opan Documentation

Release 0.4rc1

Brian Skinn

10 May 2019

Contents

Open Anharmonic is a Python 3 wrapper for computational chemistry software packages intended to enable VPT2 computation of anharmonic vibrational constants. The code is still in the preliminary stages of development; no VPT2 functionality is yet available.

Other types of calculations are under consideration.

An adjunct goal of the project is to expose an API providing convenient access to various results of standalone calculations, as well as tools to manipulate those results. In particular, *OpanXYZ* and the subclasses of *SuperOpanGrad* and *SuperOpanHess* are anticipated to be particularly useful.

Due to the large number of possible variations of computational runs, parsing of output files is challenging, and only a small number of run types have been implemented to date. More are planned, but are currently low priority.

Open Anharmonic is available on PyPI as `opan`:

```
pip install opan
```

See the *dependencies page* for package dependencies and compatible versions. Note that due to common complications on Windows systems, dependencies are **NOT** set to automatically install.

The source repository for Open Anharmonic can be found at:

> https://www.github.com/bskinn/opan

Bug reports and feature requests can be submitted as GitHub Issues. Other feedback is welcomed at:

```
bskinn at alum dot mit dot edu
```

Contents

CHAPTER 1

Open Anharmonic User's Guide

Eventually there will be an introduction to the User's Guide here...

Contents:

1.1 Open Anharmonic - Usage & Examples

This will tell about how to actually use the package, with a focus on interactive usage.

Contents

1.1.1 const

The members of this module fall into three general categories:

- *Atomic number/symbol interconversion*
- *String and numerical constants*
- *Enumerations*

Atomic Numbers & Symbols

Conversions between atomic numbers and symbols is provided by two `dict` members.

atom_num provides the atomic number of an atomic symbol passed in **ALL CAPS**:

```
>>> opan.const.atom_num['CU']
29
```

atom_sym provides the atomic symbol corresponding to a given atomic number:

```
>>> opan.const.atom_sym[96]
'CM'
```

The elements hydrogen ($Z = 1$) through lawrencium ($Z = 103$) are currently supported.

String/Numerical Constants

The purpose of most of these classes and their member values is sufficiently explained in the respective *API entries*. The classes anticipated to be most useful to users are the physical constants of *PHYS*:

```
>>> from opan.const import PHYS
>>> PHYS.ANG_PER_BOHR
0.52917721067
>>> PHYS.LIGHT_SPEED                # Atomic units
137.036
```

as well as the string representations of engineering units of *UNITS*:

```
>>> from opan.const import UNITS
>>> from opan.const import EnumUnitsRotConst as EURC
>>> UNITS.rot_const[EURC.INV_INERTIA]
'1/(amu*B^2)'
>>> UNITS.rot_const[EURC.ANGFREQ_SECS]
'1/s'
```

Two of the remaining classes, *DEF* and *PRM*, define default values that are primarily relevant to programmatic use of `opan`. In unusual circumstances, though, they may be useful in console interactions.

CIC currently covers a very limited scope (the minimum and maximum atomic numbers implemented) and will likely not be useful at the console.

Enumerations

From the perspective of the end user, enumerations in `opan` are "functional types," which don't need instantiation before use:

```
>>> opan.const.EnumDispDirection.NO_DISP
'NO_DISP'
```

The enumeration values are simple strings:

```
>>> type(opan.const.EnumDispDirection.NO_DISP)
<class 'str'>
```

While this implementation is susceptible to accidental mixing of enumerated types, it has the advantage of allowing simple `str` inputs to functions expecting enumerated values. This is anticipated to be useful in console-level interactions with a variety of program elements. For example, the engineering units to be output from *opan.utils.inertia.rot_consts()* can be specified simply with the appropriate string, instead of the fully specified enumeration object:

```
>>> from opan.utils.inertia import rot_consts
>>> from opan.const import EnumUnitsRotConst as EURC
>>> rot_consts(some_geom, some_masses, 'INV_INERTIA')       # Works fine
array(...)
```

(continues on next page)

(continued from previous page)

```
>>> rot_consts(some_geom, some_masses, EURC.INV_INERTIA)      # Also works
array(...)
```

As noted in the API documentation for *EnumIterMeta*, both iteration and membership testing with "in" are supported:

```
>>> 'NO_DISP' in opan.const.EnumDispDirection
True
>>> [e for e in sorted(opan.const.EnumDispDirection)]
['NEGATIVE', 'NO_DISP', 'POSITIVE']
```

1.1.2 error

Note: Most interactive use of opan will not require detailed knowledge of the custom errors in this module.

The custom exceptions in this module are all subclassed from *opan.error.OpanError*, which itself is a subclass of Exception. In addition to the typical error message included as part of initializing an Exception, the custom error subclasses of *OpanError* also define a typecode and a source attribute (typically a filename or other data source) to allow more finely-grained definition of error origins and types. In the below example, attempting to import the source file for this usage page as an OpenBabel XYZ file quite sensibly results in an error:

```
>>> x = opan.xyz.OpanXYZ(path='error.rst')
Traceback (most recent call last):
...
XYZError: (XYZFILE) No valid geometry found: XYZ file: error.rst
```

The custom exception *XYZError* is raised with typecode *XYZFILE*, indicating a problem with the indicated input file. The external data source causing the exception is included after the final colon (error.rst, this file). If no data source is relevant to a given exception, it is omitted.

The subclasses of *opan.const.OpanEnum* are equipped with membership testing of and iteration over their respective typecodes:

```
>>> 'XYZFILE' in XYZError
True
>>> [tc for tc in sorted(XYZError)]
['DIHED', 'NONPRL', 'OVERWRITE', 'XYZFILE']
```

Raising these exceptions follows standard syntax, save for the extra 'typecode' and 'source' parameters:

```
>>> raise XYZError(XYZError.OVERWRITE, "Spurious overwrite", "Console")
Traceback (most recent call last):
...
XYZError: (OVERWRITE) Spurious overwrite: Console
```

1.1.3 grad

Draft scratch content...

Gradient objects are a thing that have to be properly done in order for various of the automation functiony applications of the opan to work right, since different softwares make their things in different ways, but the core automating thingy has to not care what software made the data but the data has to be presented uniformly. Most console interactions with

`opan` won't care about this much, but it's worth noting here the things that can be expected from ALL thingies, even though many other thingies will likely be available for any given other various software.

Firstly, the instance members specified as having to be there by *`SuperOpanGrad`*:

- `gradient` – 1-D `np.array` of `np.float` – Gradient data in $\left(\frac{\mathrm{E_h}}{\mathrm{B}}\right)$ *units*
- `geom` – 1-D `np.array` of `np.float` – Geometry data in B *units*
- `atom_syms` – `list` of `str` – Atomic symbols in **ALL CAPS**

There will need to be a private `_load` method, but that shouldn't ever be useful or usable, interactively.

Other than that, the below subpages describe the software-specific data available in the specific subclass objects of *`SuperOpanGrad`*.

Implemented Subclasses

OrcaEngrad

Stuff

1.1.4 hess

Stuff

OrcaHess

Stuff

1.1.5 output

Stuff

OrcaOutput

Stuff

1.1.6 utils

Stuff

utils (base)

Stuff

utils.decorate

arraysqueeze

Stuff

kwargfetch

This decorator is intended for use within sets of nested functions, where a call to an "outer" function results in a chain of calls into the nesting structure. If each of these nested calls requires

Dummy test block:

```
>>> def f_2p(a, b):
...     return 2 * a + 3 * b

>>> @kwargfetch('kw', f_2p, 1, 'm')
... def testfxn(x, y, **kwargs):
...     return (y - x) * kwargs['kw']

>>> testfxn(3, 7, m=4)
... # kw=f_2p(7, 4) = 26
... # testfxn then returns 4*26 = 104
104

>>> testfxn(y=7, m=4, x=3)
... # kwargfetch is robust against positional arguments
... # passed as keyword arguments
104
```

utils.execute

Stuff

utils.inertia

Stuff

utils.symm

This module doesn't really work at present. It is planned to eventually attempt to fix it...

utils.vector

Stuff

1.1.7 vpt2

This module doesn't really exist yet. Sorry.

1.1.8 xyz

Stuff

1.2 Open Anharmonic - Theory

This will introduce the various calculations for which theory is provided here.

Contents:

1.2.1 Vector Operations

This will introduce the various vector operation sections.

1.2.2 Symmetry Operations

Introduction to discussion of implemented symmetry operations.

1.2.3 Inertial Properties

This will introduce the exposition of the inertial properties.

1.2.4 Harmonic Vibrational Frequencies

Intro about what's laid out about harmonic frequencies.

1.2.5 VPT2 Anharmonic Frequencies

Intro to everything described about VPT2.

1.2.6 Centrifugal Distortion

Intro to what's exposited about the centrifugal distortion calculations.

CHAPTER 2

Open Anharmonic API

The full public API is delineated below. Cross-references to NumPy are displayed with the `np` package abbreviation.

Note: Documented or undocumented attributes, functions, methods, etc. named with a leading underscore are **NOT** formally part of the API. Their behavior may change without notice.

2.1 opan.const

Defines objects bearing assorted constants for Open Anharmonic.

2.1.1 Module-Level Members

opan.const.**infty**
: str – Unicode infinity symbol

opan.const.**atom_num**
: dict – Atomic number lookup from element symbol

> **Note:** Keys for *atom_num* are **all uppercase** (e.g., 'AR' for argon)

opan.const.**atom_sym**
: dict – Element symbol lookup from atomic number, returned as **all uppercase**

2.1.2 Classes

Overview

Constants Classes

CIC – Application-internal code information constants

DEF – Default values for parameters intended to be user-adjustable

PHYS – Physical constants

PRM – Internal computation parameters, intended to be non-user-adjustable

UNITS – Functions returning text strings of units descriptions

Enumeration Classes

EnumIterMeta – Metaclass for iterable enumerations supporting membership testing with *in*

OpanEnum – Superclass for enumerations

Plain Enumerations

EnumCheckGeomMismatch – Mismatch type found during *check_geom()* comparison checks of two geometries.

EnumDispDirection – Displacement direction along a particular mode

EnumFileType – Various file types relevant to the software packages

EnumMassPertType – Type of atomic mass perturbation being applied

EnumSoftware – Implemented computational software packages

EnumTopType – Molecular top classification

Anharmonic (VPT2) HDF5 Repository Enumerations

EnumAnharmRepoData – Displacement-specific values

EnumAnharmRepoParam – Displacement-nonspecific values

Units Enumerations

Units implemented for numerical conversions for various physical quantities.

EnumUnitsRotConst – Rotational constants

API

class opan.const.**CIC**
Bases: object

Container for application-internal code information constants

These may need expansion into dictionaries keyed by *EnumSoftware* enum values, depending on the atoms supported by various software packages. They may also require adjustment to accommodate 'special' atom types such as ghost atoms and point charges.

Members

MAX_ATOMIC_NUM = 103
int – Maximum atomic number supported

MIN_ATOMIC_NUM = 1
int – Minimum atomic number supported

class opan.const.**DEF**
Bases: object

Container for default parameter values (possibly user-adjustable)

FILE_EXTS = {'ORCA': {'GRAD': 'engrad', 'HESS': 'hess', 'INPUTFILE': 'txt', 'OUTPUT':
dict of dict – Dictionary of dictionaries of file extensions for geometry, gradient, hessian, etc. files from the various software suites.

Access as FILE_EXTS[*EnumSoftware*][*EnumFileType*]

GRAD_COORD_MATCH_TOL = 1e-07
float – Max precision of GRAD geometries (currently *ORCA*-specific)

HESS_COORD_MATCH_TOL = 1e-06
float – Max precision of HESS geometries (currently *ORCA*-specific)

HESS_IR_MATCH_TOL = 0.01
float – Max precision of freqs in IR spectrum block (currently *ORCA*-specific)

MASS_PERT_MAG = 0.0001
float – Relative magnitude of atomic mass perturbations

ORTHONORM_TOL = 1e-08
float – Acceptable deviation from Kronecker delta for orthonormality testing

XYZ_COORD_MATCH_TOL = 1e-12
float – Max tolerable deviation between XYZ geoms (currently *ORCA*-specific)

class opan.const.**EnumAnharmRepoData**
Bases: *opan.const.OpanEnum*

Enumeration class for datatypes in VPT2 HDF5 repository.

Contains enumeration parameters to indicate the type of data to be retrieved from the on-disk HDF5 repository in the VPT2 anharmonic calculations of the *opan.vpt2* submodule.

