sphinx-example Documentation

Firstname Lastname

Dec 04, 2023

CONTENTS

1 Installation **1**
- 1.1 Dalton program . . . 1
- 1.2 LSDalton Program . . . 1
- 1.3 Developers . . . 1

2 Development guide **3**
- 2.1 Code Style . . . 3
- 2.2 Continuous Integration and Code Coverage . . . 4
- 2.3 Documentation . . . 4

3 Plotting of Spectra **5**
- 3.1 UV/vis Spectrum . . . 5
- 3.2 Two-Photon Absorption Spectrum . . . 6
- 3.3 X-ray Absorption Spectroscopy using Coupled-Cluster methods . . . 8
- 3.4 Vibrational IR and/or Raman Spectrum . . . 9

4 Calculation of NMR parameters **13**
- 4.1 Shieldings . . . 13

5 Vibrational Corrections to Properties **15**
- 5.1 Correction to NMR shieldings using a linear fitting . . . 16
- 5.2 Correction to spin-spin coupling constants using a polynomial fitting . . . 17
- 5.3 Correction to frequency-dependent polarizabilities using a linear fitting . . . 18

6 Complete Active Space Self-Consistent Field **21**
- 6.1 Specifying CAS . . . 21
- 6.2 Picking CAS based on Natural Orbital Occupations . . . 22

7 Short-Range Density Functional Theory **25**
- 7.1 HF–srDFT . . . 25
- 7.2 MP2–srDFT . . . 26
- 7.3 CAS–srDFT . . . 27

8 molecule **29**

9 basis **31**

10 qcmethod **33**

11 property **35**

12 program **37**

13 dalton 43

14 gaussian 47

15 lsdalton 49

16 mol_reader 59

17 symmetry 61

18 natural_occupation 63

19 vibrational_analysis 67

20 vibrational_averaging 71

21 spectrum 75

Python Module Index 79

Index 81

CHAPTER

ONE

INSTALLATION

The Dalton Project package can be conveniently downloaded and installed using **pip**:

```
$ pip install [--user] daltonproject
```

The Dalton Project will in general expect the external program executables to be in the **PATH** environment variable. The path to the executables can be added to **PATH** using the following command:

```
$ export PATH=/path/to/executable:${PATH}
```

An example of this could be for the Dalton program, where the program has been compiled in the **build** directory:

```
$ export PATH=/home/username/dalton/build:${PATH}
```

Note that the export command can be added to **~/.bashrc** to automatically be run whenever a new terminal is opened.

1.1 Dalton program

The Dalton program can be downloaded from here. See the README for download and install instructions.

1.2 LSDalton Program

The LSDalton program can be downloaded from here. See the README for download and install instructions.

1.3 Developers

Developers can clone the Dalton Project git repository which is hosted on GitLab (https://gitlab/daltonproject/daltonproject):

```
$ git clone https://gitlab.com/daltonproject/daltonproject.git
```

Alternatively, you can fork the project on GitLab and the clone the fork. After the clone is downloaded you change directory to the root of the Dalton Project:

```
$ cd daltonproject
```

Then install the Dalton Project package in development mode which allows you to try out your changes immediately using the installed package:

```
$ pip install [--user] --editable .
```

Note that this will interfere with/overwrite any **pip** installed version of the Dalton Project. The requirements needed for development can be installed as:

```
$ pip install [--user] -r requirements.txt
```

Before making any changes, make sure that all tests pass by running the test suite (integration tests require that path to external libraries are in the PATH environment variable):

```
$ pytest
```

The code linting and formatting tools can be setup to work automatically via pre-commit hooks. To setup the pre-commit hooks run the following from the root of the Dalton Project directory:

```
$ pip install [--user] pre-commit
$ pre-commit install
```

During a **git commit**, if any pre-commit hook fails, mostly you will simply need to **git add** the affected files and **git commit** again, because most tools will automatically reformat the files.

CHAPTER

TWO

DEVELOPMENT GUIDE

Here is a brief outline for the development goal in terms of code, readability, and maintainability.

2.1 Code Style

The code style is enforced using pre-commit hooks that are run in the GitLab CI pipeline. It will check and format Python source files using the following code-style checkers and formatters

- Flake8
- YAPF
- isort
- mypy
- Doc8
- pydocstyle

In addition, it will check and fix other file types for trailing whitespaces and more. These code-style checkers and formatters are used to make the code-base look uniform and check that it is PEP8 compliant. Note that the line-length limit is set to 118 characters even though PEP8 recommends 79.

The pre-commit hooks are set up locally by running the following two commands in the root of the Dalton Project directory:

```
$ pip install pre-commit
$ pre-commit install
```

During a **git commit**, if any pre-commit hook fails, mostly you will simply need to **git add** the affected files and **git commit** again, because most tools will automatically reformat the files. Staged files can also be checked before committing by running:

```
$ pre-commit run
```

Again, if a step fails the tools will in most cases also reformat the files that caused the failure. You will therefore need to stage those files again (**git add**) and they are then ready to **git commit**.

If you want to run pre-commit on all files this can be done with the following command:

```
$ pre-commit run --all-files
```

2.2 Continuous Integration and Code Coverage

The Dalton Project uses continuous integration through GitLab CI and code coverage through codecov.io.

The test suite for Dalton Project can be run locally using:

```
$ pytest tests/*
```

The requirements for the testing suite can be installed by:

```
$ pip install [--user] -r tests/requirements.txt
```

2.3 Documentation

The documentation is compiled using sphinx. The documentation is automatically hosted using Read the Docs. Note that **autodoc** is enabled for Google-style docstrings.

The documentation can be generated locally by running the following command:

```
$ sphinx-build docs local_docs
```

Requirements for the documentation can be installed as:

```
$ pip install [--user] -r docs/requirements.txt
```

CHAPTER

THREE

PLOTTING OF SPECTRA

This tutorial explains how to produce various types of spectra through the Dalton Project platform.

Note, that the quantum chemistry methods used in this tutorial may not be suitable for the particular problem that you want to solve. Moreover, the basis sets used here are minimal in order to allow you to progress fast through the tutorial and should under no circumstances be used in production calculations.

3.1 UV/vis Spectrum

To create a UV/vis spectrum, we need to obtain excitation energies and oscillator strengths. First, we will need to specify our molecule and basis set:

```
import matplotlib.pyplot as plt

import daltonproject as dp

molecule = dp.Molecule(input_file='water.xyz')
basis = dp.Basis(basis='cc-pVDZ')
```

The excitation energies and oscillator strengths can be obtained with any method of choice, for example HF, CASSCF, or CC, but for this example, we will obtain them using TD-DFT with the CAM-B3LYP functional. In this case, we include the six lowest excitation energies and associated oscillator strengths.

```
dft = dp.QCMethod('DFT', 'CAMB3LYP')
prop = dp.Property(excitation_energies=True)
prop.excitation_energies(states=6)
result = dp.dalton.compute(molecule, basis, dft, prop)
print('Excitation energies =', result.excitation_energies)
# Excitation energies = [ 7.708981 10.457775 11.631572 11.928512 14.928328 15.733272]

print('Oscillator strengths =', result.oscillator_strengths)
# Oscillator strengths = [0.0255 0.     0.1324 0.0883 0.0787 0.171 ]
```

To visualize the resulting spectrum, we can use the built-in plotting functionality from the spectrum module.

```
ax = dp.spectrum.plot_one_photon_spectrum(result, color='k')
plt.savefig('1pa.svg')
```

And the resulting spectrum looks like:

The plotting function returns a new matplotlib Axes object by default. An existing Axes object can also be passed to the plotting function with the `ax=` keyword argument. The frequencies at which the spectrum is by default selected based on the excitation energies, but can also be specified manually with the `frequencies=` keyword argument. Optional standard plotting arguments, such as color or line-style can be passed as additional keyword arguments, for example with `color='k'` to set a black line color.

The complete Python script for this example is:

```
import matplotlib.pyplot as plt

import daltonproject as dp

molecule = dp.Molecule(input_file='water.xyz')
basis = dp.Basis(basis='cc-pVDZ')

dft = dp.QCMethod('DFT', 'CAMB3LYP')
prop = dp.Property(excitation_energies=True)
prop.excitation_energies(states=6)
result = dp.dalton.compute(molecule, basis, dft, prop)
print('Excitation energies =', result.excitation_energies)
# Excitation energies = [ 7.708981 10.457775 11.631572 11.928512 14.928328 15.733272]

print('Oscillator strengths =', result.oscillator_strengths)
# Oscillator strengths = [0.0255 0.     0.1324 0.0883 0.0787 0.171 ]

ax = dp.spectrum.plot_one_photon_spectrum(result, color='k')
plt.savefig('1pa.svg')
```

The plot is generated by making a convolution of the excitation energies by the following equation:

$$\varepsilon(\omega) = \frac{e^2\pi^2 N_A}{\ln(10)2\pi\epsilon_0 n m_e c} \sum_i \frac{\omega f_i}{\omega_i} g(\omega, \omega_i, \gamma_i)$$

here N_a is Avogadro's constant, e is the elementary charge, m_e is the electron mass, c is the speed of light, ϵ_0 is the vacuum permittivity, n is the refractive index (approximated to be one), f_i is the calculated oscillator strength of the i th transition, and ω_i is the corresponding transition angular frequency.

It can be noted that another form is seen often in the literature:

$$\varepsilon(\omega) = \frac{e^2\pi^2 N_A}{\ln(10)2\pi\epsilon_0 n m_e c} \sum_i f_i g(\omega, \omega_i, \gamma_i)$$

This form just corresponds to approximating $\frac{\omega f_i}{\omega_i} \approx f_i$.

3.2 Two-Photon Absorption Spectrum

To create a two-photon absorption spectrum, we need to calculate excitation energies and two-photon strengths. First, we specify the molecule and basis set:

```
import matplotlib.pyplot as plt

import daltonproject as dp

molecule = dp.Molecule(input_file='water.xyz')
basis = dp.Basis(basis='cc-pVDZ')
```

The excitation energies and two-photon strengths can be calculated with many different methods, and for this example we will use TD-DFT with the CAM-B3LYP functional. In this case, we include the three lowest excitation energies and associated oscillator strengths.

```
dft = dp.QCMethod('DFT', 'CAMB3LYP')
prop = dp.Property(two_photon_absorption=True)
prop.two_photon_absorption(states=3)
result = dp.dalton.compute(molecule, basis, dft, prop)
print('Excitation energies =', result.excitation_energies)
# Excitation energies = [ 7.70898022 10.45777647 11.63157226]

print('Two-photon cross-sections =', result.two_photon_cross_sections)
# Two-photon cross-sections = [0.139 0.776 0.585]
```

To visualize the resulting spectrum, we can use the built-in plotting functionality from the spectrum module.

```
ax = dp.spectrum.plot_two_photon_spectrum(result, color='k')
plt.savefig('2pa.svg')
```

And the resulting spectrum looks like:

The complete Python script for this example is:

```
import matplotlib.pyplot as plt

import daltonproject as dp

molecule = dp.Molecule(input_file='water.xyz')
basis = dp.Basis(basis='cc-pVDZ')

dft = dp.QCMethod('DFT', 'CAMB3LYP')
prop = dp.Property(two_photon_absorption=True)
prop.two_photon_absorption(states=3)
result = dp.dalton.compute(molecule, basis, dft, prop)
print('Excitation energies =', result.excitation_energies)
# Excitation energies = [ 7.70898022 10.45777647 11.63157226]

print('Two-photon cross-sections =', result.two_photon_cross_sections)
# Two-photon cross-sections = [0.139 0.776 0.585]

ax = dp.spectrum.plot_two_photon_spectrum(result, color='k')
plt.savefig('2pa.svg')
```

3.3 X-ray Absorption Spectroscopy using Coupled-Cluster methods

Using CC methods, you have the option to employ the core-valence separation approximation to produce X-ray absorption spectra. Suppose we want to obtain such a spectrum for acrolein using the CC2 method:

```
import matplotlib.pyplot as plt

import daltonproject as dp

acrolein = dp.Molecule(input_file='acrolein.xyz')
basis = dp.Basis('STO-3G')

cc = dp.QCMethod('CC2')
exc = dp.Property(excitation_energies={'states': 6, 'cvseparation': [2, 3, 4]})
```

Here we choose to include six transitions, restricting the excitation channels to specific core orbitals specified in the `cvseparation` list.

The `cvseparation` list specifies the active orbitals. In general, they are specified for each irreducible representation (irrep). Suppose we have `cvseparation=[[1, 2, 3, 4], [2, 3], [3, 4], [1, 2, 3, 4, 5, 6]]`. This would read: from the first irrep use orbitals number 1, 2, 3, and 4; from the second use orbitals number 2 and 3; from the third use orbitals number 3 and 4; and from the fourth irrep use orbitals number 1 to 6. Note that this requires prior analysis of the orbitals using the same symmetry as in excitation energy calculations in order to identify the orbitals of interest.

If we want to include excitations from the lowest lying **s**-like orbitals of the carbons in acrolein, we thus enter `cvseparation=[2, 3, 4]`, as shown in the example above, skipping the first orbital which belongs to the oxygen.

Dalton Project will by default use any available CPU cores, and since Dalton is only uses MPI parallelization, it will set the number of MPI processes equal to number of CPU cores. However, CC in Dalton is not parallelized so we need to override the default settings:

```
settings = dp.ComputeSettings(mpi_num_procs=1)
```

Now we can run the calculation:

```
result = dp.dalton.compute(
    molecule=acrolein,
    basis=basis,
    qc_method=cc,
    properties=exc,
    compute_settings=settings,
)
```

and plot it:

```
print('Excitation energies =', result.excitation_energies)
# Excitation energies = [288.23439 288.37014 289.96145 293.82649 294.58115 296.95772]
```

The entire script to produce the plots is given below.

```
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import daltonproject as dp

acrolein = dp.Molecule(input_file='acrolein.xyz')
basis = dp.Basis('STO-3G')

cc = dp.QCMethod('CC2')
exc = dp.Property(excitation_energies={'states': 6, 'cvseparation': [2, 3, 4]})

settings = dp.ComputeSettings(mpi_num_procs=1)

result = dp.dalton.compute(
    molecule=acrolein,
    basis=basis,
    qc_method=cc,
    properties=exc,
    compute_settings=settings,
)

print('Excitation energies =', result.excitation_energies)
# Excitation energies = [288.23439 288.37014 289.96145 293.82649 294.58115 296.95772]
```

3.4 Vibrational IR and/or Raman Spectrum

To simulate vibrational IR and Raman spectra, we need vibrational frequencies and corresponding IR/Raman intensities. The vibrational frequencies are obtained from the molecular Hessian, i.e., the matrix containing second- derivatives with respect to nuclear displacements, while the dipole and polarizability gradients are needed to compute IR and Raman intensities, respectively. This is a two-step procedure where the basic properties are calculated first and subsequently used to derive the final quantities needed to plot the spectra. To proceed, we specify the molecule and level of theory:

```
import matplotlib.pyplot as plt

import daltonproject as dp

hf = dp.QCMethod(qc_method='HF')
basis = dp.Basis(basis='pcseg-0')
h2o2 = dp.Molecule(input_file='H2O2.xyz')
```

The properties should be evaluated at the equilibrium geometry. Therefore, we start by optimizing the geometry using LSDalton:

```
geo_opt = dp.Property(geometry_optimization=True)
opt_output = dp.lsdalton.compute(h2o2, basis, hf, geo_opt)
h2o2.coordinates = opt_output.final_geometry
```

Using the optimized geometry, we can proceed to calculate the Hessian, needed to derive vibrational frequencies, and dipole and polarizability gradients, from which the IR and Raman intensities are obtained. Raman intensities also depend on the frequency of the incident light. We may choose to compute the polarizability at this frequency (specified in hartree). The default is to compute static polarizability (i.e., a frequency of 0.0 hartree). Here we assume a light source with a wavelength of 633 nm corresponding to 0.07198 hartree:

```
props = dp.Property(hessian=True, dipole_gradients=True, polarizability_gradients={
↪'frequencies': [0.07198]})
prop_output = dp.lsdalton.compute(h2o2, basis, hf, props)
```

After the properties have been calculated, a vibrational analysis can be performed, which involves mass-weighting and diagonalizing the Hessian as well as projecting out translational and rotational degrees of freedom. Through this, we obtain the frequencies and the transformation matrix, allowing us to transform quantities from Cartesian coordinates to normal coordinates. If supplied, the dipole and polarizability gradients are immediately transformed and used to compute IR and Raman intensities, respectively:

```
vib_ana = dp.vibrational_analysis(molecule=h2o2,
                                  hessian=prop_output.hessian,
                                  dipole_gradients=prop_output.dipole_gradients,
                                  polarizability_gradients=prop_output.polarizability_
↪gradients)
```

The results can be used to plot spectra through the `spectrum` module. Since it uses Matplotlib, we can take advantage of its great flexibility. Here we are interested in comparing the IR and Raman spectra and therefore want to plot both in the same spectrum, which can be done as follows:

```
fig, ax1 = plt.subplots()
ax1 = dp.spectrum.plot_ir_spectrum(vib_ana, ax=ax1, color='blue', label='IR')
ax2 = ax1.twinx()
ax2 = dp.spectrum.plot_raman_spectrum(vib_ana, ax=ax2, color='red', label='Raman')
fig.legend(loc='center')
fig.savefig('ir_raman.svg')
```

The complete Python script for this example is:

```
import matplotlib.pyplot as plt

import daltonproject as dp

hf = dp.QCMethod(qc_method='HF')
basis = dp.Basis(basis='pcseg-0')
h2o2 = dp.Molecule(input_file='H2O2.xyz')

geo_opt = dp.Property(geometry_optimization=True)
opt_output = dp.lsdalton.compute(h2o2, basis, hf, geo_opt)
h2o2.coordinates = opt_output.final_geometry

props = dp.Property(hessian=True, dipole_gradients=True, polarizability_gradients={
↪'frequencies': [0.07198]})
prop_output = dp.lsdalton.compute(h2o2, basis, hf, props)

vib_ana = dp.vibrational_analysis(molecule=h2o2,
                                  hessian=prop_output.hessian,
                                  dipole_gradients=prop_output.dipole_gradients,
                                  polarizability_gradients=prop_output.polarizability_
↪gradients)

fig, ax1 = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
ax1 = dp.spectrum.plot_ir_spectrum(vib_ana, ax=ax1, color='blue', label='IR')
ax2 = ax1.twinx()
ax2 = dp.spectrum.plot_raman_spectrum(vib_ana, ax=ax2, color='red', label='Raman')
fig.legend(loc='center')
fig.savefig('ir_raman.svg')
```

CHAPTER

FOUR

CALCULATION OF NMR PARAMETERS

This tutorial explains how to calculate NMR parameters using the Dalton Project platform.

