI_ArgsUser.

quemb.molbe.solver.UserSolverArgs

`class quemb.molbe.solver.UserSolverArgs`

Methods

__init__()

quemb.molbe.solver.UserSolverArgs.__init__

`UserSolverArgs.__init__()`

quemb.molbe.ube

Bootstrap Embedding Calculation with an Unrestricted Hartree-Fock Bath

Reference

Tran, H.; Ye, H.; Van Voorhis, T. J. Chem. Phys. 153, 214101 (2020)

TODO

Add iterative UBE

Functions

initialize_pot(n_frag, relAO_per_edge)	

quemb.molbe.ube.initialize_pot

`quemb.molbe.ube.`**`initialize_pot`**(*n_frag*, *relAO_per_edge*)

Classes

UBE(mf, fobj[, scratch_dir, eri_file, ...])	

quemb.molbe.ube.UBE

`class` `quemb.molbe.ube.`**`UBE`**(*mf*, *fobj*, *scratch_dir=None*, *eri_file='eri_file.h5'*, *lo_method='lowdin'*, *pop_method=None*, *compute_hf=True*, *thr_bath=1e-10*)

Methods

__init__(mf, fobj[, scratch_dir, eri_file, ...])	Initialize Unrestricted BE Object (ube🍠)
compute_energy_full([approx_cumulant, ...])	Compute the total energy using rdms in the full basis.
get_be_error_jacobian([jac_solver])	
initialize(eri_, compute_hf)	Initialize the Bootstrap Embedding calculation.
localize(lo_method, fobj[, iao_loc_method, ...])	Molecular orbital localization
oneshot([solver, nproc, ompnum])	Perform a one-shot bootstrap embedding calculation.
optimize([solver, method, only_chem, ...])	BE optimization function
print_ini()	Print initialization banner for the MOLBE calculation.
rdm1_fullbasis([return_ao, only_rdm1, ...])	Compute the one- and two-particle reduced density matrices (RDM1 and RDM2).
read_heff([heff_file])	Read the effective Hamiltonian from a file.
save([save_file])	Save intermediate results for restart.
update_fock([heff])	Update the Fock matrix for each fragment with the effective Hamiltonian.
write_heff([heff_file])	Write the effective Hamiltonian to a file.

quemb.molbe.ube.UBE.__init__

`UBE.`**`__init__`**(*mf*, *fobj*, *scratch_dir=None*, *eri_file='eri_file.h5'*, *lo_method='lowdin'*, *pop_method=None*, *compute_hf=True*, *thr_bath=1e-10*)

Initialize Unrestricted BE Object (ube🍠)

Note

Currently only supports embedding Hamiltonian construction for molecular systems In conjunction with molbe.misc.ube2fcidump, embedding Hamiltonians can be written for external use. See `unrestricted` branch for a work-in-progress full implmentation

Parameters

- **mf** (UHF) – pyscf meanfield UHF object
- **fobj** (*FragPart*) – object that contains fragment information
- **eri_file** (str | PathLike) – h5py file with ERIs
- **lo_method** (Literal['lowdin', 'boys', 'ER', 'PM', 'IAO']) – Method for orbital localization. Supports "lowdin" (Löwdin or symmetric orthogonalization), "boys" (Foster-Boys), "PM" (Pipek-Mezey", and "ER" (Edmiston-Rudenberg). By default "lowdin"
- **pop_method** (str | None) – Method for calculating orbital population, by default 'meta-lowdin' See pyscf.lo for more details and options
- **thr_bath** (*float*,) – Threshold for bath orbitals in Schmidt decomposition

quemb.molbe.ube.UBE.compute_energy_full

UBE.**compute_energy_full**(*approx_cumulant=False*, *use_full_rdm=False*, *return_rdm=True*)

Compute the total energy using rdms in the full basis.

Parameters

- **approx_cumulant** (*bool, optional*) – If True, use an approximate cumulant for the energy computation. Default is False.
- **use_full_rdm** (*bool, optional*) – If True, use the full two-particle RDM for energy computation. Default is False.
- **return_rdm** (*bool, optional*) – If True, return the computed reduced density matrices (RDMs). Default is True.

Returns

If `return_rdm` is True, returns a tuple containing the one-particle and two-particle reduced density matrices (RDM1 and RDM2). Otherwise, returns None.

Return type

tuple of *ndarray* or *None*

Notes

This function computes the total energy in the full basis, with options to use approximate or true cumulants, and to return the reduced density matrices (RDMs). The energy components are printed as part of the function's output.

quemb.molbe.ube.UBE.get_be_error_jacobian

UBE.**get_be_error_jacobian**(*jac_solver='HF'*)

Return type

ndarray[tuple[int, ...], dtype[floating]]

quemb.molbe.ube.UBE.initialize

UBE.**initialize**(*eri_*, *compute_hf*)

Initialize the Bootstrap Embedding calculation.

Parameters

- **eri** (*ndarray*) – Electron repulsion integrals.
- **restart** (*bool, optional*) – Whether to restart from a previous calculation, by default False.
- **int_transfrom** – Which integral transformation to perform.

quemb.molbe.ube.UBE.localize

UBE.**localize**(*lo_method*, *fobj*, *iao_loc_method='lowdin'*, *iao_valence_only=False*, *pop_method=None*, *init_guess=None*, *hstack=False*, *save=True*)

Molecular orbital localization

Performs molecular orbital localization computations. For large basis, IAO is recommended augmented with PAO orbitals.

NOTE: For molecular systems, with frozen core, the core and valence are localized TOGETHER. This is not the case for periodic systems.

Parameters

- **lo_method** (Literal['lowdin', 'boys', 'ER', 'PM', 'IAO']) – Method for orbital localization. Supports "lowdin" (Löwdin or symmetric orthogonalization), "boys" (Foster-Boys), "PM" (Pipek-Mezey", and "ER" (Edmiston-Rudenberg). By default "lowdin"
- **fobj** (*FragPart*)
- **iao_loc_method** (Literal['lowdin', 'boys', 'PM', 'ER']) – Name of localization method in quantum chemistry for the IAOs and PAOs. Options include "lowdin", "boys", 'PM', 'ER' (as documented in PySCF). Default is "lowdin". If not using lowdin, we suggest using 'PM', as it is more robust than 'boys' localization and less expensive than 'ER'
- **iao_valence_only** (*bool*) – If this option is set to True, all calculation will be performed in the valence basis in the IAO partitioning. Default is False. This is an experimental feature: the returned energy is not accurate

Return type

None | ndarray[tuple[int, ...], dtype[float64]]

quemb.molbe.ube.UBE.oneshot

UBE.**oneshot**(*solver='UCCSD'*, *nproc=1*, *ompnum=4*)

Perform a one-shot bootstrap embedding calculation.

Parameters

- **solver** – High-level quantum chemistry method, by default 'CCSD'. 'CCSD', 'FCI', and variants of selected CI are supported.
- **use_cumulant** – Whether to use the cumulant energy expression, by default True.
- **nproc** – Number of processors for parallel calculations, by default 1. If set to >1, multi-threaded parallel computation is invoked.
- **ompnum** – Number of OpenMP threads, by default 4.

quemb.molbe.ube.UBE.optimize

UBE.**optimize**(*solver='CCSD'*, *method='QN'*, *only_chem=False*, *use_cumulant=True*, *conv_tol=1e-06*, *relax_density=False*, *jac_solver='HF'*, *nproc=1*, *ompnum=4*, *max_iter=500*, *trust_region=False*, *solver_args=None*)

BE optimization function

Interfaces BEOPT to perform bootstrap embedding optimization.

Parameters

- **solver** (Literal['MP2', 'CCSD', 'FCI', 'HCI', 'SHCI', 'SCI', 'DMRG']) – High-level solver for the fragment, by default "CCSD"
- **method** (str) – Optimization method, by default 'QN'
- **only_chem** (bool) – If true, density matching is not performed – only global chemical potential is optimized, by default False
- **use_cumulant** (bool) – Whether to use the cumulant energy expression, by default True.
- **conv_tol** (float) – Convergence tolerance, by default 1.e-6
- **relax_density** (bool) – Whether to use relaxed or unrelaxed densities, by default False This option is for using CCSD as solver. Relaxed density here uses Lambda amplitudes, whereas unrelaxed density only uses T amplitudes. c.f. See http://classic.chem.msu.su/cgi-bin/ceilidh.exe/gran/gamess/forum/?C34df668afbHW-7216-1405+00.htm for the distinction between the two
- **max_iter** (int) – Maximum number of optimization steps, by default 500
- **nproc** (int) – Total number of processors assigned for the optimization. Defaults to 1. When nproc > 1, Python multithreading is invoked.
- **ompnum** (int) – If nproc > 1, ompnum sets the number of cores for OpenMP parallelization. Defaults to 4
- **jac_solver** (Literal['HF', 'MP2', 'CCSD']) – Method to form Jacobian used in optimization routine, by default HF. Options include HF, MP2, CCSD
- **trust_region** (bool) – Use trust-region based QN optimization, by default False

Return type

None

quemb.molbe.ube.UBE.print_ini

UBE.**print_ini**()

Print initialization banner for the MOLBE calculation.

quemb.molbe.ube.UBE.rdm1_fullbasis

UBE.**rdm1_fullbasis**(*return_ao=True*, *only_rdm1=False*, *only_rdm2=False*, *return_lo=False*, *return_RDM2=True*, *print_energy=False*)

Compute the one- and two-particle reduced density matrices (RDM1 and RDM2).

Parameters

- **return_ao** (*bool, optional*) – Whether to return the RDMs in the AO basis. Default is True.
- **only_rdm1** (*bool, optional*) – Whether to compute only the RDM1. Default is False.

- **`only_rdm2`** (*`bool`*, *`optional`*) – Whether to compute only the RDM2. Default is False.
- **`return_lo`** (*`bool`*, *`optional`*) – Whether to return the RDMs in the localized orbital (LO) basis. Default is False.
- **`return_RDM2`** (*`bool`*, *`optional`*) – Whether to return the two-particle RDM (RDM2). Default is True.
- **`print_energy`** (*`bool`*, *`optional`*) – Whether to print the energy contributions. Default is False.

Returns

- **rdm1AO** (*numpy.ndarray*) – The one-particle RDM in the AO basis.
- **rdm2AO** (*numpy.ndarray*) – The two-particle RDM in the AO basis (if return_RDM2 is True).
- **rdm1LO** (*numpy.ndarray*) – The one-particle RDM in the LO basis (if return_lo is True).
- **rdm2LO** (*numpy.ndarray*) – The two-particle RDM in the LO basis (if return_lo and return_RDM2 are True).
- **rdm1MO** (*numpy.ndarray*) – The one-particle RDM in the molecular orbital (MO) basis (if return_ao is False).
- **rdm2MO** (*numpy.ndarray*) – The two-particle RDM in the MO basis (if return_ao is False and return_RDM2 is True).

quemb.molbe.ube.UBE.read_heff

`UBE.`**`read_heff`**`(`*heff_file='bepotfile.h5'*`)`

Read the effective Hamiltonian from a file.

Parameters

`heff_file` (*`str`*, *`optional`*) – Path to the file storing effective Hamiltonian, by default 'bepotfile.h5'.

quemb.molbe.ube.UBE.save

`UBE.`**`save`**`(`*save_file='storebe.pk'*`)`

Save intermediate results for restart.

Parameters

`save_file` (`str` | `PathLike`) – Path to the file storing restart information, by default 'storebe.pk'.

Return type

`None`

quemb.molbe.ube.UBE.update_fock

`UBE.`**`update_fock`**`(`*heff=None*`)`

Update the Fock matrix for each fragment with the effective Hamiltonian.

Parameters

`heff` (*`list of ndarray`*, *`optional`*) – List of effective Hamiltonian matrices for each fragment, by default None.

Return type

`None`

quemb.molbe.ube.UBE.write_heff

UBE.**write_heff**(*heff_file='bepotfile.h5'*)

Write the effective Hamiltonian to a file.

Parameters

heff_file (*str, optional*) – Path to the file to store effective Hamiltonian, by default 'bepotfile.h5'.