Enum Values

ENERGY = 'ENERGY'
Energy value at the given displacement

GEOM = 'GEOM'
Geometry at the given displacement (max precision available)

GRAD = 'GRAD'
Cartesian gradient vector

HESS = 'HESS'
Cartesian Hessian matrix

class opan.const.**EnumAnharmRepoParam**
Bases: *opan.const.OpanEnum*

Enumeration class for parameters in VPT2 HDF5 repository.

Contains enumeration parameters to indicate the parameter value to be retrieved from the on-disk HDF5 repository in the VPT2 anharmonic calculations of the *opan.vpt2* submodule.

Enum Values

ATOMS = 'ATOMS'
Length-*N* vector of all-caps atomic symbols

CTR_MASS = 'CTR_MASS'
Cartesian center of mass of system, *with perturbations applied*, if any

INCREMENT = 'INCREMENT'
Displacement increment in $\mathrm{B\,u}^{1/2}$. Note that these are *not* atomic *units*, which would instead be $\mathrm{B\,m_e}^{1/2}$.

PERT_MODE = 'PERT_MODE'
opan.const.EnumMassPertType indicating perturbation type

PERT_VEC = 'PERT_VEC'
Length-*3*N* vector of perturbation factors (all should be ~1)

REF_MASSES = 'REF_MASSES'
Reference values of atomic masses (unperturbed)

class opan.const.**EnumCheckGeomMismatch**
Bases: *opan.const.OpanEnum*

Enumeration for mismatch types in *check_geom()*

Only mismatches of validly constructed coordinates and atoms vector combinations are represented here; other mismatches/misconfigurations result in raised Exceptions.

Enum Values

ATOMS = 'ATOMS'
Mismatch in individual atom(s)

COORDS = 'COORDS'
Mismatch in individual coordinate(s)

DIMENSION = 'DIMENSION'
Mismatch in dimensions of the two geometries

class opan.const.**EnumDispDirection**
Bases: *opan.const.OpanEnum*

Enumeration class for displacement directions.

Contains enumeration parameters to indicate the displacement of the molecular geometry associated with a gradient, hessian or other object.

Enum Values

NEGATIVE = 'NEGATIVE'
Negative displacement along a particular normal mode

NO_DISP = 'NO_DISP'
Non-displaced geometry

POSITIVE = 'POSITIVE'
Positive displacement along a particular normal mode

class opan.const.**EnumFileType**
Bases: *opan.const.OpanEnum*

Enumeration class for the file types generated by computational codes.

Enum Values

GRAD = 'GRAD'
Files containing nuclear gradient information

HESS = 'HESS'
Files containing nuclear Hessian information

INPUTFILE = 'INPUTFILE'
Input files for defining computations

OUTPUT = 'OUTPUT'
Files containing computational output

XYZ = 'XYZ'
XYZ atomic coordinates, assumed to follow the Open Babel XYZ specification

class opan.const.**EnumIterMeta**
Bases: type

Metaclass for enumeration types allowing *in* membership testing.

__iter__()
Iterate over all defined enumeration values.

Generator iterating over all class variables whose names match their contents. For a properly constructed *OpanEnum* subclass, these are identical to the enumeration values.

Example:

```
>>> [val for val in sorted(opan.const.EnumDispDirection)]
['NEGATIVE', 'NO_DISP', 'POSITIVE']
```

__contains__(*value*)
Returns True if *value* is a valid value for the enumeration type, else False.

Example:

```
>>> 'NO_DISP' in opan.const.EnumDispDirection
True
```

class opan.const.**EnumMassPertType**
Bases: *opan.const.OpanEnum*

Enumeration class for atom mass perturbation types.

Contains enumeration parameters to indicate the type of mass perturbation to be applied to the various atoms of a geometry, in order to: (a) break inertial degeneracy sufficiently to allow VPT2 computation using a lower-symmetry formalism; and (b), in the case of linear molecules, introduce an artificial 'directional preference' to the masses of the atoms (*BY_COORD* enum value) to break the intrinsic degeneracy of the bending modes.

Enum Values

BY_ATOM = 'BY_ATOM'
Atomic masses are perturbed atom-by-atom in an isotropic fashion

BY_COORD = 'BY_COORD'
Atomic masses are perturbed anisotropically, where the perturbation factor for each atom's mass varies slightly in the x-, y-, and z-directions

NO_PERTURB = 'NO_PERTURB'
Atomic masses are used without modification

class opan.const.**EnumSoftware**
Bases: *opan.const.OpanEnum*

Enumeration class for identifying computational chemistry packages.

This enum will be expanded if/when support for additional packages is implemented.

Enum Values

ORCA = 'ORCA'
The *ORCA* program package

class opan.const.**EnumTopType**
Bases: *opan.const.OpanEnum*

Enumeration class for classifying types of molecular tops.

Contains enumeration parameters to indicate the type of molecular top associated with a particular geometry.

Inertial moments with magnitudes less than *opan.const.PRM.ZERO_MOMENT_TOL* are taken as zero. Nonzero moments by this metric are considered to be equal if their ratio differs from unity by less than *opan.const.PRM.EQUAL_MOMENT_TOL*. See *opan.utils.inertia.principals()* and the other functions defined in *opan.utils.inertia* for more details.

Enum Values

ASYMM = 'ASYMM'
Three unique non-zero moments

ATOM = 'ATOM'
Three zero principal inertial moments

LINEAR = 'LINEAR'
One zero and two equal non-zero moments

SPHERICAL = 'SPHERICAL'
Three equal, non-zero moments

SYMM_OBL = 'SYMM_OBL'
Three non-zero moments; smallest two equal

SYMM_PROL = 'SYMM_PROL'
Three non-zero moments; largest two equal

class opan.const.**EnumUnitsRotConst**
Bases: *opan.const.OpanEnum*

Units Enumeration class for rotational constants.

Contains enumeration parameters to indicate the associated/desired units of interpretation/display of a rotational constant.

String expressions of these units are provided in *UNITS.rot_const*.

Todo: Add link to exposition(?) of how RotConst expression is developed, once written.

Enum Values

ANGFREQ_ATOMIC = 'ANGFREQ_ATOMIC'
Angular frequency in atomic *units*, $\frac{1}{\mathrm{T_a}}$

ANGFREQ_SECS = 'ANGFREQ_SECS'
Angular frequency in SI units, $\frac{1}{\mathrm{s}}$ (**NOT** Hz!)

CYCFREQ_ATOMIC = 'CYCFREQ_ATOMIC'
Cyclic frequency in atomic *units*, $\frac{\mathrm{cyc}}{\mathrm{T_a}}$

CYCFREQ_HZ = 'CYCFREQ_HZ'
Cyclic frequency in Hz, $\frac{\mathrm{cyc}}{\mathrm{s}}$

CYCFREQ_MHZ = 'CYCFREQ_MHZ'
Cyclic frequency in MHz, millions of $\frac{\mathrm{cyc}}{\mathrm{s}}$

INV_INERTIA = 'INV_INERTIA'
Inverse moment of inertia, $\frac{1}{\mathrm{u\,B^2}}$. Note that the mass *units* here are *not* atomic units, which would require $\frac{1}{\mathrm{m_e\,B^2}}$.

WAVENUM_ATOMIC = 'WAVENUM_ATOMIC'
Wavenumbers in atomic *units*, $\frac{\mathrm{cyc}}{\mathrm{B}}$

WAVENUM_CM = 'WAVENUM_CM'
Wavenumbers in conventional units, $\frac{\mathrm{cyc}}{\mathrm{cm}}$

class opan.const.**OpanEnum**
Bases: object

Superclass for enumeration objects.

Metaclassed with *EnumIterMeta* to allow direct iteration and membership testing of enumeration values on the subclass type.

class opan.const.**PHYS**
Bases: object

Container for physical constants

Members

ANG_PER_BOHR = 0.52917721067
float – Angstroms per Bohr radius (source: NIST)

LIGHT_SPEED = 137.036
float – Speed of light in atomic *units*, $\frac{B}{T_a}$. Calculated from the NIST value for the speed of light in vacuum, $2.99792458e8\ \frac{m}{s}$, using *ANG_PER_BOHR* and *SEC_PER_TA* as conversion factors

ME_PER_AMU = 1822.8885
float – Electron mass per unified atomic mass unit (source: NIST)

PLANCK = 6.283185307179586
float – Standard Planck constant, equal to 2π in atomic *units* of $\frac{\mathrm{E_h\,T_a}}{\mathrm{cyc}}$

PLANCK_BAR = 1.0
float – Reduced Planck constant, unity by definition in the atomic *units* of $\mathrm{E_h\ T_a}$

SEC_PER_TA = 2.4188843265e-17
float – Seconds per atomic time unit (source: NIST)

class opan.const.**PRM**
Bases: object

Container for internal computation parameters (not user-adjustable)

Members

EQUAL_MOMENT_TOL = 0.001
float – Minimum deviation-ratio from unity below which two principal inertial moments are considered equal

MAX_SANE_DIPDER = 100.0
float – Trap value for aberrantly large dipole derivative values in *ORCA* if dipoles are not calculated in a NUMFREQ run

NON_PARALLEL_TOL = 0.001
float – Minimum angle deviation (degrees) required for two vectors to be considered non-parallel

ZERO_MOMENT_TOL = 0.001
float – Threshold value below which moments are considered equal to zero; *units* of $\mathrm{u\,B^2}$

ZERO_VEC_TOL = 1e-06
float – Vector magnitude below which a vector is considered equal to the zero vector; dimensionless or *units* of B

class opan.const.**UNITS**
Bases: object

Container for dicts providing strings describing the various display units available for physical quantities.

Dictionary keys are the enum values provided in the corresponding `EnumUnits[...]` class in this module (*opan.const*).

Dictionary	Enum	Physical Quantity
rot_const	*EnumUnitsRotConst*	Rotational constant

rot_const = {'ANGFREQ_ATOMIC': '1/Ta', 'ANGFREQ_SECS': '1/s', 'CYCFREQ_ATOMIC': 'cyc/T
dict –

2.2 opan.error

Defines custom errors for Open Anharmonic

Error classes are subclassed from `Exception` via an abstract superclass, *OpanError*, which defines several common features:

- Storage of a 'typecode' and a 'source' string along with the error message to allow passing of more fine-grained information to the exception stack
- Implementation of the *EnumIterMeta* metaclass on *OpanError*, enabling typecode validity checking with `in`
- Re-implementation of `__str__()` to enhance the usefulness of stack messages when one of these errors is raised

OpanError **Subclasses**

AnharmError – Raised as a result of *OpanVPT2* actions

GradError – Raised during parsing of or calculations using gradient data

HessError – Raised during parsing of or calculations using Hessian data

InertiaError – Raised by *opan.utils.inertia* submodule functions

OutputError – Raised during parsing of or calculations using output data

RepoError – Raised by HDF5 repository interactions

SymmError – Raised by *opan.utils.symm* submodule functions

VectorError – Raised by *opan.utils.vector* submodule functions

XYZError – Raised during parsing of or calculations using XYZ data

---

API

exception opan.error.**AnharmError**(*tc*, *msg*, *src*)

Bases: *opan.error.OpanError*

Error relating to *OpanVPT2* actions.

See the *OpanError* documentation for more information on attributes, methods, etc.

Todo: Add object references once the DOM is more established?

Typecodes

REPO = 'REPO'
OpanAnharmRepo conflict – no repo bound when assignment attempted, or attempt made to bind new repo when one already bound

STATUS = 'STATUS'
OpanVPT2 internal variables in inappropriate state for the requested operation

exception opan.error.**GradError**(*tc*, *msg*, *src*)

Bases: *opan.error.OpanError*

Error relating to parsing of or calculation from gradient data.

Not all typecodes may be relevant for all software packages.

See the *OpanError* documentation for more information on attributes, methods, etc.

Typecodes

BADATOM = 'BADATOM'
Missing or invalid atom symbols; SHOULD only be used by *SuperOpanGrad*

BADGEOM = 'BADGEOM'
Missing or invalid geometry data; SHOULD only be used by *SuperOpanGrad*

BADGRAD = 'BADGRAD'
Missing or invalid gradient data; SHOULD only be used by *SuperOpanGrad*

ENERGY = 'ENERGY'
Energy value not found

GEOMBLOCK = 'GEOMBLOCK'
Malformed or missing geometry block

GRADBLOCK = 'GRADBLOCK'
Malformed or missing gradient block

NUMATS = 'NUMATS'
Invalid number-of-atoms specification, or specification not found

OVERWRITE = 'OVERWRITE'
Object already initialized (overwrite not supported)

exception opan.error.**HessError**(*tc*, *msg*, *src*)
Bases: *opan.error.OpanError*

Error relating to parsing of or calculation from Hessian data.