Note, that the quantum chemistry methods used in this tutorial may not be suitable for the particular problem that you want to solve. Moreover, the basis sets used here are minimal in order to allow you to progress fast through the tutorial and should under no circumstances be used in production calculations.

4.1 Shieldings

NMR shieldings can be obtained with, for example, HF, CASSCF, or DFT. To calculate shieldings, create a Property object where `nmr_shieldings` is set to `True`. A minimal example is shown below.

```
import daltonproject as dp

molecule = dp.Molecule(input_file='water.xyz')
basis = dp.Basis(basis='cc-pVDZ')

dft = dp.QCMethod('DFT', 'KT3')
prop = dp.Property(nmr_shieldings=True)
result = dp.dalton.compute(molecule, basis, dft, prop)
print('NMR shieldings =', result.nmr_shieldings)
# NMR shieldings = [389.5505  36.8488  36.8488]
```

CHAPTER

FIVE

VIBRATIONAL CORRECTIONS TO PROPERTIES

In order to compare experimental calculations with theoretical calculations at a high accuracy, one needs to account for the vibrational motion of the nuclei. With this implementation based on perturbation theory to second order (VPT2), one can account for this vibrational motion and correct the equilibrium property value ad hoc. The vibrational correction to the property P is given as:

$$\Delta^{\text{VPT2}}P = -\frac{1}{2}\sum_i \frac{1}{\omega_i}\frac{\partial P}{\partial q_i}\sum_j k_{ijj}\left(\nu_j + \frac{1}{2}\right) + \frac{1}{2}\sum_i \left(\nu_i + \frac{1}{2}\right)\frac{\partial^2 P}{\partial q_i^2}$$

where k_{ijj} is the cubic force constant in cm^{-1} and ω is the harmonic frequency in cm^{-1} with all derivatives evaluated at $q = 0$. For a zero-point vibrational correction (ZPVC), ν is set to 0. Temperature dependent corrections with a rotational contribution can also be computed using a Boltzmann averaging of states:

$$\Delta^{\text{VPT2}}P = -\frac{1}{4}\sum_i \frac{1}{\omega_i}\frac{\partial P}{\partial q_i}\left(\sum_j k_{ijj}\coth\left(\frac{hc\omega_j}{2kT}\right) - \frac{kT}{2\pi c}\frac{1}{\sqrt{hc\omega_i}}\sum_\alpha \frac{a_i^{\alpha\alpha}}{I_{\alpha\alpha}^e}\right)$$

$$+\frac{1}{4}\sum_i \frac{\partial^2 P}{\partial q_i^2}\coth\left(\frac{hc\omega_i}{2kT}\right)$$

where T is the temperature in Kelvin, k is the Boltzmann constant, h is the Planck constant, I is the tensor of the principle axes of inertia and a is the tensor of linear expansion coefficients of the moment of inertia in the normal coordinates. Cubic force constants, first and second derivatives of properties are derived numerically from displaced geometry calculations using either a 3/5 point stencil or a polynomial fitting. The stepsize used for the numerical analysis can be given as an argument and is unitless (reduced normal coordinates).

The following properties are currently supported (program dependent):

- NMR shieldings - Gaussian/Dalton
- Spin-spin coupling constants - Gaussian/Dalton
- Hyperfine coupling constants - Gaussian
- Static polarizability/frequency-dependent polarizabilities - Dalton/Gaussian
- Near static optical rotation/frequency-dependent optical rotations - Dalton/Gaussian

Tips:

- Performing a geometry optimisation is always advised (unless providing an already optimised geometry).
- The temperature can be given as an argument for calculations of temperature-dependent corrections.
- The fitting method (linear/polynomial) can be changed to improve the accuracy of the results.
- Temperature, fitting method, fitting order/point stencil can be changed, and the script can be rerun interactively without explicitly calling a quantum chemistry program.

- If the stepsize is changed or the input geometry is changed in any way, then all files need to be deleted.
- Polynomial fitting results can be inspected by changing the plot_polyfittings variable to "True" in the input file.
- It is advised to create an output folder for the generated files and run the script from there (as many files can be generated!)

Note, that the quantum chemistry methods used in this tutorial may not be suitable for the particular problem that you want to solve. Moreover, the basis set used here is minimal in order to allow you to progress fast through the tutorial and should under no circumstances be used in production calculations.

5.1 Correction to NMR shieldings using a linear fitting

```
import daltonproject as dp

#  Essential settings
hf = dp.QCMethod(qc_method='HF')
basis = dp.Basis(basis='STO-3G')
molecule = dp.Molecule(input_file='water.xyz')

# Settings if running multiple calculations in parallel using job farming
settings = dp.ComputeSettings(mpi_num_procs=1, jobs_per_node=1)

# optional - optimize the geometry
geo_opt = dp.Property(geometry_optimization=True)
opt_output = dp.dalton.compute(molecule, basis, hf, geo_opt, compute_settings=settings)
molecule.coordinates = opt_output.final_geometry

# Essential settings - compute the reference geometry Hessian
hess = dp.Property(hessian=True)
prop_output = dp.dalton.compute(molecule, basis, hf, hess, compute_settings=settings)
vib_prop = dp.Property(nmr_shieldings=True)

# Essential settings for computing vibrational correction
va_settings = dp.VibAvSettings(molecule=molecule,
                               property_program='dalton',
                               is_mol_linear=False,
                               hessian=prop_output.hessian,
                               property_obj=vib_prop,
                               stepsize=0.05,
                               differentiation_method='linear',
                               temperature=0,
                               linear_point_stencil=3)

# Essential settings - Instances of Molecule class for distorted geometries
molecules = []
for i in va_settings.file_list:
    molecules.append(dp.Molecule(input_file=i))

# Essential settings - Compute Hessians + property at distorted geometries
dalton_results_hess = []
dalton_results_nmr = []
```

(continues on next page)

(continued from previous page)

```
for mol in molecules:
    dalton_results_hess.append(dp.dalton.compute(mol, basis, hf, hess, compute_
↪settings=settings))
    dalton_results_nmr.append(dp.dalton.compute(mol, basis, hf, vib_prop, compute_
↪settings=settings))

# Essential settings - Instance of ComputeVibAvCorrection class containing the correction
result = dp.ComputeVibAvCorrection(dalton_results_hess, dalton_results_nmr, va_settings)

print('\nVibrational Correction to Property:', result.vibrational_corrections)
```

To start, we will calculate corrections to NMR shieldings for water. Here the cubic force constants and property derivatives will be calculated using a 3-point linear stencil with a stepsize of 0.05. An output file containing all corrected shieldings will be generated.

5.2 Correction to spin-spin coupling constants using a polynomial fitting

```
import daltonproject as dp

# Optional - set isotopes for each atom
atom_isotopes = ['O17', 'H1', 'H1']

#  Essential settings
hf = dp.QCMethod(qc_method='HF')
basis = dp.Basis(basis='STO-3G')
molecule = dp.Molecule(input_file='water.xyz', isotopes=atom_isotopes)

# Compute settings
settings = dp.ComputeSettings(mpi_num_procs=1, jobs_per_node=1)

# optional - optimize the geometry
geo_opt = dp.Property(geometry_optimization=True)
opt_output = dp.dalton.compute(molecule, basis, hf, geo_opt, compute_settings=settings)
molecule.coordinates = opt_output.final_geometry

# Essential settings - compute the reference geometry Hessian
hess = dp.Property(hessian=True)
prop_output = dp.dalton.compute(molecule, basis, hf, hess, compute_settings=settings)
vib_prop = dp.Property(spin_spin_couplings=True)

# Essential settings for computing vibrational correction
va_settings = dp.VibAvSettings(molecule=molecule,
                               property_program='dalton',
                               is_mol_linear=False,
                               hessian=prop_output.hessian,
                               property_obj=vib_prop,
                               stepsize=0.05,
                               differentiation_method='polynomial',
                               temperature=298,
```

(continues on next page)

(continued from previous page)

```
                                polynomial_fitting_order=3,
                                plot_polyfittings=True)

# Essential settings - Instances of Molecule class for distorted geometries
molecules = []
for i in va_settings.file_list:
    molecules.append(dp.Molecule(input_file=i, isotopes=atom_isotopes))

# Essential settings - Compute Hessians + property at distorted geometries
dalton_result_hess = []
dalton_result_nmr = []
for mol in molecules:
    dalton_result_hess.append(dp.dalton.compute(mol, basis, hf, hess, compute_
↪settings=settings))
    dalton_result_nmr.append(dp.dalton.compute(mol, basis, hf, vib_prop, compute_
↪settings=settings))

# Essential settings - Instance of ComputeVibAvCorrection class containing the correction
result = dp.ComputeVibAvCorrection(dalton_result_hess, dalton_result_nmr, va_settings)

print('\nVibrational Correction to Property:', result.vibrational_corrections)
```

This time we will calculate corrections to the spin-spin coupling constants for water. The cubic force constants and property derivatives will be calculated using a 3rd-order polynomial fitting with a stepsize of 0.05. Plots of the property derivatives fitting can also be generated for inspection to ensure the fitting is resonable. Here we also altered the temperature to 298K to calculate the temperature-dependent corrections. An output file containing all corrected spin-spin coupling constants will be generated.

Graphs of the polynomial derivative fitting for each coupling and associated mode is as shown:

5.3 Correction to frequency-dependent polarizabilities using a linear fitting

```
import daltonproject as dp

# Optional - set isotope for each atom
# Most abundant isotopes are used by default
atom_isotopes = ['O17', 'H1', 'H1']

#  Essential settings
hf = dp.QCMethod(qc_method='HF', scf_threshold=1e-10)
basis = dp.Basis(basis='STO-3G')
molecule = dp.Molecule(input_file='water.xyz', isotopes=atom_isotopes)

# Compute settings
settings = dp.ComputeSettings(mpi_num_procs=1, jobs_per_node=1)

# optional - optimize the geometry
geo_opt = dp.Property(geometry_optimization=True)
```

(continues on next page)

(continued from previous page)

```
opt_output = dp.dalton.compute(molecule, basis, hf, geo_opt, compute_settings=settings)
molecule.coordinates = opt_output.final_geometry

# Essential settings - compute the reference geometry Hessian
hess = dp.Property(hessian=True)
prop_output = dp.dalton.compute(molecule, basis, hf, hess, compute_settings=settings)
vib_prop = dp.Property(polarizabilities={'frequencies': [0.1, 0.2, 0.3]})

# Essential settings for computing vibrational correction
va_settings = dp.VibAvSettings(
    molecule=molecule,
    property_program='dalton',
    is_mol_linear=False,
    hessian=prop_output.hessian,
    property_obj=vib_prop,
    stepsize=0.05,
    differentiation_method='linear',
    temperature=0,
    # polynomial_fitting_order=3,
    linear_point_stencil=3,
    plot_polyfittings=True)

# Essential settings - Instances of Molecule class for distorted geometries
molecules = []
for i in va_settings.file_list:
    molecules.append(dp.Molecule(input_file=i, isotopes=atom_isotopes))

dalton_result_hess = []
dalton_result_nmr = []
for mol in molecules:
    dalton_result_hess.append(dp.dalton.compute(mol, basis, hf, hess, compute_
↪settings=settings))
    dalton_result_nmr.append(dp.dalton.compute(mol, basis, hf, vib_prop, compute_
↪settings=settings))

# Essential settings - Instance of ComputeVibAvCorrection class containing the correction
result = dp.ComputeVibAvCorrection(dalton_result_hess, dalton_result_nmr, va_settings)

print('\nVibrational Correction to the Property:', result.vibrational_corrections)
# result.vibrational_correction =  [0.126377, 0.132178, 0.15234, 0.199484]

import numpy as np  # noqa

np.testing.assert_allclose(result.vibrational_corrections, [0.126377, 0.132178, 0.15234,␣
↪0.199484], atol=1e-6)
```

This time, we will calculate corrections to the frequency-dependent polarizabilities for water where frequencies are in a.u. Static polarizabilities are also done by default when frequency-dependent polarizabilities are requested.

Optical rotations can be requested by changing

```
vib_prop = dp.Property(polarizabilities={'frequencies': [0.1, 0.2, 0.3]})
```

to

```
vib_prop = dp.Property(optical_rotations={'frequencies': [0.1, 0.2, 0.3]})
```

Ref: Faber, R.; Kaminsky, J.; Sauer, S. P. A. In Gas Phase NMR, Jackowski, K., Jaszunski, M., Eds.; Royal Society of Chemistry, London: 2016; Chapter 7, pp 218–266

CHAPTER

SIX

COMPLETE ACTIVE SPACE SELF-CONSISTENT FIELD

User guide for how to run complete active space self-consistent field (CASSCF) through DaltonProject.

6.1 Specifying CAS

To a run a CASSCF calculation through the DaltonProject, the first to do is to specify molecule and basis set:

```
import daltonproject as dp

molecule, basis = dp.mol_reader('pyridine.mol')
```

To run a CASSCF calculation, an initial guess of orbitals is needed, one of the simplest guesses is Hartree-Fock orbitals. A Hartree-Fock calculation can be run to generate these orbitals:

```
hf = dp.QCMethod('HF')
prop = dp.Property(energy=True)
hf_result = dp.dalton.compute(molecule, basis, hf, prop)
print('Hartree-Fock energy =', hf_result.energy)
# Hartree-Fock energy = -243.637999873274
print('Electrons =', hf_result.num_electrons)
# Electrons = 42
```

No special options are needed for the Hartree-Fock calculation. The simplest possible CAS would be to include the HOMO and LUMO orbitals, i.e. two active orbitals. Since pyridine has 42 electrons, pyridine will have 21 doubly occupied orbitals. If the HOMO is to be included in the active space there will then be 20 inactive orbitals left. This choice will correspond to a CAS(2,2) from canonical Hartree-Fock orbitals. The CASSCF calculation is now ready to be setup.

```
casscf = dp.QCMethod('CASSCF')
casscf.complete_active_space(2, 2, 20)
casscf.input_orbital_coefficients(hf_result.orbital_coefficients)
prop = dp.Property(energy=True)
casscf_result = dp.dalton.compute(molecule, basis, casscf, prop)
print('CASSCF energy =', casscf_result.energy)
# CASSCF energy = -243.642555771886
```

The specification `complete_active_space(2, 2, 20)` is of the form `complete_active_space(active electrons, active orbitals, inactive orbitals)`. The line `input_orbital_coefficients(hf_output.orbital_coefficients)` specifies the initial orbitals. Here the initial orbitals are taken from the previously run Hartree-Fock calculation.

