pyopencap Documentation

pyopencap

Apr 10, 2023

CONTENTS

PyOpenCAP is the Python API for OpenCAP, an open-source software aimed at extending the functionality of quantum chemistry packages to describe resonances. PyOpenCAP uses the pybind11 library to expose C++ classes and methods, allowing calculations to be driven within a Python interpreter.

PyOpenCAP is currently capable of processing quantum chemistry data in order to perform 'projected' complex absorbing potential calculations on metastable electronic states. These calculations are able to extract resonance position and width at the cost of a single bound-state electronic structure calculation.

To get started, please see our *Getting Started* page.

If you have questions or need support, please open an issue on GitHub, or contact us directly at gayverjr@bu.edu.

PyOpenCAP is released under the MIT license.

CHAPTER

ONE

SUPPORTED PACKAGES

- OpenMolcas
- PySCF
- Q-Chem
- Psi4
- Columbus

CHAPTER

TWO

SUPPORTED METHODS

- EOM-CC
- FCI
- MS-CASPT2 (and extended variants)
- TDDFT
- ADC (through ADCC)
- MR-CI family of methods

CHAPTER

THREE

SUPPORTED POTENTIALS

- Box (analytical integrals are now available!)
- Smooth Voronoi
- *Custom* CAP functions

Please see the *keywords* section for more details.

CHAPTER

FOUR

OTHER FEATURES

- Python based eigenvalue trajectory *analysis* tools
- *Custom* numerical integration grids

CHAPTER

FIVE

IN DEVELOPMENT

- Feshbach projection approaches
- Tools for dyanmics on complex potential energy surfaces

CHAPTER

SIX

ACKNOWLEDGEMENTS

- We would like to give special thanks to Prof. Thomas Sommerfeld for generously providing us with a C++ implementation of analytical box CAP integrals. Please checkout his repository on GitHub for many implementations of L^2 methods for resonances!
- We would like to thank Prof. John Burkardt for his C++ implementation of the Incomplete Gamma function. Many of his wonderful open source codes can be found at his website.
- We would like to thank the Molecular Sciences Software Institute for funding this project. Please visit the MolSSI website for their educational resources, fellowship opportunities, and software projects!
- We would like to thank Mushir Thodika from Temple University for his help in developing the interface with Columbus.

CHAPTER

SEVEN

CONTENTS

7.1 Installation

7.1.1 Install with pip (recommended)

```
pip install pyopencap
# or
pip3 install pyopencap
```

Precompiled Python wheels are available on Pypi for almost all Linux systems and most MacOS systems, for Python versions 3.6 and later.

7.1.2 Build from source

Dependencies

Compiling PyOpenCAP from source requires first installing the following dependencies:

- C++ compiler with full C++17 language support and standard libraries
- Python3 interpreter and development libraries: version >= 3.6
- CMake: version >= 3.12
- HDF5: hierarchical data format, version >= 1.10
- Eigen: linear algebra library, version >= 3.3

All of these dependencies are available through standard package managers such as Homebrew, Conda, and yum/apt-get on Linux.

Compiler

For Mac/Linux users, any compiler which fully supports the C++17 standard should work (e.g GCC 7.x or later). If you are unsure, try updating to the latest version of your compiler.

Building the package

If your operating system/Python environment is not covered by any of our pre-built wheels, the command `pip install pyopencap` will download the tarball from Pypi and try to compile from source. You can also clone the repository and install a local version:

```
git clone https://github.com/gayverjr/opencap.git

cd opencap

pip install .
```

Compiling from source will take several minutes. To monitor your progress, you can run pip with the *–verbose* flag.

To ensure that the installation was successful, return to your home directory, start a Python shell, and type:

```
import pyopencap
```

If you cloned the repository, you can run the tests by entering the *pyopencap* directory, and running `pytest`.

7.2 Getting Started

Constructing the System object

The *System* object of PyOpenCAP contains the geometry and basis set information, as well as the overlap matrix. The constructor takes in a Python dictionary as an argument, and understands a specific set of keywords . There are four equivalent ways of specifying the geometry and basis set: qchem_fchk, rassi_h5, molden, and inline. Here, we'll use the rassi_h5 file.

```
sys_dict = {"molecule": "rassi_h5","basis_file": "path/to/rassi/file.h5"}
s = pyopencap.System(sys_dict)
smat = s.get_overlap_mat()
```

Constructing the CAP object

The CAP matrix is computed by the *CAP* object. The constructor requires a *System* object, a dictionary containing the CAP parameters, and the number of states.

```
nstates = 10
cap_dict = {"cap_type": "box",
        "cap_x":"2.76",
        "cap_y":"2.76",
        "cap_z":"4.88",
        "Radial_precision": "14",
        "angular_points": "110"}
pc = pyopencap.CAP(s,cap_dict,nstates)
```

Parsing electronic structure data from file

The *read_data()* function can read in the effective Hamiltonian and densities in one-shot when passed a Python dictionary with the right keywords.

```
es_dict = {"method" : "ms-caspt2",
           "package": "openmolcas",
  "molcas_output":"path/to/output/file.out",
  "rassi_h5":  "path/to/rassi/file.h5"}
pc.read_data(es_dict)
h0 = pc.get_H()
```

Passing densities in RAM

Alternatively, one can load in the densities one at a time using the *add_tdms()* or *add_tdm()* functions. The density matrices should be in atomic orbital basis, with the same atomic orbital ordering as the *System* (which can be verify using `check_overlap_matrix`). The example below shows how one might pass the densities from a PySCF calculation:

```
s.check_overlap_mat(pyscf_smat,"pyscf")
pc = pyopencap.CAP(s,cap_dict,10)
for i in range(0,10):
    for j in range(i,10):
        dm1 = fs.trans_rdm1(fs.ci[i],fs.ci[j],myhf.mo_coeff.shape[1],mol.nelec)
        dm1_ao = np.einsum('pi,ij,qj->pq', myhf.mo_coeff, dm1, myhf.mo_coeff.conj())
        pc.add_tdm(dm1_ao,i,j,"pyscf")
        if i!=j:
            pc.add_tdm(dm1_ao,j,i,"pyscf")
```

Once all of the densities are loaded, the CAP matrix is computed using the *compute_projected_cap()* function. The matrix can be retrieved using the *get_projected_cap()* function.

```
>>> pc.compute_projected_cap()
>>> W_mat=pc.get_projected_cap()
```

We now have our zeroth order Hamiltonian (stored in h0) and our CAP matrix(W_mat) in the state basis. Extracting resonance position and width requires analysis of the eigenvalue trajectories.

Analysis

PyOpenCAP provides user friendly tools for analysis of eigenvalue trajectories.

The *CAPHamiltonian* contains functions aimed at diagonalization of the CAP Hamiltonian over a range of eta values. Assuming one has already obtained H0 and W in the state basis as numpy matrices, it can be constructed as such:

```
from pyopencap.analysis.CAPTrajectory import CAPHamiltonian
eta_list = np.linspace(0,5000,101)
eta_list = np.around(eta_list * 1E-5,decimals=5)
CAPH = CAPHamiltonian(H0=h0,W=mat)
# equivalently
CAPH = CAPHamiltonian(pc=pc)
CAPH.run_trajectory(eta_list,cap_lambda=0.0)
```

One can easily plot the eigenvalue spectrum in au or eV (relative to a given reference energy) as follows:

```
# total energies
plt.plot(np.real(CAPH.total_energies),np.imag(CAPH.total_energies),'ro')
plt.show()
# excitation energies
plt.plot(np.real(CAPH.energies_ev(ref_energy)),np.imag(CAPH.energies_ev(ref_energy)),'ro
↪')
plt.show()
```

To analyze a given trajectory, use *track_state()*

```
traj = CAPH.track_state(1,tracking="overlap")
```

traj is now a *EigenvalueTrajectory* object, which contains helpful functions for analysis. One can plot raw and corrected trajectories:

```
plt.plot(np.real(traj.energies_ev(ref_energy)),np.imag(traj.energies_ev(ref_energy)),'-ro
↪')
plt.plot(np.real(traj.energies_ev(ref_energy,corrected=True)),np.imag(traj.energies_
↪ev(ref_energy,corrected=True)),'-bo')
```

There are also functions to help find the optimal value of the CAP strength parameter (and therefore, best estimate of resonance position and width) for uncorrected/corrected trajectories:

```
uc_energy,uc_eta_opt = traj.find_eta_opt()
corr_energy,corr_eta_opt = traj.find_eta_opt(corrected=True)
```

For more information, please see the documentation for the *CAPHamiltonian* and *EigenvalueTrajectory* classes.

See more

Please see the notebooks in our repository for detailed examples which demonstrate the full functionality of PyOpenCAP.

7.3 Theory

7.3.1 Resonances and Non-Hermitian Quantum Mechanics

Electronic resonances are metastable electronic states with finite lifetimes embedded in the ionization/detachment continuum. Common examples include temporary anions formed by electron attachment, and core-excited and core-ionized states which can undergo Auger decay or similar relaxation pathways. These states are not part of the usual L^2 Hilbert space of square integrable functions, and instead belong to the continuous spectrum of the electronic Hamiltonian. Theoretical description of resonances is generally not possible by means of conventional bound-state quantum chemistry methods, and special techniques are required to obtain accurate energies and lifetimes.

Non-Hermitian quantum mechanics (NHQM) techniques provide an attractive approach that enables adaptation of existing quantum chemistry methodologies to treat metastable electronic states. In NHQM formalisms, a resonance appears as a single square-integrable eigenstate of a non-Hermitian Hamiltonian, associated with a with a complex eigenvalue:

$$E = E_{res} - i\Gamma/2.$$

The real part of the energy (E_{res}) is the resonance position. The imaginary part $(\Gamma/2)$ is the half-width, which is inversely proportional to the lifetime of the state [Reinhardt1982].

