pysisyphus Documentation

Release 0.8.0b1.dev112+g6f77f40

Johannes Steinmetzer

Apr 19, 2024

CONTENTS:

pysisyphus is a software-suite for the exploration of potential energy surfaces in ground- and **excited states**. It implements several methods to search for stationary points (minima and first order saddle points) and the calculation of minimum energy paths by means of IRC and Chain of States methods like Nudged Elastic Band and Growing String. Furthermore it provides tools to easily analyze & modify geometries (aligning, translating, **interpolating**, ...) and to visualize the calculation results/progress.

The required energies, gradients and hessians are calculated by calling external quantum chemistry codes. *pysisyphus* can also be used as a library to implement custom quantum chemistry workflows.

If any issues arise please open an issue and I'll see if it can be fixed and my time permits it. Contrubtions are welcome, so feel free to submit a PR.

This software is still work in progress. Use at your own risk. Also take a look at the license

CHAPTER

ONE

OVERVIEW OF PYSISYPHUS

pysisyphus is a software-suite for the exploration of potential energy surfaces in ground- and **excited states**. User input is read from YAML files, but it can also be used as a python library to set up custom workflows.

Below you can find a screencast of the transition state (TS) search for the famous alanine dipeptide isomerization at the *xtb* level of theory. It starts with pre-optimizations of the initial and final geometry, a subsequent growing string calculation in delocalized internal coordinates to generate a guess for the final TS optimization and with which it concludes.

Besides being able to track excited states over the course of an optimization the idea of *pysisyphus* is to provide tools for handling the whole process of calculating reaction paths, starting from pre-optimizing the given input geometries for chain of states methods and ending with the hessian calculation on the optimized IRC endpoints.

1.1 Entry points

pysisyphus provides several entry points that can be called from the shell (command line). The available commands of the entry points can be queried with the *-h* or *--help* arguments:

```
# Run calculations (Minima optimization, TS search, IRC, NEB, GS, ...)
pysis
# Plotting of path-energies, optimization progress, IRC progress, etc ...
pysisplot
# Manipulate a .trj file or multiple .xyz files
pysistrj
```

1.1.1 pysis

Called with a YAML-input file. Simple and more complex examples can be found in the *examples* directory of this repository. Some common arguments are given below:

```
usage: pysis [-h] [--clean] --cp CP]
         [yaml]

positional arguments:
    yaml                  Start pysisyphus with input from a YAML file.

optional arguments:
    --clean               Ask for confirmation before cleaning.
    --cp CP, --copy CP    Copy .yaml file and corresponding geometries to a new
                          directory. Similar to TURBOMOLEs cpc command.
```

1.1.2 pysisplot

Visualization/plotting of the pysisyphus calculations. Some common arguments are given below:

```
usage: pysisplot (--cosforces | --cosens | --all_energies | --afir | --opt | --irc | --
↪overlaps)

optional arguments:
  --cosforces, --cf    Plot image forces along a COS.
  --cosens, --ce       Plot COS energies.
  --all_energies, -a   Plot ground and excited state energies from 'overlap_data.h5'.
  --afir               Plot AFIR and true -energies and -forces from an AFIR calculation.
  --opt                Plot optimization progress.
  --irc                Plot IRC progress.
  --overlaps, -o
```

1.1.3 pysistrj

Please see *pysistrj --help* for a list of available arguments.

1.2 Available calculators

1.2.1 Excited state capabilities

Program	Gradient	Hessian	Exc. states	Version
ORCA	y	y	TD-DFT, TDA	4.2.x
Turbomole	y	y	TD-DFT, TDA, ricc2	7.2-7.4
Gaussian16	y	y	tested TD-DFT, TDA	
PySCF	y	y	tested TD-DFT	1.7.5
OpenMOLCAS	y	n	&rasscf	Not yet derived from OverlapCalculator

1.2.2 Ground states capabilities

Program	Gradient	Hessian	Version
MOPAC2016	y	y	
XTB	y	y	6.3.2
QCEngine	y	n	>= 0.16.0
Psi4	y	y	

1.2.3 Pure python calculators & Wrappers

Program	Gradient	Hessian	Comment
Sympy 2D	y	y	Many analytical potentials (LEPS, Rosenbrock, Cerjan-Miller, Muller-Brown, ...)
Lennard-Jones	y	n	**No** periodic boundary conditions
AFIR	y	n	
ONIOM	y	n	Arbitrary number of layers with multicenter-support in the highest layer.
FakeASE	y	n	Wraps *pysisyphus* calculators so they can be used with *ase*.

1.3 Available algorithms

1.3.1 Chain Of States Methods

Algorithm	Coordinates	Comment
Nudged Elastic Band (NEB)	Cartesian, DLC planned	Climbing Image variants, Doubly nudged variant
Adaptive NEB	Cartesian	Not well tested
Free-End NEB	Cartesian	Not well tested
Simple Zero-Temperature-String	Cartesian	Equal spacing, energy-dependent spacing
Growing String Method	Cartesian, **DLC**	

1.3.2 Chain Of States Optimizer

Algorithm	Comment	Links
Steepest Descent	Backtracking variant	NEB-Optimizers
Conjugate Gradient	Backtracking variant	NEB-Optimizers
QuickMin		NEB-Optimizers
FIRE		NEB-Optimizers
BFGS		NEB-Optimizers

1.3.3 Transition state optimization

Algorithm	Comment	Links
RS-P-RFO	default	RFO-Paper, RS-Paper
RS-I-RFO		RFO-Paper, RS-Paper
TRIM		TRIM-Paper
Dimer method		

1.3.4 Intrinsic Reaction Coordinate integrators

Algorithm	Comment	Links
Damped-Velocity-Verlet		DVV-Paper
Euler	Not recommended	
EulerPC	default	Kaestner-PC, Euler-PC
Gonzales-Schlegel 2		GS2-Paper
Local Quadratic Approximation		LQA-Paper
Modified IMK		IMK-Paper
Runge-Kutta-4	Not recommended	

1.4 Additional remarks

pysisyphus uses the *tempfile* module from the python stdlib. The location of the temporary directories can be controlled by setting the **$TMPDIR** environment variable before executing *pysis*.

```
export TMPDIR=[tmpdir]
```

CHAPTER

TWO

INSTALLATION

2.1 Preparing an environment

It is good idea to install *pysisyphus* into a separate python environment, whether it is an Anaconda environment or a virtualenv environment, derived from the system python installation or a pyenv installation (preferred).

```
# Optional: Create separate Anaconda environment
conda create -n pysis-env python=3.9
# Activate conda environment
conda activate pysis-env

# Optional: Create separate virtual environment
python -m venv pysis-env
# On some linux distributions 'python3' is used instead of 'python', e.g. Ubuntu
# python3 -m venv pysis-env
# Activate  virtual environment. Preferably, an absolute path pointing to the
# 'activate' script should be used.
source pysis-env/bin/activate
```

2.2 Installation from PyPI with pip

This installs the latest stable release as published on PyPI. If you don't want to do any development work this is the preferred way of installing pysisyphus.

```
python -m pip install pysisyphus
# Installation of additional optional dependencies is also possible
# python -m pip install pysisyphus[test]
```

If you run into any problems please make sure that your pip version is recent enough. A pip upgrade is achieved via:

```
python -m pip install --upgrade pip
```

2.3 Installation from source

Decide on an **$install_dir** and clone the repository from github. If you want to change the code after installation (develop) do an editable (-e) installation with pip.

```
git clone https://github.com/eljost/pysisyphus.git $install_dir
cd $install_dir
# Install with -e if you want an editable installation
python -m pip install [-e] .
# Installation of extras is also possible. 'sphinx' is only needed if you
# want to build the documentation.
# python -m pip install [-e] .[test]
```

With an editable installation it is also easy to use other branches, besides the *master* branch, e.g., the *dev* branch to test out a new feature.

```
git switch dev
```

2.4 Setting up .pysisyphusrc

pysisyphus interfaces several quantum chemistry codes and related software (Multiwfn, Jmol). Software available to *pysisyphus* is registered in **$HOME/.pysisyphusrc**, so *pysisyphus* knows which command to execute an/or where to find the binaries.

Depending on the software different choices were made how it is registered. An example **.pysisyphusrc** is given below, with a short comment for each software.

```
# Excited state calculators

[gaussian09]
# Cmd to execute. Please ensure that g09 is on your $PATH.
cmd=g09
formchk_cmd=formchk
unfchk_cmd=unfchk

[gaussian16]
# Cmds to execute. Please ensure that the binaries are found in your $PATH.
cmd=g16
formchk=formchk
unfchk=unfchk
rwfdump=rwfdump

[openmolcas]
# Cmd to execute. Please ensure that pymolcas is on your $PATH.
cmd=pymolcas

[orca]
# ORCA needs the full path to its binary, so please provide the full path.
cmd=/scratch/programme/orca_4_2_0_linux_x86-64_openmpi314/orca

#[pyscf]
```

(continues on next page)

(continued from previous page)

```
# pyscf must not have an explicit entry in the .pysisyphusrc. pysisyphus uses
# the python API of pyscf, so it is mandatory that is installed in the same environment
# as pysisyphus.

#[turbomole]
# Turbomole must not have an explicit entry in the .pysisyphusrc. The user has to take
# care that everything is set up correctly, e.g. TM-binaries are on the PATH etc...
# The respective commands are hardcoded into pysisyphus (dscf, ridft, ricc2, ...)

# Ground state calculators

[mopac]
# Similar to Psi4. An example is given below.
cmd=/user/johannes/bin/runmopac.sh

[psi4]
# As the Psi4 installation without conda is, to put it slightly, tricky it was
# decided to allow the installation of Psi4 into a separate conda environment.
# pysisyphus then creates a Psi4 input and sends it to the (bash)-script given below
# that accepts/expects one argument. It is the responsibility of the scrip to activate
# the appropriate conda environment and submit the Psi4 input. An example runpsi4.sh
# script is given below.
cmd=/user/johannes/bin/runpsi4.sh

#[qcengine]
# QCEngine must not have an entry explicit entry in the .pysisyphusrc. It is used
# via its python interface and can be installed as an extra with pip (see above).
# The user is referenced to the QCEngine-documentation for any further questions.

[xtb]
# Cmd to execute. Please ensure that xtb is on your $PATH.
cmd=xtb

[openmolcas]
# Make sure that the MOLCAS variable is set.
cmd=pymolcas

[cfour]
# Parallelism is managed by the user by setting environment variables in the runscript,
# as the name of the environment variable for MPI parallelism depends on how CFOUR_MPI_
↪CORES
# was compiled. The GENBAS file must also be symlinked by the runscript.
# A sample is provided below.
cmd=/home/ghjones/bin/runcfour.sh

# Utilities

[wfoverlap]
# Cmd to execute. Please ensure that wfoverlap is on your $PATH. The binary/source
# can be obtained from https://github.com/sharc-md/sharc/tree/master/bin
cmd=/scratch/wfoverlap_1.0/bin/wfoverlap.x
```

(continues on next page)

(continued from previous page)

```
[mwfn]
# Cmd to execute. Please ensure that Multiwfn is on your $PATH. Otherwise put an
# absolute path here. By default pysisyphus looks up "Multiwfn", so if you would
# put a relative path here you don't have to, as this is already covered by the
# defaults.
cmd=Multiwfn

[jmol]
# Cmd to execute. The same arguments apply for jmol as for Multiwfn. "jmol" is
# already the default.
cmd=jmol
```

When the specified path/cmd is not absolute, but relative (e.g. for xtb, g16, ...) the corresponding binaries have to be available on the **$PATH** and all other environment variables have to be set up correctly by the user.

2.5 Example runpsi4.sh

```
#!/bin/bash

# Afaik this doesn't work in non-interactive shells ...
# See https://github.com/conda/conda/issues/8072
# conda activate psi4
source /scratch/programme/anaconda3/bin/activate psi4
#conda activate psi4
psi4 -o stdout $1
```

2.6 Example runmopac.sh

```
#!/bin/bash

module purge
module load mopac

MOPAC2016.exe $1
```

2.7 Example runcfour.sh

```
#!/bin/bash

module purge
module load cfour
export OMP_NUM_THREADS=4
export CFOUR_MPI_CORES=1 # The name of this environment variable depends on how you␣
↪compiled CFOUR
[ ! -f GENBAS ] && ln -s /software/cfour/basis/GENBAS
xcfour > out.log 2>&1
```

2.8 Verifying Installation

By executing `pytest -v --pyargs pysisyphus.tests` a series of quick tests can be executed, verifing successful calculator setup. Running these tests requires *pyscf* and *pytest* to be present (*pip install pyscf pytest*).

CHAPTER

THREE

INSTALLATION VIA NIX

An easy and reliable way to get a complete pysisyphus installation, including additional quantum chemistry (QC) codes, is possible through the Nix package manager. Required Nix files for the QC codes and their dependencies (linear algebra & MPI libraries etc.) are provided by the NixOS-QChem overlay.

Nix installations are fully reproducible and extremely simple to accomplish for any potential pysisyphus user, as most of the necessary configuration is already described in the respective Nix files.

The Nix Pills series provides an informative introduction to the general concepts behind Nix.

Recent stable versions of pysisyphus are also directly included in NixOS-QChem. If you do not require a development version of pysisyphus, using the NixOS-QChem overlay is the preferred option. Otherwise, continue with the instructions below.

3.1 Prerequisites

Please follow the installation instructions in the nix manual to set up a working Nix installation on your system. You may want to enable the `flakes` and `nix-command` features for Nix.

3.2 Caching

The build time can be reduced drastically, if the quantum chemistry codes are fetched from a binary cache.

Pysisyphus provides a binary cache on Cachix.

Nix >= 2.6.0 with flakes directly supports the binary cache without further action required.

Alternatively, if you are allowed to add binary caches in nix, you may simply execute:

```
nix run nixpkgs#cachix use pysisyphus
```

3.3 Usage

You may use pysisyphus directly via Flakes without cloning any repository or local installation. Some examples how to use pysisyphus with nix:

Inspect the structure of pysisyphus' flake:

```
nix flake show github:eljost/pysisyphus
```

Directly run the `pysis` command on an input :

```
nix run github:eljost/pysisyphus input.yml
```

Start an interactive shell with a pysisyphus installation

```
nix shell github:eljost/pysisyphus
```

For developing and hacking pysisyphus and for legacy Nix commands, first lone the pysisyphus repository

```
git clone https://github.com/eljost/pysisyphus.git && cd pysisyphus
```

You may obtain a development shell for pysisyphus via

```
nix develop
```

build pysisyphus in the legacy nix style

```
nix-build ./nix/default.nix
```

or obtain a development legacy-style dev-shell

```
nix-shell ./nix/shell.nix
```

The Flake also offers options to build pysisyphus with proprietary components such as ORCA or Turbomole, e.g.

```
nix run .#pysisyphusOrca
```

or build singularity or docker containers:

```
nix bundle --bundler .#toSingularityImage
```

CHAPTER

FOUR

WORKED EXAMPLE

This page provides a general overview of pysisyphus's capabilities for optimizing ground state reaction paths for the example of a tris-pericyclic reaction between a tropone derivative and dimethylfulvene.

You can find a short discussion of the reaction on Steven M. Bacharachs blog. Here we focus on how to obtain TS1. Instead of using DFT as in the original publication we'll employ the semi-emperical tight binding code xtb. It allows us to run all necessary calculations on common desktop hardware in little time (2 min, using 6 physical CPU cores).

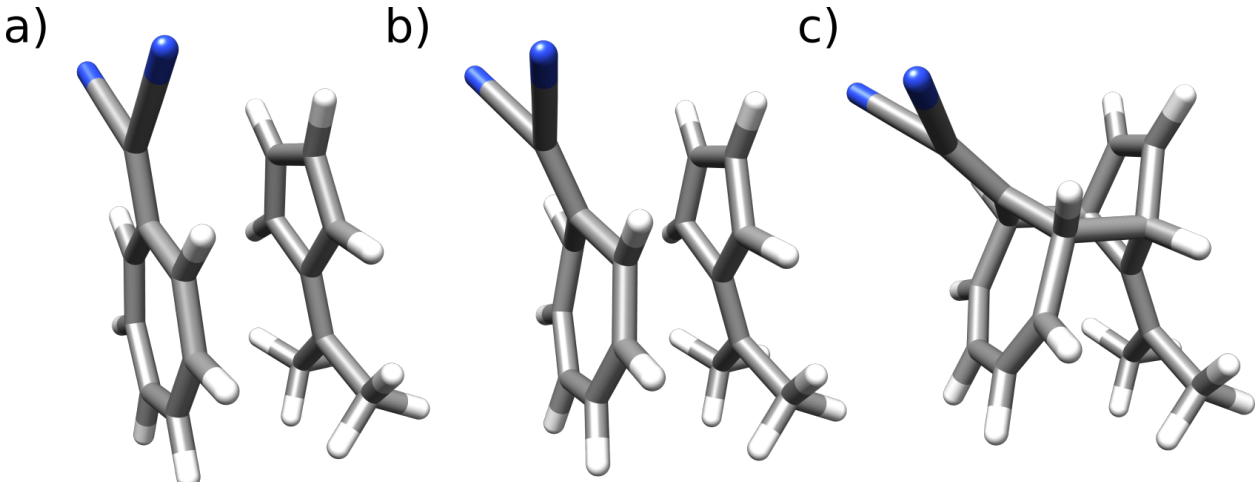

Fig. 4.1: Educts, TS and products of the tris-pericyclic reaction between a tropone derivative and dimethylfulvene.

At first, we have to create appropriate educt and product geometries using the molecular editor of our choice, e.g., avogadro or TmoleX. Given a reasonable educt geometry a suitable product geometry can be constructed by decreasing the distance between the two ring systems and optimizing the result, e.g., by xtb. Sometimes it may be easier to start with the product geometry, and obtain the educt from it. **Consistent atom ordering in both geometries is mandatory!** Usually it's a bad idea to construct both geometries independently, as it is easy to mess up the atom ordering.

Given educts (*min_xtbopt.xyz*) and product (*prod_xtbopt.xyz*) we can create the YAML input. Our goal is to obtain the barrier heights for the reaction shown in Fig. 4.1 by means of growing string (GS) calculation, transition state (TS) optimization, intrinsic reaction coordinate (IRC) calculation and subsequent optimization of the IRC endpoints. The full input is shown below. We'll go over it step by step.

```
geom:
 type: dlc
 fn: [min_xtbopt.xyz, prod_xtbopt.xyz]
preopt:
cos:
 type: gs
```

(continues on next page)

(continued from previous page)

```
 max_nodes: 18
 climb: True
opt:
 type: string
 align: False
 max_cycles: 20
tsopt:
 type: rsirfo
 do_hess: True
 thresh: gau
 hessian_recalc: 3
irc:
 type: eulerpc
 rms_grad_thresh: 0.0005
endopt:
calc:
 type: xtb
 pal: 6
```

The desired coordinate system and the file names of the input geometries are given in the `geom` block.

```
geom:
 type: dlc
 fn: [min_xtbopt.xyz, prod_xtbopt.xyz]
```

Here we chose delocalized internal coordiantes (DLC) for our GS, which is the preferred way. Alternatively Cartesian coordinates could be used by `type: cart`. If DLCs fail, Cartesian coordinates should be tried.

```
preopt:
```

The `preopt` block is given without any additional keywords, so sane defaults will be used for preoptimizing educt and product geometries (Rational function optimization (RFO) in redundant internal coordinates). Strictly, in our case preoptimization is not necessary, as we already preoptimized the geometries using xtb. But in general if is advised to span chain of states (COS) like GS or nudged elastic band (NEB) between stationary points on the potential energy surface. If the educts and products are NOT covalently bound it may be a good idea to restrict the number of preoptimization cycles to a small number (`max_cycles: 10`), as these optimizations are sometimes hard to converge. Please see Optimization of Minima for a list of possible keywords in the `preopt` block.

```
cos:
 type: gs
 max_nodes: 18
 climb: True
```

The `cos` block configures COS calculations. Here we request a GS (`gs`) with 18 nodes (images) between educt and product, resulting in a total string length of 20 nodes. By default first and last node (educt and product) are kept fixed throughout the optimization. We enable a climbing image (CI) to obtain a better TS guess. Please see *Chain Of States Methods* for further information on COS methods.

```
opt:
 type: string
 align: False
 max_cycles: 20
```

COS/GS optimization is controlled via the `opt` block. For GS optimization one should always use `type: string`.

In internal coordinates we disable automated geometry alignment, as it is not needed. We also restrict the number of optimization cycles to 20 (default 50). The chosen optimizer will do steepest descent (SD) steps when the string grew in the previous cycle, otherwise conjugate gradient (CG) steps are used. When the GS is fully grown/connected the optimizer will use limited-memory Broyden-FletcherGoldfarb-Shanno (L-BFGS) to determine more sophisticated steps.

```
tsopt:
 type: rsirfo
 do_hess: True
 thresh: gau
 hessian_recalc: 3
```

After GS convergence the highest energy image (HEI) is determined by cubic splining and used as guess for a classical TS optimization using restricted step image RFO (RSIRFO). `do_hess: True` requests a frequency calculation after the TS optimization. The Hessian is recalculated every 3th step. When the Hessian for the chosen computational method is reasonably cheap it is a good idea to recalculate it periodically. Between recalculations it's updated using the Bofill-update. Convergence critera are tightened from the default `thresh: gau_loose` to `thresh: gau`.

```
irc:
 type: eulerpc
 rms_grad_thresh: 0.0005
```

IRC integration is controlled in the `irc` block. By default the Euler-predictor-corrector (EulerPC) integrator is used. Integration is terminated when the root-mean-square (RMS) of the gradient is equal to or less than 0.0005 au. Possible inputs are given in *Intrinsic Reaction Coordinate (IRC)*.

```
endopt:
```

Similar to `preopt` the `endopt` will be executed with default arguments. It is used to optimize the IRC endpoints to stationary points and enables printing of additional information like RMS deviation of atomic positions (RMSD) between optimized endpoints and initial geometries. The RMSD values help in deciding if the obtained TS actually connects presumed educts and products.

```
calc:
 type: xtb
 pal: 6
```

The `calc` block configures the level of theory used in energy/gradient/Hessian calculations. Here we chose xtb and requested 6 CPU cores. Additional inputs for xtb can be found in the *xtb module documentation*

With everything set up we are ready to actually execute pysisyphus! Assuming the above YAML is saved to *01_pericyclic.yaml* just run

```
pysis 01_pericyclic.yaml | tee pysis.log
```

By default pysisyphus prints to STDOUT so you have to capture STDOUT explicitely. We use `tee` so everything is logged to a file and printed simulatenously. A lot of files and output will be produced so we will go over everything slowly.

```
                         d8b                          888
                         Y8P                          888
                                                      888
88888b.  888  888 .d8888b  888 .d8888b  888  888 88888b.  88888b.  888  888 .d8888b
888 "88b 888  888 88K      888 88K      888  888 888 "88b 888 "88b 888  888 88K
888  888 888  888 "Y8888b. 888 "Y8888b. 888  888 888  888 888  888 888  888 "Y8888b.
```

(continues on next page)

(continued from previous page)

```
888 d88P Y88b 888      X88 888      X88 Y88b 888 888 d88P 888  888 Y88b 888      X88
88888P"   "Y88888  88888P' 888  88888P'  "Y88888 88888P"  888  888  "Y88888  88888P'
888           888                            888 888
888      Y8b d88P                        Y8b d88P 888
888       "Y88P"                          "Y88P"  888

Version 0.5.0.post1+450.g2c1654d3 (Python 3.8.5, NumPy 1.19.2, SciPy 1.5.2)
Git commit 2c1654d35d69e7b48ac4e9b00d38afd58a8bedd4
Executed at Tue Oct  6 10:08:30 2020 on 'your fancy hostname'

If pysisyphus benefitted your research please cite:

    https://doi.org/10.1002/qua.26390

Good luck!
```

You will be greeted by a banner and some information about your current installation, which hopefully aids in reproducing your results later on, if needed. Then your input is repeated, including default values that you did not explicitely set. There you can also see the default values chosen for `preopt` and `endopt`.

```
{'calc': {'pal': 6, 'type': 'xtb'},
 'cos': {'climb': True, 'fix_ends': True, 'max_nodes': 18, 'type': 'gs'},
 'endopt': {'dump': True,
            'fragments': False,
            'max_cycles': 100,
            'overachieve_factor': 3,
            'thresh': 'gau',
            'type': 'rfo'},
 'geom': {'fn': ['min_xtbopt.xyz', 'prod_xtbopt.xyz'], 'type': 'dlc'},
 'interpol': {'between': 0, 'type': None},
 'irc': {'rms_grad_thresh': 0.0005, 'type': 'eulerpc'},
 'opt': {'align': False, 'dump': True, 'max_cycles': 30, 'type': 'string'},
 'preopt': {'dump': True,
            'max_cycles': 100,
            'overachieve_factor': 3,
            'preopt': 'both',
            'strict': False,
            'thresh': 'gau_loose',
            'trust_max': 0.3,
            'type': 'rfo'},
 'tsopt': {'do_hess': True,
           'dump': True,
           'h5_group_name': 'tsopt',
           'hessian_recalc': 3,
           'overachieve_factor': 3,
           'thresh': 'gau',
           'type': 'rsirfo'}}
```

The whole run starts with preoptimizations of educt and product. Both optimizations converge quickly, as the geometries are already preoptimized.

```
                    ###################################
```

(continues on next page)

(continued from previous page)

```
                    # RUNNING FIRST PREOPTIMIZATION #
                    ################################
Spent 0.0 s preparing the first cycle.
cycle   max(force)    rms(force)     max(step)     rms(step)       s/cycle
    0     0.000080     0.000010     0.027005     0.003934          0.1
Converged!

Final summary:
            max(forces, internal): 0.000080 hartree/(bohr,rad)
            rms(forces, internal): 0.000010 hartree/(bohr,rad)
            max(forces,cartesian): 0.000063 hartree/bohr
            rms(forces,cartesian): 0.000015 hartree/bohr
            energy: -52.35197394 hartree
Wrote final, hopefully optimized, geometry to 'first_pre_final_geometry.xyz'

Preoptimization of first geometry converged!
Saved final preoptimized structure to 'first_preopt.xyz'.
RMSD with initial geometry: 0.000000 au

                    ################################
                    # RUNNING LAST PREOPTIMIZATION #
                    ################################
Spent 0.0 s preparing the first cycle.
cycle   max(force)    rms(force)     max(step)     rms(step)       s/cycle
    0     0.000188     0.000025     0.002310     0.000592          0.1
Converged!

Final summary:
            max(forces, internal): 0.000188 hartree/(bohr,rad)
            rms(forces, internal): 0.000025 hartree/(bohr,rad)
            max(forces,cartesian): 0.000182 hartree/bohr
            rms(forces,cartesian): 0.000032 hartree/bohr
            energy: -52.39960456 hartree
Wrote final, hopefully optimized, geometry to 'last_pre_final_geometry.xyz'

Preoptimization of last geometry converged!
Saved final preoptimized structure to 'last_preopt.xyz'.
RMSD with initial geometry: 0.000000 au
```

After an optimization remaining RMS and max of the forces are reported for internal and Cartesian coordinates. If the internal force is zero, but a substantial Cartesian force remains someting went wrong, e.g., the generated coordinate system is lacking important coordinates. In such cases the generated coordinates can be examined manually `pysistrj [geom file] --internals` to determine important missing coordinates.

Preoptimizations are followed by the GS optimization.

```
                        ########################
                        # RUNNING GROWINGSTRING #
                        ########################
Spent 0.0 s preparing the first cycle.
cycle   max(force)    rms(force)     max(step)     rms(step)       s/cycle
    0     0.006403     0.001545     0.006403     0.001545          0.2
    String=2+2 HEI=02/04 (E_max-E_0)=1.4 kJ/mol
```

(continues on next page)

(continued from previous page)

```
    1     0.009367     0.001734     0.009367     0.001734          0.3
    String=3+3 HEI=03/06 (E_max-E_0)=4.1 kJ/mol
    2     0.010394     0.001807     0.010394     0.001807          0.6
    String=4+4 HEI=04/08 (E_max-E_0)=8.0 kJ/mol
    3     0.011152     0.001790     0.011152     0.001790          1.1
    String=5+5 HEI=05/10 (E_max-E_0)=12.5 kJ/mol
    4     0.010687     0.001722     0.010687     0.001722          1.0
    String=6+6 HEI=06/12 (E_max-E_0)=17.6 kJ/mol
    5     0.008751     0.001620     0.008751     0.001620          1.1
    String=7+7 HEI=07/14 (E_max-E_0)=22.8 kJ/mol
    6     0.008007     0.001517     0.008007     0.001517          1.3
    String=8+8 HEI=08/16 (E_max-E_0)=27.6 kJ/mol
    7     0.006895     0.001423     0.006895     0.001423          1.4
    String=9+9 HEI=09/18 (E_max-E_0)=31.3 kJ/mol
Starting to climb in next iteration.
    8     0.006253     0.001313     0.006253     0.001314          1.6
    String=Full HEI=10/20 (E_max-E_0)=32.8 kJ/mol
    9     0.006134     0.001169     0.059438     0.011487          1.7
    String=Full HEI=10/20 (E_max-E_0)=32.0 kJ/mol
   10     0.009555     0.001504     0.022529     0.003827          1.6
    String=Full HEI=10/20 (E_max-E_0)=31.6 kJ/mol
   11     0.009792     0.001583     0.100000     0.011614          1.7
    String=Full HEI=10/20 (E_max-E_0)=31.6 kJ/mol
   12     0.008636     0.001534     0.100000     0.011518          1.7
    String=Full HEI=10/20 (E_max-E_0)=30.7 kJ/mol
   13     0.008708     0.001448     0.100000     0.011463          1.6
    String=Full HEI=10/20 (E_max-E_0)=30.0 kJ/mol
   14     0.008217     0.001363     0.100000     0.005660          1.7
    String=Full HEI=10/20 (E_max-E_0)=29.4 kJ/mol
   15     0.007743     0.001335     0.100000     0.005297          1.7
    String=Full HEI=10/20 (E_max-E_0)=29.1 kJ/mol
   16     0.007243     0.001307     0.100000     0.005143          1.6
    String=Full HEI=10/20 (E_max-E_0)=28.8 kJ/mol
   17     0.006897     0.001285     0.100000     0.004938          1.6
    String=Full HEI=10/20 (E_max-E_0)=28.6 kJ/mol
   18     0.006772     0.001277     0.100000     0.004992          1.6
    String=Full HEI=10/20 (E_max-E_0)=28.5 kJ/mol
   19     0.006714     0.001273     0.100000     0.005006          1.6
    String=Full HEI=10/20 (E_max-E_0)=28.5 kJ/mol
Found sign 'converged'. Ending run.
Operator indicated convergence!
Wrote final, hopefully optimized, geometry to 'final_geometries.trj'
Splined HEI is at 8.07/19.00, between image 8 and 9 (0-based indexing).
Wrote splined HEI to 'splined_hei.xyz'
```

The string grows quickly and is fully grown in cycle 8. String size and barrier height between the first and HEI are reported in every cycle. From cycle 8 on, a CI is employed. The final HEI index is printed at the end. As we interpolate the HEI, the index may be a fractional number. The COS optimization is followed by a TS optimization.

```
                    ####################################
                    # RUNNING TS-OPTIMIZATION FROM COS #
                    ####################################
```

(continues on next page)

(continued from previous page)

```
Creating mixed HEI tangent, using tangents at images (8, 9).
Overlap of splined HEI tangent with these tangents:
    08: 0.982763
    09: 0.149474
Index of splined highest energy image (HEI) is 8.07.

Wrote animated HEI tangent to cart_hei_tangent.trj

Splined HEI (TS guess)
[xyz file printed here; removed for clarity]

Splined Cartesian HEI tangent
[xyz file printed here; removed for clarity]

Wrote splined HEI coordinates to 'splined_hei.xyz'
Calculating Hessian at splined TS guess.
Negative eigenvalues at splined HEI:
[-0.005069 -0.000317]
Overlaps between HEI tangent and imaginary modes:
    00: 0.856746
    01: 0.009826
Imaginary mode 0 has highest overlap with splined HEI tangent.
Spent 0.3 s preparing the first cycle.
cycle   max(force)    rms(force)     max(step)     rms(step)       s/cycle
    0     0.009004     0.001343     0.056873     0.017930          0.2
    1     0.000754     0.000227     0.158373     0.035866          0.2
    2     0.001142     0.000156     0.316150     0.051562          3.6
    3     0.000483     0.000082     0.339089     0.052646          0.2
    4     0.000370     0.000067     0.321142     0.049527          0.2
    5     0.000362     0.000050     0.243173     0.040395          3.6
    6     0.000152     0.000023     0.174987     0.032910          0.2
    7     0.000503     0.000063     0.092082     0.019454          0.2
    8     0.000199     0.000031     0.121285     0.024502          3.5
    9     0.000136     0.000019     0.061690     0.012096          0.2
Converged!

Final summary:
            max(forces, internal): 0.000136 hartree/(bohr,rad)
            rms(forces, internal): 0.000019 hartree/(bohr,rad)
            max(forces,cartesian): 0.000164 hartree/bohr
            rms(forces,cartesian): 0.000043 hartree/bohr
            energy: -52.34823514 hartree
Wrote final, hopefully optimized, geometry to 'ts_final_geometry.xyz'
Optimized TS coords:
[xyz file printed here; removed for clarity]

Wrote TS geometry to 'ts_opt.xyz'

                         ----------------------------
                         | HESSIAN AT FINAL GEOMETRY |
                         ----------------------------
```

(continues on next page)

(continued from previous page)

```
... mass-weighing cartesian hessian
... doing Eckart-projection

First 10 eigenvalues [-2.4303e-03 -3.9622e-17 -8.1858e-18 -7.8671e-19  2.1865e-18  5.
↪1589e-18
  2.6247e-17  1.3733e-05  7.3736e-05  1.2735e-04]
Imaginary frequencies: [-253.24] cm^1

Wrote final (not mass-weighted) hessian to 'calculated_final_cart_hessian'.
Wrote HD5 Hessian to 'final_hessian.h5'.
Wrote imaginary mode with =-253.24 cm^1 to 'imaginary_mode_000.trj'

Barrier between TS and first COS image: 9.8 kJ mol^1
Barrier between TS and last COS image: 134.9 kJ mol^1
```

The initial HEI TS guess features only two sizable imaginary frequencies, confirming that it is a suitable TS guess. Root 0 has the highest overlap (85%) with the HEI tangent and is chosen for maximization in the TS optimization, whereas the energy will be minimized along the remaining modes. The optimization converged quickly in 10 cycles. A final Hessian is computed at the optimized TS as we used `do_hess: True`. Only one imaginary frequency remains, which is the desired result for a first-order saddle point. All sizable imaginary modes are written to *.trj* files and can be viewed by tools like jmol.

Barrier heights between the TS and first and last COS image (no thermochemistry is included!) are reported. In this case the energy difference between the first COS image and the TS is very small, indicating an early TS, similar to the educts. This is also be confirmed by examining Fig. 4.1.

```
                              ###############
                              # RUNNING IRC #
                              ###############
Calculating energy and gradient at the TS
IRC length in mw. coords, max(|grad|) and rms(grad) in unweighted coordinates.
Norm of initial displacement step: 0.1974

                              #################
                              # IRC - FORWARD #
                              #################

        Step   IRC length      dE / au  max(|grad|)    rms(grad)
        ------------------------------------------------------------
           0     0.298147    -0.000333     0.001413     0.000394
        Integrator indicated convergence!

                              ##################
                              # IRC - BACKWARD #
                              ##################

        Step   IRC length      dE / au  max(|grad|)    rms(grad)
        ------------------------------------------------------------
           0     0.332823    -0.000938     0.004607     0.001154
           1     0.662398    -0.001367     0.006116     0.001608
           2     0.991868    -0.001878     0.008670     0.002208
           3     1.320005    -0.002558     0.012581     0.002968
           4     1.644982    -0.003249     0.015719     0.003595
```

(continues on next page)

(continued from previous page)

```
           5     1.966058    -0.003700     0.017023     0.003976
           6     2.285006    -0.003972     0.016598     0.004230
           7     2.596325    -0.004126     0.019112     0.004415
           8     2.898296    -0.004206     0.021118     0.004439
           9     3.186446    -0.004052     0.021182     0.004240
          10     3.462246    -0.003697     0.019162     0.003793
          11     3.723777    -0.003087     0.015042     0.003078
          12     3.966373    -0.002263     0.009324     0.002161
          13     4.173308    -0.001387     0.003822     0.001310
          14     4.321559    -0.000781     0.002226     0.000838
          15     4.420391    -0.000523     0.001616     0.000642
          16     4.498118    -0.000418     0.001299     0.000548
          17     4.571712    -0.000353     0.001967     0.000569
          18     4.647481    -0.000298     0.004516     0.000754
          19     4.722445    -0.000308     0.003469     0.000670
          20     4.808189    -0.000290     0.003426     0.000682
          21     4.881593    -0.000209     0.004155     0.000683
       Integrator indicated convergence!
```

The imaginary mode is used to displace the TS towards educts and product. As the TS is very similar to the educt, forward IRC integration already terminates after one cycle. Maybe, further integration steps could be forced by tightening the threshold in the `irc:` block. Backward integration terminates after 22 cycles. At first, the gradient increases and after the inflection point is passed, falls off again as a stationary point is approached. In the end both IRC endpoints are fully optimized to stationary points.

```
                    ####################################
                    # RUNNING FORWARD_END OPTIMIZATION #
                    ####################################
Spent 0.0 s preparing the first cycle.
cycle   max(force)    rms(force)     max(step)     rms(step)       s/cycle
    0     0.000491     0.000126     0.099615     0.030780          0.1
    1     0.001769     0.000333     0.121277     0.033041          0.2
    2     0.002986     0.000525     0.228218     0.054622          0.2
    3     0.002455     0.000475     0.396872     0.096049          0.2
    4     0.003106     0.000502     0.061397     0.012734          0.2
    5     0.000676     0.000187     0.040565     0.007988          0.2
    6     0.000498     0.000144     0.048609     0.012294          0.2
    7     0.000908     0.000142     0.055895     0.016806          0.2
    8     0.000715     0.000102     0.037441     0.011352          0.2
    9     0.000226     0.000047     0.034467     0.006157          0.2
   10     0.000119     0.000032     0.034646     0.006157          0.1
Converged!

Final summary:
            max(forces, internal): 0.000119 hartree/(bohr,rad)
            rms(forces, internal): 0.000032 hartree/(bohr,rad)
            max(forces,cartesian): 0.000241 hartree/bohr
            rms(forces,cartesian): 0.000073 hartree/bohr
            energy: -52.35194671 hartree
Wrote final, hopefully optimized, geometry to 'forward_end_final_geometry.xyz'

Moved 'forward_end_final_geometry.xyz' to 'forward_end_opt.xyz'.
```

(continues on next page)

(continued from previous page)

```
                    ####################################
                    # RUNNING BACKWARD_END OPTIMIZATION #
                    ####################################
Spent 0.0 s preparing the first cycle.
cycle   max(force)    rms(force)     max(step)     rms(step)       s/cycle
    0     0.004485     0.001075     0.168005     0.031009          0.1
    1     0.003805     0.000731     0.158516     0.028598          0.2
    2     0.003345     0.000610     0.047413     0.008626          0.2
    3     0.002849     0.000531     0.093820     0.017252          0.2
    4     0.001571     0.000332     0.154150     0.028740          0.2
    5     0.000752     0.000148     0.034367     0.007031          0.2
    6     0.000630     0.000107     0.017362     0.004137          0.2
    7     0.000218     0.000047     0.019096     0.003624          0.1
    8     0.000165     0.000041     0.027156     0.004814          0.2
    9     0.000400     0.000065     0.044195     0.007618          0.2
   10     0.000460     0.000076     0.039724     0.006652          0.1
   11     0.000265     0.000059     0.021454     0.003444          0.2
   12     0.000194     0.000034     0.009649     0.001594          0.1
   13     0.000146     0.000025     0.009711     0.001835          0.1
Converged!

Final summary:
            max(forces, internal): 0.000146 hartree/(bohr,rad)
            rms(forces, internal): 0.000025 hartree/(bohr,rad)
            max(forces,cartesian): 0.000183 hartree/bohr
            rms(forces,cartesian): 0.000047 hartree/bohr
            energy: -52.39959356 hartree
Wrote final, hopefully optimized, geometry to 'backward_end_final_geometry.xyz'

Moved 'backward_end_final_geometry.xyz' to 'backward_end_opt.xyz'.
```

Both optimizations converge quickly without any problems. Finally, optimized endpoint geometries are compared to the intial geometries and final barrier heights are reported. Thermochemistry is not (yet) included. Even though both endpoints are reported as dissimilar to the initial geomtries they are still very similar, confirming, that the obtained TS indeed connects presumed educts and product.

```
                    ###################################
                    # RMSDS AFTER END OPTIMIZATIONS #
                    ###################################
start geom  0 (first_preopt.xyz/min_xtbopt.xyz)
    end geom  0 ( forward_end_opt.xyz): RMSD=0.133727 au
    end geom  1 (backward_end_opt.xyz): RMSD=1.126059 au
    Optimized end geometries are dissimilar to 'first_preopt.xyz/min_xtbopt.xyz'!
start geom  1 (last_preopt.xyz/prod_xtbopt.xyz)
    end geom  0 ( forward_end_opt.xyz): RMSD=1.144102 au
    end geom  1 (backward_end_opt.xyz): RMSD=0.093044 au
    Optimized end geometries are dissimilar to 'last_preopt.xyz/prod_xtbopt.xyz'!

                ##########################################
                # BARRIER HEIGHTS AFTER END OPTIMIZATIONS #
                ##########################################
```

(continues on next page)

(continued from previous page)

```
Thermochemical corrections are NOT included!

Minimum energy of 0.0 kJ mol¹ at 'backward_end_opt.xyz'.

     forward_end_opt.xyz:   125.10 kJ mol¹
                      TS:   134.84 kJ mol¹
    backward_end_opt.xyz:     0.00 kJ mol¹

Wrote optimized end-geometries and TS to 'end_geoms_and_ts.trj'
```

Visualization/plotting of running optimizations and IRC integrations is also possible. Please see *Plotting* for further information.

All inputs can be found in the *examples/complex/08_trispericyclic* directry on github.

CHAPTER

FIVE

COORDINATE SYSTEMS

Choice of coordinate system is often crucial, for the successful outcome of geometry optimizations. Pysisyphus supports different coordinate systems. By default, molecular optimizations are carried out in redundant internal coordinates (RIC).

5.1 Supported Coordinate Systems

5.1.1 Cartesian Coordinates

- Strong coupling
- Unambiguously defined, if translation and rotation (TR) are removed
- Redundant set, if TR are not removed
- `type: cart`

5.1.2 Redundant Internal Coordinates

- Comprising bond stretches, (linear) bends, (improper) torsions and other primitives
- Less coupling, compared to Cartesians
- Easy estimation of diagonal model Hessians
- **Support constraints**
- Require sophisticated setup algorithm
- Iterative internal-Cartesian backtransformation, which may fail
- Usually highly redundant set
- `type: redund`

5.1.3 Delocalized Internal Coordinates (DLC)

- Complicated linear combinations of primitive internals
- No coupling, coordinates are orthogonal to each other, at least at the geometrey they were defined at
- Non redundant set
- More efficient compared to RIC for bigger systems (if initial DLC generation is feasible)
- Same comments apply, as for RICs
- `type: dlc`

5.1.4 Translation & Rotation Internal Coordinates (TRIC)

- Especially suited to optimize solvated/non-covalently bound systems
- Translation and rotation coordinates are assigned to every fragment
- Avoids error-prone assignment of interfragment coordinates
- See 10.1063/1.4952956 for a full discussion
- By default the B-Matrix is recalculated in every step of the internal-Cartesian backtransformation when TRIC is enabled
- `type: tric`

5.2 Supported File Formats

All formats can be read, at least to a certain extent. Coordinate files are expected to be in , if not otherwise dictated by the respective format.

Suffix	Write	Comment
.xyz	✓	Plain XYZ, single geometry.
.trj	✓	Plain XYZ, multiple geometries.
.pdb	✓	Protein Data Bank.
.molden		Restricted to [Geometries] block.
.zmat		Z-Matrix, see below for an example.
.cjson		As saved by Avogadro.
.crd		CHARMM card format
.sdf		Structure-data file

5.2.1 Z-Matrix example

```
C
N 1 1.35
H 1 1.0 2 105.0
H 2 1.4 1 105.0 3 150.0
H 2 1.4 1 110.0 3 -160.0
```

Indexing in Z-matrices is 1-based, so the first atom has index 1. Variable substitution is not supported in the current Z-matrix parser.

5.3 YAML Input

See below for an explanation of possible inputs in the *geom* section. Input related to internal coordinates is given mostly in the *coord_kwargs* subgroup, whereas atom-related input (*isotops*, *freeze_atoms*) is given one level above. See the example below:

```
geom:
 type: cart                     # Coordinate system (cart/redund/dlc/tric)
 fn: [input]                    # File name or inline input
 union: False                   # Define same set of primitives at multiple geometries
 isotopes: null                 # Specify different isotopes
 freeze_atoms: null             # Freeze Cartesians of certain atoms
 coord_kwargs:                  # Keywords that are passed to the internal coordinate␣
↪class
  define_prims: null            # Additionally define these primitives
  constrain_prims: null         # Primitive internals to be constrained
  freeze_atoms_exclude: False   # Whether to set up internal coordinates for frozen atoms
preopt:
 geom:                          # geom block in preopt takes same keywords as above
  ...                           # no 'fn' key here!
endopt:
 geom:                          # geom block in endopt takes same keywords as above
  ...                           # no 'fn' key here!
```

Employed coordinates and coordinate systems in *preopt* and *endopt* are similary controlled by a *geom* block. Same keywords are supported, as for the *geom* block, at the top level, except the *fn* key.

5.3.1 Inline input

Inline coordinates are supported for XYZ format, and expected in . Take care of proper indentation. The example below would yield RIC for the hydrogen molecule (1 bond).

```
geom:
 type: redund
 fn: |
  2

  H 0.0 0.0 0.0
  H 0.0 0.0 0.7
```

5.3.2 Types of Primitive Coordinates

Pysisyphus implements many different (primitive) internal coordinates. Every coordinate is defined by its type and a set of atom indices, e.g., 2 indices for a bond, 3 indices for a bend and 4 indices for a dihedral.

Specification of a type is necessary, as there are many different kinds of bonds, bends and dihedrals/out-of-plane. One can't just assume, that a coordinate comprised of 3 atom indices is always a regular bend, as it may also be a linear bend or a translational coordinate (TRANSLATION_X, 14), describin the mean Cartesian X coordinate of 3 atoms.

Atom indices start at 0!

```
# Primitive types
BOND = 0
AUX_BOND = 1
HYDROGEN_BOND = 2
INTERFRAG_BOND = 3
AUX_INTERFRAG_BOND = 4
BEND = 5
LINEAR_BEND = 6
LINEAR_BEND_COMPLEMENT = 7
PROPER_DIHEDRAL = 8
IMPROPER_DIHEDRAL = 9
OUT_OF_PLANE = 10
LINEAR_DISPLACEMENT = 11
LINEAR_DISPLACEMENT_COMPLEMENT = 12
# TRANSLATION = 13  # Dummy coordinate
TRANSLATION_X = 14
TRANSLATION_Y = 15
TRANSLATION_Z = 16
# ROTATION = 17  # Dummy coordinate
ROTATION_A = 18
ROTATION_B = 19
ROTATION_C = 20
# CARTESIAN = 21  # Dummy coordinate
CARTESIAN_X = 22
CARTESIAN_Y = 23
CARTESIAN_Z = 24
BONDED_FRAGMENT = 25
DUMMY_TORSION = 26
DISTANCE_FUNCTION = 27
# atan2 based coordinates
BEND2 = 28
TORSION2 = 29
```

As some of these types are quite unwieldy, several shortcuts are supported, that can be used in place of the types above.

```
# Additional shortcuts
# Using Cartesians in the framework of internal coordinates is mainly
# useful if one wants to constrain certain atoms.
"X": [PT.CARTESIAN_X],
"Y": [PT.CARTESIAN_Y],
"Z": [PT.CARTESIAN_Z],
"XY": [PT.CARTESIAN_X, PT.CARTESIAN_Y],
"XZ": [PT.CARTESIAN_X, PT.CARTESIAN_Z],
"YZ": [PT.CARTESIAN_Y, PT.CARTESIAN_Z],
"XYZ": [PT.CARTESIAN_X, PT.CARTESIAN_Y, PT.CARTESIAN_Z],
"ATOM": [PT.CARTESIAN_X, PT.CARTESIAN_Y, PT.CARTESIAN_Z],
# Primitive aliases
"B": [PT.BOND],
"A": [PT.BEND],
"A2": [PT.BEND2],
"D": [PT.PROPER_DIHEDRAL],
"DIHEDRAL": [PT.PROPER_DIHEDRAL],
"TORSION": [PT.PROPER_DIHEDRAL],
```

(continues on next page)

(continued from previous page)

```
"D2": [PT.PROPER_DIHEDRAL2],
"DIHEDRAL2": [PT.PROPER_DIHEDRAL2],
"TORSION2": [PT.PROPER_DIHEDRAL2],
# Translation & Rotation coordinates
"TRANSLATION": [PT.TRANSLATION_X, PT.TRANSLATION_Y, PT.TRANSLATION_Z],
"ROTATION": [PT.ROTATION_A, PT.ROTATION_B, PT.ROTATION_C],
# Miscellaneous
"DIST_FUNC": [PT.DISTANCE_FUNCTION]
```

5.3.3 Define Additional Primitives

Pysisyphus tries its best, to automatically come up with a reasonable set of internal coordinates, but sometimes the algorithm misses an important one. Especially at transition state guesses, where increased atom distances are common, bonds may be missed.

In such cases, additional coordinates can be requested explicitly. If additional coordinates are requested, **a nested list is expected [[coord0], [coord1], ...]**.

```
# General structure (list of coordinate lists)
define_prims: [[PrimType or Shortcut], *[atom indices], ...]

# Examples

# Additional bond between atoms 4 and 7 (0-based indexing).
# All three lines below result in the same bond; the latter two use shortcuts.
define_prims: [[0, 4, 7]]
define_prims: [[B, 4, 7]]
define_prims: [[BOND, 4, 7]]

# Wrong specification (forgot outer list/brackets):
define_prims: [0, 4, 7]

# Also define an additional dihedral, beside the bond
define_prims: [[0, 4, 7], ["D", 0, 1, 2, 3]]
```

5.3.4 Freeze Atoms

All three Cartesian coordinates (X, Y, Z) of certain atoms can be frozen, so they always remain at their initial value. By setting *freeze_atoms_exclude* in *coord_kwargs*, frozen atoms can be excluded from the internal coordinate setup. By default frozen atoms are included.

```
geom:
 type: [type]
 fn: [fn]
 freeze_atoms: [*atom indices]
 coord_kwargs:
  freeze_atoms_exclude: False

# Example; fully freeze Cartesians of first and second atom.
geom:
```

(continues on next page)

(continued from previous page)

```
type: cart
fn: input.xyz
freeze_atoms: [0, 1]
coord_kwargs:
 freeze_atoms: True
```

5.3.5 Constraints

Constraints beyond frozen atoms are currently only supported in conjunction with RIC (`type: redund`). It is not (yet) possible to modify the value of the specified coordinate via YAML input; the internal coordinate is constrained at its initial value. The same syntax as for *define_prims* is used. If the coordinate of the requested constraint is not already defined, it will be defined subsequently. There is no need to also add the constrained coordinate to *define_prims*.

```
# General structure (nested list of coordinates)
constrain_prims: [[[PrimType or Shortcut], *[atom indices]], ...]

# Examples

# Constrain Cartesian coordinate of atom 0.
# Both lines result in the same constraint.
constrain_prims: [[XYZ, 0]]
constrain_prims: [[ATOM, 0]]

# Constrain only Cartesian X and Y component of atom 0.
constrain_prims: [[XY, 0]]

# Constraint bond between atoms 4 and 7 (0-based indexing).
# All three lines below result in the same constraint; the latter two use shortcuts.
constrain_prims: [[0, 4, 7]]
constrain_prims: [[B, 4, 7]]
constrain_prims: [[BOND, 4, 7]]
```

Constraining the Cartesian coordinates (X, Y and Z) of one atom does not affect the final energy of an optimization. **But constraining more than one atome does.**

Harmonic restraints to selected primitive internals can be specified in the *calc:* section (see the *Restraint* documentation).