Not all typecodes may be relevant for all software packages.

See the *OpanError* documentation for more information on attributes, methods, etc.

Typecodes

AT_BLOCK = 'AT_BLOCK'
Malformed or missing atom/geometry specification block

BADATOM = 'BADATOM'
Missing or invalid atom symbols; SHOULD only be used by *SuperOpanHess*

BADGEOM = 'BADGEOM'
Missing or invalid geometry data; SHOULD only be used by *SuperOpanHess*

BADHESS = 'BADHESS'
Missing or invalid Hessian data; SHOULD only be used by *SuperOpanHess*

DIPDER_BLOCK = 'DIPDER_BLOCK'
Malformed dipole derivatives block

EIGVAL_BLOCK = 'EIGVAL_BLOCK'
Malformed mass-weighted-Hessian eigenvalues block

EIGVEC_BLOCK = 'EIGVEC_BLOCK'
Malformed mass-weighted-Hessian eigenvectors block

ENERGY = 'ENERGY'
Malformed or missing energy value

FREQ_BLOCK = 'FREQ_BLOCK'
Malformed or missing frequencies block

HESS_BLOCK = 'HESS_BLOCK'
Malformed or missing Hessian block

IR_BLOCK = 'IR_BLOCK'
Malformed IR spectrum block

JOB_BLOCK = 'JOB_BLOCK'
Malformed job list block

MODES_BLOCK = 'MODES_BLOCK'
Malformed or missing normal modes block

OVERWRITE = 'OVERWRITE'
Object already initialized (overwrite not supported)

POLDER_BLOCK = 'POLDER_BLOCK'
Malformed polarizability derivatives block

RAMAN_BLOCK = 'RAMAN_BLOCK'
Malformed Raman spectrum block

TEMP = 'TEMP'
Malformed or missing temperature value

exception opan.error.**InertiaError**(*tc*, *msg*, *src*)
Bases: *opan.error.OpanError*

Error relating to *opan.utils.inertia* submodule functions.

See the *OpanError* documentation for more information on attributes, methods, etc.

Typecodes

BAD_GEOM = 'BAD_GEOM'
A geometry being parsed was unsuitable for a particular type of calculation/manipulation

NEG_MOMENT = 'NEG_MOMENT'
A negative principal inertial moment was computed

TOP_TYPE = 'TOP_TYPE'
No valid molecular top type was identified

exception opan.error.**OpanError**(*tc*, *msg*, *src*)
Bases: Exception

Base class for custom errors defined for Open Anharmonic

OpanError is an abstract superclass of any custom errors defined under the Open Anharmonic umbrella. It defines all common methods shared among the various subtype error classes, such that the only contents that must be declared by a subclass are str class variables with contents identical to their names. These are recognized by the *__iter__()* defined in *opan.const.EnumIterMeta* as being the set of valid typecodes.

Parameters

- **tc** – str – String representation of typecode to be associated with the *OpanError* subclass instance. *Must* be a validly constructed typecode defined for the relevant subclass.
- **msg** – str – Explanation of the nature of the error being reported
- **src** – str – Further detail of the code/file source of the error behavior, if relevant

Raises

- NotImplementedError – Upon attempt to instantiate abstract *OpanError* base class
- KeyError – Upon instantiation with an invalid typecode

msg
str – Explanation of the nature of the error being reported

src
str – Further detail of the code source of the error behavior

subclass_name
str – String representation of the *OpanError* subclass name

tc
str – String typecode associated with the instance

exception opan.error.**OutputError**(*tc*, *msg*, *src*)
Bases: *opan.error.OpanError*

Error relating to parsing of or calculation from output data.

See the *OpanError* documentation for more information on attributes, methods, etc.

Typecodes

(none yet)

exception opan.error.**RepoError**(*tc*, *msg*, *src*)
Bases: *opan.error.OpanError*

Error relating to HDF5 repository interactions.

See the *OpanError* documentation for more information on attributes, methods, etc.

Typecodes

DATA = 'DATA'
Problem with a dataset in linked HDF5 `File`

GROUP = 'GROUP'
Problem with a group in linked HDF5 `File`

STATUS = 'STATUS'
HDF5 repo in improper status for requested operation

exception opan.error.**SymmError**(*tc*, *msg*, *src*)
Bases: *opan.error.OpanError*

Error relating to *opan.utils.symm* submodule functions.

See the *OpanError* documentation for more information on attributes, methods, etc.

Todo: Add note about this being subject to future development?

Typecodes

NOTFOUND = 'NOTFOUND'
Symmetry element expected but not found

exception opan.error.**VectorError**(*tc*, *msg*, *src*)
Bases: *opan.error.OpanError*

Error relating to *opan.utils.vector* submodule functions.

See the *OpanError* documentation for more information on attributes, methods, etc.

Typecodes

NONPRL = 'NONPRL'
Insufficient non-parallel character in some manner of calculation

ORTHONORM = 'ORTHONORM'
Vectors which should have been orthonormal were determined not to be

exception opan.error.**XYZError**(*tc*, *msg*, *src*)
Bases: *opan.error.OpanError*

Error relating to parsing of or calculation using XYZ data.

See the *OpanError* documentation for more information on attributes, methods, etc.

Typecodes

DIHED = 'DIHED'
Dihedral angle calculation requested for a set of atoms containing an insufficiently nonlinear trio of atoms

NONPRL = 'NONPRL'
Insufficient non-parallel character in some manner of calculation

OVERWRITE = 'OVERWRITE'
Object already initialized (overwrite not supported)

XYZFILE = 'XYZFILE'
Inconsistent geometry in an OpenBabel XYZ file

- *ORCA* – *.xyz* or *.trj*

2.3 opan.grad

Module implementing imports of gradient data from external computations.

The abstract superclass *SuperOpanGrad* defines a common initializer and common method(s) that for use by subclasses importing gradient data from external computational packages.

Implemented Subclasses

OrcaEngrad – Imports '.engrad' files from *ORCA*

Requirements

- The import for each external software package SHOULD have its own subclass.
- Each subclass MUST implement a `_load(self, **kwargs)` instance method as the entry point for import of gradient data.
- The gradient data MUST be stored:
  - In the instance member `self.gradient`
  - As a one-dimensional `np.array`
  - With *dtype* descended from `np.float`
  - In *units* of Hartrees per Bohr $\left(\frac{\mathrm{E_h}}{\mathrm{B}}\right)$
  - With elements ordered as:

$$\begin{bmatrix} \frac{\partial E}{\partial x_1} & \frac{\partial E}{\partial y_1} & \frac{\partial E}{\partial z_1} & \frac{\partial E}{\partial x_2} & \frac{\partial E}{\partial y_2} & \cdots & \frac{\partial E}{\partial y_N} & \frac{\partial E}{\partial z_N} \end{bmatrix}$$

- The geometry data MUST be stored:
  - In the instance member `self.geom`
  - As a one-dimensional `np.array`

  - With *dtype* descended from `np.float`
  - In *units* of Bohrs (B)
  - With elements ordered as:

$$\begin{bmatrix} x_1 & y_1 & z_1 & x_2 & y_2 & \dots & y_N & z_N \end{bmatrix}$$

- The atoms list MUST be stored:
  - In the instance member `self.atom_syms`
  - As a `list` of `str`, with each atom specified by an **all-caps** atomic symbol (*opan.const.atom_sym* may be helpful)
- Subclasses MAY define an unlimited number of methods, class variables, and/or instance variables of unrestricted type, in addition to those defined above.

Superclass

class opan.grad.**SuperOpanGrad**(***kwargs*)

Abstract superclass of gradient import classes.

Performs the following actions:

1. Ensures that the abstract superclass is not being instantiated, but instead a subclass.
2. Calls the `_load()` method on the subclass, passing all *kwargs* through unmodified.
3. Typechecks the `self.gradient`, `self.geom`, and `self.atom_syms` required members for existence, proper data type, and properly matched lengths.

The checks performed in step 3 are primarily for design-time member- and type-enforcement during development of subclasses for newly implemented external software packages, rather than run-time data checking. It is RECOMMENDED to include robust data validity checking inside each subclass, rather than relying on these tests.

Methods

check_geom(*coords*, *atoms*[, *tol*])

Check for consistency of gradient geometry with input coords/atoms.

The cartesian coordinates associated with a gradient object are considered consistent with the input *coords* if each component matches to within *tol*. If *coords* or *atoms* vectors are passed that are of different length than those stored in the instance, a `False` value is returned, rather than an exception raised.

Parameters

- **coords** – length-3N `np.float_` – Vector of stacked 'lab-frame' Cartesian coordinates
- **atoms** – length-N `str` or `int` – Vector of atom symbols or atomic numbers
- **tol** – `float`, optional Tolerance for acceptable deviation of each passed geometry coordinate from that in the instance to still be considered matching. Default value is *DEF.GRAD_COORD_MATCH_TOL*

See *opan.utils.check_geom* for details on return values and exceptions raised.

Subclasses

class opan.grad.**OrcaEngrad**(*path='...'*)
Container for gradient data generated by *ORCA*.

Initialize by passing the path to the file to be loaded as the *path* keyword argument.

Key information contained includes the gradient, the energy, the number of atoms, the geometry, and the atom IDs. For *ORCA*, the precision of the geometry is inferior to that in an XYZ file.

Methods

_load(***kwargs*)
Initialize *OrcaEngrad* object from .engrad file

Searches indicated file for energy, geometry, gradient, and number of atoms and stores in the corresponding instance variables.

Parameters **path** – str – Complete path to the .engrad file to be read.

Raises

- *GradError* – Typecode *OVERWRITE* – If OrcaEngrad instance has already been instantiated.

  Various typecodes – If indicated gradient file is malformed in some fashion

- IOError – If the indicated file does not exist or cannot be read

Class Variables

class Pat
re.compile() patterns for data parsing.

atblock
Captures the entire block of atom ID & geometry data.

atline
Extracts single lines from the atom ID / geometry block.

energy
Captures the electronic energy.

gradblock
Captures the gradient data block.

numats
Retrieves the stand-along 'number of atoms' field.

Instance Variables

atom_syms
length-N `list` of `str` – Uppercased atomic symbols for the atoms in the system.

energy
`float` – Single-point energy for the geometry.

geom
length-3N `np.float_` – Vector of the atom coordinates in B.

gradient
length-3N `np.float_` – Vector of the Cartesian gradient in $\frac{\mathrm{E_h}}{\mathrm{B}}$.

in_str
`str` – Complete text of the ENGRAD file read in to generate the *`OrcaEngrad`* instance.

num_ats
`int` – Number of atoms in the geometry ('N')

2.4 opan.hess

Module implementing imports of Hessian data from external computations.

The abstract superclass *`SuperOpanHess`* defines a common initializer and common method(s) for use by subclasses importing Hessian data from external computational packages.