7.3.2 Complex Absorbing Potential

Complex absorbing potentials (CAPs) are imaginary potentials added to the Hamiltonian, and they are routinely used for evaluation of resonance parameters. In this context, CAPs transform a resonance into a single square integrable state, rendering it accessible by means of standard bound-state techniques. To this end, the electronic Hamiltonian is augmented with an imaginary potential:

$$H^{CAP} = H - i\eta W$$

where η is the CAP strength parameter, and W is a real potential which vanishes in the vicinity of the molecular system and grows with distance.

Since the CAP-augmented Hamiltonian depends on the strength of the CAP, a choice has to be made on the optimal value of η that provides best estimates of the resonance position and width. In a complete one-electron basis, the exact resonance position and width are obtained in the limit of an infinitesimally weak CAP $(\eta \to 0^+)$. In practice when finite

basis sets are used, an optimal CAP strength η_{opt} is found by locating a stationary point on the eigenvalue trajectory E(η). A commonly used criterion is the minimum of the logarithmic velocity ($|\eta \frac{dE}{d\eta}| \rightarrow min$) [Riss1993].

7.3.3 Projected CAP

There are multiple strategies for how to incorporate CAPs into an electronic structure calculation. The most straightforward implementation is to engage the one-electron CAP term starting at the lowest level of theory (e.g. Hartree-Fock). While conceptually simple, this requires modification of electronic structure routines to handle the complex objects. Additionally, this approach requires a unique calculation for each η along the eigenvalue trajectory, which can become prohibitively expensive for larger systems or dynamical simulation.

An efficient alternative is to treat the CAP as a first order perturbation, considering only a small subset of the eigenstates of the real Hamiltonian [Sommerfeld2001]. In this so called subspace projection scheme, the CAP will be introduced in the correlated basis of the reduced subset of states:

$$W_{uv}^{CB} = \langle u|W|v\rangle$$

where u and v are eigenstates of the real Hamiltonian. Since the CAP is a one-particle operator, these expressions can easily be evaluated using the CAP matrix in atomic orbital basis evaluated separately, the one-electron reduced density matrices (ρ) for each state, and the set of transition density matrices (γ) between each pair of states that are obtained from the bound-state calculation.

$$W_{uv}^{CB} = \begin{Bmatrix} Tr\left[W^{AO}\gamma^{uv}\right], & u \neq v \\ Tr\left[W^{AO}\rho^{u}\right], & u = v \end{Bmatrix}$$

Once CAP matrix is evaluated the CAP-augmented Hamiltonian is constructed as follows:

$$H^{CAP} = H_0 - i\eta W^{CB}$$

where H_0 is an appropriate zeroth order Hamiltonian obtained from the electronic structure calculation, and W^{CB} is the CAP represented in the correlated basis. Diagonalization of this CAP-augmented Hamiltonian yields η-dependent eigenvalues that are used to extract resonance position and width. Importantly, as only a small number of states in considered (typically less than 30), finding the eigenvalues of the CAP-augmented Hamiltonian has negligible cost in comparison to the bound-state electronic structure calculation required to get the initial set of states (u,v,..). Thus, although this perturbative or projected approach introduces another parameter (number of eigenstates), the overall cost is essentially reduced to that of a single electronic structure calculation.

With the zeroth order Hamiltonian and the CAP matrix, eigenvalue trajectories can be generated by means of simple external scripts, and estimates of resonances positions and widths can be obtained from analysis of the trajectories.

7.3.4 Continuum Remover CAP

One of the major challenges of analyzing eigenvalue trajectories is the appearance of of unphysical stationary points. To separate the physical complex energy stabilized points from the non physical ones, Moiseyev and coworkers [Moiseyev2009] have proposed adding an additional artificial real valued potential to the CAP Hamiltonian.

$$H^{CAP} = H_0 + (\lambda - i\eta)W^{CB}$$

This approach is known as continuum remover-CAP or CR-CAP. The idea behind this CR-CAP is that the true resonance wave function is insensitive to such a potential due to its bound-like nature, while artificial stabilization points corresponding to the scattering states would be significantly affected by the additional real valued potential. λ can be positive or negative, and there is usually no need to optimize λ, as it is only used for the purpose of identifying the true resonance stabilization point.

7.3.5 References

7.4 Interfaces

PyOpenCAP officially supports interfaces with the OpenMolcas, Q-Chem, Psi4, Columbus, and PySCF software packages.

7.4.1 OpenMolcas

OpenMolcas is an open-source quantum chemistry package which specializes in multiconfigurational approaches to electronic structure. OpenMolcas can be used in tandem with PyOpenCAP to perform complex absorbing potential (extended)multi-state complete active space second order perturbation theory [**CAP/MS-CASPT2**] calculations, which have been shown to yield accurate energies and lifetimes for metastable electronic states. Here, we outline the steps of performing these calculations using OpenMolcas and PyOpenCAP. Some suggested readings are provided at the bottom of the page.

Step 1: Running OpenMolcas calculation

To generate the one-particle densities required to construct the CAP matrix, the RASSI module must be executed with the TRD1 keyword activated. When using XMS-CASPT2, RMS-CASPT2, or other variants which utilize rotated CASSCF wave functions, the CAP matrix will eventually be rotated into the new basis using the rotation matrix in the output (U^dagger*W*U). RASSI will save transition density matrices between each pair of CASSCF states as well as the one-particle density matrices for each state to a file titled $Jobname.rassi.h5.

Export transition densities with RASSI

```
&RASSI
TRD1
```

Generate effective Hamiltonian with MS-CASPT2

The MS-CASPT2 approach is required to generate an appropriate zeroth Hamiltonian for the projected CAP method. To activate MS-CASPT2 in OpenMolcas, use the Multistate keyword in the CASPT2 module.

```
&CASPT2
Multistate = all
```

See the OpenMolcas manual for other variants of MS-CASPT2 which can be activated in the &CASPT2 section.

Step 2: Importing the data to PyOpenCAP

System object

To run a PyOpenCAP calculation, the geometry and basis set must be imported into a `System` object. The constructor takes in a Python dictionary as an argument. The relevant keywords are discussed here, and more information is provided in the *keywords* page.

Rassi.h5

The rassi.h5 file which contains the one-particle densities also contains the geometry and basis set information. To read in from rassi, “molcas_rassi” must set as the value to the key “molecule”, and the path to the file must be set as the value to the key “basis_file”. Here is an example:

```
sys_dict = {"molecule": "molcas_rassi","basis_file": "path/to/rassi.h5"}
my_system = pyopencap.System(sys_dict)
```

Molden

Molden files generated by OpenMolcas contain the geometry and basis set information. To read in from molden, "molden" must be set as the value to the key "molecule", and the path to the file must be set as the value to the key "basis_file". Here is an example:

```
sys_dict = {"molecule": "molden","basis_file": "path/to/file.molden"}
my_system = pyopencap.System(sys_dict)
```

Inline(not recommended)

The molecule and basis set can also be specified manually. The "molecule" keyword must be set to "read", and then an additional keyword "geometry:" must be specified, with a string that contains the geometry in xyz format. The "basis_file" keyword must be set to a path to a basis set file formatted in Psi4 style, which can be downloaded from the MolSSI BSE. Other optional keyword for this section include "bohr_coordinates" and cart_bf. Please see the *keywords* section for more details. Up to G-type functions are supported.

```
sys_dict = {"geometry":    '''N  0  0   1.039
                           N  0  0   -1.039
                           X   0  0   0.0''',
                    "molecule" : "read",
                    "basis_file":"path/to/basis.bas",
                    "cart_bf":"d",
                    "bohr_coordinates:": "true"}
my_system = pyopencap.System(sys_dict)
```

One particle densities/zeroth order Hamiltonian

The CAP matrix is computed by the *CAP* object. The constructor requires a *System*, a dictionary containing the CAP parameters, the number of states, and finally the string "openmolcas", which denotes the ordering of the atomic orbital basis set. An example is provided below. Please see the keywords section for more information on the CAP parameters.

```
cap_dict = {"cap_type": "box",
            "cap_x":"2.76",
            "cap_y":"2.76",
            "cap_z":"4.88",
            "Radial_precision": "14",
            "angular_points": "110"}
nstates = 10
pc = pyopencap.CAP(my_system,cap_dict,nstates)
```

Before we can compute the CAP matrix in the state basis, we must load in the density matrices. The best way is to use the *read_data()* function. As shown below, we define a dictionary which contains the following keys: "package"(openmolcas), "method" (electronic structure method chosen), "rassi_h5"(density matrices), and "molcas_output"(output file containing effective Hamiltonian). The effective Hamiltonian can be retrieved using the *get_H()* function of the *CAP* object. Currently, only effective Hamiltonians from MS-CASPT2 calculations can be parsed from an OpenMolcas output file.

```
es_dict = { "package": "openmolcas",
       "method" : "ms-caspt2",
       "molcas_output":"path/to/output.out",
       "rassi_h5":"path/to/rassi.h5"}
```

(continues on next page)

(continued from previous page)

```
pc.read_data(es_dict)
# save the effective Hamiltonian for later use
h0 = pc.get_H()
```

Step 3: Computing the CAP matrix

Once all of the densities are loaded, the CAP matrix is computed using *compute_projected_cap()*. The matrix can be retrieved using *get_projected_cap()*.

```
pc.compute_projected_cap()
W_mat=pc.get_projected_cap()
```

Note:

When using cartesian d, f, or g-type basis functions, special care must be taken to ensure that the normalization conventions match what is used by OpenMolcas. In these cases, *compute_ao_cap()* and then *renormalize()* or *renormalize_cap()* should be invoked before calling *compute_projected_cap()*.

```
pc.compute_ao_cap(cap_dict)
pc.renormalize()
pc.compute_projected_cap()
```

Step 4: Generate and analyze eigenvalue trajectories

H0 and W, or the *CAP* object can be used to construct a *CAPHamiltonian* object.

```
from pyopencap.analysis import CAPHamiltonian
CAPH = CAPHamiltonian(H0=H0,W=W_mat)
# equivalently
CAPH = CAPHamiltonian(pc=pc)
```

See the *analysis* section for more details.