The complete Python script for this example is:

```
import daltonproject as dp

molecule, basis = dp.mol_reader('pyridine.mol')

hf = dp.QCMethod('HF')
prop = dp.Property(energy=True)
hf_result = dp.dalton.compute(molecule, basis, hf, prop)
print('Hartree-Fock energy =', hf_result.energy)
# Hartree-Fock energy = -243.637999873274
print('Electrons =', hf_result.num_electrons)
# Electrons = 42

casscf = dp.QCMethod('CASSCF')
casscf.complete_active_space(2, 2, 20)
casscf.input_orbital_coefficients(hf_result.orbital_coefficients)
prop = dp.Property(energy=True)
casscf_result = dp.dalton.compute(molecule, basis, casscf, prop)
print('CASSCF energy =', casscf_result.energy)
# CASSCF energy = -243.642555771886
```

6.2 Picking CAS based on Natural Orbital Occupations

The Dalton Project provides tools to help select a CAS based on natural orbital occupations. As an example, let us use these tools on MP2 natural orbital occupations.

```
import daltonproject as dp

molecule, basis = dp.mol_reader('pyridine.mol')
molecule.analyze_symmetry()

mp2 = dp.QCMethod('MP2')
prop = dp.Property(energy=True)
mp2_result = dp.dalton.compute(molecule, basis, mp2, prop)
print(mp2_result.filename)
# b2773ec9e68d368d0c4b6889a5ec6299edc94101
print('MP2 energy =', mp2_result.energy)
# MP2 energy = -243.9889166118
```

Note also that symmetry has been enabled. Now the natural orbital occupations can be inspected:

```
import daltonproject as dp

mp2_output = dp.dalton.OutputParser('b2773ec9e68d368d0c4b6889a5ec6299edc94101')
nat_occs = mp2_output.natural_occupations
print(dp.natural_occupation.scan_occupations(nat_occs))
```

The method `scan_occupations(natural occupation numbers)` will per default list the 14 most important strongly occupied natural orbitals and weakly occupied natural orbitals. For this example, the print will produce the output below

```
Strongly occupied natural orbitals                                  Weakly occupied natural orbitals
Symmetry    Occupation    Change in occ.    Diff. to 2     Symmetry    Occupation    Change␣
↪in occ.
   2          1.9345         0.0000         0.0655           2          0.0698         0.
↪0000
   4          1.9367         0.0022         0.0633           4          0.0630         0.
↪0068
   2          1.9674         0.0307         0.0326           2          0.0325         0.
↪0304
   3          1.9766         0.0093         0.0234           3          0.0218         0.
↪0107
   1          1.9782         0.0016         0.0218           1          0.0207         0.
↪0011
   3          1.9788         0.0006         0.0212           3          0.0180         0.
↪0027
   1          1.9832         0.0044         0.0168           1          0.0175         0.
↪0005
   3          1.9835         0.0003         0.0165           1          0.0166         0.
↪0009
   1          1.9862         0.0026         0.0138           3          0.0128         0.
↪0038
   1          1.9870         0.0008         0.0130           1          0.0126         0.
↪0002
   3          1.9886         0.0016         0.0114           1          0.0119         0.
↪0007
   1          1.9895         0.0009         0.0105           3          0.0119         0.
↪0000
   3          1.9908         0.0013         0.0092           3          0.0118         0.
↪0001
   1          1.9924         0.0015         0.0076           1          0.0105         0.
↪0012
```

The scan of the natural occupation numbers can aid the user in picking a suitable active space. In this example, the minimal active space has been highlighted in yellow. This active space has been picked as the minimal active space since the most significant jump in natural occupations happens to the next orbitals. The CASSCF calculation can now be setup.

```
import daltonproject as dp

molecule, basis = dp.mol_reader('pyridine.mol')
molecule.analyze_symmetry()
mp2_output = dp.dalton.OutputParser('b2773ec9e68d368d0c4b6889a5ec6299edc94101')

casscf = dp.QCMethod('CASSCF')
prop = dp.Property(energy=True)
nat_occs = mp2_output.natural_occupations
electrons, cas, inactive = dp.natural_occupation.pick_cas_by_thresholds(nat_occs, 1.94,␣
↪0.06)
casscf.complete_active_space(electrons, cas, inactive)
casscf.input_orbital_coefficients(mp2_output.orbital_coefficients)
casscf_result = dp.dalton.compute(molecule, basis, casscf, prop)
print('CASSCF energy =', casscf_result.energy)
# CASSCF energy = -243.699847108852
```

Since we determined the proper active space, based on natural orbital occupations, the number of active electrons, active orbitals, and inactive orbitals does not need to be manually specified. Dalton Project can determine these quantities based on the threshold we pick for the strongly occupied natural orbitals and weakly occupied natural orbitals. The method `pick_cas_by_thresholds(nat_occs, 1.94, 0.06)` is of the form `pick_cas_by_thresholds(natural occupation numbers, strongly occupied threshold, weakly occupied threshold)`. All strongly occupied natural orbitals below the threshold will be included in the active space together with all weakly occupied natural orbitals above the threshold. For this specific example `electrons=4`, `cas=[0, 2, 0, 2]` and `inactive=[11, 1, 7, 0]`.

CHAPTER

SEVEN

SHORT-RANGE DENSITY FUNCTIONAL THEORY

In the short-range density functional theory model (srDFT), the total energy is calculated as a long-range interaction by wavefunction theory and a short-range by density functional theory. The philosophy of this method is to use the good semi-local description of DFT, and then describe the long-range with wavefunction methods.

$$E_{\text{WF-srDFT}} = \left\langle \Psi^{\text{lr}} \left| -\frac{1}{2}\nabla^2 + \frac{g(r_{12},\mu)}{r_{12}} \right| \Psi^{\text{lr}} \right\rangle + E^{\text{sr}}_{\text{Hxc}}\left[\rho_{\Psi^{\text{lr}}}\right] + \int_\Omega \rho_{\Psi^{\text{lr}}}(r) v_{\text{ext}}(r) \mathrm{d}r$$

Here $g(r_{12},\mu)$ is a range-separation function, with a range-separation parameter μ. The usual form of the range-separation function is,

$$g(r_{12},\mu) = \frac{\operatorname{erf}(\mu r_{12})}{r_{12}}$$

It can be noted that KS-DFT and regular wavefunction theory is recovered in the limiting cases of $\mu \to 0$ and $\mu \to \infty$, respectively.

$$\lim_{\mu\to 0} E_{\text{WF-srDFT}} = \left\langle \phi \left| -\frac{1}{2}\nabla^2 \right| \phi \right\rangle + E_{\text{Hxc}}\left[\rho_\phi\right] + \int_\Omega \rho_\phi(r) v_{\text{ext}}(r) \mathrm{d}r = E_{\text{KS-DFT}}$$

and,

$$\lim_{\mu\to \infty} E_{\text{WF-srDFT}} = \left\langle \Psi \left| -\frac{1}{2}\nabla^2 + \frac{1}{r_{12}} \right| \Psi \right\rangle + \int_\Omega \rho_\Psi(r) v_{\text{ext}}(r) \mathrm{d}r = E_{\text{WF}}$$

7.1 HF–srDFT

In Hartree–Fock srDFT both the exchange and correlation functional are range-separated. This makes HF–srDFT similar to long-range corrected DFT (LRC-DFT) with the difference that only the exchange functional is range-separated in LRC-DFT. An HF–srDFT calculation can be run as:

```
import daltonproject as dp

molecule, basis = dp.mol_reader('water.mol')
hfsrdft = dp.QCMethod('HFsrDFT', 'SRPBEGWS')
hfsrdft.range_separation_parameter(0.4)
prop = dp.Property(energy=True)
hfsrdft_result = dp.dalton.compute(molecule, basis, hfsrdft, prop)
print('HFsrPBE energy =', hfsrdft_result.energy)
# HFsrPBE energy = -73.502309263107
```

The specification `QCMethod('HFsrDFT', 'SRPBEGWS')` signifies that an HF–srDFT calculation will be performed with the short-range PBE functional of Goll, Werner and, Stoll. Furthermore, `range_separation_parameter(0.4)` sets the range-separation parameter to $0.4\ \mathrm{bohr}^{-1}$, which is the recommended value.

The HF–srDFT method can also be used to do LRC-DFT calculations, in which case the range-separation is only applied to the exchange fucntional.

```
import daltonproject as dp

molecule, basis = dp.mol_reader('water.mol')
hfsrdft = dp.QCMethod('HFsrDFT', 'LRCPBEGWS')
hfsrdft.range_separation_parameter(0.4)
prop = dp.Property(energy=True)
hfsrdft_result = dp.dalton.compute(molecule, basis, hfsrdft, prop)
print('LRCPBE energy =', hfsrdft_result.energy)
# LRCPBE energy = -73.527807924557
```

The setup is the same as for the HF–srDFT calculation but the functional have been changed `QCMethod('HFsrDFT', 'LRCPBEGWS')`.

7.2 MP2–srDFT

MP2 with srDFT is also supported. The main reason to use this method is as an orbital generator for CAS–srDFT calculations. MP2–srDFT is setup very similar to HF–srDFT.

```
import daltonproject as dp

molecule, basis = dp.mol_reader('pyridine.mol')
molecule.analyze_symmetry()
mp2srdft = dp.QCMethod('MP2srDFT', 'SRPBEGWS')
mp2srdft.range_separation_parameter(0.4)
prop = dp.Property(energy=True)
mp2srdft_result = dp.dalton.compute(molecule, basis, mp2srdft, prop)
print(mp2srdft_result.filename)
# 282603d082ffdea3ff8ef1432f5dfe1f69c12db7
print('MP2srPBE energy =', mp2srdft_result.energy)
# MP2srPBE energy = -244.7774549828
```

Note here, that MP2srDFT have been specified instead of HFsrDFT `QCMethod('MP2srDFT', 'SRPBEGWS')`.

If the natural orbital occupations are inspected it can be seen that they are closer to 2 or 0 than for the full-range MP2 in *Picking CAS based on Natural Orbital Occupations*.

```
import daltonproject as dp

mps2srdft_output = dp.dalton.OutputParser('282603d082ffdea3ff8ef1432f5dfe1f69c12db7')
nat_occs = mps2srdft_output.natural_occupations
print(dp.natural_occupation.scan_occupations(nat_occs))
```

Inspecting the natural orbital occupations uses the same methods as for full-range MP2.

```
Strongly occupied natural orbitals                    Weakly occupied natural orbitals
Symmetry   Occupation   Change in occ.   Diff. to 2    Symmetry   Occupation   Change␣
↪in occ.
```

(continues on next page)

(continued from previous page)

```
   4          1.9879         0.0000         0.0121          2          0.0124         0.
↪0000
   2          1.9882         0.0003         0.0118          4          0.0120         0.
↪0004
-------------------------------------------------------          3          0.0026         0.
↪0094
-------------------------------------------------------          1          0.0023         0.
↪0003
-------------------------------------------------------          1          0.0020         0.
↪0002
```

7.3 CAS–srDFT

Doing CAS calculations with srDFT also requires the choosing of an active-space. The same methods presented in *Complete Active Space Self-Consistent Field* can be used for CAS–srDFT also. As an example of an CAS–srDFT calculation, lets start with the MP2 natural orbitals calculated in *MP2–srDFT*.

```
import daltonproject as dp

molecule, basis = dp.mol_reader('pyridine.mol')
molecule.analyze_symmetry()
cassrdft = dp.QCMethod('CASsrDFT', 'SRPBEGWS')
cassrdft.range_separation_parameter(0.4)
prop = dp.Property(energy=True)
mp2srdft_output = dp.dalton.OutputParser('282603d082ffdea3ff8ef1432f5dfe1f69c12db7')
nat_occs = mp2srdft_output.natural_occupations
electrons, cas, inactive = dp.natural_occupation.pick_cas_by_thresholds(nat_occs, 1.99,␣
↪0.01)
cassrdft.complete_active_space(electrons, cas, inactive)
cassrdft.input_orbital_coefficients(mp2srdft_output.orbital_coefficients)
cassrdft_result = dp.dalton.compute(molecule, basis, cassrdft, prop)
print('CASsrDFT energy =', cassrdft_result.energy)
# CASsrDFT energy = -244.761631596478
```

CHAPTER

EIGHT

MOLECULE

Molecule module.

class daltonproject.molecule.**Molecule**(*atoms: str | None = None*, *input_file: str | None = None*, *charge: int = 0*, *symmetry: bool = False*, *multiplicity: int = 1*, *isotopes: Sequence[str] | None = None*)

Molecule class.

analyze_symmetry() → None

Analyze the molecular symmetry. Note that this translates the molecule's center of mass to the origin.

atoms(*atoms: str*) → None

Specify atoms.

Parameters
atoms – Atoms specified with elements and coordinates (and optionally labels).

property coordinates: ndarray

Atomic coordinates.

gau(*filename: str*) → None

Read Gaussian input file.

Parameters
filename – Name of Gaussian input file containing charge, atoms, and coordinates.

isotopes_assign(*isotopes: Sequence[str]*) → None

Assign isotope order according to abundance.

Parameters
isotopes – Isotopes of the nuclei.

mol(*filename: str*) → None

Read dalton mol file.

Parameters
filename – Name of mol file containing charge, atoms, and coordinates.

property num_atoms: int

Return the number of atoms in the molecule.

xyz(*filename: str*) → None

Read xyz file.

Parameters
filename – Name of xyz file containing atoms and coordinates.

CHAPTER

NINE

BASIS

Basis module.

class daltonproject.basis.**Basis**(*basis: dict[str, str] | str*, *ri: dict[str, str] | str | None = None*, *admm: dict[str, str] | str | None = None*)

Specify the AO basis.

write(*basis_format: str = 'dalton'*) → None

Write basis set to file.

Parameters
basis_format – Format of the basis set file.

daltonproject.basis.**get_atom_basis**(*basis: Mapping[str, str] | str*, *num_atoms: int*, *labels: Sequence[str]*) → list[str]

Process basis set input.

Parameters

- **basis** – Basis set.
- **num_atoms** – Number of atoms.
- **labels** – Atom labels.

Returns
List containing the basis set of each individual atoms.

daltonproject.basis.**validate_basis**(*basis: Mapping[str, str] | str*) → None

Validate basis set input.

Parameters
basis – Basis set.

CHAPTER

TEN

QCMETHOD

Quantum chemistry methods module.

class daltonproject.qcmethod.**QCMethod**(*qc_method: str*, *xc_functional: str | None = None*, *coulomb: str | None = None*, *exchange: str | None = None*, *scf_threshold: float | None = None*, *exact_exchange: float | None = None*, *environment: str | None = None*)

Specifies the QC method, including all associated settings.

complete_active_space(*num_active_electrons: int*, *num_cas_orbitals: list[int] | int*, *num_inactive_orbitals: list[int] | int*) → None

Specify complete active space.

Parameters

- **num_active_electrons** – Number of active electrons. // Maybe this should be inferred just from the num_inactive_orbitals?
- **num_cas_orbitals** – List of number of orbitals in complete active space.
- **num_inactive_orbitals** – List of number of inactive (doubly occupied) orbitals.

coulomb(*coulomb: str*) → None

Specify name of Coulomb approximation.

Parameters
coulomb – Name of approximation to use for the Coulomb contribution.

environment(*environment: str*) → None

Specify the environment model.

Parameters
environment – Name of the environment model.

exact_exchange(*exact_exchange: float*) → None

Specify amount of Hartree-Fock-like exchange in DFT calculation.

Parameters
exact_exchange – Coefficient specifying amount of Hartree-Fock-like exchange, where 0.0 is only DFT exchange and 1.0 is only Hartree-Fock-like exchange.

exchange(*exchange: str*) → None

Specify name of exchange approximation.

Parameters
exchange – Name of approximation to use for the exchange contribution.

input_orbital_coefficients(*orbitals: np.ndarray | Mapping[int, np.ndarray]*) → None

Specify starting guess for orbitals.