Return type

None

1.2.2 quemb.kbe

Modules

autofrag
fragment
helper
lo
misc
pbe
pfrag
solver

quemb.kbe.autofrag

Functions

add_check_k(min1, flist, sts, ksts, nk_)	
autogen(mol, kpt[, frozen_core, n_BE, ...])	Automatic cell partitioning
be_reduce(n_BE)	
kfrag_func(site_list, numk, nk1, uNs, Ns[, nk2])	
nearestof2coord(coord1, coord2[, bond])	
sidefunc(cell, Idx, unit1, unit2, main_list, ...)	
surround(cell, sidx, unit1, unit2, flist, ...)	
warn_large_fragment()	

quemb.kbe.autofrag.add_check_k

quemb.kbe.autofrag.**add_check_k**(*min1*, *flist*, *sts*, *ksts*, *nk_*)

quemb.kbe.autofrag.autogen

quemb.kbe.autofrag.**autogen**(*mol*, *kpt*, *frozen_core=True*, *n_BE=2*, *write_geom=False*, *unitcell=1*, *gamma_2d=False*, *gamma_1d=False*, *long_bond=False*, *perpend_dist=4.0*, *perpend_dist_tol=0.001*, *nx=False*, *ny=False*, *nz=False*, *iao_valence_basis=None*, *interlayer=False*, *print_frags=True*)

Automatic cell partitioning

Partitions a unitcell into overlapping fragments as defined in BE atom-based fragmentations. It automatically detects branched chemical chains and ring systems and partitions accordingly. For efficiency, it only checks two atoms for connectivity (chemical bond) if they are within 3.5 Angstrom. This value is hardcoded as normdist. Two atoms are defined as bonded if they are within 1.8 Angstrom (1.2 for Hydrogen atom). This is also hardcoded as bond & hbond. Neighboring unitcells are used in the fragmentation, exploiting translational symmetry.

Parameters

- **mol** (*Cell*) – pyscf.pbc.gto.cell.Cell object. This is required for the options, 'autogen', and 'chain' as frag_type.
- **kpt** (*list of int*) – Number of k-points in each lattice vector dimension.
- **frozen_core** (*bool, optional*) – Whether to invoke frozen core approximation. Defaults to True.
- **n_BE** (*int, optional*) – Specifies the order of bootstrap calculation in the atom-based fragmentation, i.e. BE(n). Supported values are 1, 2, 3, and 4 Defaults to 2.
- **write_geom** (*bool, optional*) – Whether to write a 'fragment.xyz' file which contains all the fragments in Cartesian coordinates. Defaults to False.
- **iao_valence_basis** (*str, optional*) – Name of minimal basis set for IAO scheme. 'sto-3g' is sufficient for most cases. Defaults to None.
- **iao_valence_only** (*bool, optional*) – If True, all calculations will be performed in the valence basis in the IAO partitioning. This is an experimental feature. Defaults to False.
- **print_frags** (*bool, optional*) – Whether to print out the list of resulting fragments. Defaults to True.
- **interlayer** (*bool*) – Whether the periodic system has two stacked monolayers.
- **long_bond** (*bool*) – For systems with longer than 1.8 Angstrom covalent bond, set this to True otherwise the fragmentation might fail.

quemb.kbe.autofrag.be_reduce

quemb.kbe.autofrag.**be_reduce**(*n_BE*)

quemb.kbe.autofrag.kfrag_func

quemb.kbe.autofrag.**kfrag_func**(*site_list*, *numk*, *nk1*, *uNs*, *Ns*, *nk2=None*)

quemb.kbe.autofrag.nearestof2coord

quemb.kbe.autofrag.**nearestof2coord**(*coord1*, *coord2*, *bond=4.913298*)

quemb.kbe.autofrag.sidefunc

quemb.kbe.autofrag.**sidefunc**(*cell*, *Idx*, *unit1*, *unit2*, *main_list*, *sub_list*, *coord*, *n_BE*, *bond=4.913298*, *klist=[]*, *ext_list=[]*, *NK=None*, *rlist=[]*)

quemb.kbe.autofrag.surround

quemb.kbe.autofrag.**surround**(*cell*, *sidx*, *unit1*, *unit2*, *flist*, *coord*, *n_BE*, *ext_list*, *klist*, *NK*, *rlist=[]*, *bond=3.4015139999999997*)

quemb.kbe.autofrag.warn_large_fragment

quemb.kbe.autofrag.**warn_large_fragment**()

Classes

AutogenArgs([gamma_2d, gamma_1d, ...])	Additional arguments for autogen

quemb.kbe.autofrag.AutogenArgs

class quemb.kbe.autofrag.**AutogenArgs**(*gamma_2d=False*, *gamma_1d=False*, *interlayer=False*, *long_bond=False*, *perpend_dist=4.0*, *perpend_dist_tol=0.001*, *nx=False*, *ny=False*, *nz=False*)

Additional arguments for autogen

Parameters

- **gamma_2d** (bool)
- **gamma_1d** (bool)
- **interlayer** (bool) – Whether the periodic system has two stacked monolayers.
- **long_bond** (bool) – For systems with longer than 1.8 Angstrom covalent bond, set this to True otherwise the fragmentation might fail.
- **perpend_dist** (float)
- **perpend_dist_tol** (float)
- **nx** (bool)
- **ny** (bool)
- **nz** (bool)

Attributes

gamma_2d: bool

gamma_1d: bool

interlayer: bool

long_bond: bool

perpend_dist: float

perpend_dist_tol: float

nx: bool

ny: bool

nz: bool

Methods

__init__([gamma_2d, gamma_1d, interlayer, ...])	Method generated by attrs for class AutogenArgs.

quemb.kbe.autofrag.AutogenArgs.__init__

`AutogenArgs.__init__`(*gamma_2d=False*, *gamma_1d=False*, *interlayer=False*, *long_bond=False*, *perpend_dist=4.0*, *perpend_dist_tol=0.001*, *nx=False*, *ny=False*, *nz=False*)

Method generated by attrs for class AutogenArgs.

quemb.kbe.fragment

Functions

fragmentate(mol, kpt, *[, natom, frag_type, ...])	Fragment/partitioning definition

quemb.kbe.fragment.fragmentate

`quemb.kbe.fragment.fragmentate`(*mol*, *kpt*, ***, *natom=0*, *frag_type='autogen'*, *unitcell=1*, *iao_valence_basis=None*, *n_BE=2*, *frozen_core=False*, *self_match=False*, *allcen=True*, *print_frags=True*, *additional_args=None*)

Fragment/partitioning definition

Interfaces the main fragmentation function (autogen) in MolBE. It defines edge & center for density matching and energy estimation. It also forms the base for IAO/PAO partitioning for a large basis set bootstrap calculation. Fragments are constructed based on atoms within a unitcell.

Parameters

- **frag_type** (`Literal['autogen']`) – Name of fragmentation function. 'autogen' and 'chemgen' are supported. Defaults to 'autogen'
- **n_BE** (`int`) – Specifies the order of bootstrap calculation in the atom-based fragmentation, i.e. BE(n). For a simple linear system A-B-C-D, BE(1) only has fragments [A], [B], [C], [D] BE(2) has [A, B, C], [B, C, D]
- **mol** (`Cell`) – pyscf.pbc.gto.cell.Cell object. This is required for the options, 'autogen', and 'chain' as frag_type.
- **iao_valence_basis** (`str | None`) – Name of minimal basis set for IAO scheme. 'sto-3g' suffice for most cases.

- **frozen_core** (bool) – Whether to invoke frozen core approximation. This is set to False by default
- **print_frags** (bool) – Whether to print out list of resulting fragments. True by default
- **kpt** (list[int] | tuple[int, int, int]) – No. of k-points in each lattice vector direction. This is the same as kmesh.
- **additional_args** (*ChemGenArgs* | *AutogenArgs* | None) – Additional arguments for different fragmentation functions.

Return type

FragPart

Classes

FragPart(*, unitcell, mol, frag_type, ...)	

quemb.kbe.fragment.FragPart

`class quemb.kbe.fragment.FragPart`(*, *unitcell*, *mol*, *frag_type*, *AO_per_frag*, *AO_per_edge_per_frag*, *ref_frag_idx_per_edge_per_frag*, *weight_and_relAO_per_center_per_frag*, *relAO_per_edge_per_frag*, *relAO_in_ref_per_edge_per_frag*, *relAO_per_origin_per_frag*, *n_BE*, *natom*, *frozen_core*, *self_match*, *allcen*, *iao_valence_basis*, *kpt*)

Attributes

unitcell: int

mol: Cell

frag_type: str

AO_per_frag: list[list[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]

This is a list over fragments and gives the global orbital indices of all atoms in the fragment. These are ordered by the atoms in the fragment.

When using IAOs this refers to the large/working basis.

AO_per_edge_per_frag: list[list[list[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]

The global orbital indices, including hydrogens, per edge per fragment.

When using IAOs this refers to the valence/small basis.

ref_frag_idx_per_edge_per_frag: list[list[NewType(FragmentIdx, integer)]]

Reference fragment index per edge: A list over fragments: list of indices of the fragments in which an edge of the fragment is actually a center. The edge will be matched against this center. For fragments A, B: the A'th element of `.center`, if the edge of A is the center of B, will be B.

weight_and_relAO_per_center_per_frag: list[tuple[float, list[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]

The first element is a float, the second is the list The float weight makes only sense for democratic matching and is currently 1.0 everywhere anyway. We concentrate only on the second part, i.e. the list of indices.

This is a list whose entries are sequences containing the relative orbital index of the center sites within a fragment. Relative is to the own fragment.

When using IAOs this refers to the large/working basis.

relAO_per_edge_per_frag: list[list[list[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]

The relative orbital indices, including hydrogens, per edge per fragment. The index is relative to the own fragment.

When using IAOs this refers to the valence/small basis.

relAO_in_ref_per_edge_per_frag: list[list[list[NewType(RelAOIdxInRef, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]]

The relative atomic orbital indices per edge per fragment. **Note** for this variable relative means that the AO indices are relative to the other fragment where the edge is a center.

When using IAOs this refers to the valence/small basis.

relAO_per_origin_per_frag: list[list[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]

of the motif list for each fragment, this is always a `list(range(0, n))`

When using IAOs this refers to the valence/small basis.

n_BE: int

natom: int

frozen_core: bool

self_match: bool

allcen: bool

iao_valence_basis: str | None

kpt: list[int] | tuple[int, int, int]

n_frag: int

ncore: int | None

no_core_idx: list[int] | None

core_list: list[int] | None

Methods

__init__(*, unitcell, mol, frag_type, ...)	Method generated by attrs for class FragPart.
all_centers_are_origins()	
to_Frags(I, eri_file, unitcell_nkpt)	

quemb.kbe.fragment.FragPart.__init__

FragPart.**__init__**(**, unitcell, mol, frag_type, AO_per_frag, AO_per_edge_per_frag, ref_frag_idx_per_edge_per_frag, weight_and_relAO_per_center_per_frag, relAO_per_edge_per_frag, relAO_in_ref_per_edge_per_frag, relAO_per_origin_per_frag, n_BE, natom, frozen_core, self_match, allcen, iao_valence_basis, kpt*)

Method generated by attrs for class FragPart.

quemb.kbe.fragment.FragPart.all_centers_are_origins

FragPart.**all_centers_are_origins**()

Return type
`bool`

quemb.kbe.fragment.FragPart.to_Frags

FragPart.**to_Frags**(*I, eri_file, unitcell_nkpt*)

Return type
Frags

quemb.kbe.helper

Functions

get_veff(eri_, dm, S, TA, hf_veff[, ...])	Calculate the effective HF potential (Veff) for a given density matrix and electron repulsion integrals.

quemb.kbe.helper.get_veff

quemb.kbe.helper.**get_veff**(*eri_, dm, S, TA, hf_veff, return_veff0=False*)

Calculate the effective HF potential (Veff) for a given density matrix and electron repulsion integrals.

This function computes the effective potential by transforming the density matrix, computing the Coulomb (J) and exchange (K) integrals.