Officially supported methods

The following methods have been benchmarked, and the *read_data()* function is capable of parsing output files to obtain the zeroth order Hamiltonian.

- MS-CASPT2, and other variants (e.g. XMS-CASPT2) which utilize unitary rotations of the original CASSCF states. The CAP matrix will be rotated into the new basis using the rotation matrix.

Untested (use at your own risk!)

The following methods are capable of dumping densities using the TRD1 keyword of the RASSI module, but have not been benchmarked for any systems, and the zeroth order Hamiltonian cannot be parsed from the output file using the *read_data()* function. Use at your own caution, and please contact us if you find success using any of these methods so we can add official support!

- (QD)DMRG-(PC/SC)NEVPT2
- SS-CASPT2
- MC-PDFT

Suggested reading

7.4.2 PySCF

PySCF is an ab initio computational chemistry program natively implemented in Python. The major advantage of using Pyscf in tandem with OpenCAP is that calculations can be performed in one-shot within the same python script. Since PySCF allows direct control over data structures such as density matrices, the interface between PySCF and OpenCAP is seamless.

Step 1: Defining the System object

Molden(recommended)

The best way to construct the *System* object is to import the geometry and basis set from a molden file generated by a PySCF. This ensures proper ordering of the AO basis set.

```
molden_dict = {"basis_file":"molden_in.molden","molecule": "molden"}
pyscf.tools.molden.from_scf(myhf,"molden_in.molden")
s = pyopencap.System(molden_dict)
```

Inline

The molecule and basis set can also be specified inline. The "molecule" keyword must be set to "read", and then an additional keyword "geometry" must be specified, with a string that contains the geometry in xyz format. The "basis_file" keyword must be set to a path to a basis set file formatted in Psi4 style, which can be downloaded from the MolSSI BSE. Other optional keyword for this section include "bohr_coordinates" and "cart_bf". Please see the *keywords* section for more details. It is recommended to check the overlap matrix to ensure that the ordering and normalization matches. Up to G-type functions are supported.

```
pyscf_smat = scf.hf.get_ovlp(mol)
sys_dict = {"geometry":    '''N  0  0   1.039
                           N  0  0   -1.039
                           X   0  0   0.0''',
                    "molecule" : "read",
                    "basis_file":"path/to/basis.bas",
                    "cart_bf":"d",
                    "bohr_coordinates:": "true"}
s.check_overlap_mat(pyscf_smat,"pyscf")
```

Step 1: Defining the CAP object

The CAP matrix is computed by the *CAP* object. The constructor requires a *System* object, a dictionary containing the CAP parameters, and the number of states. An example is provided below. Please see the *keywords* section for more information on the CAP parameters.

```
nstates = 10
cap_dict = {"cap_type": "box",
            "cap_x":"2.76",
            "cap_y":"2.76",
            "cap_z":"4.88",
            "Radial_precision": "14",
            "angular_points": "110"}
pc = pyopencap.CAP(my_system,cap_dict,nstates)
```

Step 2: Passing the density matrices

The simplest interface is with the FCI modules. Transition densities can be obtained using the `trans_rdm1()` function:

```
fs = fci.FCI(mol, myhf.mo_coeff)
e, c = fs.kernel()
# tdm between ground and 1st excited states
dm1 = fs.trans_rdm1(fs.ci[0],fs.ci[1],myhf.mo_coeff.shape[1],mol.nelec)
```

Importantly, trans_rdm1 returns the density matrix in **MO basis**. Thus before passing it to PyOpenCAP, it **must be transformed into AO basis**:

```
dm1_ao = np.einsum('pi,ij,qj->pq', myhf.mo_coeff, dm1, myhf.mo_coeff.conj())
```

Densities are loaded in one at a time using *add_tdm()*. Ensure that the indices of each state match those of the zeroth order Hamiltonian.

```
for i in range(0,len(fs.ci)):
    for j in range(0,len(fs.ci)):
        dm1 = fs.trans_rdm1(fs.ci[i],fs.ci[j],myhf.mo_coeff.shape[1],mol.nelec)
        dm1_ao = np.einsum('pi,ij,qj->pq', myhf.mo_coeff, dm1, myhf.mo_coeff.conj())
        pc.add_tdm(dm1_ao,i,j,"pyscf")
```

Note:

The interface with PySCF is not restricted to the FCI module. The *add_tdm()* function is completely general; it requires only that the densities are in AO basis, and that the basis set ordering matches the system. Examples for ADC, EOM-EA-CCSD, and TDA-TDDFT are provided in the repository.

Step 3: Computing the CAP matrix

Once all of the densities are loaded, the CAP matrix is computed using the *compute_projected_cap()* function. The matrix can be retrieved using the *get_projected_cap()* function.

```
pc.compute_projected_cap()
W_mat=pc.get_projected_cap()
```

Note:

When using cartesian d, f, or g-type basis functions, special care must be taken to ensure that the normalization conventions match what is used by OpenMolcas. In these cases, *compute_ao_cap()* and then *renormalize()* or *renormalize_cap()* should be invoked before calling *compute_projected_cap()*.

```
pc.compute_ao_cap(cap_dict)
pc.renormalize_cap(pyscf_smat,"pyscf")
pc.compute_projected_cap()
```

Step 4: Generate and analyze eigenvalue trajectories

H0 and W can be used to construct a `CAPHamiltonian` object. In many cases, it can be advantageous to use the `export()` function, which generates an OpenCAP formatted output file, which can be used for later analysis.

```
from pyopencap.analysis import CAPHamiltonian
CAPH = CAPHamiltonian(H0=H0,W=W_mat)
CAPH.export("output.out")
```

See the *analysis* section for more details.

Officially supported methods

- Full CI
- ADC (through ADCC)
- TDA-TDDFT

Untested (use at your own risk!)

Any module which one particle transition densities available can be supported. This includes all methods which can utilize the trans_rdm1 function, including but not limited to:

- MRPT

7.4.3 QChem

PyOpenCAP supports an interface with the Q-Chem quantum chemistry package.

Importing data

System object

The geometry and basis set can be imported into a *System* object using .fchk files.

```
import pyopencap
sys_dict = {"molecule": "qchem_fchk","basis_file": "path/to/qc.fchk"}
my_system = pyopencap.System(sys_dict)
```

CAP object Densities can be read in from .fchk files, and the zeroth order Hamiltonian can be read from Q-Chem output files for EOM-CC calculations. To export the full densities to .fchk, GUI=2 must be set in the $rem card, and PROJ_CAP=3 must be set in the $complex_ccman card. See the Q-Chem manual for more details.

The following snippet can be used to read the data from a Q-Chem output and properly formatted .fchk file, and calculate the CAP matrix:

```
cap_dict = {"cap_type": "box",
            "cap_x":"2.76",
            "cap_y":"2.76",
            "cap_z":"4.88",
            "Radial_precision": "14",
            "angular_points": "110"}
nstates = 10
pc = pyopencap.CAP(my_system,cap_dict,nstates)
# read in densities
    es_dict = {"method" : "eom",
               "package": "qchem",
       "qchem_output":"path/to/output.out",
       "qchem_fchk":"path/to/qc.fchk"}
pc.read_data(es_dict)
# save the zeroth order Hamiltonian for later use
h0 = pc.get_H()
pc.compute_projected_cap()
W_mat=pc.get_projected_cap()
```

Generate and analyze eigenvalue trajectories

H0 and W, or the *CAP* object can be used to construct a *CAPHamiltonian* object.

```
from pyopencap.analysis import CAPHamiltonian
CAPH = CAPHamiltonian(H0=H0,W=W_mat)
# equivalently
CAPH = CAPHamiltonian(pc=pc)
```

Additionally, Q-Chem (starting from version 5.4) natively implements Projected CAP-EOM-CC and Projected CAP-ADC methods, and prints the necessary matrices to the output. PyOpenCAP can parse these output files to generate *CAPHamiltonian* objects.

```
from pyopencap.analysis import CAPHamiltonian
CAPH = CAPHamiltonian(output="proj-eomcc.out",irrep="B2g")
CAPH = CAPHamiltonian(output="proj-adc.out",onset="3000")
```

See the *analysis* section for more details.