Implemented Subclasses

`OrcaHess` – Imports '.hess' files from *ORCA*

Requirements

- The import for each external software package SHOULD have its own subclass.
- Each subclass MUST implement a `_load(**kwargs)` method as the entry point for import of Hessian data.
- The Hessian data MUST be stored:
  - In the instance member `self.hess`

  - As a two-dimensional `np.array`
  - With *dtype* descended from `np.float`
  - In *units* of Hartrees per Bohr-squared $\left(\frac{\mathrm{E_h}}{\mathrm{B}^2}\right)$
  - With elements arranged as:

$$\begin{bmatrix} \frac{\partial^2 E}{\partial x_1^2} & \frac{\partial^2 E}{\partial x_1 \partial y_1} & \frac{\partial^2 E}{\partial x_1 \partial z_1} & \frac{\partial^2 E}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 E}{\partial x_1 \partial y_N} & \frac{\partial^2 E}{\partial x_1 \partial z_N} \\ \frac{\partial^2 E}{\partial y_1 \partial x_1} & \frac{\partial^2 E}{\partial y_1^2} & \frac{\partial^2 E}{\partial y_1 \partial z_1} & \frac{\partial^2 E}{\partial y_1 \partial x_2} & \cdots & \frac{\partial^2 E}{\partial y_1 \partial y_N} & \frac{\partial^2 E}{\partial y_1 \partial z_N} \\ \frac{\partial^2 E}{\partial z_1 \partial x_1} & \frac{\partial^2 E}{\partial z_1 \partial y_1} & \frac{\partial^2 E}{\partial z_1^2} & \frac{\partial^2 E}{\partial z_1 \partial x_2} & \cdots & \frac{\partial^2 E}{\partial z_1 \partial y_N} & \frac{\partial^2 E}{\partial z_1 \partial z_N} \\ \frac{\partial^2 E}{\partial x_2 \partial x_1} & \frac{\partial^2 E}{\partial x_2 \partial y_1} & \frac{\partial^2 E}{\partial x_2 \partial z_1} & \frac{\partial^2 E}{\partial x_2^2} & \cdots & \frac{\partial^2 E}{\partial x_2 \partial y_N} & \frac{\partial^2 E}{\partial x_2 \partial z_N} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{\partial^2 E}{\partial y_N \partial x_1} & \frac{\partial^2 E}{\partial y_N \partial y_1} & \frac{\partial^2 E}{\partial y_N \partial z_1} & \frac{\partial^2 E}{\partial y_N \partial x_2} & \cdots & \frac{\partial^2 E}{\partial y_N^2} & \frac{\partial^2 E}{\partial y_N \partial z_N} \\ \frac{\partial^2 E}{\partial z_N \partial x_1} & \frac{\partial^2 E}{\partial z_N \partial y_1} & \frac{\partial^2 E}{\partial z_N \partial z_1} & \frac{\partial^2 E}{\partial z_N \partial x_2} & \cdots & \frac{\partial^2 E}{\partial z_N \partial y_N} & \frac{\partial^2 E}{\partial z_N^2} \end{bmatrix}$$

Note: The Hessian is elsewhere assumed to be symmetric (real-Hermitian), and thus MUST be returned as such here. Symmetric character is **NOT** explicitly checked, however!

- The geometry data MUST be stored:
  - In the instance member `self.geom`
  - As a one-dimensional `np.array`
  - With *dtype* descended from `np.float`
  - In *units* of Bohrs (B)
  - With elements ordered as:

$$\begin{bmatrix} x_1 & y_1 & z_1 & x_2 & y_2 & \cdots & y_N & z_N \end{bmatrix}$$

- The atoms list MUST be stored:
  - In the instance member `self.atom_syms`
  - As a `list` of `str`, with each atom specified by an **all-caps** atomic symbol (*`opan.const.atom_sym`* may be helpful)
- Subclasses MAY define an unlimited number of methods, class variables, and/or instance variables in addition to those defined above, of unrestricted type.

Superclass

class opan.hess.**SuperOpanHess**(***kwargs*)

Abstract superclass of Hessian import classes.

Performs the following actions:

1. Ensures that the abstract superclass is not being instantiated, but instead a subclass.
2. Calls the `_load()` method on the subclass, passing all *kwargs* through unmodified.

3. Typechecks the `self.hess`, `self.geom`, and `self.atom_syms` required members for existence, proper data type, and properly matched lengths/dimensions.

The checks performed in step 3 are primarily for design-time member and type enforcement during development of subclasses for new external software packages, rather than run-time data checking. It is RECOMMENDED to include robust data validity checking inside each subclass, rather than relying on these tests.

Methods

check_geom(*coords*, *atoms*[, *tol*])

Check for consistency of Hessian geometry with input coords/atoms.

The cartesian coordinates associated with a Hessian object are considered consistent with the input *coords* and *atoms* if each component matches to within *tol* and all atoms are identical. If *coords* or *atoms* vectors are passed that are of different length than those stored in the instance, a `False` value is returned, rather than an exception raised.

Parameters

- **coords** – length-3N `np.float_` – Vector of stacked 'lab-frame' Cartesian coordinates
- **atoms** – length-N `str` or `int` – Vector of atom symbols or atomic numbers
- **tol** – `float`, optional – Tolerance for acceptable deviation of each passed geometry coordinate from that in the instance to still be considered matching. Default value is *DEF.HESS_COORD_MATCH_TOL*

See *opan.utils.check_geom* for details on return values and exceptions raised.

Subclasses

class opan.hess.**OrcaHess**(*path='...'*)

Container for HESS data generated by *ORCA*.

Initialize by passing the path to the file to be loaded as the *path* keyword argument.

Information contained includes the Hessian matrix, the number of atoms, the atomic symbols, the atomic weights, and the geometry, as reported in the .hess file. See 'Instance Variables' below for a full list. For variables marked "required", a *HessError* is raised if the block is not found, whereas for variables marked "optional" a `None` value is stored. For either type, if the data is malformed or invalid in some fashion, a *HessError* is raised with an appropriate typecode.

Zero frequencies corresponding to translation/rotation are **not** excised from the frequencies list, normal modes, IR spectrum, Raman spectrum, etc.

The precision of the geometry is less than that reported in an .xyz file, and thus should **not** be used for generation of subsequent computations.

Output *units*:

- Hessian : Hartrees per Bohr-squared $\left(\frac{\mathrm{E_h}}{\mathrm{B^2}}\right)$
- Frequencies : Wavenumbers $\left(\frac{\mathrm{cyc}}{\mathrm{cm}}\right)$

- IR intensities (T^2 values) : $\frac{\mathrm{km}}{\mathrm{mol}}$
- Raman activities : $\frac{\mathrm{\AA}^4}{\mathrm{u}}$
- Dipole derivatives : (???)
- Polarizability derivatives : (???)
- Eigenvalues of the mass-weighted Hessian : $\frac{\mathrm{E_h}}{\mathrm{u\,B^2}}$
- Eigenvectors of the mass-weighted Hessian : (dimensionless)

Methods

_load(***kwargs*)
Initialize OrcaHess Hessian object from .hess file

Searches indicated file for data blocks within the .hess file. The geometry, Hessian block, frequencies, and normal modes must be present and will be retrieved; other blocks will be imported if present, or ignored if absent (stored as `None` in the resulting object). If malformed/inaccurate data is found in any block that is present, some flavor of *HessError* will be raised.

Parameters **path** – `str` – *kwargs* parameter specifying complete path to the .hess file to be read.

Raises

- *HessError* – (various typecodes) If indicated Hessian file is malformed in some fashion.
- `KeyError` – If invalid atomic symbol appears in .hess file.
- `IOError` – If the indicated file does not exist or cannot be read.

Class Variables

class Pat
`re.compile()` patterns for data parsing.

at_block
Captures the entire block of atom IDs & weights, as well as the geometry data.

at_line
Retrieves individual lines within the atom / geometry specification block.

dipder_block
Captures the entire dipole derivatives block.

dipder_line
Retrieves individual lines in the dipole derivatives block.

eigvals_block
Captures the entire mass-weighted Hessian eigenvalues block.

eigvals_line
Retrieves individual lines of the mass-weighted hessian eigenvalues block.

eigvecs_block
Captures the entire set of mass-weighted Hessian eigenvectors blocks.

eigvecs_sec
Extracts sections (individual blocks) of the mass-weighted Hessian eigenvectors blocks.

eigvecs_line
Retrieves individual lines of a mass-weighted Hessian eigenvectors block.

energy
Retrieves the energy reported in the .hess file.

freq_block
Captures the entire vibrational frequencies block.

freq_line
Retrieves individual lines of the frequencies block.

hess_block
Captures the entire set of Hessian blocks.

hess_sec
Extracts full-height, 3- or 6-column sections (individual blocks) of the set of Hessian blocks.

hess_line
Retrieves individual lines within a Hessian block.

jobs_block
Captures the entire job list block.

jobs_line
Retrieves individual lines of the job list block.

ir_block
Captures the entire IR spectrum block

ir_line
Retrieves individual lines of the IR spectrum block

modes_block
Captures the entire set of (weighted, column-normalized) normal modes blocks

modes_sec
Extracts full-height, 3- or 6-column sections (individual blocks) of the set of normal modes blocks

modes_line
Retrieves individual lines within a normal modes block

polder_block
Captures the polarizability derivatives block

polder_line
Retrieves individual lines in the polarizability derivatives block

raman_block
Captures the Raman spectrum block

raman_line
Retrieves individual lines in the Raman spectrum block

temp
Retrieves the 'actual temperature' field (sometimes is spuriously zero)

Instance Variables

atom_masses
length-N `list` of `np.float_` – List of atom masses as reported in the .hess file

atom_syms
length-N `list` of `str` – List of uppercase atomic symbols

dipders
3N x 3 `np.float_` – Matrix of dipole derivatives

energy
`float` – Electronic energy reported in the Hessian file

freqs
length-3N `np.float_` – Vibrational frequencies in $\frac{\mathrm{cyc}}{\mathrm{cm}}$, as reported in the Hessian file

geom
length-3N `np.float_` – Geometry vector

hess
3N x 3N `np.float_` – Cartesian Hessian matrix

hess_path
`str` – Complete path/filename from which the Hessian data was retrieved

in_str
`str` – Complete contents of the imported .hess file

ir_comps
3N x 3 `np.float_` – (T_x, T_y, T_z) components of the transition dipole for each normal mode

ir_mags
length-3N `np.float_` – T^2 values (squared-magnitudes) of the transition dipole for each mode

joblist
N x 3 `bool` – Completion status for each displacement in calculation of the Hessian

modes
3N x 3N `np.float_` – Rotation- and translation-purified, mass-weighted vibrational normal modes, with each mode (column vector) separately normalized by *ORCA*.

mwh_eigvals
length-3N `np.float_` – Eigenvalues of the mass-weighted Hessian, in $\frac{\mathrm{E_h}}{\mathrm{u\,B^2}}$

mwh_eigvecs
3N x 3N `np.float_` – Eigenvectors of the mass-weighted Hessian, as column vectors: the eigenvector of eigenvalue i would be retrieved with `mwh_eigvecs[:,i]`

num_ats
`int` – Number of atoms in the system

polders
3N x 6 np.float_ – Matrix of Cartesian polarizability derivatives

raman_acts
length-3N np.float_ – Vector of Raman activities

raman_depols
length-3N np.float_ – Vector of Raman depolarization factors

temp
float – "Actual temperature" reported in the .hess file. Occasionally stored by *ORCA* as a meaningless zero value instead of the temperature used.

2.5 opan.output

Module implementing parsing of output data from external computations.

Warning: This module **will** be refactored at some point, to introduce a superclass in the same vein as *SuperOpanGrad* and *SuperOpanHess*. This is necessary because automated execution of external computation softwares will require some unified mechanism for indicating whether a particular computation completed successfully, independent of the identify of the software package that was executed.

Warning: Further refactoring is also planned for *OrcaOutput* generally. Beware relying heavily on the behavior of this class & module.

Superclass

To be implemented

Implemented Subclasses

Note: Not yet actually a subclass of anything

OrcaOutput – Imports output files from *ORCA*

Subclasses

class opan.output.**OrcaOutput**(*file_path*)
Container for parsed textual output generated by *ORCA*.

All implemented results that are found in the indicated output are stored in the *OrcaOutput* instance. If a given quantity was not detectable, it is stored as None in the corresponding instance variable.

Note: In particular, thermochemistry from single atom/ion computations **should work**, with `None` or zero/negligible values returned for rotational and vibrational quantities.

The verbose contents of the output file are not generally retained within the *OrcaOutput* instance due to the potential for such to involve a tremendously large string. Exceptions include, if present:

- THERMOCHEMISTRY section

Contents

Methods

__init__ (*file_path*)
Initialize *OrcaOutput* object.

Imports the data found in the output file found at *file_path*.

Warning: THIS CLASS PRESENTLY ONLY WORKS ON A **VERY SMALL** SUBSET OF COMPUTATION TYPES, currently HF, LDA-DFT, GGA-DFT, and mGGA-DFT. *MAY* work on some double-hybrid or range-separated DFT.

Available data includes:

- SCF energies (incl D3BJ, gCP, COSMO outlying charge corrections)
- Thermochemistry
- Spin expectation values (actual, ideal, and deviation)

Success indicators include:

- *completed*

Checks for the 'ORCA TERMINATED NORMALLY' report at the end of the file

- *converged*

  Checks for any occurrence of successful SCF convergence in the file (questionable for anything but single-point calculations)

- *optimized*

  Checks for any occurrence of "OPTIMIZATION HAS CONVERGED" in the file (questionable for anything but a standalone OPT – i.e., not useful for a mode or internal coordinate scan)

Parameters **file_path** – str – Full path to the output file to be parsed.

Raises *OutputError* – (various typecodes) If indicated output is un-parseably malformed in some fashion

en_last()

Report the energies from the last SCF present in the output.

Returns a dict providing the various energy values from the last SCF cycle performed in the output. Keys are those of *p_en*. Any energy value not relevant to the parsed output is assigned as None.