Parameters

- **eri** (*ndarray*) – Electron repulsion integrals.
- **dm** (*ndarray*) – Density matrix. 2D array.
- **S** (*ndarray*) – Overlap matrix.
- **TA** (*ndarray*) – Transformation matrix.
- **hf_veff** (*ndarray*) – Hartree-Fock effective potential for the full system.

quemb.kbe.lo

Functions

`get_iao_k`(Co, S12, S1[, S2, ortho])	
`get_pao_k`(Ciao, S, S12)	
`get_pao_native_k`(Ciao, S, mol, iao_valence_basis)	
`get_xovlp_k`(cell, kpts[, basis])	Gets overlap matrix between the two bases and in secondary basis.
`remove_core_mo_k`(Clo, Ccore, S[, thr])	

quemb.kbe.lo.get_iao_k

`quemb.kbe.lo.`**`get_iao_k`**`(`*Co*, *S12*, *S1*, *S2=None*, *ortho=True*`)`

Parameters

- **`Co`** – occupied coefficient matrix with core
- **`S12`** – ovlp between working (large) basis and valence (minimal) basis. Can be thought of as working basis in valence basis
- **`S1`** – AO ovlp matrix, in working (large) basis
- **`S2`** – valence (minimal) AO ovlp matrix

quemb.kbe.lo.get_pao_k

`quemb.kbe.lo.`**`get_pao_k`**`(`*Ciao*, *S*, *S12*`)`

Parameters

- **`Ciao`** – output of `quemb.kbe.lo.get_iao_k()`
- **`S`** – ao ovlp matrix
- **`S12`** – valence orbitals projected into ao basis

Returns
Cpao – (orthogonalized)

Return type
ndarray

quemb.kbe.lo.get_pao_native_k

`quemb.kbe.lo.`**`get_pao_native_k`**`(`*Ciao*, *S*, *mol*, *iao_valence_basis*, *ortho=True*`)`

Parameters

- **`Ciao`** – output of `get_iao_k`
- **`S`** – ao ovlp matrix
- **`mol`** – mol object
- **`iao_valence_basis`** – basis used for valence orbitals

Returns
Cpao – (symmetrically orthogonalized)

Return type
ndarray

quemb.kbe.lo.get_xovlp_k

quemb.kbe.lo.**get_xovlp_k**(*cell*, *kpts*, *basis='sto-3g'*)

Gets overlap matrix between the two bases and in secondary basis. Used for IAOs: returns the overlap between valence (minimal) and working (large) bases and overlap in the minimal basis

Parameters

- **cell** – pyscf cell object, just need it for the working basis
- **basis** – the IAO basis, Knizia recommended 'minao'

Returns
S12 - Overlap of two basis sets, S22 - Overlap in new basis set

Return type
tuple

quemb.kbe.lo.remove_core_mo_k

quemb.kbe.lo.**remove_core_mo_k**(*Clo*, *Ccore*, *S*, *thr=0.5*)

Classes

KMF(cell[, kpts, mo_coeff, mo_energy])	
Mixin_k_Localize()	

quemb.kbe.lo.KMF

class quemb.kbe.lo.**KMF**(*cell*, *kpts=None*, *mo_coeff=None*, *mo_energy=None*)

Methods

__init__(cell[, kpts, mo_coeff, mo_energy])	

quemb.kbe.lo.KMF.__init__

KMF.**__init__**(*cell*, *kpts=None*, *mo_coeff=None*, *mo_energy=None*)

quemb.kbe.lo.Mixin_k_Localize

class quemb.kbe.lo.**Mixin_k_Localize**

Methods

__init__()	
localize(lo_method[, iao_valence_basis, ...])	Orbital localization

quemb.kbe.lo.Mixin_k_Localize.__init__

`Mixin_k_Localize.`**`__init__`**`()`

quemb.kbe.lo.Mixin_k_Localize.localize

`Mixin_k_Localize.`**`localize`**`(`*lo_method*, *iao_valence_basis='sto-3g'*, *core_basis='sto-3g'*, *iao_wannier=True*, *iao_val_core=True*`)`

Orbital localization

Performs orbital localization computations for periodic systems. For large basis, IAO is recommended augmented with PAO orbitals.

NOTE: For periodic systems, with frozen core, the core and valence are localized SEPARATELY. This is not the case of molecular systems.

Parameters

- **lo_method** (*str*) – Localization method in quantum chemistry. 'lowdin', 'boys','iao', and 'wannier' are supported.
- **iao_valence_basis** (*str*) – Name of valence basis set for IAO scheme. 'sto-3g' suffice for most cases.
- **core_basis** (*str*) – Name of core basis set for IAO scheme. 'sto-3g' suffice for most cases.
- **iao_wannier** (*bool*) – Whether to perform Wannier localization in the IAO space

quemb.kbe.misc

Functions

get_phase(cell, kpts, kmesh)	
get_phase1(cell, kpts, kmesh)	
print_energy(ecorr, e_V_Kapprox, e_F_dg, ...)	
sgeom(cell[, kmesh])	Get a supercell pyscf.pbc.gto.cell.Cell object

quemb.kbe.misc.get_phase

`quemb.kbe.misc.`**`get_phase`**`(`*cell*, *kpts*, *kmesh*`)`

quemb.kbe.misc.get_phase1

`quemb.kbe.misc.get_phase1(cell, kpts, kmesh)`

quemb.kbe.misc.print_energy

`quemb.kbe.misc.print_energy(ecorr, e_V_Kapprox, e_F_dg, e_hf, unitcell_nkpt)`

quemb.kbe.misc.sgeom

`quemb.kbe.misc.sgeom(cell, kmesh=None)`

Get a supercell pyscf.pbc.gto.cell.Cell object

Parameters

- **cell** (*Cell*)
- **kmesh** (*list of int*) – Number of k-points in each lattice vector dimension

Classes

storePBE(Nocc, hf_veff, hcore, S, C, hf_dm, ...)	

quemb.kbe.misc.storePBE

`class quemb.kbe.misc.storePBE(Nocc, hf_veff, hcore, S, C, hf_dm, hf_etot, W, lmo_coeff, enuc, ek, E_core, C_core, P_core, core_veff)`

Attributes

Nocc: int

hf_veff: ndarray[tuple[int, ...], dtype[floating]]

hcore: ndarray[tuple[int, ...], dtype[floating]]

S: ndarray[tuple[int, ...], dtype[floating]]

C: ndarray[tuple[int, ...], dtype[floating]]

hf_dm: ndarray[tuple[int, ...], dtype[floating]]

hf_etot: float

W: ndarray[tuple[int, ...], dtype[floating]]

lmo_coeff: ndarray[tuple[int, ...], dtype[floating]]

enuc: float

ek: float

E_core: float

C_core: ndarray[tuple[int, ...], dtype[floating]]

P_core: `ndarray[tuple[int, ...], dtype[floating]]`

core_veff: `ndarray[tuple[int, ...], dtype[floating]]`

Methods

`__init__`(Nocc, hf_veff, hcore, S, C, hf_dm, ...)	Method generated by attrs for class storePBE.

quemb.kbe.misc.storePBE.__init__

`storePBE.`**`__init__`**(*Nocc*, *hf_veff*, *hcore*, *S*, *C*, *hf_dm*, *hf_etot*, *W*, *lmo_coeff*, *enuc*, *ek*, *E_core*, *C_core*, *P_core*, *core_veff*)

Method generated by attrs for class storePBE.

quemb.kbe.pbe

Functions

`eritransform_parallel`(a, atom, basis, kpts, ...)	Wrapper for parallel eri transformation
`initialize_pot`(n_frag, rel_AO_per_edge_per_frag)	Initialize the potential array for bootstrap embedding.
`parallel_fock_wrapper`(dname, nao, dm, S, TA, ...)	Wrapper for parallel Fock transformation
`parallel_scf_wrapper`(dname, nao, nocc, h1, ...)	Wrapper for performing fragment scf calculation

quemb.kbe.pbe.eritransform_parallel

`quemb.kbe.pbe.`**`eritransform_parallel`**(*a*, *atom*, *basis*, *kpts*, *C_ao_emb*, *cderi*)

Wrapper for parallel eri transformation

quemb.kbe.pbe.initialize_pot

`quemb.kbe.pbe.`**`initialize_pot`**(*n_frag*, *rel_AO_per_edge_per_frag*)

Initialize the potential array for bootstrap embedding.

This function initializes a potential array for a given number of fragments (`n_frag`) and their corresponding edge indices (`rel_AO_per_edge_per_frag`). The potential array is initialized with zeros for each pair of edge site indices within each fragment, followed by an additional zero for the global chemical potential.

Parameters

- **`n_frag`** (*`int`*) – Number of fragments.
- **`rel_AO_per_edge_per_frag`** (*`list of list of list of int`*) – List of edge indices for each fragment. Each element is a list of lists, where each sublist contains the indices of edge sites for a particular fragment.

Returns

Initialized potential array with zeros.

Return type

list of *float*

quemb.kbe.pbe.parallel_fock_wrapper

quemb.kbe.pbe.**parallel_fock_wrapper**(*dname*, *nao*, *dm*, *S*, *TA*, *hf_veff*, *eri_file*)

Wrapper for parallel Fock transformation

quemb.kbe.pbe.parallel_scf_wrapper

quemb.kbe.pbe.**parallel_scf_wrapper**(*dname*, *nao*, *nocc*, *h1*, *dm_init*, *eri_file*)

Wrapper for performing fragment scf calculation

Classes

BE(mf, fobj[, eri_file, lo_method, ...])	Class for handling periodic bootstrap embedding (BE) calculations.

quemb.kbe.pbe.BE

class quemb.kbe.pbe.**BE**(*mf*, *fobj*, *eri_file='eri_file.h5'*, *lo_method='lowdin'*, *compute_hf=True*, *restart=False*, *restart_file='storebe.pk'*, *nproc=1*, *ompnum=4*, *iao_val_core=True*, *exxdiv='ewald'*, *kpts=None*, *cderi=None*, *iao_wannier=False*, *thr_bath=1e-10*, *scratch_dir=None*)

Class for handling periodic bootstrap embedding (BE) calculations.

This class encapsulates the functionalities required for performing periodic bootstrap embedding calculations, including setting up the BE environment, initializing fragments, performing SCF calculations, and evaluating energies.

mf

PySCF mean-field object.

fobj

Fragment object containing sites, centers, edges, and indices.

eri_file

Path to the file storing two-electron integrals.

lo_method

Method for orbital localization, default is 'lowdin'.

Methods

__init__(mf, fobj[, eri_file, lo_method, ...])	Constructor for BE object.
ewald_sum()	
get_be_error_jacobian([jac_solver])	
initialize(compute_hf[, restart])	Initialize the Bootstrap Embedding calculation.
localize(lo_method[, iao_valence_basis, ...])	Orbital localization
oneshot([solver, use_cumulant, nproc, ...])	Perform a one-shot bootstrap embedding calculation.
optimize([solver, method, only_chem, ...])	BE optimization function
print_ini()	Print initialization banner for the kBE calculation.
read_heff([heff_file])	Read the effective Hamiltonian from a file.
save([restart_file])	Save the current state of the BE calculation to a file.

continues on next page

Table 69 – continued from previous page

update_fock([heff])	Update the Fock matrix for each fragment with the effective Hamiltonian.
write_heff([heff_file])	Write the effective Hamiltonian to a file.

quemb.kbe.pbe.BE.__init__

`BE.__init__`(*mf*, *fobj*, *eri_file='eri_file.h5'*, *lo_method='lowdin'*, *compute_hf=True*, *restart=False*, *restart_file='storebe.pk'*, *nproc=1*, *ompnum=4*, *iao_val_core=True*, *exxdiv='ewald'*, *kpts=None*, *cderi=None*, *iao_wannier=False*, *thr_bath=1e-10*, *scratch_dir=None*)

Constructor for BE object.

Parameters

- **mf** (KRHF) – PySCF periodic mean-field object.
- **fobj** (*FragPart*) – Fragment object containing sites, centers, edges, and indices.
- **kpts** (list[list[float]] | None) – k-points in the reciprocal space for periodic computation
- **eri_file** (str | PathLike) – Path to the file storing two-electron integrals, by default 'eri_file.h5'.
- **lo_method** (str) – Method for orbital localization, by default 'lowdin'.
- **iao_wannier** (bool) – Whether to perform Wannier localization on the IAO space, by default False.
- **compute_hf** (bool) – Whether to compute Hartree-Fock energy, by default True.
- **restart** (bool) – Whether to restart from a previous calculation, by default False.
- **restart_file** (str | PathLike) – Path to the file storing restart information, by default 'storebe.pk'.
- **nproc** (int) – Number of processors for parallel calculations, by default 1. If set to >1, multi-threaded parallel computation is invoked.
- **ompnum** (int) – Number of OpenMP threads, by default 4.
- **thr_bath** (*float*,) – Threshold for bath orbitals in Schmidt decomposition
- **scratch_dir** (*WorkDir* | None) – Scratch directory.

quemb.kbe.pbe.BE.ewald_sum

`BE.ewald_sum()`

quemb.kbe.pbe.BE.get_be_error_jacobian

`BE.get_be_error_jacobian`(*jac_solver='HF'*)

Return type

ndarray[tuple[int, ...], dtype[floating]]

quemb.kbe.pbe.BE.initialize

BE.**initialize**(*compute_hf*, *restart=False*)

Initialize the Bootstrap Embedding calculation.

Parameters

- **compute_hf** (*bool*) – Whether to compute Hartree-Fock energy.
- **restart** (*bool, optional*) – Whether to restart from a previous calculation, by default False.

Return type

None

quemb.kbe.pbe.BE.localize

BE.**localize**(*lo_method*, *iao_valence_basis='sto-3g'*, *core_basis='sto-3g'*, *iao_wannier=True*, *iao_val_core=True*)

Orbital localization

Performs orbital localization computations for periodic systems. For large basis, IAO is recommended augmented with PAO orbitals.