Parameters
orbitals – Orbital coefficients.

integral_family(*integral_family: bool*) → None

Specify if internal integral intermediates should be reused for integral_family basis sets.

Parameters
integral_family – Reuse internal integral intermediates for shared-exponent angular blocks.

qc_method(*qc_method: str*) → None

Specify quantum chemistry method.

Parameters
qc_method – Name of quantum chemical method.

range_separation_parameter(*mu: float*) → None

Specify value of range separation parameter.

The range separation parameter is used in srDFT.

Parameters
mu – Range separation parameter in unit of a.u.^-1.

scf_threshold(*threshold: float*) → None

Specify convergence threshold for SCF calculations.

Parameters
threshold – Threshold for SCF convergence in Hartree.

target_state(*state: int*, *symmetry: int = 1*) → None

Specify target state for state-specific calculations.

Parameters

- **state** – Which state to be targeted, counting from 1.
- **symmetry** – Symmetry of the target state.

xc_functional(*xc_functional: str*) → None

Specify exchange-correlation functional to use with DFT methods.

Parameters
xc_functional – Name of exchange-correlation functional.

CHAPTER

ELEVEN

PROPERTY

Property module.

class daltonproject.property.**Property**(*energy: bool = False*, *dipole: bool = False*, *polarizabilities: bool | dict = False*, *first_hyperpolarizability: bool = False*, *gradients: bool | dict = False*, *hessian: bool | dict = False*, *geometry_optimization: bool | dict = False*, *transition_state: bool | dict = False*, *excitation_energies: bool | dict = False*, *two_photon_absorption: bool | dict = False*, *hyperfine_couplings: bool | dict = False*, *nmr_shieldings: bool = False*, *spin_spin_couplings: bool = False*, *dipole_gradients: bool = False*, *polarizability_gradients: bool | dict = False*, *optical_rotations: bool | dict = False*, *symmetry: bool = False*)

Define properties to be computed.

dipole() → None

Dipole moment.

dipole_gradients() → None

Compute gradients of the dipole with respect to nuclear displacements in Cartesian coordinates.

energy() → None

Energy.

excitation_energies(*states: int | Sequence[int] = 5*, *triplet: bool = False*, *cvseparation: Sequence[int] | Sequence[Sequence[int]] | None = None*) → None

Compute excitation energies.

Parameters

- **states** – Number of states.
- **triplet** – Turns on triplet excitations. Excitations are singlet as default.
- **cvseparation** – Core-valence separation. Specify the active orbitals of each symmetry species (irrep), giving the number of the orbitals. For example, [[1, 2], [1], [0]] specifies that orbitals 1 and 2 from the first irrep are active, orbital 1 from the second irrep, and no orbitals from the third and last irrep.

first_hyperpolarizability() → None

First hyperpolarizability (beta).

geometry_optimization(*method: str = 'BFGS'*) → None

Geometry optimization.

Parameters
method – Geometry optimization method.

gradients(*method: str = 'analytic'*) → None

Compute molecular gradients.

Parameters
method – Method for computing gradients.

hessian(*method: str = 'analytic'*) → None

Compute molecular Hessian.

Parameters
method – Method for computing Hessian.

hyperfine_couplings(*atoms: Sequence[int]*) → None

Compute hyperfine coupling constants.

Parameters
atoms – List of atoms by index.

nmr_shieldings() → None

Compute nuclear magnetic shielding constants.

optical_rotations(*frequencies: Sequence[float] | None = None*) → None

Optical rotations.

Optical rotations will be calculated for a given list of frequencies. If no frequencies are given, it will be calculated only at the static limit (0.001 a.u.).

Parameters
frequencies – list of frequencies given in a.u.

polarizabilities(*frequencies: Sequence[float] | None = None*) → None

Polarizabilities (alpha).

Polarizabilities will be calculated at a given list of frequencies. If no frequencies are given then the static polarizability will be calculated.

Parameters
frequencies – list of frequencies in a.u.

polarizability_gradients(*frequencies: float | Sequence[float] = 0.0*) → None

Compute gradients of the polarizability with respect to nuclear displacements in Cartesian coordinates.

Parameters
frequencies – Frequencies at which the polarizability will be evaluated (hartree).

spin_spin_couplings() → None

Compute spin-spin coupling constants.

symmetry() → None

Use molecular symmetry in program calculation.

transition_state() → None

Geometry optimization for transition states.

two_photon_absorption(*states: int | Sequence[int] = 5*) → None

Compute two-photon absorption strengths.

Parameters
states – Number of states.

CHAPTER

TWELVE

PROGRAM

Classes and functions related to interfaces to executable programs.

Defines methods that are required for interfaces to external programs that will run as executables and produce output that will be parsed.

class daltonproject.program.**ComputeSettings**(*work_dir: str | None = None*, *scratch_dir: str | None = None*, *node_list: list[str] | None = None*, *jobs_per_node: int = 1*, *mpi_command: str | None = None*, *mpi_num_procs: int | None = None*, *omp_num_threads: int | None = None*, *memory: int | None = None*, *comm_port: int | None = None*, *launcher: str | None = None*, *slurm: bool = False*, *slurm_partition: str = ''*, *slurm_walltime: str = ''*, *slurm_account: str = ''*)

Compute settings to be used by Program.compute() implementations.

The settings can be given as arguments during initialization. Defaults are used for any missing argument(s).

property comm_port

Return communication port.

property jobs_per_node: int

Return the number of jobs per node.

property launcher: str | None

Return launcher command.

property memory: int

Return the total amount of memory in MB per node.

property mpi_command: str

Return MPI command.

property mpi_num_procs: int

Return the number of MPI processes.

property node_list: list[str]

Return the list of nodes.

property num_nodes: int

Return the number of nodes.

property omp_num_threads: int

Return the number of OpenMP threads (per MPI process).

property scratch_dir: str

Return scratch directory.

property slurm: bool

Return slurm bool.

property slurm_account: str

Return account.

property slurm_partition: str

Return partition.

property slurm_walltime: str

Return SLURM walltime.

property work_dir: str

Return work directory.

class daltonproject.program.**ExcitationEnergies**(*excitation_energies: np.ndarray*, *excitation_energies_per_sym: dict[str, np.ndarray]*)

Data structure for excitation energies.

excitation_energies: ndarray

Alias for field number 0

excitation_energies_per_sym: dict[str, numpy.ndarray]

Alias for field number 1

class daltonproject.program.**HyperfineCouplings**(*polarization: np.ndarray*, *direct: np.ndarray*, *total: np.ndarray*)

Data structure for hyperfine couplings.

direct: ndarray

Alias for field number 1

polarization: ndarray

Alias for field number 0

total: ndarray

Alias for field number 2

class daltonproject.program.**NumBasisFunctions**(*tot_num_basis_functions: int*, *num_basis_functions_per_sym: list[int]*)

Data structure for the number of basis functions.

num_basis_functions_per_sym: list[int]

Alias for field number 1

tot_num_basis_functions: int

Alias for field number 0

class daltonproject.program.**NumOrbitals**(*tot_num_orbitals: int*, *num_orbitals_per_sym: list[int]*)

Data structure for the number of orbitals.

num_orbitals_per_sym: list[int]

Alias for field number 1

tot_num_orbitals: int

Alias for field number 0

class daltonproject.program.**OpticalRotations**(*frequencies: np.ndarray*, *values: np.ndarray*)

Data structure for optical rotations.

frequencies: ndarray

Alias for field number 0

values: ndarray

Alias for field number 1

class daltonproject.program.**OrbitalCoefficients**(*orbital_coefficients: np.ndarray*, *orbital_coefficients_per_sym: dict[str, np.ndarray]*)

Data structure for molecular orbital coefficients.

orbital_coefficients: ndarray

Alias for field number 0

orbital_coefficients_per_sym: dict[str, numpy.ndarray]

Alias for field number 1

class daltonproject.program.**OrbitalEnergies**(*orbital_energies: np.ndarray*, *orbital_energies_per_sym: dict[str, np.ndarray]*)

Data structure for molecular orbital energies.

orbital_energies: ndarray

Alias for field number 0

orbital_energies_per_sym: dict[str, numpy.ndarray]

Alias for field number 1

class daltonproject.program.**OrbitalEnergy**(*orbital_energy: float*, *symmetry: str*)

Data structure for a single molecular orbital energy.

orbital_energy: float

Alias for field number 0

symmetry: str

Alias for field number 1

class daltonproject.program.**OscillatorStrengths**(*oscillator_strengths: np.ndarray*, *oscillator_strengths_per_sym: dict[str, np.ndarray]*)

Data structure for oscillator strengths.

oscillator_strengths: ndarray

Alias for field number 0

oscillator_strengths_per_sym: dict[str, numpy.ndarray]

Alias for field number 1

class daltonproject.program.**OutputParser**

Abstract OutputParser class.

The implementation of this class defines methods for parsing output files produced by a program.

property dipole: ndarray

Retrieve dipole.

property dipole_gradients: ndarray

Retrieve gradients of dipole moment with respect to nuclear displacements in Cartesian coordinates.

property electronic_energy: float

Retrieve electronic energy.

property energy: float

Retrieve energy.

property excitation_energies: ndarray

Retrieve excitation energies.

property filename: str

Retrieve filename.

property final_geometry: ndarray

Retrieve final geometry.

property first_hyperpolarizability: ndarray

Retrieve the first hyperpolarizability.

property gradients: ndarray

Retrieve gradients with respect to nuclear displacements in Cartesian coordinates.

property hessian: ndarray

Retrieve second derivatives with respect to nuclear displacements in Cartesian coordinates.

property homo_energy: *OrbitalEnergy*

Retrieve the highest occupied molecular orbital energy.

property hyperfine_couplings: *HyperfineCouplings*

Retrieve hyperfine couplings.

property lumo_energy: *OrbitalEnergy*

Retrieve the lowest unoccupied molecular orbital energy.

property mo_energies: *OrbitalEnergies*

Retrieve molecular orbital energies.

property natural_occupations: np.ndarray | dict[int, np.ndarray]

Retrieve natural orbital occupations.

property nmr_shieldings: ndarray

Retrieve NMR shieldings.

property nuclear_repulsion_energy: float

Retrieve nuclear-repulsion energy.

property num_basis_functions: int | *NumBasisFunctions*

Retrieve the number of basis functions.

property num_electrons: int

Retrieve the number of electrons.

property num_orbitals: int | *NumOrbitals*

Retrieve the number of orbitals.

property optical_rotations: ndarray

Retrieve optical rotations.

property orbital_coefficients: np.ndarray | dict[int, np.ndarray]

Retrieve molecular orbital coefficients.

property oscillator_strengths: ndarray

Retrieve oscillator strengths.

property polarizabilities: ndarray

Retrieve polarizabilities.

property polarizability_gradients: *PolarizabilityGradients*

Retrieve gradients of the polarizability with respect to nuclear displacements in Cartesian coordinates.

property spin_spin_couplings: ndarray

Retrieve spin-spin couplings.

property spin_spin_labels: list[str]

Retrieve spin-spin labels.

property two_photon_cross_sections: ndarray

Retrieve two-photon cross-sections.

property two_photon_strengths: ndarray

Retrieve two-photon strengths.

property two_photon_tensors: ndarray

Retrieve two-photon tensors.

class daltonproject.program.**Polarizabilities**(*frequencies: np.ndarray*, *values: np.ndarray*)

Data structure for polarizabilities.

frequencies: ndarray

Alias for field number 0

values: ndarray

Alias for field number 1

class daltonproject.program.**PolarizabilityGradients**(*frequencies: np.ndarray*, *values: np.ndarray*)

Data structure for polarizability gradients.

frequencies: ndarray

Alias for field number 0

values: ndarray

Alias for field number 1

class daltonproject.program.**Program**

Program base class to enforce program interface.

The interface must implement a *compute()* method that prepares and executes a calculation, and returns an instance of the OutputParser class.

abstract classmethod compute(*molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *properties:* Property, *environment: Environment | None = None*, *compute_settings:* ComputeSettings | *None = None*, *filename: str | None = None*, *force_recompute: bool = False*) → *OutputParser*

Run a calculation.

Parameters

- **molecule** – Molecule on which a calculations is performed. This can also be an atom, a fragment, or any collection of atoms.
- **basis** – Basis set to use in the calculation.
- **qc_method** – Quantum chemistry method, e.g., HF, DFT, or CC, and associated settings.
- **properties** – Properties of molecule to be calculated, geometry optimization, excitation energies, etc.
- **environment** – Environment description is missing.
- **compute_settings** – Settings for the calculation, e.g., number of MPI processes and OpenMP threads, work and scratch directory, etc.
- **filename** – Optional user-specified filename that will be used for input and output files. If not specified a name will be generated as a hash of the input.
- **force_recompute** – Recompute even if the output files already exist.

Returns

OutputParser instance with a reference to the filename used in the calculation.

CHAPTER

THIRTEEN

DALTON

Initialize Dalton interface module.

class daltonproject.dalton.**OutputParser**(*filename: str*)

Parse the Dalton output files.

property dipole: ndarray

Extract dipole moment from the Dalton output file.

property electronic_energy: float

Extract the electronic energy from the Dalton output file.

property energy: float

Extract the final energy from the Dalton output file.

property excitation_energies: ndarray

Extract one-photon excitation energies from the Dalton output file.

property filename: str

Name of the the Dalton output files without the extension.

property final_geometry: ndarray

Extract final geometry from Dalton output archive (.tar.gz).

property first_hyperpolarizability: ndarray

Extract the first hyperpolarizability from the Dalton output file.

property gradients: ndarray

Extract molecular gradients from the Dalton output file.

property hessian: ndarray

Extract Hessian matrix from DALTON.HES file in the Dalton output archive (.tar.gz).

property homo_energy: *OrbitalEnergy*

Extract the highest occupied molecular orbital energy from the the Dalton output file.

property hyperfine_couplings: *HyperfineCouplings*

Extract hyperfine couplings from the Dalton output file.

property lumo_energy: *OrbitalEnergy*

Extract the lowest unoccupied molecular orbital energy from the the Dalton output file.

property mo_energies: *OrbitalEnergies*

Extract molecular orbital energies from the Dalton output file.

property natural_occupations: dict[int, numpy.ndarray]

Extract natural orbital occupation numbers from the Dalton output file.

property nmr_shieldings: ndarray

Extract NMR shielding constants from the DALTON.CM in the DALTON output archive (.tar.gz).

property nuclear_repulsion_energy: float

Extract the nuclear repulsion energy from the Dalton output file.

property num_basis_functions: *NumBasisFunctions*

Extract number of basis functions from the Dalton output file.

property num_electrons: int

Extract the number of electrons from the Dalton output file.

property num_orbitals: *NumOrbitals*

Extract the number of orbitals from the Dalton output file.

This number can be smaller than the number of basis functions, in case of linear dependencies.

property optical_rotations: *OpticalRotations*

Extract optical rotations from the Dalton output file.

property orbital_coefficients: dict[int, np.ndarray] | np.ndarray

Extract orbital coefficients from Dalton output archive (.tar.gz).

This assumes that .PUNCHOUTORBITALS is used in the Dalton input file.

property oscillator_strengths: ndarray

Extract one-photon oscillator strengths from the Dalton output file.

property polarizabilities: *Polarizabilities*

Extract polarizabilities from the Dalton output file.

property spin_spin_couplings: ndarray

Extract contributions to spin-spin coupling constants from output file.

property spin_spin_labels: list[str]

Extract spin-spin coupling labels from output file.

property two_photon_cross_sections: ndarray

Extract two-photon cross-sections from the Dalton output file.

property two_photon_strengths: ndarray

Extract two-photon transition strengths from the Dalton output file.

property two_photon_tensors: ndarray

Extract two-photon transition tensors from the Dalton output file.

daltonproject.dalton.**compute**(*molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *properties:* Property, *environment: Environment | None = None*, *compute_settings:* ComputeSettings | *None = None*, *filename: str | None = None*, *force_recompute: bool = False*) → *OutputParser*

Run a calculation using the Dalton program.

Parameters

- **molecule** – Molecule on which a calculations is performed. This can also be an atom, a fragment, or any collection of atoms.

- **basis** – Basis set to use in the calculation.
- **qc_method** – Quantum chemistry method, e.g., HF, DFT, or CC, and associated settings.
- **properties** – Properties of molecule to be calculated, geometry optimization, excitation energies, etc.
- **environment** – Environment description missing
- **compute_settings** – Settings for the calculation, e.g., number of MPI processes and OpenMP threads, work and scratch directory, etc.
- **filename** – Optional user-specified filename that will be used for input and output files. If not specified a name will be generated as a hash of the input.
- **force_recompute** – Recompute even if the output files already exist.