Returns *last_ens* – dict of np.float_ – Energies from the last SCF present in the output.

Class Variables

Enumerations

class EN

OpanEnum for the energies reported at the end of SCF cycles.

D3

Grimme's D3BJ dispersion correction [Gri10]. May or may not play nicely with D3ZERO. Likely non-functional with DFT-NL dispersion.

GCP

Grimme's geometric counterpose (gCP) correction [Kru12]

OCC

COSMO outlying charge correction

Todo: Need COSMO reference

SCFFINAL

SCF energy including gCP and D3 corrections

SCFFINALOCC

SCFFINAL energy, but also with COSMO outlying charge correction

SCFOCC
SCF energy with only the COSMO outlying charge correction (no dispersion or gCP corrections)

class SPINCONT
OpanEnum for the spin contamination values reported after each unrestricted SCF cycle.

ACTUAL
Calculated $\langle S^2 \rangle$ expectation value

DEV
Deviation of $\langle S^2 \rangle$ (calculated minus ideal)

IDEAL
Ideal $\langle S^2 \rangle$ expectation value

class THERMO
OpanEnum for the quantities reported in the THERMOCHEMISTRY block.

BLOCK
Entire THERMOCHEMISTRY block (as `str`)

E_EL
Electronic energy from the thermochemistry block, often slightly different than the last *EN.SCFFINAL* value ($\mathrm{E_h}$)

E_ROT
Thermal rotational internal energy correction ($\mathrm{E_h}$)

E_TRANS
Thermal translational internal energy correction ($\mathrm{E_h}$)

E_VIB
Thermal vibrational internal energy correction ($\mathrm{E_h}$)

E_ZPE
Zero-point energy correction ($\mathrm{E_h}$)

H_IG
Ideal-gas ($k_\mathrm{B}T$) enthalpy contribution ($\mathrm{E_h}$)

PRESS
Simulated pressure (atm)

QROT
Rotational partition function (unitless)

TEMP
Simulated temperature (K)

TS_EL
Electronic TS entropy contribution ($\mathrm{E_h}$)

TS_TRANS
Translational TS entropy contribution ($\mathrm{E_h}$)

TS_VIB
Vibrational TS entropy contribution ($\mathrm{E_h}$)

Regular Expression Patterns

p_en
`dict` of `re.compile()` for the energies reported at the end of SCF cycles. Keys are in *EN*.

p_spincont
`dict` of `re.compile()` for the spin contamination block values. Keys are in *SPINCONT*.

p_thermo
`dict` of `re.compile()` for the quantities extracted from the THERMOCHEMISTRY block. Keys are in *THERMO*.

Instance Variables

completed
`bool` – `True` if *ORCA* output reports normal termination, `False` otherwise.

converged
`bool` – `True` if SCF converged ANYWHERE in run.

Todo: Update oo.converged with any robustifications

---

en
`dict` of `list` of `np.float_`– Lists of the various energy values from the parsed output. Dict keys are those of *EN*, above. Any energy type not found in the output is assigned as an empty list.

optimized
`bool` – `True` if any OPT converged ANYWHERE in run. Fine for OPT, but ambiguous for scans.

Todo: Update oo.optimized with any robustifications

---

spincont
`dict` of `list` of `np.float_`– Lists of the various values from the spin contamination calculations in the output, if present. Empty lists if absent. Dict keys are those of *SPINCONT*, above.

src_path
`str` – Full path to the associated output file

thermo
`dict` of `np.float_`– Values from the thermochemistry block of the parsed output. Dict keys are those of *THERMO*, above.

thermo_block
`str` – Full text of the thermochemistry block, if found.

2.6 opan.utils

Utility functions for Open Anharmonic, including execution automation.

Sub-Modules

decorate – Custom Open Anharmonic decorators

execute – Functions for execution of external computational software packages

inertia – Inertia-related tools (center of mass, rotational constants, principal moments/axes, etc.)

symm – Molecular symmetry utility functions (**INCOMPLETE**)

vector – Vector utility functions

Functions

opan.utils.base.**assert_npfloatarray**(*obj*, *varname*, *desc*, *exc*, *tc*, *errsrc*)
Assert a value is an `np.array` of NumPy floats.

Pass `None` to *varname* if *obj* itself is to be checked. Otherwise, *varname* is the string name of the attribute of *obj* to check. In either case, *desc* is a string description of the object to be checked, for use in raising of exceptions.

Raises the exception *exc* with typecode *tc* if the indicated object is determined not to be an `np.array`, with a NumPy float dtype.

Intended primarily to serve as an early check for proper implementation of subclasses of *SuperOpanGrad* and *SuperOpanHess*. Early type-checking of key attributes will hopefully avoid confusing bugs downstream.

Parameters

- **obj** – (arbitrary) – Object to be checked, or object with attribute to be checked.
- **varname** – `str` or `None` – Name of the attribute of *obj* to be type-checked. `None` indicates to check *obj* itself.
- **desc** – `str` – Description of the object being checked to be used in any raised exceptions.
- **exc** – Subclass of *OpanError* to be raised on a failed typecheck.
- **tc** – Typecode of *exc* to be raised on a failed typecheck.
- **errsrc** – `str` – String description of the source of the data leading to a failed typecheck.

opan.utils.base.**check_geom**(*c1*, *a1*, *c2*, *a2*[, *tol*])
Check for consistency of two geometries and atom symbol lists

Cartesian coordinates are considered consistent with the input coords if each component matches to within *tol*. If coords or atoms vectors are passed that are of mismatched lengths, a `False` value is returned.

Both coords vectors must be three times the length of the atoms vectors or a `ValueError` is raised.

Parameters

- **c1** – length-3N `np.float_` – Vector of first set of stacked 'lab-frame' Cartesian coordinates
- **a1** – length-N `str` or `int` – Vector of first set of atom symbols or atomic numbers
- **c2** – length-3N `np.float_` – Vector of second set of stacked 'lab-frame' Cartesian coordinates
- **a2** – length-N `str` or `int` – Vector of second set of atom symbols or atomic numbers

- **tol** – float, optional – Tolerance for acceptable deviation of each geometry coordinate from that in the reference instance to still be considered matching. Default value is specified by *opan.const.DEF.XYZ_COORD_MATCH_TOL*)

Returns

- *match* – bool – Whether input coords and atoms match (True) or not (False)
- *fail_type* – *EnumCheckGeomMismatch* or None – Type of check failure

  If *match* == True:

  Returns as None

  If *match* == False:

  An *EnumCheckGeomMismatch* value indicating the reason for the failed match:

  DIMENSION – Mismatch in geometry size (number of atoms)

  COORDS – Mismatch in one or more coordinates

  ATOMS – Mismatch in one or more atoms
- *fail_loc* – length-3N bool or length-N bool or None – Mismatched elements

  If *match* == True:

  Returns as None

  If *match* == False:

  For "array-level" problems such as a dimension mismatch, a None value is returned.

  For "element-level" problems, a vector is returned indicating positions of mismatch in either *coords* or *atoms*, depending on the value of *fail_type*.

  True elements indicate **MATCHING** values

  False elements mark **MISMATCHES**

Raises ValueError – If a pair of coords & atoms array lengths is inconsistent:

```
if len(c1) != 3 * len(a1) or len(c2) != 3 * len(a2):
    raise ValueError(...)
```

opan.utils.base.**delta_fxn**(*a*, *b*)

Kronecker delta for objects *a* and *b*.

Parameters

- **a** – First object
- **b** – Second object

Returns *delta* – int – Value of Kronecker delta for provided indices, as tested by Python ==

opan.utils.base.**iterable**(*y*)

Check whether or not an object supports iteration.

Adapted directly from NumPy ~= 1.10 at commit 46d2e83.

Parameters **y** – (arbitrary) – Object to be tested.

Returns *test* – bool – Returns False if iter() raises an exception when *y* is passed to it; True otherwise.

Examples

```
>>> opan.utils.iterable([1, 2, 3])
True
>>> opan.utils.iterable(2)
False
```

opan.utils.base.**make_timestamp**(*el_time*)
Generate an hour-minutes-seconds timestamp from an interval in seconds.

Assumes numeric input of a time interval in seconds. Converts this interval to a string of the format "#h #m #s", indicating the number of hours, minutes, and seconds in the interval. Intervals greater than 24h are unproblematic.

Parameters **el_time** – int or float – Time interval in seconds to be converted to h/m/s format

Returns *stamp* – str – String timestamp in #h #m #s format

opan.utils.base.**pack_tups**(**args*)
Pack an arbitrary set of iterables and non-iterables into tuples.

Function packs a set of inputs with arbitrary iterability into tuples. Iterability is tested with *iterable()*. Non-iterable inputs are repeated in each output tuple. Iterable inputs are expanded uniformly across the output tuples. For consistency, all iterables must be the same length.

The input arguments are parsed such that bare strings are treated as **NON-ITERABLE**, through the use of a local subclass of str that cripples the __iter__() method. Any strings passed are returned in the packed tuples as standard, **ITERABLE** instances of str, however.

The order of the input arguments is retained within each output tuple.

No structural conversion is attempted on the arguments.

If all inputs are non-iterable, a list containing a single tuple will be returned.

Parameters ***args** – Arbitrary number of arbitrary mix of iterable and non-iterable objects to be packed into tuples.

Returns *tups* – list of tuple – Number of tuples returned is equal to the length of the iterables passed in **args*

Raises ValueError – If any iterable objects are of different lengths

opan.utils.base.**safe_cast**(*invar*, *totype*)
Performs a "safe" typecast.

Ensures that *invar* properly casts to *totype*. Checks after casting that the result is actually of type *totype*. Any exceptions raised by the typecast itself are unhandled.

Parameters

- **invar** – (arbitrary) – Value to be typecast.
- **totype** – type – Type to which *invar* is to be cast.

Returns *outvar* – *type 'totype'* – Typecast version of *invar*

Raises TypeError – If result of typecast is not of type *totype*

opan.utils.base.**template_subst**(*template, subs[, delims=['<', '>']]*)
Perform substitution of content into tagged string.

For substitutions into template input files for external computational packages, no checks for valid syntax are performed.

Each key in *subs* corresponds to a delimited substitution tag to be replaced in *template* by the entire text of the value of that key. For example, the dict `{"ABC": "text"}` would convert `The <ABC> is working` to `The text is working`, using the default delimiters of '<' and '>'. Substitutions are performed in iteration order from *subs*; recursive substitution as the tag parsing proceeds is thus feasible if an `OrderedDict` is used and substitution key/value pairs are added in the proper order.

Start and end delimiters for the tags are modified by *delims*. For example, to substitute a tag of the form **{|TAG|}**, the tuple `("{|","|}")` should be passed to *subs_delims*. Any elements in *delims* past the second are ignored. No checking is performed for whether the delimiters are "sensible" or not.

Parameters

- **template** – `str` – Template containing tags delimited by *subs_delims*, with tag names and substitution contents provided in *subs*
- **subs** – `dict` of `str` – Each item's key and value are the tag name and corresponding content to be substituted into the provided template.
- **delims** – iterable of `str` – Iterable containing the 'open' and 'close' strings used to mark tags in the template, which are drawn from elements zero and one, respectively. Any elements beyond these are ignored.

Returns *subst_text* – `str` – String generated from the parsed template, with all tag substitutions performed.

2.6.1 opan.utils.decorate

Custom decorators defined for Open Anharmonic.

All of these decorators were built using the class form per the exposition here .

Decorators

class opan.utils.decorate.**arraysqueeze**(**args*)

Converts selected arguments to squeezed np.arrays

Pre-applies an `np.asarray(...).squeeze()` conversion to all positional arguments according to integer indices passed, and to any keyword arguments according to any strings passed.

Each `int` argument passed instructs the decorator to convert the corresponding positional argument in the function definition.

Each `str` argument passed instructs the decorator to convert the corresponding keyword argument.

`str` parameters corresponding to keyword arguments absent in a particular function call and positional/optional argument indices beyond the range of the actual **args* of the decorated function are ignored.

> **Warning:** Likely fragile with optional arguments; needs to be tested.

Parameters ***args** (`int` or `str`) – Arguments to convert to squeezed `np.array`.

class opan.utils.decorate.**kwargfetch**(**args*, ***kwargs*)

Fetch a missing keyword argument with a custom callable & arguments

This decorator implements a form of non-persistent memoization for use in networks of inter-related and/or nested functions, where:

- External users may have reason to call any of the functions directly
- Most or all of the functions call one or more of the same specific "supporting" functions that potentially represent significant computational overhead
- Calls with identical function arguments are not likely to recur in typical use by external users, and thus fully persistent memoization would in general be a waste of memory

The memoization is implemented via injection of a specific keyword argument into a call to the wrapped function, where the inserted value is obtained from a call in turn to a specified callable using arguments drawn from the wrapped call. If the target keyword argument is already present in the wrapped call, no action is taken.