NOTE: For periodic systems, with frozen core, the core and valence are localized SEPARATELY. This is not the case of molecular systems.

Parameters

- **lo_method** (*str*) – Localization method in quantum chemistry. 'lowdin', 'boys','iao', and 'wannier' are supported.
- **iao_valence_basis** (*str*) – Name of valence basis set for IAO scheme. 'sto-3g' suffice for most cases.
- **core_basis** (*str*) – Name of core basis set for IAO scheme. 'sto-3g' suffice for most cases.
- **iao_wannier** (*bool*) – Whether to perform Wannier localization in the IAO space

quemb.kbe.pbe.BE.oneshot

BE.**oneshot**(*solver='CCSD'*, *use_cumulant=True*, *nproc=1*, *ompnum=4*, *solver_args=None*)

Perform a one-shot bootstrap embedding calculation.

Parameters

- **solver** (Literal['MP2', 'CCSD', 'FCI', 'HCI', 'SHCI', 'SCI', 'DMRG']) – High-level quantum chemistry method, by default 'CCSD'. 'CCSD', 'FCI', and variants of selected CI are supported.
- **use_cumulant** (bool) – Whether to use the cumulant energy expression, by default True.
- **nproc** (int) – Number of processors for parallel calculations, by default 1. If set to >1, threaded parallel computation is invoked.
- **ompnum** (int) – Number of OpenMP threads, by default 4.
- **clean_eri** – Whether to clean up ERI files after calculation, by default False.

Return type

None

quemb.kbe.pbe.BE.optimize

`BE.optimize(`*solver='CCSD', method='QN', only_chem=False, use_cumulant=True, conv_tol=1e-06, relax_density=False, nproc=1, ompnum=4, max_iter=500, jac_solver='HF', trust_region=False*`)`

BE optimization function

Interfaces BEOPT to perform bootstrap embedding optimization.

Parameters

- **solver** (*str, optional*) – High-level solver for the fragment, by default 'CCSD'
- **method** (*str, optional*) – Optimization method, by default 'QN'
- **only_chem** (*bool, optional*) – If true, density matching is not performed – only global chemical potential is optimized, by default False
- **use_cumulant** (bool) – Whether to use the cumulant energy expression, by default True.
- **conv_tol** (*float, optional*) – Convergence tolerance, by default 1.e-6
- **relax_density** (*bool, optional*) – Whether to use relaxed or unrelaxed densities, by default False This option is for using CCSD as solver. Relaxed density here uses Lambda amplitudes, whereas unrelaxed density only uses T amplitudes. c.f. See http://classic.chem.msu.su/cgi-bin/ceilidh.exe/gran/gamess/forum/?C34df668afbHW-7216-1405+00.htm for the distinction between the two
- **max_iter** (*int, optional*) – Maximum number of optimization steps, by default 500
- **nproc** (*int*) – Total number of processors assigned for the optimization. Defaults to 1. When nproc > 1, Python multithreading is invoked.
- **ompnum** (*int*) – If nproc > 1, ompnum sets the number of cores for OpenMP parallelization. Defaults to 4
- **jac_solver** (Literal['HF', 'MP2', 'CCSD']) – Method to form Jacobian used in optimization routine, by default HF. Options include HF, MP2, CCSD
- **trust_region** (bool) – Use trust-region based QN optimization, by default False

Return type

None

quemb.kbe.pbe.BE.print_ini

`BE.print_ini()`

Print initialization banner for the kBE calculation.

Return type

None

quemb.kbe.pbe.BE.read_heff

`BE.read_heff(`*heff_file='bepotfile.h5'*`)`

Read the effective Hamiltonian from a file.

Parameters

heff_file (*str, optional*) – Path to the file storing effective Hamiltonian, by default 'bepotfile.h5'.

quemb.kbe.pbe.BE.save

`BE.save(restart_file='storebe.pk')`

Save the current state of the BE calculation to a file.

Parameters
restart_file (*str, optional*) – Path to the file storing restart information, by default 'storebe.pk'.

Return type
None

quemb.kbe.pbe.BE.update_fock

`BE.update_fock(heff=None)`

Update the Fock matrix for each fragment with the effective Hamiltonian.

Parameters
heff (*list of ndarray, optional*) – List of effective Hamiltonian matrices for each fragment, by default None.

quemb.kbe.pbe.BE.write_heff

`BE.write_heff(heff_file='bepotfile.h5')`

Write the effective Hamiltonian to a file.

Parameters
heff_file (*str, optional*) – Path to the file to store effective Hamiltonian, by default 'bepotfile.h5'.

quemb.kbe.pfrag

Classes

Frags(*, AO_in_frag, ifrag, AO_per_edge, ...)	Class for handling fragments in periodic bootstrap embedding.

quemb.kbe.pfrag.Frags

`class quemb.kbe.pfrag.Frags(*, AO_in_frag, ifrag, AO_per_edge, ref_frag_idx_per_edge, relAO_per_edge, relAO_in_ref_per_edge, weight_and_relAO_per_center, relAO_per_origin, eri_file, unitcell_nkpt, unitcell)`

Class for handling fragments in periodic bootstrap embedding.

This class contains various functionalities required for managing and manipulating fragments for periodic BE calculations.

Methods

__init__(*, AO_in_frag, ifrag, AO_per_edge, ...)	Constructor function for `Frags` class.
cons_fock(hf_veff, S, dm[, eri_])	Construct the Fock matrix for the fragment.
cons_h1(h1)	Construct the one-electron Hamiltonian for the fragment.

continues on next page

Table 71 – continued from previous page

get_nsocc(S, C, nocc[, ncore])	Get the number of occupied orbitals for the fragment.
scf([heff, fs, eri, dm0])	Perform self-consistent field (SCF) calculation for the fragment.
sd(lao, lmo, nocc, thr_bath[, cell, kpts, ...])	Perform Schmidt decomposition for the fragment.
set_udim(cout)	
update_ebe_hf([rdm_hf, mo_coeffs, eri, ...])	
update_heff(u[, cout, do_chempot, only_chem])	Update the effective Hamiltonian for the fragment.

quemb.kbe.pfrag.Frags.__init__

`Frags.__init__`(**, AO_in_frag, ifrag, AO_per_edge, ref_frag_idx_per_edge, relAO_per_edge, relAO_in_ref_per_edge, weight_and_relAO_per_center, relAO_per_origin, eri_file, unitcell_nkpt, unitcell*)

Constructor function for `Frags` class.

Parameters

- **AO_in_frag** (Sequence[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]) – list of AOs in the fragment (i.e. pbe.AO_per_frag[i] or FragPart.AO_per_frag[i]) Read more detailed description in *quemb.kbe.fragment.FragPart*.
- **ifrag** (int) – fragment index ($\in [0, \text{pbe.n\_frag} - 1]$)
- **AO_per_edge** (Sequence[Sequence[NewType(GlobalAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]) – list of lists of edge site AOs for each atom in the fragment. Read more detailed description in *quemb.kbe.fragment.FragPart*.
- **ref_frag_idx_per_edge** (Sequence[NewType(FragmentIdx, integer)]) – list of fragment indices where edge site AOs are center site. Read more detailed description in *quemb.kbe.fragment.FragPart*.
- **rel_AO_per_edge** – list of lists of indices for edge site AOs within the fragment, Read more detailed description in *quemb.kbe.fragment.FragPart*.
- **relAO_in_ref_per_edge** (Sequence[Sequence[NewType(RelAOIdxInRef, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]) – list of lists of indices within the fragment specified in `center` that points to the edge site AOs. Read more detailed description in *quemb.kbe.fragment.FragPart*.
- **relAO_per_origin** (Sequence[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]) – indices of the origin in the fragment, by default None
- **weight_and_relAO_per_center** (tuple[float, Sequence[NewType(RelAOIdx, NewType(AOIdx, NewType(OrbitalIdx, integer)))]]) – weight used for energy contributions, by default None
- **eri_file** (str | PathLike) – two-electron integrals stored as h5py file, by default 'eri_file.h5'
- **unitcell_nkpt** (int)
- **unitcell** (int)

quemb.kbe.pfrag.Frags.cons_fock

`Frags.cons_fock(hf_veff, S, dm, eri_=None)`

Construct the Fock matrix for the fragment.

Parameters

- **hf_veff** (*ndarray*) – Hartree-Fock effective potential.
- **S** (*ndarray*) – Overlap matrix.
- **dm** (*ndarray*) – Density matrix.
- **eri** (*ndarray, optional*) – Electron repulsion integrals, by default None.

quemb.kbe.pfrag.Frags.cons_h1

`Frags.cons_h1(h1)`

Construct the one-electron Hamiltonian for the fragment.

Parameters

h1 (*ndarray*) – One-electron Hamiltonian matrix.

quemb.kbe.pfrag.Frags.get_nsocc

`Frags.get_nsocc(S, C, nocc, ncore=0)`

Get the number of occupied orbitals for the fragment.

Parameters

- **S** (*ndarray*) – Overlap matrix.
- **C** (*ndarray*) – Molecular orbital coefficients.
- **nocc** (*int*) – Number of occupied orbitals.
- **ncore** (*int, optional*) – Number of core orbitals, by default 0.

Returns

Projected density matrix.

Return type

ndarray

quemb.kbe.pfrag.Frags.scf

`Frags.scf(heff=None, fs=False, eri=None, dm0=None)`

Perform self-consistent field (SCF) calculation for the fragment.

Parameters

- **heff** (*ndarray, optional*) – Effective Hamiltonian, by default None.
- **fs** (*bool, optional*) – Flag for full SCF, by default False.
- **eri** (*ndarray, optional*) – Electron repulsion integrals, by default None.
- **dm0** (*ndarray, optional*) – Initial density matrix, by default None.

quemb.kbe.pfrag.Frags.sd

`Frags.sd`(*lao*, *lmo*, *nocc*, *thr_bath*, *cell=None*, *kpts=None*, *kmesh=None*, *h1=None*)

Perform Schmidt decomposition for the fragment.

Parameters

- **lao** (*ndarray*) – Orthogonalized AOs
- **lmo** (*ndarray*) – Local molecular orbital coefficients.
- **nocc** (*int*) – Number of occupied orbitals.
- **cell** (*Cell*) – PySCF pbc.gto.cell.Cell object defining the unit cell and lattice vectors.
- **kpts** (*list of list of float*) – k-points in the reciprocal space for periodic computations
- **kmesh** (*list of int*) – Number of k-points in each lattice vector direction

Return type

`None`

quemb.kbe.pfrag.Frags.set_udim

`Frags.set_udim`(*cout*)

quemb.kbe.pfrag.Frags.update_ebe_hf

`Frags.update_ebe_hf`(*rdm_hf=None*, *mo_coeffs=None*, *eri=None*, *return_e1=False*, *unrestricted=False*)

quemb.kbe.pfrag.Frags.update_heff

`Frags.update_heff`(*u*, *cout=None*, *do_chempot=True*, *only_chem=False*)

Update the effective Hamiltonian for the fragment.

quemb.kbe.solver

Functions

schmidt_decomp_svd(rdm, Frag_sites[, thr_bath])	Perform decomposition on the orbital coefficients in the real space.

quemb.kbe.solver.schmidt_decomp_svd

`quemb.kbe.solver.schmidt_decomp_svd`(*rdm*, *Frag_sites*, *thr_bath=1e-10*)

Perform decomposition on the orbital coefficients in the real space.

This function decomposes the molecular orbitals into fragment and environment parts using the Schmidt decomposition method. It computes the transformation matrix (TA) which includes both the fragment orbitals and the entangled bath.

Parameters

- **rdm** (*ndarray*) – Density matrix (HF) in the real space.
- **Frag_sites** (*list of int*) – List of fragment sites (indices).
- **thr_bath** (*float,*) – Threshold for bath orbitals in Schmidt decomposition

Returns
Transformation matrix (TA) including both fragment and entangled bath orbitals.

Return type
ndarray

1.2.3 quemb.shared

Modules

config	Configure quemb
external	
helper	
io	
manage_scratch	
numba_helpers	
typing	Define some types that do not fit into one particular module

quemb.shared.config

Configure quemb

One can modify settings in one session or create an RC-file. See examples below.