5.3.6 Isotopes

Different isotope masses can be requested. The system works similar to Gaussians system. A list of pairs is expected, where the first number specifies the atom and the second number is either an integer or a float. If it is an integer, the isotope mass closest to this integer is looked up in an internal database. Floats are used as is.

```
# General structure (nested list of coordinates)
isotopes: [[[atom index], [new mass, integer/float], ...]

# Modify the mass of atom with index 2 (hydrogen in this case)
# Both lines give identical results (deuterium).
# In the second line, the mass is given directly.
```

(continues on next page)

(continued from previous page)

```
isotopes: [[2, 2]]
isotopes: [[2, 2.014101778]]
```

Different isotope masses affect calculated frequencies and IRCs. Atoms can be fixed in IRC calculations by specifying a very high mass. **YAML does not recognize 1e9 as float**, take care to add a dot (**1.e9**).

```
# Fix atom 0 in IRC calculation.
isotopes: [[0, 1.e9]]
```

5.4 Geometry & RedundantCoords

The central class in pysisyphus, handling coordinates and delegating calculations to (external QC) codes, is the *Geometry* class, similar to ASE's *Atoms*.

class pysisyphus.Geometry.**Geometry**(*atoms*, *coords*, *fragments=None*, *coord_type='cart'*, *coord_kwargs=None*, *isotopes=None*, *freeze_atoms=None*, *comment=''*, *name=''*)

__init__(*atoms*, *coords*, *fragments=None*, *coord_type='cart'*, *coord_kwargs=None*, *isotopes=None*, *freeze_atoms=None*, *comment=''*, *name=''*)

Object representing atoms in a coordinate system.

The Geometry represents atoms and their positions in coordinate system. By default cartesian coordinates are used, but internal coordinates are also possible.

Parameters

- **atoms** (*iterable*) -- Iterable of length N, containing element symbols.
- **coords** (*1d iterable*) -- 1d iterable of length 3N, containing the cartesian coordinates of N atoms.
- **fragments** (*dict, optional*) -- Dict with different keys denoting different fragments. The values contain lists of atom indices.
- **coord_type** (*{"cart", "redund"}, optional*) -- Type of coordinate system to use. Right now cartesian (cart) and redundand (redund) are supported.
- **coord_kwargs** (*dict, optional*) -- Dictionary containing additional arguments that get passed to the constructor of the internal coordinate class.
- **isotopes** (*iterable of pairs, optional*) -- Iterable of pairs consisting of 0-based atom index and either an integer or a float. If an integer is given the closest isotope mass will be selected. Given a float, this float will be directly used as mass.
- **freeze_atoms** (*iterable of integers*) -- Specifies which atoms should remain fixed at their initial positions.
- **comment** (*str, optional*) -- Comment string.
- **name** (*str, optional*) -- Verbose name of the geometry, e.g. methanal or water. Used for printing

align_principal_axes()

Align the principal axes to the cartesian axes.

https://math.stackexchange.com/questions/145023

property all_energies

Return energies of all states that were calculated.

This will also set self.energy, which may NOT be the ground state, but the state correspondig to the 'root' attribute of the calculator.

approximate_radius()

Approximate molecule radius from the biggest atom distance along an axis.

as_ase_atoms()

as_g98_list()

Returns data for fake Gaussian98 standard orientation output.

Returns
g98_list -- List with one row per atom. Every row contains [center number, atomic number, atomic type (always 0 for now), X Y Z coordinates in Angstrom.

Return type
list

as_xyz(*comment=''*, *atoms=None*, *cart_coords=None*)

Current geometry as a string in XYZ-format.

Parameters

- **comment** (*str, optional*) -- Will be written in the second line (comment line) of the XYZ-string.
- **cart_coords** (*np.array, 1d, shape (3 * atoms.size,)*) -- Cartesians for dumping instead of self._coords.

Returns
xyz_str -- Current geometry as string in XYZ-format.

Return type
str

assert_cart_coords(*coords*)

assert_compatibility(*other*)

Assert that two Geometries can be substracted from each other.

Parameters
other (Geometry) -- Geometry for comparison.

atom_indices()

Dict with atom types as key and corresponding indices as values.

Returns
inds_dict -- Unique atom types as keys, corresponding indices as values.

Return type
dict

property atom_types

atom_xyz_iter()

property atomic_numbers

property bond_sets

calc_double_ao_overlap(*geom2*)

calc_energy_and_forces()

Force a calculation of the current energy and forces.

property cart_coords

property cart_forces

property cart_gradient

property cart_hessian

center()

property center_of_mass

Returns the center of mass.

Returns
R -- Center of mass.

Return type
np.array, shape(3,)

center_of_mass_at(*coords3d*)

Returns the center of mass at given coords3d.

Parameters
coords3d (*np.array, shape(N, 3)*) -- Cartesian coordiantes.

Returns
R -- Center of mass.

Return type
np.array, shape(3,)

property centroid

Geometric center of the Geometry.

Returns
R -- Geometric center of the Geometry.

Return type
np.array, shape(3,)

clear()

Reset the object state.

property comment

```
coord_types = {'cart': None, 'cartesian': <class
'pysisyphus.intcoords.CartesianCoords.CartesianCoords'>, 'dlc': <class
'pysisyphus.intcoords.DLC.DLC'>, 'hdlc': <class 'pysisyphus.intcoords.DLC.HDLC'>,
'hredund': <class 'pysisyphus.intcoords.RedundantCoords.HybridRedundantCoords'>,
'mwcartesian': <class 'pysisyphus.intcoords.CartesianCoords.MWCartesianCoords'>,
'redund': <class 'pysisyphus.intcoords.RedundantCoords.RedundantCoords'>, 'tmtric':
<class 'pysisyphus.intcoords.RedundantCoords.TMTRIC'>, 'tric': <class
'pysisyphus.intcoords.RedundantCoords.TRIC'>}
```

property coords

1d vector of atomic coordinates.

Returns
coords -- 1d array holding the current coordinates.

Return type
np.array

property coords3d

Coordinates in 3d.

Returns
coords3d -- Coordinates of the Geometry as 2D array.

Return type
np.array

property coords_by_type

Coordinates in 3d by atom type and their corresponding indices.

Returns

- **cbt** (*dict*) -- Dictionary with the unique atom types of the Geometry as keys. It's values are the 3d coordinates of the corresponding atom type.
- **inds** (*dict*) -- Dictionary with the unique atom types of the Geometry as keys. It's values are the original indices of the 3d coordinates in the whole coords3d array.

copy(*coord_type=None*, *coord_kwargs=None*)

Returns a new Geometry object with same atoms and coordinates.

Parameters

- **coord_type** (*str*) -- Desired coord_type, defaults to current coord_type.
- **coord_kwargs** (*dict, optional*) -- Any desired coord_kwargs that will be passed to the RedundantCoords object.

Returns
geom -- New Geometry object with the same atoms and coordinates.

Return type
Geometry

copy_all(*coord_type=None*, *coord_kwargs=None*)

property covalent_radii

del_atoms(*inds*, ***kwargs*)

describe()

dump_xyz(*fn*, *cart_coords=None*, ***kwargs*)

eckart_projection(*mw_hessian*, *return_P=False*, *full=False*)

property energy

Energy of the current atomic configuration.

Returns
energy -- Energy of the current atomic configuration.

Return type
float

fd_coords3d_gen(*step_size=0.001*)

Iterator returning 3d Cartesians for finite-differences.

property forces

Energy of the current atomic configuration.

Returns
force -- 1d array containing the forces acting on the atoms. Negative of the gradient.

Return type
np.array

get_energy_and_cart_forces_at(*cart_coords*)

get_energy_and_cart_hessian_at(*cart_coords*)

get_energy_and_forces_at(*coords*)

Calculate forces and energies at the given coordinates.

The results are not saved in the Geometry object.

get_energy_at(*coords*)

get_energy_at_cart_coords(*cart_coords*)

get_fragments(*regex*)

get_imag_frequencies(*hessian=None*, *thresh=1e-06*)

get_normal_modes(*cart_hessian=None*, *full=False*)

Normal mode wavenumbers, eigenvalues and Cartesian displacements Hessian.

get_restart_info()

get_sphere_radius(*offset=4*)

get_subgeom(*indices*, *coord_type='cart'*, *sort=False*)

Return a Geometry containing a subset of the current Geometry.

Parameters

- **indices** (*iterable of ints*) -- Atomic indices that the define the subset of the current Geometry.
- **coord_type** (*str, ("cart", "redund"), optional*) -- Coordinate system of the new Geometry.

Returns
sub_geom -- Subset of the current Geometry.

Return type
Geometry

get_subgeom_without(*indices*, ***kwargs*)

get_temporary_coords(*coords*)

get_thermoanalysis(*energy=None*, *cart_hessian=None*, *T=298.15*, *p=101325*, *point_group='c1'*)

get_trans_rot_projector(*full=False*)

property gradient

Negative of the force.

Returns
gradient -- 1d array containing the negative of the current forces.

Return type
np.array

property hessian

Matrix of second derivatives of the energy in respect to atomic displacements.

Returns
hessian -- 2d array containing the second derivatives of the energy with respect to atomic/coordinate displacements depending on the type of coordiante system.

Return type
np.array

property inertia_tensor

property is_analytical_2d

jmol(*atoms=None*, *cart_coords=None*)

Show geometry in jmol.

property layers

mass_weigh_hessian(*hessian*)

property masses

property masses_rep

property mm_inv

Inverted mass matrix.

Returns a diagonal matrix containing the inverted atomic masses.

property mm_sqrt_inv

Inverted square root of the mass matrix.

modes3d()

property moving_atoms

moving_atoms_jmol()

property mw_coords

Mass-weighted coordinates.

Returns
mw_coords -- 1d array containing the mass-weighted cartesian coordiantes.

Return type
np.array

property mw_gradient

Mass-weighted gradient.

Returns
mw_gradient -- Returns the mass-weighted gradient.

Return type
np.array

property mw_hessian

Mass-weighted hessian.

Returns
mw_hessian -- 2d array containing the mass-weighted hessian M^(-1/2) H M^(-1/2).

Return type
np.array

principal_axes_are_aligned()

Check if the principal axes are aligned with the cartesian axes.

Returns
aligned -- Wether the principal axes are aligned or not.

Return type
bool

reparametrize()

reset_coords(*new_typed_prims=None*)

rmsd(*geom*)

rotate(*copy=False*, *rng=None*)

set_calculator(*calculator*, *clear=True*)

Reset the object and set a calculator.

set_coord(*ind*, *coord*)

Set a coordinate by index.

Parameters

- **ind** (*int*) -- Index in of the coordinate to set in the self.coords array.
- **coord** (*float*) -- Coordinate value.

set_coords(*coords*, *cartesian=False*, *update_constraints=False*)

set_h5_hessian(*fn*)

set_restart_info(*restart_info*)

set_results(*results*)

Save the results from a dictionary.

Parameters
results (*dict*) -- The keys in this dict will be set as attributes in the current object, with the corresponding item as value.

standard_orientation()

property sum_formula

tmp_xyz_handle(*atoms=None*, *cart_coords=None*)

property total_mass

unweight_mw_hessian(*mw_hessian*)

Unweight a mass-weighted hessian.

Parameters
mw_hessian (*np.array*) -- Mass-weighted hessian to be unweighted.

Returns
hessian -- 2d array containing the hessian.

Return type
np.array

property vdw_radii

vdw_volume(***kwargs*)

without_hydrogens()

zero_frozen_forces(*cart_forces*)

pysisyphus.Geometry.**get_trans_rot_projector**(*cart_coords*, *masses*, *full=False*)

pysisyphus.Geometry.**get_trans_rot_vectors**(*cart_coords*, *masses*, *rot_thresh=1e-06*)

Vectors describing translation and rotation.

These vectors are used for the Eckart projection by constructing a projector from them.

See Martin J. Field - A Pratcial Introduction to the simulation of Molecular Systems, 2007, Cambridge University Press, Eq. (8.23), (8.24) and (8.26) for the actual projection.

See also https://chemistry.stackexchange.com/a/74923.

Parameters

- **cart_coords** (*np.array, 1d, shape (3 * atoms.size,)*) -- Atomic masses in amu.
- **masses** (*iterable, 1d, shape (atoms.size,)*) -- Atomic masses in amu.

Returns
ortho_vecs -- 2d array containing row vectors describing translations and rotations.

Return type
np.array(6, 3*atoms.size)

pysisyphus.Geometry.**inertia_tensor**(*coords3d*, *masses*)

Inertita tensor.

x^2 xy xz |

(x y z)^T . (x y z) = | xy y^2 yz |

xz yz z^2 |

Basic infrastructure to work with internal coordinates is provided by *RedundantCoords*.

class pysisyphus.intcoords.RedundantCoords.**HybridRedundantCoords**(**args*, ***kwargs*)

class pysisyphus.intcoords.RedundantCoords.**RedundantCoords**(*atoms*, *coords3d*, *masses=None*, *bond_factor=1.3*, *typed_prims=None*, *define_prims=None*, *constrain_prims=None*, *freeze_atoms=None*, *freeze_atoms_exclude=False*, *internals_with_frozen=False*, *define_for=None*, *bonds_only=False*, *check_bends=True*, *rebuild=True*, *bend_min_deg=15*, *dihed_max_deg=175*, *lb_min_deg=175*, *weighted=False*, *min_weight=0.3*, *svd_inv_thresh=0.000316*, *recalc_B=False*, *tric=False*, *hybrid=False*, *hbond_angles=False*, *rm_for_frag=None*)

property B

Wilson B-Matrix

property B_inv

Generalized inverse of the Wilson B-Matrix.

property B_inv_prim

Generalized inverse of the primitive Wilson B-Matrix.

property B_prim

Wilson B-Matrix

property Bt_inv

Transposed generalized inverse of the Wilson B-Matrix.

property Bt_inv_prim

Transposed generalized inverse of the primitive Wilson B-Matrix.

property C

Diagonal matrix. Entries for constraints are set to one.

property P

Projection matrix onto B. See [1] Eq. (4).

backtransform_hessian(*redund_hessian*, *int_gradient=None*)

Transform Hessian in internal coordinates to Cartesians.

property bend_atom_indices

property bend_indices

property bond_atom_indices

property bond_indices

property bond_typed_prims

property cartesian_indices

clear()

property constrained_indices

property coords

property coords3d

property dihedral_atom_indices

property dihedral_indices

eval(*coords3d*, *attr=None*)

get_K_matrix(*int_gradient=None*)

get_index_of_typed_prim(*typed_prim*)
Index in self.typed_prims for the supplied typed_prim.

get_prim_internals_by_indices(*indices*)

inv_B(*B*)

inv_Bt(*B*)

property linear_bend_indices

log(*message*)

log_int_grad_msg(*int_gradient*)

property outofplane_indices

property prim_coords

property prim_indices_set

property prim_internals

property primitives

print_typed_prims()

project_hessian(*H*, *shift=1000*)
Expects a hessian in internal coordinates. See Eq. (11) in [1].

project_vector(*vector*)
Project supplied vector onto range of B.

return_inds(*slice_*)

property rotation_indices

set_inds_from_typed_prims(*typed_prims*)

set_primitive_indices(*atoms*, *coords3d*)

transform_forces(*cart_forces*)
Combination of Eq. (9) and (11) in [1].

transform_hessian(*cart_hessian*, *int_gradient=None*)

Transform Cartesian Hessian to internal coordinates.

transform_int_step(*int_step*, *update_constraints=False*, *pure=False*)

property translation_indices

property typed_prims

class pysisyphus.intcoords.RedundantCoords.**TMTRIC**(*atoms*, **args*, ***kwargs*)

class pysisyphus.intcoords.RedundantCoords.**TRIC**(**args*, ***kwargs*)

5.5 Related Literature

1. The efficient optimization of molecular geometries using redundant internal coordinates
2. The generation and use of delocalized internal coordinates in geometry optimization
3. Geometry optimization in redundant internal coordinates
4. Geometry optimization made simple with translation and rotation coordinates

CHAPTER

SIX

CALCULATORS

Exploring potential energy surfaces (PESs) requires calculation of energies and its derivatives (gradients, Hessian matrices).

For testing purposes and method development, pysisyphus implements various analytical 2d potentials, as they allow fast evaluations of the aforementioned quantities. Actual (production) calculations are carried out by wrapping existing quantum chemistry codes (ORCA, TURBOMOLE, Gaussian etc.) and delegating the required calculations to them. Pysisyphus generates all necessary inputs, executes the QC code and parses their output for the requested quantities.

Furthermore pysisyphus provides several "meta"-calculators which wrap other (actual) calculators, to modify calculated energies and forces. Examples for this are the *Dimer* calculator, used for carrying out transition state searches or the *ONIOM* calculator, allowing multi-level calculations comprising different levels of theory.

External forces, e.g. a restraining spherical potential or harmonic restraints on primitive internal coordinates (stretches, bends, torsion) can be applied with *ExternalPotential*.

6.1 YAML input

Possible keywords for the YAML input can be derived from inspecting the possible arguments of the *Calculator* baseclass (see below) and the possible arguments of the respective calculator, e.g. ORCA or XTB.

The most commonly used keywords, derived from the *Calculator* baseclass are *mem*, handling the requested memory per core in MB, *pal*, handling the number of requested CPU cores, *charge*, the total charge of the system and *mult*, the systems multiplicity.

For (excited state, ES) calculations carried out with calculators derived from *OverlapCalculator* additional keywords are possible. The most common keywords controlling ES calculations are *track*, activating ES tracking, *ovlp_type*, selecting the tracking algorithm and *ovlp_with*, handling the selection of the reference cycle.

An example input highlighting the most important keywords is shown below.

```
geom:
 [... omitted ...]
calc:
 type: orca                    # Calculator type, e.g. g09/g16/openmolcas/
                               # orca/pyscf/turbomole/dftb+/mopac/psi4/xtb

 pal: 1                        # Number of CPU cores
 mem: 1000                     # Memory per core
 charge: 0                     # Charge
 mult: 1                       # Multiplicity
 # Keywords for ES calculations
 track: False                  # Activate ES tracking
```

(continues on next page)

(continued from previous page)

```
 ovlp_type: tden               # Tracking algorithm
 ovlp_with: previous           # Reference cycle selection
 # Additional calculator specific keywords
 [... omitted ...]
opt:
 [... omitted ...]
```

6.2 Calculator base classes

class pysisyphus.calculators.Calculator.**Calculator**(*calc_number=0*, *charge=0*, *mult=1*, *base_name='calculator'*, *pal=1*, *mem=1000*, *keep_kind='all'*, *check_mem=True*, *retry_calc=0*, *last_calc_cycle=None*, *clean_after=True*, *out_dir='qm_calcs'*, *force_num_hess=False*, *num_hess_kwargs=None*)

__init__(*calc_number=0*, *charge=0*, *mult=1*, *base_name='calculator'*, *pal=1*, *mem=1000*, *keep_kind='all'*, *check_mem=True*, *retry_calc=0*, *last_calc_cycle=None*, *clean_after=True*, *out_dir='qm_calcs'*, *force_num_hess=False*, *num_hess_kwargs=None*)

Base-class of all calculators.

Meant to be extended.

Parameters

- **calc_number** (*int, default=0*) -- Identifier of the Calculator. Used in distinguishing it from other Calculators, e.g. in ChainOfStates calculations. Also used in the creation of filenames.
- **charge** (*int, default=0*) -- Molecular charge.
- **mult** (*int, default=1*) -- Molecular multiplicity (1 = singlet, 2 = doublet, ...)
- **base_name** (*str, default=calculator*) -- Generated filenames will start with this string.
- **pal** (*int, default=1*) -- Positive integer that gives the number of physical cores to use on 1 node.
- **mem** (*int, default=1000*) -- Mememory per core in MB. The total amount of memory is given as mem*pal.
- **check_mem** (*bool, default=True*) -- Whether to adjust the requested memory if too much is requested.
- **retry_calc** (*int, default=0*) -- Number of additional retries when calculation failed.
- **last_calc_cycle** (*int*) -- Internal variable used in restarts.
- **clean_after** (*bool*) -- Delete temporary directory were calculations were executed after a calculation.
- **out_dir** (*str*) -- Path that is prepended to generated filenames.
- **force_hess_kwargs** (*bool, default False*) -- Force numerical Hessians.
- **num_hess_kwargs** (*dict*) -- Keyword arguments for finite difference Hessian calculation.

apply_keep_kind()

apply_set_plans(*kept_fns*, *set_plans=None*)

build_set_plans(*_set_plans=None*)

clean(*path*)

Delete the temporary directory.

Parameters
path (*Path*) -- Directory to delete.

conf_key = None

force_num_hessian()

Always calculate numerical Hessians.

classmethod geom_from_fn(*fn*, ***kwargs*)

get_cmd(*key='cmd'*)

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_num_hessian(*atoms*, *coords*, ***prepare_kwargs*)

get_restart_info()

Return a dict containing chkfiles.

Returns
restart_info -- Dictionary holding the calculator state. Used for restoring calculaters in restarted calculations.

Return type
dict

keep(*path*)

Backup calculation results.

Parameters
path (*Path*) -- Temporary directory of the calculation.

Returns
kept_fns -- Dictonary holding the filenames that were backed up. The keys correspond to the type of file.

Return type
dict

log(*message=''*)

Write a log message.

Wraps the logger variable.

Parameters
message (*str*) -- Message to be logged.

make_fn(*name*, *counter=None*, *return_str=False*)

Make a full filename.

Return a full filename including the calculator name and the current counter given a suffix.

Parameters

- **name** (*str*) -- Suffix of the filename.
- **counter** (*int, optional*) -- If not given use the current calc_counter.
- **return_str** (*int, optional*) -- Return a string instead of a Path when True.

Returns
fn -- Filename.

Return type
str

property name

popen(*cmd*, *cwd=None*)

prepare(*inp*)

Prepare a temporary directory and write input.

Similar to prepare_path, but the input is also written into the prepared directory.

6.2.1 Paramters

inp
[str] Input to be written into the file `self.inp_fn` in the prepared directory.

returns

path: Path
Prepared directory.

prepare_coords(*atoms*, *coords*)

Get 3d coords in Angstrom.

Reshape internal 1d coords to 3d and convert to Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns
coords -- 3D-array holding coordinates in Angstrom.

Return type
np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

`prepare_path`(*use_in_run=False*)

Get a temporary directory handle.

Create a temporary directory that can later be used in a calculation.

Parameters

use_in_run (*bool, option*) -- Sets the internal variable `self.path_already_prepared` that is later read by `self.run()`. No new temporary directory will be created in `self.run()`.

Returns

path: Path
Prepared directory.

`prepare_pattern`(*raw_pat*)

Prepare globs.

Transforms an entry of `self.to_keep` into a glob and a key suitable for the use in `self.keep()`.

Parameters

raw_pat (*str*) -- Entry of `self.to_keep`

Returns

- **pattern** (*str*) -- Glob that can be used in Path.glob()
- **multi** (*bool*) -- Flag if glob may match multiple files.
- **key** (*str*) -- A key to be used in the `kept_fns` dict.

`prepare_turbo_coords`(*atoms*, *coords*)

Get a Turbomole coords string.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns
coords -- String holding coordinates in Turbomole coords format.

Return type
str

`prepare_xyz_string`(*atoms*, *coords*)

Returns a xyz string in Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns
xyz_str -- Coordinates in .xyz format.

Return type
string

`print_capabilities`()

print_out_fn(*path*)

Print calculation output.

Prints the output of a calculator after a calculation.

Parameters
path (*Path*) -- Temporary directory of the calculation.

restore_org_hessian()

Restore original 'get_hessian' method, which may also fallback to numerical Hessians, if not implemented.

run(*inp*, *calc*, *add_args=None*, *env=None*, *shell=False*, *hold=False*, *keep=True*, *cmd=None*, *inc_counter=True*, *run_after=True*, *parser_kwargs=None*, *symlink=True*)

Run a calculation.

The bread-and-butter method to actually run an external quantum chemistry code.

Parameters

- **inp** (*str*) -- Input for the external program that is written to the temp-dir.
- **calc** (*str, hashable*) -- Key (and more or less type of calculation) to select the right parsing function from `self.parser_funcs`.
- **add_args** (*iterable, optional*) -- Additional arguments that will be appended to the program call.
- **env** (*Environment, optional*) -- A potentially modified environment for the subprocess call.
- **shell** (*bool, optional*) -- Use a shell to execute the program call. Need for Turbomole were we chain program calls like dscf; escf.
- **hold** (*bool, optional*) -- Wether to remove the temporary directory after the calculation.
- **keep** (*bool, optional*) -- Wether to backup files as specified in `self.to_keep()`. Usually you want this.
- **cmd** (*str or iterable, optional*) -- Overwrites `self.base_cmd`.
- **inc_counter** (*bool, optional*) -- Wether to increment the counter after a calculation.

Returns
results -- Dictionary holding all applicable results of the calculations like the energy, a forces vector and/or excited state energies from TDDFT.

Return type
dict

run_after(*path*)

Meant to be extended.

This method is called after a calculation was done, but before entering `self.keep()` and `self.clean()`. Can be used to call tools like formchk or ricctools.

set_restart_info(*restart_info*)

Sets restart information (chkfiles etc.) on the calculator.

Parameters
restart_info (*dict*) -- Dictionary holding the calculator state. Used for restoring calculaters in restarted calculations.

verify_chkfiles(*chkfiles*)

Checks if given chkfiles exist and return them as Paths

Parameters
chkfiles (*dict*) -- Dictionary holding the chkfiles. The keys correspond to the attribute names, the values are strs holding the (potentially full) filename (path).

Returns
paths -- Dictionary of Paths.

Return type
dict

class pysisyphus.calculators.Calculator.**HessKind**(*value*)

An enumeration.

NUMERICAL = 2

ORG = 1

class pysisyphus.calculators.Calculator.**KeepKind**(*value*)

An enumeration.

ALL = 1

LATEST = 2

NONE = 3

class pysisyphus.calculators.Calculator.**SetPlan**(*key*, *name=None*, *condition=<function SetPlan.<lambda>>*, *fail=None*)

condition()

fail: Optional[Callable] = None

key: str

name: Optional[str] = None

6.3 OverlapCalculator base class

class pysisyphus.calculators.OverlapCalculator.**NTOs**(*ntos*, *lambdas*)

Bases: tuple

lambdas

Alias for field number 1

ntos

Alias for field number 0

class pysisyphus.calculators.OverlapCalculator.**OverlapCalculator**(**args*, *track=False*, *ovlp_type='tden'*, *double_mol=False*, *ovlp_with='previous'*, *adapt_args=(0.5, 0.3, 0.6)*, *cdds=None*, *orient=''*, *dump_fn='overlap_data.h5'*, *h5_dump=False*, *conf_thresh=0.001*, *mos_ref='cur'*, *mos_renorm=True*, ***kwargs*)

Bases: *Calculator*

H5_MAP = {'Ca': 'Ca_list', 'Cb': 'Cb_list', 'Xa': 'Xa_list', 'Xb': 'Xb_list', 'Ya': 'Ya_list', 'Yb': 'Yb_list', 'all_energies': 'all_energies_list', 'coords': 'coords_list', 'ref_roots': 'reference_roots', 'roots': 'roots_list'}

OVLP_TYPE_VERBOSE = {'nto': 'natural transition orbital overlap', 'nto_org': 'original natural transition orbital overlap', 'tden': 'transition density matrix overlap', 'top': 'transition orbital pair overlap', 'wf': 'wavefunction overlap'}

VALID_CDDS = (None, 'calc', 'render')

VALID_KEYS = ['wf', 'tden', 'nto', 'nto_org', 'top']

VALID_XY = ('X', 'X+Y', 'X-Y')

calc_cdd_cube(*root*, *cycle=-1*)

conf_thresh

self.dyn_roots = int(dyn_roots) if self.dyn_roots != 0:

self.dyn_roots = 0 self.log("dyn_roots = 0 is hardcoded right now")

property data_model

dump_overlap_data()

static from_overlap_data(*h5_fn*, *set_wfow=False*)

get_ci_coeffs_for(*ind*)

get_h5_group()

get_indices(*indices=None*)

A new root is determined by selecting the overlap matrix row corresponding to the reference root and checking for the root with the highest overlap (at the current geometry).

The overlap matrix is usually formed by a double loop like:

overlap_matrix = np.empty((ref_states, cur_states)) for i, ref_state in enumerate(ref_states):

for j, cur_state in enumerate(cur_states):
overlap_matrix[i, j] = make_overlap(ref_state, cur_state)

So the reference states run along the rows. Thats why the ref_state index comes first in the 'indices' tuple.

static get_mo_norms(*C*, *S_AO*)

get_orbital_matrices(*indices=None*, *S_AO=None*)

Return MO coefficents and AO overlaps for the given indices.

If not provided, a AO overlap matrix is constructed from one of the MO coefficient matrices (controlled by self.mos_ref). Also, if requested one of the two MO coefficient matrices is re-normalized.

get_ref_mos(*C_ref*, *C_cur*)

static get_sao_from_mo_coeffs(*C*)

Recover AO overlaps from given MO coefficients.

For MOs in the columns of mo_coeffs:

S_AO = C^{1}^T C^{1} S_AO C = C^{1}^T (S_AO C)^T = C^{1} C^T S_AO^T = C^{1} C^T S_AO C = I

get_tden_overlaps(*indices=None*, *S_AO=None*)

get_top_differences(*indices=None*, *S_AO=None*)

Transition orbital projection.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

render_cdd_cube()

static renorm_mos(*C*, *S_AO*)

store_overlap_data(*atoms*, *coords*, *path=None*, *overlap_data=None*)

property stored_calculations

track_root(*ovlp_type=None*)

Check if a root flip occured occured compared to the previous cycle by calculating the overlap matrix wrt. a reference cycle.

pysisyphus.calculators.OverlapCalculator.**get_data_model**(*exc_state_num*, *occ_a*, *virt_a*, *occ_b*, *virt_b*, *ovlp_type*, *atoms*, *max_cycles*)

6.4 Calculators with Excited state capabilities

6.4.1 Gaussian09

class pysisyphus.calculators.Gaussian09.**Gaussian09**(**args*, ***kwargs*)

conf_key = 'gaussian09'

6.4.2 Gaussian16

class pysisyphus.calculators.Gaussian16.**Gaussian16**(*route*, *gbs=''*, *gen=''*, *keep_chk=False*, *stable=''*, *fchk=None*, ***kwargs*)

Bases: *OverlapCalculator*

conf_key = 'gaussian16'

get_chkfiles()

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

make_exc_str()

make_fchk(*path*)

make_gbs_str()

parse_635r_dump(*dump_path*, *roots*, *nmos*)

parse_all_energies(*fchk=None*)

parse_charges(*path=None*)

parse_double_mol(*path*, *out_fn=None*)

static parse_fchk(*fchk_path*, *keys*)

parse_force(*path*)

parse_hessian(*path*)

parse_keyword(*text*)

parse_log(**args*, ***kwargs*)

parse_stable(*path*)

parse_tddft(*path*)

prepare_input(*atoms*, *coords*, *calc_type*, *did_stable=False*, *point_charges=None*)

Meant to be extended.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

reuse_data(*path*)

run_after(*path*)

Meant to be extended.

This method is called after a calculation was done, but before entering `self.keep()` and `self.clean()`. Can be used to call tools like formchk or ricctools.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

run_double_mol_calculation(*atoms*, *coords1*, *coords2*)

run_rwfdump(*path*, *rwf_index*, *chk_path=None*)

run_stable(*atoms*, *coords*, ***prepare_kwargs*)

set_chkfiles(*chkfiles*)

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

6.4.3 OpenMolcas

Pysisyphus currently supports energy and gradient calculations utilizing the *&rasscf* and/or the *&mcpdft* sections. Neither analytical nor numerical Hessians are yet implemented for the OpenMolcas-calculator.

Two keywords are always required: *inporb* and *basis*, with the former pointing to a *.RasOrb* file and the latter containing the selected atomic orbital basis, e.g., *ano-rcc-vdzp*. Additional input for the *&gateway*, *&rasscf* and *&mcpdft* sections can be given via the respective keyword(s).

Due to restrictions of the current design, simple keywords that don't take further arguments as *cmsi* in *&rasscf* or *grad* and *mspdft* in *&mcpdft* still must be given with a trailing colon. See below for an example.

Listing 6.1: Optimization using compressed-multi-state PDFT.

```
geom:
 type: redund
 fn: |
  4

  C        -1.0398336639         0.0            0.0
  S         0.6002429216         0.0            0.0
  H -1.6321592382        -0.94139864        0.0
  H        -1.6321592382         0.94139864        0.0
calc:
 type: openmolcas
 basis: cc-pvdz
 charge: 0
 mult: 1
 inporb: /home/johannes/tmp/359_cmspdft/359_cmspdft.RasOrb
 rasscf:
  ciroot: 3 3 1
  mdrlxroot: 2
  cmsi:
 mcpdft:
  ksdft: t:pbe
  grad:
  mspdft:
```

(continues on next page)

(continued from previous page)

```
opt:
 thresh: gau
```

class pysisyphus.calculators.OpenMolcas.**OpenMolcas**(*basis*, *inporb*, *rasscf=None*, *gateway=None*, *mcpdft=None*, *track=True*, ***kwargs*)

Bases: *Calculator*

build_gateway_str()

build_mcpdft_str()

build_rasscf_str()

build_rassi_str()

build_str_from_dict(*dct*)

conf_key = 'openmolcas'

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

get_pal_env()

get_root()

parse_energies(*text*)

parse_energy(*path*)

parse_gradient(*path*)

parse_rassi_track(*path*)

prepare_coords(*atoms*, *coords*)

Get 3d coords in Angstrom.

Reshape internal 1d coords to 3d and convert to Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns

coords -- 3D-array holding coordinates in Angstrom.

Return type

np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

reattach(*last_calc_cycle*)

run_calculation(*atoms*, *coords*, *calc_type='energy'*)

6.4.4 ORCA 4.2.1 / 5.0.1

class pysisyphus.calculators.ORCA.**ORCA**(*keywords*, *blocks=''*, *gbw=None*, *do_stable=False*, *numfreq=False*, *json_dump=True*, ***kwargs*)

Bases: *OverlapCalculator*

__init__(*keywords*, *blocks=''*, *gbw=None*, *do_stable=False*, *numfreq=False*, *json_dump=True*, ***kwargs*)

ORCA calculator.

Wrapper for creating ORCA input files for energy, gradient and Hessian calculations. The PAL and memory inputs must not be given in the keywords and/or blocks, as they are handled by the 'pal' and 'memory' arguments.

Parameters

- **keywords** (*str*) -- Keyword line, as normally given in ORCA, excluding the leading "!".
- **blocks** (*str, optional*) -- ORCA block input(s), e.g. for TD-DFT calculations (%tddft ... end). As the blocks start with a leading "%", wrapping the input in quotes ("") is required, otherwise the parsing will fail.
- **gbw** (*str, optional*) -- Path to an input gbw file, which will be used as initial guess for the first calculation. Will be overriden later, with the path to the gbw file of a previous calculation.
- **do_stable** (*bool, optional*) -- Run stability analysis until a stable wavefunction is obtained, before every calculation.
- **numfreq** (*bool, optional*) -- Use numerical frequencies instead of analytical ones.
- **json_dump** (*bool, optional*) -- Whether to dump the wavefunction to JSON via orca_2json. The JSON can become very large in calculations comprising many basis functions.

check_termination(**args*, ***kwargs*)

clean_tmp(*path*)

conf_key = 'orca'

get_block_str()

get_chkfiles()

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_moinp_str(*gbw*)

get_stable_wavefunction(*atoms*, *coords*)

parse_all_energies(*text=None*, *triplets=None*)

static parse_atoms_coords(*inp*, **args*, ***kwargs*)

static parse_cis(*cis*)

Simple wrapper of external function.

Currently, only returns X and Y.

parse_energy(*path*)

parse_engrad(*path*)

static parse_engrad_info(*inp*, **args*, ***kwargs*)

static parse_gbw(*gbw_fn*)

static parse_hess_file(*inp*, **args*, ***kwargs*)

parse_hessian(*path*)

parse_mo_numbers(*out_fn*)

parse_stable(*path*)

prepare_input(*atoms*, *coords*, *calc_type*, *point_charges=None*, *do_stable=False*)

Meant to be extended.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

reattach(*last_calc_cycle*)

run_after(*path*)

Meant to be extended.

This method is called after a calculation was done, but before entering `self.keep()` and `self.clean()`. Can be used to call tools like formchk or ricctools.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

Basically some kind of dummy method that can be called to execute ORCA with the stored cmd of this calculator.

set_chkfiles(*chkfiles*)

set_mo_coeffs(*mo_coeffs=None*, *gbw=None*)

static set_mo_coeffs_in_gbw(*in_gbw_fn*, *out_gbw_fn*, *mo_coeffs*)

See self.parse_gbw.

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

pysisyphus.calculators.ORCA.**get_exc_ens_fosc**(*wf_fn*, *cis_fn*, *log_fn*)

pysisyphus.calculators.ORCA.**get_name**(*text*)

Return string that comes before first character & offset.

pysisyphus.calculators.ORCA.**make_sym_mat**(*table_block*)

pysisyphus.calculators.ORCA.**parse_orca_cis**(*cis_fn*)

Read binary CI vector file from ORCA.
Loosly based on TheoDORE 1.7.1, Authors: S. Mai, F. Plasser https://sourceforge.net/p/theodore-qc

pysisyphus.calculators.ORCA.**parse_orca_gbw**(*gbw_fn*)

Adapted from https://orcaforum.kofo.mpg.de/viewtopic.php?f=8&t=3299

The first 5 long int values represent pointers into the file:

Pointer @+0: Internal ORCA data structures Pointer @+8: Geometry Pointer @+16: BasisSet Pointer @+24: Orbitals Pointer @+32: ECP data

pysisyphus.calculators.ORCA.**save_orca_pc_file**(*point_charges*, *pc_fn*, *hardness=None*)

6.4.5 PySCF 1.7.6

6.4.6 Turbomole 7.x

Pysisyphus does not implement a wrapper for *define*, so the user has to manually prepare a directory containing a valid control file. An automated *define* wrapper, restricted to ground state functionality, is available via the QCEngine project, to which I contributed the Turbomole harness.

Care should be taken to include only the minimum amount of necessary files in the *control_path* directory, e.g., *(auxbasis, basis, control, coord, mos)* for a closed-shell calculation using RI. **A gradient file must not be present in control_path, as well as other subdirectories and files with .out extension.** The *coord* file, while not strictly required, should be kept too, to facilitate testing of the setup with standalone Turbomole.

It may be a good idea to pre-converge the calculation in *control_path*, to see if the setup is correct and actually works. Resulting files like *energy*, *statistics* can be deleted; *mos* should be kept, as the converged MOs are reused in pysisyphus.

If an excited-state optimization is desired, care has to be taken, to include **$exopt [n]** for TD-DFT/TDA or the **geoopt state=([n])** (ricc2)! Tracking of excited states is currently possible for closed shell *egrad* and *ricc2* calculations.

The current implementation was tested against Turbomole 7.4.1 and QCEngine 0.19.0. Please see examples/complex/11_turbomole_gs_tsopt for a full example where Turbmole is utilized in a growing string calculation. The same example, using QCEngine, is found in examples/complex/12_qcengine_turbomole_gs_tsopt. MOs are not reused with the QCEngine calculator, so the native pysisyphus calculator is probably faster.

class pysisyphus.calculators.Turbomole.**Turbomole**(*control_path=None*, *simple_input=None*, *root=None*, *double_mol_path=None*, *cosmo_kwargs=None*, ***kwargs*)

Bases: *OverlapCalculator*

append_control(*to_append*, *log_msg=''*, ***kwargs*)

conf_key = 'turbomole'

get_chkfiles()

get_energy(*atoms*, *coords*, ***prepare_kwargs*)
Meant to be extended.

get_forces(*atoms*, *coords*, *cmd=None*, ***prepare_kwargs*)
Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_pal_env()

get_ricc2_root(*text*)

static make_molden(*path*)

parse_all_energies()

parse_cc2_vectors(*ccre*)

parse_ci_coeffs()

parse_double_mol(*path*)

Parse a double molecule overlap matrix from Turbomole output to be used with WFOWrapper.

parse_energy(*path*)

parse_force(*path*)

parse_gs_energy()

Several places are possible: $subenergy from control file total energy from turbomole.out Final MP2 energy from turbomole.out with ADC(2) Final CC2 energy from turbomole.out with CC(2)

parse_hessian(*path*, *fn=None*)

parse_mos()

parse_td_vectors(*text*)

For TDA calculations only the X vector is present in the ciss_a/etc. file. In TDDFT calculations there are twise as much items compared with TDA. The first half corresponds to (X+Y) and the second half to (X-Y). X can be calculated as X = ((X+Y)+(X-Y))/2. Y is then given as Y = (X+Y)-X. The normalization can then by checked as

np.concatenate((X, Y)).dot(np.concatenate((X, -Y)))

and should be 1.

static parse_tddft_tden(*inp*, **args*, ***kwargs*)

prepare_input(*atoms*, *coords*, *calc_type*, *point_charges=None*)

To rectify this we have to construct the basecmd dynamically and construct it ad hoc. We could set a RI flag in the beginning and select the correct scf binary here from it. Then we select the following binary on demand, e.g. aoforce or rdgrad or egrad etc.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

prepare_point_charges(*point_charges*)

$point_charges <x> <y> <z> <q>

prepare_td(*text*)

run_after(*path*)

Meant to be extended.

This method is called after a calculation was done, but before entering `self.keep()` and `self.clean()`. Can be used to call tools like formchk or ricctools.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

run_double_mol_calculation(*atoms*, *coords1*, *coords2*)

set_chkfiles(*chkfiles*)

set_occ_and_mo_nums(*text*)

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

sub_control(*pattern*, *repl*, *log_msg=''*, ***kwargs*)

pysisyphus.calculators.Turbomole.**control_from_simple_input**(*simple_inp*, *charge*, *mult*, *cosmo_kwargs=None*)

Create control file from 'simple input'.

See examples/opt/26_turbomole_simple_input/ for an example.

pysisyphus.calculators.Turbomole.**get_cosmo_data_groups**(*atoms*, *epsilon*, *rsolv=None*, *dcosmo_rs=None*)

pysisyphus.calculators.Turbomole.**index_strs_for_atoms**(*atoms*)

pysisyphus.calculators.Turbomole.**render_data_groups**(*raw_data_groups*)

6.4.7 DFTB+ 20.x

class pysisyphus.calculators.DFTBp.**DFTBp**(*parameter*, **args*, *slakos=None*, *root=None*, ***kwargs*)

Bases: *OverlapCalculator*

conf_key = 'dftbp'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

static get_excited_state_str(*root*, *forces=False*)

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

static get_gen_str(*atoms*, *coords*)

hubbard_derivs = {'3ob': {'Br': -0.0573, 'C': -0.1492, 'Ca': -0.034, 'Cl': -0.0697, 'F': -0.1623, 'H': -0.1857, 'I': -0.0433, 'K': -0.0339, 'Mg': -0.02, 'N': -0.1535, 'Na': -0.0454, 'O': -0.1575, 'P': -0.14, 'S': -0.11, 'Zn': -0.03}}

max_ang_moms = {'3ob': {'Br': 'd', 'C': 'p', 'Ca': 'p', 'Cl': 'd', 'F': 'p', 'H': 's', 'I': 'd', 'K': 'p', 'Mg': 'p', 'N': 'p', 'Na': 'p', 'O': 'p', 'P': 'd', 'S': 'd', 'Zn': 'd'}, 'mio-ext': {'C': 'p', 'H': 's', 'N': 'p', 'O': 'p'}}

parse_all_energies(*out_fn=None*, *exc_dat=None*)

parse_energy(*path*)

static parse_exc_dat(*text*)

parse_forces(*path*)

parse_total_energy(*text*)

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

pysisyphus.calculators.DFTBp.**parse_mo**(*eigvec*)

pysisyphus.calculators.DFTBp.**parse_xplusy**(*text*)

6.5 Calculators with Ground state capabilities

6.5.1 MOPAC 2016

class pysisyphus.calculators.MOPAC.**MOPAC**(*method='PM7'*, ***kwargs*)

Bases: *Calculator*

http://openmopac.net/manual/

CALC_TYPES = {'energy': '1SCF', 'gradient': '1SCF GRADIENTS', 'hessian': 'DFORCE FORCE LET'}

METHODS = ['am1', 'pm3', 'pm6', 'pm6-dh2', 'pm6-d3', 'pm6-dh+', 'pm6-dh2', 'pm6-dh2x', 'pm6-d3h4', 'pm6-d3h4x', 'pm7', 'pm7-ts']

MULT_STRS = {1: 'SINGLET', 2: 'DOUBLET', 3: 'TRIPLET', 4: 'QUARTET', 5: 'QUINTET', 6: 'SEXTET', 7: 'SEPTET', 8: 'OCTET'}

base_cmd

Do only SCF AUX: Creates a checkpoint file NOREO: Dont reorient geometry

Type
1SCF

conf_key = 'mopac'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

parse_energy(*path*)

static parse_energy_from_aux(*inp*, **args*, ***kwargs*)

parse_grad(*path*)

parse_hessian(*path*)

static parse_hessian_from_aux(*inp*, **args*, ***kwargs*)

prepare_coords(*atoms*, *coords*, *opt=False*)

Get 3d coords in Angstrom.

Reshape internal 1d coords to 3d and convert to Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns

coords -- 3D-array holding coordinates in Angstrom.

Return type

np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*, *opt=False*)

Meant to be extended.

read_aux(*path*)

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

6.5.2 Psi4

class pysisyphus.calculators.Psi4.**Psi4**(*method*, *basis*, *to_set=None*, *to_import=None*, *pcm='iefpcm'*, *solvent=None*, *write_fchk=False*, ***kwargs*)

Bases: *Calculator*

conf_key = 'psi4'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_fchk_str()

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

parse_energy(*path*)

parse_grad(*path*)

parse_hessian(*path*)

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

6.5.3 QCEngine

6.5.4 XTB 6.x

class pysisyphus.calculators.XTB.**OptResult**(*opt_geom*, *opt_log*)

Bases: `tuple`

opt_geom

Alias for field number 0

opt_log

Alias for field number 1

class pysisyphus.calculators.XTB.**XTB**(*gbsa=''*, *alpb=''*, *gfn=2*, *acc=1.0*, *iterations=250*, *etemp=None*, *retry_etemp=None*, *restart=False*, *topo=None*, *topo_update=None*, *quiet=False*, ***kwargs*)

Bases: *Calculator*

__init__(*gbsa=''*, *alpb=''*, *gfn=2*, *acc=1.0*, *iterations=250*, *etemp=None*, *retry_etemp=None*, *restart=False*, *topo=None*, *topo_update=None*, *quiet=False*, ***kwargs*)

XTB calculator.

Wrapper for running energy, gradient and Hessian calculations by XTB.

Parameters

- **gbsa** (*str, optional*) -- Solvent for GBSA calculation, by default no solvent model is used.
- **alpb** (*str, optional*) -- Solvent for ALPB calculation, by default no solvent model is used.
- **gfn** (*int or str, must be (0, 1, 2, or "ff")*) -- Hamiltonian for the XTB calculation (GFN0, GFN1, GFN2, or GFNFF).
- **acc** (*float, optional*) -- Accuracy control of the calculation, the lower the tighter several numerical thresholds are chosen.
- **iterations** (*int, optional*) -- The number of iterations in SCC calculation.
- **topo** (*str, optional*) -- Path the a GFNFF-topolgy file. As setting up the topology may take some time for sizable systems, it may be desired to reuse the file.
- **topo_update** (*int*) -- Integer controlling the update interval of the GFNFF topology update. If supplied, the topolgy will be recreated every N-th calculation.
- **mem** (*int*) -- Mememory per core in MB.
- **quiet** (*bool, optional*) -- Suppress creation of log files.

static check_termination(*inp*, **args*, ***kwargs*)

conf_key = 'xtb'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_mdrestart_str(*coords*, *velocities*)

coords and velocities have to given in au!

get_pal_env()

get_retry_args()

parse_charges(*fn=None*)

parse_charges_from_json(*fn=None*)

parse_energy(*path*)

parse_gradient(*path*)

parse_hessian(*path*)

parse_md(*path*)

parse_opt(*path*, *keep_log=False*)

parse_topo(*path*)

prepare_add_args(*xcontrol=None*)

prepare_coords(*atoms*, *coords*)

Get 3d coords in Angstrom.

Reshape internal 1d coords to 3d and convert to Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns

coords -- 3D-array holding coordinates in Angstrom.

Return type

np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*, *point_charges=None*)

Meant to be extended.

reattach(*last_calc_cycle*)

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

run_md(*atoms*, *coords*, *t*, *dt*, *velocities=None*, *dump=1*)

Expecting t and dt in fs, even though xtb wants t in ps!

run_opt(*atoms*, *coords*, *keep=True*, *keep_log=False*)

run_topo(*atoms*, *coords*)

write_mdrestart(*path*, *mdrestart_str*)

6.5.5 Dalton

class pysisyphus.calculators.Dalton.**Dalton**(*basis*, *method='hf'*, ***kwargs*)

Bases: *Calculator*

compute(*mol*, *prop*)

conf_key = 'dalton'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

prepare_input(*atoms*, *coords*)

Meant to be extended.

6.5.6 OpenBabel

class pysisyphus.calculators.OBabel.**OBabel**(*ff='gaff'*, *mol=None*, ***kwargs*)

Bases: *Calculator*

conv_dict = {'kcal/mol': 627.5094740630558, 'kj/mol': 2625.4996394798254}

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

setup(*atoms*, *coords*)

6.5.7 CFOUR

class pysisyphus.calculators.CFOUR.**CFOUR**(*cfour_input*, *wavefunction_dump=True*, *initden_file=None*, ***kwargs*)

Bases: *Calculator*

__init__(*cfour_input*, *wavefunction_dump=True*, *initden_file=None*, ***kwargs*)

CFOUR calculator.

Wrapper handling CFOUR ground state energy and gradient calculations.

Parameters

- **cfour_input** (*dict*) -- CFOUR keywords and values. Note: "on" must be encapsulated in quotes to avoid being translated to True by YAML.
- **wavefunction_dump** (*bool, optional*) -- Whether or not to keep ground state SCF orbitals for each geometry step.
- **initden_file** (*str, optional*) -- Path to an input initden file for use as a guess SCF density.

conf_key = 'cfour'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

keep(*path*)

Backup calculation results.

Parameters
path (*Path*) -- Temporary directory of the calculation.

Returns
kept_fns -- Dictonary holding the filenames that were backed up. The keys correspond to the type of file.

Return type
dict

parse_energy(*path*)

parse_gradient(*path*)

prepare(*inp*)

Prepare a temporary directory and write input.

Similar to prepare_path, but the input is also written into the prepared directory.

6.5.7.1 Paramters

inp
[str] Input to be written into the file `self.inp_fn` in the prepared directory.

returns

path: Path
Prepared directory.

prepare_coords(*atoms*, *coords*)

Get 3d coords in Bohr

Reshape internal 1d coords to 3d.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns
coords -- 3D-array holding coordinates in Bohr.

Return type
np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

run_calculation(*atoms*, *coords*, *calc_type*, ***prepare_kwargs*)

pysisyphus.calculators.CFOUR.**calc_centroid**(*coords_3d*)

pysisyphus.calculators.CFOUR.**calc_rot_matrix**(*source_coords_3d*, *target_coords_3d*, *source_centroid*, *target_centroid*)

pysisyphus.calculators.CFOUR.**parse_cfour_energy**(*out_fn*)

pysisyphus.calculators.CFOUR.**parse_cfour_gradient**(*grd_fn*, *out_fn*, *coords_pysis_frame_3d*)

pysisyphus.calculators.CFOUR.**read_cfour_geom**(*out_fn*)

pysisyphus.calculators.CFOUR.**rotate_gradient**(*gradient*, *coords_pysis_frame_3d*, *coords_comp_frame_3d*)

6.6 Meta (wrapping) Calculators

6.6.1 ExternalPotential

6.6.2 Restraint

```
# General input structure for restraints
calc:
 type: ext
 # Multiple potentials could be specified here as a list
 potentials:
   # Primitive internal coordinate restraint
   - type: restraint
     # List of restraints; could also be multiple restraints. Each restraint is given as
     # list of 2 or 3 items.
     #
     # The first item always specifies an internal coordinate,
     # whereas the second argument is a force constant (in atomic units; actual units
     # depend on the coordinate). Optionally a reference value (third argument) can be
     # given. If omitted, the initial coordinate value is used as reference value.
     restraints: [[[BOND, 0, 1], 10, 3.0]]
     # The commented out input below would restrain the bond at its initial value.
     #restraints: [[[BOND, 0, 1], 10]]
     # Multiple restraints are specified as given below.
```

(continues on next page)

(continued from previous page)

```
    #restraints: [[[BOND, 0, 1], 10], [[BEND, 0, 1, 2], 1.0]]
calc:
 type: [actual calculator that is wrapped by ExternalPotential]
```

6.6.3 HarmonicSphere

6.6.4 LogFermi

6.6.5 RMSD

6.6.6 DFT-D3

Method to add DFT-D3 dispersion corrections as an external potential via the program developed by the Grimme group <https://www.chemie.uni-bonn.de/grimme/de/software/dft-d3/get_dft-d3>.