Note: The API description below is wholly non-intuitive and likely impossible to follow. The examples provided in the *User's Guide* will probably be much more illuminating.

Parameters

- **args[0]** – `str` – Name of the keyword argument to be injected into the call to the wrapped function
- **args[1]** – `callable()` – Object to call to generate value to be injected into the target keyword argument (*args[0]*)
- **args[2..n]** – `int` or `str` – Indicate which positional (`int`) and keyword (`str`) parameters of the wrapped function call are to be passed to the `callable()` of *args[1]* as positional parameters, in the order provided within *args[2..n]*
- **kwargs** – `int` or `str` – Indicate which positional (`int`) and keyword (`str`) parameters of the wrapped function call are to be passed to the `callable()` of *args[1]* as keyword parameters, where the keys indicated here in *kwargs* are those used in the call to *args[1]*

2.6.2 opan.utils.execute

Functions to enable Open Anharmonic to execute external software packages.

Functions

`opan.utils.execute.`**`execute_orca`**(*inp_tp*, *work_dir*, *exec_cmd*, *subs=None*, *subs_delims=('<', '>')*, *sim_name='orcarun'*, *inp_ext='txt'*, *out_ext='out'*, *wait_to_complete=True*, *bohrs=False*)

Executes *ORCA* on a dynamically constructed input file.

> **Warning:** Function is still under active development! Execution with *wait_to_complete* == `True` should be robust, however.

Execution

Generates an *ORCA* input file dynamically from information passed into the various arguments, performs the run, and returns with exit info and computation results *(in some fashion; still under development)*. Any required resources (.gbw, .xyz, etc.) MUST already be present in *work_dir*. No check for pre-existing files of the same base name is made; any such will be overwritten.

ORCA MUST be called using a wrapper script; this function does not implement the redirection necessary to send output from a direct *ORCA* call to a file on disk.

If *wait_to_complete* is `True`, the `subprocess.call()` syntax will be used and the function will not return until execution of the wrapper script completes. If `False`, *[indicate what will be returned if not waiting]*.

Todo: execute_orca: The different output modes, depending on waiting or not.

The command to call *ORCA* must be specified in the parameter list syntax of the *args* argument to the `subprocess.Popen` constructor. The implementation is flexible and general, to allow interface with local scripts for, e.g., submission to a job queue in a shared-resource environment.

Valid *ORCA* input syntax of the resulting text is NOT checked before calling *ORCA*.

No mechanism is implemented to detect hangs of *ORCA*. Periodic manual oversight is recommended.

Template Substitution

See *utils.template_subst* for implementation details of the tag substitution mechanism.

Here, in addition to performing any substitutions on the input file template as indicated by *subs*, the special tags **INP** and **OUT**, enclosed with the *subs_delims* delimiters, will be replaced with `sim_name + '.' + inp_ext` and `sim_name + '.' + out_ext`, respectively, in the input template and in all elements of *exec_cmd* before executing the call. In the special case of *inp_ext* == `None`, the **INP** tag will be replaced with just *sim_name* (no extension), and similarly for **OUT** if *out_ext* == `None`. The tag **NAME** will be replaced just with *sim_name* in all cases.

inp_ext and *out_ext* must be different, to avoid collisions.

Return Values

The information returned depends on the value of *wait_to_complete*:

If *wait_to_complete* == `True`:

A `tuple` of objects is returned, with elements of type

```
(OrcaOutput, OpanXYZ, OrcaEngrad, OrcaHess)
```

These objects contain the corresponding results from the computation, if the latter exist, or `None` if they are missing.

If *wait_to_complete* == `False`:

TBD, but current intention is to return the PID of the spawned subprocess.

Signature

Parameters

- **inp_tp** – `str` – Template text for the input file to be generated.
- **work_dir** – `str` – Path to base working directory. Must already exist and contain any resource files (.gbw, .xyz, etc.) required for the calculation.
- **exec_cmd** – `list` of `str` – Sequence of strings defining the *ORCA* execution call in the syntax of the `Popen` constructor. This call must be to a local script; stream redirection of the forked process is not supported in this function.
- **subs** – `dict` of `str`, optional – Substitutions to be performed in the template (see *Template Substitution*, above).
- **subs_delims** – 2-`tuple` of `str`, optional – Tag delimiters passed directly to *template_subst()*. Defaults to `('<','>')`.

- **sim_name** – str, optional – Basename to use for the input/output/working files. If omitted, "orcarun" will be used.
- **inp_ext** – str, optional – Extension to be used for the input file generated (default is 'txt').
- **out_ext** – str, optional – Extension to be used for the output file generated (default is 'out').
- **wait_to_complete** – bool, optional – Whether to wait within this function for *ORCA* execution to complete (True), or to spawn/fork a child process and return (False). Default is True. False **IS NOT YET IMPLEMENTED**.
- **bohrs** – bool, optional – Flag to indicate the units (Bohrs or Angstroms) of the coordinates in .xyz and .trj files.

Returns *[varies]* – tuple of objects or int PID. Varies depending on *wait_to_complete*; see *Return Values* above

Raises

- ValueError – If *inp_ext* and *out_ext* are identical.
- KeyError – If special tag names **INP**, **OUT**, or **NAME** are defined in *subs*
- TypeError – If any elements in *subs* are not tuples

2.6.3 opan.utils.inertia

Utilities for calculation of inertia tensor, principal axes/moments, and rotational constants.

These functions are housed separately from the *opan.vpt2* VPT2 module since they may have broader applicability to other envisioned capabilites of Open Anharmonic.

Functions

opan.utils.inertia.**ctr_geom**(*geom*, *masses*)

Returns geometry shifted to center of mass.

Helper function to automate / encapsulate translation of a geometry to its center of mass.

Parameters

- **geom** – length-3N np.float_ – Original coordinates of the atoms
- **masses** – length-N OR length-3N np.float_ – Atomic masses of the atoms. Length-3N option is to allow calculation of a per-coordinate perturbed value.

Returns *ctr_geom* – length-3N np.float_ – Atomic coordinates after shift to center of mass

Raises ValueError – If shapes of *geom* & *masses* are inconsistent

opan.utils.inertia.**ctr_mass**(*geom*, *masses*)

Calculate the center of mass of the indicated geometry.

Take a geometry and atom masses and compute the location of the center of mass.

Parameters

- **geom** – length-3N np.float_ – Coordinates of the atoms
- **masses** – length-N OR length-3N np.float_ – Atomic masses of the atoms. Length-3N option is to allow calculation of a per-coordinate perturbed value.

Returns *ctr* – length-3 np.float_ – Vector location of center of mass

Raises `ValueError` – If *geom* & *masses* shapes are inconsistent

`opan.utils.inertia.`**`inertia_tensor`**(*geom*, *masses*)

Generate the 3x3 moment-of-inertia tensor.

Compute the 3x3 moment-of-inertia tensor for the provided geometry and atomic masses. Always recenters the geometry to the center of mass as the first step.

Reference for inertia tensor: [Kro92], Eq. (2.26)

Todo: Replace cite eventually with link to exposition in user guide.

Parameters

- **`geom`** – length-3N `np.float_` – Coordinates of the atoms
- **`masses`** – length-N OR length-3N `np.float_` – Atomic masses of the atoms. Length-3N option is to allow calculation of a per-coordinate perturbed value.

Returns *tensor* – 3 x 3 `np.float_` – Moment of inertia tensor for the system

Raises `ValueError` – If shapes of *geom* & *masses* are inconsistent

`opan.utils.inertia.`**`principals`**(*geom*, *masses*[, *on_tol*])

Principal axes and moments of inertia for the indicated geometry.

Calculated by `scipy.linalg.eigh()`, since the moment of inertia tensor is symmetric (real-Hermitian) by construction. More convenient to compute both the axes and moments at the same time since the eigenvectors must be processed to ensure repeatable results.

The principal axes (inertia tensor eigenvectors) are processed in a fashion to ensure repeatable, **identical** generation, including orientation AND directionality.

Todo: Add ref to exposition in webdocs once written up.

Parameters

- **`geom`** – length-3N `np.float_` – Coordinates of the atoms
- **`masses`** – length-N OR length-3N `np.float_` – Atomic masses of the atoms. Length-3N option is to allow calculation of a per-coordinate perturbed value.
- **`on_tol`** – `np.float_`, optional – Tolerance for deviation from unity/zero for principal axis dot products within which axes are considered orthonormal. Default is *`opan.const.DEF.ORTHONORM_TOL`*.

Returns

- *moments* – length-3 `np.float_` – Principal inertial moments, sorted in increasing order ($0 \leq I_A \leq I_B \leq I_C$)
- *axes* – 3 x 3 `np.float_` – Principal axes, as column vectors, sorted with the principal moments and processed for repeatability. The axis corresponding to `moments[i]` is retrieved as `axes[:,i]`
- *top* – *`EnumTopType`* – Detected molecular top type

opan.utils.inertia.**rot_consts**(*geom*, *masses*[, *units*[, *on_tol*]])

Rotational constants for a given molecular system.

Calculates the rotational constants for the provided system with numerical value given in the units provided in *units*. The orthnormality tolerance *on_tol* is required in order to be passed through to the *principals()* function.

If the system is linear or a single atom, the effectively-zero principal moments of inertia will be assigned values of *opan.const.PRM.ZERO_MOMENT_TOL* before transformation into the appropriate rotational constant units.

The moments of inertia are always sorted in increasing order as $0 \leq I_A \leq I_B \leq I_C$; the rotational constants calculated from these will thus always be in **decreasing** order as $B_A \geq B_B \geq B_C$, retaining the ordering and association with the three principal `axes[:,i]` generated by *principals()*.

Parameters

- **geom** – length-3N `np.float_` – Coordinates of the atoms
- **masses** – length-N OR length-3N `np.float_` – Atomic masses of the atoms. Length-3N option is to allow calculation of a per-coordinate perturbed value.
- **units** – *EnumUnitsRotConst*, optional – Enum value indicating the desired units of the output rotational constants. Default is *INV_INERTIA* $\left(\frac{1}{\mathrm{uB^2}}\right)$
- **on_tol** – `np.float_`, optional – Tolerance for deviation from unity/zero for principal axis dot products, within which axes are considered orthonormal. Default is *opan.const.DEF.ORTHONORM_TOL*

Returns *rc* – length-3 `np.float_` – Vector of rotational constants in the indicated units

2.6.4 opan.utils.symm

Utilities submodule for molecular symmetry operations and detection.

Warning: Module is **NON-FUNCTIONAL**

[Assumes molecule has already been translated to center-of-mass.]

[Molecular geometry is a vector, in order of x1, y1, z1, x2, y2, z2, . . .]

[Will need to harmonize the matrix typing; currently things are just passed around as np.array for the most part.]

Todo: Complete symm module docstring, including the member functions

2.6.5 opan.utils.vector

Submodule for miscellaneous vector operations

Functions implemented here are (to the best of this author's knowledge) not available in NumPy or SciPy.

Functions

opan.utils.vector.**ortho_basis**(*normal*[, *ref_vec*])

Generates an orthonormal basis in the plane perpendicular to *normal*

The orthonormal basis generated spans the plane defined with *normal* as its normal vector. The handedness of *on1* and *on2* in the returned basis is such that:

$$\mathsf{on1} \times \mathsf{on2} = \frac{\mathsf{normal}}{\|\mathsf{normal}\|}$$

normal must be expressible as a one-dimensional `np.array` of length 3.

Parameters

- **normal** – length-3 `np.float_` – The orthonormal basis output will span the plane perpendicular to *normal*.
- **ref_vec** – length-3 `np.float_`, optional – If specified, *on1* will be the normalized projection of *ref_vec* onto the plane perpendicular to *normal*. Default is `None`.

Returns

- *on1* – length-3 `np.float_` – First vector defining the orthonormal basis in the plane normal to *normal*
- *on2* – length-3 `np.float_` – Second vector defining the orthonormal basis in the plane normal to *normal*

Raises

- `ValueError` – If *normal* or *ref_vec* is not expressible as a 1-D vector with 3 elements
- *VectorError* – (typecode *NONPRL*) If *ref_vec* is specified and it is insufficiently non-parallel with respect to *normal*

opan.utils.vector.**orthonorm_check**(*a*[, *tol*[, *report*]])

Checks orthonormality of the column vectors of a matrix.