Examples

```
>>> from quemb.shared.config import settings
>>>
>>> settings.SCRATCH_ROOT = "/scratch"
Changes the default root for the scratch directory
for this python session.
```

```
>>> from quemb.shared.config import dump_settings
>>>
>>> dump_settings()
Creates ~/.quembrc.yml file that allows changes to persist.
```

To specify the print level, do

```
>>> import logging
>>> logger = logging.getLogger()
>>> logger.setLevel(logging.DEBUG)
in your input file. Currently the options are DEBUG or INFO.
For more information on how logging works, see
https://docs.python.org/3/howto/logging.html
```

Functions

dump_settings()	Writes settings to `~/.quembrc.yml`

quemb.shared.config.dump_settings

`quemb.shared.config.`**`dump_settings`**`()`

Writes settings to `~/.quembrc.yml`

Return type
`None`

Classes

Settings([SCRATCH_ROOT, ...])

quemb.shared.config.Settings

`class` `quemb.shared.config.`**`Settings`**`(`*SCRATCH_ROOT=PosixPath('/var/folders/fg/2sf1l59d7nl0y23y4zcfn0_40000gn/T'), INTEGRAL_TRANSFORM_MAX_MEMORY=50*`)`

Attributes

`SCRATCH_ROOT:` `Path`

`INTEGRAL_TRANSFORM_MAX_MEMORY:` `float`

Methods

__init__([SCRATCH_ROOT, ...])	Method generated by attrs for class Settings.

quemb.shared.config.Settings.__init__

`Settings.`**`__init__`**`(`*SCRATCH_ROOT=PosixPath('/var/folders/fg/2sf1l59d7nl0y23y4zcfn0_40000gn/T'), INTEGRAL_TRANSFORM_MAX_MEMORY=50*`)`

Method generated by attrs for class Settings.

quemb.shared.external

Modules

ccsd_rdm
cphf_utils
cpmp2_utils

continues on next page

Table 77 – continued from previous page

jac_utils
lo_helper
optqn
uccsd_eri
unrestricted_utils

quemb.shared.external.ccsd_rdm

Functions

make_rdm1_ccsd_t1(t1)
make_rdm1_uccsd(ucc[, relax])
make_rdm2_uccsd(ucc[, relax, with_dm1])
make_rdm2_urlx(t1, t2[, with_dm1])

quemb.shared.external.ccsd_rdm.make_rdm1_ccsd_t1

`quemb.shared.external.ccsd_rdm.`**`make_rdm1_ccsd_t1`**(*t1*)

quemb.shared.external.ccsd_rdm.make_rdm1_uccsd

`quemb.shared.external.ccsd_rdm.`**`make_rdm1_uccsd`**(*ucc*, *relax=False*)

quemb.shared.external.ccsd_rdm.make_rdm2_uccsd

`quemb.shared.external.ccsd_rdm.`**`make_rdm2_uccsd`**(*ucc*, *relax=False*, *with_dm1=True*)

quemb.shared.external.ccsd_rdm.make_rdm2_urlx

`quemb.shared.external.ccsd_rdm.`**`make_rdm2_urlx`**(*t1*, *t2*, *with_dm1=True*)

quemb.shared.external.cphf_utils

Functions

cphf_kernel(C, moe, eri, no, v)
cphf_kernel_batch(C, moe, eri, no, vs)

continues on next page

Table 79 – continued from previous page

get_cphf_A(C, moe, eri, no)	
get_cphf_rhs(C, no, v)	
get_cpuhf_A(C, moe, eri, no)	eri could be a single numpy array (in spinless basis) or a tuple/list of 3 numpy arrays in the order [aa,bb,ab]
get_cpuhf_A_spin_eri(C, moe, eri, no)	
get_cpuhf_A_spinless_eri(C, moe, eri, no)	
get_cpuhf_u(C, moe, eri, no, vpot)	
get_cpuhf_u_batch(C, moe, eri, no, vpots)	
get_dP_lagrangian(C, no)	
get_full_u(C, moe, eri, no, v, u[, thresh])	
get_full_u_batch(C, moe, eri, no, vs, us[, ...])	
get_rhf_dP_from_u(C, no, u)	
get_uhf_dP_from_u(C, no, u)	
get_zvec(C, moe, eri, no)	
uvo_as_full_u_batch(nao, no, us)	

quemb.shared.external.cphf_utils.cphf_kernel

quemb.shared.external.cphf_utils.**cphf_kernel**(*C*, *moe*, *eri*, *no*, *v*)

quemb.shared.external.cphf_utils.cphf_kernel_batch

quemb.shared.external.cphf_utils.**cphf_kernel_batch**(*C*, *moe*, *eri*, *no*, *vs*)

quemb.shared.external.cphf_utils.get_cphf_A

quemb.shared.external.cphf_utils.**get_cphf_A**(*C*, *moe*, *eri*, *no*)

quemb.shared.external.cphf_utils.get_cphf_rhs

quemb.shared.external.cphf_utils.**get_cphf_rhs**(*C*, *no*, *v*)

quemb.shared.external.cphf_utils.get_cpuhf_A

quemb.shared.external.cphf_utils.**get_cpuhf_A**(*C*, *moe*, *eri*, *no*)

eri could be a single numpy array (in spinless basis) or a tuple/list of 3 numpy arrays in the order [aa,bb,ab]

quemb.shared.external.cphf_utils.get_cpuhf_A_spin_eri

`quemb.shared.external.cphf_utils.`**`get_cpuhf_A_spin_eri`**(*C*, *moe*, *eri*, *no*)

quemb.shared.external.cphf_utils.get_cpuhf_A_spinless_eri

`quemb.shared.external.cphf_utils.`**`get_cpuhf_A_spinless_eri`**(*C*, *moe*, *eri*, *no*)

quemb.shared.external.cphf_utils.get_cpuhf_u

`quemb.shared.external.cphf_utils.`**`get_cpuhf_u`**(*C*, *moe*, *eri*, *no*, *vpot*)

quemb.shared.external.cphf_utils.get_cpuhf_u_batch

`quemb.shared.external.cphf_utils.`**`get_cpuhf_u_batch`**(*C*, *moe*, *eri*, *no*, *vpots*)

quemb.shared.external.cphf_utils.get_dP_lagrangian

`quemb.shared.external.cphf_utils.`**`get_dP_lagrangian`**(*C*, *no*)

quemb.shared.external.cphf_utils.get_full_u

`quemb.shared.external.cphf_utils.`**`get_full_u`**(*C*, *moe*, *eri*, *no*, *v*, *u*, *thresh=100000000.0*)

quemb.shared.external.cphf_utils.get_full_u_batch

`quemb.shared.external.cphf_utils.`**`get_full_u_batch`**(*C*, *moe*, *eri*, *no*, *vs*, *us*, *thresh=10000000000.0*)

quemb.shared.external.cphf_utils.get_rhf_dP_from_u

`quemb.shared.external.cphf_utils.`**`get_rhf_dP_from_u`**(*C*, *no*, *u*)

Return type
`ndarray`[`tuple`[`int`, `...`], `dtype`[`float64`]]

quemb.shared.external.cphf_utils.get_uhf_dP_from_u

`quemb.shared.external.cphf_utils.`**`get_uhf_dP_from_u`**(*C*, *no*, *u*)

quemb.shared.external.cphf_utils.get_zvec

`quemb.shared.external.cphf_utils.`**`get_zvec`**(*C*, *moe*, *eri*, *no*)

quemb.shared.external.cphf_utils.uvo_as_full_u_batch

`quemb.shared.external.cphf_utils.`**`uvo_as_full_u_batch`**(*nao*, *no*, *us*)

quemb.shared.external.cpmp2_utils

Functions

`get_Diajb_r`(moe, no)	
`get_Diajb_u`(moe, no)	
`get_Pmp2_r`(t2l, t2r)	
`get_Pmp2_u`(t2l, t2r)	
`get_dF_r`(no, V, C, Q, u)	
`get_dF_u`(no, V, C, Q, u)	
`get_dPmp2_batch_r`(C, moe, V, no, Qs[, aorep])	Derivative of oo and vv block of the MP2 density
`get_dPmp2_batch_u`(C, moe, V, no, Qs[, aorep])	no = [noa, nob] V = [Vaa, Vbb, Vab] C = [Ca, Cb] moe = [moea, moeb]
`get_dVovov_r`(no, V, C, u)	
`get_dVovov_u`(no, V, C, u)	
`get_dmoe_F_r`(C, dF)	
`get_dmoe_F_u`(C, dF)	
`get_full_u_F_r`(no, C, moe, dF, u)	
`get_full_u_F_u`(no, C, moe, dF, u)	

quemb.shared.external.cpmp2_utils.get_Diajb_r

`quemb.shared.external.cpmp2_utils.`**`get_Diajb_r`**(*moe*, *no*)

quemb.shared.external.cpmp2_utils.get_Diajb_u

`quemb.shared.external.cpmp2_utils.`**`get_Diajb_u`**(*moe*, *no*)

quemb.shared.external.cpmp2_utils.get_Pmp2_r

`quemb.shared.external.cpmp2_utils.`**`get_Pmp2_r`**(*t2l*, *t2r*)

quemb.shared.external.cpmp2_utils.get_Pmp2_u

`quemb.shared.external.cpmp2_utils.`**`get_Pmp2_u`**(*t2l*, *t2r*)

quemb.shared.external.cpmp2_utils.get_dF_r

`quemb.shared.external.cpmp2_utils.`**`get_dF_r`**(*no*, *V*, *C*, *Q*, *u*)

quemb.shared.external.cpmp2_utils.get_dF_u

quemb.shared.external.cpmp2_utils.**get_dF_u**(*no*, *V*, *C*, *Q*, *u*)

quemb.shared.external.cpmp2_utils.get_dPmp2_batch_r

quemb.shared.external.cpmp2_utils.**get_dPmp2_batch_r**(*C*, *moe*, *V*, *no*, *Qs*, *aorep=True*)

Derivative of oo and vv block of the MP2 density

quemb.shared.external.cpmp2_utils.get_dPmp2_batch_u

quemb.shared.external.cpmp2_utils.**get_dPmp2_batch_u**(*C*, *moe*, *V*, *no*, *Qs*, *aorep=True*)

no = [noa, nob] V = [Vaa, Vbb, Vab] C = [Ca, Cb] moe = [moea, moeb]

quemb.shared.external.cpmp2_utils.get_dVovov_r

quemb.shared.external.cpmp2_utils.**get_dVovov_r**(*no*, *V*, *C*, *u*)

quemb.shared.external.cpmp2_utils.get_dVovov_u

quemb.shared.external.cpmp2_utils.**get_dVovov_u**(*no*, *V*, *C*, *u*)

quemb.shared.external.cpmp2_utils.get_dmoe_F_r

quemb.shared.external.cpmp2_utils.**get_dmoe_F_r**(*C*, *dF*)

quemb.shared.external.cpmp2_utils.get_dmoe_F_u

quemb.shared.external.cpmp2_utils.**get_dmoe_F_u**(*C*, *dF*)

quemb.shared.external.cpmp2_utils.get_full_u_F_r

quemb.shared.external.cpmp2_utils.**get_full_u_F_r**(*no*, *C*, *moe*, *dF*, *u*)

quemb.shared.external.cpmp2_utils.get_full_u_F_u

quemb.shared.external.cpmp2_utils.**get_full_u_F_u**(*no*, *C*, *moe*, *dF*, *u*)

quemb.shared.external.jac_utils

Functions

get_Vmogen_r(no, V, C, pattern)
get_dPccsdurlx_batch_u(C, moe, eri, no, vpots)
get_dVmogen_r(no, V, C, u, pattern)
get_dt1ao_an(no, V, C, moe, Qs[, us])

continues on next page

Table 81 – continued from previous page

get_t1(no, nv, moe, Vovov, Voovo, Vvovv)	

quemb.shared.external.jac_utils.get_Vmogen_r

quemb.shared.external.jac_utils.**get_Vmogen_r**(*no*, *V*, *C*, *pattern*)

quemb.shared.external.jac_utils.get_dPccsdurlx_batch_u

quemb.shared.external.jac_utils.**get_dPccsdurlx_batch_u**(*C*, *moe*, *eri*, *no*, *vpots*)

quemb.shared.external.jac_utils.get_dVmogen_r

quemb.shared.external.jac_utils.**get_dVmogen_r**(*no*, *V*, *C*, *u*, *pattern*)

quemb.shared.external.jac_utils.get_dt1ao_an

quemb.shared.external.jac_utils.**get_dt1ao_an**(*no*, *V*, *C*, *moe*, *Qs*, *us=None*)

quemb.shared.external.jac_utils.get_t1

quemb.shared.external.jac_utils.**get_t1**(*no*, *nv*, *moe*, *Vovov*, *Voovo*, *Vvovv*)

quemb.shared.external.lo_helper

Functions

cano_orth(A[, thr, ovlp])	Canonically orthogonalize columns of A
dot_gen(A, B[, ovlp])	Return product A.T @ B or A.T @ ovlp @ B
get_aoind_by_atom(mol[, atomind_by_motif])	Return a list across all atoms (motifs).
get_cano_orth_mat(A[, thr, ovlp])	Perform canonical orthogonalization of A
get_symm_mat_pow(A, p[, check_symm, thresh])	A ** p where A is symmetric
get_symm_orth_mat(A[, thr, ovlp])	Perform symmetric orthogonalization of A
*reorder_by_atom_*(Clo, aoind_by_atom, S[, thr])	Reorder the ~LOCALIZED~ Clo orbitals by atom
symm_orth(A[, thr, ovlp])	Symmetrically orthogonalize columns of A

quemb.shared.external.lo_helper.cano_orth

quemb.shared.external.lo_helper.**cano_orth**(*A*, *thr=1e-06*, *ovlp=None*)