7.4.4 PSI4

PSI4 is a C++/Python core that easily interfaces with and is extended by standalone community projects. The major advantage of using PSI4 in tandem with PyOpenCAP is that calculations can be performed in one-shot within the same python script. Since PSI4 allows direct control over data structures such as density matrices, the interface between PSI4 and PyOpenCAP is seamless. Our interface has been tested for the Psi4 dev build, which is available via conda:

```
conda install -c psi4/label/dev psi4
```

Step 1: Defining the System object

Molden(recommended)

The best way to construct the *System* object is to import the geometry and basis set from a molden file generated by a PSI4. This ensures proper ordering of the AO basis set.

```
psi4.molden(wfn, 'molden_in.molden')
molden_dict = {"basis_file":"molden_in.molden","molecule": "molden"}
s = pyopencap.System(molden_dict)
```

Inline

The molecule and basis set can also be specified inline. The "molecule" keyword must be set to "read", and then an additional keyword "geometry" must be specified, with a string that contains the geometry in xyz format. The "basis_file" keyword must be set to a path to a basis set file formatted in Psi4 style, which can be downloaded from the MolSSI BSE. Other optional keyword for this section include "bohr_coordinates" and "cart_bf". Please see the *keywords* section for more details. It is recommended to check the overlap matrix to ensure that the ordering and normalization matches. Up to G-type functions are supported.

```
E, wfn = psi4.energy('scf', return_wfn=True)
mints = psi4.core.MintsHelper(wfn.basisset())
S_mat = np.asarray(mints.ao_overlap())
sys_dict = {"geometry":    '''N  0  0   1.039
                           N  0  0   -1.039
                           X   0  0   0.0''',
                    "molecule" : "read",
                    "basis_file":"path/to/basis.bas",
                    "cart_bf":"d",
                    "bohr_coordinates:": "true"}
s.check_overlap_mat(S_mat,"psi4")
```

Step 1: Defining the CAP object

The CAP matrix is computed by the *CAP* object. The constructor requires a *System* object, a dictionary containing the CAP parameters, and the number of states. An example is provided below. Please see the keywords section for more information on the CAP parameters.

```
cap_dict = {"cap_type": "box",
            "cap_x":"2.76",
            "cap_y":"2.76",
            "cap_z":"4.88",
            "Radial_precision": "14",
            "angular_points": "110"}
nstates = 10
pc = pyopencap.CAP(my_system,cap_dict,nstates)
```

Step 2: Passing the density matrices

The simplest interface is with the full CI module. One can request one particle densities to be calculated by using the *opdm* and *tdm* options:

```
psi4.set_options({"opdm":True,"num_roots":nstates,"tdm":True,"dipmom":True})
ci_energy, ci_wfn = psi4.energy('FCI', return_wfn=True)
```

Densities are now available through the *get_opdm* function. One must be careful to ensure that the densities are represented in AO basis before passing to PyOpenCAP using the *add_tdm()* function:

```
for i in range(0,nstates):
    for j in range(i,nstates):
        opdm_mo = ci_wfn.get_opdm(i, j, "SUM", True)
        opdm_so = psi4.core.triplet(ci_wfn.Ca(), opdm_mo, ci_wfn.Ca(), False, False,␣
↪True)
        opdm_ao = psi4.core.Matrix(n_bas,n_bas)
        opdm_ao.remove_symmetry(opdm_so,so2ao)
        pc.add_tdm(opdm_ao.to_array(),i,j,"psi4")
        if not i==j:
            pc.add_tdm(opdm_ao.to_array().conj().T,j,i,"psi4")
```

Please see the PSI4 documentation for more details, or our repository for an example.

Note:

The interface with Psi4 is not restricted to FCI. The *add_tdm()* function is completely general; it requires only that the densities are in AO basis, and that the basis set ordering matches the system. An example for ADC is provided in the repository.

Step 3: Computing the CAP matrix

Once all of the densities are loaded, the CAP matrix is computed using the *compute_projected_cap()* function. The matrix can be retrieved using the *get_projected_cap()* function.

```
pc.compute_projected_cap()
W_mat=pc.get_projected_cap()
```

Step 4: Generate and analyze eigenvalue trajectories

H0 and W can be used to construct a `CAPHamiltonian` object. In many cases, it can be advantageous to use the `export()` function, which generates an OpenCAP formatted output file, which can be used for later analysis.

```
from pyopencap.analysis import CAPHamiltonian
CAPH = CAPHamiltonian(H0=H0,W=W_mat)
CAPH.export("output.out")
```

See the *analysis* section for more details.

Officially supported methods

- Full CI
- ADC (through ADCC)

7.4.5 Columbus

COLUMBUS is a collection of programs designed primarily for multi-reference (MR) calculations on electronic ground and excited states of atoms and molecules. Here, we outline the steps of performing CAP-MRCI calculations with Columbus and OpenCAP, though the steps are broadly applicable to any of the methods. These steps have only been tested for the serial version of Columbus.

Step 1: Running MRCI calculation

When running the MRCI calculation, ensure that transition moments between each pair of relevant states are requested. Once the MRCI calculation is finished, navigate to the WORK directory. Assuming one has set up the input properly, the following files will be needed

- *cid1trfl*: files for each pair of states, including state density matrices (i.e. cid1trfl.FROMdrt1.state1TOdrt1.state1)
- *ciudgsm*: file which contains final MRCI energies and convergence information
- *tranls*: file which contains information on active space/frozen orbitals, which is necessary to fully reconstruct density matrices in AO basis

Also, the MO coefficients will be needed, which are located in the MOLDEN directory:

- *molden_mo_mc.sp*: optimized MO coefficients from MCSCF calculation in .molden format

Step 2: Generating human readable density matrix files

The next step is to convert the *cid1trfl* files into a human readable format. The Columbus utility *iwfmt.x* can be used for this purpose. We provide a bash script in the main repository `utilities/write_iwfmt.bash` which can be executed in the WORK directory to generate these files, assuming that the $COLUMBUS environment variable is properly set.

Step 2: Importing the data to PyOpenCAP

System object

To run a PyOpenCAP calculation, the geometry and basis set must be imported into a *System* object. The constructor takes in a Python dictionary as an argument.

Molden (recommended)

Molden files generated by Columbus contain the geometry and basis set information. To read in from molden, "molden" must be set as the value to the key "molecule", and the path to the file must be set as the value to the key "basis_file". Here is an example:

```
sys_dict = {"molecule": "molden","basis_file": "path/to/file.molden"}
my_system = pyopencap.System(sys_dict)
```

Inline(not recommended)

The molecule and basis set can also be specified manually. The "molecule" keyword must be set to "read", and then an additional keyword "geometry:" must be specified, with a string that contains the geometry in xyz format. The "basis_file" keyword must be set to a path to a basis set file formatted in Psi4 style, which can be downloaded from the MolSSI BSE. Other optional keyword for this section include "bohr_coordinates" and cart_bf. Please see the *keywords* section for more details. Up to G-type functions are supported.

```
sys_dict = {"geometry":    '''N  0  0   1.039
                           N  0  0   -1.039
                           X   0  0   0.0''',
                    "molecule" : "read",
                    "basis_file":"path/to/basis.bas",
                    "cart_bf":"d",
                    "bohr_coordinates:": "true"}
my_system = pyopencap.System(sys_dict)
```

One particle densities/zeroth order Hamiltonian

The CAP matrix is computed by the *CAP* object. The constructor requires a *System*, a dictionary containing the CAP parameters, the number of states, and finally the string "openmolcas", which denotes the ordering of the atomic orbital basis set. An example is provided below. Please see the keywords section for more information on the CAP parameters.

```
cap_dict = {"cap_type": "box",
            "cap_x":"2.76",
            "cap_y":"2.76",
            "cap_z":"4.88"}
nstates = 10
pc = pyopencap.CAP(my_system,cap_dict,nstates)
```

Before we can compute the CAP matrix in the state basis, we must load in the density matrices. Due to the large number of files generated by Columbus, we have provided a *colparser* utility to manage the data.

A *colparser* object is instantiated using the *tranls* file and the MO coefficients:

```
parser = colparser('data_files/molden_mo_mc.sp', 'data_files/tranls')
```

The zeroth order Hamiltonian, which is diagonal for MR-CI, can be read in from the *ciudgsm* file as follows:

```
H0 = parser.get_H0(filename='data_files/ciudgsm')
```

Densities are loaded in one at a time using *pyopencap.analysis.colparser.sdm_ao()* / *pyopencap.analysis.colparser.tdm_ao()* and *add_tdm()*. To specify which tdm/sdm to parse, one can use state and optionally DRT indices:

```
for i in range(0,nstates):
        for j in range(i,nstates):
                if i==j:
                        # Indices start from 0 in pyopencap, but from 1 in Columbus file␣
↪names
                        dm1_ao = parser.sdm_ao(i+1,data_dir='data_files',DRTn=1)
                        pc.add_tdm(dm1_ao,i,j,'molden')
                else:
                        # Indices start from 0 in pyopencap, but from 1 in Columbus file␣
↪names
                        dm1_ao = parser.tdm_ao(i+1, j+1,drtFrom=1,drtTo=1,data_dir='data_
↪files')
                        pc.add_tdm(dm1_ao,i,j,'molden')
                        pc.add_tdm(dm1_ao.conj().T,j,i,'molden')
pc.compute_projected_cap()
W=pc.get_projected_cap()
```

In this example, the files are assumed to located in `./data_files` with names `cid1trfl.FROMdrt{drtFrom}.state{i}TOdrt{drtTo}.state{i}.iwfmt`, which is consistent with them having been generated by the `utilities/write_iwfmt.bash` script.

Alternatively, one can absolute paths:

```
dm1_ao = parser.sdm_ao(1,filename='data_files/cid1trfl.FROMdrt1.state1TOdrt1.state1.iwfmt
↪')
pc.add_tdm(dm1_ao,0,0,'molden')
```

Step 4: Generate and analyze eigenvalue trajectories

H0 and W, or the *CAP* object can be used to construct a *CAPHamiltonian* object.

```
from pyopencap.analysis import CAPHamiltonian
CAPH = CAPHamiltonian(H0=H0,W=W_mat)
# equivalently
CAPH = CAPHamiltonian(pc=pc)
```

See the *analysis* section for more details.