Returns

OutputParser instance that contains the filename of the output produced in the calculation and can be used to extract results from the output.

daltonproject.dalton.**compute_farm**(*molecules: Sequence[*Molecule*]*, *basis:* Basis, *qc_method:* QCMethod, *properties:* Property, *compute_settings:* ComputeSettings | *None* = *None*, *filenames: Sequence[str]* | *None* = *None*, *force_recompute: bool* = *False*) → list[*OutputParser*]

Run a series of calculations using the Dalton program.

Parameters

- **molecules** – List of molecules on which calculations are performed.
- **basis** – Basis set to use in the calculations.
- **qc_method** – Quantum chemistry method, e.g., HF, DFT, or CC, and associated settings used for all calculations.
- **properties** – Properties of molecule to be calculated, geometry optimization, excitation energies, etc., computed for all molecules.
- **compute_settings** – Settings for the calculations, e.g., number of MPI processes and OpenMP threads, work and scratch directory, and more.
- **filenames** – Optional list of user-specified filenames that will be used for input and output files. If not specified names will be generated as a hash of the individual inputs.
- **force_recompute** – Recompute even if the output files already exist.

Returns

List of OutputParser instances each of which contains the filename of the output produced in the calculation and can be used to extract results from the output.

CHAPTER

FOURTEEN

GAUSSIAN

Initialize Gaussian interface module.

class daltonproject.gaussian.**OutputParser**(*filename: str*)

Parse the gaussian output files.

property energy: float

Extract energy from the formatted gaussian output file.

property filename: str

Retrieve filename.

property final_geometry: ndarray

Extract final geometry from the gaussian output file.

property hessian: ndarray

Extract Hessian matrix from the gaussian formatted check point file.

property hyperfine_couplings: ndarray

Extract hyperfine couplings in atomic units from gaussian formatted checkpoint file.

property nmr_shieldings: ndarray

Extract NMR shielding constants from the gaussian formatted checkpoint file.

property optical_rotations: *OpticalRotations*

Extract optical rotation from gaussian output file.

property polarizabilities: *Polarizabilities*

Extract polarizability tensor(s) from the gaussian formatted checkpoint file.

property spin_spin_couplings: ndarray

Extract contributions to spin-spin coupling constants from gaussian output file.

property spin_spin_labels: list[str]

Extract spin-spin coupling labels from gaussian output file.

daltonproject.gaussian.**compute**(*molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *properties:* Property, *environment: Environment | None = None*, *compute_settings:* ComputeSettings | *None = None*, *filename: str | None = None*, *force_recompute: bool = False*) → *OutputParser*

Run a calculation using the Gaussian program.

Parameters

- **molecule** – Molecule on which a calculations is performed. This can also be an atom, a fragment, or any collection of atoms.

- **basis** – Basis set to use in the calculation.
- **qc_method** – Quantum chemistry method, e.g., HF, DFT, or CC, and associated settings.
- **properties** – Properties of molecule to be calculated, geometry optimization, excitation energies, etc.
- **environment** – Environment description missing
- **compute_settings** – Settings for the calculation, e.g., number of MPI processes and OpenMP threads, work and scratch directory, etc.
- **filename** – Optional user-specified filename that will be used for input and output files. If not specified a name will be generated as a hash of the input.
- **force_recompute** – Recompute even if the output files already exist.

Returns

OutputParser instance that contains the filename of the output produced in the calculation and can be used to extract results from the output.

daltonproject.gaussian.**compute_farm**(*molecules: Sequence[*Molecule*]*, *basis:* Basis, *qc_method:* QCMethod, *properties:* Property, *compute_settings:* ComputeSettings | *None* = *None*, *filenames: Sequence[str]* | *None* = *None*, *force_recompute: bool* = *False*) → list[*OutputParser*]

Run a series of calculations using the Gaussian program.

Parameters

- **molecules** – List of molecules on which calculations are performed.
- **basis** – Basis set to use in the calculations.
- **qc_method** – Quantum chemistry method, e.g., HF, DFT, or CC, and associated settings used for all calculations.
- **properties** – Properties of molecule to be calculated, geometry optimization, excitation energies, etc., computed for all molecules.
- **compute_settings** – Settings for the calculations, e.g., number of MPI processes and OpenMP threads, work and scratch directory, and more.
- **filenames** – Optional list of user-specified filenames that will be used for input and output files. If not specified names will be generated as a hash of the individual inputs.
- **force_recompute** – Recompute even if the output files already exist.

Returns

List of OutputParser instances each of which contains the filename of the output produced in the calculation and can be used to extract results from the output.

CHAPTER

FIFTEEN

LSDALTON

Initialize LSDalton interface module.

class daltonproject.lsdalton.**OutputParser**(*filename: str*)

Parse LSDalton output files.

property dipole_gradients: ndarray

Extract Cartesian dipole gradients from OpenRSP tensor file.

property electronic_energy: float

Extract the electronic energy from LSDalton output file.

property energy: float

Extract the final energy from LSDalton output file.

property excitation_energies: ndarray

Extract one-photon excitation energies from LSDalton output file.

property filename: str

Name of the LSDalton output files without the extension.

property final_geometry: ndarray

Extract final geometry from LSDalton output file.

property gradients: ndarray

Extract molecular gradient from LSDalton output file.

property hessian: ndarray

Extract molecular hessian from OpenRSP tensor file.

property nuclear_repulsion_energy: float

Extract the nuclear repulsion energy from LSDalton output file.

property oscillator_strengths: ndarray

Extract one-photon oscillator strengths from LSDalton output file.

property polarizability_gradients: *PolarizabilityGradients*

Extract Cartesian polarizability gradients from OpenRSP tensor file.

daltonproject.lsdalton.**compute**(*molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *properties:* Property, *environment: Environment | None = None*, *compute_settings:* ComputeSettings | *None = None*, *filename: str | None = None*, *force_recompute: bool = False*) → *OutputParser*

Run a calculation using the LSDalton program.

Parameters

- **molecule** – Molecule on which a calculations is performed. This can also be an atom, a fragment, or any collection of atoms.
- **basis** – Basis set to use in the calculation.
- **qc_method** – Quantum chemistry method, e.g., HF, DFT, or CC, and associated settings.
- **properties** – Properties of molecule to be calculated, geometry optimization, excitation energies, etc.
- **environment** – Environment description missing.
- **compute_settings** – Settings for the calculation, e.g., number of MPI processes and OpenMP threads, work and scratch directory, etc.
- **filename** – Optional user-specified filename that will be used for input and output files. If not specified a name will be generated as a hash of the input.
- **force_recompute** – Recompute even if the output files already exist.

Returns

OutputParser instance that contains the filename of the output produced in the calculation and can be used to extract results from the output.

daltonproject.lsdalton.**compute_farm**(*molecules: Sequence[*Molecule*]*, *basis:* Basis, *qc_method:* QCMethod, *properties:* Property, *compute_settings:* ComputeSettings | *None* = *None*, *filenames: Sequence[str]* | *None* = *None*, *force_recompute: bool* = *False*) → list[*OutputParser*]

Run a series of calculations using the LSDalton program.

Parameters

- **molecules** – List of molecules on which calculations are performed.
- **basis** – Basis set to use in the calculations.
- **qc_method** – Quantum chemistry method, e.g., HF, DFT, or CC, and associated settings used for all calculations.
- **properties** – Properties of molecule to be calculated, geometry optimization, excitation energies, etc., computed for all molecules.
- **compute_settings** – Settings for the calculations, e.g., number of MPI processes and OpenMP threads, work and scratch directory, and more.
- **filenames** – Optional list of user-specified filenames that will be used for input and output files. If not specified names will be generated as a hash of the individual inputs.
- **force_recompute** – Recompute even if the output files already exist.

Returns

List of OutputParser instances each of which contains the filename of the output produced in the calculation and can be used to extract results from the output.

daltonproject.lsdalton.**coulomb_matrix**(*density_matrix: ndarray*, *molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *geometric_derivative_order: int* = *0*, *magnetic_derivative_order: int* = *0*) → ndarray

Calculate the Coulomb matrix.

Parameters

- **density_matrix** – Density matrix
- **molecule** – Molecule object.

- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns
Coulomb matrix

daltonproject.lsdalton.**diagonal_density**(*hamiltonian: ndarray*, *metric: ndarray*, *molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod) → ndarray

Form an AO density matrix from occupied MOs through diagonalization.

Parameters

- **hamiltonian** – Hamiltonian matrix.
- **metric** – Metric matrix (i.e. overlap matrix)
- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.

Returns
Density matrix

daltonproject.lsdalton.**electronic_electrostatic_potential**(*density_matrix: ndarray*, *points: ndarray*, *molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *ep_derivative_order: tuple[int, int] = (0, 0)*, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate the electronic electrostatic potential at a set of points.

Derivatives of the electrostatic potential can be calculated using the ep_derivative_order argument, e.g., ep_derivative_order = (0, 1) will calculate both the zeroth- and first-order derivatives while ep_derivative_order = (1, 1) will calculate first-order derivatives only.

Parameters

- **density_matrix** – Density matrix.
- **points** – Set of points where the electrostatic potential will be evaluated.
- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **ep_derivative_order** – Range of orders of the derivative of the electrostatic potential.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns
Electronic electrostatic potential

daltonproject.lsdalton.**electrostatic_potential**(*density_matrix: ndarray*, *points: ndarray*, *molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *ep_derivative_order: tuple[int, int] = (0, 0)*, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate the molecular electrostatic potential at a set of points.

Derivatives of the electrostatic potential can be calculated using the ep_derivative_order argument, e.g., ep_derivative_order = (0, 1) will calculate both the zeroth- and first-order derivatives while ep_derivative_order = (1, 1) will calculate first-order derivatives only.

Parameters

- **density_matrix** – Density matrix.
- **points** – Set of points where the electrostatic potential will be evaluated.
- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **ep_derivative_order** – Range of orders of the derivative of the electrostatic potential.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns

Point-wise electronic multipol-moment potential

daltonproject.lsdalton.**electrostatic_potential_integrals**(*points: ndarray*, *molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *ep_derivative_order: tuple[int, int] = (0, 0)*, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate electrostatic-potential integrals.

Derivatives of the electrostatic potential can be calculated using the ep_derivative_order argument, e.g., ep_derivative_order = (0, 1) will calculate both the zeroth- and first-order derivatives while ep_derivative_order = (1, 1) will calculate first-order derivatives only.

Let a and b be AO basis functions, C a point, and ξ and ζ Cartesian components, then order 0: $(ab|C) = \int a(r)b(r)V(r,C)dr$, with the potential: $V(r,C) = 1/|r-C|$ order 1: $\int a(r)b(r)V_{\xi}(r,C)dr$, with first-order potential derivative: $V_{\xi}(r,C) = \partial V(r,C)\partial C_{\xi} = C_{\xi}/|r-C|^3$ order 2: $\int a(r)b(r)V_{\xi,\zeta}(r,C)dr$, with second-order potential derivative: $V_{\xi,\zeta}(r,C) = \partial^2 V(r,C)\partial C_{\xi}\partial C_{\zeta}$

Parameters

- **points** – Set of points where the electrostatic-potential integrals will be evaluated.
- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **ep_derivative_order** – Range of orders of the derivative of the electrostatic potential.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.

- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns
Electrostatic-potential integrals

daltonproject.lsdalton.**eri**(*specs: str*, *molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate electron-repulsion integrals for different operator choices.

$$g_{ijkl} = \int \mathrm{d}\mathbf{r} \int \mathrm{d}\mathbf{r}' \chi_i(\mathbf{r}) \chi_j(\mathbf{r}') g(\mathbf{r}, \mathbf{r}') \chi_k(\mathbf{r}) \chi_l(\mathbf{r}')$$

The first character of the *specs* string specifies the operator $g(\mathbf{r}, \mathbf{r}')$. Valid values are:

- C for Coulomb ($g(\mathbf{r}, \mathbf{r}') = \frac{1}{|\mathbf{r}-\mathbf{r}'|}$)
- G for Gaussian geminal (g)
- F geminal divided by the Coulomb operator ($\frac{g}{|\mathbf{r}-\mathbf{r}'|}$)
- D double commutator ($[[T, g], g]$)
- 2 Gaussian geminal operator squared (g^2)

The last four characters of the *specs* string specify the AO basis to use for the four basis functions χ_i, χ_j, χ_k, and χ_l. Valid values are:

- R for regular AO basis
- D for auxiliary (RI) AO basis
- E for empty AO basis (i.e. for 2- and 3-center ERIs)
- N for nuclei

Parameters

- **specs** – A 5-character-long string with the specification for AO basis and operator to use (see description above).
- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns
Electron-repulsion integral tensor.

daltonproject.lsdalton.**eri4**(*molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate four-center electron-repulsion integrals.

$$(ab|cd) = g_{acbd}$$

with

$$g(\mathbf{r}, \mathbf{r}') = \frac{1}{|\mathbf{r} - \mathbf{r}'|}$$

Parameters

- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns

Four-center electron-repulsion integral tensor.

daltonproject.lsdalton.**exchange_correlation**(*density_matrix: ndarray*, *molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → tuple[float, numpy.ndarray]

Calculate the exchange-correlation energy and matrix.

Parameters

- **density_matrix** – Density matrix.
- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns

Exchange-correlation energy and matrix

daltonproject.lsdalton.**exchange_matrix**(*density_matrix: ndarray*, *molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate the exchange matrix.

Parameters

- **density_matrix** – Density matrix.
- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns

Exchange matrix

daltonproject.lsdalton.**fock_matrix**(*density_matrix: ndarray*, *molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate the Fock/KS matrix.

Parameters

- **density_matrix** – Density matrix
- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns

Fock/KS matrix

daltonproject.lsdalton.**kinetic_energy_matrix**(*molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate the kinetic energy matrix.

Parameters

- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns

Kinetic energy matrix

daltonproject.lsdalton.**multipole_interaction_matrix**(*moments: ndarray*, *points: ndarray*, *molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod, *multipole_orders: tuple[int, int] = (0, 0)*, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate the electron-multipole electrostatic interaction matrix in atomic-orbital (AO) basis.

Minimum and maximum orders of the multipole moments are provided by the multipole_orders argument, e.g., multipole_orders = (0, 1) includes charges (order 0) and dipoles (order 1) while multipole_orders = (1, 1) includes only dipoles.

Parameters

- **moments** – Multipole moments.
- **points** – Positions of the multipole moments.
- **molecule** – Molecule object.
- **basis** – Basis object.

- **qc_method** – QCMethod object.
- **multipole_orders** – Minimum and maximum orders of the multipole moments, where 0 = charge, 1 = dipole, etc.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns
Electron-multipole electrostatic interaction matrix

daltonproject.lsdalton.**nuclear_electron_attraction_matrix**(*molecule: Molecule*, *basis: Basis*, *qc_method: QCMethod*, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate the nuclear-electron attraction matrix.

Parameters

- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns
Nuclear-electron attraction matrix

daltonproject.lsdalton.**nuclear_electrostatic_potential**(*points: ndarray*, *molecule: Molecule*, *ep_derivative_order: tuple[int, int] = (0, 0)*, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate the nuclear electrostatic potential at a set of points.

Derivatives of the electrostatic potential can be calculated using the ep_derivative_order argument, e.g., ep_derivative_order = (0, 1) will calculate both the zeroth- and first-order derivatives while ep_derivative_order = (1, 1) will calculate first-order derivatives only.

Parameters

- **points** – Set of points where the electrostatic potential will be evaluated.
- **molecule** – Molecule object.
- **ep_derivative_order** – Range of orders of the derivative of the electrostatic potential.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns
Nuclear electrostatic potential

daltonproject.lsdalton.**nuclear_energy**(*molecule:* Molecule) → float

Calculate the nuclear-repulsion energy.

Parameters
molecule – Molecule object.

Returns
Nuclear-repulsion energy

daltonproject.lsdalton.**num_atoms**(*molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod) → int

Return the number of atoms.

Parameters

- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.

Returns
Number of atoms

daltonproject.lsdalton.**num_basis_functions**(*molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod) → int

Return the number of basis functions.

Parameters

- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.