Canonically orthogonalize columns of A

Return type

ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]

quemb.shared.external.lo_helper.dot_gen

quemb.shared.external.lo_helper.**dot_gen**(*A*, *B*, *ovlp=None*)

Return product A.T @ B or A.T @ ovlp @ B

Return type

ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]

quemb.shared.external.lo_helper.get_aoind_by_atom

quemb.shared.external.lo_helper.**get_aoind_by_atom**(*mol*, *atomind_by_motif=None*)

Return a list across all atoms (motifs). Each element contains a list of AO indices for that atom (or motif, if atomind_by_motif True)

quemb.shared.external.lo_helper.get_cano_orth_mat

quemb.shared.external.lo_helper.**get_cano_orth_mat**(*A*, *thr=1e-06*, *ovlp=None*)

Perform canonical orthogonalization of A

Return type

ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]

quemb.shared.external.lo_helper.get_symm_mat_pow

quemb.shared.external.lo_helper.**get_symm_mat_pow**(*A*, *p*, *check_symm=True*, *thresh=1e-08*)

A ** p where A is symmetric

Note:

For integer p, it calls numpy.linalg.matrix_power

quemb.shared.external.lo_helper.get_symm_orth_mat

quemb.shared.external.lo_helper.**get_symm_orth_mat**(*A*, *thr=1e-06*, *ovlp=None*)

Perform symmetric orthogonalization of A

Return type

ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]

quemb.shared.external.lo_helper.reorder_by_atom_

quemb.shared.external.lo_helper.**reorder_by_atom_**(*Clo*, *aoind_by_atom*, *S*, *thr=0.5*)

Reorder the ~LOCALIZED~ Clo orbitals by atom

quemb.shared.external.lo_helper.symm_orth

quemb.shared.external.lo_helper.**symm_orth**(*A*, *thr=1e-06*, *ovlp=None*)

Symmetrically orthogonalize columns of A

Return type

ndarray[tuple[int, ...], dtype[TypeVar(T_dtype_co, bound= generic, covariant=True)]]

quemb.shared.external.optqn

Functions

ccsdres_func(mf, vpots, eri, nsocc)	
get_atbe_Jblock_frag(fobj, res_func)	
get_be_error_jacobian(n_frag, Fobjs[, ...])	
get_be_error_jacobian_selffrag(self[, ...])	
get_vpots_frag(nao, rel_AO_per_edge, AO_in_frag)	
hfres_func(mf, vpots, eri, nsocc)	
line_search_LF(func, xold, fold, dx, iter_)	Adapted from D.-H.
mp2res_func(mf, vpots, eri, nsocc)	
trustRegion(func, xold, fold, Binv[, c])	Perform Trust Region Optimization.

quemb.shared.external.optqn.ccsdres_func

`quemb.shared.external.optqn.`**`ccsdres_func`**`(`*mf*, *vpots*, *eri*, *nsocc*`)`

quemb.shared.external.optqn.get_atbe_Jblock_frag

`quemb.shared.external.optqn.`**`get_atbe_Jblock_frag`**`(`*fobj*, *res_func*`)`

Return type
`tuple[ndarray[tuple[int, ...], dtype[float64]], ndarray[tuple[int, ...], dtype[float64]], list, list, list, float, int]`

quemb.shared.external.optqn.get_be_error_jacobian

`quemb.shared.external.optqn.`**`get_be_error_jacobian`**`(`*n_frag*, *Fobjs*, *jac_solver='HF'*`)`

quemb.shared.external.optqn.get_be_error_jacobian_selffrag

`quemb.shared.external.optqn.`**`get_be_error_jacobian_selffrag`**`(`*self*, *jac_solver='HF'*`)`

quemb.shared.external.optqn.get_vpots_frag

`quemb.shared.external.optqn.`**`get_vpots_frag`**`(`*nao*, *rel_AO_per_edge*, *AO_in_frag*`)`

Return type
`list[ndarray[tuple[int, ...], dtype[float64]]]`

quemb.shared.external.optqn.hfres_func

`quemb.shared.external.optqn.`**`hfres_func`**`(`*mf*, *vpots*, *eri*, *nsocc*`)`

Return type
`tuple[list[ndarray[tuple[int, ...], dtype[float64]]], ndarray[tuple[int, ...], dtype[float64]]]`

quemb.shared.external.optqn.line_search_LF

`quemb.shared.external.optqn.`**`line_search_LF`**(*func*, *xold*, *fold*, *dx*, *iter_*)

Adapted from D.-H. Li and M. Fukushima, Optimization Metheods and Software, 13, 181 (2000)

quemb.shared.external.optqn.mp2res_func

`quemb.shared.external.optqn.`**`mp2res_func`**(*mf*, *vpots*, *eri*, *nsocc*)

quemb.shared.external.optqn.trustRegion

`quemb.shared.external.optqn.`**`trustRegion`**(*func*, *xold*, *fold*, *Binv*, *c=0.5*)

Perform Trust Region Optimization.

See "A Broyden Trust Region Quasi-Newton Method for Nonlinear Equations" (https://www.iaeng.org/IJCS/issues_v46/issue_3/IJCS_46_3_09.pdf)" Algorithm 1 for more information

Parameters

- **func** (*Callable*) – Cost function
- **xold** (*list or ndarray*) – Current x_p (potentials in BE optimization)
- **fold** (*list or ndarray*) – Current f(x_p) (error vector)
- **Binv** (*ndarray*) – Inverse of Jacobian approximate (B^{-1}); This is updated in Broyden's Method through Sherman-Morrison formula
- **c** (*float, optional*) – Initial value of trust radius $\in (0, 1)$, by default 0.5

Returns

xnew, fnew – x_{p+1} and f_{p+1}. These values are used to proceed with Broyden's Method.

Return type

tuple

Classes

FrankQN(func, x0, f0, J0[, trust, max_space])	Quasi Newton Optimization

quemb.shared.external.optqn.FrankQN

`class quemb.shared.external.optqn.`**`FrankQN`**(*func*, *x0*, *f0*, *J0*, *trust=0.5*, *max_space=500*)

Quasi Newton Optimization

Performs quasi newton optimization. Interfaces many functionalities of the frankestein code originally written by Hong-Zhou Ye

Methods

__init__(func, x0, f0, J0[, trust, max_space])
get_Bnfn(n)

continues on next page

Table 85 – continued from previous page

next_step(iter[, trust_region])

quemb.shared.external.optqn.FrankQN.__init__

FrankQN.**__init__**(*func*, *x0*, *f0*, *J0*, *trust=0.5*, *max_space=500*)

quemb.shared.external.optqn.FrankQN.get_Bnfn

FrankQN.**get_Bnfn**(*n*)

quemb.shared.external.optqn.FrankQN.next_step

FrankQN.**next_step**(*iter*, *trust_region=False*)

quemb.shared.external.uccsd_eri

Functions

frank_get_fock(mycc, vhf, frozen)
frank_get_veff(dm, Vss, Vos)
make_eris_incore(mycc, Vss, Vos[, mo_coeff, ...])

quemb.shared.external.uccsd_eri.frank_get_fock

quemb.shared.external.uccsd_eri.**frank_get_fock**(*mycc*, *vhf*, *frozen*)

quemb.shared.external.uccsd_eri.frank_get_veff

quemb.shared.external.uccsd_eri.**frank_get_veff**(*dm*, *Vss*, *Vos*)

quemb.shared.external.uccsd_eri.make_eris_incore

quemb.shared.external.uccsd_eri.**make_eris_incore**(*mycc*, *Vss*, *Vos*, *mo_coeff=None*, *ao2mofn=None*, *frozen=False*)

quemb.shared.external.unrestricted_utils

Functions

make_uhf_obj(fobj_a, fobj_b[, frozen])	Constructs UHF object from the alpha and beta components
restore_eri_gen(targetsym, eri, norb1, norb2)	An extension of PySCF's ao2mo.restore to Vaabb.
uccsd_restore_eris(symm, fobj_a, fobj_b)	restore ERIs in the correct spin spaces

quemb.shared.external.unrestricted_utils.make_uhf_obj

`quemb.shared.external.unrestricted_utils.make_uhf_obj(fobj_a, fobj_b, frozen=False)`

Constructs UHF object from the alpha and beta components

quemb.shared.external.unrestricted_utils.restore_eri_gen

`quemb.shared.external.unrestricted_utils.restore_eri_gen(targetsym, eri, norb1, norb2)`

An extension of PySCF's ao2mo.restore to Vaabb.

quemb.shared.external.unrestricted_utils.uccsd_restore_eris

`quemb.shared.external.unrestricted_utils.uccsd_restore_eris(symm, fobj_a, fobj_b)`

restore ERIs in the correct spin spaces

quemb.shared.helper

Functions

add_docstring(doc)	Add a docstring to a function as decorator.
add_init_docstring(obj)	Add a sensible docstring to the __init__ method of an attrs class
argsort(seq[, key])	Returns the index that sorts a sequence.
clean_overlap(M[, epsilon])	We assume that M is a (not necessarily square) overlap matrix between ortho-normal vectors.
clear_directory(path)	Clean the **contents** of a directory, including subdirectories, but not the directory itself.
copy_docstring(f)	Copy docstring from another function as decorator.
delete_multiple_files(*args)	
ensure(condition[, message])	This function can be used instead of `assert`, if the test should be always executed.
gauss_sum(n)	Return the sum $\sum_{i=1}^{n} i$
get_calling_function_name()	Do stack inspection shenanigan to obtain the name of the calling function
get_flexible_n_eri(p_max, q_max, r_max, s_max)	Return the number of unique ERIs but allowing different number of orbitals.
jitclass([cls_or_spec, spec])	Decorator to make a class jit-able.
n_eri(n)	
n_symmetric(n)	The number if symmetry-equivalent pairs i <= j, for i <= n and j <= n
*ncore_*(z)	
njit([f])	Type-safe jit wrapper that caches the compiled function
normalize_column_signs(arr[, epsilon])	Divide each column by the sign of its first non-zero entry (if any).
ravel_C(a, b, n_cols)	Flatten the index a, b assuming row-mayor/C indexing
ravel_Fortran(a, b, n_rows)	Flatten the index a, b assuming column-mayor/Fortran indexing

continues on next page

Table 88 – continued from previous page

ravel_eri_idx(a, b, c, d)	Return compound index given four indices using Yoshimine sort and assuming 8-fold permutational symmetry
ravel_symmetric(a, b)	Flatten the index a, b assuming symmetry.
symmetric_different_size(m, n)	Return the number of unique elements in a symmetric matrix of different row and column length
union_of_seqs(*seqs)	Merge multiple sequences into a single `OrderedSet`.
unravel_eri_idx(i)	Invert *ravel_eri_idx()*
unravel_symmetric(i)	
unused(*args)	

quemb.shared.helper.add_docstring

`quemb.shared.helper.`**`add_docstring`**`(`*doc*`)`

Add a docstring to a function as decorator.

Is useful for programmatically generating docstrings.

Parameters
doc (*str*) – A docstring.

Return type
`Callable[[TypeVar(_Function, bound= Callable)], TypeVar(_Function, bound= Callable)]`

Example

```
>>> @add_docstring("Returns 'asdf'")
>>> def f():
>>>     return 'asdf'
is equivalent to
>>> def f():
>>>     "Returns 'asdf'"
>>>     return 'asdf'
```

quemb.shared.helper.add_init_docstring

`quemb.shared.helper.`**`add_init_docstring`**`(`*obj*`)`

Add a sensible docstring to the __init__ method of an attrs class

Makes only sense if the attributes are type-annotated. Is a stopgap measure until https://github.com/sphinx-doc/sphinx/issues/10682 is solved.

Return type
`type`

quemb.shared.helper.argsort

`quemb.shared.helper.`**`argsort`**`(`*seq*, *key=None*`)`

Returns the index that sorts a sequence.