This is for use with calculators that do not natively provide D3 corrections (e.g. OpenMolcas). Usage mirrors that of other external potentials, with an example given below.

```
# General input structure for restraints
calc:
 type: ext
 # Multiple potentials could be specified here as a list
 potentials:
   # Add atom-pairwise D3 dispersion correction as a differentiable, external potential
   - type: d3
     # Functional is specified in TURBOMOLE format, all lower case.
     functional: pbe
     # Optional Becke-Johnson damping, default false, recommended true
     bjdamping: true

calc:
 type: [actual calculator that is wrapped by ExternalPotential]
```

class pysisyphus.calculators.DFTD3.**DFTD3**(*geom*, *functional*, *bjdamping=False*, ***kwargs*)

calc(*coords3d*, *gradient=False*)

conf_key = 'dftd3'

parse_energy(*path*)

parse_gradient(*path*)

6.6.7 AFIR

class pysisyphus.calculators.AFIR.**AFIR**(*calculator*, *fragment_indices*, *gamma*, *rho=1*, *p=6*, *ignore_hydrogen=False*, *zero_hydrogen=True*, *complete_fragments=True*, *dump=True*, *h5_fn='afir.h5'*, *h5_group_name='afir'*, ***kwargs*)

Bases: *Calculator*

__init__(*calculator*, *fragment_indices*, *gamma*, *rho=1*, *p=6*, *ignore_hydrogen=False*, *zero_hydrogen=True*, *complete_fragments=True*, *dump=True*, *h5_fn='afir.h5'*, *h5_group_name='afir'*, ***kwargs*)

Artifical Force Induced Reaction calculator.

Currently, there are no automated drivers to run large-scale AFIR calculations with many different initial orientations and/or increasing collision energy parameter . Nontheless, selected AFIR calculations can be carried out by hand. After convergence, artificial potential & forces, as well as real energies and forces can be plotted with 'pysisplot --afir'. The highest energy point along the AFIR path can then be selected for a subsequent TS-optimization, e.g. via 'pysistrj --get [index] optimzation.trj'.

Future versions of pysisyphus may provide drivers for more automatted AFIR calculations.

Parameters

- **calculator** (*Calculator*) -- Actual QC calculator that provides energies and its derivatives, that are modified by the AFIR calculator, e.g., ORCA or Psi4.
- **fragment_indices** (List[List[int]]) -- List of lists of integers, specifying the separate fragments. If the indices in theses lists don't comprise all atoms in the molecule, the reamining indices will be added as a separate fragment. If a AFIR calculation is carried out with 2 fragments and 'complete_fragments' is True (see below) it is enough to specify only the indices of one fragment, e.g., for a system of 10 atoms 'fragment_indices=[[0,1,2,3]]' is enough. The second system will be set up automatically with indices [4,5,6,7,8,9].
- **gamma** (Union[_SupportsArray[dtype[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]]) -- Collision energy parameter in au. For 2 fragments it can be a single integer, while for > 2 fragments a list of gammas must be given, specifying the pair-wise collision energy parameters. For 3 fragments 3 gammas must be given [_01, _02, _12], for 4 fragments 6 gammas would be required [_01, _02, _03, _12, _13, _23] and so on.
- **rho** (Union[_SupportsArray[dtype[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]], default: 1) -- Direction of the artificial force, either 1 or -1. The same comments as for gamma apply. For 2 fragments a single integer is enough, for > 2 fragments a list of rhos must be given (see above). For rho=1 fragments are pushed together, for rho=-1 fragments are pulled apart.
- **p** (int, default: 6) -- Exponent p used in the calculation of the weight function . Defaults to 6 and probably does not have to be changed.
- **ignore_hydrogen** (bool, default: False) -- Whether hydrogens are ignored in the calculation of the artificial force. All weights between atom pairs containing hydrogen will be set to 0.
- **zero_hydrogen** (bool, default: True) -- Whether to use 0.0 as covalent radius for hydrogen in the weight function. Compared to 'ignore_hydrogen', which results in zero weights for all atom pairs involving hydrogen, 'zero_hydrogen' may be non-zero, depending on the covalent radius of the second atom in the pair.

- **complete_fragments** (bool, default: True) -- Whether an incomplete specification in 'fragment_indices' is automatically completed.
- **dump** (bool, default: True) -- Whether an HDF5 file is created.
- **h5_fn** (str, default: 'afir.h5') -- Filename of the HDF5 file used for dumping.
- **h5_group_name** (str, default: 'afir') -- HDF5 group name used for dumping.
- ****kwargs** -- Keyword arguments passed to the Calculator baseclass.

afir_fd_hessian_wrapper(*coords3d*, *afir_grad_func*)

property charge

dump_h5(*atoms*, *coords*, *results*)

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

init_h5_group(*atoms*, *max_cycles=None*)

log_fragments()

property mult

set_atoms_and_funcs(*atoms*, *coords*)

Initially atoms was also an argument to the constructor of AFIR. I removed it so creation becomes easier. The first time a calculation is requested with a proper atom set everything is set up (cov. radii, afir function and corresponding gradient). Afterwards there is only a check if atoms != None and it is expected that all functions are properly set.

fragment_indices can also be incomplete w.r.t. to the number of atoms. If the sum of the specified fragment atoms is less than the number of atoms present then all remaining unspecified atoms will be gathered in one fragment.

write_fragment_geoms(*atoms*, *coords*)

exception pysisyphus.calculators.AFIR.**CovRadiiSumZero**

Bases: Exception

pysisyphus.calculators.AFIR.**afir_closure**(*fragment_indices*, *cov_radii*, *gamma*, *rho=1*, *p=6*, *prefactor=1.0*, *logger=None*)

rho=1 pushes fragments together, rho=-1 pulls fragments apart.

pysisyphus.calculators.AFIR.**get_data_model**(*atoms*, *max_cycles*)

6.6.8 ONIOM

class pysisyphus.calculators.ONIOMv2.**LayerCalc**(*models*, *total_size*, *parent_layer_calc=None*)

Bases: object

property charge

do_parent(*with_parent*)

static energy_from_results(*model_energies*, *parent_energy=None*)

get_energy(*atoms*, *coords*, *with_parent=True*)

get_forces(*atoms*, *coords*, *with_parent=True*)

get_hessian(*atoms*, *coords*, *with_parent=True*)

property mult

run_calculations(*atoms*, *coords*, *method*)

class pysisyphus.calculators.ONIOMv2.**Link**(*ind*, *parent_ind*, *atom*, *g*)

Bases: tuple

atom

Alias for field number 2

g

Alias for field number 3

ind

Alias for field number 0

parent_ind

Alias for field number 1

class pysisyphus.calculators.ONIOMv2.**Model**(*name*, *calc_level*, *calc*, *parent_name*, *parent_calc_level*, *parent_calc*, *atom_inds*, *parent_atom_inds*, *use_link_atoms=True*)

Bases: object

as_calculator(*cap=False*)

as_geom(*all_atoms*, *all_coords*)

capped_atoms_coords(*all_atoms*, *all_coords*)

create_bond_vec_getters(*atoms*)

create_links(*atoms*, *coords*, *debug=False*)

get_energy(*atoms*, *coords*, *point_charges=None*, *parent_correction=True*, *cap=True*)

get_forces(*atoms*, *coords*, *point_charges=None*, *parent_correction=True*, *cap=True*)

get_hessian(*atoms*, *coords*, *point_charges=None*, *parent_correction=True*, *cap=True*)

get_jacobian()

get_sparse_jacobian()

log(*message=''*)

parse_charges()

class pysisyphus.calculators.ONIOMv2.ModelDummyCalc(*model*, *cap=False*)

Bases: object

get_energy(*atoms*, *coords*)

get_forces(*atoms*, *coords*)

class pysisyphus.calculators.ONIOMv2.ONIOM(*calcs*, *models*, *geom*, *layers=None*, *embedding=''*, *real_key='real'*, *use_link_atoms=True*, **args*, ***kwargs*)

Bases: *Calculator*

__init__(*calcs*, *models*, *geom*, *layers=None*, *embedding=''*, *real_key='real'*, *use_link_atoms=True*, **args*, ***kwargs*)

layer: list of models
len(layer) == 1: normal ONIOM, len(layer) >= 1: multicenter ONIOM.

model:
(sub)set of all atoms that resides in a certain layer and has a certain calculator.

atom_inds_in_layer(*index*, *exclude_inner=False*)

Returns list of atom indices in layer at index.

Atoms that also appear in inner layer can be excluded on request.

Parameters

- **index** (*int*) -- pasd
- **exclude_inner** (*bool, default=False, optional*) -- Whether to exclude atom indices that also appear in inner layers.

Returns
atom_indices -- List containing the atom indices in the selected layer.

Return type
list

calc_layer(*atoms*, *coords*, *index*, *parent_correction=True*)

property charge

embeddings = {'': '', 'electronic': 'Electronic embedding', 'electronic_rc': 'Electronic embedding with redistributed charges', 'electronic_rcd': 'Electronic embedding with redistributed charges and dipoles'}

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_layer_calc(*layer_ind*)

property model_iter

property mult

run_calculation(*atoms*, *coords*)

run_calculations(*atoms*, *coords*, *method*)

pysisyphus.calculators.ONIOMv2.**atom_inds_to_cart_inds**(*atom_inds*)

pysisyphus.calculators.ONIOMv2.**cap_fragment**(*atoms*, *coords*, *fragment*, *link_atom='H'*, *g=None*)

pysisyphus.calculators.ONIOMv2.**get_embedding_charges**(*embedding*, *layer*, *parent_layer*, *coords3d*)

pysisyphus.calculators.ONIOMv2.**get_g_value**(*atom*, *parent_atom*, *link_atom*)

6.6.9 Dimer

class pysisyphus.calculators.Dimer.**Dimer**(*calculator*, **args*, *N_raw=None*, *length=0.0189*, *rotation_max_cycles=15*, *rotation_method='fourier'*, *rotation_thresh=0.0001*, *rotation_tol=1*, *rotation_max_element=0.001*, *rotation_interpolate=True*, *rotation_disable=False*, *rotation_disable_pos_curv=True*, *rotation_remove_trans=True*, *trans_force_f_perp=True*, *bonds=None*, *N_hessian=None*, *bias_rotation=False*, *bias_translation=False*, *bias_gaussian_dot=0.1*, *seed=None*, *write_orientations=True*, *forward_hessian=True*, ***kwargs*)

Bases: *Calculator*

property C

Shortcut for the curvature.

property N

add_gaussian(*atoms*, *center*, *N*, *height=0.1*, *std=0.0529*, *max_cycles=50*, *dot_ref=None*)

property can_bias_f0

property can_bias_f1

property coords0

property coords1

curvature(*f1*, *f2*, *N*)

Curvature of the mode represented by the dimer.

direct_rotation(*optimizer*, *prev_step*)

do_dimer_rotations(*rotation_thresh=None*)

property energy0

property f0

property f1

property f1_bias

property f2

Never calculated explicitly, but estimated from f0 and f1.

fourier_rotation(*optimizer*, *prev_step*)

get_N_raw_from_hessian(*h5_fn*, *root=0*)

get_forces(*atoms*, *coords*)

Meant to be extended.

get_gaussian_energies(*coords*, *sum_=True*)

get_gaussian_forces(*coords*, *sum_=True*)

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

remove_translation(*displacement*)

property rot_force

rotate_coords1(*rad*, *theta*)

Rotate dimer and produce new coords1.

set_N_raw(*coords*)

property should_bias_f0

May lead to calculation of f0 and/or f1 if present!

property should_bias_f1

May lead to calculation of f0 and/or f1 if present!

update_orientation(*coords*)

class pysisyphus.calculators.Dimer.Gaussian(*height*, *center*, *std*, *N*)

Bases: object

energy(*R*, *height=None*)

forces(*R*, *height=None*)

exception pysisyphus.calculators.Dimer.RotationConverged

Bases: Exception

6.7 Pure Python calculators

6.7.1 Sympy 2D Potentials

class pysisyphus.calculators.AnaPotBase.**AnaPotBase**(*V_str*, *scale=1.0*, *xlim=(-1, 1)*, *ylim=(-1, 1)*, *levels=None*, *use_sympify=True*, *minima=None*, *saddles=None*)

Bases: *Calculator*

anim_coords(*coords*, *interval=50*, *show=False*, *title_func=None*)

anim_opt(*opt*, *energy_profile=False*, *colorbar=False*, *figsize=(8, 6)*, *show=False*)

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

classmethod get_geom(*coords*, *atoms=('X',)*, *V_str=None*, *calc_kwargs=None*, *geom_kwargs=None*)

get_geoms_from_stored_coords(*coords_list*, *i=None*, *calc_kwargs=None*, *geom_kwargs=None*)

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_minima(*i=None*, *calc_kwargs=None*, *geom_kwargs=None*)

get_path(*num*, *minima_inds=None*)

get_saddles(*i=None*, *calc_kwargs=None*, *geom_kwargs=None*)

plot(*levels=None*, *show=False*, ***figkwargs*)

plot3d(*levels=None*, *show=False*, *zlim=None*, *vmin=None*, *vmax=None*, *resolution=100*, *rcount=50*, *ccount=50*, *nan_above=None*, *init_view=None*, *colorbar=False*, ***figkwargs*)

plot_coords(*xs*, *ys*, *enum=True*, *show=False*, *title=None*)

plot_eigenvalue_structure(*grid=50*, *levels=None*, *show=False*)

plot_geoms(*geoms*, ***kwargs*)

plot_irc(*irc*, **args*, ***kwargs*)

plot_opt(*opt*, **args*, ***kwargs*)

statistics()

6.7.2 Lennard-Jones

class pysisyphus.calculators.LennardJones.**LennardJones**(*sigma=1.8897261251*, *epsilon=1*, *rc=None*)

Bases: *Calculator*

calculate(*coords3d*)

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

6.7.3 FakeASE

class pysisyphus.calculators.FakeASE.**FakeASE**(*calc*)

Bases: object

get_atoms_coords(*atoms*)

get_forces(*atoms=None*)

get_potential_energy(*atoms=None*)

6.7.4 TIP3P

class pysisyphus.calculators.TIP3P.**TIP3P**(*rc=9.44863062728914*)

Bases: *Calculator*

Transferable Intermolecular Potential 3 Point

aHOH = 104.52

calculate(*coords3d*)

charges

coulomb_energy = (multiple of elem. charge * multiple of elem. charge)
/ (distance in bohr) * 1 / (4 * pi * vacuum permittivity)

coulomb_prefactor converts everything to atomic units and it is ... drum roll ... 1. from scipy.constants import value as pcval self.coulomb_prefactor = (1 / (4 * np.pi) * pcval("elementary charge")**2

/ pcval("Hartree energy") / pcval("Bohr radius") / pcval("vacuum electric permittivity")

)

coulomb(*coords3d*)

epsilon = 0.0002423919586315716

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

```
qH = 0.417

qO = -0.834

rOH = 1.8088458464917874

sigma = 5.953790025507198
```

CHAPTER

SEVEN

MINIMIZATION

Searching for minimum energy geometries of molecules is preferably done using second order methods that employ hessian information. Using the plain hessian H for step prediction through $p = -H^{-1}g$, with g being the gradient vector, may yield erroneous uphill steps when H has negative eigenvalues.

This can be understood by calculating the step in the basis of the hessian eigenvectors V.

$$\tilde{H} = V^T H V \tag{7.1}$$

$$\tilde{g} = V^T g \tag{7.2}$$

$$\tilde{p}_i = -\frac{\tilde{g}_i}{\tilde{H}_{ii}} \tag{7.3}$$

(7.4)

$\tilde{H}, \tilde{g}$ and $\tilde{p}$ are transformed into the eigenvector basis and the subscript i indicates the component belongs to the i-th eigenvalue (-vector) of H. As the gradient always points into the direction of greater function values dividing it by a negative eigenvalues $\tilde{H}_{ii}$ will lead to a step in uphill direction along the i-th eigenvector. The step in the original basis is obtained by a simple back-transformation:

$$p = V\tilde{p}$$

A step in downhill direction can be ensured by introducing an appropriate shift parameter λ that must be smaller than the smallest eigenvalue of H:

$$\tilde{p}_i = -\frac{\tilde{g}_i}{\tilde{H}_{ii} - \lambda}$$

In *pysisyphus* the shift parameter λ is obtained by the Rational Function Optimization approach.

When the root-mean-square of the predicted step p drops below a certain threshold (default is 0.0025 au or rad) controlled GDIIS is tried. If GDIIS fails or is not yet possible a polynomial line-search is conducted. Using the latest two energies and the projection of the latest two gradients onto the last step the coefficients of a cubic polynomial are defined unambiguously. By requiring $\mathrm{d}^2 f(x)/\mathrm{d}x^2 >= 0$ and the equality holding at exactly one point also a quartic polynomial can be determined.

For now line-searches using a cubic polynomial are disabled as they seem to degrade the optimizer performance, so only the constrained quartic polynomial is tried. By also using projected hessian information the coefficients of a quintic polynomial can be determined, but this also seems to be inferior compared to the constrained quartic polynomial.

Convergence is indicated when the root-mean-square and the maximum value of the gradient and step drop below a certain threshold. It is not uncommon that the gradient convergence is achieved before step convergence. By using *overachieve_factor: [n, float > 1]* in the YAML input convergence will be signalled, when gradient convergence is overachieved by factor *n*.

7.1 YAML Example

Below you can find an example YAML-input including the most important options that the user may want to modify when using the RFOptimizer.

```yaml
opt:
 type: rfo                      # Optimization algorithm
 max_cycles: 50                 # Maximum number of optimization cycles

 overachieve_factor: 2          # Indicate convergence, regardless of the
                                # proposed step when max(grad) and rms(grad)
                                # are overachieved by factor [n]

 do_hess: True                  # Calculate the hessian at the final geometry
                                # after the optimization.

 #hessian_recalc: None          # Recalculate exact hessian every n-th cylce

 #hessian_recalc_adapt: None    # Expects a float. Recalculate exact hessian
                                # whenever the gradient norms drops below
                                # 1/[n] of the gradient norm at the last hessian
                                # recalculation.

 #hessian_init: fischer         # Type of model hessian. Other options are: 'calc,
                                # simple, xtb, lindh, swart, unit'

 #hessian_update: bfgs          # Hessian-update. Other options are: 'flowchart,
                                # damped_bfgs, bofill'. bofill is not recommended
                                # for minimum searches.
 #small_eigval_thresh: 1e-8     # Neglect eigenvalues and corresponding eigenvectors
                                # below this threshold.

 #max_micro_cycles: 50          # No. of micro cycles for the RS-variants. Does not apply
                                # to TRIM.

 #trust_radius: 0.3             # Initial trust radius.
 #trust_max: 1.0                # Max. trust radius
 #trust_min: 0.1                # Min. trust radius

 #line_search: True             # Do line search

 #gdiis_thresh: 0.0025          # May do GDIIS if rms(step) falls below this threshold
 #gediis_thresh: 0.01           # May do GEDIIS if rms(grad) falls below this threshold
 #gdiis: True                   # Do controlled GDIIS after 'gdiis_thresh' is reached
 #gediis: False                 # Do GEDIIS after 'gediis_thresh' is reached

calc:
 type: turbomole
 control_path: control_path_pbe0_def2svp_s1     # Path to the prepared calculation
 track: True                                    # Activate excited state tracking
 ovlp_type: tden                                # Track with transition density matrix␣
↪overlaps
 charge: 0
```

(continues on next page)

(continued from previous page)

```
 mult: 1
 pal: 4
 mem: 2000

geom:
 type: redund
 fn: cytosin.xyz
```

Further examples for optimizations from *.yaml* input can be found here.

7.2 Convergence Thresholds

Optimization convergence is indicated when four critera are met. Each cycle the root-mean-square and the absolute maximum of force and proposed step vectors are checked. The default criteria in pysisyphus correspond to the *opt=loose* in Gaussian. Different thresholds can easily be requested by supplying the respective keyword in the YAML input.

Convergence is also indicated, when max(force) and rms(force) are overachieved, by a certain factor (*overachieve_factor*, default=3), similar to ORCAs optimizer. Further keywords, controlling the convergence check are found below.

```
opt:
 ...
 thresh: gau_loose           # See table below for different thresholds.
 overachieve_factor: 3
 rms_force: null             # Derive four critera from this value
 rms_force_only: False       # Only check rms(force)
 max_force_only: False       # Only check max(force)
 # Chain-of-States specific
 coord_diff_thresh: 0.01     # Terminate on insignificant coordinate change
 reparam_thresh: 0.001       # Terminate on insignificant coordinate change upon␣
↪reparametrization
 ...
```

Threshold	max(force)	rms(forces)	max(step)	rms(step)	Comment
gau_loose	2.5e-3	1.7e-3	1.0e-2	6.7e-3	default
gau	4.5e-4	3.0e-4	1.8e-3	1.2e-3	
gau_tight	1.5e-5	1.0e-5	6.0e-5	4.0e-5	
gau_vtight	2.0e-6	1.0e-6	6.0e-6	4.0e-6	
baker	3.0e-4	2.0e-4	3.0e-4	2.0e-4	energy difference and step are also checked
never	2.0e-6	1.0e-6	6.0e-6	4.0e-6	dummy thresholds; never converges

7.3 Available Optimizers

Base class for optimizers that utilize Hessian information.

class pysisyphus.optimizers.Optimizer.**ConvInfo**(*cur_cycle*, *energy_converged*, *max_force_converged*, *rms_force_converged*, *max_step_converged*, *rms_step_converged*, *desired_eigval_structure*)

Bases: object

cur_cycle: int

desired_eigval_structure: bool

energy_converged: bool

get_convergence()

is_converged()

max_force_converged: bool

max_step_converged: bool

rms_force_converged: bool

rms_step_converged: bool

class pysisyphus.optimizers.Optimizer.**Optimizer**(*geometry*, *thresh='gau_loose'*, *max_step=0.04*, *max_cycles=150*, *min_step_norm=1e-08*, *assert_min_step=True*, *rms_force=None*, *rms_force_only=False*, *max_force_only=False*, *force_only=False*, *converge_to_geom_rms_thresh=0.05*, *align=False*, *align_factor=1.0*, *dump=False*, *dump_restart=False*, *print_every=1*, *prefix=''*, *reparam_thresh=0.001*, *reparam_check_rms=True*, *reparam_when='after'*, *overachieve_factor=0.0*, *check_eigval_structure=False*, *restart_info=None*, *check_coord_diffs=True*, *coord_diff_thresh=0.01*, *fragments=None*, *monitor_frag_dists=0*, *out_dir='.'*, *h5_fn='optimization.h5'*, *h5_group_name='opt'*)

Bases: object

__init__(*geometry*, *thresh='gau_loose'*, *max_step=0.04*, *max_cycles=150*, *min_step_norm=1e-08*, *assert_min_step=True*, *rms_force=None*, *rms_force_only=False*, *max_force_only=False*, *force_only=False*, *converge_to_geom_rms_thresh=0.05*, *align=False*, *align_factor=1.0*, *dump=False*, *dump_restart=False*, *print_every=1*, *prefix=''*, *reparam_thresh=0.001*, *reparam_check_rms=True*, *reparam_when='after'*, *overachieve_factor=0.0*, *check_eigval_structure=False*, *restart_info=None*, *check_coord_diffs=True*, *coord_diff_thresh=0.01*, *fragments=None*, *monitor_frag_dists=0*, *out_dir='.'*, *h5_fn='optimization.h5'*, *h5_group_name='opt'*)

Optimizer baseclass. Meant to be subclassed.

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.

- **thresh** (Literal['gau_loose', 'gau', 'gau_tight', 'gau_vtight', 'baker', 'never'], default: 'gau_loose') -- Convergence threshold.
- **max_step** (float, default: 0.04) -- Maximum absolute component of the allowed step vector. Utilized in optimizers that don't support a trust region or line search.
- **max_cycles** (int, default: 150) -- Maximum number of allowed optimization cycles.
- **min_step_norm** (float, default: 1e-08) -- Minimum norm of an allowed step. If the step norm drops below this value a ZeroStepLength-exception is raised. The unit depends on the coordinate system of the supplied geometry.
- **assert_min_step** (bool, default: True) -- Flag that controls whether the norm of the proposed step is check for being too small.
- **rms_force** (Optional[float], default: None) -- Root-mean-square of the force from which user-defined thresholds are derived. When 'rms_force' is given 'thresh' is ignored.
- **rms_force_only** (bool, default: False) -- When set, convergence is signalled only based on rms(forces).
- **max_force_only** (bool, default: False) -- When set, convergence is signalled only based on max(**|forces|**).
- **force_only** (bool, default: False) -- When set, convergence is signalled only based on max(**|forces|**) and rms(forces).
- **converge_to_geom_rms_thresh** (float, default: 0.05) -- Threshold for the RMSD with another geometry. When the RMSD drops below this threshold convergence is signalled. Only used with Growing Newton trajectories.
- **align** (bool, default: False) -- Flag that controls whether the geometry is aligned in every step onto the coordinates of the previous step. Must not be used with internal coordinates.
- **align_factor** (float, default: 1.0) -- Factor that controls the strength of the alignment. 1.0 means full alignment, 0.0 means no alignment. The factor mixes the rotation matrix of the alignment with the identity matrix.
- **dump** (bool, default: False) -- Flag to control dumping/writing of optimization progress to the filesystem
- **dump_restart** (bool, default: False) -- Flag to control whether restart information is dumped to the filesystem.
- **print_every** (int, default: 1) -- Report optimization progress every nth cycle.
- **prefix** (str, default: '') -- Short string that is prepended to several files created by the optimizer. Allows distinguishing several optimizations carried out in the same directory.
- **reparam_thresh** (float, default: 0.001) -- Controls the minimal allowed similarity between coordinates after two successive reparametrizations. Convergence is signalled if the coordinates did not change significantly.
- **reparam_check_rms** (bool, default: True) -- Whether to check for (too) similar coordinates after reparametrization.
- **reparam_when** (Optional[Literal['before', 'after']], default: 'after') -- Reparametrize before or after calculating the step. Can also be turned off by setting it to None.
- **overachieve_factor** (float, default: 0.0) -- Signal convergence when max(forces) and rms(forces) fall below the chosen threshold, divided by this factor. Convergence of max(step) and rms(step) is ignored.

- **check_eigval_structure** (bool, default: False) -- Check the eigenvalues of the modes we maximize along. Convergence requires them to be negative. Useful if TS searches are started from geometries close to a minimum.
- **restart_info** (default: None) -- Restart information. Undocumented.
- **check_coord_diffs** (bool, default: True) -- Whether coordinates of chain-of-sates images are checked for being too similar.
- **coord_diff_thresh** (float, default: 0.01) -- Unitless threshold for similary checking of COS image coordinates. The first image is assigned 0, the last image is assigned to 1.
- **fragments** (Optional[Tuple], default: None) -- Tuple of lists containing atom indices, defining two fragments.
- **monitor_frag_dists** (int, default: 0) -- Monitor fragment distances for N cycles. The optimization is terminated when the interfragment distances falls below the initial value after N cycles.
- **out_dir** (str, default: '.') -- String poiting to a directory where optimization progress is dumped.
- **h5_fn** (str, default: 'optimization.h5') -- Basename of the HDF5 file used for dumping.
- **h5_group_name** (str, default: 'opt') -- Groupname used for dumping of this optimization.

check_convergence(*step=None*, *multiple=1.0*, *overachieve_factor=None*)

Check if the current convergence of the optimization is equal to or below the required thresholds, or a multiple thereof. The latter may be used in initiating the climbing image.

dump_restart_info()

final_summary()

fit_rigid(***, *vectors=None*, *vector_lists=None*, *hessian=None*)

get_path_for_fn(*fn*, *with_prefix=True*)

get_restart_info()

log(*message*, *level=50*)

make_conv_dict(*key*, *rms_force=None*, *rms_force_only=False*, *max_force_only=False*, *force_only=False*)

abstract optimize()

postprocess_opt()

prepare_opt()

print_opt_progress(*conv_info*)

procrustes()

Wrapper for procrustes that passes additional arguments along.

report_conv_thresholds()

run()

scale_by_max_step(*steps*)

set_restart_info(*restart_info*)

write_cycle_to_file()

write_image_trjs()

write_results()

write_to_out_dir(*out_fn*, *content*, *mode='w'*)

pysisyphus.optimizers.Optimizer.**get_data_model**(*geometry*, *is_cos*, *max_cycles*)

class pysisyphus.optimizers.HessianOptimizer.**HessianOptimizer**(*geometry*, *trust_radius=0.5*, *trust_update=True*, *trust_min=0.1*, *trust_max=1*, *max_energy_incr=None*, *hessian_update='bfgs'*, *hessian_init='fischer'*, *hessian_recalc=None*, *hessian_recalc_adapt=None*, *hessian_xtb=False*, *hessian_recalc_reset=False*, *small_eigval_thresh=1e-08*, *line_search=False*, *alpha0=1.0*, *max_micro_cycles=25*, *rfo_overlaps=False*, ***kwargs*)

Bases: *Optimizer*

__init__(*geometry*, *trust_radius=0.5*, *trust_update=True*, *trust_min=0.1*, *trust_max=1*, *max_energy_incr=None*, *hessian_update='bfgs'*, *hessian_init='fischer'*, *hessian_recalc=None*, *hessian_recalc_adapt=None*, *hessian_xtb=False*, *hessian_recalc_reset=False*, *small_eigval_thresh=1e-08*, *line_search=False*, *alpha0=1.0*, *max_micro_cycles=25*, *rfo_overlaps=False*, ***kwargs*)

Baseclass for optimizers utilizing Hessian information.

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.
- **trust_radius** (`float`, default: `0.5`) -- Initial trust radius in whatever unit the optimization is carried out.
- **trust_update** (`bool`, default: `True`) -- Whether to update the trust radius throughout the optimization.
- **trust_min** (`float`, default: `0.1`) -- Minimum trust radius.
- **trust_max** (`float`, default: `1`) -- Maximum trust radius.
- **max_energy_incr** (`Optional[float]`, default: `None`) -- Maximum allowed energy increased after a faulty step. Optimization is aborted when the threshold is exceeded.
- **hessian_update** (`Literal['none', None, False, 'bfgs', 'damped_bfgs', 'flowchart', 'bofill', 'ts_bfgs', 'ts_bfgs_org', 'ts_bfgs_rev']`, default: `'bfgs'`) -- Type of Hessian update. Defaults to BFGS for minimizations and Bofill for saddle point searches.

- **hessian_init** (Literal['calc', 'unit', 'fischer', 'lindh', 'simple', 'swart', 'xtb', 'xtb1', 'xtbff'], default: 'fischer') -- Type of initial model Hessian.
- **hessian_recalc** (Optional[int], default: None) -- Recalculate exact Hessian every n-th cycle instead of updating it.
- **hessian_recalc_adapt** (Optional[float], default: None) -- Use a more flexible scheme to determine Hessian recalculation. Undocumented.
- **hessian_xtb** (bool, default: False) -- Recalculate the Hessian at the GFN2-XTB level of theory.
- **hessian_recalc_reset** (bool, default: False) -- Whether to skip Hessian recalculation after reset. Undocumented.
- **small_eigval_thresh** (float, default: 1e-08) -- Threshold for small eigenvalues. Eigenvectors belonging to eigenvalues below this threshold are discardewd.
- **line_search** (bool, default: False) -- Whether to carry out a line search. Not implemented by a subclassing optimizers.
- **alpha0** (float, default: 1.0) -- Initial alpha for restricted-step (RS) procedure.
- **max_micro_cycles** (int, default: 25) -- Maximum number of RS iterations.
- **rfo_overlaps** (bool, default: False) -- Enable mode-following in RS procedure.
- ****kwargs** -- Keyword arguments passed to the Optimizer baseclass.

filter_small_eigvals(*eigvals*, *eigvecs*, *mask=False*)

get_alpha_step(*cur_alpha*, *rfo_eigval*, *step_norm*, *eigvals*, *gradient*)

get_augmented_hessian(*eigvals*, *gradient*, *alpha=1.0*)

static get_newton_step(*eigvals*, *eigvecs*, *gradient*)

get_newton_step_on_trust(*eigvals*, *eigvecs*, *gradient*, *transform=True*)

Step on trust-radius.

See Nocedal 4.3 Iterative solutions of the subproblem

get_rs_step(*eigvals*, *eigvecs*, *gradient*, *name='RS'*)

static get_shifted_step_trans(*eigvals*, *gradient_trans*, *shift*)

get_step_func(*eigvals*, *gradient*, *grad_rms_thresh=0.01*)

housekeeping()

Calculate gradient and energy. Update trust radius and hessian if needed. Return energy, gradient and hessian for the current cycle.

log_negative_eigenvalues(*eigvals*, *pre_str=''*)

prepare_opt(*hessian_init=None*)

property prev_eigvec_max

property prev_eigvec_min

static quadratic_model(*gradient*, *hessian*, *step*)

reset()

rfo_dict = {'max': (-1, 'max'), 'min': (0, 'min')}

static rfo_model(*gradient*, *hessian*, *step*)

save_hessian()

set_new_trust_radius(*coeff*, *last_step_norm*)

solve_rfo(*rfo_mat*, *kind='min'*, *prev_eigvec=None*)

update_hessian()

update_trust_radius()

pysisyphus.optimizers.HessianOptimizer.**dummy_hessian_update**(*H*, *dx*, *dg*)

class pysisyphus.optimizers.RFOptimizer.**RFOptimizer**(*geometry*, *line_search=True*, *gediis=False*, *gdiis=True*, *gdiis_thresh=0.0025*, *gediis_thresh=0.01*, *gdiis_test_direction=True*, *max_micro_cycles=25*, *adapt_step_func=False*, ***kwargs*)

Bases: *HessianOptimizer*

__init__(*geometry*, *line_search=True*, *gediis=False*, *gdiis=True*, *gdiis_thresh=0.0025*, *gediis_thresh=0.01*, *gdiis_test_direction=True*, *max_micro_cycles=25*, *adapt_step_func=False*, ***kwargs*)

Rational function Optimizer.

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.
- **line_search** (bool, default: True) -- Whether to carry out implicit line searches.
- **gediis** (bool, default: False) -- Whether to enable GEDIIS.
- **gdiis** (bool, default: True) -- Whether to enable GDIIS.
- **gdiis_thresh** (float, default: 0.0025) -- Threshold for rms(forces) to enable GDIIS.
- **gediis_thresh** (float, default: 0.01) -- Threshold for rms(step) to enable GEDIIS.
- **gdiis_test_direction** (bool, default: True) -- Whether to the overlap of the RFO step and the GDIIS step.
- **max_micro_cycles** (int, default: 25) -- Number of restricted-step microcycles. Disabled by default.
- **adapt_step_func** (bool, default: False) -- Whether to switch between shifted Newton and RFO-steps.
- ****kwargs** -- Keyword arguments passed to the Optimizer/HessianOptimizer baseclass.

optimize()

postprocess_opt()

CHAPTER

EIGHT

MINIMIZATION WITH MICROITERATIONS

Pysisyphus allows layered optimizations using microiterations via the *LayerOpt* class. Required energies and its derivatives (gradients, Hessians) can either be calculated with pysisyphus' own ONIOM implementation or supplied through Unix sockets via the i-PI protocol.

In ONIOM calculations a small (model) system is treated using a high level of theory, while its surroundings (real system) are described at a lower, more economic level of theory. There may also be several (optional) intermediate layers. Please see the figure below.

Fig. 8.1: General structure of a n-layer ONIOM calculation in pysisyphus. Every box corresponds to a layer. Indices of the respective layers are given as red numbers in the top right corner of each layer.

Searching stationary points of such systems using classical optimization approaches with simultanous relaxation of model and real system can be computationally quite costly. In cases where the model system converges before the real system, many unnecessary model system energy and gradients evaluations may be required to fully relax the total system, as one step requires gradient evaluations in all layers. Also, it may be prohibitively expensive to optimize the whole system using internal coordinates, so it would be desireable to treat only the model system using internal coordinates.

Optimization with microiterations offers a way to potentially reduce the number of costly calculations in the innermost layer. By fully relaxing the outer layers before taking a step in the innermost layer, superfluous gradient evaluations can be avoided. As the overall calculation time of an ONIOM energy/gradient is usually dominated by the innermost layer, great compuational saving can by realized by decreasing the required number of such calculations.

In the present approach, outer layers are optimized in Cartesian coordinates with an economic optimizer like (preconditioned) limited memory BFGS (L-BFGS) or conjugate gradient (CG). As L-BFGS avoids operations with steep scaling costs like matrix diagonalization, relaxation of large system comprising thousands of atoms becomes possible with negligible computational overhead. In contrast to the outer layers, the inner layer is usually optimized using internal coordinates and an optimizer that utilizes an explicit Hessian matrix, as this often allows for the most efficient optimizations.

As energy and gradient evaluations in the outer layers should be cheap the inferior performance of optimizations in Cartesian coordinates should not lead to an overall runtime increase.

8.1 YAML input

When pysisyphus' native ONIOM calculator is to be used, appropriate input has to be given in the `calc:` section, that is, layer composition and the respective calculators. If energies & gradients are sent via sockets and the i-PI-protocol `calc:` can be left empty but the appropriate socket addresses have to specified later in the `opt:` section (see below). A separate socket/address is required for each layer.

Independent of the actual layer, pysisyphus always expects vectorial quantities with appropriate shapes of the total system. The expected size of a force vector for a system comprising `N` atoms is `3N`. Vector components not belonging to frozen atoms not belonging to a given layer are ignored and may be zero. **It is expected, that the full valid ONIOM forces for the total system are sent with the innermost layer**. These forces are then also used to check for optimization convergence.

A layered optimization is requested via `type: layer` in the `opt:` section. If no further input is given, appropriate defaults will be used. Micoiterations for the outer layers `0 to n-2` will be carried out using Cartesian coordinates and regularized L-BFGS, while the innermost layer is optimized in internal coordinates (`type: tric`) using rational function optimization (`type: rfo`).

If a more fine grained control is desired each layer can be controlled independently by further keywords in the `layers:` list. When `layers:` is present in the input all layers must be specified, but empty list entries are possible (see the lines only containing a comment in the example below). **One must start with the outmost layer**; the last list item corresponds to the innermost layer. Geometry and/or optimizer setup of each layer is controlled by the `geom:` and `opt:` sections in the list of layers with the usual keywords.

The usual keywords to control the optimization are supported, e.g., `max_cycles`, `thresh` etc.

```
geom:
 type:
  cartesian
 fn: [filename of initial geometry]
calc:
 type:
  oniom:
   ...
opt:
 type:
  layer:
   [layers:
     - # dummy for layer 0 (real system)
     - # dummy for layer 1 layer
     - # ...
     # - # dummy for layer n-1 (model system)
     - geom:
        type:
         cartesian:
       opt:
        type:
         lbfgs:
   ]
 # thresh: gau_loose
 # max_cycles: 150
```

Below you can find a full example for the ONIOM2-optimization of Hexaphenylethane using pysisyphus' ONIOM implementation.

8.2 Example using pysisyphus' ONIOM

```
# 2-layer ONIOM optimization of Hexaphenylethan. The model
# system comprises the two carbons of the central ethan moiety.
# The model system is described GFN2-XTB while the real system is
# treated using GFN-FF.
geom:
 type: cart
 fn: |
  68

  C           1.2639    0.6380    0.4785
  C           2.1638    1.9009    0.1349
  C           2.4167    2.3403   -1.2006
  C           3.2096    3.4576   -1.4755
  C           2.7467    2.7036    1.1595
  C           3.5368    3.8191    0.8739
  C           3.7741    4.1922   -0.4416
  H           2.0100    1.8572   -2.0700
  H           3.3815    3.7580   -2.5014
  H           2.5944    2.5089    2.1979
  H           3.9618    4.4009    1.6820
  H           4.3848    5.0587   -0.6598
  C           0.7891    0.0052   -0.8980
  C          -1.0360   -0.5225   -2.5243
  C          -0.0432   -1.1375   -3.4197
  C           1.2448   -1.1452   -3.0690
  C          -0.6577    0.0144   -1.3528
  H          -1.4492    0.4671   -0.7765
  H          -2.0793   -0.4955   -2.8238
  H          -0.3570   -1.5703   -4.3642
  C           1.6732   -0.5526   -1.7810
  H           1.9838   -1.5872   -3.7297
  H           2.7287   -0.5813   -1.6050
  C          -0.0387    1.1726    1.2081
  C          -0.2929    2.5599    1.4341
  C          -1.0635    0.2863    1.6524
  C          -2.2174    0.7426    2.2933
  C          -2.4082    2.0998    2.5118
  C          -1.4505    3.0060    2.0772
  H           0.3724    3.3423    1.1178
  H          -1.0153   -0.7691    1.5014
  H          -2.9711    0.0356    2.6161
  H          -1.6065    4.0659    2.2335
  H          -3.3046    2.4506    3.0066
  C           2.1016   -0.4883    1.4312
  C           3.4003   -1.0269    0.6933
  C           4.4239   -0.1425    0.2420
  C           5.5730   -0.6012   -0.4056
  C           3.6509   -2.4147    0.4669
  C           4.8038   -2.8631   -0.1829
  C           5.7602   -1.9589   -0.6243
```

(continues on next page)

(continued from previous page)

```
  H           4.3778    0.9134    0.3928
  H           6.3259    0.1044   -0.7337
  H           2.9864   -3.1958    0.7882
  H           4.9572   -3.9235   -0.3393
  H           6.6530   -2.3115   -1.1243
  C           2.5704    0.1468    2.8077
  C           4.3159    0.7126    4.4630
  C           3.3714    1.2844    5.3047
  C           2.0334    1.2840    4.9354
  C           3.9303    0.1495    3.2436
  H           4.7333   -0.2864    2.6778
  H           5.3575    0.6988    4.7582
  H           3.6749    1.7164    6.2495
  C           1.6376    0.7248    3.7185
  H           1.2932    1.7141    5.5981
  H           0.5868    0.7426    3.5290
  C           1.2016   -1.7498    1.7785
  C           0.9537   -2.1874    3.1154
  C           0.6140   -2.5534    0.7571
  C          -0.1758   -3.6680    1.0474
  C          -0.4081   -4.0392    2.3643
  C           0.1611   -3.3038    3.3950
  H           1.3648   -1.7032    3.9822
  H           0.7614   -2.3600   -0.2822
  H          -0.6047   -4.2504    0.2418
  H          -0.0069   -3.6027    4.4220
  H          -1.0186   -4.9050    2.5861
calc:
 type: oniom
 calcs:
  real:
   type: xtb
   gfn: ff
   pal: 6
  high:
   type: xtb
   pal: 6
 models:
  # The "real" does not have to be described, only the "real" system.
  high:
   # Use ethan-carbons for the model system
   inds: [0,34]
   # Use appropriate calculator for model system.
   calc: high
opt:
 # Use LayerOpt
 type: layers
 thresh: gau
 # When the 'layer' key is present, the user has to provide one
 # entry per layer in the optimization. When the layer is empty default
 # values are used. The layers are given from the outmost to the innermost
 # layer.
```

(continues on next page)

(continued from previous page)

```
# In the example above no other arguments for the outmost layer (# real dummy),
# while redundant internal coordinates are used for the innermost layer.
#
# When pysisyphus own ONIOM implementation is used no additional atom indices
# must be given here.
layers:
 - # real dummy
 - geom:
     type: redund
do_hess: True
```

8.3 Example with sockets & i-PI-protocol

Whereas pysisyphus can figure out the layer composition for the microcycles when its own ONIOM calculator is used, the user has to specify it when using sockets and the i-PI-protocol. When covalent bonds between layers exist link atoms (LAs) must be introduced. Given a layer with index `n`, atoms in layer `n-1` that are replaced by LAs in the given layer are called link atom hosts (LAH). A given layer and its LAHs in **higher** layers must be optimized together.

Fig. 8.2: Layer structure of a ONIOM2 optimization using microiterations with link atoms. A given layer as well as connected link atom hosts in **higher** layers must be optimized simultaneously.

A simple example for a ONIOM2 optimization with microiterations is found below. Here, ethanal is optimized, with the model system comprising the carbonyl group (atoms 4, 5 and 6). This introduces a link atom between the two carbons 4 and 0, with carbon 0 becoming a LAH.

Fig. 8.3: ONIOM2 optimization of ethanal. All atoms surrounded by the dashed blue line are optimized in the innermost layer. Hydrogens are shown white, carbons grey and oxygen red.

In contrast to the first example using the native ONIOM implementation the user can omit any input in the `calc:` section. Now the socket addresses have to given for every layer, starting with the total system. For the total system the atom indices can be omitted, as it is assumed that it comprises all atoms. For the remaining layers, the indices of all moving atmos including LAHs in higher layers have to be given.

```
geom:
 type: cartesian
 fn: lib:acetaldehyd_oniom.xyz
calc:
opt:
 type: layers
 layers:
  # Atom indices of the total system / (first layer) don't have to be
  # specified.
  - address: ./layer0sock
  # Optimize atoms of the model system [4, 5, 6] and the link atom host 0.
  - indices: [0,4,5,6]
    address: ./layer1sock
 thresh: gau
# The assert: section can be left out in actual inputs,
```

(continues on next page)

(continued from previous page)

```
# it is here for testing purposes.
assert:
 opt_geom.energy: -115.53522653
```

Another sensible choice for optimizing outer layers besides (regularized) L-BFGS may be preconditioned L-BFGS (*type: plbfgs*).

class pysisyphus.optimizers.LayerOpt.**LayerOpt**(*geometry*, *layers=None*, ***kwargs*)

Bases: *Optimizer*

check_convergence(**args*, ***kwargs*)

Check if we must use the model optimizer to signal convergence.

property layer_num: int

property model_opt

Return the persistent optimizer belonging to the model system. We don't have to supply any coordinates to the optimizer of the most expensive layer, as it is persistent, as well as the associated geometry.

optimize()

Return type
None

postprocess_opt()

Return type
None

print_opt_progress(**args*, ***kwargs*)

Pick the correct method to report opt_progress.

When the model optimizer decides convergence we also report optimization progress using its data and not the data from LayerOpt, where the total ONIOM gradient is stored.

class pysisyphus.optimizers.LayerOpt.**Layers**(*geometry*, *opt_thresh*, *layers=None*)

Bases: object

classmethod from_oniom_calculator(*geometry*, *oniom_calc=None*, *layers=None*, ***kwargs*)

pysisyphus.optimizers.LayerOpt.**get_geom_kwargs**(*layer_ind*, *layer_mask*)

pysisyphus.optimizers.LayerOpt.**get_opt_kwargs**(*opt_key*, *layer_ind*, *thresh*)

class pysisyphus.optimizers.LBFGS.**LBFGS**(*geometry*, *keep_last=7*, *beta=1*, *max_step=0.2*, *double_damp=True*, *gamma_mult=False*, *line_search=False*, *mu_reg=None*, *max_mu_reg_adaptions=10*, *control_step=True*, ***kwargs*)

Bases: *Optimizer*

__init__(*geometry*, *keep_last=7*, *beta=1*, *max_step=0.2*, *double_damp=True*, *gamma_mult=False*, *line_search=False*, *mu_reg=None*, *max_mu_reg_adaptions=10*, *control_step=True*, ***kwargs*)

Limited-memory BFGS optimizer.

See [1] Nocedal, Wright - Numerical Optimization, 2006 for a general discussion of LBFGS. See pysisyphus.optimizers.hessian_updates for the references related to double damping and pysisyphus.optimizers.closures for references related to regularized LBFGS.

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.
- **keep_last** (int, default: 7) -- History size. Keep last 'keep_last' steps and gradient differences.
- **beta** (float, default: 1) -- Force constant in -(H + I)^{1}g.
- **max_step** (float, default: 0.2) -- Upper limit for the absolute component of the step vector in whatever unit the optimization is carried out.
- **double_damp** (bool, default: True) -- Use double damping procedure to modify steps s and gradient differences y to ensure sy > 0.
- **gamma_mult** (bool, default: False) -- Estimate from previous cycle. Eq. (7.20) in [1]. See 'beta' argument.
- **line_search** (bool, default: False) -- Enable implicit linesearches.
- **mu_reg** (Optional[float], default: None) -- Initial guess for regularization constant in regularized LBFGS.
- **max_mu_reg_adaptions** (int, default: 10) -- Maximum number of trial steps in regularized LBFGS.
- **control_step** (bool, default: True) -- Wheter to scale down the proposed step its biggest absolute component is equal to or below 'max_step'
- ****kwargs** -- Keyword arguments passed to the Optimizer baseclass.

get_lbfgs_step(*forces*)

optimize()

postprocess_opt()

reset()

class pysisyphus.optimizers.PreconLBFGS.**PreconLBFGS**(*geometry*, *alpha_init=1.0*, *history=7*, *precon=True*, *precon_update=1*, *precon_getter_update=None*, *precon_kind='full'*, *max_step_element=None*, *line_search='armijo'*, *c_stab=None*, ***kwargs*)

Bases: *Optimizer*

__init__(*geometry*, *alpha_init=1.0*, *history=7*, *precon=True*, *precon_update=1*, *precon_getter_update=None*, *precon_kind='full'*, *max_step_element=None*, *line_search='armijo'*, *c_stab=None*, ***kwargs*)

Preconditioned limited-memory BFGS optimizer.

See pysisyphus.optimizers.precon for related references.

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.
- **alpha_init** (float, default: 1.0) -- Initial scaling factor for the first trial step in the excplicit line search.
- **history** (int, default: 7) -- History size. Keep last 'history' steps and gradient differences.
- **precon** (bool, default: True) -- Wheter to use preconditioning or not.
- **precon_update** (int, default: 1) -- Recalculate preconditioner P in every n-th cycle with the same topology.

- **precon_getter_update** (Optional[int], default: None) -- Recalculate topology for preconditioner P in every n-th cycle. It is usually sufficient to only determine the topology once at the beginning.
- **precon_kind** (Literal['full', 'full_fast', 'bonds', 'bonds_bends'], default: 'full') -- What types of primitive internal coordinates to consider in the preconditioner.
- **max_step_element** (Optional[float], default: None) -- Maximum component of the absolute step vector when no line search is carried out.
- **line_search** (Literal['armijo', 'armijo_fg', 'strong_wolfe', 'hz', None, False], default: 'armijo') -- Whether to use explicit line searches and if so, which kind of line search.
- **c_stab** (Optional[float], default: None) -- Regularization constant c in $(H + cI)^1$ in atomic units.
- ****kwargs** -- Keyword arguments passed to the Optimizer baseclass.

get_precon_getter()

optimize()

prepare_opt()

scale_max_element(*step*, *max_step_element*)

CHAPTER

NINE

RELAXED SCANS

Pysisyphus currently supports 1d-relaxed scans, where one internal coordinate is kept frozen, while the remaining ones are relaxed. A sample input is given below.

```yaml
geom:
 type: redund
 fn: lib:h2o2_hf_321g_opt.xyz
calc:
 type: pyscf
 basis: 321g
 pal: 4
scan:
 type: BOND
 indices: [2, 3]
 # start is optional; if not specified the initial value of the primitive
 # internal coordinate is used.
 start: 3.0
 end: 5.0
 # steps is optional and can be omitted, but then step_size must be given
 steps: 5
 # step_size: 0.4
 # Enable symmetric scan in both directions, starting from the initial coordinates
 # symmetric: False
 #
 # By specifying an 'opt' block, the default optimization values, e.g.
 # convergence thresholds can be altered.
 # opt:
  # thresh: gau
```

If no *start* value is specified in the *scan:* block, then the initial value is used. Specification of *steps* is mandatory, as it determines the number of optimizations to carry out. If not directly given (*step_size*), a suitable step size is derived from an *end* value. Either *end* or *step_size* must be given.

The values *step_size*, *start*, *end have to be provided in atomic units or radians*. When the appropriate YAML constructor is used, inputs can still be in Ångström or degrees. The constructor *!angstrom* converts input in Ångström to Bohr and *!deg* converts input from degrees to radians. See below for an example.

```yaml
[...]
scan:
 type: DIHEDRAL
 indices: [0, 1, 2, 3]
 steps: 5
```

(continues on next page)

(continued from previous page)

```
 step_size: !deg 10.
[...]
```

In the example above, a relaxed scan around a torsion is carried out. In total, 5 steps of 10 degree each will be taken. By using the YAML constructor *!deg*, the input can be given in degrees, instead of radians. Internally, the degrees input is then converted to the appropriate unit, in this case radians.

Naive relaxed scan around the equilibrium value of a primitive internal may be affected by hysteresis, if the system under study is sufficiently complicated.

Relaxed scans are always carried out in redundant internal coordinates. Please see Optimization of Minima for the available optimization options. Supported primitive internals (*type*) are found in: *Types of Primitive Coordinates*.

All geometries, including the initial one are written to *relaxed_scan.trj* in the current working directory.

CHAPTER

TEN

EXCITED STATE OPTIMIZATION

pysisyphus offers excited state (ES) tracking, to follow diadiabatic states along an optimization. ES tracking is enabled by putting *track: True* in the *calc* section of your YAML input. In ES optimizations pysisyphus may use additional programs (wfoverlap, Multiwfn, jmol) if requested. Please see the *installation instructions* for information on how to set them up.