Returns
Number of basis functions

daltonproject.lsdalton.**num_electrons**(*molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod) → int

Return the number of electrons.

Parameters

- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.

Returns
Number of electrons

daltonproject.lsdalton.**num_ri_basis_functions**(*molecule:* Molecule, *basis:* Basis, *qc_method:* QCMethod) → int

Return the number of auxiliary (RI) basis functions.

Parameters

- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.

Returns
Number of auxiliary (RI) basis functions

daltonproject.lsdalton.**overlap_matrix**(*molecule: Molecule*, *basis: Basis*, *qc_method: QCMethod*, *geometric_derivative_order: int = 0*, *magnetic_derivative_order: int = 0*) → ndarray

Calculate the overlap matrix.

Parameters

- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.
- **geometric_derivative_order** – Order of the derivative with respect to nuclear displacements.
- **magnetic_derivative_order** – Order of the derivative with respect to magnetic field.

Returns

Overlap matrix

daltonproject.lsdalton.**ri2**(*molecule: Molecule*, *basis: Basis*, *qc_method: QCMethod*) → ndarray

Calculate two-center electron-repulsion integrals.

$$(I|J) = g_{I00J}$$

with

$$g(\mathbf{r}, \mathbf{r}') = \frac{1}{|\mathbf{r} - \mathbf{r}'|}$$

Parameters

- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.

Returns

Two-center RI integrals

daltonproject.lsdalton.**ri3**(*molecule: Molecule*, *basis: Basis*, *qc_method: QCMethod*) → ndarray

Calculate three-center electron-repulsion integrals.

$$(ab|I) = g_{a0bI}$$

with

$$g(\mathbf{r}, \mathbf{r}') = \frac{1}{|\mathbf{r} - \mathbf{r}'|}$$

Parameters

- **molecule** – Molecule object.
- **basis** – Basis object.
- **qc_method** – QCMethod object.

Returns

Three-center RI integrals

CHAPTER

SIXTEEN

MOL_READER

Dalton molecule file reader module.

daltonproject.mol_reader.**mol_reader**(*filename: str*) → tuple[*daltonproject.molecule.Molecule*, *daltonproject.basis.Basis*]

Read Dalton molecule input file.

Parameters

filename – Name of Dalton molecule input file containing atoms and coordinates.

Returns

Molecule and Basis objects.

CHAPTER

SEVENTEEN

SYMMETRY

Symmetry analysis.

class daltonproject.symmetry.**SymmetryAnalysis**(*elements: List[str]*, *xyz: ndarray*, *symmetry_threshold: float = 0.001*, *labels: Optional[List[str]] = None*)

Class for determining the symmetry of a molecule.

symmetry_threshold

Threshold for determination of symmetry. Default is 1e-3.

Type
float

max_rotations

Maximum order of rotation to be checked for. Default and maximum value is 20.

Type
int

xyz

Cartesian coordinates of molecule.

Type
np.ndarray

symmetry_elements

Symmetry elements of molecule.

Type
Dict[str, np.ndarray]

find_symmetry() → Tuple[ndarray, str]

Find the symmetry of the molecule.

Returns
Coordinates and point group of the molecule

get_reflection_matrix(*normal_vector: ndarray*) → ndarray

Calculate reflection matrix.

Parameters
normal_vector – Normal vector to the reflection plane. Is assumed to go through origin.

Returns
Reflection matrix.

`get_rotation_matrix`(*axis: ndarray*, *theta: float*) → ndarray

Calculate rotation matrix.

Parameters

- **axis** – Axis of rotations. Is assumed to go through origin.
- **theta** – Angle of rotation.

Returns

Rotation matrix.

`rotate_molecule_pseudo_convention`() → ndarray

Rotate the molecule, so that it is suitable for Dalton's generator input.

Returns

Coordinates of rotated molecule.

CHAPTER

EIGHTEEN

NATURAL_OCCUPATION

Natural orbital occupation module.

class daltonproject.natural_occupation.**CAS**(*active_electrons: int*, *active_orbitals: int | list[int]*, *inactive_orbitals: int | list[int]*)

Data structure for complete active space specification.

active_electrons: int

Alias for field number 0

active_orbitals: int | list[int]

Alias for field number 1

inactive_orbitals: int | list[int]

Alias for field number 2

class daltonproject.natural_occupation.**NatOrbOcc**(*num_irreps: int*, *strong_nat_occ: dict[int, np.ndarray]*, *weak_nat_occ: dict[int, np.ndarray]*, *strong_nat_occ_nosym: np.ndarray*, *strong_nat_occ_nosym2sym_idx: np.ndarray*, *weak_nat_occ_nosym: np.ndarray*, *weak_nat_occ_nosym2sym_idx: np.ndarray*)

Data structure for natural orbital occupations.

num_irreps: int

Alias for field number 0

strong_nat_occ: dict[int, numpy.ndarray]

Alias for field number 1

strong_nat_occ_nosym: ndarray

Alias for field number 3

strong_nat_occ_nosym2sym_idx: ndarray

Alias for field number 4

weak_nat_occ: dict[int, numpy.ndarray]

Alias for field number 2

weak_nat_occ_nosym: ndarray

Alias for field number 5

weak_nat_occ_nosym2sym_idx: ndarray

Alias for field number 6

daltonproject.natural_occupation.**pick_cas_by_thresholds**(*natural_occupations: Mapping[int, ndarray]*, *max_strong_occupation: float*, *min_weak_occupation: float*) → *CAS*

Get parameters needed for a CAS calculation based on simple thresholds.

Parameters

- **natural_occupations** – Natural occupation numbers.
- **max_strong_occupation** – Maximum occupation number of strongly occupied natural orbitals to be included in CAS.
- **min_weak_occupation** – Minimum occupation number of weakly occupied natural orbitals to be included in CAS.

Returns

Number of active electrons, CAS, and number of inactive orbitals.

daltonproject.natural_occupation.**scan_occupations**(*natural_occupations: Mapping[int, ndarray]*, *strong_threshold: float = 1.995*, *weak_threshold: float = 0.002*, *max_orbitals: int = 14*) → str

Determine thresholds for picking active space.

Will not print orbitals outside the bounds of "strong_threshold" and "weak_threshold".

Parameters

- **natural_occupations** – Natural occupation numbers.
- **strong_threshold** – Strongly occupied natural orbitals with occupations above this threshold will not be printed.
- **weak_threshold** – Weakly occupied natural orbitals with occupations below this threshold will not be printed.
- **max_orbitals** – Maximum number of strongly and weakly occupied orbitals to print.

Returns

String to be printed or saved to file.

daltonproject.natural_occupation.**sort_natural_occupations**(*natural_occupations: np.ndarray | Mapping[int, np.ndarray]*) → *NatOrbOcc*

Sort natural orbital occupation numbers.

Parameters

natural_occupations – Natural occupation numbers.

Returns

Sorted natural orbital occupation numbers.

daltonproject.natural_occupation.**trim_natural_occupations**(*natural_occupations: Mapping[int, ndarray]*, *strong_threshold: float = 1.995*, *weak_threshold: float = 0.002*) → str

Print relevant natural occupations in a nice format.

Parameters

- **natural_occupations** – Natural occupation numbers.
- **strong_threshold** – Strongly occupied natural orbitals with occupations above this threshold will not be printed.

- **weak_threshold** – Weakly occupied natural orbitals with occupations below this threshold will not be printed.

Returns

String to be printed or saved to file.

CHAPTER

NINETEEN

VIBRATIONAL_ANALYSIS

Vibrational analysis.

class daltonproject.vibrational_analysis.**VibrationalAnalysis**(*frequencies: np.ndarray*, *ir_intensities: np.ndarray | None = None*, *raman_intensities: np.ndarray | None = None*, *cartesian_displacements: np.ndarray | None = None*)

Data structure for vibrational properties.

cartesian_displacements: np.ndarray | None

Alias for field number 3

frequencies: np.ndarray

Alias for field number 0

ir_intensities: np.ndarray | None

Alias for field number 1

raman_intensities: np.ndarray | None

Alias for field number 2

daltonproject.vibrational_analysis.**compute_ir_intensities**(*dipole_gradients: ndarray*, *transformation_matrix: ndarray*, *is_linear: bool = False*) → ndarray

Compute infrared molar decadic absorption coefficient in m^2 / (s * mol).

Parameters

- **dipole_gradients** – Gradients of the dipole moment (au) with respect to nuclear displacements in Cartesian coordinates.
- **transformation_matrix** – Transformation matrix to convert from Cartesian to normal coordinates.
- **is_linear** – Indicate if the molecule is linear.

Returns

IR intensities in m^2 / (s * mol).

daltonproject.vibrational_analysis.**compute_raman_intensities**(*polarizability_gradients: ndarray*, *polarizability_frequency: float*, *vibrational_frequencies: ndarray*, *transformation_matrix: ndarray*, *is_linear: bool = False*) → ndarray

Compute Raman differential scattering cross-section in C^4 * s^2 / (J * m^2 * kg).

Parameters

- **polarizability_gradients** – Gradients of the polarizability with respect to nuclear displacements in Cartesian coordinates.
- **polarizability_frequency** – Frequency of the incident light in au.
- **vibrational_frequencies** – Vibrational frequencies in cm^-1.
- **transformation_matrix** – Transformation matrix to convert from Cartesian to normal coordinates.
- **is_linear** – Indicate if the molecule is linear.

Returns

Raman intensities in C^4 * s^2 / (J * m^2 * kg).

daltonproject.vibrational_analysis.**normal_coords**(*molecule: Molecule*, *hessian: np.ndarray*, *is_linear: bool = False*, *isotopes: Sequence[float] | None = None*) → np.ndarray

Compute normal coordinates.

Parameters

- **molecule** – Molecule object.
- **hessian** – Second derivatives of energy with respect to nuclear displacements.
- **is_linear** – Indicate if the molecule is linear.
- **isotopes** – List of isotopes for each atom in the molecule.

Returns

q_mat = normal mode coordinates.

daltonproject.vibrational_analysis.**normal_mode_eigensolver**(*molecule: Molecule*, *hessian: np.ndarray*, *is_linear: bool = False*, *isotopes: Sequence[float] | None = None*) → NormalModeEigenSolution

Calculate eigenvalues and eigenvectors of the Hessian matrix.

Parameters

- **molecule** – Molecule object.
- **hessian** – Second derivatives of energy with respect to nuclear displacements in Cartesian coordinates.
- **is_linear** – Indicate if the molecule is linear.
- **isotopes** – List of isotopes for each atom in the molecule.

Returns

Eigenvalues of the mass-weighted Hessian matrix. transformation_matrix: Eigenvectors of the mass-weighted Hessian matrix. frequencies: Vibrational frequencies in cm^-1.

Return type

eigen_vals

daltonproject.vibrational_analysis.**number_modes**(*molecule:* Molecule, *is_linear: bool*) → int

Return number of normal modes.

Parameters

- **molecule** – Molecule object.
- **is_linear** – Indicate if the molecule is linear.

Returns
Number of normal modes.

Return type
num_modes

daltonproject.vibrational_analysis.**vibrational_analysis**(*molecule:* Molecule, *hessian: np.ndarray*, *dipole_gradients: np.ndarray | None = None*, *polarizability_gradients:* PolarizabilityGradients *| None = None*) → *VibrationalAnalysis*

Perform vibrational analysis.

Parameters

- **molecule** – Molecule object.
- **hessian** – Second derivatives with respect to nuclear displacements in Cartesian coordinates.
- **dipole_gradients** – Gradient of the dipole moment with respect to nuclear displacements in Cartesian coordinates.
- **polarizability_gradients** – Gradient of the polarizability with respect to nuclear displacements in Cartesian coordinates.

Returns
Harmonic vibrational frequencies (cm^-1). Optionally, IR intensities (m^2 / (s * mol)), Raman intensities (C^4 * s^2 / (J * m^2 * kg)), and transformation matrix.

CHAPTER

TWENTY

VIBRATIONAL_AVERAGING

Main driver for calculation of vibrational corrections to properties.

class daltonproject.vibrational_averaging.**ComputeVibAvCorrection**(*hess_objects: Sequence[OutputParser]*, *prop_objects: Sequence[OutputParser]*, *vibav_settings: VibAvSettings*)

ComputeVibAvCorrection class for computing a vibrational correction to a property.

call_vibav() → None

Compute vibrational correction to properties.

coefficients_of_inertia_expansion(*mode_vector*) → ndarray

Calculate coefficients of inertia expansion.

Parameters
mode_vector – Normal coordinate vector.

Returns
Coefficients of inertia expansion in ang*amu^(1/2).

Return type
coeff_expansion_tensor

compute_vib_correction(*property_mat: np.ndarray*, *property_labels: Sequence[str | Sequence[str]]*) → *ComputeVibCorrection*

Vibrational correction to properties.

Parameters

- **property_mat** – property matrix.
- **property_labels** – Property labels.

Returns
ComputeVibCorrection class object.

The vibrational correction to a property P is then given by $\Delta^{\text{VPT2}}P = -\frac{1}{4}\sum_i \frac{1}{\omega_i}\frac{\partial P}{\partial q_i}\sum_j k_{ijj} + \frac{1}{4}\sum_i \frac{\partial^2 P}{\partial q_i^2}$. Temperature dependent corrections to properties are calculated using $\Delta^{\text{VPT2}}P = -\frac{1}{4}\sum_i \frac{1}{\omega_i}\frac{\partial P}{\partial q_i}\left(\sum_j k_{ijj}\coth\left(\frac{hc\omega_j}{2kT}\right)\right) + \frac{1}{4}\sum_i \frac{\partial^2 P}{\partial q_i^2}\coth\left(\frac{hc\omega_i}{2kT}\right)$

conversion_factor() → ndarray

Calculate conversion factor (bohr^-2*amu^-1) to convert cubic force constants to cm^-1.

Returns
Conversion factor.

cubic_fc(*q_mat_displaced: ndarray*, *factor: ndarray*) → ndarray

Calculate cubic force constants numerically.

Parameters

- **q_mat_displaced** – Displaced normal coordinate matrix.
- **factor** – Conversion factor in units of bohr^-2*amu^-1.

Returns
Cubic force constants in units of cm^-1.

Cubic force constants are calculated numerically using the following equation: $\Phi_{ijk} \simeq \frac{1}{3}\left(\frac{\Phi_{jk}(\partial Q_i)-\Phi_{jk}(\partial Q_i)}{2\partial Q_i} + \frac{\Phi_{ki}(\partial Q_j)-\Phi_{ki}(\partial Q_j)}{2\partial Q_j} + \frac{\Phi_{ij}(\partial Q_k)-\Phi_{ij}(\partial Q_k)}{2\partial Q_k}\right)$.

Ref: Barone, V. J. Phys. Chem. 2005, 122, 014108

displace_q_geometries() → ndarray

Generate displaced normal mode geometries.

Returns
Displaced normal coordinate matrix.

mean_displacement() → tuple[numpy.ndarray, numpy.ndarray]

Calculate mean displacements.

Returns
Mean displacement (dimensionless). q2: Mean displacement squared (dimensionless).

Return type
q

Mean displacement is defined as: $\left\langle \phi^{(1)} \left| q_i \right| \phi^{(0)} \right\rangle = \frac{1}{4}\sum_i \frac{1}{\omega_i}\sum_j k_{ijj}$ Mean displacement squared is defined as: $\left\langle \phi^{(0)} \left| q_i q_j \right| \phi^{(0)} \right\rangle = \frac{1}{2}$

process_property_parsers() → None

Process property values to ensure correct dimensionality.

rotational_contribution() → ndarray

Calculate rotational contribution to a temperature dependent vibrational correction.

Returns
Rotational contribution to vibrational correction in ang^(-1)*amu^(-1/2).

Return type
quotient

write_output_file(*total_corrections: ndarray*) → None

Write output file for vibrational correction results.