Officially supported methods

MR-CISD has been officially tested, though the interface should work with other methods. Please contact us if you find success or have issues using any other methods so we can add official support!

7.5 Analysis Tools

PyOpenCAP provides user friendly tools for analysis of eigenvalue trajectories in the form of *CAPHamiltonian* and *EigenvalueTrajectory* objects.

Basic usage

The *CAPHamiltonian* contains functions aimed at diagonalization of the CAP Hamiltonian over a range of eta values. Assuming one has already obtained H0 and W in the state basis as numpy matrices:

```
from pyopencap.analysis.CAPTrajectory import CAPHamiltonian
eta_list = np.linspace(0,2000,101)
eta_list = eta_list * 1E-5
CAPH = CAPHamiltonian(H0=h0,W=mat)
CAPH.run_trajectory(eta_list,cap_lambda=0.0)
# track the 4th state
traj = CAPH.track_state(4,tracking="overlap")
```

Alternatively, one can read in H0 and W from OpenCAP/Q-Chem output files:

```
from pyopencap.analysis.CAPTrajectory import CAPHamiltonian
eta_list = np.linspace(0,2000,101)
eta_list = eta_list * 1E-5
CAPH = CAPHamiltonian(output_file="path/to/output.out")
CAPH.run_trajectory(eta_list,cap_lambda=0.0)
# track the 4th state
traj = CAPH.track_state(4,tracking="overlap")
```

In both snippets, *traj* is now a *EigenvalueTrajectory* object, which contains helpful functions for analysis. For example, one can find the optimal value of the CAP strength parameter for uncorrected/corrected trajectories:

```
uc_energy,uc_eta_opt = traj.find_eta_opt()
corr_energy,corr_eta_opt = traj.find_eta_opt(corrected=True)
```

For more information, please see the documentation for the *CAPHamiltonian* and *EigenvalueTrajectory* classes.

7.5.1 CAPHamiltonian

This section briefly describes how to use the CAPHamiltonian object to generate eigenvalue trajectories.

Initialization

CAPHamiltonian objects can be initialized in one of two ways. The first is to pass H0 and W as numpy arrays:

```
from pyopencap.analysis.CAPTrajectory import CAPHamiltonian
CAPH = CAPHamiltonian(H0=h0,W=mat)
```

The other is to read them in from an OpenCAP output file, or from a Q-Chem output file generated by a Projected CAP-EOM-CC or Projected CAP-ADC calculation.

```
CAPH = CAPHamiltonian(output="path/to/output.out")
```

If one wishes to exclude some of the states from the analysis, this can be accomplished through the by placing their indices in a list (starting from 0) and passing it into the *exclude_states* keyword argument:

```
exclude_states = [2,5,7]
CAPH = CAPHamiltonian(H0=h0,W=mat,exclude_states=exclude_states)
```

Similarly, the *include_states* argument includes only the desired states. Note that these two keywords are incompatible.

```
include_states = [0,1,2,3,4]
CAPH = CAPHamiltonian(H0=h0,W=mat,include_states=include_states)
```

Importantly, in all cases, **the W matrix is assumed to be pre-multiplied by a factor of -1.0**.

Diagonalization

The *run_trajectory()* function diagonalizes the CAP Hamiltonian over a range of eta values (and at a specified value of the cap lambda parameter if using a CR-CAP).

```
eta_list = np.linspace(0,2000,101)
eta_list = eta_list * 1E-5
CAPH.run_trajectory(eta_list,cap_lambda=0.0)
```

Since the W matrix is assumed to be multiplied by a factor of -1.0 upon instantiation, the following matrix is actually diagonalized at each step:

$$H^{CAP} = H_0 + (i\eta - \lambda)W$$

and each eigenpair is stored in a *Root* object. After all of the diagonalizations are finished, individual states can be tracked using the *track_state()* function:

```
traj = CAPH.track_state(4,tracking="overlap")
```

The *traj* variable is a *EigenvalueTrajectory* object, which contains helpful functions for analysis. Indices for states start from 0, and there are two options for tracking states: "overlap" (the default), and "energy". See *EigenvalueTrajectory* for more details.

Visualization

The energies of all states computed are stored in the *total_energies* instance variable of the *CAPHamiltonian* object. This can very useful for graphical searches e.g.

```
import matplotlib.pyplot as plt
import numpy as np
plt.plot(np.real(CAPH.total_energies),np.imag(CAPH.total_energies),'ro')
plt.show()
```

There is also a function *energies_ev()* which returns the excitation energies in eV with respect to specified reference energy.

```
E_ev = CAPH.energies_ev(ref_energy)
plt.plot(np.real(E_ev),np.imag(CAPH.energies_ev(E_ev),'ro')
plt.show()
```

class pyopencap.analysis.**CAPHamiltonian**(*pc=None*, *H0=None*, *W=None*, *output=None*, *irrep=''*, *onset=''*)

Projected CAP Hamiltonian handler for generating eigenvalue trajectories.

The instance variables H0,W etc. are only set after *run_trajectory* is executed. The original matrices passed/parsed when the object is constructed are stored in _H0, _W, etc. This makes it easy to run multiple trajectories with different states included in the projection scheme without having to construct a new object.

H0

Zeroth order Hamiltonian in state basis

Type
np.ndarray of float: default=None

W

CAP matrix in state basis. -1 prefactor is assumed.

Type
np.ndarray of float: default=None

nstates

Number of states

Type
int

total_energies

Energies of all states found by repeated diagonalization of CAP Hamiltonian

Type
list of float

etas

List of CAP strengths in trajectory.

Type
list of float

cap_lambda

Real CAP strength used for continuum remover CAP. Set to 0.0 by default.

Type
float

__init__(*pc=None*, *H0=None*, *W=None*, *output=None*, *irrep=''*, *onset=''*)

Initializes CAPHamiltonian object from H0 and W matrix in state basis.

Object can be initialized in one of two ways. The user can pass H0 and W directly as numpy matrices, or they can specify a path to a properly formatted output file (either OpenCAP or Q-Chem) which contains these two matrices.

W matrix is assumed to already have a -1 prefactor, as that is how OpenCAP output is formatted.

Parameters

- **pc** (*CAP*: default=None) – PyOpenCAP CAP object
- **H0** (*np.ndarray of float: default=None*) – Zeroth order Hamiltonian in state basis
- **W** (*np.ndarray of float: default=None*) – CAP matrix in state basis. -1 prefactor is assumed.
- **output** (*str: default=None*) – Path to Q-Chem or OpenCAP output file.
- **irrep** (*str: default=None*) – Title of irreducible representation of state of interest. Only compatible with Q-Chem projected CAP-EOM-CC outputs. Set to 'all' to include all symmetries in CAP projection.
- **onset** (*str: default=None*) – Title of CAP onset. Only compatible with Q-Chem projected CAP-ADC outputs.

energies_ev(*ref_energy*)

Returns excitation energies of all calculated states in eV with respect to specified reference energy.

Parameters
ref_energy (*float*) – Reference energy

Returns
E_eV – Excitation energies in eV with respect to specified reference energy.

Return type
list of floats

export(*finame*)

Exports Zeroth order Hamiltonian and CAP matrix to an OpenCAP formatted output file for further analysis. Useful for saving the results of an expensive electronic structure calculation performed in a python environment.

Parameters
finame (*str*) – File handle to export data.

run_trajectory(*eta_list*, *cap_lambda=0.0*, *exclude_states=None*, *include_states=None*, *biorthogonalize=False*)

Diagonalizes CAP Hamiltonian over range of eta values.

CAP Hamiltonian is defined as H^CAP = H0 + i*eta*W - cap_lambda * W. W matrix is assumed to already have a -1 prefactor, as that is how OpenCAP output is formatted. Recommended range for eta_list is between 1E-5 and 1E-2, though this can vary widely based on system and CAP shape.

Parameters

- **eta_list** (*iterable object*) – List of eta values to use
- **cap_lambda** (*float, default=0.0*) – Real CAP strength to use for continuum remover CAP

- **exclude_states** (*list of int, default=None*) – List of states to exclude from subspace projection. Not compatible with include_states parameter.
- **include_states** (*list of int, default=None*) – List of states to include in subspace projection. Not compatible with exclude_states parameter.
- **biorthogonalize** (*bool, default=False*) – Biorthogonalize left and right eigenvectors. If false, left eigenvectors are assumed to equal right eigenvectors for density matrix correction

track_state(*state_idx*, *tracking='overlap'*, *correction='derivative'*)

Tracks eigenvalue trajectory over range of eta values.

Parameters

- **state_idx** (*int*) – Index of state to track
- **tracking** (*str, default="overlap"*) – Method to use to track the state. Options are "overlap", which tracks based on eigenvector overlap, and "energy" which tracks based on energy.
- **correction** (*str, default="derivative"*) – Choice of correction scheme. Either "density" or "derivative".

Returns

traj – Eigenvalue trajectory for further analysis.

Return type

EigenvalueTrajectory

7.5.2 EigenvalueTrajectory

This section briefly describes how to use the EigenvalueTrajectory object to analyze eigenvalue trajectories.

Initialization

EigenvalueTrajectory objects are generated by the *track_state()* function of the *CAPHamiltonian* class. The state index i starts from 0, and the first state in the trajectory is the ith eigen pair generated by the first diagonalization at $\eta = 0$.