Please consider reading the relevant sections (2 and 3; 4 discusses examples) of the pysisyphus paper. Additionally the user should think about the relevance between equilibrium/non-equilibrium solvation when calculating ES-gradients with implicit solvation. Serveral programs, e.g., Gaussian use equilibrium solvation when doing ES-optimization. It is in the responsibility of the user to add the relevant keywords when using pysisyphus, e.g., `EqSolv` for Gaussian.

10.1 YAML Example

A bare-bone input for the S_1 optimization of the 1H-amino-keto tautomer of cytosin at the TD-DFT/PBE0/def2-SVP level of theory using ORCA is shown below. The full example is found here.

```yaml
opt:
calc:
 type: orca
 keywords: pbe0 def2-svp rijcosx def2/J def2-svp/C
 # Calculate 2 ES by TD-DFT, follow the first one
 blocks: "%tddft nroots 2 iroot 1 tda false end"
 charge: 0
 mult: 1
 pal: 4
 mem: 2000
 # ES-tracking related keywords follow from here
 # Enable ES-tracking, this is important.
 track: True
 # Track ES by transition density overlaps
 ovlp_type: tden
geom:
 type: redund
 fn: cytosin.xyz
```

Additional keywords are possible in the *calc* section. The default values are shown below.

```yaml
calc:
 # Controls calculation of charge-density-differences cubes and rendering
 # Cubes are calcualted by Multiwfn, rendering is handled by jmol.
```

(continues on next page)

(continued from previous page)

```
#
# Possible values are: (None, calc, render).
#  None: No cube calculation/rendering
#  calc: Cube calculation by Multiwfn.
#  render: Same as 'calc' and cubes are then rendered by jmol.
cdds: None
# Overlap type. Using 'wf' requires the external wfoverlap binary. The remaining
# options are implemented directly in pysisyphus.
#
# Possible values are (wf, tden, nto, nto_org).
#  wf: Wavefunction overlaps using the external wfoverlap program.
#  tden: Transition density matrix overlaps.
#  nto: Natural transition orbital overlaps.
#  nto_org: Natural transition orbital overlaps as described by Garcia.
#  top: Transition orbital projection
ovlp_type: wf
# Controls the reference cycle that is used in the overlap calculation. The default
# 'adapt' is recommended.
#
# Possible values are (first, previous, adapt)
#  first: Keep first calculation as reference cycle. Reliable when only minor
#         geometrical changes are expected.
#  previous: Use previous cycle as reference.
#  adapat: Use adaptive algorithm. Please see the pysisyphus paper for a discussion
#          at the end of section 3.
ovlp_with: adapt
# Thresholds controlling the update of the reference cycle.
# The first number specifies the minimum overlap that must be exceeded, for an update
# of the reference cycle. Assuming a value of 0.5 (50 %), the reference cycle update
# is skipped, if the overlaps between the current states and the reference state don't
# exceed 50 %.
# The last two numbers define an interval for the ratio between the second highest
# overlap, and the highest overlap. If the ratio is small, e.g., below 0.3, then both
# states are sufficiently different, and no reference cycle update is needed. If the␣
↪ratio
# is bigger (> 0.6), then the states are quite similar, and an update is currently not
# advised.
# Possible values [three positive floats between 0. and 1.]
adapt_args: [0.5, 0.3, 0.6]
# Explicitly calculate the AO-overlap matrix in a double molecule calculation. Only
# supported by Turbomole and Gaussian calculators. If False, the approximate AO
# overlap matrix is reconstructed from inverting the MO-coefficient matrix.
#
# Possible values: (True, False)
double_mol: False
# Absolute CI-coefficients below this threshold are ignored in the overlap calculation.
#
# Possible values: positive float
conf_thresh: 0.0001
#
# nto/natural transition orbital specific
#
```

(continues on next page)

(continued from previous page)

```
# Number of NTOs to consider in the overlap calculation. Only relevant for 'nto'
# and 'nto_org' ovlp_types.
#
# Possible values: positive integer
use_ntos: 4
# Dynamically decide on number of NTOs according to their participation ratio. Only
# relevant for 'nto_org'
#
# Possible values: boolean
pr_nto: False
#
# wfoverlap/wavefunction overlaps specific
#
# Number of core orbitals to neglect in a wfoverlap calculation. Only relevant
# for the 'wf' ovlp_type. Must be >= 0.
#
# Possible values: positive integer
ncore: 0
#
# tden/transition density matrix specific
#
# Controls which set of MO coefficients (at current cycle, or the reference cycle)
# is used to recover the AO overlap matrix.
#
# Possible values: (ref, cur)
mos_ref: cur
# Controls whether the set of MO coefficents that was NOT used for recovering the AO
# overlap matrix is re-normalized, using the recovered AO overlap matrix. If set to
# True and mos_ref = cur, then the MO coefficients at the reference cycle will be re-
# normalized, and vice versa.
#
# Possible values: (True, False)
mos_renorm: True
```

By brief reasoning it would seem that `mos_ref: ref` and `mos_renorm: True` are more sensible choices, which is possibly true. Right now the present defaults are kept for legacy reasons, and I'll update them after testing out the alternatives.

Please also see Example - Excited State Tracking for possible visualizations when optimizing ES.

10.2 Optimization of Conical Intersections

pysisyphus implements the projected gradient method using an updated branching plane, as developed by Maede, Ohno and Morokuma. Currently, CI-optimization is not enabled for YAML input. An illustrative example is found in *tests/test_conic_intersect*.

CHAPTER

ELEVEN

CHAIN OF STATES METHODS

A chain of states (COS) comprises a set of distinct states (images) and is usually spanned between two minima on a potential energy surface (PES).

When properly relaxed, a COS coincides with a minimum energy path (MEP), or is a good approximation to it. Tangents can be defined for every COS image and together they make up a discretized path that describes the reaction/chemical transformation. The tangents are also used to divide the COS gradient into perpendicular and parallel components, w.r.t the tangents. Relaxation (optimization) of the COS is achieved by minimizing the perpendicular component of the gradient, so only the parallel component remains.

pysisyphus offers different COS implementations, namely Nudged Elasic Band (NEB), including its Adaptive and Free-End (and Free-End-Adaptive) modificiations and different flavors of String methods (Growing String Method, GSM, and Simple Zero Temperature String, SZTS). The GSM implementation is also available for with internal coordinates.

11.1 String method

(When discussing String methods the COS 'images' are usually called 'nodes'.)

In the string method the whole COS is periodically (every n-th cycle) reparametrized by fitting a spline through all nodes and redistributing the nodes them along the spline, according to a predefined parametrization. By choosing between different parametrizations equal node spacing or higher resolution around the highest energy image (HEI) can be achieved. The tangents needed for the gradient projection are obtained as first derivatives of the spline.

Reparametrization every n-th cycle impedes efficient string optimization, and prevents the use of optimizers with some kind of history Conjugate Gradient (CG). The optimizer history is reset after each reparametrization and a simple Steepest Descent (SD) step is done after reparametrization.

11.2 Nudged Elastic Band

No reparametrization takes place in the NEB method. The parallel gradient component along the tangent is projected out and replaced by an artificial spring force. In principle, optimizing NEBs should be easier as there is no reparametrization and more sophisticated optimizers beyond SD can and should be employed.

11.3 General remarks

Converged COS produce good guesses for subsequent TS searches, when the (splined) HEI is determined. In *pysisyphus* subsequent TS searches are easily started by including *tsopt:* in the YAML input. Knowledge of the initial and final images of the COS is used to construct a more complete set of internal coordinates for the TS guess and it is less likely that important coordinates are missed. The initial imaginary mode to follow uphill is selected as the one featuring the highest overlap with HEI tangent.

11.4 YAML example(s)

Below you can find an example YAML-input including the most important options that the user may want to modify when running a GSM optimization.

```
precontr:                               # Preconditioning of translation & rotation
preopt:                                 # Preoptimize inital and final geometry
cos:
 type: gs                               # Do a growing string
 max_nodes: 9                           # Total string will have 9 + 2 == 11 images
 climb: False                           # Enable climbing image (CI), usually a good␣
↪idea.
 climb_rms: 0.005                       # rms(forces) threshold for enabling CI
 climb_lanczos: False                   # Use tangent obtained from Lanczos algorithm␣
↪for CI.
 climb_lanczos_rms: 0.005               # rms(forces) threshold for enabling Lanczos␣
↪algorithm.
 reparam_check: rms                     # Criterian for growing new frontier nodes (rms/
↪norm).
 perp_thresh: 0.05                      # Threshold for growing new frontier nodes.
 reparam_every: 2                       # Reparametrize every n-th cycle when NOT fully␣
↪grown.
 reparam_every_full: 3                  # Reparametrize every n-th cycle when fully␣
↪grown.
opt:
 type: string                           # Optimizer for GrowingString
 stop_in_when_full: -1                  # Stop string optimization N cycles after fully␣
↪grown
                                        # Disabled by -1. Usually it's a better idea to
                                        # further converge the string, after is fully␣
↪grown.
 align: False                           # Disable Kabsch algorithm. Should be True with
                                        # type: cart
 scale_step: global                     # Scale down step as whole (global) or per image
                                        # (per_image)
tsopt:
 type: rsprfo                           # Continue with TS-optimization of highest␣
↪energy images
                                        # (HEI) using the RS-P-RFO algorithm
 do_hess: True                          # Calculate hessian at optimized TS geometry
 trust_max: 0.3
 thresh: gau_loose
calc:
```

(continues on next page)

(continued from previous page)

```
 type: orca
 keywords: "b3lyp 6-31G* rijcosx"
 pal: 4
 charge: 0
 mult: 1
geom:
 type: dlc
 fn: [first_preopt.xyz, last_preopt.xyz] # Run GrowingString in delocalized internal␣
↪coordinates
                                         # (preferred).
```

For NEB optimizations a different optimizer (not `type: string`) should be used, e.g., QuickMin `type: qm` in the beginning, and `type: lbfgs` later on. It is not yet possible to specify two different optimizers that are used in stages, so if is desired it must be done manually.

```
# Taken from examples/complex/06_diels_alder...
geom:
 type: cart
 fn: diels_alder.trj
calc:
 type: xtb
 charge: 0
 mult: 1
 pal: 4
preopt:
 max_cycles: 5
interpol:                       # In NEBs the whole path is interpolated beforehand.
 type: redund                   # Possible values: redund|idpp|lst|linear
 between: 10                    # Interpolate n-geometries between every pair of supplied
                                # geometries. For two this yields 1 + 10 + 1 == 12␣
↪images,
                                # for three geometries this yields 1 + 10 + 1 + 10 + 1␣
↪== 23
                                # geometries.
cos:
 type: neb
 climb: False
 align_fixed: True              # Align the fixed atoms of the initial and final images␣
↪along the path.
opt:
 type: lbfgs
 align: True                   # Align the image of the current step with the image of␣
↪the previous step
 align_factor: 0.9             # If full alignment is not desired, a factor between 0␣
↪and 1
                               # can be specified.
 rms_force: 0.01
 max_step: 0.04
tsopt:
 type: rsirfo
 do_hess: True
 max_cycles: 75
```

(continues on next page)

(continued from previous page)

```
 thresh: gau_tight
 hessian_recalc: 7
irc:
 type: eulerpc
 rms_grad_thresh: 0.0005
endopt:
```

Further examples for COS optimizations from *.yaml* input can be found here.

11.5 General advice for COS optimizations

- Start from optimized geometries or use the *preopt:* key in the YAML input.
- Consider fixing the initial and final images *fix_ends: True*
- Always use *align: True* when optimizing a COS in cartesian coordinates to remove translation and rotation. *align: True* must not be used when running a COS with DLC.
- Don't over-converge a COS. It is usually a better idea to converge a COS loosely and use the highest energy image (HEI) as a guess for a subsequent transition state (TS) search.
- If possible use a climbing image (CI) *climb: True*
- When running a growing string calculation (*type: gs*) use *stop_in_when_full: [n]* in the *opt:* section with a small integer *[n]* to stop the COS relaxation after the string is fully grown

11.6 Chain Of States base class

Base class for chain of state methods

class pysisyphus.cos.ChainOfStates.**ChainOfStates**(*images*, *fix_first=True*, *fix_last=True*, *align_fixed=True*, *climb=False*, *climb_rms=0.005*, *climb_lanczos=False*, *climb_lanczos_rms=0.005*, *climb_fixed=True*, *energy_min_mix=False*, *scheduler=None*, *progress=False*)

as_xyz(*comments=None*)

property atoms

calculate_forces()

property calculator

property cart_coords

Return a flat 1d array containing the cartesian coordinates of all images.

check_for_climbing_start(*ref_rms*)

clear()

concurrent_force_calcs(*images_to_calculate*, *image_indices*)

property coords
Return a flat 1d array containing the coordinates of all images.

property coords3d

describe()

property energy

property forces

get_climbing_forces(*ind*)

get_climbing_indices()

get_dask_client()

get_fixed_indices()

get_hei_index(*energies=None*)
Return index of highest energy image.

get_image_calc_counter_sum()

get_perpendicular_forces(*i*)
[1] Eq. 12

get_splined_hei()

get_tangent(*i*, *kind='upwinding'*, *lanczos_guess=None*, *disable_lanczos=False*)
[1] Equations (8) - (11)

get_tangents()

property gradient

property image_coords

property image_inds

property last_index

log(*message*)

logger = <Logger cos (DEBUG)>

property masses_rep

property max_image_num

property moving_images

property moving_indices
Returns the indices of the images that aren't fixed and can be optimized.

par_image_calc(*image*)

property perpendicular_forces

prepare_opt_cycle(*last_coords*, *last_energies*, *last_forces*)

Implements additional logic in preparation of the next optimization cycle.

Should be called by the optimizer at the beginning of a new optimization cycle. Can be used to implement additional logic as needed for AdaptiveNEB etc.

property results

rms(*arr*)

Root mean square

Returns the root mean square of the given array.

Parameters
arr (*iterable of numbers*)

Returns
rms -- Root mean square of the given array.

Return type
float

set_climbing_forces(*forces*)

set_coords_at(*i*, *coords*)

Called from helpers.procrustes with cartesian coordinates. Then tries to set cartesian coordinate as self.images[i].coords which will raise an error when coord_type != "cart".

set_images(*indices*, *images*)

set_vector(*name*, *vector*, *clear=False*)

set_zero_forces_for_fixed_images()

This is always done in cartesian coordinates, independent of the actual coord_type of the images as setting forces only work with cartesian forces.

valid_coord_types = ('cart', 'cartesian', 'dlc')

zero_fixed_vector(*vector*)

11.7 Chain Of State Methods

11.7.1 Nudged Elastic Band (NEB)

class pysisyphus.cos.NEB.**NEB**(*images*, *variable_springs=False*, *k_max=0.3*, *k_min=0.1*, *perp_spring_forces=None*, *bandwidth=None*, ***kwargs*)

Bases: *ChainOfStates*

fmt_k()

property forces

get_parallel_forces(*i*)

get_quenched_dneb_forces(*i*)

See [3], Sec. VI and [4] Sec. D.

get_spring_forces(*i*)

property parallel_forces

set_variable_springs()

update_springs()

11.7.2 Adaptive NEB

class pysisyphus.cos.AdaptiveNEB.**AdaptiveNEB**(*images*, *adapt=True*, *adapt_fact=0.25*, *adapt_between=1*, *scale_fact=False*, *keep_hei=True*, *free_ends=True*, ***kwargs*)

Bases: *NEB*

__init__(*images*, *adapt=True*, *adapt_fact=0.25*, *adapt_between=1*, *scale_fact=False*, *keep_hei=True*, *free_ends=True*, ***kwargs*)

(Free-End) Adaptive Nudged Elastic Band.

Parameters

- **images** (*list of Geometry objects*) -- Images of the band.
- **adapt** (*bool, default True*) -- Whether to adapt the image number or not. This switch is included to support the FreeEndNEB class, that is just a thin wrapper around this class.
- **adapt_fact** (*positive float*) -- Factor that is used to decide wether to adapt. The inital threshold is calculated by multiplying the RMS force of the band with this factor. When the RMS of the force falls below this threshold adaption takes place.
- **adapat_between** (*positive integer*) -- Number of images to interpolate between the highest energy image and its neighbours. The number of starting images must be higher or equal then 2*adapt_between+3, as we reuse/transfer the calculators from the starting images onto the new ones.
- **scale_fact** (*bool, default False*) -- Whether to increase adapt_fact in deeper levels. This may lead to earlier adapation.
- **keep_hei** (*bool, optional*) -- Whether to keep the highest energy image (usually a very good idea) or to interpolate only between the neighbouring images.
- **free_ends** (*bool, default True*) -- Whether to use modified forces on the end images.

adapt_this_cycle(*forces*)

Decide wether to adapt.

Parameters
forces (*np.array*) -- Forces of the previous optimization cycle.

Returns
adapt -- Flag that indicates if adaption should take place in this cycle.

Return type
bool

property forces

See Eq. (7) in [2].

prepare_opt_cycle(**args*, ***kwargs*)

Check for adaption and adapt if needed.

See ChainOfStates.prepare_opt_cycle for a complete docstring.

update_adapt_thresh(*forces*)

Update the adaption threshold.

Parameters
forces (*np.array*) -- Forces of the previous optimization cycle.

11.7.3 Free-End NEB

class pysisyphus.cos.FreeEndNEB.**FreeEndNEB**(**args*, *fix_first=False*, *fix_last=False*, ***kwargs*)

Bases: *AdaptiveNEB*

__init__(**args*, *fix_first=False*, *fix_last=False*, ***kwargs*)

Simple Free-End-NEB method.

Derived from AdaptiveNEB with disabled adaptation. Only implements Eq. (7) from [2]. For other implementations please see the commit 01bc8812ca6f1cd3645d43e0337d9e3c5fb0ba55. There the other variants are present but I think Eq. (7) in [2] is the simplest & best bet.

11.7.4 Simple Zero-Temperature String

class pysisyphus.cos.SimpleZTS.**SimpleZTS**(*images*, *param='equal'*, ***kwargs*)

Bases: *ChainOfStates*

reparametrize()

11.8 Growing Chain Of States base class

Base class for growing chain of state methods

class pysisyphus.cos.GrowingChainOfStates.**GrowingChainOfStates**(*images*, *calc_getter*, *max_nodes=10*, ***kwargs*)

property arc_dims

get_new_image_from_coords(*coords*, *index*)

property max_image_num

new_node_coords(*k*)

prepare_opt_cycle(**args*, ***kwargs*)

Implements additional logic in preparation of the next optimization cycle.

Should be called by the optimizer at the beginning of a new optimization cycle. Can be used to implement additional logic as needed for AdaptiveNEB etc.

set_new_node(*k*)

11.9 Growing Chain Of State Methods

11.9.1 Growing String Method

class pysisyphus.cos.GrowingString.**GrowingString**(*images*, *calc_getter*, *perp_thresh=0.05*, *param='equi'*, *reparam_every=2*, *reparam_every_full=3*, *reparam_tol=None*, *reparam_check='rms'*, *max_micro_cycles=5*, *reset_dlc=True*, *climb=False*, ***kwargs*)

Bases: *GrowingChainOfStates*

property forces

property full_string_image_inds

property fully_grown

Returns wether the string is fully grown. Don't count the first and last node.

get_additional_print()

get_cur_param_density(*kind=None*)

get_new_image(*ref_index*)

Get new image by taking a step from self.images[ref_index] towards the center of the string.

get_tangent(*i*)

[1] Equations (8) - (11)

property image_inds

property left_size

property lf_ind

Index of the left frontier node in self.images.

property nodes_missing

Returns the number of nodes to be grown.

reparam_cart(*desired_param_density*)

reparam_dlc(*desired_param_density*, *thresh=0.001*)

reparametrize()

reset_geometries(*ref_geometry*)

property rf_ind

Index of the right frontier node in self.images.

property right_size

set_coords(*image*, *coords*)

spline(*tangents=False*)

property string_size

CHAPTER

TWELVE

TRANSITION STATE OPTIMIZATION

To cite Frank Jensen: "Locating transition states (TSs) is black magic, especially in internal coordinates" and I can say this is true. The most promising TS optimizers in *pysisyphus* employ second derivative information (hessian) but locating TS is also possible using only first derivatives by means of the *dimer method* (DM).

TS optimizations should preferably be carried out in internal coordinates (*type: redund|dlc*). Before starting the actual TS search, one should **always** check the internal coordinates that pysisyphus sets up, by executing *pysistrj [xyz] --internals*, with *[xyz]* corresponding to the geometry you plan to use for the optimization. **Check carefully** if all supposed reaction coordinates are present. **If they are missing** define them manually with the *add_prims* key in the YAML input.

Calculation of the exact hessian can be avoided by using the DM by using *rx_coords: [list of primitives]* in combination with *hessian_init: fischer|lindh|swart|simple*. By using both options, a diagonal model hessian is calcualted and modified for use in a TS optimization.

The DM can be used with a powerful preconditioned LBFGS optimizer (`type: lbfgs`).

12.1 Hessian Based TS Optimization

12.1.1 YAML example

Below you can find an example YAML-input including the most important options that the user may want to modify for the Hessian-based optimizers (RS-I-RFO, RS-P-RFO, TRIM).

```
tsopt:
 type: rsirfo|rsprfo|trim          # Optimization algorithm

 do_hess: True                     # Calculate the hessian at the final geometry
                                   # after the optimization.

 hessian_recalc: 1                 # Recalculate the exact hessian every n-th cylce
                                   # Very costly but a small number like 5 may be a good
                                   # idea.

 prim_coord: [BOND, 3, 18]         # Select the mode to follow uphill by overlap with
                                   # this primitive internal coordinate. Expects exactly
                                   # one typed primitive.

rx_modes: [[[[BOND, 3, 18], 1]]]   # Select initial mode based on overlaps with a
                                   # constructed mode(s). Can be seen as generalization of
                                   # prim_coord. Expects a a list of (lists of pairs,␣
```

(continues on next page)

(continued from previous page)

```
↪with each
                                # pair comprising a typed primitive and a phase␣
↪factor).
                                #
                                # See examples/tsopt/{05..,06..} for examples.

 #rx_coords: [[BOND, 3, 18],]  # Obtain initial mode by modifying a model Hessian. ␣
↪Has
                                # to be used with 'hess_init: swart|fischer|lindh|xtb'
                                # to avoid calculation of exact hessian.
                                # Also requires 'redund' or 'dlc' to work.

 #root: 0                      # Follow the n-th imaginary mode uphill, minimize
                                # along the others.

 #hessian_init: calc           # Initial hessian is calculated exactly.

 #hessian_update: bofill       # Bofill hessian-update. Should not be modified for
                                # TS optimizations.

 #hessian_ref: [path]          # Expects a path to a precalculated hessian at any
                                # level of theory. The emode to maximize along is␣
↪selected
                                # by highest overlap with imaginary mode from the␣
↪reference
                                # hessian.


 #max_micro_cycles: 50         # No. of micro cycles for the RS-variants. Does not␣
↪apply
                                # to TRIM.

 #trust_radius: 0.3            # Initial trust radius.
 #trust_max: 1.0               # Max. trust radius
 #trust_min: 0.1               # Min. trust radius
calc:
 type: xtb
 pal: 4
 charge: 0
 mult: 1
geom:
 type: redund
 fn: shaked.geom_000.xyz
 add_prims: [[24, 20], ]       # If using internal coordinates ALWAYS check the␣
↪coordinates
                                # that pysisyphus generates (pysistrj [xyz] --
↪internals). If
                                # some important (reaction) coordinates appears to be␣
↪missing
                                # define them manually. In this example
                                # we add a bond. If additional internal coordinates can
                                # derived from the added primitives pysisyphus will do
```

(continues on next page)

(continued from previous page)

```
                                    # it.
```

Second-derivative-free TS optimization can be done using the *Dimer* method. Attached you can find a sample input.

Further examples for TS optimizations from *.yaml* input can be found here.

12.2 Dimer Method

pysisyphus implements the dimer method (DM) as calculator, wrapping another calculator for the actual energy and gradient calculations. All DM configurations have to be done in the `calc:` section. The best results are obtained by optimizing the DM with LBFGS in connection with a preconditioner (PLBFGS), estimated from the Lindh model Hessian. in constrast to the TS optimizers mentioned above, PBLFGS is configured in the `opt` section of the YAML input. DM related information is logged to *dimer.log*. The current DM orientation is saved in files with extension *.N*. Animated *.trj* files of the DM orientation are saved to *.N.trj*.

12.2.1 YAML example

Below you can find an example for the DM, applied to the isomerization of HCN. Default values are commented out. See examples/tsopt/02_hcn_tsopt_dimer for the full example.

```
opt:
 type: plbfgs    # Preconditioned LBFGS
 thresh: baker   # Baker threshold, comparable to the Gaussian defaults
 do_hess: True   # Calculate Hessian at the end. May not be applicable to systems
                 # where Hessian calculation is infeasible.
calc:
 type: dimer     # Dimer calculator wrapping PySCF at HF/3-21G level of theory
 calc:
  type: pyscf
  basis: 321g
  pal: 2
  charge: 0
  mult: 1
 #N_raw: [file]               # Path to a file containing an initial dimer orientation
 #length: 0.0189              # Separation of dimer images from common midpoint
 #rotation_max_cycles: 15     # Maximum number of rotations
 #rotation_method: fourier
 #rotation_thresh: 1e-4       # rms(rotationa force) threshold
 #rotation_tol: 1             # Rotation tolerance in degree. If a proposed trial
                              # rotation angle is below the tolerance the rotation
                              # will be skipped.
 #rotation_max_element: 0.001  # Max. step element for "rotation_method: direct"
 #rotation_interpolate: True   # Interpolate force on image after rotation
 #rotation_remove_trans: True  # Remove overall translation from N-vector
 #seed: 20182503               # Seed for the RNG for reproducability reasons.
geom:
 type: cart
 fn: 01_hcn.xyz
```

12.3 General advice for TS optimizations

- Use as many exact hessians as your computational budget allows (*hessian_recalc: [n]*)
- Use tight convergence settings for your SCF. In contrast to codes like ORCA *pysisyphus* does not enforces this automatically
- Grid-based methods (DFT, some RI-types) may need finer grids to reduce numerical noise
- Try to preoptimize the TS using a cheap method (xtb) that allows *hessian_recalc: 1*
- Decrease current & allowed trust radius (*trust_radius: 0.1* and *trust_max: 0.3|0.5*)
- Check for orrect coordinate setup

12.4 TSHessianOptimizer base class

Base class for TS optimizers that empoly a hessian.

class pysisyphus.tsoptimizers.TSHessianOptimizer.**TSHessianOptimizer**(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Bases: *HessianOptimizer*

Optimizer to find first-order saddle points.

__init__(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Baseclass for transition state optimizers utilizing Hessian information.

Several arguments expect a typed primitive or an iterable of typed primitives. A typed primitive is specified as (PrimType, int, int, ...), e.g., for a bond between atoms 0 and 1: (BOND, 0, 1) or for a bend between the atom triple 0, 1, 2 as (BEND, 0, 1, 2).

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.
- **roots** (Optional[List[int]], default: None) -- Indices of modes to maximize along, e.g., to optimize saddle points of 2nd order. Overrides 'root'.
- **root** (int, default: 0) -- Index of imaginary mode to maximize along. Shortcut for 'roots' with only one root.

- **hessian_ref** (Optional[str], default: None) -- Filename pointing to a pysisyphus HDF5 Hessian.
- **rx_modes** (*iterable of (iterable of (typed_prim, phase_factor))*) -- Select initial root(s) by overlap with a modes constructed from the given typed primitives with respective phase factors.
- **prim_coord** (*typed_prim*) -- Select initial root/imaginary mode by overlap with this internal coordinate. Shortcut for 'rx_modes' with only one internal coordinate.
- **rx_coords** (*iterable of (typed_prim)*) -- Construct imaginary mode comprising the given typed prims by modifying a model Hessian.
- **hessian_init** (Literal['calc', 'unit', 'fischer', 'lindh', 'simple', 'swart', 'xtb', 'xtb1', 'xtbff'], default: 'calc') -- Type of initial model Hessian.
- **hessian_update** (Literal['none', None, False, 'bfgs', 'damped_bfgs', 'flowchart', 'bofill', 'ts_bfgs', 'ts_bfgs_org', 'ts_bfgs_rev'], default: 'bofill') -- Type of Hessian update. Defaults to BFGS for minimizations and Bofill for saddle point searches.
- **hessian_recalc_reset** (bool, default: True) -- Whether to skip Hessian recalculation after reset. Undocumented.
- **max_micro_cycles** (int, default: 50) -- Maximum number of RS iterations.
- **trust_radius** (float, default: 0.3) -- Initial trust radius in whatever unit the optimization is carried out.
- **trust_max** (float, default: 0.5) -- Maximum trust radius.
- **augment_bonds** (bool, default: False) -- Try to derive additional streching coordinates from the imaginary mode.
- **min_line_search** (bool, default: False) -- Carry out line search along the imaginary mode.
- **max_line_search** (bool, default: False) -- Carry out line search in the subspace that is minimized.
- **assert_neg_eigval** (bool, default: False) -- Check for the existences for at least one significant negative eigenvalue. If enabled and no negative eigenvalue is present the optimization will be aborted.
- ****kwargs** -- Keyword arguments passed to the HessianOptimizer/Optimizer baseclass.

static do_line_search(*e0*, *e1*, *g0*, *g1*, *prev_step*, *maximize*, *logger=None*)

prepare_opt(**args*, ***kwargs*)

property root

property roots

step_and_grad_from_line_search(*energy*, *gradient_trans*, *eigvecs*, *min_indices*, *max_indices*)

update_ts_mode(*eigvals*, *eigvecs*)

valid_updates = ('bofill', 'ts_bfgs', 'ts_bfgs_org', 'ts_bfgs_rev')

12.5 TS-Optimizer using hessian information

12.5.1 Restricted-Step Partitioned-RFO

class pysisyphus.tsoptimizers.RSPRFOptimizer.**RSPRFOptimizer**(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Bases: *TSHessianOptimizer*

optimize()

12.5.2 Restricted-Step Image-Function-RFO

class pysisyphus.tsoptimizers.RSIRFOptimizer.**RSIRFOptimizer**(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Bases: *TSHessianOptimizer*

optimize()

12.5.3 Trust-Region Image-Function Method

class pysisyphus.tsoptimizers.TRIM.**TRIM**(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Bases: *TSHessianOptimizer*

optimize()

12.6 TS-Optimization without hessian information

12.6.1 Dimer method

class pysisyphus.calculators.Dimer.**Dimer**(*calculator*, **args*, *N_raw=None*, *length=0.0189*, *rotation_max_cycles=15*, *rotation_method='fourier'*, *rotation_thresh=0.0001*, *rotation_tol=1*, *rotation_max_element=0.001*, *rotation_interpolate=True*, *rotation_disable=False*, *rotation_disable_pos_curv=True*, *rotation_remove_trans=True*, *trans_force_f_perp=True*, *bonds=None*, *N_hessian=None*, *bias_rotation=False*, *bias_translation=False*, *bias_gaussian_dot=0.1*, *seed=None*, *write_orientations=True*, *forward_hessian=True*, ***kwargs*)

property C

Shortcut for the curvature.

property N

add_gaussian(*atoms*, *center*, *N*, *height=0.1*, *std=0.0529*, *max_cycles=50*, *dot_ref=None*)

property can_bias_f0

property can_bias_f1

property coords0

property coords1

curvature(*f1*, *f2*, *N*)

Curvature of the mode represented by the dimer.

direct_rotation(*optimizer*, *prev_step*)

do_dimer_rotations(*rotation_thresh=None*)

property energy0

property f0

property f1

property f1_bias

property f2

Never calculated explicitly, but estimated from f0 and f1.

fourier_rotation(*optimizer*, *prev_step*)

get_N_raw_from_hessian(*h5_fn*, *root=0*)

get_forces(*atoms*, *coords*)

Meant to be extended.

get_gaussian_energies(*coords*, *sum_=True*)

get_gaussian_forces(*coords*, *sum_=True*)

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

remove_translation(*displacement*)

property rot_force

rotate_coords1(*rad*, *theta*)

Rotate dimer and produce new coords1.

set_N_raw(*coords*)

property should_bias_f0

May lead to calculation of f0 and/or f1 if present!

property should_bias_f1

May lead to calculation of f0 and/or f1 if present!

update_orientation(*coords*)

class pysisyphus.calculators.Dimer.**Gaussian**(*height*, *center*, *std*, *N*)

energy(*R*, *height=None*)

forces(*R*, *height=None*)

exception pysisyphus.calculators.Dimer.**RotationConverged**

CHAPTER

THIRTEEN

INTRINSIC REACTION COORDINATE (IRC)

By default two paths are integrated in plus- and minus-direction of the imaginary mode (transition vector, TV) at the transition state (TS). If the IRC is started from a non-stationary point with non-vanishing gradient the *downhill: True* argument can be set to integrate only one path and skip the initial hessian calculation.

IRCs are integrated in mass-weighted coordinates and the default integrator is *EulerPC*.

The initial displacement from the TS is done by requiring a certain energy lowering $\mathrm{d}E$ from moving along the TV and calculating the corresponding step length from a quadratic potential: $\mathrm{d}E = \frac{1}{2}(k \cdot \mathrm{d}q^2)$, with k being the eigenvalue of the TV (imaginary mode) and $\mathrm{d}q$ the required step length. The default required energy lowering is 0.0005 au. Alternatively the initial can be done by a prescribed length: *displ: length* and *displ_length: [desired step]* (default is $0.1\sqrt{\mathrm{amu}} \cdot \mathrm{bohr}$).

The resulting endpoints of the IRC integration can be further optimized to stationary poins by adding the *endopt:* section (vide infra). By setting *fragments: True* in *endopt* separate fragments will be detected and optimized individually. This may be useful if the molecules are only weakly bound. By setting *fragments: total* the total system, as well as the separate fragments will be optimized.

By default IRC path data is dumped to *dump_fn: irc_data.h5* every *dump_every: 5* cycles. IRC paths can be plotted with *pysisplot --irc*.

13.1 YAML example(s)

Below you can find an example YAML-input including the most important options that the user may want to modify.

```yaml
geom:
 fn: hfabstraction_ts_opt_xtb.xyz   # Input coordinates
calc:
 type: xtb                          # extended tight-binding calculator
 pal: 4
 charge: 0
 mult: 1
irc:
 type: eulerpc                      # Similar to EulerPC from Gaussian

 #rms_grad_thresh: 0.001            # Convergence threshold
 #displ: energy|length|energy_cubic # How to do the initial displacement
 #displ_energy: 0.001               # Energy lowering in au (Hartree)
 #displ_length: 0.1                 # Step length along the TV
 #forward: True
 #backward: True
 #downhill: False                   # Only integrate downhill, disables forward/backward
```

(continues on next page)

(continued from previous page)

```
 #hessian_init: null                # Path to HDF5 Hessian file
 #displ_third_h5: null              # Path to HDF5 file containing third derivative data
endopt:
 #fragments: False|True|total       # Detect & optimize fragments separately. Default is
                                    # False. When set to 'total' the total system as well
                                    # as the fragments are optimized.
 do_hess: False                     # Frequency calculation at the end
```

Further examples for IRC calculations from *.yaml* input can be found here.

13.2 IRC base class

Base class for IRC integrators from which actual IRC integrators are derived.

class pysisyphus.irc.IRC.**IRC**(*geometry*, *step_length=0.1*, *max_cycles=125*, *downhill=False*, *forward=True*, *backward=True*, *root=0*, *hessian_init=None*, *displ='energy'*, *displ_energy=0.001*, *displ_length=0.1*, *displ_third_h5=None*, *rms_grad_thresh=0.001*, *hard_rms_grad_thresh=None*, *energy_thresh=1e-06*, *imag_below=0.0*, *force_inflection=True*, *check_bonds=True*, *out_dir='.'*, *prefix=''*, *dump_fn='irc_data.h5'*, *dump_every=5*)

__init__(*geometry*, *step_length=0.1*, *max_cycles=125*, *downhill=False*, *forward=True*, *backward=True*, *root=0*, *hessian_init=None*, *displ='energy'*, *displ_energy=0.001*, *displ_length=0.1*, *displ_third_h5=None*, *rms_grad_thresh=0.001*, *hard_rms_grad_thresh=None*, *energy_thresh=1e-06*, *imag_below=0.0*, *force_inflection=True*, *check_bonds=True*, *out_dir='.'*, *prefix=''*, *dump_fn='irc_data.h5'*, *dump_every=5*)

Base class for IRC calculations.

Parameters

- **geometry** (Geometry) -- Transtion state geometry, or initial geometry for downhill run.
- **step_length** (*float, optional*) -- Step length in unweighted coordinates.
- **max_cycles** (*int, optional*) -- Positive integer, controlloing the maximum number of IRC steps taken in a direction (forward/backward/downhill).
- **downhill** (*bool, default=False*) -- Downhill run from a non-stationary point with non-vanishing gradient. Disables forward and backward runs.
- **forward** (*bool, default=True*) -- Integrate IRC in positive s direction.
- **backward** (*bool, default=True*) -- Integrate IRC in negative s direction.
- **root** (*int, default=0*) -- Use n-th root for initial displacement from TS.
- **hessian_init** (*str, default=None*) -- Path to Hessian HDF5 file, e.g., from a previous TS calculation.
- **displ** (*str, one of ("energy", "length", "energy_cubic")*) -- Controlls initial displacement from the TS. 'energy' assumes a quadratic model, from which a step length for a given energy lowering (see 'displ_energy') is determined. 'length' corresponds to a displacement along the transition vector. 'energy_cubic' considers 3rd derivatives of the energy along the transition vector.
- **displ_energy** (*float, default=1e-3*) -- Required energy lowering from the TS in au (Hartree). Used with 'displ: energy|energy_cubic'.

- **displ_length** (*float, default=0.1*) -- Step length along the transition vector. Used only with 'displ: length'.
- **displ_third_h5** (*str, optional*) -- Path to HDF5 file containing 3rd derivative information. Used with 'displ: energy_cubic'.
- **rms_grad_thresh** (*float, default=1e-3,*) -- Convergence is signalled when to root mean square of the unweighted gradient is less than or equal to this value
- **energy_thresh** (*float, default=1e-6,*) -- Signal convergence when the energy difference between two points is equal to or less than 'energy_thresh'.
- **imag_below** (*float, default=0.0*) -- Require the wavenumber of the imaginary mode to be below the given threshold. If given, it should be a negative number.
- **force_inflection** (*bool, optional*) -- Don't indicate convergence before passing an inflection point.
- **check_bonds** (*bool, optional, default=True*) -- Report whether bonds are formed/broken along the IRC, w.r.t the TS.
- **out_dir** (*str, optional*) -- Dump everything into 'out_dir' directory instead of the CWD.
- **prefix** (*str, optional*) -- Short string that is prepended to all files that are created by this class, e.g., trajectories and HDF5 dumps.
- **dump_fn** (*str, optional*) -- Base name for the HDF5 files.
- **dump_every** (*int, optional*) -- Dump to HDF5 every n-th cycle.

property coords

dump_data(*dump_fn=None*, *full=False*)

dump_ends(*path*, *prefix*, *coords=None*, *trj=False*)

property energy

get_conv_fact(*mw_grad*, *min_fact=2.0*)

get_endpoint_and_ts_geoms()

get_full_irc_data()

get_irc_data()

get_path_for_fn(*fn*)

property gradient

initial_displacement()

Returns non-mass-weighted steps in +s and -s direction for initial displacement from the TS. Earlier version only returned one step, that was later multiplied by either 1 or -1, depending on the desired IRC direction (forward/backward). The current implementation directly returns two steps for forward and backward direction. Whereas for plus and minus steps for displ 'length' and displ 'energy'

step_plus = -step_minus

is valid, it is not valid for dipsl 'energy_cubic' anymore. The latter step is formed as

x(ds) = ds * v0 + ds**2 * v1

so
x(ds) != -x(ds)

as
ds * v0 + ds**2 * v1 != -ds * v0 - ds**2 * v1 .

So, all required step are formed directly and later used as appropriate.

See
https://aip.scitation.org/doi/pdf/10.1063/1.454172 https://pubs.acs.org/doi/10.1021/j100338a027 https://aip.scitation.org/doi/pdf/10.1063/1.459634

irc(*direction*)

log(*msg*)

property m_sqrt

mass_weigh_hessian(*hessian*)

property mw_coords

property mw_gradient

postprocess()

prepare(*direction*)

report_bonds(*prefix*, *bonds*)

report_conv_thresholds()

run()

set_data(*prefix*)

unweight_vec(*vec*)

valid_displs = ('energy', 'length', 'energy_cubic')

13.3 IRC Integrators

13.3.1 Damped-Velocity-Verlet integrator

class pysisyphus.irc.DampedVelocityVerlet.**DampedVelocityVerlet**(*geometry*, *v0=0.04*, *dt0=0.5*, *error_tol=0.003*, *max_cycles=150*, ***kwargs*)

Bases: *IRC*

damp_velocity(*velocity*)

estimate_error(*new_mw_coords*)

mw_grad_to_acc(*mw_grad*)

Takes care of the units for the mass-weighted gradient. Converts units of a mass-weighted gradient [Hartree/(Bohr*amu)] to units of acceleration [sqrt(amu)*Bohr/fs^2]. The 1e30 comes from converting second^2 to femto second^2.

prepare(*direction*)

step()

13.3.2 Euler integrator

Not recommended as it only produces reasonable results with very small step sizes.

class pysisyphus.irc.Euler.**Euler**(*geometry*, *step_length=0.01*, ***kwargs*)

Bases: *IRC*

step()

13.3.3 Euler-Predictor-Corrector integrator

Recommended IRC integrator and default choice.

class pysisyphus.irc.EulerPC.**EulerPC**(**args*, *hessian_recalc=None*, *hessian_update='bofill'*, *hessian_init='calc'*, *max_pred_steps=500*, *loose_cycles=3*, *dump_dwi=False*, *scipy_method=None*, *corr_func='mbs'*, ***kwargs*)

Bases: *IRC*

corrector_step(*init_mw_coords*, *step_length*, *dwi*)

get_integration_length_func(*init_mw_coords*)

prepare(**args*, ***kwargs*)

scipy_corrector_step(*init_mw_coords*, *step_length*, *dwi*)

Solve IRC equation dx/ds = -g/|g| on DWI PES in mass-weighted coordinates. Integration done until self.step_length in unweighted coordinates is achieved.

step()

13.3.4 Gonzales-Schlegel-2 integrator

class pysisyphus.irc.GonzalezSchlegel.**GonzalezSchlegel**(*geometry*, *max_micro_cycles=20*, *micro_step_thresh=0.001*, *hessian_recalc=None*, *line_search=False*, ***kwargs*)

Bases: *IRC*

micro_step(*counter*)

Constrained optimization on a hypersphere.

perp_component(*vec*, *perp_to*)

postprocess()

step()

13.3.5 Local-Quadratic-Approximation integrator

class pysisyphus.irc.LQA.**LQA**(*geometry*, *N_euler=5000*, ***kwargs*)

Bases: *IRC*

step()

13.3.6 Modified-Ishida-Morokuma-Komornicki integrator

Similar to the algorithm implemented by ORCA 4.

class pysisyphus.irc.IMKMod.**IMKMod**(*geometry*, *corr_first=True*, *corr_first_size=0.5*, *corr_first_energy=True*, *corr_bisec_size=0.0025*, *corr_bisec_energy=True*, ***kwargs*)

Bases: *IRC*

fit_grad_and_energies(*energy0*, *grad0*, *energy1*, *direction*)

fit_parabola(*x*, *y*)

step()

13.3.7 Runge-Kutta-4 integrator

Not recommended, as it is very slow.

class pysisyphus.irc.RK4.**RK4**(*geometry*, *step_length=0.1*, *max_cycles=125*, *downhill=False*, *forward=True*, *backward=True*, *root=0*, *hessian_init=None*, *displ='energy'*, *displ_energy=0.001*, *displ_length=0.1*, *displ_third_h5=None*, *rms_grad_thresh=0.001*, *hard_rms_grad_thresh=None*, *energy_thresh=1e-06*, *imag_below=0.0*, *force_inflection=True*, *check_bonds=True*, *out_dir='.'*, *prefix=''*, *dump_fn='irc_data.h5'*, *dump_every=5*)

Bases: *IRC*

get_k(*mw_coords*)

step()

CHAPTER

FOURTEEN

DIABATIZIATION

Adiabatic states can be diabatized with a suitable rotation matrix, which in turn can be determined by maximizing a cost funtion that depends on the rotation matrix and some selected adiabatic properties (see below). Pysisyphus implements property-based direct diabatizion, as outlined by Subotnik[1] and Truhlar et al.[2]

Currently, dipole moments, the trace of the primitive quadrupole tensor and the electrostatic potential can be utilized for diabatization.

14.1 General Algorithm

Diabatic states $\{\Xi_A\}$ can be obtained via unitary transformation of adiabatic states $\{\Psi_I\}$.

$$\Xi_A = \sum_{I=1} \Psi_I U_{IA}$$

which is equivalent to

$$\boldsymbol{\Xi} = \mathbf{U}^\intercal \Psi$$

with rotation matrix $\mathbf{U}$, as well as adiabatic and diabatic state vectors $\boldsymbol{\Psi}$ and $\boldsymbol{\Xi}$. Given two adiabatic states Ψ_1 and Ψ_2 $\mathbf{U}$ is defined as

$$\mathbf{U} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} .$$

The related diabatic states Ξ_A and Ξ_B are

$$\boldsymbol{\Xi} = \mathbf{U}^\intercal \boldsymbol{\Psi} \tag{14.1}$$

$$\begin{bmatrix} \Xi_A \\ \Xi_B \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} \Psi_1 \\ \Psi_2 \end{bmatrix} \tag{14.2}$$

(14.3)

$$\Xi_A = \cos(\theta)\Psi_1 + \sin(\theta)\Psi_2 \tag{14.4}$$

$$\Xi_B = -\sin(\theta)\Psi_1 - \cos(\theta)\Psi_2 \tag{14.5}$$

(14.6)

Similarly, diabatic expectation values are calculated as linear combination of adiabatic expectation values:

$$\Xi_A|\hat{O}|\Xi_B = \sum_{IJ} U_{IA}\Psi_I|\hat{O}|\Psi_J U_{JB}$$

[1] https://doi.org/10.1063/1.3042233
[2] https://doi.org/10.1063/1.4894472, https://doi.org/10.1063/1.4948728

or in matrix notation

$$\mathbf{O}_{\mathrm{dia}} = \mathbf{U}^{\intercal}\mathbf{O}_{\mathrm{adia}}\mathbf{U} \; .$$

By maximizing a cost function that depends on diabatic properties $\mathbf{O}_{\mathrm{dia}}$, a rotation matrix $\mathbf{U}$ suitable for diabatization can be determined.

With a known rotation matrix $\mathbf{U}$, the diagonal adiabatic electronic energy matrix $\mathbf{V}$ can be transformed to its diabatic equivalent $\mathbf{D}$, e.g, to determine diabatic couplings $|D_{12}|$.

$$\mathbf{D} = \mathbf{U}^{\intercal}\mathbf{V}\mathbf{U} \tag{14.7}$$

$$\begin{bmatrix} D_{11} & D_{12} \\ D_{12} & D_{22} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} V_{11} & 0 \\ 0 & V_{22} \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{14.8}$$

Depending on the employed properties, different cost functions $f(\mathbf{U})$ are used. Subotnik[Page 129, 1] proposed to maximize the magnitude of the diabatic dipole moments:

$$f_D(\mathbf{U}) = \sum_{A=1} |\Xi_A|\hat{\mu}|\Xi_A|^2 \; .$$

This is the same cost function as in classical Boys localization of molecular orbitals. By also considering the trace of the primitive quadrupole tensor, $f_D(\mathbf{U})$ is extended to $f_{DQ}(\mathbf{U})$:

$$f_{DQ}(\mathbf{U}) = f_D(\mathbf{U}) + \sum_{A=1} \sum_{j=1}^{N_Q} \alpha_j |\Xi_A|tr(\hat{Q})|\Xi_A|^2 \; .$$

The contribution of the quadrupole moments to $f(\mathbf{U})$ is controlled by the scaling factor α_j. Here, subscript j refers to the j-th expansion center. Currently, only one expansion center ($N_Q = 1$) can be used in pysisyphus. By default α_j is set to $10\ a_0^{-2}$.

In some cases, considering only the dipole moments is not enough to discriminate different adiabatic states and quadrupole moments have to be taken into account. Several examples are outlined by Truhlar et al.[Page 129, 2]

Slight improvements may be possible by incorporating the electronic component of the electrostatic potential (ESP) Φ.

$$f_{DQ\Phi}(\mathbf{U}) = f_{DQ}(\mathbf{U}) + \sum_{A=1} \sum_{k=1}^{N_\Phi} \beta_k |\Xi_A|\hat{\Phi}|\Xi_A|^2$$

Here again, N_Φ denotes the number of expansion centers where the ESP is calculated. Similar to the quadrupole moments, only one expansion center is currently supported in pysisyphus. β_k is a scaling factor that controls the contribution of the electrostatic potential to the overall cost function $f_{DQ\Phi}(\mathbf{U})$.

Maximization of cost function f is achieved via repeated 2 x 2 rotations (Jacobi sweeps) of diabatic state pairs. The formulas to calculate the required rotation angle are given by Kleier[3] and Pulay et al.[4] The maximization is started with a unit-matrix guess for $\mathbf{U}$ and usually converges quite rapidly.

A different approach based on unitary optimization is outlined by Lehtola,[5] but is not available in pysisyphus.

Diabatization is carried out via the *pysisdia* entry hook. See below for an example.

[3] https://doi.org/10.1063/1.1681683
[4] https://doi.org/10.1002/jcc.540140615
[5] https://doi.org/10.1021/ct400793q

14.2 Example

Diabatization of 3 adiabatic states using dipole moments requires the 3 adiabatic permanent dipole moments (0,0; 1,1; 2,2), as well as the respective transition dipole moments (0,1; 0,2; 1,2).

Listing 14.1: ../tests/test_diabatization/guanine_indole_dia.yaml

```yaml
# Guanine-Indole cation, wB97X-D/6-31G*, TDA, 3 states.
# Values are taken from the Table S2 in the SI of
# https://doi.org/10.1063/5.0035593
adiabatic_energies: [0.0, 0.717, 0.905]
adiabatic_labels: [GS, LE, CT]
dipoles:
  - [0, 0, -1.253, 0.270, -1.876]
  - [1, 1, -1.691, -0.653, -2.083]
  - [2, 2, 0.948, 0.733, -1.327]
  - [0, 1, -1.224, -0.279, -0.325]
  - [0, 2, 2.227, 0.866, 0.735]
  - [1, 2, -1.574, -1.029, -0.517]
```

Executing *pysisdia guanine_indole_dia.yaml* produces the following output:

```
                        ###################
                        # D-DIABATIZATION #
                        ###################

Dipole moments
--------------
[[[-1.253 -1.224  2.227]
  [-1.224 -1.691 -1.574]
  [ 2.227 -1.574  0.948]]

 [[ 0.27  -0.279  0.866]
  [-0.279 -0.653 -1.029]
  [ 0.866 -1.029  0.733]]

 [[-1.876 -0.325  0.735]
  [-0.325 -2.083 -0.517]
  [ 0.735 -0.517 -1.327]]]

Starting Jacobi sweeps.
000: P=169.63193103 dP=169.63193103
001: P=172.49529007 dP=  2.86335904
002: P=172.49529008 dP=  0.00000000
Jacobi sweeps converged in 3 cycles.

All energies are given in eV.

Adiabatic energy matrix V
-------------------------
[[0.    0.    0.   ]
 [0.    0.717 0.   ]
 [0.    0.    0.905]]
```

(continues on next page)

(continued from previous page)

```
Rotation matrix U
-----------------
[[ 0.16289082  0.84920213  0.50231695]
 [-0.84058556  0.38601857 -0.38000734]
 [-0.51660672 -0.36034067  0.77670593]]
det(U)=1.0000

Diabatic energy matrix D = UVU
------------------------------
[[ 0.74814945 -0.06418359 -0.13410224]
 [-0.06418359  0.2243505  -0.35846691]
 [-0.13410224 -0.35846691  0.64950005]]

Diabatic states  sorted by energy
---------------------------------
0: ₁, 0.2244 eV
1: ₂, 0.6495 eV
2: ₀, 0.7481 eV

Composition of diabatic states
----------------------------------
₀ = + 0.1629·₀(GS) - 0.8406·₁(LE) - 0.5166·₂(CT)
₁ = + 0.8492·₀(GS) + 0.3860·₁(LE) - 0.3603·₂(CT)
₂ = + 0.5023·₀(GS) - 0.3800·₁(LE) + 0.7767·₂(CT)

Weights U²
----------
[[0.02653342 0.72114427 0.25232232]
 [0.70658408 0.14901034 0.14440558]
 [0.2668825  0.1298454  0.6032721 ]]

Absolute diabatic couplings
---------------------------
|D₀₁| = 0.0642 eV
|D₀₂| = 0.1341 eV
|D₁₂| = 0.3585 eV
```

14.3 YAML input

All possible input options are outlined below. The numbers are just dummy values.

```yaml
# Adiabatic energies, of the states to diabatize.
#
# List of floats.
#   [V0, V1, ...]
# May be absolute or relative energies.
adiabatic_energies: [0.0, 0.6541, 0.7351]
# Dipole moments.
#
```

(continues on next page)

(continued from previous page)

```
# List of lists of length 5, each containing 2 integers followed by 3 floats.
#   [state0, state1, dpm_x, dpm_y, dpm_z]
# The integers are adiabatic state indices, the 3 floats are the X, Y and Z
# components of a dipole moment vector.
# If both integers are the same, the 3 floats correspond to the permanent dipole moment
# of a state. Otherwise, they correspond to a transition dipole moment
# between two states.
dipoles: [
 [0, 0, -1.0, -2.0, -3.0],
 [1, 1, -2.0, -1.0, -3.0],
 [0, 1, -2.0, -1.0, -3.0],
]

#
# Optional input below
#

# Adiabatic state labels.
#
# List of strings.
#   [label1, label2, ...]
# Must be of same length as adiabatic energies.
adiabiatc_labels: [GS, LE, CT]
# Energy unit.
#
# Literal["eV", "Eh"]
unit: eV
# Trace of primitive quadrupole moment tensor.
#
# List of lists of length 3, each containing 2 integers followed by 1 float.
#   [state0, state1, tr_qpm] = [state0, state1, (qpm_xx + qpm_yy + qpm_zz)
# The same comments as for the dipole moments apply.
quadrupoles: [
 [0, 0, 1.0],
 [1, 1, -14.0],
 [0, 1, 33.0],
]
# Quadrupole moment scaling factor alpha in 1/a_0^2.
#
# Float
alpha: 10.0
# Electronic component of electrostatic potential in au
epots: [
 [0, 0.48],
 [1, 0.12],
]
# Electrostatic potential scaling factor beta in a_0.
#
# Float
beta: 1.0
```

CHAPTER

FIFTEEN

PLOTTING

pysisyphus offers extensive visualization capabilities via the *pysisplot* command. All relevant information produced by pysisyphus and the interfaced quantum chemistry programs is dumped to HDF5 files, namely:

Filename	Description
optimization.h5	Produced by all optimizers when *dump: True* (default).
afir.h5	Contains true and modified energies & forces in AFIR calculations.
forward_irc_data.h5	IRC data from forward run.
backward_irc_data.h5	IRC data from backward run.
finished_irc_data.h5	IRC data from forward & backward run.
overlap_data.h5	From excited state calculations by OverlapCalculator classes.