Parameters
total_corrections – Property values.

class daltonproject.vibrational_averaging.**ComputeVibCorrection**(*property_first_derivatives: np.ndarray*, *property_second_derivatives: np.ndarray*, *mean_displacement_q: np.ndarray*, *mean_displacement_q2: np.ndarray*, *cubic_force_constants: np.ndarray*, *vibrational_corrections: np.ndarray*)

Data structure for vibrational correction to properties.

cubic_force_constants: ndarray

Alias for field number 4

mean_displacement_q: ndarray

Alias for field number 2

mean_displacement_q2: ndarray

Alias for field number 3

property_first_derivatives: ndarray

Alias for field number 0

property_second_derivatives: ndarray

Alias for field number 1

vibrational_corrections: ndarray

Alias for field number 5

class daltonproject.vibrational_averaging.**VibAvConstants**

Modified constants used in vibrational averaging.

property c_cm: float

Return speed of light in units of cm/s.

property hbar_a: float

Return reduced Planck constant in units of angstrom^2 * kg/s.

property m2au: float

Convert from meter to bohr.

class daltonproject.vibrational_averaging.**VibAvSettings**(*molecule:* Molecule, *property_program: str*, *is_mol_linear: bool*, *hessian: np.ndarray*, *property_obj:* Property, *stepsize: float | int*, *differentiation_method: str*, *temperature: float | int*, *polynomial_fitting_order: int | None = None*, *linear_point_stencil: int | None = None*, *plot_polyfittings: bool | None = False*)

VibAvSettings class for initialising user input and generating distorted geometry .xyz or .gau files for property and Hessian calculations.

check_property_assignment()

Check to see if property and program combination is supported.

displace_coordinates(*factor: int*, *eq_freq: float*, *mode_num: int*, *reordered_q_mat: ndarray*) → ndarray

Displace equilibrium coordinates.

Parameters

- **factor** – Factor to multiply normal coordinate by.
- **eq_freq** – An equilibrium frequency.
- **mode_num** – Mode number.
- **reordered_q_mat** – Reordered normal coordinates.

Returns
Displaced coordinates.

Return type
displaced_coords

generate_displaced_geometries() → list[str]

Write loop for displaced geometries.

Returns
Sequence of filenames for displaced geometries.

Return type
file_list

static step_modifier(*stepsize: float*, *eq_freq: float*) → float

Modify stepsize from units of reduced normal coordinates to units of normal coordinates.

Normal coordinates Q are converted to reduced normal coordinates q: Q = $(\hbar/2\pi c\omega)^{0.5}q$.

Parameters

- **stepsize** – Step size in reduced normal coordinates.
- **eq_freq** – An equilibrium frequency.

Returns
Modified stepsize in normal coordinates.

write_gau_input(*coords: ndarray*, *filename: str*) → None

Write .gau file for a given displaced geometry.

Parameters

- **coords** – Coordinates to write.
- **filename** – Name of the file to write.

write_xyz_input(*coords: ndarray*, *filename: str*) → None

Write .xyz file for a given displaced geometry.

Parameters

- **coords** – Coordinates to write.
- **filename** – Name of the file to write.

CHAPTER

TWENTYONE

SPECTRUM

Spectrum module.

daltonproject.spectrum.**convolute_one_photon_absorption**(*excitation_energies: ~numpy.ndarray, oscillator_strengths: ~numpy.ndarray, energy_range: ~numpy.ndarray, broadening_function: ~collections.abc.Callable[[~numpy.ndarray, float, float], ~numpy.ndarray] = <function gaussian>, broadening_factor: float = 0.4*) → ndarray

Convolute one-photon absorption.

The convolution is based on the equations of: A. Rizzo, S. Coriani, and K. Ruud, in Computational Strategies for Spectroscopy. From Small Molecules to Nano Systems, edited by V. Barone (John Wiley and Sons, 2012) Chap. 2, pp. 77–135

Parameters

- **excitation_energies** – Excitation energies in eV.
- **oscillator_strengths** – Oscillator strengths.
- **energy_range** – Energy range of the spectrum in eV.
- **broadening_function** – Broadening function.
- **broadening_factor** – Broadening factor in eV.

Returns

Convoluted spectrum of molar absorptivity in L / (mol * cm).

daltonproject.spectrum.**convolute_two_photon_absorption**(*excitation_energies: ~numpy.ndarray, two_photon_strengths: ~numpy.ndarray, energy_range: ~numpy.ndarray, broadening_function: ~collections.abc.Callable[[~numpy.ndarray, float, float], ~numpy.ndarray] = <function lorentzian>, broadening_factor: float = 0.1*) → ndarray

Convolute two-photon absorption.

Parameters

- **excitation_energies** – Excitation energies in eV.
- **two_photon_strengths** – Two-photon absorption strengths in au.

- **energy_range** – Energy range of the spectrum in eV.
- **broadening_function** – Broadening function.
- **broadening_factor** – Broadening factor in eV.

Returns
Convoluted spectrum of two-photon absorption cross-section in GM.

daltonproject.spectrum.**convolute_vibrational_spectrum**(*vibrational_frequencies: ~numpy.ndarray, intensities: ~numpy.ndarray, energy_range: ~numpy.ndarray, broadening_function: ~collections.abc.Callable[[~numpy.ndarray, float, float], ~numpy.ndarray] = <function lorentzian>, broadening_factor: float = 3.0*) → ndarray

Convolute vibrational spectrum (e.g. IR and Raman).

Parameters

- **vibrational_frequencies** – Vibrational frequencies in cm^-1.
- **intensities** – Intensities in SI unit.
- **energy_range** – Energy range of the spectrum in cm^-1.
- **broadening_function** – Broadening function.
- **broadening_factor** – Broadening factor in cm^-1.

Returns
Convoluted spectrum in unit of the intensities * s.

daltonproject.spectrum.**gaussian**(*x: ndarray*, *x0: float*, *broadening_factor: float*) → ndarray

Calculate normalized Gaussian broadening function.

Parameters

- **x** – Where to evaluate the Gaussian.
- **x0** – Center of Gaussian.
- **broadening_factor** – Half width at half maximum.

Returns
A Gaussian.

daltonproject.spectrum.**lorentzian**(*x: ndarray*, *x0: float*, *broadening_factor: float*) → ndarray

Calculate normalized Lorentzian broadening function.

Parameters

- **x** – Where to evaluate the Lorentzian.
- **x0** – Center of Lorentzian.
- **broadening_factor** – Half width at half maximum.

Returns
A Lorentzian.

daltonproject.spectrum.**plot_ir_spectrum**(*vibrational_analysis: VibrationalAnalysis, ax: Axes | None = None, energy_range: Sequence | None = None, broadening_function: Callable[[np.ndarray, float, float], np.ndarray] = <function lorentzian>, broadening_factor: float = 3.0, **plot_kwargs*) → Axes

Plot IR spectrum.

Parameters

- **vibrational_analysis** – VibrationalAnalysis object that contains frequencies in cm^-1 and IR intensities in m^2 / (s * mol).
- **ax** – Axes object (matplotlib). If specified, plot will be added to the provided Axes object, otherwise a new one is created.
- **energy_range** – Energy range of the plot in cm^-1. It will be generated from the vibrational frequencies if it is not specified.
- **broadening_function** – Broadening function.
- **broadening_factor** – Broadening factor in cm^-1.
- **plot_kwargs** – Optional keyword arguments to be passed directly to the matplotlib plot function (for example color='k').

Returns

Axes object with a plot of energy (cm^-1) vs molar absorptivity (m^2 / mol).

daltonproject.spectrum.**plot_one_photon_spectrum**(*output: OutputParser, ax: Axes | None = None, energy_range: Sequence | None = None, broadening_function: Callable[[np.ndarray, float, float], np.ndarray] = <function gaussian>, broadening_factor: float = 0.4, **plot_kwargs*) → Axes

Plot one-photon absorption spectrum.

Parameters

- **output** – OutputParser object that contains excitation energies in eV and oscillator strengths.
- **ax** – Axes object (matplotlib). If specified, plot will be added to the provided Axes object, otherwise a new one is created.
- **energy_range** – Energy range of the plot in eV. It will be generated from the excitation energies if it is not specified.
- **broadening_function** – Broadening function.
- **broadening_factor** – Broadening factor in eV.
- **plot_kwargs** – Optional keyword arguments to be passed directly to the matplotlib plot function (for example color='k').

Returns

Axes object with a plot of energy (eV) vs molar absorptivity (L / (mol * cm)).

daltonproject.spectrum.**plot_raman_spectrum**(*vibrational_analysis: VibrationalAnalysis, ax: Axes | None = None, energy_range: Sequence | None = None, broadening_function: Callable[[np.ndarray, float, float], np.ndarray] = <function lorentzian>, broadening_factor: float = 3.0, **plot_kwargs*) → Axes

Plot Raman spectrum.

Parameters

- **vibrational_analysis** – VibrationalAnalysis object that contains frequencies in cm^-1 and Raman intensities in C^4*s^2/(J*m^2*kg).

- **ax** – Axes object (matplotlib). If specified, plot will be added to the provided Axes object, otherwise a new one is created.
- **energy_range** – Energy range of the plot in cm^-1. It will be generated from the vibrational frequencies if it is not specified.
- **broadening_function** – Broadening function.
- **broadening_factor** – Broadening factor in cm^-1.
- **plot_kwargs** – Optional keyword arguments to be passed directly to the matplotlib plot function (for example color='k').

Returns

Axes object with a plot of energy (cm^-1) vs Raman scattering cross-section (C^4*s^3/(J*m^2*kg)).

daltonproject.spectrum.**plot_spectrum**(*x: ndarray*, *y: ndarray*, *xlabel: str*, *ylabel: str*, *ax: Axes*, ***plot_kwargs*) → Axes

Plot spectrum.

daltonproject.spectrum.**plot_two_photon_spectrum**(*output: OutputParser*, *ax: Axes | None = None*, *energy_range: Sequence | None = None*, *broadening_function: Callable[[np.ndarray, float, float], np.ndarray] = <function gaussian>*, *broadening_factor: float = 0.4*, ***plot_kwargs*) → Axes

Plot two-photon absorption spectrum.

Parameters

- **output** – OutputParser object that contains excitation energies in eV and two-photon absorption strengths in au.
- **ax** – Axes object (matplotlib). If specified, plot will be added to the provided Axes object, otherwise a new one is created.
- **energy_range** – Energy range of the plot in eV. It will be generated from the excitation energies if it is not specified.
- **broadening_function** – Broadening function.
- **broadening_factor** – Broadening factor in eV.
- **plot_kwargs** – Optional keyword arguments to be passed directly to the matplotlib plot function (for example color='k').

Returns

Axes object with a plot of energy (eV) vs two-photon absorption cross-section (GM).

PYTHON MODULE INDEX

d

daltonproject.basis, 31
daltonproject.dalton, 43
daltonproject.gaussian, 47
daltonproject.lsdalton, 49
daltonproject.mol_reader, 59
daltonproject.molecule, 29
daltonproject.natural_occupation, 63
daltonproject.program, 37
daltonproject.property, 35
daltonproject.qcmethod, 33
daltonproject.spectrum, 75
daltonproject.symmetry, 61
daltonproject.vibrational_analysis, 67
daltonproject.vibrational_averaging, 71

INDEX

A

active_electrons (*daltonproject.natural_occupation.CAS attribute*), 63
active_orbitals (*daltonproject.natural_occupation.CAS attribute*), 63
analyze_symmetry() (*daltonproject.molecule.Molecule method*), 29
atoms() (*daltonproject.molecule.Molecule method*), 29

B

Basis (*class in daltonproject.basis*), 31

C

c_cm (*daltonproject.vibrational_averaging.VibAvConstants property*), 73
call_vibav() (*daltonproject.vibrational_averaging.ComputeVibAvCorrection method*), 71
cartesian_displacements (*daltonproject.vibrational_analysis.VibrationalAnalysis attribute*), 67
CAS (*class in daltonproject.natural_occupation*), 63
check_property_assignment() (*daltonproject.vibrational_averaging.VibAvSettings method*), 73
coefficients_of_inertia_expansion() (*daltonproject.vibrational_averaging.ComputeVibAvCorrection method*), 71
comm_port (*daltonproject.program.ComputeSettings property*), 37
complete_active_space() (*daltonproject.qcmethod.QCMethod method*), 33
compute() (*daltonproject.program.Program class method*), 41
compute() (*in module daltonproject.dalton*), 44
compute() (*in module daltonproject.gaussian*), 47
compute() (*in module daltonproject.lsdalton*), 49
compute_farm() (*in module daltonproject.dalton*), 45
compute_farm() (*in module daltonproject.gaussian*), 48
compute_farm() (*in module daltonproject.lsdalton*), 50
compute_ir_intensities() (*in module daltonproject.vibrational_analysis*), 67
compute_raman_intensities() (*in module daltonproject.vibrational_analysis*), 67
compute_vib_correction() (*daltonproject.vibrational_averaging.ComputeVibAvCorrection method*), 71
ComputeSettings (*class in daltonproject.program*), 37
ComputeVibAvCorrection (*class in daltonproject.vibrational_averaging*), 71
ComputeVibCorrection (*class in daltonproject.vibrational_averaging*), 72
conversion_factor() (*daltonproject.vibrational_averaging.ComputeVibAvCorrection method*), 71
convolute_one_photon_absorption() (*in module daltonproject.spectrum*), 75
convolute_two_photon_absorption() (*in module daltonproject.spectrum*), 75
convolute_vibrational_spectrum() (*in module daltonproject.spectrum*), 76
coordinates (*daltonproject.molecule.Molecule property*), 29
coulomb() (*daltonproject.qcmethod.QCMethod method*), 33
coulomb_matrix() (*in module daltonproject.lsdalton*), 50
cubic_fc() (*daltonproject.vibrational_averaging.ComputeVibAvCorrection method*), 71
cubic_force_constants (*daltonproject.vibrational_averaging.ComputeVibCorrection attribute*), 73

D

daltonproject.basis
    module, 31
daltonproject.dalton
    module, 43
daltonproject.gaussian
    module, 47
daltonproject.lsdalton
    module, 49
daltonproject.mol_reader

module, 59
daltonproject.molecule
module, 29
daltonproject.natural_occupation
module, 63
daltonproject.program
module, 37
daltonproject.property
module, 35
daltonproject.qcmethod
module, 33
daltonproject.spectrum
module, 75
daltonproject.symmetry
module, 61
daltonproject.vibrational_analysis
module, 67
daltonproject.vibrational_averaging
module, 71
diagonal_density() (*in module daltonproject.lsdalton*), 51
dipole (*daltonproject.dalton.OutputParser property*), 43
dipole (*daltonproject.program.OutputParser property*), 39
dipole() (*daltonproject.property.Property method*), 35
dipole_gradients (*daltonproject.lsdalton.OutputParser property*), 49
dipole_gradients (*daltonproject.program.OutputParser property*), 39
dipole_gradients() (*daltonproject.property.Property method*), 35
direct (*daltonproject.program.HyperfineCouplings attribute*), 38
displace_coordinates() (*daltonproject.vibrational_averaging.VibAvSettings method*), 74
displace_q_geometries() (*daltonproject.vibrational_averaging.ComputeVibAvCorrection method*), 72

E

electronic_electrostatic_potential() (*in module daltonproject.lsdalton*), 51
electronic_energy (*daltonproject.dalton.OutputParser property*), 43
electronic_energy (*daltonproject.lsdalton.OutputParser property*), 49
electronic_energy (*daltonproject.program.OutputParser property*), 40
electrostatic_potential() (*in module daltonproject.lsdalton*), 51
electrostatic_potential_integrals() (*in module daltonproject.lsdalton*), 52
energy (*daltonproject.dalton.OutputParser property*), 43
energy (*daltonproject.gaussian.OutputParser property*), 47
energy (*daltonproject.lsdalton.OutputParser property*), 49
energy (*daltonproject.program.OutputParser property*), 40
energy() (*daltonproject.property.Property method*), 35
environment() (*daltonproject.qcmethod.QCMethod method*), 33
eri() (*in module daltonproject.lsdalton*), 53
eri4() (*in module daltonproject.lsdalton*), 53
exact_exchange() (*daltonproject.qcmethod.QCMethod method*), 33
exchange() (*daltonproject.qcmethod.QCMethod method*), 33
exchange_correlation() (*in module daltonproject.lsdalton*), 54
exchange_matrix() (*in module daltonproject.lsdalton*), 54
excitation_energies (*daltonproject.dalton.OutputParser property*), 43
excitation_energies (*daltonproject.lsdalton.OutputParser property*), 49
excitation_energies (*daltonproject.program.ExcitationEnergies attribute*), 38
excitation_energies (*daltonproject.program.OutputParser property*), 40
excitation_energies() (*daltonproject.property.Property method*), 35
excitation_energies_per_sym (*daltonproject.program.ExcitationEnergies attribute*), 38
ExcitationEnergies (*class in daltonproject.program*), 38