State Tracking

At each diagonalization of the CAP Hamiltonian, left and right η-dependent eigenpairs are computed and biorthogonalized. States are tracked using using one of two criterion: *overlap* and *energy*, which is controlled by the **tracking** keyword argument of *track_state()*.

When overlap tracking is used (the default), at each step, the state with maximum overlap with the previous state is chosen as the next point on the trajectory. The overlap is simply calculated as the absolute value of the dot product between the right eigenvectors.

When energy tracking is used, at each step, the state with the minimum difference in energy from the previous state is chosen. Since the energies are complex, the difference is computed as the modulus of the difference of the two complex energies.

Each state in the trajectory is stored as a *Root* object stored in the *states* attribute. Lists of uncorrected energies and η values (in order of smallest to largest η value) are also stored in the *uncorrected_energies* and *etas* class attributes for convenience.

Corrected Trajectories

Raw uncorrected energies obtained from diagonalizaiton of the CAP Hamiltonian can be sensitive to CAP onset and basis set quality. Practitioners of CAP theory often report so called *corrected* energies, the exact form of which may vary from publication to publication.

We implement two forms of 1st-order corrections: *density* and *derivative*, which is controlled by the **correction** keyword argument of *track_state()*.

The default is the first-order correction of [Cederbaum2002], which has the form:

$U(\eta) = E(\eta) - \eta \frac{\partial E(\eta)}{\partial \eta}$.

One can also use the density matrix correction of [Jagau2014] :

$U(\eta) = E(\eta) + i\eta Tr[\gamma(\eta)W]$,

where the trace expression is evaluated using the matrix elements of the CAP in the correlated basis:

$Tr[\gamma(\eta)W] = \sum_{kl} c_k^L(\eta) c_l^R(\eta) W_{kl}^{CB}$,

and c^L and c^R xrefer to the components of the bi-orthogonalized left and right eigenvectors respectively. Note that this option should only be used when *biorthogonalize=True* is passed to *run_trajectory()*.

The two approaches can be related to one another by the Hellman-Feynman theoreom, and in our experience yield nearly identical results.

Corrected trajectories are automatically computed when one obtains an *EigenvalueTrajectory* object generated by the *track_state()* function, and are stored in the class attribute *corrected_energies*.

```
# the various options for state tracking and corrections
traj = CAPH.track_state(1,tracking="energy",correction="density")
traj = CAPH.track_state(1,tracking="overlap",correction="derivative")
```

η_{opt}

As briefly outlined in the *theory*, the key to any CAP calculation is to find the optimal value of the CAP strength parameter η_{opt}. The *EigenvalueTrajectory* class has a function *find_eta_opt* for exactly that purpose. For uncorrected trajectories, η_{opt} is calculated as

$min|\eta \frac{dE}{d\eta}|$

For corrected trajectories, η_{opt} is calculated as

$min|\eta \frac{dU}{d\eta}|$

The derivative is calculated numerically by means of finite differences.

```
uc_energy,uc_eta_opt = traj.find_eta_opt()
corr_energy,corr_eta_opt = traj.find_eta_opt(corrected=True)
```

The presence of nonphysical stationary points can sometimes result in this function returning smaller values of η_{opt} than desired. One can specify the *start_idx* keyword argument to begin the search starting from a specified index along the trajectory.

```
uc_energy,uc_eta_opt = traj.find_eta_opt(start_idx=20)
```

Visualization

It can often be helpful to visualize trajectories graphically. In addition to access to the class attributes *uncorrected_energies* and *corrected_energies*, we also provide some helper functions which process the data in useful ways for visualization.

For instance, *energies_ev()* returns the excitation energies in eV with respect to specified reference energy.

```
import matplotlib.pyplot as plt
UC_ev = traj.energies_ev(ref_energy)
Corr_ev = traj.energies_ev(ref_energy,corrected=True)
plt.plot(np.real(UC_ev),np.imag(UC_ev),'ro', label='Uncorrected')
plt.plot(np.real(Corr_ev),np.imag(Corr_ev),'ro', label='Corrected')
plt.show()
```

get_logarithmic_velocities() returns the value of $\eta \frac{\partial E(\eta)}{\partial \eta}$ (or $|\eta \frac{dU}{d\eta}| \rightarrow min$ if the corrected keyword argument is set to True) for each point along the trajectory.

```
derivs = traj.get_logarithmic_velocities()
plt.plot(traj.etas,derivs)
plt.show()
```

References

class pyopencap.analysis.**EigenvalueTrajectory**(*state_idx*, *init_roots*, *W*, *tracking='overlap'*, *correction='derivative'*, *biorthogonalized=False*)

Eigenvalue trajectory generated by repeated diagonalizations of CAP Hamiltonian over range of eta values.

States are tracked using either eigenvector overlap or energy criterion. Corrected energies are obtained using density matrix or first order correction.

roots

List of states in trajectory.

Type
list of *Root*

uncorrected_energies

List of uncorrected energies in trajectory.

Type
list of float

corrected_energies

List of corrected energies in trajectory.

Type
list of float

W

CAP matrix in state basis. -1 prefactor is assumed.

Type
np.ndarray of float: default=None

etas

List of CAP strengths in trajectory.

Type
list of float

tracking

Method to use to track the state

Type
str, default="overlap"

correction

Choice of correction scheme. Either "density" or "derivative". Density matrix correction should only be used when the eigenvectors have been biorthogonalized. See *run_trajectory()*.

Type
str, default="derivative"

__init__(*state_idx*, *init_roots*, *W*, *tracking='overlap'*, *correction='derivative'*, *biorthogonalized=False*)

Initializes EigenvalueTrajectory object, which tracks a state starting from the first diagonalization at eta = 0.

Parameters

- **state_idx** (*int*) – Index of state to track
- **init_roots** (list of *Root*) – Initial set of roots generated by diagonalization at eta = 0
- **W** (*np.ndarray of float: default=None*) – CAP matrix in state basis. -1 prefactor is assumed.
- **tracking** (*str, default="overlap"*) – Method to use to track the state
- **correction** (*str, default="derivative"*) – Choice of correction scheme. Either "density" or "derivative".
- **biorthogonalized** (*bool default=False*) – Whether eigenvectors have been biorthogonalized

energies_ev(*ref_energy*, *corrected=False*)

Returns excitation energies of all states in trajectory in eV with respect to specified reference energy.

Parameters

- **ref_energy** (*float*) – Reference energy
- **corrected** (*bool, default=False*) – Set to true if analyzing corrected trajectory

Returns
E_eV – Excitation energies in eV with respect to specified reference energy.

Return type
list of floats

find_eta_opt(*corrected=False*, *start_idx=1*, *end_idx=-1*, *ref_energy=0.0*, *units='au'*, *return_root=False*)

Finds optimal cap strength parameter for eigenvalue trajectory, as defined by eta_opt = min|eta*dE/deta|.

The range of self.etas[start_idx:end_idx] (in python slice notation) is searched for the optimal value of CAP strength parameter.

Parameters

- **corrected** (*bool, default=False*) – Set to true if searching for stationary point on corrected trajectory
- **start_idx** (*int, default=1*) – Starting slice index
- **end_idx** (*int, default=1*) – Ending slice index
- **ref_energy** (*float, default=0.0*) – Reference energy to define excitation energy.
- **units** (*str, default="au"*) – Options are “au” or “eV”
- **return_root** (*bool, default=False*) – Whether to return *Root* object associated with optimal value of eta

Returns

- **E_res** (*complex float*) – Complex energy at optimal value of eta
- **eta_opt** (*float*) – Optimal value of eta
- **root** (*Root*) – Only returned when return_root is set to true

get_energy(*eta*, *corrected=False*, *ref_energy=0.0*, *units='au'*, *return_root=False*)

Returns total energy at given value of eta.

Note that if the eta provided is not in self.etas, the nearest value will be used.

Parameters

- **eta** (*float*) – Value of CAP strength parameter
- **corrected** (*bool, default=False*) – Set to true if analyzing corrected trajectory
- **ref_energy** (*float, default=0.0*) – Reference energy to define excitation energy.
- **units** (*str, default="au"*) – Options are “au” or “eV”
- **return_root** (*bool, default=False*) – Whether to return *Root* object associated with optimal value of eta

Returns

- **E** (*float*) – Energy at given value of eta
- **root** (*Root*) – Only returned when return_root is set to true

get_logarithmic_velocities(*corrected=False*)

Returns eta*dE/deta for each point on eigenvalue trajectory.

Useful for plotting when dealing with multiple potential stationary points.

Parameters
corrected (*bool, default=False*) – Set to true if analyzing corrected trajectory

Returns
derivs – eta*dE/deta for each point on eigenvalue trajectory

Return type
np.array of float

7.5.3 Root

class pyopencap.analysis.**Root**(*energy*, *eta*, *Reigvc*, *Leigvc*)

Root obtained from diagonalizing CAP Hamiltonian at given value of eta.

eta

CAP strength parameter

Type
float

energy

Total energy of state

Type
float

Reigvc

Bi-orthogonalized right eigenvector for state

Type
list of float

Leigvc

Bi-orthogonalized left eigenvector for state

Type
list of float

__init__(*energy*, *eta*, *Reigvc*, *Leigvc*)

Initializes root object.

Parameters

- **eta** (*float*) – CAP strength parameter
- **energy** (*float*) – Total energy of state
- **eigvc** (*list of float*) – Eigenvector for state

7.5.4 Columbus Parser

class pyopencap.analysis.**colparser**(*molden_file*, *tranls*)

A class that parses COLUMBUS electronic structure package generated files to generate state density and transition density matrices for projected-CAP calculation on MR-CI level.