A help message that shows possible usages of *pysisplot* is displayed by

```
pysisplot --help
```

If needed, results from multiple optimizations are written to different HD5-groups of the same HDF5-file. Available groups can be listed either by the HDF5-tool *h5ls* or *pysisplot --h5_list [hd55 file]*. The desired group for visualization is then selected by the *--h5_group [group name]* argument.

15.1 Example - Diels-Alder reaction

The plotting capabilities of *pysisplot* are demonstrated for the example of a Diels-Alder reaction between ethene and 1-4-butadiene (see examples/06_diels_alder_xtb_preopt_neb_tsopt_irc) at xtb-GFN2 level of theory. Educts and product of the reaction are preoptimized for 5 cycles, then interpolated in internal coordinates. A chain of states (COS) is optimized loosely via a LBFGS optimizer. The highest energy image (HEI) is employed for a subsequent transition state (TS) optimization. Finally the intrinsic reaction coordinate (IRC) is obtained by means of an Euler-Predictor-Corrector integrator.

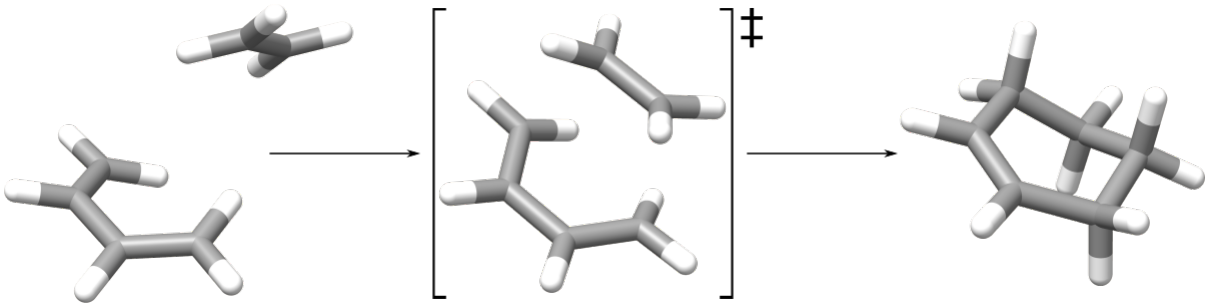

Fig. 15.1: Educts, TS and products of the Diels-Alder reaction between ethene and 1-4-butadiene.

15.1.1 Plotting optimization progress

The progress of an optimization can be plotted via

```
pysisplot --opt [--h5_fn optimization.h5] [--h5_group opt]
```

Arguments given in parantheses are the assumed defaults. The result is shown below for the preoptimizations of educts and product.

For COS optimizations all image energies of one optimization cycle are summed into a total energy, which is then plotted in the first panel. *pysisplot --opt* may not be the best choice in these cases. Use *pysisplot --cosens* and *pysisplot --cosforces* instead (see below).

15.1.2 Plotting COS optimization progress

Compared to simple surface-walking optimizations of single molecules, COSs consist of multiple images, each with its own energy and force vector. In this case a simple plot as shown above is not suitable. Instead of *pysisplot --opt* a better visualization is offered by *pysisplot --cosens* and *pysisplot --cosforces*. The latter two commands are compatible with all COS methods available in pysisphus.

```
pysisplot --cosens [--h5_fn optimization.h5] [--h5_group opt]
```

This produces three plots.

1. Animated. COS image energies along the optimization.
2. Static. Energies of last optimization cycle.
3. Static. Energies of all cycles with earlier cycles given in a lighter shade and more recent cycles in a darker shade. The last cycle is splined and the position of the splined HEI is indicated.

Please note that equidistant image spacing is assumed for the latter two plots. Here only the two latter plots are shown.

The forces acting on the respective COS images can also be plotted.

```
pysisplot --coforces [--h5_fn optimization.h5] [--h5_group opt]
```

Please note that nothing is plotted for images 0 and 11, as they remained fixed in the optimization.

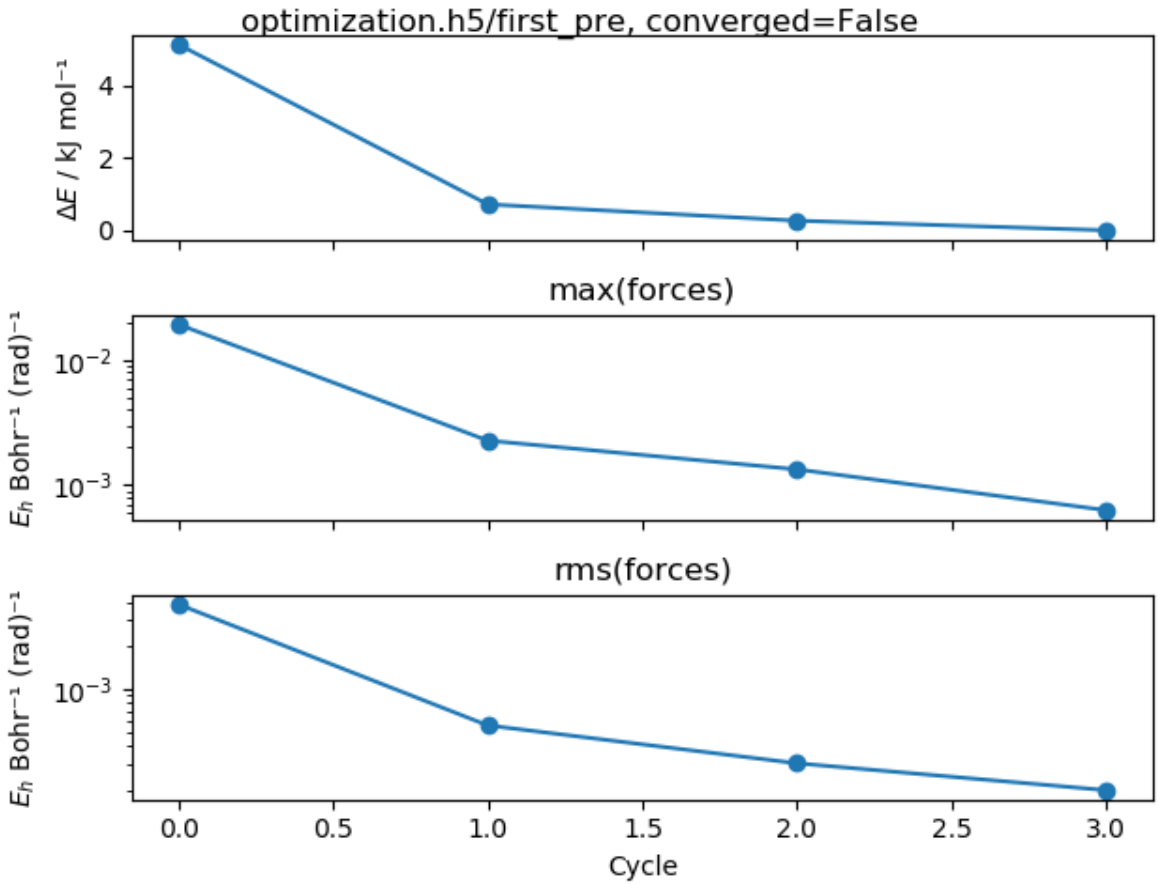

Fig. 15.2: Preoptimization of the Diels-Alader reaction educts ethene + 1,4-butadiene. The upper panel shows the energy change along the optimization. Middle and lower panel show the max and rms of the force. The HDF5 filename and group are noted in the image title.

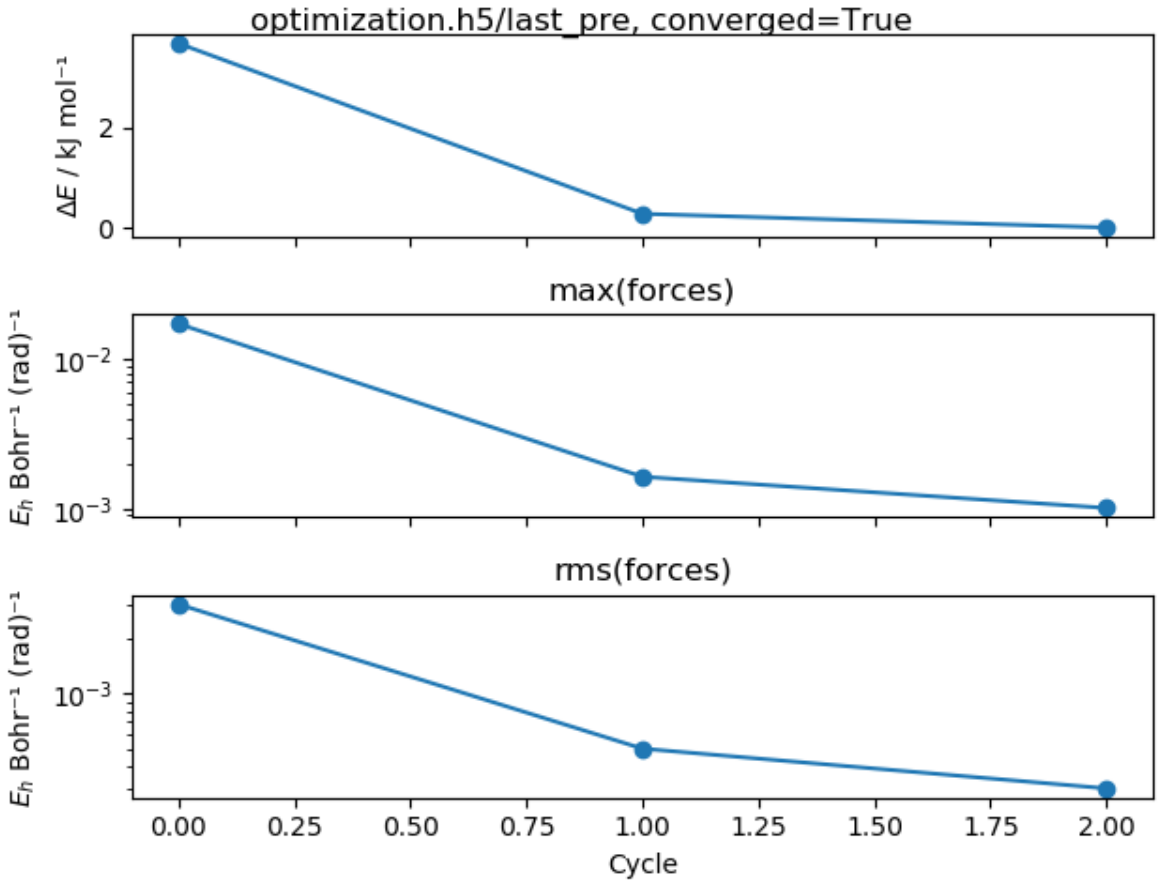

Fig. 15.3: Preoptimization of the Diels-Alader reaction product. Compared to the educts the optimization already converged after 4 cycles.

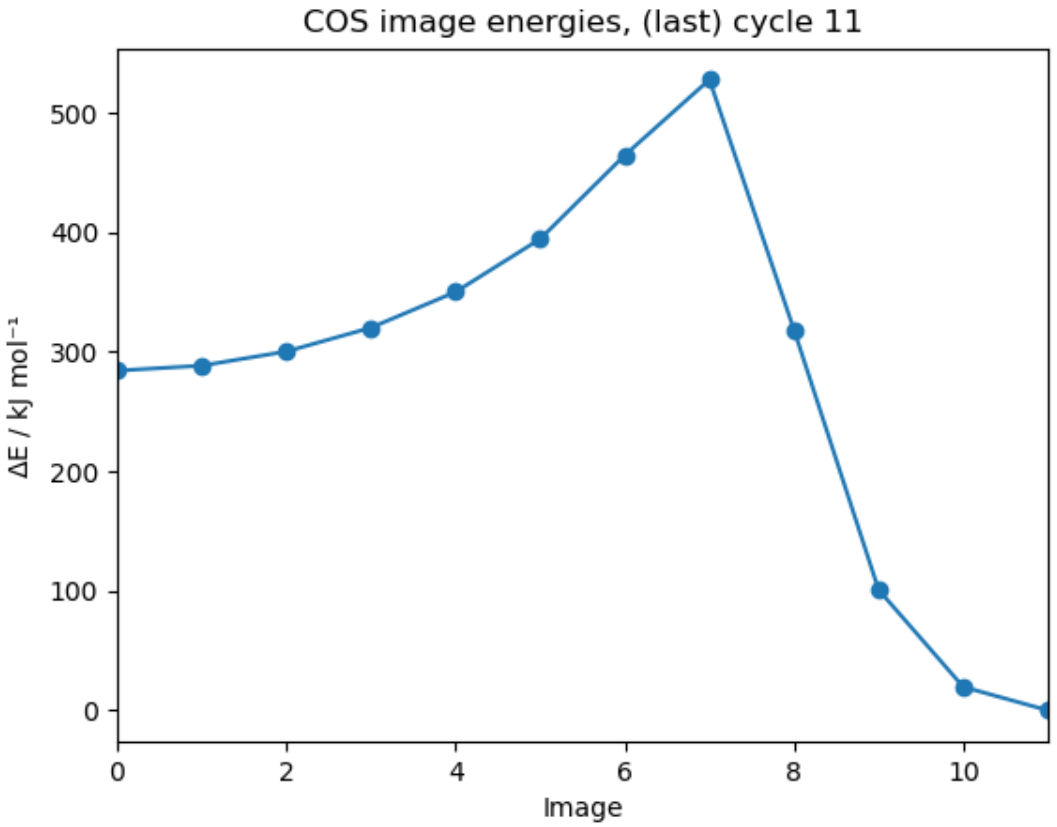

Fig. 15.4: COS image energies of the last (most recent) optimization cycle.

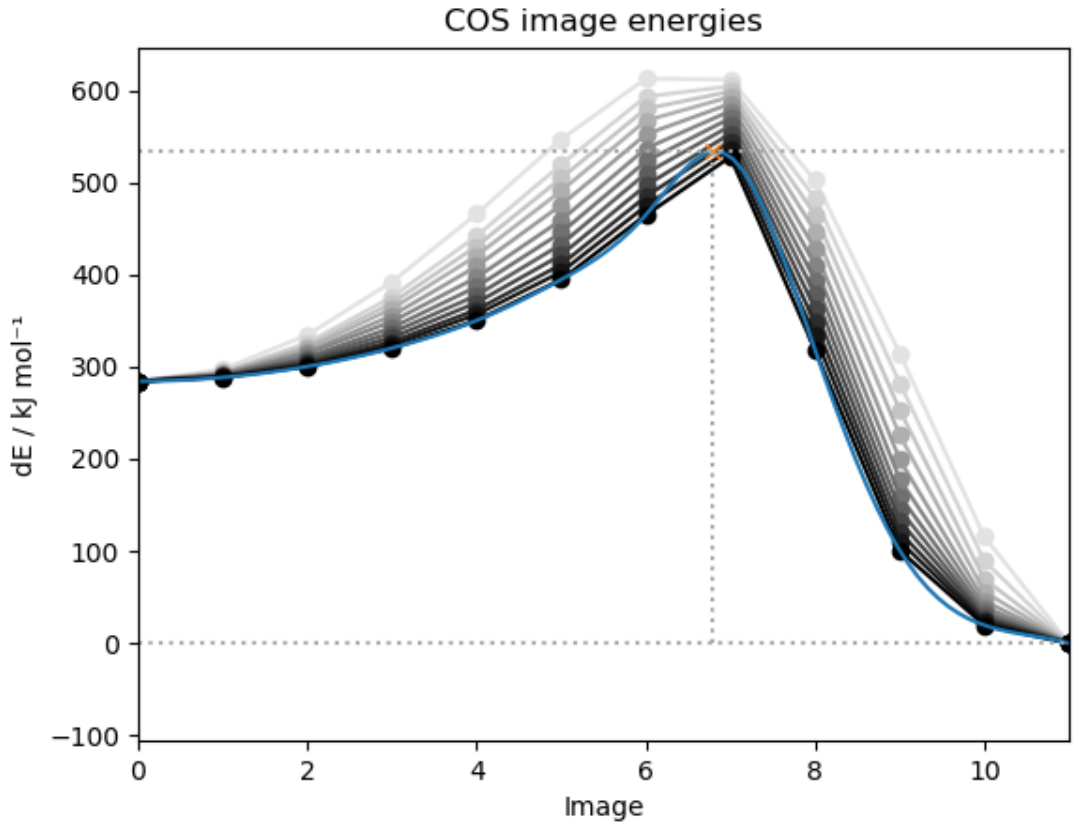

Fig. 15.5: COS image energies of all optimization cycles. Not that the acutal difference between image geometries are not taken into account. Equidistance is assumed. Later (more recent) cycles are given in a darker shade.

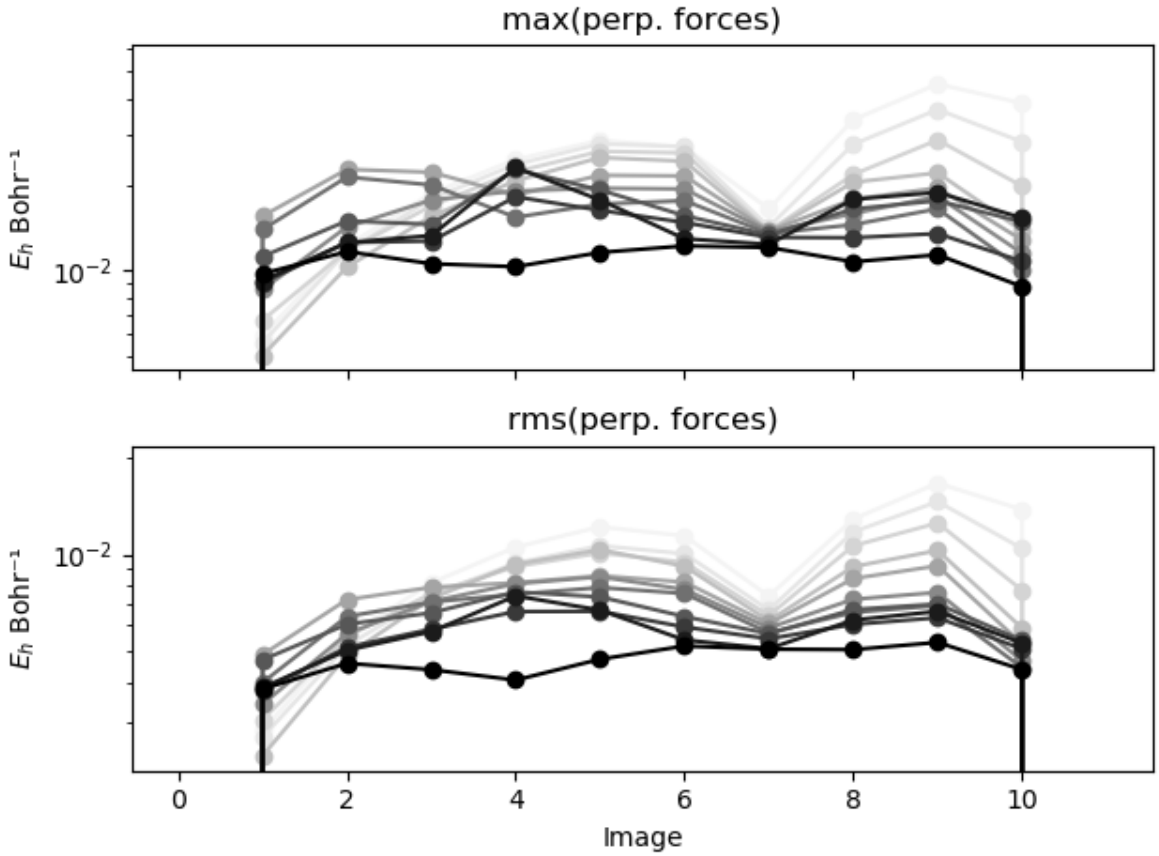

Fig. 15.6: Maximum component and root-mean-square (rms) of the perpendicular component of the force, acting on the COS images.

15.1.3 Plotting TS-optimization progress

The TS-optimization progress is plotted with *pysisplot --opt --h5_group tsopt*. Here we explicitly selected a different HDF5 group by *--h5_group*.

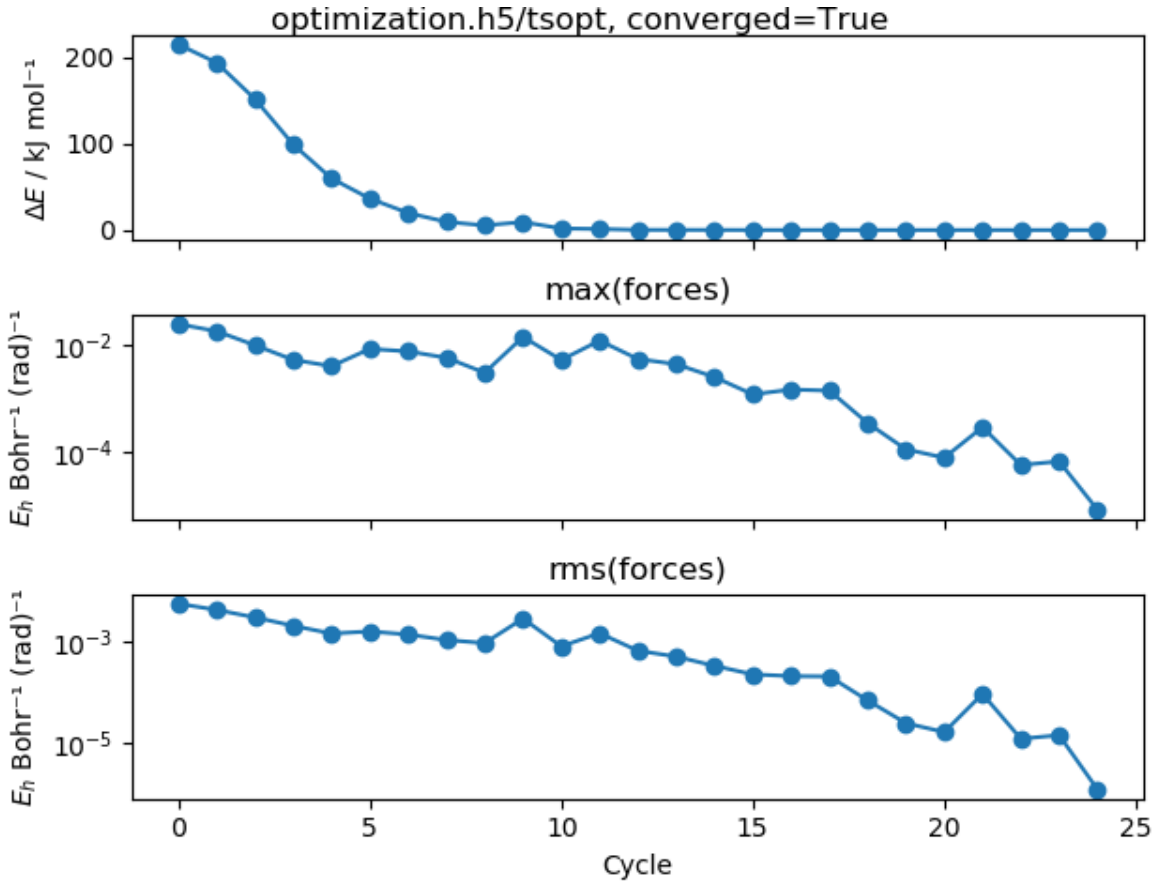

Fig. 15.7: Progress of the TS optimization, started from the HEI.

15.1.4 Plotting the Intrinsic Reaction Coordinate

IRC profiles are easily plotted by

```
pysisplot --irc
```

Multiple plots may appear, depending on the progress of the IRC. The IRC coordinate is given in mass-weighted cartesian coordinates, whereas gradients are given in non-mass-weighted units.

Evidently the IRC integration failed at the end, as can be seen from the the bunched up points, but unless you want to do some kind of transition-state-theory (TST; not supported by pysisyphus) calculations this should not be a problem.

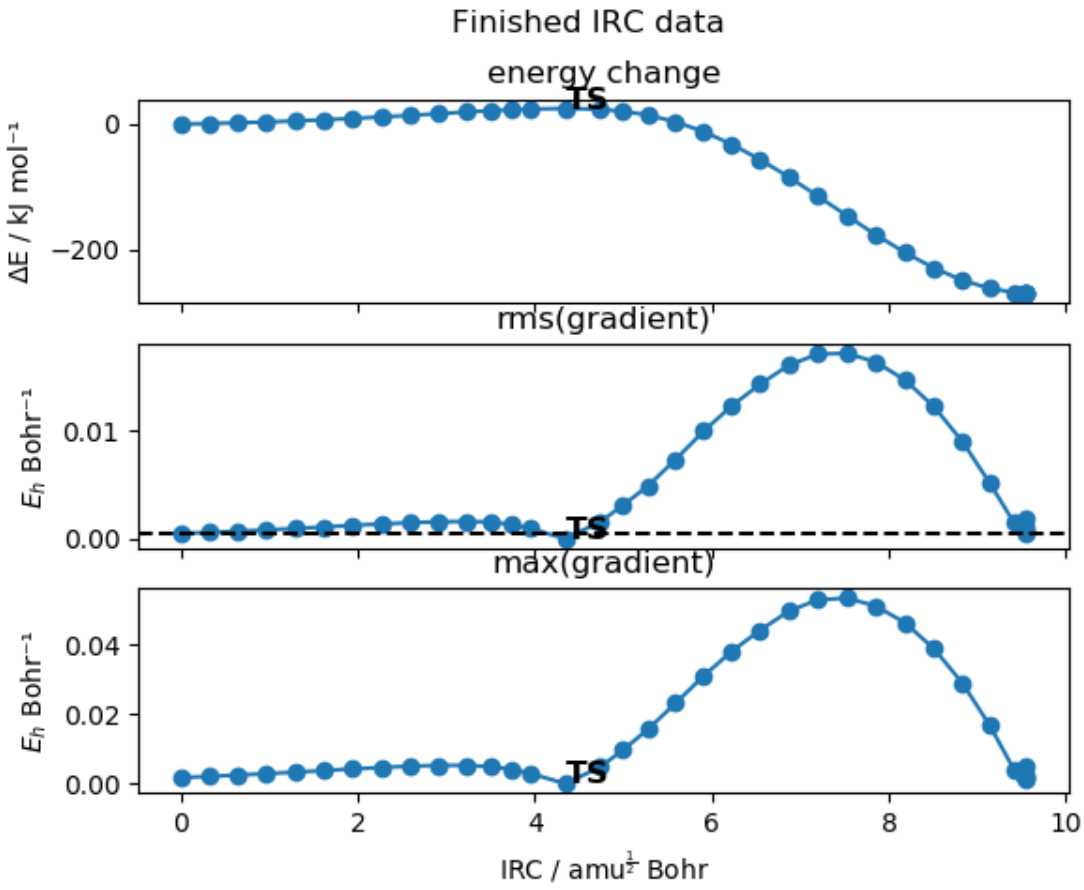

Fig. 15.8: IRC for the Diels-Alder reaction between ethene and 1,4-butadiene.

15.2 Example - AFIR

pysisplot is able to visualize AFIR calculations and to highlight intersting geometries along the optimization. Shown below is an example taken from the AFIR-Paper . By using AFIR the S_N2 between OH^- and fluoromethylene can be forced, yielding methanol and the fluorine anion. The corresponding unit test can be found in the *tests/test_afir* directory of the repository.

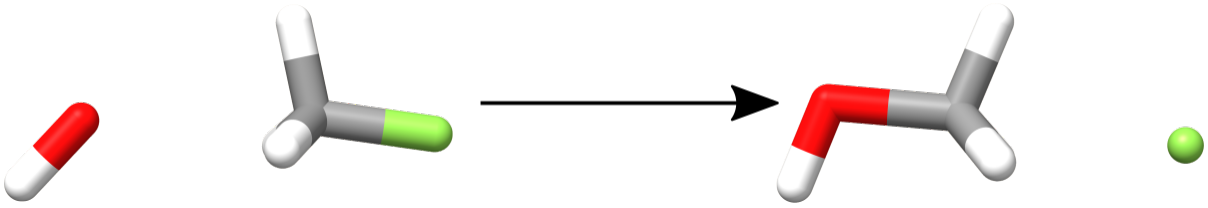

Fig. 15.9: Formation of methanol by means of a S_N2 reaction.

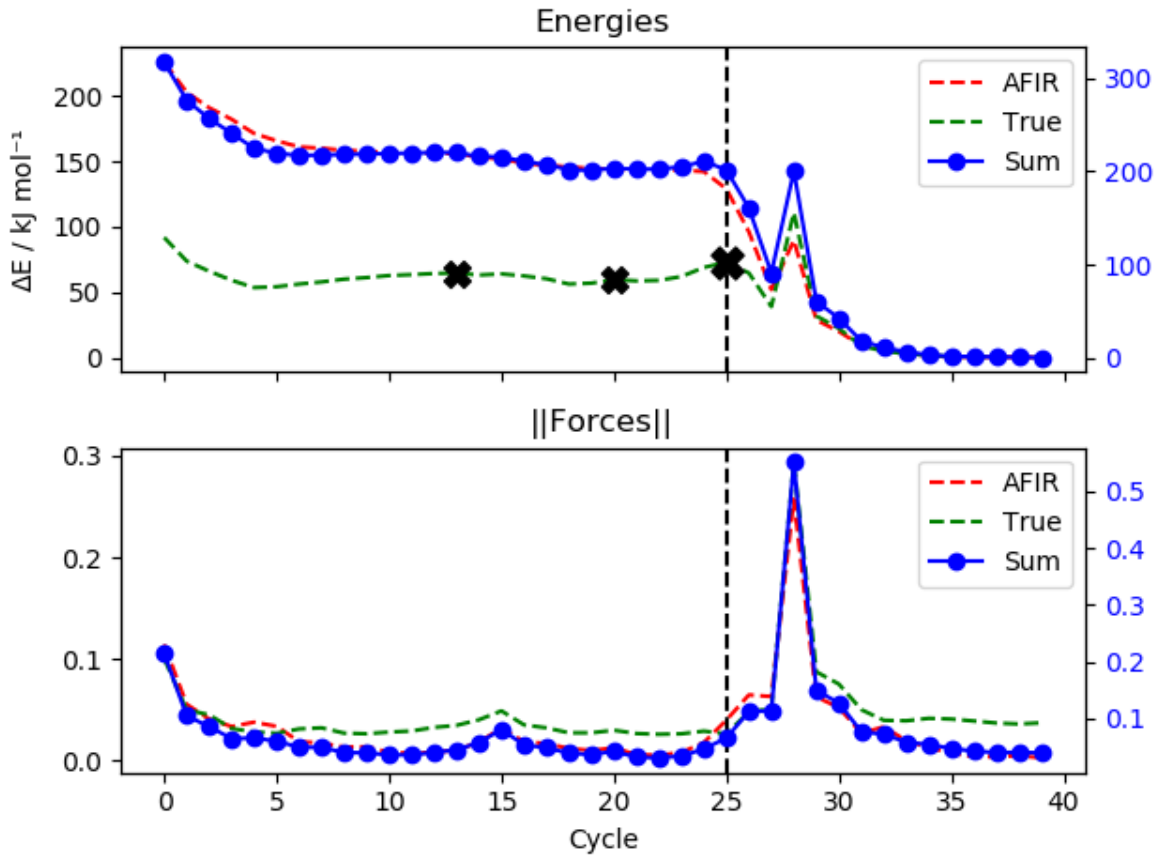

Fig. 15.10: Energy profile and force norms along the S_N2 reaction.

15.3 Example - Excited State Tracking

pysisyphus is aware of excited states (ES) and can track them using various approaches over the course of an optimization or an IRC. By calculating the overlap matrices between ESs at given geometry and a reference geometry pysisyphus can track the desired ES. All relevant data is stored in *overlap_data.h5*.

Optimizing an ES is demonstrated for the S_1 of the 1H-amino-keto tautomer of Cytosin at the PBE0/def-SVP level of theory. A corresponding test can be found under (*tests/test_cytosin_opt*). Right after the first optimization cycle a root flip occurs and the S_1 and S_2 switch. The potential energy curves along the optimization are plotted by:

```
pysisplot --all_energies
pysisplot -a
```

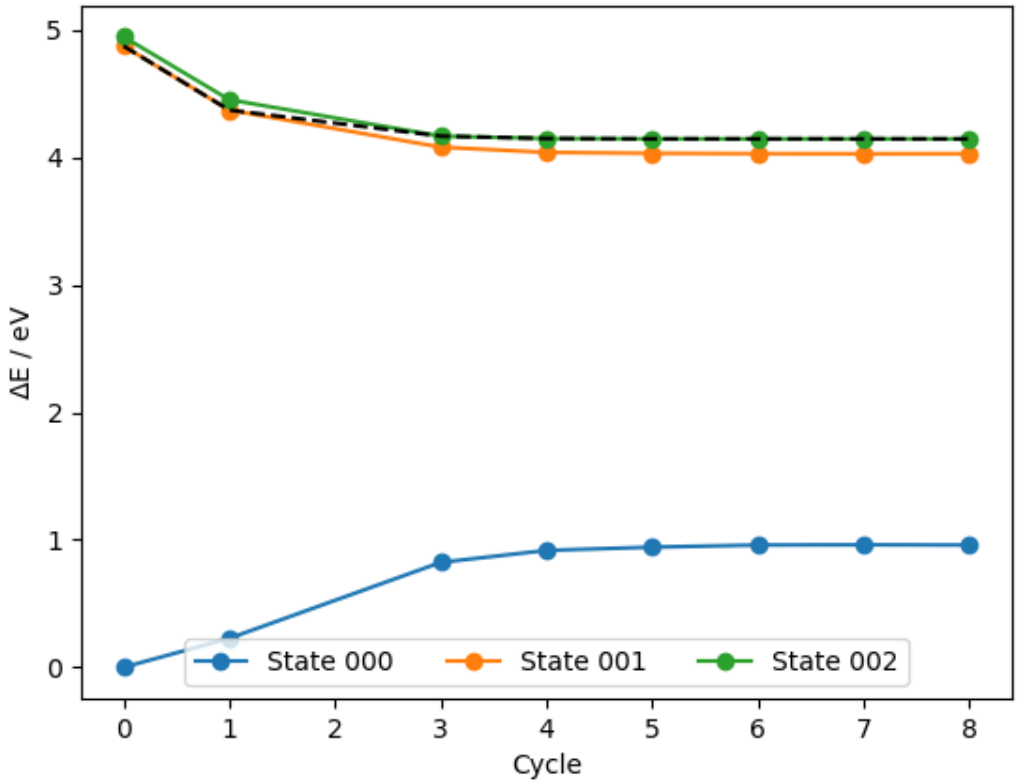

Fig. 15.11: Potential energy curves along the S_1 optimization of the 1H-amino-keto tautomer of Cytosin at the PBE0/def2-SVP level of theory. The root actually followed is indicated by a dashed line.

The calculated overlap matrices can be plotted by:

```
pysisplot --overlaps
pysisplot -o
```

If the calculation was set up to calculate charge-density-differences (CDDs) via MultiWFN and to render them via Jmol then the CDD images displayed beside the overlap matrices.

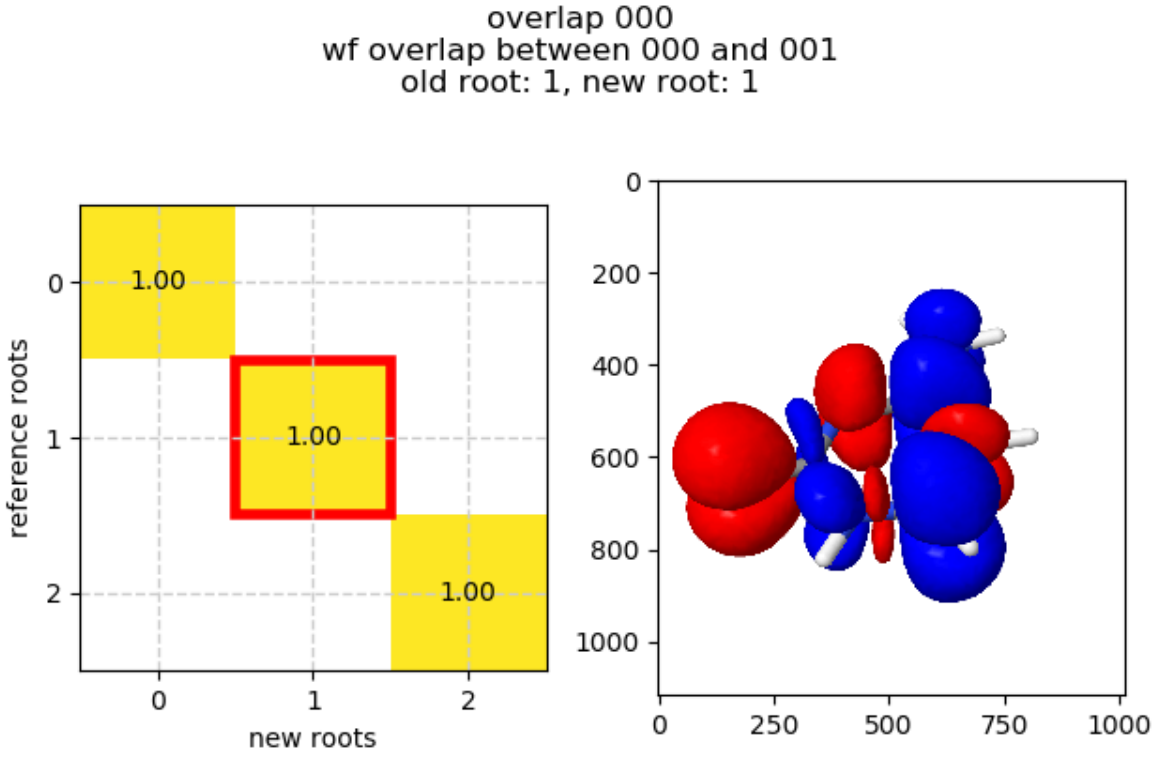

Fig. 15.12: Cytosin S_1 optimization. Overlaps between first and second cycle. No root flips occured. All ES are in the same order as in the reference geometry at cycle 0.

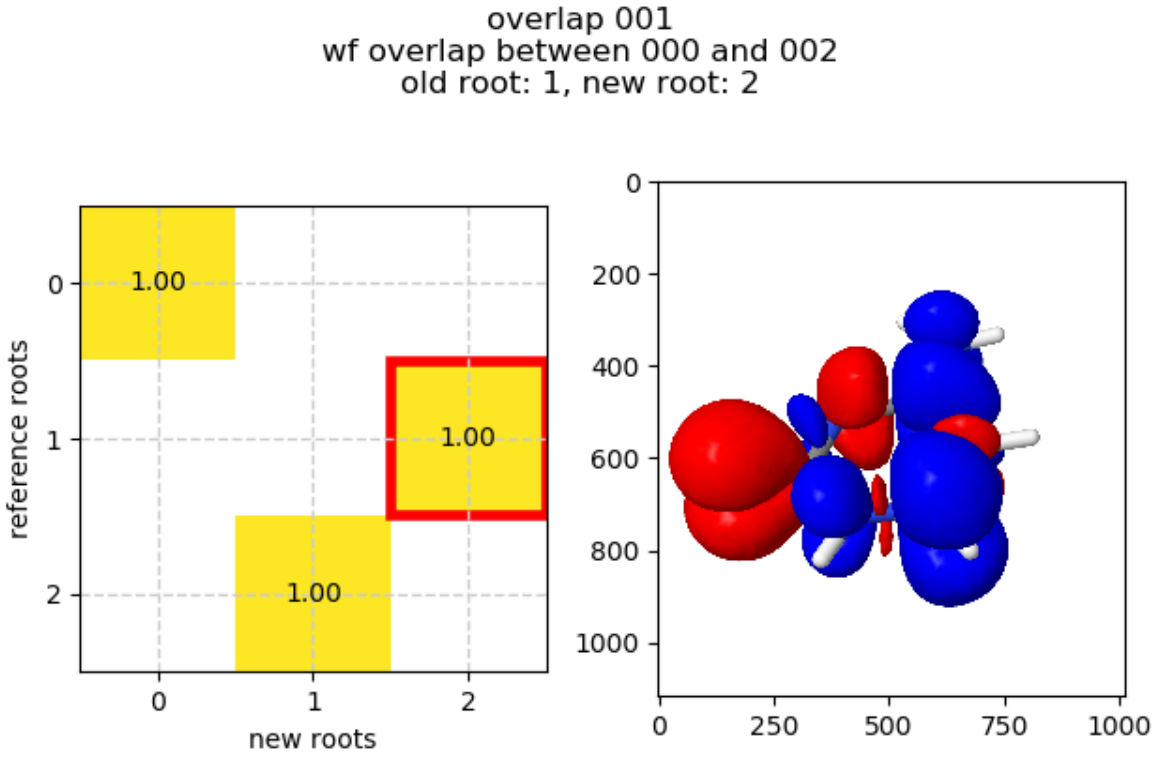

Fig. 15.13: Cytosin S_1 optimization. Overlaps between second and third cycle with root flip. The S_1 and S_2 switch their order.

CHAPTER

SIXTEEN

WORKING WITH GEOMETRIES

TBD

Pleas see *pysistrj --help* for a list of possible invocations.

```
usage: Utility to transform .xyz and .trj files. [-h]
                                                 (--between BETWEEN | --align | --split␣
↪| --reverse | --cleantrj | --spline | --first FIRST | --every EVERY | --center | --
↪centerm | --translate TRANSLATE TRANSLATE TRANSLATE | --append | --join | --match | --
↪std | --shake | --internals | --get GET | --origin | --fragsort)
                                                 [--scale SCALE] [--seed SEED]
                                                 [--idpp | --lst | --redund]
                                                 [--noipalign] [--bohr]
                                                 [--noxyz]
                                                 [--atoms ATOMS [ATOMS ...]]
                                                 [--add_prims ADD_PRIMS]
                                                 fns [fns ...]

positional arguments:
  fns                   Filenames of .xyz and/or .trj files (xyz and trj can
                        be mixed).

optional arguments:
  -h, --help            show this help message and exit
  --between BETWEEN     Interpolate additional images.
  --align               Align geometries onto the first geometry.
  --split               Split a supplied geometries in multiple .xyz files.
  --reverse             Reverse a .trj file.
  --cleantrj            Keep only the first four columns of xyz/trj files.
  --spline              Evenly redistribute geometries along a splined path.
  --first FIRST         Copy the first N geometries to a new .trj file.
  --every EVERY         Create new .trj with every N-th geometry. Always
                        includes the first and last point.
  --center              Move the molecules centroid into the origin.
  --centerm             Move the molecules center of mass into the origin.
  --translate TRANSLATE TRANSLATE TRANSLATE
                        Translate the molecule by the given vector given in
                        Ångström.
  --append              Combine the given .xyz files into one .xyz file.
  --join                Combine the given .xyz/.trj files into one .trj file.
  --match               Resort the second .xyz file so the atom order matches
                        the first .xyz file. Uses the hungarian method.
```

(continues on next page)

(continued from previous page)

```
  --std                 Move supplied geometry to its standard orientation.
  --shake               Shake (randomly displace) coordiantes.
  --internals           Print automatically generated internals.
  --get GET             Get n-th geometry. Expects 0-based index input.
  --origin              Translate geometry, so that min(X/Y/Z) == 0.
  --fragsort            Resort atoms by fragments.
  --idpp                Interpolate using Image Dependent Pair Potential.
  --lst                 Interpolate by linear synchronous transit.
  --redund              Interpolate in internal coordinates.
  --noipalign           Don't align geometries when interpolating.
  --bohr                Input geometries are in Bohr instead of Angstrom.
  --noxyz               Disable dumping of single .xyz files.
  --atoms ATOMS [ATOMS ...]
                        Used with --internals. Only print primitives including
                        the given atoms.
  --add_prims ADD_PRIMS
                        Used with --internals. Define additional primitives.
                        Expects a string representation of a nested list that
                        can be parsed as YAML e.g.
                        [[10,30],[1,2,3],[4,5,6,7]].

  --scale SCALE         Scales the displacement in --shake.
  --seed SEED           Initialize the RNG for reproducible results.
```

CHAPTER

SEVENTEEN

PYSISYPHUS

17.1 pysisyphus package

17.1.1 Subpackages

17.1.1.1 pysisyphus.benchmarks package

Submodules

pysisyphus.benchmarks.Benchmark module

class pysisyphus.benchmarks.Benchmark.**Benchmark**(*name*, *exclude=None*, *inv_exclude=False*, *only=None*, *coord_type='cart'*, *calc_getter=None*)

Bases: object

data_funcs = {'baker': <function get_baker_data>, 'baker_ts': <function get_baker_ts_data>, 'birkholz_rx': <function get_birkholz_rx_data>, 'precon_pos_rot': <function get_precon_pos_rot_data>, 's22': <function get_s22_data>, 'xtb_rx': <function get_xtb_rx_data>, 'zimmerman': <function get_zimmerman_data>, 'zimmerman_xtb': <function get_zimmerman_xtb_data>}

property geom_iter

property geoms

get_geoms(*id_*, *set_calculator=True*)

pysisyphus.benchmarks.data module

pysisyphus.benchmarks.data.**get_baker_data**()

10.1002/jcc.540140910

HF/STO-3G

pysisyphus.benchmarks.data.**get_baker_ts_data**()

10.1002/(SICI)1096-987X(199605)17:7<888::AID-JCC12>3.0.CO;2-7

HF/3-21G

pysisyphus.benchmarks.data.**get_birkholz_rx_data**()
https://doi.org/10.1002/jcc.23910

pysisyphus.benchmarks.data.**get_precon_pos_rot_data**()
https://doi.org/10.1002/jcc.23910

pysisyphus.benchmarks.data.**get_s22_data**()
https://doi.org/10.1039/B600027D

pysisyphus.benchmarks.data.**get_xtb_rx_data**()

pysisyphus.benchmarks.data.**get_zimmerman_data**()
https://dx.doi.org/10.1021/ct400319w

pysisyphus.benchmarks.data.**get_zimmerman_xtb_data**()

Reoptimization of
https://dx.doi.org/10.1021/ct400319w

at the gfn2-xtb level of theory.

Includes set 1 (first 72 entries), minus (0-based) ids
7, 8, 13, 14, 15, 34, 39

Module contents

17.1.1.2 pysisyphus.calculators package

Submodules

pysisyphus.calculators.AFIR module

class pysisyphus.calculators.AFIR.**AFIR**(*calculator*, *fragment_indices*, *gamma*, *rho=1*, *p=6*, *ignore_hydrogen=False*, *zero_hydrogen=True*, *complete_fragments=True*, *dump=True*, *h5_fn='afir.h5'*, *h5_group_name='afir'*, ***kwargs*)

Bases: *Calculator*

__init__(*calculator*, *fragment_indices*, *gamma*, *rho=1*, *p=6*, *ignore_hydrogen=False*, *zero_hydrogen=True*, *complete_fragments=True*, *dump=True*, *h5_fn='afir.h5'*, *h5_group_name='afir'*, ***kwargs*)

Artifical Force Induced Reaction calculator.

Currently, there are no automated drivers to run large-scale AFIR calculations with many different initial orientations and/or increasing collision energy parameter . Nontheless, selected AFIR calculations can be carried out by hand. After convergence, artificial potential & forces, as well as real energies and forces can be plotted with 'pysisplot --afir'. The highest energy point along the AFIR path can then be selected for a subsequent TS-optimization, e.g. via 'pysistrj --get [index] optimzation.trj'.

Future versions of pysisyphus may provide drivers for more automatted AFIR calculations.

Parameters

- **calculator** (*Calculator*) -- Actual QC calculator that provides energies and its derivatives, that are modified by the AFIR calculator, e.g., ORCA or Psi4.