Parameters

- **seq** (Sequence[TypeVar(_T_comparable, bound= *SupportsRichComparison*)]) – The sequence to be sorted.
- **key** (Callable[[TypeVar(_T_comparable, bound= *SupportsRichComparison*)], *SupportsRichComparison*] | None) – Apply function before comparing. Behaves exactly as the same argument to sorted().

Return type
list[int]

quemb.shared.helper.clean_overlap

quemb.shared.helper.**clean_overlap**(*M*, *epsilon=1e-12*)

We assume that M is a (not necessarily square) overlap matrix between ortho-normal vectors. We clean for floating point noise and return an integer matrix with only 0s and 1s.

Return type
ndarray[tuple[int, ...], dtype[int64]]

quemb.shared.helper.clear_directory

quemb.shared.helper.**clear_directory**(*path*)

Clean the **contents** of a directory, including subdirectories, but not the directory itself.

Return type
None

quemb.shared.helper.copy_docstring

quemb.shared.helper.**copy_docstring**(*f*)

Copy docstring from another function as decorator.

Return type
Callable[[TypeVar(_Function, bound= Callable)], TypeVar(_Function, bound= Callable)]

quemb.shared.helper.delete_multiple_files

quemb.shared.helper.**delete_multiple_files**(**args*)

Return type
None

quemb.shared.helper.ensure

quemb.shared.helper.**ensure**(*condition*, *message=''*)

This function can be used instead of **assert**, if the test should be always executed.

Return type
None

quemb.shared.helper.gauss_sum

quemb.shared.helper.**gauss_sum**(*n*)

Return the sum $\sum_{i=1}^{n} i$

Parameters
: **n** (TypeVar(_T_Integral, bound= int | integer))

Return type
: TypeVar(_T_Integral, bound= int | integer)

quemb.shared.helper.get_calling_function_name

quemb.shared.helper.**get_calling_function_name**()

Do stack inspection shenanigan to obtain the name of the calling function

Return type
: str

quemb.shared.helper.get_flexible_n_eri

quemb.shared.helper.**get_flexible_n_eri**(*p_max*, *q_max*, *r_max*, *s_max*)

Return the number of unique ERIs but allowing different number of orbitals.

This is for example the situation for a tuple μ, ν, κ, i, where μ, ν, κ are AOs and i is a fragment orbital. This function returns the number of unique ERIs $g_{\mu,\nu,\kappa,i}$.

Parameters

- **p_max** (TypeVar(_T_Integral, bound= int | integer))
- **q_max** (TypeVar(_T_Integral, bound= int | integer))
- **r_max** (TypeVar(_T_Integral, bound= int | integer))
- **s_max** (TypeVar(_T_Integral, bound= int | integer))

Return type
: TypeVar(_T_Integral, bound= int | integer)

quemb.shared.helper.jitclass

quemb.shared.helper.**jitclass**(*cls_or_spec=None*, *spec=None*)

Decorator to make a class jit-able.

The rationale is the same as for *njit()*, and described there.

For a more detailed explanation of numba jitclasses, see https://numba.readthedocs.io/en/stable/user/jitclass.html

Return type
: TypeVar(T) | Callable[[TypeVar(T)], TypeVar(T)]

quemb.shared.helper.n_eri

quemb.shared.helper.**n_eri**(*n*)

Return type
: TypeVar(_T_Integral, bound= int | integer)

quemb.shared.helper.n_symmetric

quemb.shared.helper.**n_symmetric**(*n*)

The number if symmetry-equivalent pairs i <= j, for i <= n and j <= n

Return type

TypeVar(_T_Integral, bound= int | integer)

quemb.shared.helper.ncore_

quemb.shared.helper.**ncore_**(*z*)

Return type

int

quemb.shared.helper.njit

quemb.shared.helper.**njit**(*f=None*, *, *nogil*, ***kwargs*)

Type-safe jit wrapper that caches the compiled function

With this jit wrapper, you can actually use static typing together with numba. The crucial declaration is that the decorated function's interface is preserved, i.e. mapping `Function` to `Function`. Otherwise the following example would not raise a type error:

```
@numba.njit
def f(x: int) -> int:
    return x

f(2.0)   # No type error
```

While the same example, using this custom `njit()` would raise a type error.

In addition to type safety, this wrapper also sets `cache=True` by default.

Return type

TypeVar(_Function, bound= Callable) | Callable[[TypeVar(_Function, bound= Callable)], TypeVar(_Function, bound= Callable)]

quemb.shared.helper.normalize_column_signs

quemb.shared.helper.**normalize_column_signs**(*arr*, *epsilon=1e-05*)

Divide each column by the sign of its first non-zero entry (if any).

Can be used to compare two MO matrices for (near-)equality, because it fixes the sign factor for both.

Parameters

arr (ndarray[tuple[int, ...], dtype[floating]]) – A 2D numpy array.

Return type

ndarray[tuple[int, ...], dtype[float64]]

Returns

New array with columns divided by the sign of their first non-zero element.

quemb.shared.helper.ravel_C

`quemb.shared.helper.`**`ravel_C`**`(`*a*, *b*, *n_cols*`)`

Flatten the index a, b assuming row-mayor/C indexing

The resulting indexation for a 3 by 4 matrix looks like this:

```
0   1   2   3
4   5   6   7
8   9  10  11
```

Parameters

- **a** (TypeVar(_T_Integral, bound= int | integer))
- **b** (TypeVar(_T_Integral, bound= int | integer))
- **n_cols** (TypeVar(_T_Integral, bound= int | integer))

Return type
TypeVar(_T_Integral, bound= int | integer)

quemb.shared.helper.ravel_Fortran

`quemb.shared.helper.`**`ravel_Fortran`**`(`*a*, *b*, *n_rows*`)`

Flatten the index a, b assuming column-mayor/Fortran indexing

The resulting indexation for a 3 by 4 matrix looks like this:

```
0   3   6   9
1   4   7  10
2   5   8  11
```

Parameters

- **a** (TypeVar(_T_Integral, bound= int | integer))
- **b** (TypeVar(_T_Integral, bound= int | integer))
- **n_rows** (TypeVar(_T_Integral, bound= int | integer))

Return type
TypeVar(_T_Integral, bound= int | integer)

quemb.shared.helper.ravel_eri_idx

`quemb.shared.helper.`**`ravel_eri_idx`**`(`*a*, *b*, *c*, *d*`)`

Return compound index given four indices using Yoshimine sort and assuming 8-fold permutational symmetry

Return type
TypeVar(_T_Integral, bound= int | integer)

quemb.shared.helper.ravel_symmetric

`quemb.shared.helper.`**`ravel_symmetric`**`(`*a*, *b*`)`

Flatten the index a, b assuming symmetry.

The resulting indexation for a matrix looks like this:

```
0
1   2
3   4   5
6   7   8   9
```

Parameters

- **a** (TypeVar(_T_Integral, bound= int | integer))
- **b** (TypeVar(_T_Integral, bound= int | integer))

Return type
TypeVar(_T_Integral, bound= int | integer)

quemb.shared.helper.symmetric_different_size

quemb.shared.helper.**symmetric_different_size**(*m*, *n*)

Return the number of unique elements in a symmetric matrix of different row and column length

This is for example the situation for pairs μ, i where μ is an AO and i is a fragment orbital.

The assumed structure of the symmetric matrix is:

```
*   *   *   *
*   *   *   *
*   *   0   0
*   *   0   0
```

where the stars denote non-zero elements.

Parameters

- **m** (TypeVar(_T_Integral, bound= int | integer))
- **n** (TypeVar(_T_Integral, bound= int | integer))

Return type
TypeVar(_T_Integral, bound= int | integer)

quemb.shared.helper.union_of_seqs

quemb.shared.helper.**union_of_seqs**(**seqs*)

Merge multiple sequences into a single OrderedSet.

This preserves the order of the elements in each sequence, and of the arguments to this function, but removes duplicates. (Always the first occurrence of an element is kept.)

```
merge_seq([1, 2], [2, 3], [1, 4]) -> OrderedSet([1, 2, 3, 4])
```

Return type
OrderedSet[TypeVar(T)]

quemb.shared.helper.unravel_eri_idx

quemb.shared.helper.**unravel_eri_idx**(*i*)

Invert *ravel_eri_idx()*

Return type
`tuple[int, int, int, int]`

quemb.shared.helper.unravel_symmetric

quemb.shared.helper.**unravel_symmetric**(*i*)

Return type
`tuple[int, int]`

quemb.shared.helper.unused

quemb.shared.helper.**unused**(**args*)

Return type
`None`

Classes

FunctionTimer([stats])	
Timer([message])	Simple class to time code execution

quemb.shared.helper.FunctionTimer

class quemb.shared.helper.**FunctionTimer**(*stats=NOTHING*)

Attributes

stats: `dict`

Methods

__init__([stats])	Method generated by attrs for class FunctionTimer.
print_top([n])	Print the top-n functions by total accumulated time.
reset()	
timeit(func)	Decorator to time a function and record stats using Timer.

quemb.shared.helper.FunctionTimer.__init__

FunctionTimer.**__init__**(*stats=NOTHING*)

Method generated by attrs for class FunctionTimer.

quemb.shared.helper.FunctionTimer.print_top

FunctionTimer.**print_top**(*n=10*)

Print the top-n functions by total accumulated time.

Return type
`None`

quemb.shared.helper.FunctionTimer.reset

FunctionTimer.**reset**()

Return type
`None`

quemb.shared.helper.FunctionTimer.timeit

FunctionTimer.**timeit**(*func*)

Decorator to time a function and record stats using Timer.

Return type
`Callable`[[`ParamSpec`(_P, bound= `None`)], `TypeVar`(_R)]

quemb.shared.helper.Timer

class quemb.shared.helper.**Timer**(*message='Elapsed time'*)

Simple class to time code execution

Attributes

message: `str`

start: `float`

Methods

__init__([message])	Method generated by attrs for class Timer.
elapsed()	
str_elapsed([message])	

quemb.shared.helper.Timer.__init__

Timer.**__init__**(*message='Elapsed time'*)

Method generated by attrs for class Timer.

quemb.shared.helper.Timer.elapsed

Timer.**elapsed**()

Return type
`float`

quemb.shared.helper.Timer.str_elapsed

Timer.**str_elapsed**(*message=None*)

Return type
`str`

quemb.shared.io

Functions

write_cube(be_object, cube_file_path, *[, ...])	Write cube files of embedding orbitals from a BE object.

quemb.shared.io.write_cube

`quemb.shared.io.`**`write_cube`**(*be_object*, *cube_file_path*, *, *fragment_idx=None*, *orbital_idx=None*, *cubegen_kwargs=None*)

Write cube files of embedding orbitals from a BE object.

Parameters

- **`be_object`** (*BE*) – BE object containing the fragments, each of which contains embedding orbitals.
- **`cube_file_path`** (`str` | `PathLike`) – Directory to write the cube files to. Directory is created, if it does not exist yet.
- **`fragment_idx`** (`Sequence[int]` | `None`) – Index of the fragments to write the cube files for. If None, write all fragments.
- **`orbital_idx`** (`Sequence[int]` | `None`) – Index of the orbitals to write. If None, writes all (per) fragment.
- **`cubegen_kwargs`** (`dict[str, Any]` | `None`) – Keyword arguments passed to cubegen.orbital.

Return type

`None`

quemb.shared.manage_scratch

Classes

WorkDir(path[, cleanup_at_end, ensure_empty])	Manage a scratch area.

quemb.shared.manage_scratch.WorkDir

`class` `quemb.shared.manage_scratch.`**`WorkDir`**(*path*, *cleanup_at_end=True*, *ensure_empty=False*)

Manage a scratch area.

Upon initialisation of the object the workdir `path` is created, if it does not exist yet. One can either enter a `path` themselves, or if it is `None`, then the path is automatically determined by the environment, i.e. are we on SLURM, where is the scratch etc. Read more at *`from_environment()`* for more details.

Note that the / is overloaded for this class and it can be used as `pathlib.Path` in that regard, see example below.

If `cleanup_at_end` is true, then *`cleanup()`* method is registered to be called when the program terminates with no errors and deletes the scratch directory. The `WorkDir` also exists as a ContextManager; then the cleanup is performed when leaving the ContextManager. Again, assuming that `cleanup_at_end` is true.