__init__(*molden_file*, *tranls*)

Initializes the colparser class

Parameters

- **molden_file** (*str*) – molden MO filename (generated in MOLDEN/ folder in COLUMBUS calculation Directory)
- **tranls** (*str*) – Path to tranls file generated by Columbus in WORK directory

get_H0(*correction_type='eci+pople'*, *filename='ciudgsm'*)

Parses energies from a Columbus ciudgsm file.

Parameters

- **correction_type** (*str, optional*) – One of {'eci+pople', 'eci', 'eci+dv1', 'eci+dv2', 'eci+dv3' }. Default is 'eci+pople'
- **filename** (*str, optional*) – Path to Columbus ciudgsm file located in WORK directory. If unspecified, assumed to be './ciudgsm'.

Notes

See https://aip.scitation.org/doi/pdf/10.1063/1.5144267 for discussion of corrections.

Returns
H0_mat – Diagonal hamiltonian with CI energies.

Return type
np.ndarray

mo_summary()

Prints information about active space and symmetries of MOs.

Returns
Total number of basis functions (nbft) Number of basis functions in each symmetry block (NBPSY). Number of orbitals in each of the symmetry blocks.(NMPSY<= NBSPSY) Number of frozen orbitals in each of the symmetry blocks (NFCSY). Character labels for the symmetry blocks (SLABEL).

Return type
str

sdm_ao(*i*, *DRTn=1*, *data_dir='.'*, *filename=None*)

Returns state density matrix in atomic orbital basis by parsing a Columbus cid1trfl.iwfmt file.

Parameters

- **i** (*int*) – State index
- **DRTn** (*int, optional*) – DRT index
- **data_dir** (*str, optional*) – Directory to search for .iwfmt file. Should not be used in conjunction with *filename* kwarg
- **filename** (*str, optional*) – Path to file to parse. If not specified, the filename is assumed to be cid1trfl.FROMdrt{drtFrom}.state{i}TOdrt{drtTo}.state{i}.iwfmt in the current directory.

Returns
sdm – State density matrix in AO basis

Return type
np.ndarray

sdm_ao_cid1fl(*i*, *DRTn*)

Read CI state densities from state density cid1fl*.iwfmt files.

Currently NYI.

Parameters

- **i** (*int*) – State index
- **DRTn** (*int*) – DRT index

Raises
NotImplementedError –

Returns
State density matrix in AO basis

Return type
np.ndarray

tdm_ao(*iFROM*, *iTO*, *drtFrom=1*, *drtTo=1*, *data_dir='.'*, *filename=None*)

Returns transition density matrix in atomic orbital basis by parsing a Columbus cid1trfl.iwfmt file.

Parameters

- **iFROM** (*int*) – Initial state index and final state indices respectively.
- **iTO** (*int*) – Initial state index and final state indices respectively.
- **drtFrom** (*int, optional*) – DRT indices
- **drtTo** (*int, optional*) – DRT indices
- **data_dir** (*str, optional*) – Directory to search for .iwfmt file. Should not be used in conjunction with *filename* kwarg
- **filename** (*str, optional*) – Path to file to parse. If not specified, the filename is assumed to be cid1trfl.FROMdrt{drtFrom}.state{iFrom}TOdrt{drtTo}.state{iTO}.iwfmt in the current directory.

Returns
tdm – Transition density matrix in AO basis

Return type
np.ndarray

7.6 Custom CAPs and Grids

Starting with PyOpenCAP version 1.2, users can now specify customized CAP functions and numerical integration grids.

7.6.1 Custom CAPs

Python functions with the signature `vector<double>,vector<double>,vector<double>,vector<double> --> vector<double>` can be used as CAP functions by the *CAP* class for numerical integration. An example is provided below:

```
# this defines a box CAP of with cutoffs of 3 bohr in each coordinate
def box_cap(x,y,z,w):
    cap_values = []
    cap_x = 3.00
    cap_y = 3.00
    cap_z = 3.00
    for i in range(0,len(x)):
        result = 0
        if np.abs(x[i])>cap_x:
            result += (np.abs(x[i])-cap_x) * (np.abs(x[i])-cap_x)
        if np.abs(y[i])>cap_y:
            result += (np.abs(y[i])-cap_y) * (np.abs(y[i])-cap_y)
        if np.abs(z[i])>cap_z:
```

(continues on next page)

(continued from previous page)

```
            result += (np.abs(z[i])-cap_z) * (np.abs(z[i])-cap_z)
        result = w[i]*result
        cap_values.append(result)
    return cap_values

cap_dict = {"cap_type": "custom"}
pc = pyopencap.CAP(s,cap_dict,5,box_cap)
```

7.6.2 Custom Grids

Custom grids for numerical integration can be specified using the *compute_cap_on_grid* function. The arguments are assumed to be 1D arrays of equal size. The function can be called repeatedly for a cumulative sum in the case of atomic grids. An example is provided below:

```
for i in range(0,Natoms):
    x,y,z,w = get_grid_for_atom(atoms[i])
    pc.compute_cap_on_grid(x,y,z,w)
# final sum is cumulative
pc.compute_projected_cap()
```

7.7 API

PyOpenCAP exposes two OpenCAP classes to Python: *System* and *CAP*.

7.7.1 pyopencap.System

The System class is used to store the molecular geometry and the basis set. Upon construction, it automatically computes the overlap matrix which can be accessed and used to verify the the ordering of the atomic orbital basis set.

class pyopencap.**System**

__init__(*self: pyopencap.pyopencap_cpp.System*, *sys_dict: dict*) → None

Constructs System object from python dictionary.

get_overlap_mat(*self: pyopencap.pyopencap_cpp.System*, *ordering: str = 'molden'*, *basis_file: str = ''*) → numpy.ndarray[numpy.float64[m, n]]

Returns overlap matrix. Supported orderings: pyscf, openmolcas, qchem, psi4, molden.

check_overlap_mat(*self: pyopencap.pyopencap_cpp.System*, *smat: numpy.ndarray[numpy.float64[m, n]]*, *ordering: str*, *basis_file: str = ''*) → bool

Compares input overlap matrix to internal overlap matrix to check basis set ordering. Supported orderings: pyscf, openmolcas, qchem, psi4, molden.

get_basis_ids(*self: pyopencap.pyopencap_cpp.System*) → str

Returns a string of the basis function ids. Each ID has the following format:atom index,shell number,l,m

7.7.2 pyopencap.CAP

The *CAP* class is used to compute the CAP matrix first in AO basis, and then in wave function basis using the one-particle densities which are passed in. It is also capable of parsing OpenMolcas output files to obtain the zeroth order Hamiltonian and return it to the user.

class pyopencap.**CAP**

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: pyopencap.pyopencap_cpp.CAP, system: pyopencap.pyopencap_cpp.System, cap_dict: dict, nstates: int) -> None

Constructs CAP object from system, cap dictionary, and number of states.

2. __init__(self: pyopencap.pyopencap_cpp.CAP, system: pyopencap.pyopencap_cpp.System, cap_dict: dict, nstates: int, cap_func: Callable[[List[float], List[float], List[float], List[float]], List[float]]) -> None

Constructs CAP object from system, cap dictionary, number of states, and cap function.

add_tdm(*self: pyopencap.pyopencap_cpp.CAP*, *tdm: numpy.ndarray[numpy.float64[m, n]]*, *initial_idx: int*, *final_idx: int*, *ordering: str*, *basis_file: str = ''*) → None

Adds spin-traced tdm to CAP object at specified indices. The optional argument basis_file is required when using the OpenMolcas interface, and it must point to the path to the rassi.5 file. Supported orderings: pyscf, openmolcas, qchem, psi4, molden.

add_tdms(*self: pyopencap.pyopencap_cpp.CAP*, *alpha_density: numpy.ndarray[numpy.float64[m, n]]*, *beta_density: numpy.ndarray[numpy.float64[m, n]]*, *initial_idx: int*, *final_idx: int*, *ordering: str*, *basis_file: str = ''*) → None

Adds alpha/beta tdms to CAP object at specified indices. The optional argument basis_file is required when using the OpenMolcas interface, and it must point to the path to the rassi.5 file. Supported orderings: pyscf, openmolcas, qchem, psi4, molden.

compute_ao_cap(*self: pyopencap.pyopencap_cpp.CAP*, *cap_dict: dict*, *cap_func: Callable[[List[float], List[float], List[float], List[float]], List[float]] = None*) → None

Computes CAP matrix in AO basis.

compute_projected_cap(*self: pyopencap.pyopencap_cpp.CAP*) → None

Computes CAP matrix in state basis using transition density matrices.

get_H(*self: pyopencap.pyopencap_cpp.CAP*) → numpy.ndarray[numpy.float64[m, n]]

Returns zeroth order Hamiltonian read from file.

get_ao_cap(*self: pyopencap.pyopencap_cpp.CAP*, *ordering: str = 'molden'*, *basis_file: str = ''*) → numpy.ndarray[numpy.float64[m, n]]

Returns CAP matrix in AO basis. Supported orderings: pyscf, openmolcas, qchem, psi4, molden.

get_projected_cap(*self: pyopencap.pyopencap_cpp.CAP*) → numpy.ndarray[numpy.float64[m, n]]

Returns CAP matrix in state basis.

read_data(*self: pyopencap.pyopencap_cpp.CAP*, *es_dict: dict*) → None

Reads electronic structure data specified in dictionary.

renormalize(*self: pyopencap.pyopencap_cpp.CAP*) → None

Re-normalizes AO CAP using electronic structure data.

renormalize_cap(*self: pyopencap.pyopencap_cpp.CAP*, *smat: numpy.ndarray[numpy.float64[m, n]]*, *ordering: str*, *basis_file: str = ''*) → None

Re-normalizes AO CAP matrix using input overlap matrix.

compute_cap_on_grid(*self: pyopencap.pyopencap_cpp.CAP*, *x: numpy.ndarray[numpy.float64]*, *y: numpy.ndarray[numpy.float64]*, *z: numpy.ndarray[numpy.float64]*, *w: numpy.ndarray[numpy.float64]*) → None

Computes CAP matrix on supplied grid. Sum will cumulated for each successive grid until compute_projected_cap is called.