F

filename (*daltonproject.dalton.OutputParser property*), 43
filename (*daltonproject.gaussian.OutputParser property*), 47
filename (*daltonproject.lsdalton.OutputParser property*), 49
filename (*daltonproject.program.OutputParser property*), 40
final_geometry (*daltonproject.dalton.OutputParser property*), 43
final_geometry (*daltonproject.gaussian.OutputParser property*), 47
final_geometry (*daltonproject.lsdalton.OutputParser property*), 49
final_geometry (*daltonproject.program.OutputParser property*), 40

find_symmetry() (*daltonproject.symmetry.SymmetryAnalysis method*), 61
first_hyperpolarizability (*daltonproject.dalton.OutputParser property*), 43
first_hyperpolarizability (*daltonproject.program.OutputParser property*), 40
first_hyperpolarizability() (*daltonproject.property.Property method*), 35
fock_matrix() (*in module daltonproject.lsdalton*), 54
frequencies (*daltonproject.program.OpticalRotations attribute*), 39
frequencies (*daltonproject.program.Polarizabilities attribute*), 41
frequencies (*daltonproject.program.PolarizabilityGradients attribute*), 41
frequencies (*daltonproject.vibrational_analysis.VibrationalAnalysis attribute*), 67

G

gau() (*daltonproject.molecule.Molecule method*), 29
gaussian() (*in module daltonproject.spectrum*), 76
generate_displaced_geometries() (*daltonproject.vibrational_averaging.VibAvSettings method*), 74
geometry_optimization() (*daltonproject.property.Property method*), 35
get_atom_basis() (*in module daltonproject.basis*), 31
get_reflection_matrix() (*daltonproject.symmetry.SymmetryAnalysis method*), 61
get_rotation_matrix() (*daltonproject.symmetry.SymmetryAnalysis method*), 61
gradients (*daltonproject.dalton.OutputParser property*), 43
gradients (*daltonproject.lsdalton.OutputParser property*), 49
gradients (*daltonproject.program.OutputParser property*), 40
gradients() (*daltonproject.property.Property method*), 36

H

hbar_a (*daltonproject.vibrational_averaging.VibAvConstants property*), 73
hessian (*daltonproject.dalton.OutputParser property*), 43
hessian (*daltonproject.gaussian.OutputParser property*), 47
hessian (*daltonproject.lsdalton.OutputParser property*), 49
hessian (*daltonproject.program.OutputParser property*), 40
hessian() (*daltonproject.property.Property method*), 36
homo_energy (*daltonproject.dalton.OutputParser property*), 43
homo_energy (*daltonproject.program.OutputParser property*), 40
hyperfine_couplings (*daltonproject.dalton.OutputParser property*), 43
hyperfine_couplings (*daltonproject.gaussian.OutputParser property*), 47
hyperfine_couplings (*daltonproject.program.OutputParser property*), 40
hyperfine_couplings() (*daltonproject.property.Property method*), 36
HyperfineCouplings (*class in daltonproject.program*), 38

I

inactive_orbitals (*daltonproject.natural_occupation.CAS attribute*), 63
input_orbital_coefficients() (*daltonproject.qcmethod.QCMethod method*), 33
integral_family() (*daltonproject.qcmethod.QCMethod method*), 34
ir_intensities (*daltonproject.vibrational_analysis.VibrationalAnalysis attribute*), 67
isotopes_assign() (*daltonproject.molecule.Molecule method*), 29

J

jobs_per_node (*daltonproject.program.ComputeSettings property*), 37

K

kinetic_energy_matrix() (*in module daltonproject.lsdalton*), 55

L

launcher (*daltonproject.program.ComputeSettings property*), 37
lorentzian() (*in module daltonproject.spectrum*), 76
lumo_energy (*daltonproject.dalton.OutputParser property*), 43
lumo_energy (*daltonproject.program.OutputParser property*), 40

M

m2au (*daltonproject.vibrational_averaging.VibAvConstants property*), 73

max_rotations (*daltonproject.symmetry.SymmetryAnalysis attribute*), 61
mean_displacement() (*daltonproject.vibrational_averaging.ComputeVibAvCorrection method*), 72
mean_displacement_q (*daltonproject.vibrational_averaging.ComputeVibCorrection attribute*), 73
mean_displacement_q2 (*daltonproject.vibrational_averaging.ComputeVibCorrection attribute*), 73
memory (*daltonproject.program.ComputeSettings property*), 37
mo_energies (*daltonproject.dalton.OutputParser property*), 43
mo_energies (*daltonproject.program.OutputParser property*), 40
module
    daltonproject.basis, 31
    daltonproject.dalton, 43
    daltonproject.gaussian, 47
    daltonproject.lsdalton, 49
    daltonproject.mol_reader, 59
    daltonproject.molecule, 29
    daltonproject.natural_occupation, 63
    daltonproject.program, 37
    daltonproject.property, 35
    daltonproject.qcmethod, 33
    daltonproject.spectrum, 75
    daltonproject.symmetry, 61
    daltonproject.vibrational_analysis, 67
    daltonproject.vibrational_averaging, 71
mol() (*daltonproject.molecule.Molecule method*), 29
mol_reader() (*in module daltonproject.mol_reader*), 59
Molecule (*class in daltonproject.molecule*), 29
mpi_command (*daltonproject.program.ComputeSettings property*), 37
mpi_num_procs (*daltonproject.program.ComputeSettings property*), 37
multipole_interaction_matrix() (*in module daltonproject.lsdalton*), 55

N

NatOrbOcc (*class in daltonproject.natural_occupation*), 63
natural_occupations (*daltonproject.dalton.OutputParser property*), 43
natural_occupations (*daltonproject.program.OutputParser property*), 40
nmr_shieldings (*daltonproject.dalton.OutputParser property*), 44
nmr_shieldings (*daltonproject.gaussian.OutputParser property*), 47
nmr_shieldings (*daltonproject.program.OutputParser property*), 40
nmr_shieldings() (*daltonproject.property.Property method*), 36
node_list (*daltonproject.program.ComputeSettings property*), 37
normal_coords() (*in module daltonproject.vibrational_analysis*), 68
normal_mode_eigensolver() (*in module daltonproject.vibrational_analysis*), 68
nuclear_electron_attraction_matrix() (*in module daltonproject.lsdalton*), 56
nuclear_electrostatic_potential() (*in module daltonproject.lsdalton*), 56
nuclear_energy() (*in module daltonproject.lsdalton*), 56
nuclear_repulsion_energy (*daltonproject.dalton.OutputParser property*), 44
nuclear_repulsion_energy (*daltonproject.lsdalton.OutputParser property*), 49
nuclear_repulsion_energy (*daltonproject.program.OutputParser property*), 40
num_atoms (*daltonproject.molecule.Molecule property*), 29
num_atoms() (*in module daltonproject.lsdalton*), 57
num_basis_functions (*daltonproject.dalton.OutputParser property*), 44
num_basis_functions (*daltonproject.program.OutputParser property*), 40
num_basis_functions() (*in module daltonproject.lsdalton*), 57
num_basis_functions_per_sym (*daltonproject.program.NumBasisFunctions attribute*), 38
num_electrons (*daltonproject.dalton.OutputParser property*), 44
num_electrons (*daltonproject.program.OutputParser property*), 40
num_electrons() (*in module daltonproject.lsdalton*), 57
num_irreps (*daltonproject.natural_occupation.NatOrbOcc attribute*), 63
num_nodes (*daltonproject.program.ComputeSettings property*), 37
num_orbitals (*daltonproject.dalton.OutputParser property*), 44
num_orbitals (*daltonproject.program.OutputParser property*), 40
num_orbitals_per_sym (*daltonproject.program.NumOrbitals attribute*), 38
num_ri_basis_functions() (*in module daltonpro-*

ject.lsdalton), 57
NumBasisFunctions (*class in daltonproject.program*), 38
number_modes() (*in module daltonproject.vibrational_analysis*), 68
NumOrbitals (*class in daltonproject.program*), 38

O

omp_num_threads (*daltonproject.program.ComputeSettings property*), 37
optical_rotations (*daltonproject.dalton.OutputParser property*), 44
optical_rotations (*daltonproject.gaussian.OutputParser property*), 47
optical_rotations (*daltonproject.program.OutputParser property*), 40
optical_rotations() (*daltonproject.property.Property method*), 36
OpticalRotations (*class in daltonproject.program*), 39
orbital_coefficients (*daltonproject.dalton.OutputParser property*), 44
orbital_coefficients (*daltonproject.program.OrbitalCoefficients attribute*), 39
orbital_coefficients (*daltonproject.program.OutputParser property*), 41
orbital_coefficients_per_sym (*daltonproject.program.OrbitalCoefficients attribute*), 39
orbital_energies (*daltonproject.program.OrbitalEnergies attribute*), 39
orbital_energies_per_sym (*daltonproject.program.OrbitalEnergies attribute*), 39
orbital_energy (*daltonproject.program.OrbitalEnergy attribute*), 39
OrbitalCoefficients (*class in daltonproject.program*), 39
OrbitalEnergies (*class in daltonproject.program*), 39
OrbitalEnergy (*class in daltonproject.program*), 39
oscillator_strengths (*daltonproject.dalton.OutputParser property*), 44
oscillator_strengths (*daltonproject.lsdalton.OutputParser property*), 49
oscillator_strengths (*daltonproject.program.OscillatorStrengths attribute*), 39
oscillator_strengths (*daltonproject.program.OutputParser property*), 41
oscillator_strengths_per_sym (*daltonproject.program.OscillatorStrengths attribute*), 39
OscillatorStrengths (*class in daltonproject.program*), 39
OutputParser (*class in daltonproject.dalton*), 43
OutputParser (*class in daltonproject.gaussian*), 47
OutputParser (*class in daltonproject.lsdalton*), 49
OutputParser (*class in daltonproject.program*), 39
overlap_matrix() (*in module daltonproject.lsdalton*), 58

P

pick_cas_by_thresholds() (*in module daltonproject.natural_occupation*), 63
plot_ir_spectrum() (*in module daltonproject.spectrum*), 76
plot_one_photon_spectrum() (*in module daltonproject.spectrum*), 77
plot_raman_spectrum() (*in module daltonproject.spectrum*), 77
plot_spectrum() (*in module daltonproject.spectrum*), 78
plot_two_photon_spectrum() (*in module daltonproject.spectrum*), 78
Polarizabilities (*class in daltonproject.program*), 41
polarizabilities (*daltonproject.dalton.OutputParser property*), 44
polarizabilities (*daltonproject.gaussian.OutputParser property*), 47
polarizabilities (*daltonproject.program.OutputParser property*), 41
polarizabilities() (*daltonproject.property.Property method*), 36
polarizability_gradients (*daltonproject.lsdalton.OutputParser property*), 49
polarizability_gradients (*daltonproject.program.OutputParser property*), 41
polarizability_gradients() (*daltonproject.property.Property method*), 36
PolarizabilityGradients (*class in daltonproject.program*), 41
polarization (*daltonproject.program.HyperfineCouplings attribute*), 38
process_property_parsers() (*daltonproject.vibrational_averaging.ComputeVibAvCorrection method*), 72
Program (*class in daltonproject.program*), 41
Property (*class in daltonproject.property*), 35
property_first_derivatives (*daltonproject.vibrational_averaging.ComputeVibCorrection attribute*), 73
property_second_derivatives (*daltonproject.vibrational_averaging.ComputeVibCorrection attribute*), 73

Q

qc_method() (*daltonproject.qcmethod.QCMethod method*), 34
QCMethod (*class in daltonproject.qcmethod*), 33

R

raman_intensities (*daltonproject.vibrational_analysis.VibrationalAnalysis attribute*), 67
range_separation_parameter() (*daltonproject.qcmethod.QCMethod method*), 34
ri2() (*in module daltonproject.lsdalton*), 58
ri3() (*in module daltonproject.lsdalton*), 58
rotate_molecule_pseudo_convention() (*daltonproject.symmetry.SymmetryAnalysis method*), 62
rotational_contribution() (*daltonproject.vibrational_averaging.ComputeVibAvCorrection method*), 72

S

scan_occupations() (*in module daltonproject.natural_occupation*), 64
scf_threshold() (*daltonproject.qcmethod.QCMethod method*), 34
scratch_dir (*daltonproject.program.ComputeSettings property*), 37
slurm (*daltonproject.program.ComputeSettings property*), 38
slurm_account (*daltonproject.program.ComputeSettings property*), 38
slurm_partition (*daltonproject.program.ComputeSettings property*), 38
slurm_walltime (*daltonproject.program.ComputeSettings property*), 38
sort_natural_occupations() (*in module daltonproject.natural_occupation*), 64
spin_spin_couplings (*daltonproject.dalton.OutputParser property*), 44
spin_spin_couplings (*daltonproject.gaussian.OutputParser property*), 47
spin_spin_couplings (*daltonproject.program.OutputParser property*), 41
spin_spin_couplings() (*daltonproject.property.Property method*), 36
spin_spin_labels (*daltonproject.dalton.OutputParser property*), 44
spin_spin_labels (*daltonproject.gaussian.OutputParser property*), 47
spin_spin_labels (*daltonproject.program.OutputParser property*), 41
step_modifier() (*daltonproject.vibrational_averaging.VibAvSettings static method*), 74
strong_nat_occ (*daltonproject.natural_occupation.NatOrbOcc attribute*), 63
strong_nat_occ_nosym (*daltonproject.natural_occupation.NatOrbOcc attribute*), 63
strong_nat_occ_nosym2sym_idx (*daltonproject.natural_occupation.NatOrbOcc attribute*), 63
symmetry (*daltonproject.program.OrbitalEnergy attribute*), 39
symmetry() (*daltonproject.property.Property method*), 36
symmetry_elements (*daltonproject.symmetry.SymmetryAnalysis attribute*), 61
symmetry_threshold (*daltonproject.symmetry.SymmetryAnalysis attribute*), 61
SymmetryAnalysis (*class in daltonproject.symmetry*), 61

T

target_state() (*daltonproject.qcmethod.QCMethod method*), 34
tot_num_basis_functions (*daltonproject.program.NumBasisFunctions attribute*), 38
tot_num_orbitals (*daltonproject.program.NumOrbitals attribute*), 38
total (*daltonproject.program.HyperfineCouplings attribute*), 38
transition_state() (*daltonproject.property.Property method*), 36
trim_natural_occupations() (*in module daltonproject.natural_occupation*), 64
two_photon_absorption() (*daltonproject.property.Property method*), 36
two_photon_cross_sections (*daltonproject.dalton.OutputParser property*), 44
two_photon_cross_sections (*daltonproject.program.OutputParser property*), 41
two_photon_strengths (*daltonproject.dalton.OutputParser property*), 44
two_photon_strengths (*daltonproject.program.OutputParser property*), 41
two_photon_tensors (*daltonproject.dalton.OutputParser property*), 44
two_photon_tensors (*daltonproject.program.OutputParser property*), 41

V

validate_basis() (*in module daltonproject.basis*), 31
values (*daltonproject.program.OpticalRotations attribute*), 39
values (*daltonproject.program.Polarizabilities attribute*), 41
values (*daltonproject.program.PolarizabilityGradients attribute*), 41
VibAvConstants (*class in daltonproject.vibrational_averaging*), 73
VibAvSettings (*class in daltonproject.vibrational_averaging*), 73
vibrational_analysis() (*in module daltonproject.vibrational_analysis*), 69
vibrational_corrections (*daltonproject.vibrational_averaging.ComputeVibCorrection attribute*), 73
VibrationalAnalysis (*class in daltonproject.vibrational_analysis*), 67

W

weak_nat_occ (*daltonproject.natural_occupation.NatOrbOcc attribute*), 63
weak_nat_occ_nosym (*daltonproject.natural_occupation.NatOrbOcc attribute*), 63
weak_nat_occ_nosym2sym_idx (*daltonproject.natural_occupation.NatOrbOcc attribute*), 63
work_dir (*daltonproject.program.ComputeSettings property*), 38
write() (*daltonproject.basis.Basis method*), 31
write_gau_input() (*daltonproject.vibrational_averaging.VibAvSettings method*), 74
write_output_file() (*daltonproject.vibrational_averaging.ComputeVibAvCorrection method*), 72
write_xyz_input() (*daltonproject.vibrational_averaging.VibAvSettings method*), 74

X

xc_functional() (*daltonproject.qcmethod.QCMethod method*), 34
xyz (*daltonproject.symmetry.SymmetryAnalysis attribute*), 61
xyz() (*daltonproject.molecule.Molecule method*), 29