If a one-dimensional `np.array` is passed to *a*, it is treated as a single column vector, rather than a row matrix of length-one column vectors.

The matrix *a* does not need to be square, though it must have at least as many rows as columns, since orthonormality is only possible in N-space with a set of no more than N vectors. (This condition is not directly checked.)

Parameters

- **a** – R x S `np.float_` – 2-D array of column vectors to be checked for orthonormality.
- **tol** – `np.float_`, optional – Tolerance for deviation of dot products from one or zero. Default value is *opan.const.DEF.ORTHONORM_TOL*.
- **report** – `bool`, optional – Whether to record and return vectors / vector pairs failing the orthonormality condition. Default is `False`.

Returns

- *o* – `bool` – Indicates whether column vectors of *a* are orthonormal to within tolerance *tol*.
- *n_fail* – `list` of `int`, or `None` –

  If *report* == `True`:

  A list of indices of column vectors failing the normality condition, or an empty list if all vectors are normalized.

  If *report* == `False`:

  `None`

- *o_fail* – `list` of 2-tuples of `int`, or `None` –

  If *report* == `True`:

  A list of 2-tuples of indices of column vectors failing the orthogonality condition, or an empty list if all vectors are orthogonal.

  If *report* == `False`:

  `None`

opan.utils.vector.**parallel_check**(*vec1*, *vec2*)

Checks whether two vectors are parallel OR anti-parallel.

Vectors must be of the same dimension.

Parameters

- **vec1** – length-R `np.float_` – First vector to compare
- **vec2** – length-R `np.float_` – Second vector to compare

Returns *par* – `bool` – `True` if (anti-)parallel to within *opan.const.PRM.NON_PARALLEL_TOL* degrees. `False` otherwise.

opan.utils.vector.**proj**(*vec*, *vec_onto*)

Vector projection.

Calculated as:

$$\text{vec\_onto} * \frac{\text{vec} \cdot \text{vec\_onto}}{\text{vec\_onto} \cdot \text{vec\_onto}}$$

Parameters

- **vec** – length-R `np.float_` – Vector to project
- **vec_onto** – length-R `np.float_` – Vector onto which *vec* is to be projected

Returns *proj_vec* – length-R `np.float_` – Projection of *vec* onto *vec_onto*

opan.utils.vector.**rej**(*vec*, *vec_onto*)

Vector rejection.

Calculated by subtracting from *vec* the projection of *vec* onto *vec_onto*:

$$\text{vec} - \text{proj}\,(\text{vec},\ \text{vec\_onto})$$

Parameters

- **vec** – length-R `np.float_` – Vector to reject
- **vec_onto** – length-R `np.float_` – Vector onto which *vec* is to be rejected

Returns *rej_vec* – length-R `np.float_` – Rejection of *vec* onto *vec_onto*

opan.utils.vector.**vec_angle**(*vec1*, *vec2*)

Angle between two R-dimensional vectors.

Angle calculated as:

$$\arccos\left[\frac{\text{vec1} \cdot \text{vec2}}{\|\text{vec1}\|\ \|\text{vec2}\|}\right]$$

Parameters

- **vec1** – length-R `np.float_` – First vector

- **vec2** – length-R np.float_ – Second vector

Returns *angle* – np.float_ – Angle between the two vectors in degrees

2.7 opan.vpt2

Submodule implementing VPT2 anharmonic computations.

Warning: This module is under active development. API &c. may change with little notice.

Sub-Modules

repo – HDF5 repository for OpanVPT2

Classes

OpanVPT2 – Core driver class for VPT2 anharmonic calculations.

API

class opan.vpt2.base.**OpanVPT2**
Container for data from VPT2 anharmonic calculations.

To be added...

2.7.1 opan.vpt2.repo

Sub-module for HDF5 repo interactions for VPT2 anharmonic calculations.

Warning: Module is under active development. API &c. may change with little notice.

Classes

class opan.vpt2.repo.**OpanAnharmRepo**(*fname=None*)
HDF5 repo interface for VPT2 anharmonic calculations.

Operations here **DO NOT** ensure consistency with the surrounding data. Such consistency must be checked/handled at a higher level. This is just a wrapper class to facilitate I/O interactions!

Currently no chunking or any filters are used. May or may not be worth robustification. Things stored are not likely to be huge....

Todo: Main docstring for OpanAnharmRepo (renamed to VPT2...)

Instantiation

__init__(...)

Todo: REPO.__init__ docstring

Class Variables

To be documented

Instance Variables

To be documented

Methods

To be documented

2.8 opan.xyz

Module implementing OpenBabel XYZ parsing and interpretation.

The single class `OpanXYZ` imports molecular geometries in the OpenBabel XYZ format , with the following variations:

- Coordinates of any precision will be read, not just the 10.5 specified by OpenBabel
- Both atomic symbols and atomic numbers are valid
- Multiple geometries/frames are supported, but the number of atoms and their sequence in the atoms list must be maintained in all geometries.

Contents

Note: For *angle_iter()* and *dihed_iter()*, using a None parameter with *invalid_error* == True is **guaranteed** to raise an error.

geom_iter() – Geometries

displ_iter() – Displacement vectors

dist_iter() – Euclidean distances

angle_iter() – Span angles

dihed_iter() – Dihedral angles

Class Definition

class opan.xyz.**OpanXYZ**(***kwargs*)

Container for OpenBabel XYZ data.

Initializer can be called in one of two forms:

```
OpanXYZ(path='path/to/file')
OpanXYZ(atom_syms=[{array of atoms}], coords=[{array of coordinates}])
```

If *path* is specified, *atom_syms* and *coords* are ignored, and the instance will contain all validly formatted geometries present in the OpenBabel file at the indicated path.

Note: Certain types of improperly formatted geometry blocks, such as one with alphabetic characters on the 'number-of-atoms' line, may not raise errors during loading, but instead just result in import of fewer geometries/frames than expected.

If initialized with the *atom_syms* and *coords* keyword arguments, the instance will contain the single geometry represented by the provided inputs.

In both forms, the optional keyword argument *bohrs* can be specified, to indicate the units of the coordinates as Bohrs (True) or Angstroms (False). Angstrom and Bohr units are the default for the *path* and *atom_syms*/*coords* forms, respectively. The units of all coordinates stored in the instance are **Bohrs**.

'N' and 'G' in the below documentation refer to the number of atoms per geometry and the number of geometries present in the file, respectively. Note that **ALL** geometries present **MUST** contain the same number of atoms, and the elements must all **FALL IN THE SAME SEQUENCE** in each geometry/frame. No error will be raised if positions of like atoms are swapped, but for obvious reasons this will almost certainly cause semantic difficulties in downstream computations.

Note: In *ORCA* '.xyz' files contain the highest precision geometry information of any output (save perhaps the textual output generated by the program), and are stored in Angstrom units.

Class Variables

p_coords
re.compile() pattern – Retrieves individual lines from coordinate blocks matched by *p_geom*

p_geom
re.compile() pattern – Retrieves full OpenBabel XYZ geometry frames/blocks

LOAD_DATA_FLAG = 'NOT FILE'
str – Flag for irrelevant data in *atom_syms*/*coords* initialization mode.

Instance Variables

Except where indicated, all str and list-of-str values are stored as *LOAD_DATA_FLAG* when initialized with *atom_syms*/*coords* arguments.

atom_syms
length-N str – Atomic symbols for the atoms (all uppercase)

descs
length-G str – Text descriptions for each geometry included in a loaded file

geoms
length-G list of length-3N np.float_ – Molecular geometry/geometries read from file or passed to *coords* argument

in_str
str – Complete contents of the input file

num_atoms
int – Number of atoms per geometry, N

num_geoms
int – Number of geometries, G

XYZ_path
str – Full path to imported OpenBabel file

Methods

geom_single(*g_num*)
Retrieve a single geometry.

The atom coordinates are returned with each atom's x/y/z coordinates grouped together:

```
[A1x, A1y, A1z, A2x, A2y, A2z, ...]
```

An alternate method to achieve the same effect as this function is by simply indexing into *geoms*:

```
>>> x = opan.xyz.OpanXYZ(path='...')
>>> x.geom_single(g_num)     # One way to do it
>>> x.geoms[g_num]           # Another way
```

Parameters **g_num** – int – Index of the desired geometry

Returns *geom* – length-3N np.float_ – Vector of the atomic coordinates for the geometry indicated by *g_num*

Raises IndexError – If an invalid (out-of-range) *g_num* is provided

displ_single(*g_num*, *at_1*, *at_2*)
Displacement vector between two atoms.

Returns the displacement vector pointing from *at_1* toward *at_2* from geometry *g_num*. If *at_1* == *at_2* a strict zero vector is returned.

Displacement vector is returned in units of Bohrs.

Parameters

- **g_num** – int – Index of the desired geometry
- **at_1** – int – Index of the first atom
- **at_2** – int – Index of the second atom

Returns *displ* – length-3 np.float_ – Displacement vector from *at_1* to *at_2*

Raises IndexError – If an invalid (out-of-range) *g_num* or *at_#* is provided

dist_single(*g_num*, *at_1*, *at_2*)
Distance between two atoms.

Parameters

- **g_num** – int – Index of the desired geometry
- **at_1** – int – Index of the first atom
- **at_2** – int – Index of the second atom

Returns *dist* – np.float_ – Distance in Bohrs between *at_1* and *at_2* from geometry *g_num*

Raises IndexError – If an invalid (out-of-range) *g_num* or *at_#* is provided

angle_single(*g_num*, *at_1*, *at_2*, *at_3*)
Spanning angle among three atoms.

The indices *at_1* and *at_3* can be the same (yielding a trivial zero angle), but *at_2* must be different from both *at_1* and *at_3*.

Parameters

- **g_num** – int – Index of the desired geometry
- **at_1** – int – Index of the first atom
- **at_2** – int – Index of the second atom
- **at_3** – int – Index of the third atom

Returns *angle* – np.float_ – Spanning angle in degrees between *at_1*-*at_2*-*at_3*, from geometry *g_num*

Raises

- IndexError – If an invalid (out-of-range) *g_num* or *at_#* is provided
- ValueError – If *at_2* is equal to either *at_1* or *at_3*

dihed_single(*g_num*, *at_1*, *at_2*, *at_3*, *at_4*)
Dihedral/out-of-plane angle among four atoms.

Returns the out-of-plane angle among four atoms from geometry *g_num*, in degrees. The reference plane is spanned by *at_1*, *at_2* and *at_3*. The out-of-plane angle is defined such that a positive angle represents a counter-clockwise rotation of the projected *at_3→at_4* vector with respect to the reference plane when looking from *at_3* toward *at_2*. Zero rotation corresponds to occlusion of *at_1* and *at_4*; that is, the case where the respective rejections of *at_1 →at_2* and *at_3→at_4* onto *at_2→at_3* are ANTI-PARALLEL.

Todo: Pull the above to User Guide eventually, with figures.

All four atom indices must be distinct. Both of the atom trios 1-2-3 and 2-3-4 must be sufficiently nonlinear, as diagnosed by a bend angle different from 0 or 180 degrees by at least *PRM.NON_PARALLEL_TOL*.

Parameters

- **g_num** – int – Index of the desired geometry
- **at_1** – int – Index of the first atom
- **at_2** – int – Index of the second atom
- **at_3** – int – Index of the third atom
- **at_4** – int – Index of the fourth atom

Returns *dihed* – np.float_ – Out-of-plane/dihedral angle in degrees for the indicated *at_#*, drawn from geometry *g_num*

Raises

- IndexError – If an invalid (out-of-range) *g_num* or *at_#* is provided
- ValueError – If any indices *at_#* are equal
- *XYZError* – (typecode *DIHED*) If either of the atom trios (1-2-3 or 2-3-4) is too close to linearity

Generators

geom_iter(*g_nums*)
Iterator over a subset of geometries.

The indices of the geometries to be returned are indicated by an iterable of ints passed as *g_nums*.

As with *geom_single()*, each geometry is returned as a length-3N np.float_ with each atom's x/y/z coordinates grouped together:

```
[A1x, A1y, A1z, A2x, A2y, A2z, ...]
```

In order to use NumPy slicing or advanced indexing, *geoms* must first be explicitly converted to np.array, e.g.:

```
>>> x = opan.xyz.OpanXYZ(path='...')
>>> np.array(x.geoms)[[2,6,9]]
```

Parameters **g_nums** – length-R iterable of int – Indices of the desired geometries

Yields *geom* – length-3N `np.float_` – Vectors of the atomic coordinates for each geometry indicated in *g_nums*

Raises `IndexError` – If an item in *g_nums* is invalid (out of range)

displ_iter(*g_nums*, *ats_1*, *ats_2*)
Iterator over indicated displacement vectors.