Examples

```
>>> with WorkDir('./test_dir', cleanup_at_end=True) as scratch:
>>>     with open(scratch / 'test.txt', 'w') as f:
>>>         f.write('hello world')
'./test_dir' does not exist anymore, if the outer contextmanager is left
without errors.
```

Attributes

path: `Final[Annotated[Path]]`

cleanup_at_end: `Final[bool]`

ensure_empty: `Final[bool]`

Methods

__init__(path[, cleanup_at_end, ensure_empty])	Method generated by attrs for class WorkDir.
cleanup([ignore_error])	Conditionally cleanup the working directory.
from_environment(*[, user_defined_root, ...])	Create a WorkDir based on the environment.

quemb.shared.manage_scratch.WorkDir.__init__

`WorkDir.`**`__init__`**(*path*, *cleanup_at_end=True*, *ensure_empty=False*)

Method generated by attrs for class WorkDir.

quemb.shared.manage_scratch.WorkDir.cleanup

`WorkDir.`**`cleanup`**(*ignore_error=False*)

Conditionally cleanup the working directory.

Parameters

`ignore_errors` – Ignore `FileNotFoundError`, and only that exception, when deleting.

Return type

`None`

quemb.shared.manage_scratch.WorkDir.from_environment

`classmethod` `WorkDir.`**`from_environment`**(*, *user_defined_root=None*, *prefix=None*, *cleanup_at_end=True*, *ensure_empty=False*)

Create a WorkDir based on the environment.

The naming scheme is `f"{user_defined_root}/{prefix}{SLURM_JOB_ID}"` on systems with `SLURM`. If `SLURM` is not available, then the process ID is used instead.

Parameters

- **`user_defined_root`** (`str` | `PathLike` | `None`) – The root directory where to create temporary directories e.g. `/tmp` or `/scratch`. If `None`, then the `SCRATCH_ROOT` value from *`quemb.shared.config.Settings`* is taken.
- **`prefix`** (`str` | `None`) – The prefix for the subdirectory.
- **`cleanup_at_end`** (`bool`) – Perform cleanup when calling `self.cleanup`.

- **ensure_empty** (bool) – Delete the contents of the directory, if it already exists.

Return type
`Self`

quemb.shared.numba_helpers

Functions

extend_with(D_1, D_2)	Extend dictionary with values from `D_2`
merge_dictionaries(dictionaries)	Merge a sequence of numba dictionaries
to_numba_dict(py_dict)	

quemb.shared.numba_helpers.extend_with

`quemb.shared.numba_helpers.`**`extend_with`**`(`*D_1*, *D_2*`)`

Extend dictionary with values from `D_2`

Works inplace on `D_1`.

Return type
`None`

Parameter

D_1 : D_2 :

quemb.shared.numba_helpers.merge_dictionaries

`quemb.shared.numba_helpers.`**`merge_dictionaries`**`(`*dictionaries*`)`

Merge a sequence of numba dictionaries

Return type
`Dict`

Parameter

dictionaries :

quemb.shared.numba_helpers.to_numba_dict

`quemb.shared.numba_helpers.`**`to_numba_dict`**`(`*py_dict*`)`

Return type
`Dict[TypeVar(Key, bound= Hashable), TypeVar(Val)]`

Classes

PreIncr()	
SortedIntSet()	A sorted set implementation using Numba's jitclass.

quemb.shared.numba_helpers.PreIncr

class quemb.shared.numba_helpers.**PreIncr**

Attributes

class_type = jitclass.PreIncr#153991eb0<count:int64>

count: int

Implement a simple pre-incr class, that increments and returns a value. more or less equivalent to C's `++i`. Starts with -1, i.e. the first returned value is 0.

Methods

__init__()	
incr()	

quemb.shared.numba_helpers.PreIncr.__init__

PreIncr.**__init__**()

quemb.shared.numba_helpers.PreIncr.incr

PreIncr.**incr**()

quemb.shared.numba_helpers.SortedIntSet

class quemb.shared.numba_helpers.**SortedIntSet**

A sorted set implementation using Numba's jitclass.

This class maintains a set of unique integers in sorted order. Internally, it uses a dictionary for fast membership checks and a list for ordered storage of elements. All operations are JIT-compiled for high performance in numerical code.

_lookup

A dictionary mapping integers to booleans, used for fast membership testing and ensuring uniqueness.

Type

numba.typed.Dict

items

A list of sorted, unique integers representing the elements of the set in ascending order.

Type

numba.typed.List

Examples

```
>>> s = SortedIntSet()
>>> s.add(3)
>>> s.add(1)
>>> s.add(2)
```

(continues on next page)

(continued from previous page)

```
>>> s.items
[1, 2, 3]
```

```
>>> 2 in s
True
```

```
>>> s.remove(2)
>>> s.items
[1, 3]
```

Attributes

class_type = jitclass.SortedIntSet#152282b10<_lookup:DictType[int64,bool]<iv=None>, items:ListType[int64]>

Methods

__init__()
add(val)
remove(val)

quemb.shared.numba_helpers.SortedIntSet.__init__

SortedIntSet.**__init__**()

quemb.shared.numba_helpers.SortedIntSet.add

SortedIntSet.**add**(*val*)

quemb.shared.numba_helpers.SortedIntSet.remove

SortedIntSet.**remove**(*val*)

quemb.shared.typing

Define some types that do not fit into one particular module

In particular it enables barebone typechecking for the shape of numpy arrays

Inspired by https://stackoverflow.com/questions/75495212/type-hinting-numpy-arrays-and-batches

Note that most numpy functions return `ndarray[Any, Any]` i.e. the type is mostly useful to document intent to the developer.

Module Attributes

T_dtype_co	Type annotation of a generic covariant type.
Vector	Type annotation of a vector.
Matrix	Type annotation of a matrix.
Tensor3D	Type annotation of a tensor.
Tensor4D	Type annotation of a tensor.
Tensor5D	Type annotation of a tensor.
Tensor6D	Type annotation of a tensor.
Tensor7D	Type annotation of a tensor.
Tensor	Type annotation of a tensor.
PathLike	Type annotation for pathlike objects.
KwargDict	Type annotation for dictionaries holding keyword arguments.
T	A generic type variable, without any constraints.
OrbitalIdx	The index of an orbital.
AOIdx	The index of an atomic orbital.
GlobalAOIdx	The global index of an atomic orbital, i.e. not per fragment.
RelAOIdx	The relative AO index.
RelAOIdxInRef	The relative AO index, relative to the reference fragment.
MOIdx	The index of a molecular orbital.
GlobalMOIdx	The global index of a molecular orbital, i.e. not per fragment.
ShellIdx	The shell index ...
FragmentIdx	The index of a Fragment.
MotifIdx	The index of a heavy atom, i.e. of a motif.
CenterIdx	In a given fragment, this is the index of a center.
EdgeIdx	An edge is the complement of the set of centers in a fragment.
OriginIdx	In a given BE fragment, this is the origin of the remaining fragment after subsets have been removed.

quemb.shared.typing.T_dtype_co

quemb.shared.typing.**T_dtype_co** = **+T_dtype_co**

Type annotation of a generic covariant type.

quemb.shared.typing.Vector

quemb.shared.typing.**Vector**

Type annotation of a vector.

alias of ndarray[tuple[int], dtype[*T_dtype_co*]]

quemb.shared.typing.Matrix

quemb.shared.typing.**Matrix**

Type annotation of a matrix.

alias of ndarray[tuple[int, ...], dtype[*T_dtype_co*]]

quemb.shared.typing.Tensor3D

`quemb.shared.typing.`**`Tensor3D`**

Type annotation of a tensor.

alias of `ndarray`[`tuple`[`int`, …], `dtype`[*`T_dtype_co`*]]

quemb.shared.typing.Tensor4D

`quemb.shared.typing.`**`Tensor4D`**

Type annotation of a tensor.

alias of `ndarray`[`tuple`[`int`, …], `dtype`[*`T_dtype_co`*]]

quemb.shared.typing.Tensor5D

`quemb.shared.typing.`**`Tensor5D`**

Type annotation of a tensor.

alias of `ndarray`[`tuple`[`int`, …], `dtype`[*`T_dtype_co`*]]

quemb.shared.typing.Tensor6D

`quemb.shared.typing.`**`Tensor6D`**

Type annotation of a tensor.

alias of `ndarray`[`tuple`[`int`, …], `dtype`[*`T_dtype_co`*]]

quemb.shared.typing.Tensor7D

`quemb.shared.typing.`**`Tensor7D`**

Type annotation of a tensor.

alias of `ndarray`[`tuple`[`int`, …], `dtype`[*`T_dtype_co`*]]

quemb.shared.typing.Tensor

`quemb.shared.typing.`**`Tensor`**

Type annotation of a tensor.

alias of `ndarray`[`tuple`[`int`, …], `dtype`[*`T_dtype_co`*]]

quemb.shared.typing.PathLike

`quemb.shared.typing.`**`PathLike:  TypeAlias = str | os.PathLike`**

Type annotation for pathlike objects.

quemb.shared.typing.KwargDict

`quemb.shared.typing.`**`KwargDict`**

Type annotation for dictionaries holding keyword arguments.

alias of `dict`[`str`, `Any`]

quemb.shared.typing.T

`quemb.shared.typing.T = ~T`

A generic type variable, without any constraints.

quemb.shared.typing.OrbitalIdx

`quemb.shared.typing.OrbitalIdx = quemb.shared.typing.OrbitalIdx`

The index of an orbital.

quemb.shared.typing.AOIdx

`quemb.shared.typing.AOIdx = quemb.shared.typing.AOIdx`

The index of an atomic orbital.

quemb.shared.typing.GlobalAOIdx

`quemb.shared.typing.GlobalAOIdx = quemb.shared.typing.GlobalAOIdx`

The global index of an atomic orbital, i.e. not per fragment. This is basically the result of the *func:pyscf.gto.mole.Mole.aoslice_by_atom* method.

quemb.shared.typing.RelAOIdx

`quemb.shared.typing.RelAOIdx = quemb.shared.typing.RelAOIdx`

The relative AO index. This is relative to the own fragment.

quemb.shared.typing.RelAOIdxInRef

`quemb.shared.typing.RelAOIdxInRef = quemb.shared.typing.RelAOIdxInRef`

The relative AO index, relative to the reference fragment. For example for an edge in fragment 1 it is the AO index of the same atom interpreted as center in fragment 2.

quemb.shared.typing.MOIdx

`quemb.shared.typing.MOIdx = quemb.shared.typing.MOIdx`

The index of a molecular orbital.

quemb.shared.typing.GlobalMOIdx

`quemb.shared.typing.GlobalMOIdx = quemb.shared.typing.GlobalMOIdx`

The global index of a molecular orbital, i.e. not per fragment.

quemb.shared.typing.ShellIdx

`quemb.shared.typing.ShellIdx = quemb.shared.typing.ShellIdx`

The shell index …

quemb.shared.typing.FragmentIdx

`quemb.shared.typing.FragmentIdx = quemb.shared.typing.FragmentIdx`

The index of a Fragment.

quemb.shared.typing.MotifIdx

`quemb.shared.typing.`**`MotifIdx = quemb.shared.typing.MotifIdx`**

The index of a heavy atom, i.e. of a motif. If hydrogen atoms are not treated differently, then every atom is a motif, and this type is equivalent to `AtomIdx`.

quemb.shared.typing.CenterIdx

`quemb.shared.typing.`**`CenterIdx = quemb.shared.typing.CenterIdx`**

In a given fragment, this is the index of a center. A center was used to generate a fragment around it. Since a fragment can swallow other smaller fragments, there is only one origin per fragment but multiple centers, which are the origins of the swallowed fragments.

quemb.shared.typing.EdgeIdx

`quemb.shared.typing.`**`EdgeIdx = quemb.shared.typing.EdgeIdx`**

An edge is the complement of the set of centers in a fragment.

quemb.shared.typing.OriginIdx

`quemb.shared.typing.`**`OriginIdx = quemb.shared.typing.OriginIdx`**

In a given BE fragment, this is the origin of the remaining fragment after subsets have been removed. Since a fragment can swallow other smaller fragments, there is only one origin per fragment but multiple centers, which are the origins of the swallowed fragments.

In the following example, we have drawn two fragments, one around atom A and one around atom B. The fragment around atom A is completely contained in the fragment around B, hence we remove it. The remaining fragment around B has the origin B, and swallowed the fragment around A. Hence its centers are A and B. The set of edges is the complement of the set of centers, hence in this case for the fragment around B the set of edges is the one-element set {C}.

```
__________
|        |  BE2 fragment around A
|        |
___________________
|        |        |   BE2 fragment around B
|        |        |
A ------ B ------ C ------ D
|        |        |        |
```

Classes

SupportsRichComparison(*args, **kwargs)	

quemb.shared.typing.SupportsRichComparison

class `quemb.shared.typing.`**`SupportsRichComparison`**(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	

quemb.shared.typing.SupportsRichComparison.__init__

`SupportsRichComparison.`**`__init__`**`(`**args*, ***kwargs*`)`

PYTHON MODULE INDEX

INDEX