- **fragment_indices** (List[List[int]]) -- List of lists of integers, specifying the separate fragments. If the indices in theses lists don't comprise all atoms in the molecule, the reamining indices will be added as a separate fragment. If a AFIR calculation is carried out with 2 fragments and 'complete_fragments' is True (see below) it is enough to specify only the indices of one fragment, e.g., for a system of 10 atoms 'fragment_indices=[[0,1,2,3]]' is enough. The second system will be set up automatically with indices [4,5,6,7,8,9].
- **gamma** (Union[_SupportsArray[dtype[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]]) -- Collision energy parameter in au. For 2 fragments it can be a single integer, while for > 2 fragments a list of gammas must be given, specifying the pair-wise collision energy parameters. For 3 fragments 3 gammas must be given [_01, _02, _12], for 4 fragments 6 gammas would be required [_01, _02, _03, _12, _13, _23] and so on.
- **rho** (Union[_SupportsArray[dtype[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]], default: 1) -- Direction of the artificial force, either 1 or -1. The same comments as for gamma apply. For 2 fragments a single integer is enough, for > 2 fragments a list of rhos must be given (see above). For rho=1 fragments are pushed together, for rho=-1 fragments are pulled apart.
- **p** (int, default: 6) -- Exponent p used in the calculation of the weight function . Defaults to 6 and probably does not have to be changed.
- **ignore_hydrogen** (bool, default: False) -- Whether hydrogens are ignored in the calculation of the artificial force. All weights between atom pairs containing hydrogen will be set to 0.
- **zero_hydrogen** (bool, default: True) -- Whether to use 0.0 as covalent radius for hydrogen in the weight function. Compared to 'ignore_hydrogen', which results in zero weights for all atom pairs involving hydrogen, 'zero_hydrogen' may be non-zero, depending on the covalent radius of the second atom in the pair.
- **complete_fragments** (bool, default: True) -- Whether an incomplete specification in 'fragment_indices' is automatically completed.
- **dump** (bool, default: True) -- Whether an HDF5 file is created.
- **h5_fn** (str, default: 'afir.h5') -- Filename of the HDF5 file used for dumping.
- **h5_group_name** (str, default: 'afir') -- HDF5 group name used for dumping.
- ****kwargs** -- Keyword arguments passed to the Calculator baseclass.

afir_fd_hessian_wrapper(*coords3d*, *afir_grad_func*)

property charge

dump_h5(*atoms*, *coords*, *results*)

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

init_h5_group(*atoms*, *max_cycles=None*)

log_fragments()

property mult

set_atoms_and_funcs(*atoms*, *coords*)

Initially atoms was also an argument to the constructor of AFIR. I removed it so creation becomes easier. The first time a calculation is requested with a proper atom set everything is set up (cov. radii, afir function and corresponding gradient). Afterwards there is only a check if atoms != None and it is expected that all functions are properly set.

fragment_indices can also be incomplete w.r.t. to the number of atoms. If the sum of the specified fragment atoms is less than the number of atoms present then all remaining unspecified atoms will be gathered in one fragment.

write_fragment_geoms(*atoms*, *coords*)

exception pysisyphus.calculators.AFIR.**CovRadiiSumZero**

Bases: Exception

pysisyphus.calculators.AFIR.**afir_closure**(*fragment_indices*, *cov_radii*, *gamma*, *rho=1*, *p=6*, *prefactor=1.0*, *logger=None*)

rho=1 pushes fragments together, rho=-1 pulls fragments apart.

pysisyphus.calculators.AFIR.**get_data_model**(*atoms*, *max_cycles*)

pysisyphus.calculators.AnaPot module

class pysisyphus.calculators.AnaPot.**AnaPot**(***kwargs*)

Bases: *AnaPotBase*

pysisyphus.calculators.AnaPot2 module

class pysisyphus.calculators.AnaPot2.**AnaPot2**

Bases: *AnaPotBase*

We can't use sympify as it replaces 1/tan by cot and this isn't supported by numpy when we call lambdify.

class pysisyphus.calculators.AnaPot2.**AnaPot2_**

Bases: *Calculator*

get_energy(*atoms*, *coords*)

Meant to be extended.

pysisyphus.calculators.AnaPot3 module

class pysisyphus.calculators.AnaPot3.**AnaPot3**

Bases: *AnaPotBase*

pysisyphus.calculators.AnaPot4 module

class pysisyphus.calculators.AnaPot4.**AnaPot4**

Bases: *AnaPotBase*

pysisyphus.calculators.AnaPotBase module

class pysisyphus.calculators.AnaPotBase.**AnaPotBase**(*V_str*, *scale=1.0*, *xlim=(-1, 1)*, *ylim=(-1, 1)*, *levels=None*, *use_sympify=True*, *minima=None*, *saddles=None*)

Bases: *Calculator*

anim_coords(*coords*, *interval=50*, *show=False*, *title_func=None*)

anim_opt(*opt*, *energy_profile=False*, *colorbar=False*, *figsize=(8, 6)*, *show=False*)

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

classmethod get_geom(*coords*, *atoms=('X',)*, *V_str=None*, *calc_kwargs=None*, *geom_kwargs=None*)

get_geoms_from_stored_coords(*coords_list*, *i=None*, *calc_kwargs=None*, *geom_kwargs=None*)

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_minima(*i=None*, *calc_kwargs=None*, *geom_kwargs=None*)

get_path(*num*, *minima_inds=None*)

get_saddles(*i=None*, *calc_kwargs=None*, *geom_kwargs=None*)

plot(*levels=None*, *show=False*, ***figkwargs*)

plot3d(*levels=None*, *show=False*, *zlim=None*, *vmin=None*, *vmax=None*, *resolution=100*, *rcount=50*, *ccount=50*, *nan_above=None*, *init_view=None*, *colorbar=False*, ***figkwargs*)

plot_coords(*xs*, *ys*, *enum=True*, *show=False*, *title=None*)

plot_eigenvalue_structure(*grid=50*, *levels=None*, *show=False*)

plot_geoms(*geoms*, ***kwargs*)

plot_irc(*irc*, **args*, ***kwargs*)

plot_opt(*opt*, **args*, ***kwargs*)

statistics()

pysisyphus.calculators.AnaPotCBM module

class pysisyphus.calculators.AnaPotCBM.**AnaPotCBM**

Bases: *AnaPotBase*

pysisyphus.calculators.AtomAtomTransTorque module

class pysisyphus.calculators.AtomAtomTransTorque.**AtomAtomTransTorque**(*geom*, *frags*, *A_mats*, *kappa=2.0*)

Bases: object

__init__(*geom*, *frags*, *A_mats*, *kappa=2.0*)

Atom-atom translational and torque forces.

See A.5. [1], Eq. (A6).

get_forces(*atoms*, *coords*)

pysisyphus.calculators.Calculator module

class pysisyphus.calculators.Calculator.**Calculator**(*calc_number=0*, *charge=0*, *mult=1*, *base_name='calculator'*, *pal=1*, *mem=1000*, *keep_kind='all'*, *check_mem=True*, *retry_calc=0*, *last_calc_cycle=None*, *clean_after=True*, *out_dir='qm_calcs'*, *force_num_hess=False*, *num_hess_kwargs=None*)

Bases: object

__init__(*calc_number=0*, *charge=0*, *mult=1*, *base_name='calculator'*, *pal=1*, *mem=1000*, *keep_kind='all'*, *check_mem=True*, *retry_calc=0*, *last_calc_cycle=None*, *clean_after=True*, *out_dir='qm_calcs'*, *force_num_hess=False*, *num_hess_kwargs=None*)

Base-class of all calculators.

Meant to be extended.

Parameters

- **calc_number** (*int, default=0*) -- Identifier of the Calculator. Used in distinguishing it from other Calculators, e.g. in ChainOfStates calculations. Also used in the creation of filenames.
- **charge** (*int, default=0*) -- Molecular charge.
- **mult** (*int, default=1*) -- Molecular multiplicity (1 = singlet, 2 = doublet, ...)
- **base_name** (*str, default=calculator*) -- Generated filenames will start with this string.
- **pal** (*int, default=1*) -- Positive integer that gives the number of physical cores to use on 1 node.

- **mem** (*int, default=1000*) -- Mememory per core in MB. The total amount of memory is given as mem*pal.
- **check_mem** (*bool, default=True*) -- Whether to adjust the requested memory if too much is requested.
- **retry_calc** (*int, default=0*) -- Number of additional retries when calculation failed.
- **last_calc_cycle** (*int*) -- Internal variable used in restarts.
- **clean_after** (*bool*) -- Delete temporary directory were calculations were executed after a calculation.
- **out_dir** (*str*) -- Path that is prepended to generated filenames.
- **force_hess_kwargs** (*bool, default False*) -- Force numerical Hessians.
- **num_hess_kwargs** (*dict*) -- Keyword arguments for finite difference Hessian calculation.

apply_keep_kind()

apply_set_plans(*kept_fns*, *set_plans=None*)

build_set_plans(*_set_plans=None*)

clean(*path*)

Delete the temporary directory.

Parameters
path (*Path*) -- Directory to delete.

conf_key = None

force_num_hessian()

Always calculate numerical Hessians.

classmethod geom_from_fn(*fn*, ***kwargs*)

get_cmd(*key='cmd'*)

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_num_hessian(*atoms*, *coords*, ***prepare_kwargs*)

get_restart_info()

Return a dict containing chkfiles.

Returns
restart_info -- Dictionary holding the calculator state. Used for restoring calculaters in restarted calculations.

Return type
dict

keep(*path*)

Backup calculation results.

Parameters
path (*Path*) -- Temporary directory of the calculation.

Returns
kept_fns -- Dictonary holding the filenames that were backed up. The keys correspond to the type of file.

Return type
dict

log(*message=''*)

Write a log message.

Wraps the logger variable.

Parameters
message (*str*) -- Message to be logged.

make_fn(*name*, *counter=None*, *return_str=False*)

Make a full filename.

Return a full filename including the calculator name and the current counter given a suffix.

Parameters

- **name** (*str*) -- Suffix of the filename.
- **counter** (*int, optional*) -- If not given use the current calc_counter.
- **return_str** (*int, optional*) -- Return a string instead of a Path when True.

Returns
fn -- Filename.

Return type
str

property name

popen(*cmd*, *cwd=None*)

prepare(*inp*)

Prepare a temporary directory and write input.

Similar to prepare_path, but the input is also written into the prepared directory.

Paramters

inp
[str] Input to be written into the file `self.inp_fn` in the prepared directory.

returns

path: Path
Prepared directory.

prepare_coords(*atoms*, *coords*)

Get 3d coords in Angstrom.

Reshape internal 1d coords to 3d and convert to Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns
coords -- 3D-array holding coordinates in Angstrom.

Return type
np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

prepare_path(*use_in_run=False*)

Get a temporary directory handle.

Create a temporary directory that can later be used in a calculation.

Parameters
use_in_run (*bool, option*) -- Sets the internal variable `self.path_already_prepared` that is later read by `self.run()`. No new temporary directory will be created in `self.run()`.

Returns

path: Path
Prepared directory.

prepare_pattern(*raw_pat*)

Prepare globs.

Transforms an entry of `self.to_keep` into a glob and a key suitable for the use in `self.keep()`.

Parameters
raw_pat (*str*) -- Entry of `self.to_keep`

Returns

- **pattern** (*str*) -- Glob that can be used in Path.glob()
- **multi** (*bool*) -- Flag if glob may match multiple files.
- **key** (*str*) -- A key to be used in the `kept_fns` dict.

prepare_turbo_coords(*atoms*, *coords*)

Get a Turbomole coords string.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns
coords -- String holding coordinates in Turbomole coords format.

Return type
str

prepare_xyz_string(*atoms*, *coords*)

Returns a xyz string in Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns

xyz_str -- Coordinates in .xyz format.

Return type

string

print_capabilities()

print_out_fn(*path*)

Print calculation output.

Prints the output of a calculator after a calculation.

Parameters

path (*Path*) -- Temporary directory of the calculation.

restore_org_hessian()

Restore original 'get_hessian' method, which may also fallback to numerical Hessians, if not implemented.

run(*inp*, *calc*, *add_args=None*, *env=None*, *shell=False*, *hold=False*, *keep=True*, *cmd=None*, *inc_counter=True*, *run_after=True*, *parser_kwargs=None*, *symlink=True*)

Run a calculation.

The bread-and-butter method to actually run an external quantum chemistry code.

Parameters

- **inp** (*str*) -- Input for the external program that is written to the temp-dir.
- **calc** (*str, hashable*) -- Key (and more or less type of calculation) to select the right parsing function from `self.parser_funcs`.
- **add_args** (*iterable, optional*) -- Additional arguments that will be appended to the program call.
- **env** (*Environment, optional*) -- A potentially modified environment for the subprocess call.
- **shell** (*bool, optional*) -- Use a shell to execute the program call. Need for Turbomole were we chain program calls like dscf; escf.
- **hold** (*bool, optional*) -- Wether to remove the temporary directory after the calculation.
- **keep** (*bool, optional*) -- Wether to backup files as specified in `self.to_keep()`. Usually you want this.
- **cmd** (*str or iterable, optional*) -- Overwrites `self.base_cmd`.
- **inc_counter** (*bool, optional*) -- Wether to increment the counter after a calculation.

Returns

results -- Dictionary holding all applicable results of the calculations like the energy, a forces vector and/or excited state energies from TDDFT.

Return type
dict

run_after(*path*)

Meant to be extended.

This method is called after a calculation was done, but before entering `self.keep()` and `self.clean()`. Can be used to call tools like formchk or ricctools.

set_restart_info(*restart_info*)

Sets restart information (chkfiles etc.) on the calculator.

Parameters
restart_info (*dict*) -- Dictionary holding the calculator state. Used for restoring calculaters in restarted calculations.

verify_chkfiles(*chkfiles*)

Checks if given chkfiles exist and return them as Paths

Parameters
chkfiles (*dict*) -- Dictionary holding the chkfiles. The keys correspond to the attribute names, the values are strs holding the (potentially full) filename (path).

Returns
paths -- Dictionary of Paths.

Return type
dict

class pysisyphus.calculators.Calculator.**HessKind**(*value*)

Bases: Enum

An enumeration.

NUMERICAL = 2

ORG = 1

class pysisyphus.calculators.Calculator.**KeepKind**(*value*)

Bases: Enum

An enumeration.

ALL = 1

LATEST = 2

NONE = 3

class pysisyphus.calculators.Calculator.**SetPlan**(*key*, *name=None*, *condition=<function SetPlan.<lambda>>*, *fail=None*)

Bases: object

condition()

fail: Optional[Callable] = None

key: str

name: Optional[str] = None

pysisyphus.calculators.CerjanMiller module

class pysisyphus.calculators.CerjanMiller.**CerjanMiller**(*a=1*, *b=1*, *c=1*)

Bases: *AnaPotBase*

pysisyphus.calculators.Composite module

class pysisyphus.calculators.Composite.**Composite**(*final*, *keys_calcs=None*, *calcs=None*, *remove_translation=False*, ***kwargs*)

Bases: *Calculator*

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_final_energy(*energies*)

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

pysisyphus.calculators.ConicalIntersection module

class pysisyphus.calculators.ConicalIntersection.**CIQuantities**(*energy1*, *gradient1*, *energy2*, *gradient2*, *energy_diff*, *gradient_diff*, *gradient_mean*, *P*, *x*, *y*, *energy*, *forces*)

Bases: object

P: ndarray

energy: float

energy1: float

energy2: float

energy_diff: float

forces: ndarray

gradient1: ndarray

gradient2: ndarray

gradient_diff: ndarray

gradient_mean: ndarray

x: ndarray

y: ndarray

class pysisyphus.calculators.ConicalIntersection.**ConicalIntersection**(*calculator1*, *calculator2*, ***kwargs*)

Bases: *Calculator*

Calculator for conical intersection optimization.

Based on [1].

get_ci_quantities(*atoms*, *coords*, ***prepare_kwargs*)

Relavent quantities including branching plane and projector P.

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Energy of calculator 1.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Projected gradient for CI optimization.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Projected Hessian.

pysisyphus.calculators.ConicalIntersection.**get_P**(*x*, *y*)

Projector to project out components in branching plane.

pysisyphus.calculators.ConicalIntersection.**update_y**(*x*, *x_prev*, *y_prev*)

Update approximate coupling derivative vector y.

pysisyphus.calculators.DFTBp module

class pysisyphus.calculators.DFTBp.**DFTBp**(*parameter*, **args*, *slakos=None*, *root=None*, ***kwargs*)

Bases: *OverlapCalculator*

conf_key = 'dftbp'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

static get_excited_state_str(*root*, *forces=False*)

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

static get_gen_str(*atoms*, *coords*)

hubbard_derivs = {'3ob': {'Br': -0.0573, 'C': -0.1492, 'Ca': -0.034, 'Cl': -0.0697, 'F': -0.1623, 'H': -0.1857, 'I': -0.0433, 'K': -0.0339, 'Mg': -0.02, 'N': -0.1535, 'Na': -0.0454, 'O': -0.1575, 'P': -0.14, 'S': -0.11, 'Zn': -0.03}}

max_ang_moms = {'3ob': {'Br': 'd', 'C': 'p', 'Ca': 'p', 'Cl': 'd', 'F': 'p', 'H': 's', 'I': 'd', 'K': 'p', 'Mg': 'p', 'N': 'p', 'Na': 'p', 'O': 'p', 'P': 'd', 'S': 'd', 'Zn': 'd'}, 'mio-ext': {'C': 'p', 'H': 's', 'N': 'p', 'O': 'p'}}

parse_all_energies(*out_fn=None*, *exc_dat=None*)

parse_energy(*path*)

static parse_exc_dat(*text*)

parse_forces(*path*)

parse_total_energy(*text*)

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

pysisyphus.calculators.DFTBp.**parse_mo**(*eigvec*)

pysisyphus.calculators.DFTBp.**parse_xplusy**(*text*)

pysisyphus.calculators.Dalton module

class pysisyphus.calculators.Dalton.**Dalton**(*basis*, *method='hf'*, ***kwargs*)

Bases: *Calculator*

compute(*mol*, *prop*)

conf_key = 'dalton'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

prepare_input(*atoms*, *coords*)

Meant to be extended.

pysisyphus.calculators.Dimer module

class pysisyphus.calculators.Dimer.**Dimer**(*calculator*, **args*, *N_raw=None*, *length=0.0189*, *rotation_max_cycles=15*, *rotation_method='fourier'*, *rotation_thresh=0.0001*, *rotation_tol=1*, *rotation_max_element=0.001*, *rotation_interpolate=True*, *rotation_disable=False*, *rotation_disable_pos_curv=True*, *rotation_remove_trans=True*, *trans_force_f_perp=True*, *bonds=None*, *N_hessian=None*, *bias_rotation=False*, *bias_translation=False*, *bias_gaussian_dot=0.1*, *seed=None*, *write_orientations=True*, *forward_hessian=True*, ***kwargs*)

Bases: *Calculator*

property C

Shortcut for the curvature.

property N

add_gaussian(*atoms*, *center*, *N*, *height=0.1*, *std=0.0529*, *max_cycles=50*, *dot_ref=None*)

property can_bias_f0

property can_bias_f1

property coords0

property coords1

curvature(*f1*, *f2*, *N*)

Curvature of the mode represented by the dimer.

direct_rotation(*optimizer*, *prev_step*)

do_dimer_rotations(*rotation_thresh=None*)

property energy0

property f0

property f1

property f1_bias

property f2

Never calculated explicitly, but estimated from f0 and f1.

fourier_rotation(*optimizer*, *prev_step*)

get_N_raw_from_hessian(*h5_fn*, *root=0*)

get_forces(*atoms*, *coords*)

Meant to be extended.

get_gaussian_energies(*coords*, *sum_=True*)

get_gaussian_forces(*coords*, *sum_=True*)

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

remove_translation(*displacement*)

property rot_force

rotate_coords1(*rad*, *theta*)

Rotate dimer and produce new coords1.

set_N_raw(*coords*)

property should_bias_f0

May lead to calculation of f0 and/or f1 if present!

property should_bias_f1

May lead to calculation of f0 and/or f1 if present!

update_orientation(*coords*)

class pysisyphus.calculators.Dimer.**Gaussian**(*height*, *center*, *std*, *N*)

Bases: object

energy(*R*, *height=None*)

forces(*R*, *height=None*)

exception pysisyphus.calculators.Dimer.**RotationConverged**

Bases: Exception

pysisyphus.calculators.Dummy module

class pysisyphus.calculators.Dummy.**Dummy**(*calc_number=0*, *charge=0*, *mult=1*, *base_name='calculator'*, *pal=1*, *mem=1000*, *keep_kind='all'*, *check_mem=True*, *retry_calc=0*, *last_calc_cycle=None*, *clean_after=True*, *out_dir='qm_calcs'*, *force_num_hess=False*, *num_hess_kwargs=None*)

Bases: *Calculator*

get_energy(**args*, ***kwargs*)

Meant to be extended.

get_forces(**args*, ***kwargs*)

Meant to be extended.

get_hessian(**args*, ***kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

raise_exception()

run_calculation(**args*, ***kwargs*)

pysisyphus.calculators.EGO module

class pysisyphus.calculators.EGO.**EGO**(*calculator*, *ref_geom*, *max_force=0.175*, ***kwargs*)

Bases: *Calculator*

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_mods(*atoms*, *coords*)

property ref_hessian

property s

pysisyphus.calculators.EnergyMin module

class pysisyphus.calculators.EnergyMin.**EnergyMin**(*calculator1*, *calculator2*, *mix=False*, *alpha=0.02*, *sigma=3.5*, *min_energy_diff=0.0*, *check_after=0*, ***kwargs*)

Bases: *Calculator*

__init__(*calculator1*, *calculator2*, *mix=False*, *alpha=0.02*, *sigma=3.5*, *min_energy_diff=0.0*, *check_after=0*, ***kwargs*)

Use energy and derivatives of the calculator with lower energy.

This calculators carries out two calculations with different settings and returns the results of the lower energy one. This can be used to consider flips between a singlet and a triplet PES etc.

Parameters

- **calculator1** (*Calculator*) -- Wrapped QC calculator that provides energies and its derivatives.
- **calculator2** (*Calculator*) -- Wrapped QC calculator that provides energies and its derivatives.
- **mix** (bool, default: False) -- Enable mixing of both forces, according to the approach outlined in [2]. Can be used to optimize guesses for MECPs. Pass
- **alpha** (float, default: 0.02) -- Smoothing parameter in Hartree. See [2] for a discussion.
- **sigma** (float, default: 3.5) -- Unitless gap size parameter. The final gap becomes smaller for bigga sigmas. Has to be adapted for each case. See [2] for a discussion (p. 407 right column and p. 408 left column.)
- **min_energy_diff** (float, default: 0.0) -- Energy difference in Hartree. When set to a value != 0 and the energy difference between both calculators drops below this value, execution of both calculations is diabled for 'check_after' cycles. In these cycles the calculator choice remains fixed. After 'check_after' cycles, both energies will be calculated and it is checked, if the previous calculator choice remains valid. In conjunction with 'check_after' both arguments can be used to save computational ressources.
- **check_after** (int, default: 0) -- Amount of cycles in which the calculator choice remains fixed.
- ****kwargs** -- Keyword arguments passed to the Calculator baseclass.

do_calculations(*name*, *atoms*, *coords*, ***prepare_kwargs*)

get_chkfiles()

Return type
dict

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

set_chkfiles(*chkfiles*)

pysisyphus.calculators.ExternalPotential module

class pysisyphus.calculators.ExternalPotential.**ExternalPotential**(*calculator=None*, *potentials=None*, *geom=None*, ***kwargs*)

Bases: *Calculator*

available_potentials = {'d3': <class 'pysisyphus.calculators.DFTD3.DFTD3'>, 'harmonic_sphere': <class 'pysisyphus.calculators.ExternalPotential.HarmonicSphere'>, 'logfermi': <class 'pysisyphus.calculators.ExternalPotential.LogFermi'>, 'restraint': <class 'pysisyphus.calculators.ExternalPotential.Restraint'>, 'rmsd': <class 'pysisyphus.calculators.ExternalPotential.RMSD'>}

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_potential_energy(*coords*)

get_potential_forces(*coords*)

class pysisyphus.calculators.ExternalPotential.**HarmonicSphere**(*k*, *radius*, *origin=(0.0, 0.0, 0.0)*, *geom=None*)

Bases: object

calc(*coords3d*, *gradient=False*)

instant_pressure(*coords3d*)

property surface_area

In Bohr**2

class pysisyphus.calculators.ExternalPotential.**LogFermi**(*beta*, *radius*, *T=300*, *origin=(0.0, 0.0, 0.0)*, *geom=None*)

Bases: object

__init__(*beta*, *radius*, *T=300*, *origin=(0.0, 0.0, 0.0)*, *geom=None*)

As described in the XTB docs.

https://xtb-docs.readthedocs.io/en/latest/xcontrol.html#confining-in-a-cavity

calc(*coords3d*, *gradient=False*)

class pysisyphus.calculators.ExternalPotential.**RMSD**(*geom*, *k*, *beta=0.5*, *atom_indices=None*)

Bases: object

__init__(*geom*, *k*, *beta=0.5*, *atom_indices=None*)

Restrain based on RMSD with a reference geometry.

As described in https://doi.org/10.1021/acs.jctc.0c01306, Eq. (5).

Parameters

- **geom** (*Geometry*) -- Reference geometry for RMSD calculation.
- **k** (float) -- Gaussian height in units of energy. Should be a negative number if the system under study should stay close to the reference geometry (pulling k). A positive Gaussian height k results in forces that push the system under study away from the reference geometry (pushing k).
- **b** -- Gaussian width in inverse units of lengths.
- **atom_indices** (Union[_SupportsArray[dtype[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]], None], default: None) -- Optional, numpy array or iterable of integer atom indices. Restricts the RMSD calculation to these atoms. If omitted, all atoms are used.

calc(*coords3d*, *gradient=False*)

class pysisyphus.calculators.ExternalPotential.**Restraint**(*restraints*, *geom=None*)

Bases: object

calc(*coords3d*, *gradient=False*)

static calc_prim_restraint(*prim*, *coords3d*, *force_const*, *ref_val*)

pysisyphus.calculators.FakeASE module

class pysisyphus.calculators.FakeASE.**FakeASE**(*calc*)

Bases: object

get_atoms_coords(*atoms*)

get_forces(*atoms=None*)

get_potential_energy(*atoms=None*)

pysisyphus.calculators.FourWellAnaPot module

class pysisyphus.calculators.FourWellAnaPot.**FourWellAnaPot**

Bases: *AnaPotBase*

pysisyphus.calculators.FreeEndNEBPot module

class pysisyphus.calculators.FreeEndNEBPot.**FreeEndNEBPot**

Bases: *AnaPotBase*

__init__()

Analyitcal potential as described in [1] Appendix A

pysisyphus.calculators.Gaussian09 module

class pysisyphus.calculators.Gaussian09.**Gaussian09**(**args*, ***kwargs*)

Bases: *Gaussian16*

conf_key = 'gaussian09'

pysisyphus.calculators.Gaussian16 module

class pysisyphus.calculators.Gaussian16.**Gaussian16**(*route*, *gbs=''*, *gen=''*, *keep_chk=False*, *stable=''*, *fchk=None*, ***kwargs*)

Bases: *OverlapCalculator*

conf_key = 'gaussian16'

get_chkfiles()

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

make_exc_str()

make_fchk(*path*)

make_gbs_str()

parse_635r_dump(*dump_path*, *roots*, *nmos*)

parse_all_energies(*fchk=None*)

parse_charges(*path=None*)

parse_double_mol(*path*, *out_fn=None*)

static parse_fchk(*fchk_path*, *keys*)

parse_force(*path*)

parse_hessian(*path*)

parse_keyword(*text*)

parse_log(**args*, ***kwargs*)

parse_stable(*path*)

parse_tddft(*path*)

prepare_input(*atoms*, *coords*, *calc_type*, *did_stable=False*, *point_charges=None*)

Meant to be extended.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

reuse_data(*path*)

run_after(*path*)

Meant to be extended.

This method is called after a calculation was done, but before entering `self.keep()` and `self.clean()`. Can be used to call tools like formchk or ricctools.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

run_double_mol_calculation(*atoms*, *coords1*, *coords2*)

run_rwfdump(*path*, *rwf_index*, *chk_path=None*)

run_stable(*atoms*, *coords*, ***prepare_kwargs*)

set_chkfiles(*chkfiles*)

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

pysisyphus.calculators.HardSphere module

class pysisyphus.calculators.HardSphere.**HardSphere**(*geom*, *frags*, *kappa=1.0*, *permutations=False*, *frag_radii=None*, *radii_offset=0.9452*)

Bases: `object`

__init__(*geom*, *frags*, *kappa=1.0*, *permutations=False*, *frag_radii=None*, *radii_offset=0.9452*)

Intra-Image Inter-Molecular Hard-Sphere force.

See A.2. in [1], Eq. (A1).

get_forces(*atoms*, *coords*, *kappa=None*)

class pysisyphus.calculators.HardSphere.**PWHardSphere**(*geom*, *frags*, *sub_frags*, *kappa=1.0*)

Bases: `object`

__init__(*geom*, *frags*, *sub_frags*, *kappa=1.0*)

Inter-Molecular pairwise Hard-Sphere forces between atoms.

Hardsphere forces are only applied between certain atoms of given fragments, but the whole fragment is moved. Can be used to remove atom inter-molecular atom clashes.

get_forces(*atoms*, *coords*, *kappa=None*)

pysisyphus.calculators.IDPPCalculator module

class pysisyphus.calculators.IDPPCalculator.**IDPPCalculator**(*target*)

Bases: *Calculator*

get_forces(*atoms*, *coords*)

Meant to be extended.

pysisyphus.calculators.IPIClient module

pysisyphus.calculators.IPIClient.**calc_ipi_client**(*addr*, *atoms*, *calc*, *queue=None*, ***kwargs*)

pysisyphus.calculators.IPIClient.**ipi_client**(*addr*, *atoms*, *energy_getter*, *forces_getter*, *hessian_getter=None*, *hdrlen=12*)

pysisyphus.calculators.IPIServer module

class pysisyphus.calculators.IPIServer.**IPIServer**(**args*, *address=None*, *host=None*, *port=None*, *unlink=True*, *hdrlen=12*, *max_retries=0*, *verbose=False*, ***kwargs*)

Bases: *Calculator*

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

listen_for(*atoms*, *coords*, *kind='forces'*)

listen_for_client_atom_num(*atom_num*)

listen_for_energy()

listen_for_forces(*atom_num*)

listen_for_hessian(*atom_num*)

listen_kinds = ('coords', 'energy', 'forces', 'hessian')

reset_client_connection()

retried_listen_for(*atoms*, *coords*)

unlink(*address*)

pysisyphus.calculators.LEPSBase module

class pysisyphus.calculators.LEPSBase.**LEPSBase**(*pot_type='leps'*)

Bases: *AnaPotBase*

pysisyphus.calculators.LEPSExpr module

class pysisyphus.calculators.LEPSExpr.**LEPSExpr**

Bases: object

G(*a*, *b*)

Gaussian function.

Gdimer(*x*, *y*, *A*, *x0*, *y0*, *sx*, *sy*)

J(*d*, *alpha*, *r0*, *r*)

Quantum mechanical exchange interaction.

Q(*d*, *alpha*, *r0*, *r*)

Coulomb interactions.

V_LEPS(*x=None*, *y=None*, *abc=None*)

Equation (A.1) in [1]. Mimics reaction involving three atoms confined to motion along a line.

V_dimer()

III. Results Section A in [3]. Two additional saddle points from two added gaussians.

V_harmonic()

Equation (A.2) in [1]. A and C are fixed, only B can move. A condensed phase environment is represented by adding a harmonic oscillator degree of freedom.

V_tot()

Equation (A.3) in [1]. Additional saddle point.

__init__()

Generates sympy expression for various LEPS potentials.

get_dimer()

get_expr(*pot_type='leps'*)

get_harmonic()

get_leps()

get_tot()

pysisyphus.calculators.LennardJones module

class pysisyphus.calculators.LennardJones.**LennardJones**(*sigma=1.8897261251*, *epsilon=1*, *rc=None*)

Bases: *Calculator*

calculate(*coords3d*)

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

pysisyphus.calculators.MOPAC module

class pysisyphus.calculators.MOPAC.**MOPAC**(*method='PM7'*, ***kwargs*)

Bases: *Calculator*

http://openmopac.net/manual/

CALC_TYPES = {'energy': '1SCF', 'gradient': '1SCF GRADIENTS', 'hessian': 'DFORCE FORCE LET'}

METHODS = ['am1', 'pm3', 'pm6', 'pm6-dh2', 'pm6-d3', 'pm6-dh+', 'pm6-dh2', 'pm6-dh2x', 'pm6-d3h4', 'pm6-d3h4x', 'pm7', 'pm7-ts']

MULT_STRS = {1: 'SINGLET', 2: 'DOUBLET', 3: 'TRIPLET', 4: 'QUARTET', 5: 'QUINTET', 6: 'SEXTET', 7: 'SEPTET', 8: 'OCTET'}

base_cmd

Do only SCF AUX: Creates a checkpoint file NOREO: Dont reorient geometry

Type
1SCF

conf_key = 'mopac'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

parse_energy(*path*)

static parse_energy_from_aux(*inp*, **args*, ***kwargs*)

parse_grad(*path*)

parse_hessian(*path*)

static parse_hessian_from_aux(*inp*, **args*, ***kwargs*)

prepare_coords(*atoms*, *coords*, *opt=False*)

Get 3d coords in Angstrom.

Reshape internal 1d coords to 3d and convert to Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns
coords -- 3D-array holding coordinates in Angstrom.

Return type
np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*, *opt=False*)

Meant to be extended.

read_aux(*path*)

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

pysisyphus.calculators.MullerBrownSympyPot module

class pysisyphus.calculators.MullerBrownSympyPot.**MullerBrownPot**

Bases: *AnaPotBase*

pysisyphus.calculators.MultiCalc module

class pysisyphus.calculators.MultiCalc.**MultiCalc**(*calcs*, ***kwargs*)

Bases: *Calculator*

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

pysisyphus.calculators.MultiCalc.**calcs_from_dict**(*calc_dict*, *base_name*, *calc_number*, *charge*, *mult*, *pal*, *mem*)

pysisyphus.calculators.OBabel module

class pysisyphus.calculators.OBabel.**OBabel**(*ff='gaff'*, *mol=None*, ***kwargs*)

Bases: *Calculator*

conv_dict = {'kcal/mol': 627.5094740630558, 'kj/mol': 2625.4996394798254}

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

setup(*atoms*, *coords*)

pysisyphus.calculators.ONIOMv2 module

class pysisyphus.calculators.ONIOMv2.**LayerCalc**(*models*, *total_size*, *parent_layer_calc=None*)

Bases: object

property charge

do_parent(*with_parent*)

static energy_from_results(*model_energies*, *parent_energy=None*)

get_energy(*atoms*, *coords*, *with_parent=True*)

get_forces(*atoms*, *coords*, *with_parent=True*)

get_hessian(*atoms*, *coords*, *with_parent=True*)

property mult

run_calculations(*atoms*, *coords*, *method*)

class pysisyphus.calculators.ONIOMv2.**Link**(*ind*, *parent_ind*, *atom*, *g*)

Bases: tuple

atom

Alias for field number 2

g

Alias for field number 3

ind

Alias for field number 0

parent_ind

Alias for field number 1

class pysisyphus.calculators.ONIOMv2.**Model**(*name*, *calc_level*, *calc*, *parent_name*, *parent_calc_level*, *parent_calc*, *atom_inds*, *parent_atom_inds*, *use_link_atoms=True*)

Bases: object

as_calculator(*cap=False*)

as_geom(*all_atoms*, *all_coords*)

capped_atoms_coords(*all_atoms*, *all_coords*)

create_bond_vec_getters(*atoms*)

create_links(*atoms*, *coords*, *debug=False*)

get_energy(*atoms*, *coords*, *point_charges=None*, *parent_correction=True*, *cap=True*)

get_forces(*atoms*, *coords*, *point_charges=None*, *parent_correction=True*, *cap=True*)

get_hessian(*atoms*, *coords*, *point_charges=None*, *parent_correction=True*, *cap=True*)

get_jacobian()

get_sparse_jacobian()

log(*message=''*)

parse_charges()

class pysisyphus.calculators.ONIOMv2.ModelDummyCalc(*model*, *cap=False*)

Bases: object

get_energy(*atoms*, *coords*)

get_forces(*atoms*, *coords*)

class pysisyphus.calculators.ONIOMv2.ONIOM(*calcs*, *models*, *geom*, *layers=None*, *embedding=''*, *real_key='real'*, *use_link_atoms=True*, **args*, ***kwargs*)

Bases: *Calculator*

__init__(*calcs*, *models*, *geom*, *layers=None*, *embedding=''*, *real_key='real'*, *use_link_atoms=True*, **args*, ***kwargs*)

layer: list of models
len(layer) == 1: normal ONIOM, len(layer) >= 1: multicenter ONIOM.

model:
(sub)set of all atoms that resides in a certain layer and has a certain calculator.

atom_inds_in_layer(*index*, *exclude_inner=False*)

Returns list of atom indices in layer at index.

Atoms that also appear in inner layer can be excluded on request.

Parameters

- **index** (*int*) -- pasd
- **exclude_inner** (*bool, default=False, optional*) -- Whether to exclude atom indices that also appear in inner layers.

Returns
atom_indices -- List containing the atom indices in the selected layer.

Return type
list

calc_layer(*atoms*, *coords*, *index*, *parent_correction=True*)

property charge

embeddings = {'': '', 'electronic': 'Electronic embedding', 'electronic_rc': 'Electronic embedding with redistributed charges', 'electronic_rcd': 'Electronic embedding with redistributed charges and dipoles'}

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_layer_calc(*layer_ind*)

property model_iter

property mult

run_calculation(*atoms*, *coords*)

run_calculations(*atoms*, *coords*, *method*)

pysisyphus.calculators.ONIOMv2.**atom_inds_to_cart_inds**(*atom_inds*)

pysisyphus.calculators.ONIOMv2.**cap_fragment**(*atoms*, *coords*, *fragment*, *link_atom='H'*, *g=None*)

pysisyphus.calculators.ONIOMv2.**get_embedding_charges**(*embedding*, *layer*, *parent_layer*, *coords3d*)

pysisyphus.calculators.ONIOMv2.**get_g_value**(*atom*, *parent_atom*, *link_atom*)

pysisyphus.calculators.ORCA module

class pysisyphus.calculators.ORCA.**ORCA**(*keywords*, *blocks=''*, *gbw=None*, *do_stable=False*, *numfreq=False*, *json_dump=True*, ***kwargs*)

Bases: *OverlapCalculator*

__init__(*keywords*, *blocks=''*, *gbw=None*, *do_stable=False*, *numfreq=False*, *json_dump=True*, ***kwargs*)

ORCA calculator.

Wrapper for creating ORCA input files for energy, gradient and Hessian calculations. The PAL and memory inputs must not be given in the keywords and/or blocks, as they are handled by the 'pal' and 'memory' arguments.

Parameters

- **keywords** (*str*) -- Keyword line, as normally given in ORCA, excluding the leading "!".
- **blocks** (*str, optional*) -- ORCA block input(s), e.g. for TD-DFT calculations (%tddft ... end). As the blocks start with a leading "%", wrapping the input in quotes ("") is required, otherwise the parsing will fail.
- **gbw** (*str, optional*) -- Path to an input gbw file, which will be used as initial guess for the first calculation. Will be overriden later, with the path to the gbw file of a previous calculation.
- **do_stable** (*bool, optional*) -- Run stability analysis until a stable wavefunction is obtained, before every calculation.
- **numfreq** (*bool, optional*) -- Use numerical frequencies instead of analytical ones.
- **json_dump** (*bool, optional*) -- Whether to dump the wavefunction to JSON via orca_2json. The JSON can become very large in calculations comprising many basis functions.

check_termination(**args*, ***kwargs*)

clean_tmp(*path*)

conf_key = 'orca'

get_block_str()

get_chkfiles()

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_moinp_str(*gbw*)

get_stable_wavefunction(*atoms*, *coords*)

parse_all_energies(*text=None*, *triplets=None*)

static parse_atoms_coords(*inp*, **args*, ***kwargs*)

static parse_cis(*cis*)

Simple wrapper of external function.

Currently, only returns X and Y.

parse_energy(*path*)

parse_engrad(*path*)

static parse_engrad_info(*inp*, **args*, ***kwargs*)

static parse_gbw(*gbw_fn*)

static parse_hess_file(*inp*, **args*, ***kwargs*)

parse_hessian(*path*)

parse_mo_numbers(*out_fn*)

parse_stable(*path*)

prepare_input(*atoms*, *coords*, *calc_type*, *point_charges=None*, *do_stable=False*)

Meant to be extended.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

reattach(*last_calc_cycle*)

run_after(*path*)

Meant to be extended.

This method is called after a calculation was done, but before entering `self.keep()` and `self.clean()`. Can be used to call tools like formchk or ricctools.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

Basically some kind of dummy method that can be called to execute ORCA with the stored cmd of this calculator.

set_chkfiles(*chkfiles*)

set_mo_coeffs(*mo_coeffs=None*, *gbw=None*)

static set_mo_coeffs_in_gbw(*in_gbw_fn*, *out_gbw_fn*, *mo_coeffs*)
See self.parse_gbw.

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

pysisyphus.calculators.ORCA.**get_exc_ens_fosc**(*wf_fn*, *cis_fn*, *log_fn*)

pysisyphus.calculators.ORCA.**get_name**(*text*)
Return string that comes before first character & offset.

pysisyphus.calculators.ORCA.**make_sym_mat**(*table_block*)

pysisyphus.calculators.ORCA.**parse_orca_cis**(*cis_fn*)

Read binary CI vector file from ORCA.
Loosly based on TheoDORE 1.7.1, Authors: S. Mai, F. Plasser https://sourceforge.net/p/theodore-qc

pysisyphus.calculators.ORCA.**parse_orca_gbw**(*gbw_fn*)
Adapted from https://orcaforum.kofo.mpg.de/viewtopic.php?f=8&t=3299

The first 5 long int values represent pointers into the file:

Pointer @+0: Internal ORCA data structures Pointer @+8: Geometry Pointer @+16: BasisSet Pointer @+24: Orbitals Pointer @+32: ECP data

pysisyphus.calculators.ORCA.**save_orca_pc_file**(*point_charges*, *pc_fn*, *hardness=None*)

pysisyphus.calculators.ORCA5 module

class pysisyphus.calculators.ORCA5.**ORCA5**(*keywords*, *blocks=''*, *gbw=None*, *do_stable=False*, *numfreq=False*, *json_dump=True*, ***kwargs*)

Bases: *ORCA*

conf_key = 'orca5'

pysisyphus.calculators.OpenMM module

class pysisyphus.calculators.OpenMM.**OpenMM**(*topology*, *params*, ***kwargs*)

Bases: *Calculator*

get_charges()

get_dipole_moment(*coords3d*, *reference=None*, *masses=None*)

get_energy(*atoms*, *coords*, ***prepare_kwargs*)
Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)
Meant to be extended.

report_charges()
Total charge of the system.

Adapted from modeller.py _addIons() in OpenMM.

pysisyphus.calculators.OpenMolcas module

class pysisyphus.calculators.OpenMolcas.**OpenMolcas**(*basis*, *inporb*, *rasscf=None*, *gateway=None*, *mcpdft=None*, *track=True*, ***kwargs*)

Bases: *Calculator*

build_gateway_str()

build_mcpdft_str()

build_rasscf_str()

build_rassi_str()

build_str_from_dict(*dct*)

conf_key = 'openmolcas'

get_energy(*atoms*, *coords*)
Meant to be extended.

get_forces(*atoms*, *coords*)
Meant to be extended.

get_pal_env()

get_root()

parse_energies(*text*)

parse_energy(*path*)

parse_gradient(*path*)

parse_rassi_track(*path*)

prepare_coords(*atoms*, *coords*)
Get 3d coords in Angstrom.

Reshape internal 1d coords to 3d and convert to Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns
coords -- 3D-array holding coordinates in Angstrom.

Return type
np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*)
Meant to be extended.

reattach(*last_calc_cycle*)

run_calculation(*atoms*, *coords*, *calc_type='energy'*)

pysisyphus.calculators.OverlapCalculator module

class pysisyphus.calculators.OverlapCalculator.**NTOs**(*ntos*, *lambdas*)

Bases: tuple

lambdas

Alias for field number 1

ntos

Alias for field number 0

class pysisyphus.calculators.OverlapCalculator.**OverlapCalculator**(**args*, *track=False*, *ovlp_type='tden'*, *double_mol=False*, *ovlp_with='previous'*, *adapt_args=(0.5, 0.3, 0.6)*, *cdds=None*, *orient=''*, *dump_fn='overlap_data.h5'*, *h5_dump=False*, *conf_thresh=0.001*, *mos_ref='cur'*, *mos_renorm=True*, ***kwargs*)

Bases: *Calculator*

H5_MAP = {'Ca': 'Ca_list', 'Cb': 'Cb_list', 'Xa': 'Xa_list', 'Xb': 'Xb_list', 'Ya': 'Ya_list', 'Yb': 'Yb_list', 'all_energies': 'all_energies_list', 'coords': 'coords_list', 'ref_roots': 'reference_roots', 'roots': 'roots_list'}

OVLP_TYPE_VERBOSE = {'nto': 'natural transition orbital overlap', 'nto_org': 'original natural transition orbital overlap', 'tden': 'transition density matrix overlap', 'top': 'transition orbital pair overlap', 'wf': 'wavefunction overlap'}

VALID_CDDS = (None, 'calc', 'render')

VALID_KEYS = ['wf', 'tden', 'nto', 'nto_org', 'top']

VALID_XY = ('X', 'X+Y', 'X-Y')

calc_cdd_cube(*root*, *cycle=-1*)

conf_thresh

self.dyn_roots = int(dyn_roots) if self.dyn_roots != 0:

self.dyn_roots = 0 self.log("dyn_roots = 0 is hardcoded right now")

property data_model

dump_overlap_data()

static from_overlap_data(*h5_fn*, *set_wfow=False*)

get_ci_coeffs_for(*ind*)

get_h5_group()

get_indices(*indices=None*)

A new root is determined by selecting the overlap matrix row corresponding to the reference root and checking for the root with the highest overlap (at the current geometry).

The overlap matrix is usually formed by a double loop like:

overlap_matrix = np.empty((ref_states, cur_states)) for i, ref_state in enumerate(ref_states):

for j, cur_state in enumerate(cur_states):
overlap_matrix[i, j] = make_overlap(ref_state, cur_state)

So the reference states run along the rows. Thats why the ref_state index comes first in the 'indices' tuple.

static get_mo_norms(*C*, *S_AO*)

get_orbital_matrices(*indices=None*, *S_AO=None*)

Return MO coefficents and AO overlaps for the given indices.

If not provided, a AO overlap matrix is constructed from one of the MO coefficient matrices (controlled by self.mos_ref). Also, if requested one of the two MO coefficient matrices is re-normalized.

get_ref_mos(*C_ref*, *C_cur*)

static get_sao_from_mo_coeffs(*C*)

Recover AO overlaps from given MO coefficients.

For MOs in the columns of mo_coeffs:

S_AO = C^{1}^T C^{1} S_AO C = C^{1}^T (S_AO C)^T = C^{1} C^T S_AO^T = C^{1} C^T S_AO C = I

get_tden_overlaps(*indices=None*, *S_AO=None*)

get_top_differences(*indices=None*, *S_AO=None*)

Transition orbital projection.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

render_cdd_cube()

static renorm_mos(*C*, *S_AO*)

store_overlap_data(*atoms*, *coords*, *path=None*, *overlap_data=None*)

property stored_calculations

track_root(*ovlp_type=None*)

Check if a root flip occured occured compared to the previous cycle by calculating the overlap matrix wrt. a reference cycle.

pysisyphus.calculators.OverlapCalculator.**get_data_model**(*exc_state_num*, *occ_a*, *virt_a*, *occ_b*, *virt_b*, *ovlp_type*, *atoms*, *max_cycles*)

pysisyphus.calculators.Psi4 module

class pysisyphus.calculators.Psi4.**Psi4**(*method*, *basis*, *to_set=None*, *to_import=None*, *pcm='iefpcm'*, *solvent=None*, *write_fchk=False*, ***kwargs*)

Bases: *Calculator*

conf_key = 'psi4'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_fchk_str()

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

parse_energy(*path*)

parse_grad(*path*)

parse_hessian(*path*)

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

pysisyphus.calculators.PyPsi4 module

class pysisyphus.calculators.PyPsi4.**PyPsi4**(*method*, *basis*, ***kwargs*)

Bases: *Calculator*

get_forces(*atoms*, *coords*)

Meant to be extended.

pysisyphus.calculators.PySCF module

pysisyphus.calculators.PyXTB module

class pysisyphus.calculators.PyXTB.**PyXTB**(**args*, *gfn=2*, *acc=None*, *verbosity=0*, *keep_calculator=False*, ***kwargs*)

Bases: *Calculator*

get_calculator(*atoms*, *coords*)

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

pysisyphus.calculators.QCEngine module

pysisyphus.calculators.Rastrigin module

class pysisyphus.calculators.Rastrigin.**Rastrigin**

Bases: *AnaPotBase*

http://www.sfu.ca/~ssurjano/rastr.html

pysisyphus.calculators.Remote module

class pysisyphus.calculators.Remote.**Remote**(*remote_calc*, *host*, *prefix=''*, ***kwargs*)

Bases: *Calculator*

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

parse_results()

run_calculation(*atoms*, *coords*, *run_func='get_energy'*)

pysisyphus.calculators.Rosenbrock module

class pysisyphus.calculators.Rosenbrock.**Rosenbrock**

Bases: *AnaPotBase*

pysisyphus.calculators.SocketCalc module

class pysisyphus.calculators.SocketCalc.**SocketCalc**(**args*, *host='localhost'*, *port=8080*, ***kwargs*)

Bases: *Calculator*

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

listen_for(*atoms*, *coords*, *request*)

valid_requests = ('energy', 'forces', 'hessian')

pysisyphus.calculators.TIP3P module

class pysisyphus.calculators.TIP3P.**TIP3P**(*rc=9.44863062728914*)

Bases: *Calculator*

Transferable Intermolecular Potential 3 Point

aHOH = 104.52

calculate(*coords3d*)

charges

coulomb_energy = (multiple of elem. charge * multiple of elem. charge)
/ (distance in bohr) * 1 / (4 * pi * vacuum permittivity)

coulomb_prefactor converts everything to atomic units and it is ... drum roll ... 1. from scipy.constants import value as pcval self.coulomb_prefactor = (1 / (4 * np.pi) * pcval("elementary charge")**2

/ pcval("Hartree energy") / pcval("Bohr radius") / pcval("vacuum electric permittivity")

)

coulomb(*coords3d*)

epsilon = 0.0002423919586315716

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

qH = 0.417

qO = -0.834

rOH = 1.8088458464917874

sigma = 5.953790025507198

pysisyphus.calculators.TransTorque module

class pysisyphus.calculators.TransTorque.**TransTorque**(*frags*, *iter_frags*, *a_mats*, *b_mats*, *weight_func=None*, *skip=True*, *kappa=1.0*, *b_coords3d=None*, *do_trans=True*)

Bases: object

__init__(*frags*, *iter_frags*, *a_mats*, *b_mats*, *weight_func=None*, *skip=True*, *kappa=1.0*, *b_coords3d=None*, *do_trans=True*)

Translational and torque forces. See A.4. [1], Eqs. (A3) - (A5).

get_forces(*atoms*, *coords*, *kappa=None*)

get_forces_naive(*atoms*, *coords*, *kappa=None*)

set_N_invs()

pysisyphus.calculators.TransTorque.**get_trans_torque_forces**(*mfrag*, *a_coords3d*, *b_coords3d*, *a_mats*, *b_mats*, *m*, *frags*, *N_inv*, *weight_func=None*, *skip=True*, *kappa=1*, *do_trans=True*)

pysisyphus.calculators.Turbomole module

class pysisyphus.calculators.Turbomole.**Turbomole**(*control_path=None*, *simple_input=None*, *root=None*, *double_mol_path=None*, *cosmo_kwargs=None*, ***kwargs*)

Bases: *OverlapCalculator*

append_control(*to_append*, *log_msg=''*, ***kwargs*)

conf_key = 'turbomole'

get_chkfiles()

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, *cmd=None*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_pal_env()

get_ricc2_root(*text*)

static make_molden(*path*)

parse_all_energies()

parse_cc2_vectors(*ccre*)

parse_ci_coeffs()

parse_double_mol(*path*)

Parse a double molecule overlap matrix from Turbomole output to be used with WFOWrapper.

parse_energy(*path*)

parse_force(*path*)

parse_gs_energy()

Several places are possible: $subenergy from control file total energy from turbomole.out Final MP2 energy from turbomole.out with ADC(2) Final CC2 energy from turbomole.out with CC(2)

parse_hessian(*path*, *fn=None*)

parse_mos()

parse_td_vectors(*text*)

For TDA calculations only the X vector is present in the ciss_a/etc. file. In TDDFT calculations there are twise as much items compared with TDA. The first half corresponds to (X+Y) and the second half to (X-Y). X can be calculated as X = ((X+Y)+(X-Y))/2. Y is then given as Y = (X+Y)-X. The normalization can then by checked as

np.concatenate((X, Y)).dot(np.concatenate((X, -Y)))

and should be 1.

static parse_tddft_tden(*inp*, **args*, ***kwargs*)

prepare_input(*atoms*, *coords*, *calc_type*, *point_charges=None*)

To rectify this we have to construct the basecmd dynamically and construct it ad hoc. We could set a RI flag in the beginning and select the correct scf binary here from it. Then we select the following binary on demand, e.g. aoforce or rdgrad or egrad etc.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

prepare_point_charges(*point_charges*)

$point_charges <x> <y> <z> <q>

prepare_td(*text*)

run_after(*path*)

Meant to be extended.

This method is called after a calculation was done, but before entering `self.keep()` and `self.clean()`. Can be used to call tools like formchk or ricctools.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

run_double_mol_calculation(*atoms*, *coords1*, *coords2*)

set_chkfiles(*chkfiles*)

set_occ_and_mo_nums(*text*)

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

sub_control(*pattern*, *repl*, *log_msg=''*, ***kwargs*)

pysisyphus.calculators.Turbomole.**control_from_simple_input**(*simple_inp*, *charge*, *mult*, *cosmo_kwargs=None*)

Create control file from 'simple input'.