Displacements are in Bohrs as with *displ_single()*.

See *above* for more information on calling options.

Parameters

- **g_nums** – `int` or length-R iterable `int` or `None` – Index/indices of the desired geometry/geometries
- **ats_1** – `int` or length-R iterable `int` or `None` – Index/indices of the first atom(s)
- **ats_2** – `int` or length-R iterable `int` or `None` – Index/indices of the second atom(s)
- **invalid_error** – `bool`, optional – If `False` (the default), `None` values are returned for results corresponding to invalid indices. If `True`, exceptions are raised per normal.

Yields *displ* – `np.float_` – Displacement vector in Bohrs between each atom pair of *ats_1* → *ats_2* from the corresponding geometries of *g_nums*.

Raises

- `IndexError` – If an invalid (out-of-range) *g_num* or *at_#* is provided.
- `ValueError` – If all iterable objects are not the same length.

dist_iter(*g_nums*, *ats_1*, *ats_2*)
Iterator over selected interatomic distances.

Distances are in Bohrs as with *dist_single()*.

See *above* for more information on calling options.

Parameters

- **g_nums** – `int` or length-R iterable `int` or `None` – Index/indices of the desired geometry/geometries
- **ats_1** – `int` or iterable `int` or `None` – Index/indices of the first atom(s)
- **ats_2** – `int` or iterable `int` or `None` – Index/indices of the second atom(s)
- **invalid_error** – `bool`, optional – If `False` (the default), `None` values are returned for results corresponding to invalid indices. If `True`, exceptions are raised per normal.

Yields *dist* – `np.float_` – Interatomic distance in Bohrs between each atom pair of *ats_1* and *ats_2* from the corresponding geometries of *g_nums*.

Raises

- `IndexError` – If an invalid (out-of-range) *g_num* or *at_#* is provided.
- `ValueError` – If all iterable objects are not the same length.

angle_iter(*g_nums*, *ats_1*, *ats_2*, *ats_3*)
Iterator over selected atomic angles.

Angles are in degrees as with *angle_single()*.

See *above* for more information on calling options.

Parameters

- **g_nums** – int or iterable int or None – Index of the desired geometry
- **ats_1** – int or iterable int or None – Index of the first atom
- **ats_2** – int or iterable int or None – Index of the second atom
- **ats_3** – int or iterable int or None – Index of the third atom
- **invalid_error** – bool, optional – If False (the default), None values are returned for results corresponding to invalid indices. If True, exceptions are raised per normal.

Yields *angle* – np.float_ – Spanning angles in degrees between corresponding *ats_1*-*ats_2*-*ats_3*, from geometry/geometries *g_nums*

Raises

- IndexError – If an invalid (out-of-range) *g_num* or *at_#* is provided.
- ValueError – If all iterable objects are not the same length.
- ValueError – If any *ats_2* element is equal to either the corresponding *ats_1* or *ats_3* element.

dihed_iter(*g_nums*, *ats_1*, *ats_2*, *ats_3*, *ats_4*)

Iterator over selected dihedral angles.

Angles are in degrees as with *dihed_single()*.

See *above* for more information on calling options.

Parameters

- **g_nums** – int or iterable int or None – Indices of the desired geometry
- **ats_1** – int or iterable int or None – Indices of the first atoms
- **ats_2** – int or iterable int or None – Indices of the second atoms
- **ats_3** – int or iterable int or None – Indices of the third atoms
- **ats_4** – int or iterable int or None – Indices of the fourth atoms
- **invalid_error** – bool, optional – If False (the default), None values are returned for results corresponding to invalid indices. If True, exceptions are raised per normal.

Yields *dihed* – np.float_ – Out-of-plane/dihedral angles in degrees for the indicated atom sets *ats_1*-*ats_2*-*ats_3*-*ats_4*, drawn from the respective *g_nums*.

Raises

- IndexError – If an invalid (out-of-range) *g_num* or *at_#* is provided.
- ValueError – If all iterable objects are not the same length.
- ValueError – If any corresponding *ats_#* indices are equal.
- *XYZError* – (typecode *DIHED*) If either of the atom trios (1-2-3 or 2-3-4) is too close to linearity for any group of *ats_#*

CHAPTER 3

Notational and Algebraic Conventions

In order to aid comprehension, this documentation strives to obey the following formatting/notation conventions.

In text:

- Function/method parameters are set in *italics*.
- When not used for emphasis, **bold** text is used to set apart words/phrases with specific contextual meaning.
- Code snippets are set in fixed font, colored red, and boxed: `[v for v in range(6)]`.
- References to Python objects are set in fixed font, colored black (in most cases), and boxed: *OpanXYZ*. Where practical, they will be linked to the relevant documentation (via `intersphinx`).
- Code examples are set in fixed font and boxed the full width of the page:

```
from opan.utils.vector import proj
pvec = proj(vec1, vec2)
```

  Interactive usage examples are formatted like a Python console session, as per `doctest`:

```
>>> 1 + 2 + 3
6
```

- Where included, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this documentation are to be interpreted as described in **RFC 2119**.
- Mathematical symbols and engineering units are set as in equations, below.

In equations:

- Roman variables are set in serif italics: $x + C$
- Lowercase Greek variables are set in italics: $\frac{\theta+\pi}{\gamma}$
- Uppercase Greek variables are set in upright serif: $\Phi + \Theta$
- Vectors are set in bold, upright, serif Roman symbols: $\mathbf{r}_2 - \mathbf{r}_1$

- Matrices are set as bold, uppercase, serif Roman or Greek symbols: $\mathbf{M}^{-1/2}\mathbf{F}\mathbf{M}^{-1/2}$
- Engineering units are set in upright Roman (serif) equation type: $\left(\frac{\mathrm{E_h}}{\mathrm{B^2}}\right)$

Common symbols:

- N – the number of atoms in the system of interest
- G – the number of geometries in an OpenBabel XYZ file
- $R, S, ...$ – Arbitrary integers

CHAPTER 4

Supported Software Packages

At this development stage, Open Anharmonic is being built with support only for ORCA. Pending community response, support for other packages may be added. In anticipation of such interest, the Open Anharmonic framework is being designed to facilitate extensibility to other computational packages to the extent possible.

Proper installation/configuration of external software packages and construction of suitable input templates, execution scripts, etc. are entirely the responsibility of the user. **It is the responsibility of the user to ascertain whether usage of these software packages in concert with Open Anharmonic is in accord with their respective licenses!!**

Descriptions are quoted from the respective software websites.

ORCA

> The program ORCA is a modern electronic structure program package written by F. Neese, with contributions from many current and former coworkers and several collaborating groups. ... ORCA is a flexible, efficient and easy-to-use general purpose tool for quantum chemistry with specific emphasis on spectroscopic properties of open-shell molecules. It features a wide variety of standard quantum chemical methods ranging from semiempirical methods to DFT to single- and multireference correlated ab initio methods. It can also treat environmental and relativistic effects.

CHAPTER 5

Open Anharmonic Dependencies

Open Anharmonic is compatible only with Python >= 3.4. Open Anharmonic depends on the following Python libraries and has been tested to work on the versions indicated:

- numpy (>=1.10,<1.12)
- scipy (>=0.12,<0.18)
- h5py (>=2.4,<2.7)

Unix/Linux and MacOS users should likely be able to obtain suitable versions of these packages either through `pip` or the respective OS package managers. Windows users should obtain these packages either via anaconda (or a similar prepackaged environment) or from Christoph Gohlke's 'aftermarket' repository.

CHAPTER 6

Units of Measure

6.1 Atomic Units

- B – Bohr radius, atomic unit of length, equal to 0.52917721067 Å or $0.52917721067 \times 10^{-10}$ m – [NISTCUU_B]
- $\mathrm{E_h}$ – Hartree, atomic unit of energy, equal to 4.35974465 J – [NISTCUU_Eh]
- $\mathrm{m_e}$ – Electron mass, atomic unit of mass, equal to $9.10938356 \times 10^{-31}$ kg – [NISTCUU_me]
- $\mathrm{T_a}$ – Atomic time unit, equal to $2.4188843265 \times 10^{-17}$ s – [NISTCUU_Ta]

6.2 Other Units

- cyc – Numerical value of 2π $\left(\frac{\mathrm{cyc}}{2\pi} = 1\right)$, serving to convert angular frequencies into cyclic frequencies.
- u – unified atomic mass unit; equal to 1822.8885 $\mathrm{m_e}$ [NISTCUU_me_u] or $1.660539040 \times 10^{-27}$ kg [NISTCUU_u_kg].

CHAPTER 7

References

From the NIST Reference on Constants, Units and Uncertainty

CHAPTER 8

Documentation To-Do

Todo: Add link to exposition(?) of how RotConst expression is developed, once written.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/const.py:docstring of opan.const.EnumUnitsRotConst, line 9.)

Todo: Add object references once the DOM is more established?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/error.py:docstring of opan.error.AnharmError, line 6.)

Todo: Add note about this being subject to future development?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/error.py:docstring of opan.error.SymmError, line 6.)

Todo: Need COSMO reference

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/output.py:docstring of opan.output.OrcaOutput, line 90.)

Todo: Update oo.converged with any robustifications

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/output.py:docstring of opan.output.OrcaOutput, line 238.)

Todo: Update oo.optimized with any robustifications

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/output.py:docstring of opan.output.OrcaOutput, line 253.)

Todo: execute_orca: The different output modes, depending on waiting or not.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/utils/execute.py:docstring of opan.utils.execute.execute_orca, line 24.)

Todo: Replace cite eventually with link to exposition in user guide.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/utils/inertia.py:docstring of opan.utils.inertia.inertia_tensor, line 9.)

Todo: Add ref to exposition in webdocs once written up.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/utils/inertia.py:docstring of opan.utils.inertia.principals, line 12.)

Todo: Complete symm module docstring, including the member functions

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/utils/symm.py:docstring of opan.utils.symm, line 12.)

Todo: Main docstring for OpanAnharmRepo (renamed to VPT2...)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/vpt2/repo.py:docstring of opan.vpt2.repo.OpanAnharmRepo, line 10.)

Todo: REPO.__init__ docstring

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/vpt2/repo.py:docstring of opan.vpt2.repo.OpanAnharmRepo.__init__, line 1.)

Todo: Pull the above to User Guide eventually, with figures.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/opan/checkouts/latest/opan/xyz.py:docstring of opan.xyz.OpanXYZ.dihed_single, line 18.)

Indices and Tables

- genindex
- modindex
- search

Bibliography

[Gri10] S. Grimme, J. Antony, S. Erlich, H. Krieg. *J Chem Phys* **132**(15): 154104, 2010. doi:10.1063/1.3382344

[Kro92] H.W. Kroto. "Molecular Rotation Spectra." 1st Dover ed. Mineola, NY: Dover Publications, 1992. Google Books

[Kru12] H. Kruse, S. Grimme. *J Chem Phys* **136**(15): 154101, 2012. doi:10.1063/1.3700154

[NISTCUU_B] "Atomic Unit of Length." http://physics.nist.gov/cgi-bin/cuu/Value?tbohrrada0\T1\textbar{}search_for=atomic+length. Accessed 20 Feb 2016.

[NISTCUU_Eh] "Hartree-Joule Relationship." http://physics.nist.gov/cgi-bin/cuu/Value?hrj\T1\textbar{}search_for=hartree. Accessed 20 Feb 2016.

[NISTCUU_me] "Electron Mass." http://physics.nist.gov/cgi-bin/cuu/Value?me\T1\textbar{}search_for=electron+mass. Accessed 20 Feb 2016.

[NISTCUU_me_u] "Electron Mass in u." http://physics.nist.gov/cgi-bin/cuu/Value?meu\T1\textbar{}search_for=u+in+electron+mass. Accessed 25 Jun 2016.

[NISTCUU_Ta] "Atomic Unit of Time." http://physics.nist.gov/cgi-bin/cuu/Value?aut\T1\textbar{}search_for=atomic+time. Accessed 20 Feb 2016.

[NISTCUU_u_kg] "Atomic Mass Unit-Kilogram Relationship." http://physics.nist.gov/cgi-bin/cuu/Value?ukg\T1\textbar{}search_for=atomic+mass. Accessed 25 Jun 2016.

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

P

Q

R

S

T

U

V

W

X

Z