7.8 Keywords

PyOpenCAP uses Python dictionaries which contain key/value pairs to specify the parameters of the calculation. Here, we outline the valid key/value combinations. Importantly, **all key value pairs should be specified as strings**.

7.8.1 System keywords

The *System* object contains the basis set and geometry information, which can be obtained in a few different ways.

Keyword	Required	Valid values	Description
molecule	yes	molden,qchem_fchk, rassi_h5,inline	Specifies which format to read the molecular geometry. If "inline" is chosen, the "geometry" keyword is also required.
geometry	no	See below	Specifies the geometry in an inline format described below. Required when the "molecule" field is set to "inline".
basis_file	yes	path to basis file	Specifies the path to the basis file. When "molecule" is set to "molden","rassi_h5", or "qchem_fchk", this field should be set to a path to a file of the specified type. When "molecule" is set to "inline", this field should be set to a path to a basis set file formatted in "Psi4" style.
cart_bf	no	'd', 'df', 'dfg' 'dg', 'f', 'g', 'fg'	Controls the use of pure or Cartesian angular forms of GTOs. The letters corresponding to the angular momenta listed in this field will be expanded in cartesians, those not listed will be expanded in pure GTOs. For example, "df" means d and f-type functions will be cartesian, and all others will be pure harmonic. This keyword is only active when "molecule" is set to "inline".
bohr_coordinates	no	"True" or "False"	Set this keyword to true when the coordinates specified in "geometry" keyword are in bohr units. This keyword is only active when "molecule" is set to "inline".

When specifying the geometry inline, use the following format:

```
atom1 x-coordinate y-coordinate z-coordinate

atom2 x-coordinate y-coordinate z-coordinate ...
```

Ghost centers with zero nuclear charge can be specified using the symbol "X".

Units are assumed to be Angstroms unless the bohr_coordinates keyword is set to True.

Example:

```
sys_dict = {"geometry":    '''N  0  0   1.039
                           N  0  0   -1.039
                           X   0  0   0.0''',
                       "molecule" : "read",
                       "basis_file":"path/to/basis.bas",
                       "cart_bf":"d",
                       "bohr_coordinates:": "true"}
```

7.8.2 CAP keywords

PyOpenCAP supports Voronoi and Box-type absorbing potentials. We also allow some customization of the numerical grid used for integration. Please see https://github.com/dftlibs/numgrid for more details on the radial_precision and angular_points keywords.

General Keywords

Keyword	Required	Default/valid values	Description
cap_type	yes	"box" or "voronoi"	Type of absorbing potential.
radial_precision	no	"16"	Radial precision for numerical integration grid. A precision of 1x10^(-N), where N is the value specified is used.
angular_points	no	"590"	Number of angular points used for the grid. See https://github.com/dftlibs/numgrid for allowed numbers of points.
thresh	no	"7"	Threshold for exponents of GTO which contribute to CAP integrals. All GTOs with exponents smaller than $1.0x10^{-thresh}$ will be discarded for CAP integrals. If you're getting errors about allocating the grid, try reducing thresh.
do_numerical	no	True for box CAPs, false for other CAPs	Analytical [Santra1999] integrals are available for Box CAPs only, and are used by default. All other CAPs must be integrated numerically.

Box CAP

A quadratic potential which encloses the system in a 3D rectangular box. Analytical integrals are available for these types of CAPs.

$$W = W_x + W_y + W_z$$

$$W_\alpha = \begin{Bmatrix} 0 & |r_\alpha| < R_\alpha^0 \\ (r_\alpha - R_\alpha^0)^2 & |r_\alpha| > R_\alpha^0 \end{Bmatrix}$$

Keyword	Description
cap_x	Onset of CAP in x-direction. Specify in bohr units.
cap_y	Onset of CAP in y-direction. Specify in bohr units.
cap_y	Onset of CAP in z-direction. Specify in bohr units.

Smooth Voronoi CAP

A quadratic potential which uniformly wraps around the system at a specified cutoff radius. The edges between between Voronoi cells are smoothed out to make the potential more amenable to numerical integration [Sommerfeld2015].

$$W(\vec{r}) = \begin{Bmatrix} 0 & r_{WA} \leq r_{cut} \\ (r_{WA} - r_{cut})^2 & r_{WA} > r_{cut} \end{Bmatrix}$$

$$r_{WA}(\vec{r}) = \sqrt{\frac{\sum_i w_i |\vec{r} - \vec{R}_i|^2}{\sum_i w_i}}$$

$$w_i = \frac{1}{(|\vec{r} - \vec{R}_i|^2 - r_{min}^2 + 1a.u.)^2}$$

$$r_{min} = \min_i |\vec{r} - \vec{R}_i|$$

Keyword	Description
r_cut	Cutoff radius for Voronoi CAP. Specify in bohr units.

Example

```
cap_dict = {"cap_type": "box",
            "cap_x":"2.76",
            "cap_y":"2.76",
            "cap_z":"4.88",
            "Radial_precision": "14",
            "angular_points": "110"}
```

Electronic structure keywords

The *read_data()* function is able to parse the zeroth order Hamiltonian and load the densities when supplied with an appropriate formatted dictionary. All keywords must be specified to use this function. Currently, this is only supported for calculations using the OpenMolcas and Q-Chem interfaces.

Keyword	Description
method	Electronic structure method used in the calculation. Valid options are "MS-CASPT2", "EOM", and "TDDFT".
molcas_output	Path to OpenMolcas output file.
h0_file	Path to Zeroth order Hamiltonian file. Can be full matrix or diagonal. See https://github.com/gayverjr/opencap/tree/main/examples/opencap
package	"OpenMolcas" or "QChem"
rassi_h5	Path to OpenMolcas rassi.h5 file.
qchem_output	Path to Q-Chem output file.
qchem_fchk	Path to Q-Chem .fchk file.

Example:

```
es_dict = { "package": "openmolcas",
"method" : "ms-caspt2",
       "molcas_output":"path/to/output.out",
       "rassi_h5":"path/to/rassi.h5"}
pc.read_data(es_dict)
```

7.8.3 References

BIBLIOGRAPHY

[Riss1993] Riss, U. V.; Meyer, H. D. Calculation of Resonance Energies and Widths Using the Complex Absorbing Potential Method. *J. Phys. B At. Mol. Opt. Phys.* **1993**, 26 (23), 4503–4535.

[Sommerfeld2001] Sommerfeld, T.; Santra, R. Efficient Method to Perform CAP/CI Calculations for Temporary Anions. *Int. J. Quantum Chem.* **2001**, 82 (5), 218–226.

[Reinhardt1982] Reinhardt, W. P. Complex Coordinates in the Theory of Atomic and Molecular Structure and Dynamics. *Annu. Rev. Phys. Chem.* **1982**, 33 (1), 223–255.

[Moiseyev2009] Sajeev, Y.; Vysotskiy, V.; Cederbaum, L. S.; Moiseyev, N. Continuum Remover-Complex Absorbing Potential: Efficient Removal of the Nonphysical Stabilization Points. *J. Chem. Phys.* **2009**, 131 (21), 211102.

[Phung2020] Phung, Q. M.; Komori, Y.; Yanai, T.; Sommerfeld, T.; Ehara, M. Combination of a Voronoi-Type Complex Absorbing Potential with the XMS-CASPT2 Method and Pilot Applications. *J. Chem. Theory Comput.* **2020**, 16 (4), 2606–2616.

[Kunitsa2017] Kunitsa, A. A.; Granovsky, A. A.; Bravaya, K. B. CAP-XMCQDPT2 Method for Molecular Electronic Resonances. *J. Chem. Phys.* **2017**, 146 (18), 184107.

[Al-Saadon2019] Al-Saadon, R.; Shiozaki, T.; Knizia, G. Visualizing Complex-Valued Molecular Orbitals. *J. Phys. Chem. A* **2019**, 123 (14), 3223–3228.

[Cederbaum2002] Santra, R.; Cederbaum, L. S. Non-Hermitian Electronic Theory and Applications to Clusters. *Phys. Rep.* **2002**, 368 (1), 1–117.

[Jagau2014] Jagau TC, Zuev D, Bravaya KB, Epifanovsky E, Krylov AI. A Fresh Look at Resonances and Complex Absorbing Potentials: Density Matrix-Based Approach. *J Phys Chem Lett.* **2014** 5, 2, 310–315

[Sommerfeld2015] Sommerfeld, T.; Ehara, M. Complex Absorbing Potentials with Voronoi Isosurfaces Wrapping Perfectly around Molecules. *J. Chem. Theory Comput.* **2015**, 11 (10), 4627–4633.

[Santra1999] Santra, R.; Cederbaum, L. ~S.; Meyer, H.-D. Electronic Decay of Molecular Clusters: Non-Stationary States Computed by Standard Quantum Chemistry Methods. *Chem. Phys. Lett.* **1999**, 303 (3), 413–419.

INDEX

Symbols

A

C

E

F

G

H

L

M

N

R

S

T

U

W