See examples/opt/26_turbomole_simple_input/ for an example.

pysisyphus.calculators.Turbomole.**get_cosmo_data_groups**(*atoms*, *epsilon*, *rsolv=None*, *dcosmo_rs=None*)

pysisyphus.calculators.Turbomole.**index_strs_for_atoms**(*atoms*)

pysisyphus.calculators.Turbomole.**render_data_groups**(*raw_data_groups*)

pysisyphus.calculators.WFOWrapper module

class pysisyphus.calculators.WFOWrapper.**WFOWrapper**(*occ_mo_num*, *virt_mo_num*, *conf_thresh=0.001*, *calc_number=0*, *out_dir='./'*, *wfow_mem=8000*, *ncore=0*, *debug=False*)

Bases: object

ci_coeffs_above_thresh(*ci_coeffs*, *thresh=None*)

property conf_thresh

fake_turbo_mos(*mo_coeffs*)

Create a mos file suitable for TURBOMOLE input. All MO eigenvalues are set to 0.0. There is also a little deviation in the formatting (see turbo_fmt()) but it works ...

generate_all_dets(*occ_set1*, *virt_set1*, *occ_set2*, *virt_set2*)

Generate all possible single excitation determinant strings from union(occ_mos) to union(virt_mos).

get_from_to_sets(*ci_coeffs*)

get_gs_line(*ci_coeffs_with_gs*)

log(*message*)

logger = <Logger wfoverlap (DEBUG)>

make_det_string(*inds*)

Return spin adapted strings.

make_dets_header(*cic*, *dets_list*)

make_full_dets_list(*all_inds*, *det_strings*, *ci_coeffs*)

matrix_types = {'ortho': 'Orthonormalized overlap matrix', 'ovlp': 'Overlap matrix', 'renorm': 'Renormalized overlap matrix'}

parse_wfoverlap_out(*text*, *type_='ortho'*)

Returns overlap matrix.

set_from_nested_list(*nested*)

wf_overlap(*cycle1*, *cycle2*, *ao_ovlp=None*)

pysisyphus.calculators.WFOWrapper2 module

class pysisyphus.calculators.WFOWrapper2.**WFOWrapper2**(*overlap_data*, *calc_number=0*, *conf_thresh=0.0001*, *out_dir='./'*)

Bases: object

all_overlaps()

ci_coeffs_above_thresh(*ci_coeffs*, *thresh=1e-05*)

static fake_turbo_mos(*mo_coeffs*)

Create a mos file suitable for TURBOMOLE input. All MO eigenvalues are set to 0.0. There is also a little deviation in the formatting (see turbo_fmt()) but it works ...

generate_all_dets(*occ_set1*, *virt_set1*, *occ_set2*, *virt_set2*)

Generate all possible single excitation determinant strings from union(occ_mos) to union(virt_mos).

get_iteration(*ind*)

property last_two_coords

log(*message*)

logger = <Logger wfoverlap (DEBUG)>

make_det_string(*inds*)

Return spin adapted strings.

make_dets_header(*cic*, *dets_list*)

make_full_dets_list(*all_inds*, *det_strings*, *ci_coeffs*)

matrix_types = {'ortho': 'Orthonormalized overlap matrix', 'ovlp': 'Overlap matrix', 'renorm': 'Renormalized overlap matrix'}

parse_wfoverlap_out(*text*, *type_='ortho'*)

Returns overlap matrix.

set_data()

set_from_nested_list(*nested*)

wf_overlap(*ind1=-2*, *ind2=-1*, *ao_ovlp=None*)

pysisyphus.calculators.XTB module

class pysisyphus.calculators.XTB.**OptResult**(*opt_geom*, *opt_log*)

Bases: tuple

opt_geom

Alias for field number 0

opt_log

Alias for field number 1

class pysisyphus.calculators.XTB.**XTB**(*gbsa=''*, *alpb=''*, *gfn=2*, *acc=1.0*, *iterations=250*, *etemp=None*, *retry_etemp=None*, *restart=False*, *topo=None*, *topo_update=None*, *quiet=False*, ***kwargs*)

Bases: *Calculator*

__init__(*gbsa=''*, *alpb=''*, *gfn=2*, *acc=1.0*, *iterations=250*, *etemp=None*, *retry_etemp=None*, *restart=False*, *topo=None*, *topo_update=None*, *quiet=False*, ***kwargs*)

XTB calculator.

Wrapper for running energy, gradient and Hessian calculations by XTB.

Parameters

- **gbsa** (*str, optional*) -- Solvent for GBSA calculation, by default no solvent model is used.
- **alpb** (*str, optional*) -- Solvent for ALPB calculation, by default no solvent model is used.

- **gfn** (*int or str, must be (0, 1, 2, or "ff")*) -- Hamiltonian for the XTB calculation (GFN0, GFN1, GFN2, or GFNFF).
- **acc** (*float, optional*) -- Accuracy control of the calculation, the lower the tighter several numerical thresholds are chosen.
- **iterations** (*int, optional*) -- The number of iterations in SCC calculation.
- **topo** (*str, optional*) -- Path the a GFNFF-topolgy file. As setting up the topology may take some time for sizable systems, it may be desired to reuse the file.
- **topo_update** (*int*) -- Integer controlling the update interval of the GFNFF topology update. If supplied, the topolgy will be recreated every N-th calculation.
- **mem** (*int*) -- Mememory per core in MB.
- **quiet** (*bool, optional*) -- Suppress creation of log files.

static check_termination(*inp*, **args*, ***kwargs*)

conf_key = 'xtb'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_mdrestart_str(*coords*, *velocities*)

coords and velocities have to given in au!

get_pal_env()

get_retry_args()

parse_charges(*fn=None*)

parse_charges_from_json(*fn=None*)

parse_energy(*path*)

parse_gradient(*path*)

parse_hessian(*path*)

parse_md(*path*)

parse_opt(*path*, *keep_log=False*)

parse_topo(*path*)

prepare_add_args(*xcontrol=None*)

prepare_coords(*atoms*, *coords*)

Get 3d coords in Angstrom.

Reshape internal 1d coords to 3d and convert to Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns
coords -- 3D-array holding coordinates in Angstrom.

Return type
np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*, *point_charges=None*)

Meant to be extended.

reattach(*last_calc_cycle*)

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

run_md(*atoms*, *coords*, *t*, *dt*, *velocities=None*, *dump=1*)

Expecting t and dt in fs, even though xtb wants t in ps!

run_opt(*atoms*, *coords*, *keep=True*, *keep_log=False*)

run_topo(*atoms*, *coords*)

write_mdrestart(*path*, *mdrestart_str*)

pysisyphus.calculators.parser module

pysisyphus.calculators.parser.**make_float_class**(***kwargs*)

pysisyphus.calculators.parser.**parse_turbo_ccre0_ascii**(*text*)

pysisyphus.calculators.parser.**parse_turbo_exstates**(*text*)

Parse excitation energies (first blocks) from an exstates file.

pysisyphus.calculators.parser.**parse_turbo_exstates_re**(*text*)

pysisyphus.calculators.parser.**parse_turbo_gradient**(*path*)

pysisyphus.calculators.parser.**parse_turbo_mos**(*text*)

pysisyphus.calculators.parser.**to_float**(*s*, *loc*, *toks*)

Module contents

class pysisyphus.calculators.**AFIR**(*calculator*, *fragment_indices*, *gamma*, *rho=1*, *p=6*, *ignore_hydrogen=False*, *zero_hydrogen=True*, *complete_fragments=True*, *dump=True*, *h5_fn='afir.h5'*, *h5_group_name='afir'*, ***kwargs*)

Bases: *Calculator*

__init__(*calculator*, *fragment_indices*, *gamma*, *rho=1*, *p=6*, *ignore_hydrogen=False*, *zero_hydrogen=True*, *complete_fragments=True*, *dump=True*, *h5_fn='afir.h5'*, *h5_group_name='afir'*, ***kwargs*)

Artifical Force Induced Reaction calculator.

Currently, there are no automated drivers to run large-scale AFIR calculations with many different initial orientations and/or increasing collision energy parameter . Nontheless, selected AFIR calculations can be carried out by hand. After convergence, artificial potential & forces, as well as real energies and forces can be plotted with 'pysisplot --afir'. The highest energy point along the AFIR path can then be selected for a subsequent TS-optimization, e.g. via 'pysistrj --get [index] optimzation.trj'.

Future versions of pysisyphus may provide drivers for more automatted AFIR calculations.

Parameters

- **calculator** (*Calculator*) -- Actual QC calculator that provides energies and its derivatives, that are modified by the AFIR calculator, e.g., ORCA or Psi4.
- **fragment_indices** (List[List[int]]) -- List of lists of integers, specifying the separate fragments. If the indices in theses lists don't comprise all atoms in the molecule, the reamining indices will be added as a separate fragment. If a AFIR calculation is carried out with 2 fragments and 'complete_fragments' is True (see below) it is enough to specify only the indices of one fragment, e.g., for a system of 10 atoms 'fragment_indices=[[0,1,2,3]]' is enough. The second system will be set up automatically with indices [4,5,6,7,8,9].
- **gamma** (Union[_SupportsArray[dtype[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]]) -- Collision energy parameter in au. For 2 fragments it can be a single integer, while for > 2 fragments a list of gammas must be given, specifying the pair-wise collision energy parameters. For 3 fragments 3 gammas must be given [_01, _02, _12], for 4 fragments 6 gammas would be required [_01, _02, _03, _12, _13, _23] and so on.
- **rho** (Union[_SupportsArray[dtype[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]], default: 1) -- Direction of the artificial force, either 1 or -1. The same comments as for gamma apply. For 2 fragments a single integer is enough, for > 2 fragments a list of rhos must be given (see above). For rho=1 fragments are pushed together, for rho=-1 fragments are pulled apart.
- **p** (int, default: 6) -- Exponent p used in the calculation of the weight function . Defaults to 6 and probably does not have to be changed.
- **ignore_hydrogen** (bool, default: False) -- Whether hydrogens are ignored in the calculation of the artificial force. All weights between atom pairs containing hydrogen will be set to 0.
- **zero_hydrogen** (bool, default: True) -- Whether to use 0.0 as covalent radius for hydrogen in the weight function. Compared to 'ignore_hydrogen', which results in zero weights for all atom pairs involving hydrogen, 'zero_hydrogen' may be non-zero, depending on the covalent radius of the second atom in the pair.

- **complete_fragments** (bool, default: True) -- Whether an incomplete specification in 'fragment_indices' is automatically completed.
- **dump** (bool, default: True) -- Whether an HDF5 file is created.
- **h5_fn** (str, default: 'afir.h5') -- Filename of the HDF5 file used for dumping.
- **h5_group_name** (str, default: 'afir') -- HDF5 group name used for dumping.
- ****kwargs** -- Keyword arguments passed to the Calculator baseclass.

afir_fd_hessian_wrapper(*coords3d*, *afir_grad_func*)

property charge

dump_h5(*atoms*, *coords*, *results*)

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

init_h5_group(*atoms*, *max_cycles=None*)

log_fragments()

property mult

set_atoms_and_funcs(*atoms*, *coords*)

Initially atoms was also an argument to the constructor of AFIR. I removed it so creation becomes easier. The first time a calculation is requested with a proper atom set everything is set up (cov. radii, afir function and corresponding gradient). Afterwards there is only a check if atoms != None and it is expected that all functions are properly set.

fragment_indices can also be incomplete w.r.t. to the number of atoms. If the sum of the specified fragment atoms is less than the number of atoms present then all remaining unspecified atoms will be gathered in one fragment.

write_fragment_geoms(*atoms*, *coords*)

class pysisyphus.calculators.**AtomAtomTransTorque**(*geom*, *frags*, *A_mats*, *kappa=2.0*)

Bases: object

__init__(*geom*, *frags*, *A_mats*, *kappa=2.0*)

Atom-atom translational and torque forces.

See A.5. [1], Eq. (A6).

get_forces(*atoms*, *coords*)

class pysisyphus.calculators.**CFOUR**(*cfour_input*, *wavefunction_dump=True*, *initden_file=None*, ***kwargs*)

Bases: *Calculator*

__init__(*cfour_input*, *wavefunction_dump=True*, *initden_file=None*, ***kwargs*)

CFOUR calculator.

Wrapper handling CFOUR ground state energy and gradient calculations.

Parameters

- **cfour_input** (*dict*) -- CFOUR keywords and values. Note: "on" must be encapsulated in quotes to avoid being translated to True by YAML.
- **wavefunction_dump** (*bool, optional*) -- Whether or not to keep ground state SCF orbitals for each geometry step.
- **initden_file** (*str, optional*) -- Path to an input initden file for use as a guess SCF density.

conf_key = 'cfour'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

keep(*path*)

Backup calculation results.

Parameters
path (*Path*) -- Temporary directory of the calculation.

Returns
kept_fns -- Dictonary holding the filenames that were backed up. The keys correspond to the type of file.

Return type
dict

parse_energy(*path*)

parse_gradient(*path*)

prepare(*inp*)

Prepare a temporary directory and write input.

Similar to prepare_path, but the input is also written into the prepared directory.

Paramters

inp
[str] Input to be written into the file `self.inp_fn` in the prepared directory.

returns

path: Path
Prepared directory.

prepare_coords(*atoms*, *coords*)

Get 3d coords in Bohr

Reshape internal 1d coords to 3d.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns
coords -- 3D-array holding coordinates in Bohr.

Return type
np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

run_calculation(*atoms*, *coords*, *calc_type*, ***prepare_kwargs*)

class pysisyphus.calculators.**Composite**(*final*, *keys_calcs=None*, *calcs=None*, *remove_translation=False*, ***kwargs*)

Bases: *Calculator*

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_final_energy(*energies*)

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

class pysisyphus.calculators.**ConicalIntersection**(*calculator1*, *calculator2*, ***kwargs*)

Bases: *Calculator*

Calculator for conical intersection optimization.

Based on [1].

get_ci_quantities(*atoms*, *coords*, ***prepare_kwargs*)

Relavent quantities including branching plane and projector P.

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Energy of calculator 1.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Projected gradient for CI optimization.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Projected Hessian.

class pysisyphus.calculators.**DFTBp**(*parameter*, **args*, *slakos=None*, *root=None*, ***kwargs*)

Bases: *OverlapCalculator*

conf_key = 'dftbp'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

static get_excited_state_str(*root*, *forces=False*)

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

static get_gen_str(*atoms*, *coords*)

hubbard_derivs = {'3ob': {'Br': -0.0573, 'C': -0.1492, 'Ca': -0.034, 'Cl': -0.0697, 'F': -0.1623, 'H': -0.1857, 'I': -0.0433, 'K': -0.0339, 'Mg': -0.02, 'N': -0.1535, 'Na': -0.0454, 'O': -0.1575, 'P': -0.14, 'S': -0.11, 'Zn': -0.03}}

max_ang_moms = {'3ob': {'Br': 'd', 'C': 'p', 'Ca': 'p', 'Cl': 'd', 'F': 'p', 'H': 's', 'I': 'd', 'K': 'p', 'Mg': 'p', 'N': 'p', 'Na': 'p', 'O': 'p', 'P': 'd', 'S': 'd', 'Zn': 'd'}, 'mio-ext': {'C': 'p', 'H': 's', 'N': 'p', 'O': 'p'}}

parse_all_energies(*out_fn=None*, *exc_dat=None*)

parse_energy(*path*)

static parse_exc_dat(*text*)

parse_forces(*path*)

parse_total_energy(*text*)

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

class pysisyphus.calculators.**DFTD4**(*method=None*, *damp_params=None*, *model_params=None*, ***kwargs*)

Bases: *Calculator*

get_dispersion(*atoms*, *coords*, *grad*)

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_model(*atoms*, *coords*)

class pysisyphus.calculators.**Dimer**(*calculator*, **args*, *N_raw=None*, *length=0.0189*, *rotation_max_cycles=15*, *rotation_method='fourier'*, *rotation_thresh=0.0001*, *rotation_tol=1*, *rotation_max_element=0.001*, *rotation_interpolate=True*, *rotation_disable=False*, *rotation_disable_pos_curv=True*, *rotation_remove_trans=True*, *trans_force_f_perp=True*, *bonds=None*, *N_hessian=None*, *bias_rotation=False*, *bias_translation=False*, *bias_gaussian_dot=0.1*, *seed=None*, *write_orientations=True*, *forward_hessian=True*, ***kwargs*)

Bases: *Calculator*

property C

Shortcut for the curvature.

property N

add_gaussian(*atoms*, *center*, *N*, *height=0.1*, *std=0.0529*, *max_cycles=50*, *dot_ref=None*)

property can_bias_f0

property can_bias_f1

property coords0

property coords1

curvature(*f1*, *f2*, *N*)

Curvature of the mode represented by the dimer.

direct_rotation(*optimizer*, *prev_step*)

do_dimer_rotations(*rotation_thresh=None*)

property energy0

property f0

property f1

property f1_bias

property f2

Never calculated explicitly, but estimated from f0 and f1.

fourier_rotation(*optimizer*, *prev_step*)

get_N_raw_from_hessian(*h5_fn*, *root=0*)

get_forces(*atoms*, *coords*)

Meant to be extended.

get_gaussian_energies(*coords*, *sum_=True*)

get_gaussian_forces(*coords*, *sum_=True*)

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

remove_translation(*displacement*)

property rot_force

rotate_coords1(*rad*, *theta*)

Rotate dimer and produce new coords1.

set_N_raw(*coords*)

property should_bias_f0

May lead to calculation of f0 and/or f1 if present!

property should_bias_f1

May lead to calculation of f0 and/or f1 if present!

update_orientation(*coords*)

class pysisyphus.calculators.**Dummy**(*calc_number=0*, *charge=0*, *mult=1*, *base_name='calculator'*, *pal=1*, *mem=1000*, *keep_kind='all'*, *check_mem=True*, *retry_calc=0*, *last_calc_cycle=None*, *clean_after=True*, *out_dir='qm_calcs'*, *force_num_hess=False*, *num_hess_kwargs=None*)

Bases: *Calculator*

get_energy(**args*, ***kwargs*)

Meant to be extended.

get_forces(**args*, ***kwargs*)

Meant to be extended.

get_hessian(**args*, ***kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

raise_exception()

run_calculation(**args*, ***kwargs*)

class pysisyphus.calculators.**EGO**(*calculator*, *ref_geom*, *max_force=0.175*, ***kwargs*)

Bases: *Calculator*

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_mods(*atoms*, *coords*)

property ref_hessian

property s

class pysisyphus.calculators.**EnergyMin**(*calculator1*, *calculator2*, *mix=False*, *alpha=0.02*, *sigma=3.5*, *min_energy_diff=0.0*, *check_after=0*, ***kwargs*)

Bases: *Calculator*

__init__(*calculator1*, *calculator2*, *mix=False*, *alpha=0.02*, *sigma=3.5*, *min_energy_diff=0.0*, *check_after=0*, ***kwargs*)

Use energy and derivatives of the calculator with lower energy.

This calculators carries out two calculations with different settings and returns the results of the lower energy one. This can be used to consider flips between a singlet and a triplet PES etc.

Parameters

- **calculator1** (*Calculator*) -- Wrapped QC calculator that provides energies and its derivatives.
- **calculator2** (*Calculator*) -- Wrapped QC calculator that provides energies and its derivatives.
- **mix** (bool, default: False) -- Enable mixing of both forces, according to the approach outlined in [2]. Can be used to optimize guesses for MECPs. Pass
- **alpha** (float, default: 0.02) -- Smoothing parameter in Hartree. See [2] for a discussion.
- **sigma** (float, default: 3.5) -- Unitless gap size parameter. The final gap becomes smaller for bigga sigmas. Has to be adapted for each case. See [2] for a discussion (p. 407 right column and p. 408 left column.)
- **min_energy_diff** (float, default: 0.0) -- Energy difference in Hartree. When set to a value != 0 and the energy difference between both calculators drops below this value, execution of both calculations is diabled for 'check_after' cycles. In these cycles the calculator choice remains fixed. After 'check_after' cycles, both energies will be calculated and it is checked, if the previous calculator choice remains valid. In conjunction with 'check_after' both arguments can be used to save computational ressources.
- **check_after** (int, default: 0) -- Amount of cycles in which the calculator choice remains fixed.
- ****kwargs** -- Keyword arguments passed to the Calculator baseclass.

do_calculations(*name*, *atoms*, *coords*, ***prepare_kwargs*)

get_chkfiles()

Return type
dict

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

set_chkfiles(*chkfiles*)

class pysisyphus.calculators.**ExternalPotential**(*calculator=None*, *potentials=None*, *geom=None*, ***kwargs*)

Bases: *Calculator*

available_potentials = {'d3': <class 'pysisyphus.calculators.DFTD3.DFTD3'>, 'harmonic_sphere': <class 'pysisyphus.calculators.ExternalPotential.HarmonicSphere'>, 'logfermi': <class 'pysisyphus.calculators.ExternalPotential.LogFermi'>, 'restraint': <class 'pysisyphus.calculators.ExternalPotential.Restraint'>, 'rmsd': <class 'pysisyphus.calculators.ExternalPotential.RMSD'>}

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_potential_energy(*coords*)

get_potential_forces(*coords*)

class pysisyphus.calculators.**FakeASE**(*calc*)

Bases: object

get_atoms_coords(*atoms*)

get_forces(*atoms=None*)

get_potential_energy(*atoms=None*)

class pysisyphus.calculators.**Gaussian09**(**args*, ***kwargs*)

Bases: *Gaussian16*

conf_key = 'gaussian09'

class pysisyphus.calculators.**Gaussian16**(*route*, *gbs=''*, *gen=''*, *keep_chk=False*, *stable=''*, *fchk=None*, ***kwargs*)

Bases: *OverlapCalculator*

conf_key = 'gaussian16'

get_chkfiles()

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

make_exc_str()

make_fchk(*path*)

make_gbs_str()

parse_635r_dump(*dump_path*, *roots*, *nmos*)

parse_all_energies(*fchk=None*)

parse_charges(*path=None*)

parse_double_mol(*path*, *out_fn=None*)

static parse_fchk(*fchk_path*, *keys*)

parse_force(*path*)

parse_hessian(*path*)

parse_keyword(*text*)

parse_log(**args*, ***kwargs*)

parse_stable(*path*)

parse_tddft(*path*)

prepare_input(*atoms*, *coords*, *calc_type*, *did_stable=False*, *point_charges=None*)

Meant to be extended.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

reuse_data(*path*)

run_after(*path*)

Meant to be extended.

This method is called after a calculation was done, but before entering `self.keep()` and `self.clean()`. Can be used to call tools like formchk or ricctools.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

run_double_mol_calculation(*atoms*, *coords1*, *coords2*)

run_rwfdump(*path*, *rwf_index*, *chk_path=None*)

run_stable(*atoms*, *coords*, ***prepare_kwargs*)

set_chkfiles(*chkfiles*)

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

class pysisyphus.calculators.**HardSphere**(*geom*, *frags*, *kappa=1.0*, *permutations=False*, *frag_radii=None*, *radii_offset=0.9452*)

Bases: `object`

__init__(*geom*, *frags*, *kappa=1.0*, *permutations=False*, *frag_radii=None*, *radii_offset=0.9452*)

Intra-Image Inter-Molecular Hard-Sphere force.

See A.2. in [1], Eq. (A1).

get_forces(*atoms*, *coords*, *kappa=None*)

class pysisyphus.calculators.**IPIServer**(**args*, *address=None*, *host=None*, *port=None*, *unlink=True*, *hdrlen=12*, *max_retries=0*, *verbose=False*, ***kwargs*)

Bases: *Calculator*

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

listen_for(*atoms*, *coords*, *kind='forces'*)

listen_for_client_atom_num(*atom_num*)

listen_for_energy()

listen_for_forces(*atom_num*)

listen_for_hessian(*atom_num*)

listen_kinds = ('coords', 'energy', 'forces', 'hessian')

reset_client_connection()

retried_listen_for(*atoms*, *coords*)

unlink(*address*)

class pysisyphus.calculators.**LennardJones**(*sigma=1.8897261251*, *epsilon=1*, *rc=None*)

Bases: *Calculator*

calculate(*coords3d*)

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

class pysisyphus.calculators.**MOPAC**(*method='PM7'*, ***kwargs*)

Bases: *Calculator*

http://openmopac.net/manual/

CALC_TYPES = {'energy': '1SCF', 'gradient': '1SCF GRADIENTS', 'hessian': 'DFORCE FORCE LET'}

METHODS = ['am1', 'pm3', 'pm6', 'pm6-dh2', 'pm6-d3', 'pm6-dh+', 'pm6-dh2', 'pm6-dh2x', 'pm6-d3h4', 'pm6-d3h4x', 'pm7', 'pm7-ts']

MULT_STRS = {1: 'SINGLET', 2: 'DOUBLET', 3: 'TRIPLET', 4: 'QUARTET', 5: 'QUINTET', 6: 'SEXTET', 7: 'SEPTET', 8: 'OCTET'}

base_cmd

Do only SCF AUX: Creates a checkpoint file NOREO: Dont reorient geometry

Type
1SCF

conf_key = 'mopac'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

parse_energy(*path*)

static parse_energy_from_aux(*inp*, **args*, ***kwargs*)

parse_grad(*path*)

parse_hessian(*path*)

static parse_hessian_from_aux(*inp*, **args*, ***kwargs*)

prepare_coords(*atoms*, *coords*, *opt=False*)

Get 3d coords in Angstrom.

Reshape internal 1d coords to 3d and convert to Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns
coords -- 3D-array holding coordinates in Angstrom.

Return type
np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*, *opt=False*)

Meant to be extended.

read_aux(*path*)

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

class pysisyphus.calculators.MultiCalc(*calcs*, ***kwargs*)

Bases: *Calculator*

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

class pysisyphus.calculators.OBabel(*ff='gaff'*, *mol=None*, ***kwargs*)

Bases: *Calculator*

conv_dict = {'kcal/mol': 627.5094740630558, 'kj/mol': 2625.4996394798254}

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

setup(*atoms*, *coords*)

class pysisyphus.calculators.ONIOM(*calcs*, *models*, *geom*, *layers=None*, *embedding=''*, *real_key='real'*, *use_link_atoms=True*, **args*, ***kwargs*)

Bases: Calculator

__init__(*calcs*, *models*, *geom*, *layers=None*, *embedding=''*, *real_key='real'*, *use_link_atoms=True*, **args*, ***kwargs*)

layer: list of models

len(layer) == 1: normal ONIOM, len(layer) >= 1: multicenter ONIOM.

model:

(sub)set of all atoms that resides in a certain layer and has a certain calculator.

atom_inds_in_layer(*index*, *exclude_inner=False*)

Returns list of atom indices in layer at index.

Atoms that also appear in inner layer can be excluded on request.

Parameters

- **index** (*int*) -- pasd
- **exclude_inner** (*bool, default=False, optional*) -- Whether to exclude atom indices that also appear in inner layers.

Returns

atom_indices -- List containing the atom indices in the selected layer.

Return type

list

calc_layer(*atoms*, *coords*, *index*, *parent_correction=True*)

property charge

embeddings = {'': '', 'electronic': 'Electronic embedding', 'electronic_rc': 'Electronic embedding with redistributed charges', 'electronic_rcd': 'Electronic embedding with redistributed charges and dipoles'}

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

get_hessian(*atoms*, *coords*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_layer_calc(*layer_ind*)

property model_iter

property mult

run_calculation(*atoms*, *coords*)

run_calculations(*atoms*, *coords*, *method*)

class pysisyphus.calculators.ORCA(*keywords*, *blocks=''*, *gbw=None*, *do_stable=False*, *numfreq=False*, *json_dump=True*, ***kwargs*)

Bases: OverlapCalculator

__init__(*keywords*, *blocks=''*, *gbw=None*, *do_stable=False*, *numfreq=False*, *json_dump=True*, ***kwargs*)

ORCA calculator.

Wrapper for creating ORCA input files for energy, gradient and Hessian calculations. The PAL and memory inputs must not be given in the keywords and/or blocks, as they are handled by the 'pal' and 'memory' arguments.

Parameters

- **keywords** (*str*) -- Keyword line, as normally given in ORCA, excluding the leading "!".
- **blocks** (*str, optional*) -- ORCA block input(s), e.g. for TD-DFT calculations (%tddft ... end). As the blocks start with a leading "%", wrapping the input in quotes ("") is required, otherwise the parsing will fail.
- **gbw** (*str, optional*) -- Path to an input gbw file, which will be used as initial guess for the first calculation. Will be overriden later, with the path to the gbw file of a previous calculation.
- **do_stable** (*bool, optional*) -- Run stability analysis until a stable wavefunction is obtained, before every calculation.
- **numfreq** (*bool, optional*) -- Use numerical frequencies instead of analytical ones.
- **json_dump** (*bool, optional*) -- Whether to dump the wavefunction to JSON via orca_2json. The JSON can become very large in calculations comprising many basis functions.

check_termination(**args*, ***kwargs*)

clean_tmp(*path*)

conf_key = 'orca'

get_block_str()

get_chkfiles()

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_moinp_str(*gbw*)

get_stable_wavefunction(*atoms*, *coords*)

parse_all_energies(*text=None*, *triplets=None*)

static parse_atoms_coords(*inp*, **args*, ***kwargs*)

static parse_cis(*cis*)

Simple wrapper of external function.

Currently, only returns X and Y.

parse_energy(*path*)

parse_engrad(*path*)

static parse_engrad_info(*inp*, **args*, ***kwargs*)

static parse_gbw(*gbw_fn*)

static parse_hess_file(*inp*, **args*, ***kwargs*)

parse_hessian(*path*)

parse_mo_numbers(*out_fn*)

parse_stable(*path*)

prepare_input(*atoms*, *coords*, *calc_type*, *point_charges=None*, *do_stable=False*)

Meant to be extended.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

reattach(*last_calc_cycle*)

run_after(*path*)

Meant to be extended.

This method is called after a calculation was done, but before entering `self.keep()` and `self.clean()`. Can be used to call tools like formchk or ricctools.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

Basically some kind of dummy method that can be called to execute ORCA with the stored cmd of this calculator.

set_chkfiles(*chkfiles*)

set_mo_coeffs(*mo_coeffs=None*, *gbw=None*)

static set_mo_coeffs_in_gbw(*in_gbw_fn*, *out_gbw_fn*, *mo_coeffs*)

See self.parse_gbw.

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

class pysisyphus.calculators.**ORCA5**(*keywords*, *blocks=''*, *gbw=None*, *do_stable=False*, *numfreq=False*, *json_dump=True*, ***kwargs*)

Bases: *ORCA*

conf_key = 'orca5'

class pysisyphus.calculators.OpenMolcas(*basis*, *inporb*, *rasscf=None*, *gateway=None*, *mcpdft=None*, *track=True*, ***kwargs*)

Bases: Calculator

build_gateway_str()

build_mcpdft_str()

build_rasscf_str()

build_rassi_str()

build_str_from_dict(*dct*)

conf_key = 'openmolcas'

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

get_pal_env()

get_root()

parse_energies(*text*)

parse_energy(*path*)

parse_gradient(*path*)

parse_rassi_track(*path*)

prepare_coords(*atoms*, *coords*)

Get 3d coords in Angstrom.

Reshape internal 1d coords to 3d and convert to Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns

coords -- 3D-array holding coordinates in Angstrom.

Return type

np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

reattach(*last_calc_cycle*)

run_calculation(*atoms*, *coords*, *calc_type='energy'*)

class pysisyphus.calculators.**Psi4**(*method*, *basis*, *to_set=None*, *to_import=None*, *pcm='iefpcm'*, *solvent=None*, *write_fchk=False*, ***kwargs*)

Bases: *Calculator*

conf_key = 'psi4'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_fchk_str()

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

parse_energy(*path*)

parse_grad(*path*)

parse_hessian(*path*)

prepare_input(*atoms*, *coords*, *calc_type*)

Meant to be extended.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

class pysisyphus.calculators.**PyPsi4**(*method*, *basis*, ***kwargs*)

Bases: *Calculator*

get_forces(*atoms*, *coords*)

Meant to be extended.

class pysisyphus.calculators.**PyXTB**(**args*, *gfn=2*, *acc=None*, *verbosity=0*, *keep_calculator=False*, ***kwargs*)

Bases: *Calculator*

get_calculator(*atoms*, *coords*)

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

class pysisyphus.calculators.**Remote**(*remote_calc*, *host*, *prefix=''*, ***kwargs*)

Bases: *Calculator*

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

parse_results()

run_calculation(*atoms*, *coords*, *run_func='get_energy'*)

class pysisyphus.calculators.**TIP3P**(*rc=9.44863062728914*)

Bases: *Calculator*

Transferable Intermolecular Potential 3 Point

aHOH = 104.52

calculate(*coords3d*)

charges

coulomb_energy = (multiple of elem. charge * multiple of elem. charge)
/ (distance in bohr) * 1 / (4 * pi * vacuum permittivity)

coulomb_prefactor converts everything to atomic units and it is ... drum roll ... 1. from scipy.constants import value as pcval self.coulomb_prefactor = (1 / (4 * np.pi) * pcval("elementary charge")**2

/ pcval("Hartree energy") / pcval("Bohr radius") / pcval("vacuum electric permittivity")

)

coulomb(*coords3d*)

epsilon = 0.0002423919586315716

get_energy(*atoms*, *coords*)

Meant to be extended.

get_forces(*atoms*, *coords*)

Meant to be extended.

qH = 0.417

qO = -0.834

rOH = 1.8088458464917874

sigma = 5.953790025507198

class pysisyphus.calculators.**TransTorque**(*frags*, *iter_frags*, *a_mats*, *b_mats*, *weight_func=None*, *skip=True*, *kappa=1.0*, *b_coords3d=None*, *do_trans=True*)

Bases: object

__init__(*frags*, *iter_frags*, *a_mats*, *b_mats*, *weight_func=None*, *skip=True*, *kappa=1.0*, *b_coords3d=None*, *do_trans=True*)

Translational and torque forces. See A.4. [1], Eqs. (A3) - (A5).

get_forces(*atoms*, *coords*, *kappa=None*)

get_forces_naive(*atoms*, *coords*, *kappa=None*)

set_N_invs()

class pysisyphus.calculators.**Turbomole**(*control_path=None*, *simple_input=None*, *root=None*, *double_mol_path=None*, *cosmo_kwargs=None*, ***kwargs*)

Bases: *OverlapCalculator*

append_control(*to_append*, *log_msg=''*, ***kwargs*)

conf_key = 'turbomole'

get_chkfiles()

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, *cmd=None*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_pal_env()

get_ricc2_root(*text*)

static make_molden(*path*)

parse_all_energies()

parse_cc2_vectors(*ccre*)

parse_ci_coeffs()

parse_double_mol(*path*)

Parse a double molecule overlap matrix from Turbomole output to be used with WFOWrapper.

parse_energy(*path*)

parse_force(*path*)

parse_gs_energy()

Several places are possible: $subenergy from control file total energy from turbomole.out Final MP2 energy from turbomole.out with ADC(2) Final CC2 energy from turbomole.out with CC(2)

parse_hessian(*path*, *fn=None*)

parse_mos()

parse_td_vectors(*text*)

For TDA calculations only the X vector is present in the ciss_a/etc. file. In TDDFT calculations there are twise as much items compared with TDA. The first half corresponds to (X+Y) and the second half to (X-Y). X can be calculated as X = ((X+Y)+(X-Y))/2. Y is then given as Y = (X+Y)-X. The normalization can then by checked as

np.concatenate((X, Y)).dot(np.concatenate((X, -Y)))

and should be 1.

static parse_tddft_tden(*inp*, **args*, ***kwargs*)

prepare_input(*atoms*, *coords*, *calc_type*, *point_charges=None*)

To rectify this we have to construct the basecmd dynamically and construct it ad hoc. We could set a RI flag in the beginning and select the correct scf binary here from it. Then we select the following binary on demand, e.g. aoforce or rdgrad or egrad etc.

prepare_overlap_data(*path*)

This method has to implement the calculator specific parsing of MO-coefficients, CI-coefficients and energies. Should return a filename pointing to TURBOMOLE like mos, a MO coefficient array and a CI coefficient array.

prepare_point_charges(*point_charges*)

$point_charges <x> <y> <z> <q>

prepare_td(*text*)

run_after(*path*)

Meant to be extended.

This method is called after a calculation was done, but before entering `self.keep()` and `self.clean()`. Can be used to call tools like formchk or ricctools.

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

run_double_mol_calculation(*atoms*, *coords1*, *coords2*)

set_chkfiles(*chkfiles*)

set_occ_and_mo_nums(*text*)

store_and_track(*results*, *func*, *atoms*, *coords*, ***prepare_kwargs*)

sub_control(*pattern*, *repl*, *log_msg=''*, ***kwargs*)

class pysisyphus.calculators.**XTB**(*gbsa=''*, *alpb=''*, *gfn=2*, *acc=1.0*, *iterations=250*, *etemp=None*, *retry_etemp=None*, *restart=False*, *topo=None*, *topo_update=None*, *quiet=False*, ***kwargs*)

Bases: *Calculator*

__init__(*gbsa=''*, *alpb=''*, *gfn=2*, *acc=1.0*, *iterations=250*, *etemp=None*, *retry_etemp=None*, *restart=False*, *topo=None*, *topo_update=None*, *quiet=False*, ***kwargs*)

XTB calculator.

Wrapper for running energy, gradient and Hessian calculations by XTB.

Parameters

- **gbsa** (*str, optional*) -- Solvent for GBSA calculation, by default no solvent model is used.
- **alpb** (*str, optional*) -- Solvent for ALPB calculation, by default no solvent model is used.
- **gfn** (*int or str, must be (0, 1, 2, or "ff")*) -- Hamiltonian for the XTB calculation (GFN0, GFN1, GFN2, or GFNFF).
- **acc** (*float, optional*) -- Accuracy control of the calculation, the lower the tighter several numerical thresholds are chosen.
- **iterations** (*int, optional*) -- The number of iterations in SCC calculation.

- **topo** (*str, optional*) -- Path the a GFNFF-topolgy file. As setting up the topology may take some time for sizable systems, it may be desired to reuse the file.
- **topo_update** (*int*) -- Integer controlling the update interval of the GFNFF topology update. If supplied, the topolgy will be recreated every N-th calculation.
- **mem** (*int*) -- Mememory per core in MB.
- **quiet** (*bool, optional*) -- Suppress creation of log files.

static check_termination(*inp*, **args*, ***kwargs*)

conf_key = 'xtb'

get_energy(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_forces(*atoms*, *coords*, ***prepare_kwargs*)

Meant to be extended.

get_hessian(*atoms*, *coords*, ***prepare_kwargs*)

Get Hessian matrix. Fall back to numerical Hessian, if not overriden.

Preferrably, this method should provide an analytical Hessian.

get_mdrestart_str(*coords*, *velocities*)

coords and velocities have to given in au!

get_pal_env()

get_retry_args()

parse_charges(*fn=None*)

parse_charges_from_json(*fn=None*)

parse_energy(*path*)

parse_gradient(*path*)

parse_hessian(*path*)

parse_md(*path*)

parse_opt(*path*, *keep_log=False*)

parse_topo(*path*)

prepare_add_args(*xcontrol=None*)

prepare_coords(*atoms*, *coords*)

Get 3d coords in Angstrom.

Reshape internal 1d coords to 3d and convert to Angstrom.

Parameters

- **atoms** (*iterable*) -- Atom descriptors (element symbols).
- **coords** (*np.array, 1d*) -- 1D-array holding coordinates in Bohr.

Returns

coords -- 3D-array holding coordinates in Angstrom.

Return type
np.array, 3d

prepare_input(*atoms*, *coords*, *calc_type*, *point_charges=None*)
Meant to be extended.

reattach(*last_calc_cycle*)

run_calculation(*atoms*, *coords*, ***prepare_kwargs*)

run_md(*atoms*, *coords*, *t*, *dt*, *velocities=None*, *dump=1*)
Expecting t and dt in fs, even though xtb wants t in ps!

run_opt(*atoms*, *coords*, *keep=True*, *keep_log=False*)

run_topo(*atoms*, *coords*)

write_mdrestart(*path*, *mdrestart_str*)

17.1.1.3 pysisyphus.cos package

Submodules

pysisyphus.cos.AdaptiveNEB module

class pysisyphus.cos.AdaptiveNEB.**AdaptiveNEB**(*images*, *adapt=True*, *adapt_fact=0.25*, *adapt_between=1*, *scale_fact=False*, *keep_hei=True*, *free_ends=True*, ***kwargs*)

Bases: *NEB*

__init__(*images*, *adapt=True*, *adapt_fact=0.25*, *adapt_between=1*, *scale_fact=False*, *keep_hei=True*, *free_ends=True*, ***kwargs*)

(Free-End) Adaptive Nudged Elastic Band.

Parameters

- **images** (*list of Geometry objects*) -- Images of the band.
- **adapt** (*bool, default True*) -- Whether to adapt the image number or not. This switch is included to support the FreeEndNEB class, that is just a thin wrapper around this class.
- **adapt_fact** (*positive float*) -- Factor that is used to decide wether to adapt. The inital threshold is calculated by multiplying the RMS force of the band with this factor. When the RMS of the force falls below this threshold adaption takes place.
- **adapat_between** (*positive integer*) -- Number of images to interpolate between the highest energy image and its neighbours. The number of starting images must be higher or equal then 2*adapt_between+3, as we reuse/transfer the calculators from the starting images onto the new ones.
- **scale_fact** (*bool, default False*) -- Whether to increase adapt_fact in deeper levels. This may lead to earlier adapation.
- **keep_hei** (*bool, optional*) -- Whether to keep the highest energy image (usually a very good idea) or to interpolate only between the neighbouring images.
- **free_ends** (*bool, default True*) -- Whether to use modified forces on the end images.

adapt_this_cycle(*forces*)

Decide wether to adapt.

Parameters
forces (*np.array*) -- Forces of the previous optimization cycle.

Returns
adapt -- Flag that indicates if adaption should take place in this cycle.

Return type
bool

property forces

See Eq. (7) in [2].

prepare_opt_cycle(**args*, ***kwargs*)

Check for adaption and adapt if needed.

See ChainOfStates.prepare_opt_cycle for a complete docstring.

update_adapt_thresh(*forces*)

Update the adaption threshold.

Parameters
forces (*np.array*) -- Forces of the previous optimization cycle.

pysisyphus.cos.ChainOfStates module

class pysisyphus.cos.ChainOfStates.**ChainOfStates**(*images*, *fix_first=True*, *fix_last=True*, *align_fixed=True*, *climb=False*, *climb_rms=0.005*, *climb_lanczos=False*, *climb_lanczos_rms=0.005*, *climb_fixed=True*, *energy_min_mix=False*, *scheduler=None*, *progress=False*)

Bases: object

as_xyz(*comments=None*)

property atoms

calculate_forces()

property calculator

property cart_coords

Return a flat 1d array containing the cartesian coordinates of all images.

check_for_climbing_start(*ref_rms*)

clear()

concurrent_force_calcs(*images_to_calculate*, *image_indices*)

property coords

Return a flat 1d array containing the coordinates of all images.

property coords3d

describe()

property energy

property forces

get_climbing_forces(*ind*)

get_climbing_indices()

get_dask_client()

get_fixed_indices()

get_hei_index(*energies=None*)

Return index of highest energy image.

get_image_calc_counter_sum()

get_perpendicular_forces(*i*)

[1] Eq. 12

get_splined_hei()

get_tangent(*i*, *kind='upwinding'*, *lanczos_guess=None*, *disable_lanczos=False*)

[1] Equations (8) - (11)

get_tangents()

property gradient

property image_coords

property image_inds

property last_index

log(*message*)

logger = <Logger cos (DEBUG)>

property masses_rep

property max_image_num

property moving_images

property moving_indices

Returns the indices of the images that aren't fixed and can be optimized.

par_image_calc(*image*)

property perpendicular_forces

prepare_opt_cycle(*last_coords*, *last_energies*, *last_forces*)

Implements additional logic in preparation of the next optimization cycle.

Should be called by the optimizer at the beginning of a new optimization cycle. Can be used to implement additional logic as needed for AdaptiveNEB etc.

property results

rms(*arr*)

Root mean square

Returns the root mean square of the given array.

Parameters
arr (*iterable of numbers*)

Returns
rms -- Root mean square of the given array.

Return type
float

set_climbing_forces(*forces*)

set_coords_at(*i*, *coords*)

Called from helpers.procrustes with cartesian coordinates. Then tries to set cartesian coordinate as self.images[i].coords which will raise an error when coord_type != "cart".

set_images(*indices*, *images*)

set_vector(*name*, *vector*, *clear=False*)

set_zero_forces_for_fixed_images()

This is always done in cartesian coordinates, independent of the actual coord_type of the images as setting forces only work with cartesian forces.

valid_coord_types = ('cart', 'cartesian', 'dlc')

zero_fixed_vector(*vector*)

pysisyphus.cos.FreeEndNEB module

class pysisyphus.cos.FreeEndNEB.**FreeEndNEB**(**args*, *fix_first=False*, *fix_last=False*, ***kwargs*)

Bases: *AdaptiveNEB*

__init__(**args*, *fix_first=False*, *fix_last=False*, ***kwargs*)

Simple Free-End-NEB method.

Derived from AdaptiveNEB with disabled adaptation. Only implements Eq. (7) from [2]. For other implementations please see the commit 01bc8812ca6f1cd3645d43e0337d9e3c5fb0ba55. There the other variants are present but I think Eq. (7) in [2] is the simplest & best bet.

pysisyphus.cos.FreezingString module

class pysisyphus.cos.FreezingString.**FreezingString**(*images*, *calc_getter*, *max_nodes=10*, *opt_steps=3*)

Bases: object

property allcoords

as_xyz()

property coords

property energy

property forces

property fully_grown

get_new_image(*coords*, *index*)

get_tangent()

property left_frontier

reparametrize()

property right_frontier

set_new_frontier_nodes()

pysisyphus.cos.GrowingChainOfStates module

class pysisyphus.cos.GrowingChainOfStates.**GrowingChainOfStates**(*images*, *calc_getter*, *max_nodes=10*, ***kwargs*)

Bases: *ChainOfStates*

property arc_dims

get_new_image_from_coords(*coords*, *index*)

property max_image_num

new_node_coords(*k*)

prepare_opt_cycle(**args*, ***kwargs*)

Implements additional logic in preparation of the next optimization cycle.

Should be called by the optimizer at the beginning of a new optimization cycle. Can be used to implement additional logic as needed for AdaptiveNEB etc.

set_new_node(*k*)

pysisyphus.cos.GrowingNT module

class pysisyphus.cos.GrowingNT.**GrowingNT**(*geom*, *step_len=0.5*, *rms_thresh=0.0017*, *r=None*, *final_geom=None*, *between=None*, *bonds=None*, *r_update=True*, *r_update_thresh=1.0*, *stop_after_ts=False*, *require_imag_freq=0.0*, *hessian_at_ts=False*, *out_dir='.'*, *dump=True*)

Bases: object

property P

Projector that keeps perpendicular component.

as_xyz()

property atoms

calc_hessian_for(*other_geom*)

property cart_coords

property cart_forces

check_convergence(**args*, ***kwargs*)

clear_passed()

property coords

property energy

property forces

get_additional_print()

get_energy_and_forces_at(*coords*)

get_energy_at(*coords*)

get_path(*fn*)

static get_r(*geom*, *final_geom*, *bonds*, *r*)

grow_image()

initialize()

log(*message*)

logger = <Logger cos (DEBUG)>

property r

Parallel/search direction.

reparametrize()

Check if GNT can be grown.

pysisyphus.cos.GrowingString module

class pysisyphus.cos.GrowingString.**GrowingString**(*images*, *calc_getter*, *perp_thresh=0.05*, *param='equi'*, *reparam_every=2*, *reparam_every_full=3*, *reparam_tol=None*, *reparam_check='rms'*, *max_micro_cycles=5*, *reset_dlc=True*, *climb=False*, ***kwargs*)

Bases: *GrowingChainOfStates*

property forces

property full_string_image_inds

property fully_grown

Returns wether the string is fully grown. Don't count the first and last node.

get_additional_print()

get_cur_param_density(*kind=None*)

get_new_image(*ref_index*)

Get new image by taking a step from self.images[ref_index] towards the center of the string.

get_tangent(*i*)

[1] Equations (8) - (11)

property image_inds

property left_size

property lf_ind

Index of the left frontier node in self.images.

property nodes_missing

Returns the number of nodes to be grown.

reparam_cart(*desired_param_density*)

reparam_dlc(*desired_param_density*, *thresh=0.001*)

reparametrize()

reset_geometries(*ref_geometry*)

property rf_ind

Index of the right frontier node in self.images.

property right_size

set_coords(*image*, *coords*)

spline(*tangents=False*)

property string_size

pysisyphus.cos.NEB module

class pysisyphus.cos.NEB.**NEB**(*images*, *variable_springs=False*, *k_max=0.3*, *k_min=0.1*, *perp_spring_forces=None*, *bandwidth=None*, ***kwargs*)

Bases: *ChainOfStates*

fmt_k()

property forces

get_parallel_forces(*i*)

get_quenched_dneb_forces(*i*)

See [3], Sec. VI and [4] Sec. D.

get_spring_forces(*i*)

property parallel_forces

set_variable_springs()

update_springs()

pysisyphus.cos.SimpleZTS module

class pysisyphus.cos.SimpleZTS.**SimpleZTS**(*images*, *param='equal'*, ***kwargs*)

Bases: *ChainOfStates*

reparametrize()

Module contents

17.1.1.4 pysisyphus.db package

Submodules

pysisyphus.db.generate_db module

pysisyphus.db.generate_db.**clean**()

pysisyphus.db.generate_db.**generate_db**()

pysisyphus.db.generate_db.**parse_args**(*args*)

pysisyphus.db.generate_db.**run**()

pysisyphus.db.helpers module

pysisyphus.db.helpers.**full_name**(*molecule_name*, *level_name*)

pysisyphus.db.helpers.**get_molecules_levels**()

pysisyphus.db.helpers.**get_path**(*molecule_name*, *level_name*)

pysisyphus.db.level module

pysisyphus.db.molecules module

class pysisyphus.db.molecules.**Molecule**(*name*, *fn*, *charge*, *mult*, *density*, *resname*)

Bases: tuple

charge

Alias for field number 2

density

Alias for field number 4

fn

Alias for field number 1

mult

Alias for field number 3

name

Alias for field number 0

resname

Alias for field number 5

Module contents

17.1.1.5 pysisyphus.drivers package

Submodules

pysisyphus.drivers.afir module

class pysisyphus.drivers.afir.**AFIRPath**(*atoms*, *cart_coords*, *energies*, *forces*, *charge*, *mult*, *opt_is_converged=None*, *gamma=None*, *path_indices=None*)

Bases: object

atoms: tuple

cart_coords: ndarray

charge: int

compatible(*other*)

dump_trj(*fn*)

energies: ndarray

forces: ndarray

gamma: Optional[float] = None

mult: int

opt_is_converged: Optional[bool] = None

path_indices: Optional[List[int]] = None

pysisyphus.drivers.afir.**analyze_afir_path**(*energies*)

pysisyphus.drivers.afir.**automatic_fragmentation**(*atoms*, *coords3d*, *frag1*, *frag2*, *cycles=2*, *p=6*, *bond_factor=1.25*)

Automatic fragmentation scheme as described in SC-AFIR paper [1].

pysisyphus.drivers.afir.**coordinates_similar**(*test_coords3d*, *ref_coords3d*, *rmsd_thresh=0.01*)

Return type
Tuple[bool, int]

pysisyphus.drivers.afir.**decrease_distance**(*coords3d*, *m*, *n*, *frac=0.8*)

pysisyphus.drivers.afir.**determine_target_pairs**(*atoms*, *coords3d*, *min_=1.25*, *max_=5.0*, *active_atoms=None*)

Determine possible target m, n atom pairs for SC-AFIR calculations.

Return type
List[Tuple[int]]

pysisyphus.drivers.afir.**determine_target_pairs_for_geom**(*geom*, ***kwargs*)

Determine possible target m, n atom pairs for SC-AFIR calculations from geom.

Return type
List[Tuple[int]]

pysisyphus.drivers.afir.**find_candidates**(*center*, *bond_sets*)

pysisyphus.drivers.afir.**generate_random_union**(*geoms*, *offset=2.0*, *rng=None*)

Unite fragments into one Geometry with random fragment orientations.

Center, rotate and translate from origin acoording to approximate radius and an offset. Displace along +x, -x, +y, -y, +z, -z.

Results for > 3 fragments don't look so pretty ;).

pysisyphus.drivers.afir.**generate_random_union_ref**(*geoms*, *rng=None*, *opt_kwargs=None*)

Unite fragments into one Geometry with random fragment orientations.

pysisyphus.drivers.afir.**geom_similar**(*test_geom*, *ref_geoms*, ***kwargs*)

Return type
bool

pysisyphus.drivers.afir.**lstsqs_with_reference**(*coords3d*, *ref_coords3d*, *freeze_atoms=None*)

Least-squares w.r.t. reference coordinates while keeping some atoms frozen.

pysisyphus.drivers.afir.**opt_afir_path**(*geom*, *calc_getter*, *afir_kwargs*, *opt_kwargs=None*, *out_dir=None*)

Minimize geometry with AFIR calculator.

pysisyphus.drivers.afir.**prepare_mc_afir**(*geoms*, *rng=None*, ***kwargs*)

Wrapper for generate_random_union(_ref).

pysisyphus.drivers.afir.**prepare_sc_afir**(*geom*, *m*, *n*, *bond_factor=1.2*)

Create perturbed geometry, determine fragments and set AFIR calculator.

pysisyphus.drivers.afir.**relax_afir_path**(*atoms*, *cart_coords*, *calc_getter*, *images=15*, *out_dir=None*)

Sample imagef from AFIR path and do COS relaxation.

pysisyphus.drivers.afir.**run_afir_path**(*geom*, *calc_getter*, *out_dir*, *gamma_max*, *gamma_interval*, *rng*, *ignore_bonds=None*, *bond_factor=1.2*, *afir_kwargs=None*, *opt_kwargs=None*)

Driver for AFIR minimizations with increasing gamma values.

pysisyphus.drivers.afir.**run_afir_paths**(*afir_key*, *geoms*, *calc_getter*, *afir_kwargs=None*, *opt_kwargs=None*, *seed=None*, *N_sample=None*, *rmsd_thresh=0.25*, ***kwargs*)

pysisyphus.drivers.afir.**run_mc_afir_paths**(*geoms*, *calc_getter*, *gamma_max*, *rng*, *N_max=5*, *gamma_interval=(0.0, 1.0)*, *afir_kwargs=None*, *opt_kwargs=None*)

pysisyphus.drivers.afir.**run_sc_afir_paths**(*geom*, *calc_getter*, *gamma_max*, *rng*, *N_max=5*, *N_sample=0*, *gamma_interval=(1.0, 1.0)*, *afir_kwargs=None*, *opt_kwargs=None*, *target_pairs=None*)

pysisyphus.drivers.afir.**weight_function**(*atoms*, *coords3d*, *i*, *j*, *p=6*)

pysisyphus.drivers.barriers module

pysisyphus.drivers.barriers.**do_endopt_ts_barriers**(*ts_geom*, *left_geoms*, *right_geoms=None*, *left_fns=None*, *right_fns=None*, *do_thermo=False*, *T=298.15*, *p=101325*, *calc_getter=None*, *solv_calc_getter=None*, *do_standard_state_corr=False*)

pysisyphus.drivers.birkholz module

pysisyphus.drivers.birkholz.**birkholz_interpolation**(*geoms*, *calc_getter*, *recreate=True*)

pysisyphus.drivers.birkholz.**bond_order**(*r*, *r0*, *b=2*)

Bond order for given bond length and reference length.

Eq. (3) in [1].

pysisyphus.drivers.birkholz.**bond_orders**(*coords3d*, *bond_indices*, *r0s*)

List of bond orders.

pysisyphus.drivers.birkholz.**bond_orders_for_geom**(*geom*, *bond_indices*)

Wrapper for bond_orders for simple use with Geometry.

pysisyphus.drivers.birkholz.**get_r0s**(*geom*, *bond_indices*)

Reference bond lengths as sum of covalent radii.

pysisyphus.drivers.birkholz.**length_for_bond_order**(*bo*, *r0*, *b=2*)

Return bond length for given bond order and reference length.

Eq. (3) in [1].

pysisyphus.drivers.merge module

pysisyphus.drivers.merge.**align_on_subset**(*geom1*, *union*, *del1=None*)

Align 'union' onto subset of 'geom1'

pysisyphus.drivers.merge.**hardsphere_merge**(*geom1*, *geom2*)

pysisyphus.drivers.merge.**merge_geoms**(*geom1*, *geom2*, *geom1_del=None*, *geom2_del=None*, *make_bonds=None*)

Merge geom1 and geom2 while keeping the original coordinates.

Supports deleting certain atoms.

pysisyphus.drivers.merge.**merge_opt**(*union*, *bond_diff*, *ff='mmff94'*)

Fragment merging along given bond by forcefield optimization.

pysisyphus.drivers.merge.**merge_with_frozen_geom**(*frozen_geom*, *lig_geom*, *make_bonds*, *frozen_del*, *lig_del*, *ff='mmff94'*)

pysisyphus.drivers.merge.**parse_args**(*args*)

pysisyphus.drivers.merge.**prepare_merge**(*geom1*, *bond_diff* , *geom2=None*, *del1=None*, *del2=None*, *dump=False*)

pysisyphus.drivers.merge.**run_merge**()

pysisyphus.drivers.opt module

class pysisyphus.drivers.opt.**OptResult**(*opt*, *geom*, *fn*)

Bases: object

fn: Path

geom: *Geometry*

opt: *Optimizer*

pysisyphus.drivers.opt.**get_optimal_bias**(*ref_geom*, *calc_getter*, *opt_key*, *opt_kwargs*, *k_max*, *k_min=0.0*, *rmsd_target=0.188973*, *rmsd_thresh=None*, *rmsd_kwargs=None*, *k_thresh=0.001*, *strict=True*)

Driver to determine optimal bias value k for RMSD restraint.

Parameters

- **ref_geom** (*Geometry*) -- Reference geometry. Starting point of the optimizations and reference for RMSD calculations.
- **calc_getter** (Callable) -- Function that returns the actual calculator, providing the energy and its derivatives.
- **opt_key** (str) -- Determines optimizer type. See pysisyphus.optimizers.cls_map.
- **opt_kwargs** (Dict) -- Optional dict of arguments passed to the optimizer.
- **k_max** (float) -- Maximum absolute value of bias factor k. Must be a > k_min.
- **k_max** -- Minimum absolute value of bias factor k. Must be a positive number >= 0.0. Defaults to 0.0.
- **rmsd_target** (float, default: 0.188973) -- Target RMSD value in au. Defaults to 0.188973 a0 (approx. 0.1).
- **rmsd_thresh** (Optional[float], default: None) -- Allowed deviation from rmsd_target in au. If omitted, 5% of rmsd_target are used.
- **rmsd_kwargs** (Optional[Dict], default: None) -- Additional keyword arguments that are passed to the RMSD class, e.g., atom_indices.
- **k_thresh** (float, default: 0.001) -- When the absolute value of k_bias - k_min or k_max becomes smaller than k_thresh, the bisection is aborted.
- **strict** (default: True) -- If True, AssertionError is raised when an optimization did not converged.

Return type

Tuple[*OptResult*, float, bool]

Returns

- *opt_result* -- OptimizationResult object containig the Optimizer object.
- *k_opt* -- Optimal value of k_bias.
- *valid_k* -- Whether an appropriate bias value k was found.

pysisyphus.drivers.opt.**opt_davidson**(*opt*, *tsopt=True*, *res_rms_thresh=0.0001*)

pysisyphus.drivers.opt.**run_opt**(*geom*, *calc_getter*, *opt_key*, *opt_kwargs=None*, *iterative=False*, *iterative_max_cycles=5*, *iterative_thresh=-15*, *iterative_scale=2.0*, *cart_hessian=None*, *print_thermo=False*, *title='Optimization'*, *copy_final_geom=None*, *level=0*)

pysisyphus.drivers.perf module

pysisyphus.drivers.perf.**print_perf_results**(*results*)

pysisyphus.drivers.perf.**run_perf**(*geom*, *calc_getter*, *pals=None*, *mems=None*, *pal_range=None*, *mem_range=None*, *repeat=1*, *kind='forces'*)

pysisyphus.drivers.pka module

pysisyphus.drivers.pka.**G_aq_from_h5_hessian**(*h5_hessian*, *solv_en*, *T=298.15*, *p=101325*)

pysisyphus.drivers.pka.**direct_cycle**(*acid_h5*, *base_h5*, *acid_solv_en*, *base_solv_en*, *G_aq_H=None*, *G_gas_H=-6.28*, *dG_solv_H=-265.9*, *T=298.15*, *p=101325*)

pysisyphus.drivers.precon_pos_rot module

prp a901cdfacc579eb63b193cbc9043212e8b57746f pysis 340ab6105ac4156f0613b4d0e8f080d9f195530c do_trans accidentally disabled in transtorque

class pysisyphus.drivers.precon_pos_rot.**SteepestDescent**(*geom*, *max_cycles=1000*, *max_step=0.05*, *rms_force=0.05*, *rms_force_only=True*, *prefix=None*, *dump=False*, *dump_every=100*, *print_every=100*)

Bases: `object`

run()

pysisyphus.drivers.precon_pos_rot.**center_fragments**(*frag_list*, *geom*)

pysisyphus.drivers.precon_pos_rot.**form_A**(*frags*, *which_frag*, *formed_bonds*)

Construct the A-matrices.

AR[(m, n)] (AP[(m, n)]) contains the subset of atoms in Rm (Pm) that forms bonds with Rn (Pn).

pysisyphus.drivers.precon_pos_rot.**get_fragments_and_bonds**(*geoms*)

pysisyphus.drivers.precon_pos_rot.**get_rot_mat**(*coords3d_1*, *coords3d_2*, *center=False*)

pysisyphus.drivers.precon_pos_rot.**get_steps_to_active_atom_mean**(*frag_lists*, *iter_frag_lists*, *ind_dict*, *coords3d*, *skip=True*)

pysisyphus.drivers.precon_pos_rot.**get_which_frag**(*frags*)

pysisyphus.drivers.precon_pos_rot.**precon_pos_rot**(*reactants*, *products*, *prefix=None*, *config={'s2_hs_kappa': 1.0, 's4_hs_kappa': 50.0, 's4_v_kappa': 1.0, 's4_w_kappa': 1.0, 's5_hs_kappa': 10.0, 's5_rms_force': 0.01, 's5_trans': True, 's5_v_kappa': 1.0, 's5_w_kappa': 3.0, 's5_z_kappa': 2.0}*)

pysisyphus.drivers.precon_pos_rot.**report_frags**(*rgeom*, *pgeom*, *rfrags*, *pfrags*, *rbond_diff*, *pbond_diff*)

pysisyphus.drivers.precon_pos_rot.**report_mats**(*name*, *mats*)

pysisyphus.drivers.precon_pos_rot.**rotate_inplace**(*frags*, *union*, *bonds*)

pysisyphus.drivers.precon_pos_rot.**run_precontr**(*reactant_geom*, *product_geom*, ***kwargs*)

pysisyphus.drivers.rates module

class pysisyphus.drivers.rates.**ReactionRates**(*from_*, *barrier*, *barrier_si*, *temperature*, *imag_wavenumber*, *imag_frequency*, *rate_eyring*, *kappa_eyring*, *rate_wigner=None*, *kappa_wigner=None*, *rate_bell=None*, *kappa_bell=None*, *rate_eckart=None*, *kappa_eckart=None*)

Bases: object

barrier: float

barrier_si: float

from_: str

imag_frequency: float

imag_wavenumber: float

kappa_bell: Optional[float] = None

kappa_eckart: Optional[float] = None

kappa_eyring: float

kappa_wigner: Optional[float] = None

rate_bell: Optional[float] = None

rate_eckart: Optional[float] = None

rate_eyring: float

rate_wigner: Optional[float] = None

temperature: float

pysisyphus.drivers.rates.**bell_corr**(*temperature*, *imag_frequency*)

Tunneling correction according to Bell.

See https://onlinelibrary.wiley.com/doi/10.1002/anie.201708489 eq. (1) and eq. (2).

Parameters

- **temperature** (float) -- Temperature in Kelvin.
- **imag_frequency** (float) -- Imaginary frequency in 1/s.

Returns
Unitless tunneling correction according to Bell. Negative kappas are meaningless.

Return type
kappa

pysisyphus.drivers.rates.**eckart_corr**(*fw_barrier_height*, *bw_barrier_height*, *temperature*, *imag_frequency*)

Tunneling correction according to Eckart.

See [3], [4] and [5]. The correction should be independent of the order fw_barrier_height/bw_barrier_height.

Parameters

- **fw_barrier_height** (float) -- Barrier height in forward direction in Hartree.
- **bw_barrier_height** (float) -- Barrier height in backward direction in Hartree.
- **temperature** (float) -- Temperature in Kelvin.
- **imag_frequency** (float) -- Frequency in 1/s of the imaginary mode at the TS.

Returns
Unitless tunneling correction according to Eckart.

Return type
kappa

pysisyphus.drivers.rates.**eckart_corr_brown**(*fw_barrier_height*, *bw_barrier_height*, *temperature*, *imag_frequency*)

Tunneling correction according to Eckart.

Wrapper for the TUNL subroutine as given in the appendix of [5].

Parameters

- **fw_barrier_height** (float) -- Barrier height in forward direction in Hartree.
- **bw_barrier_height** (float) -- Barrier height in backward direction in Hartree.
- **temperature** (float) -- Temperature in Kelvin.
- **imag_frequency** (float) -- Frequency in 1/s of the imaginary mode at the TS.

Returns
Unitless tunneling correction according to Eckart.

Return type
kappa

pysisyphus.drivers.rates.**eyring_rate**(*barrier_height*, *temperature*, *trans_coeff=1.0*)

Rate constant in 1/s from the Eyring equation.

See https://pubs.acs.org/doi/10.1021/acs.organomet.8b00456 "Reaction Rates and Barriers" on p. 3234 and eq. (8).

Parameters

- **barrier_height** (float) -- Barrier height (energy, enthalpy, gibbs energy, ...) in Hartree.
- **temperature** (Union[_SupportsArray[dtype[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]]) -- Temperature in Kelvin.
- **trans_coeff** (float, default: 1.0) -- Unitless transmission coefficient, e.g., obtained from Wigner or Eckart correction.

Returns
Eyring reaction rate in 1/s.

Return type
rate

pysisyphus.drivers.rates.**get_rates**(*temperature*, *reactant_thermos*, *ts_thermo*, *product_thermos=None*)

pysisyphus.drivers.rates.**get_rates_for_geoms**(*temperature*, *reactant_geoms*, *ts_geom*, *product_geoms*)

pysisyphus.drivers.rates.**get_rates_for_hessians**(*temperature*, *reactant_h5s*, *ts_h5*, *product_h5s*)

pysisyphus.drivers.rates.**harmonic_tst_rate**(*barrier_height*, *temperature*, *rs_part_func*, *ts_part_func*, *trans_coeff=1.0*)

Rate constant in 1/s from harmonic TST.

See http://dx.doi.org/10.18419/opus-9841, chapter 5. Contrary to the Eyring rate this function does only takes a scalar temperature as the partition functions are also functions of the temperature and would have to be recalculated for different temperatures.

A possible extension would be to also support multiple rs/ts partition functions, one for each temperature.

Parameters

- **barrier_height** (float) -- Barrier height (energy, enthalpy, gibbs energy, ...) in Hartree.
- **rs_part_func** (float) -- Partition function of the reactant state.
- **ts_part_func** (float) -- Partition function of the transition state.
- **temperature** (float) -- Temperature in Kelvin.
- **trans_coeff** (float, default: 1.0) -- Unitless transmission coefficient, e.g., obtained from Wigner or Eckart correction.

Returns
HTST reaction rate in 1/s.

Return type
rate

pysisyphus.drivers.rates.**render_rx_rates**(*rx_rates*)

Return type
str

pysisyphus.drivers.rates.**tunl**(*alph1*, *alph2*, *U*, *strict=False*)

Eckart correction factor for rate constants according to Brown.

Python adaption of the TUNL subroutine in 4. Appendix of [5].

Parameters

- **alph1** (float) -- Unitless barrier height descriptor. 2 V1 / (h nu*); see (2) in [5].

- **alph2** (float) -- Unitless barrier heigth descriptor. 2 V2 / (h nu*); see (2) in [5].
- **u** -- Unitless curvature descriptor. h nu* / kT; see (2) in [5].
- **strict** (bool, default: False) -- If enabled, arguments are bound checked. Will raise AssertionError if they are out of bonds. TUNL was found to yield accurate results when the arguments are within bounds.

Returns
Unitless tunneling correction according to Eckart.

Return type
G

pysisyphus.drivers.rates.**wigner_corr**(*temperature*, *imag_frequency*)

Tunneling correction according to Wigner.

See https://doi.org/10.1002/qua.25686 eq. (12) and https://doi.org/10.1007/978-3-642-59033-7_9 for the original publication.

Parameters

- **temperature** (float) -- Temperature in Kelvin.
- **imag_frequency** (float) -- Imaginary frequency in 1/s.

Returns
Unitless tunneling correction according to Wigner.

Return type
kappa

pysisyphus.drivers.replace module

pysisyphus.drivers.replace.**get_bond_subgeom**(*geom*, *ind*, *invert=False*)

pysisyphus.drivers.replace.**normalize_replacements**(*replacements*)

pysisyphus.drivers.replace.**parse_args**(*args*)

pysisyphus.drivers.replace.**replace_atom**(*geom*, *ind*, *repl_geom*, *repl_ind*, *return_full=True*, *opt=False*, *use_xtb=False*, *charge=0*, *mult=1*)

Replace atom with fragment.

pysisyphus.drivers.replace.**replace_atoms**(*geom*, *replacements*, *opt=False*, *use_xtb=False*, *charge=0*, *mult=1*)

pysisyphus.drivers.replace.**run**()

pysisyphus.drivers.replace.**run_opt**(*geom*, *freeze_inds*, *ff='uff'*, *use_xtb=False*, *charge=0*, *mult=1*)

pysisyphus.drivers.scan module

pysisyphus.drivers.scan.**relaxed_1d_scan**(*geom*, *calc_getter*, *constrain_prims*, *start*, *step_size*, *steps*, *opt_key*, *opt_kwargs*, *pref=None*, *callback=None*)

pysisyphus.drivers.scan.**relaxed_scan**(*geom*, *calc_getter*, *constrain_prims*, *target_values*, *title*, *max_cycles=25*, *trust_radius=0.5*, *thresh=0.01*, *dump=True*)

Relaxed scan, allowing fixing of multiple primitive internals.

pysisyphus.drivers.thermo module

pysisyphus.drivers.thermo.**parse_args**(*args*)

pysisyphus.drivers.thermo.**run_thermo**()

Module contents

17.1.1.6 pysisyphus.dynamics package

Submodules

pysisyphus.dynamics.Gaussian module

class pysisyphus.dynamics.Gaussian.**DummyColvar**

Bases: object

eval(*x*)

class pysisyphus.dynamics.Gaussian.**Gaussian**(*w=1*, *s=1*, *x0=None*, *colvar=None*, *dump_name=None*)

Bases: object

__init__(*w=1*, *s=1*, *x0=None*, *colvar=None*, *dump_name=None*)

See:
https://doi.org/10.1016/j.cpc.2018.02017

V(f(x)) = w * exp(-(f(x) - f0)**2 / (2*s**2))

F(x) = -dV/dx = -dV/df * df/dx

See also:
https://doi.org/10.1103/PhysRevLett.90.238302

calculate(*coords*, *x0=None*, *gradient=False*)

Return potential and gradient for Gaussian(s) centered at x0.

dump(*step*, *s*, *w*, *center*)

eval(*coords*, *x0=None*)

gradient(*coords*, *x0=None*)

property s

value(*coords*, *x0=None*)

property w

pysisyphus.dynamics.colvars module

class pysisyphus.dynamics.colvars.**CVBend**(*indices*, ***kwargs*)

Bases: *Colvar*

value(*c3d*)

class pysisyphus.dynamics.colvars.**CVDistance**(*indices*, ***kwargs*)

Bases: *Colvar*

value(*c3d*)

class pysisyphus.dynamics.colvars.**CVTorsion**(*indices*, ***kwargs*)

Bases: *Colvar*

value(*c3d*)

class pysisyphus.dynamics.colvars.**Colvar**(*force_agrad=False*)

Bases: object

eval(*coords*)

gradient(*c3d*)

abstract value(*c3d*)

pysisyphus.dynamics.colvars.**get_colvar**(*key*, *kwargs*)

pysisyphus.dynamics.driver module

class pysisyphus.dynamics.driver.**MDResult**(*coords*, *t_ps*, *step*, *terminated*, *T*, *E_tot*)

Bases: tuple

E_tot

Alias for field number 5

T

Alias for field number 4

coords

Alias for field number 0

step

Alias for field number 2

t_ps

Alias for field number 1

terminated

Alias for field number 3

pysisyphus.dynamics.driver.**get_data_model**(*atoms*, *dump_steps*)

pysisyphus.dynamics.driver.**md**(*geom*, *v0*, *steps*, *dt*, *remove_com_v=True*, *thermostat=None*, *T=298.15*, *timecon=100*, *term_funcs=None*, *constraints=None*, *constraint_kwargs=None*, *gaussians=None*, *verbose=True*, *print_stride=50*, *dump_stride=None*, *h5_group_name='run'*)

Velocity verlet integrator.

Parameters

- **geom** (Geometry) -- The system for which the dynamics are to be run.
- **v0** (*np.array, floats*) -- Initial velocities in Bohr/fs.
- **steps** (*float*) -- Number of simulation steps.
- **dt** (*float*) -- Timestep in fs.
- **remove_com_v** (*bool, default=True*) -- Remove center-of-mass velocity.
- **thermostat** (*str, optional, default None*) -- Which and whether to use a thermostat.
- **T** (*float, optional, default=None*) -- Desired temperature in thermostated runs.
- **timecon** (*float*) -- Timeconsanst of the thermostat in fs.
- **term_funcs** (*dict, optional*) -- Iterable of functions that are called with the atomic coordinates in every MD cycle and result in termination
- **constraints** (*2d iterable, optional*) -- 2D iterable containing atom indices describing constrained bond lengths. of the MD integration when they evaluate to true.
- **constraint_kwargs** (*dict, optional*) -- Keyword arguments for the constraint algorithm.
- **gaussians** (*list, optional, default=None*) -- List of Gaussians to be used in a metadynamics run.
- **verbose** (*bool, default=True*) -- Do additional printing when True.
- **print_stride** (*int, default=50*) -- Report every n-th step.
- **dump_stride** (*int, default=None*) -- If given, MD data will be dumped to a HDF5 file every n-th step.
- **str** (*h5_group_name =*) -- Name of the HDF5 group, used for dumping.
- **optional** -- Name of the HDF5 group, used for dumping.

pysisyphus.dynamics.helpers module

pysisyphus.dynamics.helpers.**dump_coords**(*atoms*, *coords*, *trj_fn*)

pysisyphus.dynamics.helpers.**energy_forces_getter_closure**(*geom*)

pysisyphus.dynamics.helpers.**get_mb_velocities**(*masses*, *cart_coords*, *T*, *remove_com_v=True*, *remove_rot_v=True*, *seed=None*)

Initial velocities from Maxwell-Boltzmann distribution.

Parameters

- **masses** (*np.array, 1d, shape (number of atoms,)*) -- Atomic masses in amu.

- **cart_coords** (*iterable, 1d, shape (3 * number of atoms,)*) -- Atomic cartesian coordinates. Needed for removal of rotation.
- **T** (*float*) -- Temperature in Kelvin.
- **remove_com_v** (*bool, default=True, optional*) -- Whether to remove center-of-mass velocity.
- **remove_rot_v** (*bool, default=True, optional*) -- Whether to remove rotational velocity.
- **seed** (*int, default=None, optional*) -- Seed for the random-number-generator.

Returns
v -- Initial velocities in Bohr/fs.

Return type
np.array, 2d, shape (number of atoms, 3)

pysisyphus.dynamics.helpers.**get_mb_velocities_for_geom**(*geom*, *T*, *remove_com_v=True*, *remove_rot_v=True*, *seed=None*)

Initial velocities from Maxwell-Boltzmann distribution.

See 'get_mb_velocities' for explanation.

pysisyphus.dynamics.helpers.**kinetic_energy_for_temperature**(*atom_number*, *T*, *fixed_dof=0*)

Kinetic energy for given temperature and number of atoms.

Each atom has three degrees of freedom (1/2 * 3 == 3/2).

Parameters

- **atom_number** (*int*) -- Number of atoms. Each atom has three degrees of freedom.
- **T** (*float*) -- Temperature in Kelvin.
- **fixed_dof** (*int, optional, default=0*) -- Number of fixed degrees of freedom, e.g. 3 when the center-of-mass velocity is removed.

Returns
E_kin -- Kinetic energy in Hartree.

Return type
float

pysisyphus.dynamics.helpers.**kinetic_energy_from_velocities**(*masses*, *velocities*)

Kinetic energy for given velocities and masses.

Parameters

- **masses** (*1d array, shape (number of atoms,)*) -- Atomic masses in amu.
- **velocities** (*2d array, (number of atoms, 3)*) -- Atomic velocities in Bohr/fs.

Returns
E_kin -- Kinetic energy in Hartree.

Return type
float

pysisyphus.dynamics.helpers.**remove_com_velocity**(*v*, *masses*, *keep_norm=True*)

Remove center-of-mass velocity.

Returned units vary with the given input units.

Parameters

- **v** (*np.array, 2d, shape (number of atoms, 3)*) -- Velocities.
- **masses** (*np.array, 1d, shape (number of atoms,)*) -- Atomic masses.
- **keep_norm** (*bool, default=True*) -- Whether to rescale v to its original norm, after removal of v_com.

Returns
v -- Velocities without center-of-mass velocity.

Return type
np.array

pysisyphus.dynamics.helpers.**scale_velocities_to_energy**(*masses*, *v*, *E_desired*)

Scale velocities to a given temperature.

Parameters

- **masses** (*np.array, 1d, shape (number of atoms,)*) -- Atomic masses in amu.
- **v** (*np.array, 2d, shape (number of atoms, 3)*) -- (Unscaled) velocities in Bohr/fs.
- **E_desired** (*float*) -- Desired kinetic energy in Hartree.

Returns
v -- Scaled velocities in Bohr/fs.

Return type
np.array, 2d, shape (number of atoms, 3)

pysisyphus.dynamics.helpers.**scale_velocities_to_temperatue**(*masses*, *v*, *T_desired*, *fixed_dof=0*)

Scale velocities to a given temperature.

Parameters

- **masses** (*np.array, 1d, shape (number of atoms,)*) -- Atomic masses in amu.
- **v** (*np.array, 2d, shape (number of atoms, 3)*) -- (Unscaled) velocities in Bohr/fs.
- **T_desired** (*float*) -- Desired temperature in Kelvin.
- **fixed_dof** (*int, optional, default=0*) -- Number of fixed degrees of freedom, e.g. 3 when the center-of-mass velocity is removed.

Returns
v -- Scaled velocities in Bohr/fs.

Return type
np.array, 2d, shape (number of atoms, 3)

pysisyphus.dynamics.helpers.**temperature_for_kinetic_energy**(*atom_number*, *E_kin*, *fixed_dof=0*)

Temperature for given kinetic energy and atom number.

Each atom has three degrees of freedom (1/2 * 3 == 3/2).

Parameters

- **atom_number** (*int*) -- Number of atoms. Each atom has three degrees of freedom.
- **E_kin** (*float*) -- Kinetic energy in Hartree.
- **fixed_dof** (*int, optional, default=0*) -- Number of fixed degrees of freedom, e.g. 3 when the center-of-mass velocity is removed.

Returns
Temperature -- Temperature in Kelvin.

Return type
float

pysisyphus.dynamics.helpers.**unscaled_velocity_distribution**(*masses*, *T*, *seed=None*)

exp(- v^2 * m / (2 * kT)) ln() - 1/2 * v^2 * m / kT v^2 -2 * ln() * kT / m v ±(-2 * ln() * kT / m)**0.5 v ±(-2 * k)**0.5 * (ln() * T / m)**0.5

The first term of the RHS is constant. As we later scale the velocities we neglect it. Don't use these velocities unscaled!

v ±(ln() * T / m)**0.5

pysisyphus.dynamics.lincs module

pysisyphus.dynamics.lincs.**lincs_closure**(*geom*, *constraints*, *order=4*)

Drop conn_nums and keep **conns_** in a list of list, instead of an array. We could do everything with

for i, conn_constr in enumerate(conns)
pass

instead of the pseudo-code way as given in the paper.

pysisyphus.dynamics.mdp module

pysisyphus.dynamics.mdp.**MDPResult**

alias of MDResult

pysisyphus.dynamics.mdp.**mdp**(*geom*, *steps*, *dt*, *term_funcs=None*, *steps_init=None*, *E_excess=0.0*, *displ_length=0.1*, *epsilon=0.0005*, *ascent_alpha=0.05*, *max_ascent_steps=25*, *max_init_trajs=10*, *dump=True*, *seed=None*, *external_md=False*)

pysisyphus.dynamics.mdp.**parse_raw_term_func**(*raw_term_func*)

pysisyphus.dynamics.mdp.**parse_raw_term_funcs**(*raw_term_funcs*)

pysisyphus.dynamics.mdp.**run_md**(*geom*, *dt*, *steps*, *v0=None*, *term_funcs=None*, *external=False*)

pysisyphus.dynamics.rattle module

pysisyphus.dynamics.rattle.**rattle_closure**(*geom*, *constraints*, *dt*, *tol=0.001*, *max_cycles=25*, *energy_forces_getter=None*, *remove_com_v=True*)

pysisyphus.dynamics.thermostats module

pysisyphus.dynamics.thermostats.**berendsen_closure**(*sigma*, *dof*, *dt*, *tau=100*, *rng=None*)

https://doi.org/10.1063/1.448118

pysisyphus.dynamics.thermostats.**csvr_closure**(*sigma*, *dof*, *dt*, *tau=100*, *rng=None*)

Parameters

- **sigma** (*float*) -- Target average value of the kinetic energy (1/2 dof k_b T) in the same units as cur_kinetic_energy.
- **dof** (*int*) -- Degrees of freedom.
- **tau** (*float*) -- Timeconstant of the thermostat. tau : float Timeconstant of the thermostat.
- **rng** (*numpy.random.Generator, optional*) -- Instances of a random number generator (RNG). If it is not provided the module-level RNG will be used.

pysisyphus.dynamics.thermostats.**csvr_closure_2**(*sigma*, *dof*, *dt*, *tau=100*, *rng=None*)

pysisyphus.dynamics.thermostats.**sum_noises**(*num*, *rng=None*)

Parameters

- **num** (*int*) -- Number of independent Gaussian noises to be squared.
- **rng** (*numpy.random.Generator, optional*) -- Instances of a random number generator (RNG). If it is not provided the module-level RNG will be used.

Module contents

17.1.1.7 pysisyphus.intcoords package

Submodules

pysisyphus.intcoords.Bend module

class pysisyphus.intcoords.Bend.**Bend**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Primitive*

pysisyphus.intcoords.Bend2 module

class pysisyphus.intcoords.Bend2.**Bend2**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Bend*

pysisyphus.intcoords.BondedFragment module

class pysisyphus.intcoords.BondedFragment.**BondedFragment**(*indices*, *bond_indices*, ***kwargs*)

Bases: *Primitive*

pysisyphus.intcoords.Cartesian module

class pysisyphus.intcoords.Cartesian.**Cartesian**(**args*, ***kwargs*)

Bases: *Primitive*

class pysisyphus.intcoords.Cartesian.**CartesianX**(**args*, ***kwargs*)

Bases: *Cartesian*

cart_axis = 0

class pysisyphus.intcoords.Cartesian.**CartesianY**(**args*, ***kwargs*)

Bases: *Cartesian*

cart_axis = 1

class pysisyphus.intcoords.Cartesian.**CartesianZ**(**args*, ***kwargs*)

Bases: *Cartesian*

cart_axis = 2

pysisyphus.intcoords.CartesianCoords module

class pysisyphus.intcoords.CartesianCoords.**CartesianCoords**(*atoms*, *coords3d*, *masses*, *freeze_atoms=None*, *, *mass_weighted=False*, ***kwargs*)

Bases: *CoordSys*

property coords: ndarray[Any, dtype[_ScalarType_co]]

Getter for coordinates in this coordinate system.

property coords3d: ndarray[Any, dtype[_ScalarType_co]]

Getter for 3d Cartesian coordinates.

property inv_masses_rep_sqrt: ndarray[Any, dtype[_ScalarType_co]]

property masses: ndarray[Any, dtype[_ScalarType_co]]

property masses_sqrt: ndarray[Any, dtype[_ScalarType_co]]

project_hessian(*hessian*)

Project Hessian in the current coordinate system.

transform_forces(*cart_forces*)

Transform Cartesian forces to this coordinate system.

Return type
ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]

transform_hessian(*cart_hessian*, *int_gradient=None*)

Transform Cartesian Hessian to this coordinate system.

transform_int_step(*step*, *update_constraints=False*, *pure=False*)

Transform step in this coordinate system to Cartesians.

Return type
ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]

property typed_prims: List

List of (primitive) internal coordinates.

May be empty, e.g., when the coordinate system is Cartesian.

class pysisyphus.intcoords.CartesianCoords.**MWCartesianCoords**(**args*, ***kwargs*)

Bases: *CartesianCoords*

pysisyphus.intcoords.Coords module

class pysisyphus.intcoords.Coords.**CoordSys**(**args*, ***kwargs*)

Bases: Protocol

abstract property coords: ndarray[Any, dtype[_ScalarType_co]]

Getter for coordinates in this coordinate system.

abstract property coords3d: ndarray[Any, dtype[_ScalarType_co]]

Getter for 3d Cartesian coordinates.

abstract project_hessian(*hessian*)

Project Hessian in the current coordinate system.

Return type
ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]

abstract transform_forces(*cart_forces*)

Transform Cartesian forces to this coordinate system.

Return type
ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]

abstract transform_hessian(*cart_hessian*, *int_gradient*)

Transform Cartesian Hessian to this coordinate system.

Return type
ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]

abstract transform_int_step(*step*, *update_constraints=False*, *pure=False*)

Transform step in this coordinate system to Cartesians.

Return type
ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]

abstract property typed_prims: List

List of (primitive) internal coordinates.

May be empty, e.g., when the coordinate system is Cartesian.

pysisyphus.intcoords.DLC module

class pysisyphus.intcoords.DLC.**DLC**(**args*, *full_set=True*, ***kwargs*)

Bases: *RedundantCoords*

property B

Wilson B-Matrix in the non-redundant subspace.

property U

backtransform_hessian(**args*, ***kwargs*)

Transform Hessian in internal coordinates to Cartesians.

property constraints

property coords

freeze_primitives(*typed_prims*)

Freeze primitive internal coordinates.

Parameters

typed_prims (*iterable of typed primitives*) -- Iterable containing typed_primitives, starting with a PrimType and followed by atom indices.

get_active_set(*B*, *inv_thresh=None*)

See [5] between Eq. (7) and Eq. (8) for advice regarding the threshold.

get_constrained_U(*constraint_vecs*)

project_hessian(*H*)

As we already work in the non-redundant subspace we don't have to project/shift the hessian as we do it for simple redundant internal coordinates.

project_primitive_on_active_set(*prim_ind*)

reset_constraints()

set_active_set()

transform_hessian(*cart_hessian*, *int_gradient=None*)

Transform Cartesian Hessian to DLC.

transform_int_step(*step*, **args*, ***kwargs*)

As the transformation is done in primitive internal coordinates we convert the DLC back to primitive coordinates.

class pysisyphus.intcoords.DLC.**HDLC**(**args*, ***kwargs*)

Bases: *DLC*

pysisyphus.intcoords.DistanceFunction module

class pysisyphus.intcoords.DistanceFunction.**DistanceFunction**(*indices*, *coeff=-1*, ***kwargs*)

Bases: *Primitive*

pysisyphus.intcoords.DummyTorsion module

class pysisyphus.intcoords.DummyTorsion.**DummyTorsion**(*indices*, **args*, *fix_inner=True*, ***kwargs*)

Bases: *Torsion*

static get_coords3d_and_indices_ext(*coords3d*, *indices*)

static get_fourth_coords(*coords3d*, *indices*, *r=1.889*, *theta=90*)

M N <- add

/

u v

/

P

m, o, p, n = indices

pysisyphus.intcoords.LinearBend module

class pysisyphus.intcoords.LinearBend.**LinearBend**(**args*, *complement=False*, ***kwargs*)

Bases: *Primitive*

calculate(*coords3d*, *indices=None*, *gradient=False*)

jacobian(*coords3d*, *indices=None*)

pysisyphus.intcoords.LinearDisplacement module

class pysisyphus.intcoords.LinearDisplacement.**LinearDisplacement**(**args*, *complement=False*, ***kwargs*)

Bases: *Primitive*

calculate(*coords3d*, *indices=None*, *gradient=False*)

jacobian(*coords3d*, *indices=None*)

pysisyphus.intcoords.OutOfPlane module

class pysisyphus.intcoords.OutOfPlane.**OutOfPlane**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Primitive*

[1] https://doi.org/10.1002/(SICI)1096-987X(19990730)20:10<1067::AID-JCC9>3.0.CO;2-V
Lee, 1999

pysisyphus.intcoords.PrimTypes module

pysisyphus.intcoords.PrimTypes.**PT**

alias of *PrimTypes*

class pysisyphus.intcoords.PrimTypes.**PrimTypes**(*value*)

Bases: *OrderedEnum*

An enumeration.

AUX_BOND = 1

AUX_INTERFRAG_BOND = 4

BEND = 5

BEND2 = 28

BOND = 0

BONDED_FRAGMENT = 25

CARTESIAN = 21

CARTESIAN_X = 22

CARTESIAN_Y = 23

CARTESIAN_Z = 24

DISTANCE_FUNCTION = 27

DUMMY_IMPROPER = 30

DUMMY_TORSION = 26

HYDROGEN_BOND = 2

IMPROPER_DIHEDRAL = 9

INTERFRAG_BOND = 3

LINEAR_BEND = 6

LINEAR_BEND_COMPLEMENT = 7

LINEAR_DISPLACEMENT = 11

LINEAR_DISPLACEMENT_COMPLEMENT = 12

OUT_OF_PLANE = 10

PROPER_DIHEDRAL = 8

PROPER_DIHEDRAL2 = 29

ROBUST_TORSION1 = 31

ROBUST_TORSION2 = 32

ROTATION = 17

ROTATION_A = 18

ROTATION_B = 19

ROTATION_C = 20

TRANSLATION = 13

TRANSLATION_X = 14

TRANSLATION_Y = 15

TRANSLATION_Z = 16

pysisyphus.intcoords.PrimTypes.**get_bonded_frag_coord**()

pysisyphus.intcoords.PrimTypes.**get_dist_func**()

pysisyphus.intcoords.PrimTypes.**get_rot_coord**(*cls*)

pysisyphus.intcoords.PrimTypes.**normalize_prim_input**(*prim_inp*)

Normalize input for define_prims and constrain_prims

The intcoords.RedundantCoords constructor expects lists of integer lists (tuples) for arguments like 'define_prims' and 'constrain_prims'. The first item of every list determines the type of primitive coordinate. Currently there are about 20 different types and it is hard to remember all of them.

So we also allow a more human friendly input, that is normalized here. The most common primitives are:

0: BOND 5: BEND 8: PROPER_DIHEDRAL

This function maps inputs like ["BOND", 1, 2] to [PrimTypes.BOND, 1, 2] etc.

Always returns a list of tuples, as some prim_inps expand to multiple coordinates, e.g., XYZ or ATOM.

pysisyphus.intcoords.PrimTypes.**normalize_prim_inputs**(*prim_inps*)

pysisyphus.intcoords.PrimTypes.**prim_for_human**(*prim_type*, *val*)

pysisyphus.intcoords.PrimTypes.**prims_from_prim_inputs**(*prim_inps*)

pysisyphus.intcoords.Primitive module

class pysisyphus.intcoords.Primitive.**Primitive**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: object

calculate(*coords3d*, *indices=None*, *gradient=False*)

jacobian(*coords3d*, *indices=None*)

log(*msg*, *lvl=10*)

log_dbg(*msg*)

static parallel(*u*, *v*, *thresh=1e-06*)

static rho(*atoms*, *coords3d*, *indices*)

set_cross_vec(*coords3d*, *indices*)

weight(*atoms*, *coords3d*, *f_damping=0.12*)

pysisyphus.intcoords.RedundantCoords module

class pysisyphus.intcoords.RedundantCoords.**HybridRedundantCoords**(**args*, ***kwargs*)

Bases: *RedundantCoords*

class pysisyphus.intcoords.RedundantCoords.**RedundantCoords**(*atoms*, *coords3d*, *masses=None*, *bond_factor=1.3*, *typed_prims=None*, *define_prims=None*, *constrain_prims=None*, *freeze_atoms=None*, *freeze_atoms_exclude=False*, *internals_with_frozen=False*, *define_for=None*, *bonds_only=False*, *check_bends=True*, *rebuild=True*, *bend_min_deg=15*, *dihed_max_deg=175*, *lb_min_deg=175*, *weighted=False*, *min_weight=0.3*, *svd_inv_thresh=0.000316*, *recalc_B=False*, *tric=False*, *hybrid=False*, *hbond_angles=False*, *rm_for_frag=None*)

Bases: object

property B

Wilson B-Matrix

property B_inv

Generalized inverse of the Wilson B-Matrix.

property B_inv_prim

Generalized inverse of the primitive Wilson B-Matrix.

property B_prim
Wilson B-Matrix

property Bt_inv
Transposed generalized inverse of the Wilson B-Matrix.

property Bt_inv_prim
Transposed generalized inverse of the primitive Wilson B-Matrix.

property C
Diagonal matrix. Entries for constraints are set to one.

property P
Projection matrix onto B. See [1] Eq. (4).

backtransform_hessian(*redund_hessian*, *int_gradient=None*)
Transform Hessian in internal coordinates to Cartesians.

property bend_atom_indices

property bend_indices

property bond_atom_indices

property bond_indices

property bond_typed_prims

property cartesian_indices

clear()

property constrained_indices

property coords

property coords3d

property dihedral_atom_indices

property dihedral_indices

eval(*coords3d*, *attr=None*)

get_K_matrix(*int_gradient=None*)

get_index_of_typed_prim(*typed_prim*)
Index in self.typed_prims for the supplied typed_prim.

get_prim_internals_by_indices(*indices*)

inv_B(*B*)

inv_Bt(*B*)

property linear_bend_indices

log(*message*)

log_int_grad_msg(*int_gradient*)

property outofplane_indices

property prim_coords

property prim_indices_set

property prim_internals

property primitives

print_typed_prims()

project_hessian(*H*, *shift=1000*)

Expects a hessian in internal coordinates. See Eq. (11) in [1].

project_vector(*vector*)

Project supplied vector onto range of B.

return_inds(*slice_*)

property rotation_indices

set_inds_from_typed_prims(*typed_prims*)

set_primitive_indices(*atoms*, *coords3d*)

transform_forces(*cart_forces*)

Combination of Eq. (9) and (11) in [1].

transform_hessian(*cart_hessian*, *int_gradient=None*)

Transform Cartesian Hessian to internal coordinates.

transform_int_step(*int_step*, *update_constraints=False*, *pure=False*)

property translation_indices

property typed_prims

class pysisyphus.intcoords.RedundantCoords.**TMTRIC**(*atoms*, **args*, ***kwargs*)

Bases: *TRIC*

class pysisyphus.intcoords.RedundantCoords.**TRIC**(**args*, ***kwargs*)

Bases: *RedundantCoords*

pysisyphus.intcoords.Rotation module

class pysisyphus.intcoords.Rotation.**Rotation**(*indices*, **args*, *ref_coords3d*, ***kwargs*)

Bases: *Primitive*

See (II. Theory) in [1], Eq. (3) - (14)

index = None

class pysisyphus.intcoords.Rotation.**RotationA**(*indices*, **args*, *ref_coords3d*, ***kwargs*)

Bases: *Rotation*

index = 0

class pysisyphus.intcoords.Rotation.**RotationB**(*indices*, **args*, *ref_coords3d*, ***kwargs*)

Bases: *Rotation*

index = 1

class pysisyphus.intcoords.Rotation.**RotationC**(*indices*, **args*, *ref_coords3d*, ***kwargs*)

Bases: *Rotation*

index = 2

pysisyphus.intcoords.Rotation.**compare_to_geometric**(*c3d*, *ref_c3d*, *dR*, *dF*, *dqdx*, *dvdx*, *atol=1e-14*)

pysisyphus.intcoords.Stretch module

class pysisyphus.intcoords.Stretch.**Stretch**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Primitive*

pysisyphus.intcoords.Torsion module

class pysisyphus.intcoords.Torsion.**Torsion**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Primitive*

pysisyphus.intcoords.Torsion2 module

class pysisyphus.intcoords.Torsion2.**Torsion2**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Torsion*

pysisyphus.intcoords.Translation module

class pysisyphus.intcoords.Translation.**Translation**(**args*, ***kwargs*)

Bases: *Primitive*

See (II. Theory) in [1], Eq. (2)

class pysisyphus.intcoords.Translation.**TranslationX**(**args*, ***kwargs*)

Bases: *Translation*

cart_axis = 0

class pysisyphus.intcoords.Translation.**TranslationY**(**args*, ***kwargs*)

Bases: *Translation*

cart_axis = 1

class pysisyphus.intcoords.Translation.**TranslationZ**(**args*, ***kwargs*)

Bases: *Translation*

cart_axis = 2

pysisyphus.intcoords.augment_bonds module

pysisyphus.intcoords.augment_bonds.**augment_bonds**(*geom*, *root=0*, *proj=False*)

pysisyphus.intcoords.augment_bonds.**find_missing_bonds_by_projection**(*geom*, *hessian*, *bond_factor=2.0*, *bond_thresh=0.35*, *concerted_thresh=0.35*, *root=0*)

pysisyphus.intcoords.augment_bonds.**find_missing_strong_bonds**(*geom*, *hessian*, *bond_factor=1.7*, *thresh=0.3*, *root=0*)

pysisyphus.intcoords.derivatives module

pysisyphus.intcoords.derivatives.**d2q_a**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**d2q_a2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend2, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**d2q_b**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*)
Stretch, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**d2q_d**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**d2q_d2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion2, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**d2q_lb**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Bend, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**d2q_ld**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Displacement, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**d2q_oop**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*)
OutOfPlane, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**d2q_rd1**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion1, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**d2q_rd2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion2, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**dq_a**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend, first derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**dq_a2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend2, first derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**dq_b**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*)
Stretch, first derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**dq_d**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion, first derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**dq_d2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion2, first derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**dq_lb**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Bend, first derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**dq_ld**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Displacement, first derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**dq_oop**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*)
OutOfPlane, first derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**dq_rd1**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion1, first derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**dq_rd2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion2, first derivative wrt. Cartesians

pysisyphus.intcoords.derivatives.**q_a**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend

pysisyphus.intcoords.derivatives.**q_a2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend2

pysisyphus.intcoords.derivatives.**q_b**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*)
Stretch

pysisyphus.intcoords.derivatives.**q_d**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion

pysisyphus.intcoords.derivatives.**q_d2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion2

pysisyphus.intcoords.derivatives.**q_lb**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Bend

pysisyphus.intcoords.derivatives.**q_ld**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Displacement

pysisyphus.intcoords.derivatives.**q_oop**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*)
OutOfPlane

pysisyphus.intcoords.derivatives.**q_rd1**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion1

pysisyphus.intcoords.derivatives.**q_rd2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion2

pysisyphus.intcoords.eval module

class pysisyphus.intcoords.eval.**PrimInternal**(*inds*, *val*, *grad=None*)
Bases: object

pysisyphus.intcoords.eval.**augment_primitives**(*missing_prims*, *coords3d*, *prim_indices*, *fragments*)

pysisyphus.intcoords.eval.**check_primitives**(*coords3d*, *primitives*, *B=None*, *thresh=1e-06*, *logger=None*)

pysisyphus.intcoords.eval.**eval_B**(*coords3d*, *primitives*)

pysisyphus.intcoords.eval.**eval_primitives**(*coords3d*, *primitives*)

pysisyphus.intcoords.exceptions module

exception pysisyphus.intcoords.exceptions.**DifferentCoordLengthsException**

Bases: Exception

exception pysisyphus.intcoords.exceptions.**DifferentPrimitivesException**

Bases: Exception

exception pysisyphus.intcoords.exceptions.**NeedNewInternalsException**(*coords3d*, **args*, *invalid_inds=None*, *invalid_prims=None*, ***kwargs*)

Bases: Exception

exception pysisyphus.intcoords.exceptions.**PrimitiveNotDefinedException**(*typed_prim*, **args*, ***kwargs*)

Bases: Exception

exception pysisyphus.intcoords.exceptions.**RebuiltInternalsException**(**args*, *typed_prims=None*, ***kwargs*)

Bases: Exception

pysisyphus.intcoords.findiffs module

pysisyphus.intcoords.findiffs.**fin_diff_B**(*primitive*, *coords3d*, *delta=1e-06*)

Derivatives of a primitive internal gradient wrt its defining cartesian coordinates.

pysisyphus.intcoords.findiffs.**fin_diff_prim**(*primitive*, *coords3d*, *delta=1e-06*)

Derivatives of a primitive internal gradient wrt its defining cartesian coordinates.

pysisyphus.intcoords.generate_derivatives module

class pysisyphus.intcoords.generate_derivatives.**FuncResult**(*d0*, *d1*, *d2*, *f0*, *f1*, *f2*)

Bases: tuple

d0

Alias for field number 0

d1

Alias for field number 1

d2

Alias for field number 2

f0

Alias for field number 3

f1

Alias for field number 4

f2

Alias for field number 5

pysisyphus.intcoords.generate_derivatives.**generate_wilson**(*generate=None*, *out_fn='derivatives.py'*, *use_mpmath=False*)

pysisyphus.intcoords.generate_derivatives.**make_deriv_funcs**(*base_expr*, *dx*, *args*, *names*, *comment*, *use_mpmath=True*)

pysisyphus.intcoords.generate_derivatives.**make_py_func**(*exprs*, *args=None*, *name=None*, *comment=''*, *use_mpmath=False*)

pysisyphus.intcoords.helpers module

pysisyphus.intcoords.helpers.**form_coordinate_union**(*geom1*, *geom2*)

pysisyphus.intcoords.helpers.**get_bond_difference**(*geom1*, *geom2*, *bond_factor=1.3*)

Return formed and broken bonds when going from geom1 to geom2.

pysisyphus.intcoords.helpers.**get_bond_differences_verbose**(*geom1*, *geom2*, *bond_factor=1.3*, *key1='geom1'*, *key2='geom2'*)

pysisyphus.intcoords.helpers.**get_step**(*geom*, *coords*)

pysisyphus.intcoords.helpers.**get_tangent**(*prims1*, *prims2*, *dihedral_inds*, *normalize=False*)

Normalized tangent between primitive internal coordinates.

Tangent pointing from prims1 to prims2 in primitive internal coordinates, taking into account the periodicity of dihedral angles.

Parameters

- **prims1** (*np.array*) -- 1d-array of primitive internal coordinates in the order (stretches, bends, dihedrals).
- **prims2** (*np.array*) -- See prims1.
- **dihedral_inds** (*list of int*) -- Dihedral indices in prims1 and prims2.

Returns

tangent -- 1d array containing the normalized tangent pointing from prims1 to prims2.

Return type

np.array

pysisyphus.intcoords.helpers.**get_weighted_bond_mode**(*weighted_bonds*, *coords3d*, *remove_translation=True*)

pysisyphus.intcoords.helpers.**get_weighted_bond_mode_getter**(*target_weighted_bonds*, *bond_factor=1.2*, *fractional=False*)

Create input for intcoords.helpers.get_weighted_bond_mode.

Compared to the rest of pysisyphus this method uses a slightly lowered bond factor, so it is more strict regarding what is considered a bond and what not.

pysisyphus.intcoords.helpers.**interfragment_distance**(*frag1*, *frag2*, *coords3d*)

Return type
float

pysisyphus.intcoords.helpers.**merge_coordinate_definitions**(*geom1*, *geom2*)

pysisyphus.intcoords.helpers.**verbose_bond_difference**(*formed*, *broken*, *key1*, *key2*, *atoms=None*)

pysisyphus.intcoords.mp_derivatives module

pysisyphus.intcoords.mp_derivatives.**d2q_a**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**d2q_a2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend2, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**d2q_b**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*)
Stretch, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**d2q_d**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**d2q_d2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion2, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**d2q_lb**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Bend, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**d2q_ld**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Displacement, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**d2q_oop**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*)
OutOfPlane, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**d2q_rd1**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion1, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**d2q_rd2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion2, 2nd derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**dq_a**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend, first derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**dq_a2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend2, first derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**dq_b**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*)
Stretch, first derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**dq_d**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion, first derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**dq_d2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion2, first derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**dq_lb**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Bend, first derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**dq_ld**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Displacement, first derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**dq_oop**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*)
OutOfPlane, first derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**dq_rd1**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion1, first derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**dq_rd2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion2, first derivative wrt. Cartesians

pysisyphus.intcoords.mp_derivatives.**q_a**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend

pysisyphus.intcoords.mp_derivatives.**q_a2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*)
Bend2

pysisyphus.intcoords.mp_derivatives.**q_b**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*)
Stretch

pysisyphus.intcoords.mp_derivatives.**q_d**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion

pysisyphus.intcoords.mp_derivatives.**q_d2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
Torsion2

pysisyphus.intcoords.mp_derivatives.**q_lb**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Bend

pysisyphus.intcoords.mp_derivatives.**q_ld**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *n0*, *n1*, *n2*, *p0*, *p1*, *p2*)
Linear Displacement

pysisyphus.intcoords.mp_derivatives.**q_oop**(*m0*, *m1*, *m2*, *n0*, *n1*, *n2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*)
OutOfPlane

pysisyphus.intcoords.mp_derivatives.**q_rd1**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion1

pysisyphus.intcoords.mp_derivatives.**q_rd2**(*m0*, *m1*, *m2*, *o0*, *o1*, *o2*, *p0*, *p1*, *p2*, *n0*, *n1*, *n2*)
RobustTorsion2

pysisyphus.intcoords.setup module

class pysisyphus.intcoords.setup.**CoordInfo**(*bonds*, *hydrogen_bonds*, *interfrag_bonds*, *aux_interfrag_bonds*, *bends*, *linear_bends*, *linear_bend_complements*, *proper_dihedrals*, *improper_dihedrals*, *translation_inds*, *rotation_inds*, *cartesian_inds*, *typed_prims*, *fragments*)

Bases: tuple

aux_interfrag_bonds
Alias for field number 3

bends
Alias for field number 4

bonds
Alias for field number 0

cartesian_inds
Alias for field number 11

fragments
Alias for field number 13

hydrogen_bonds
Alias for field number 1

improper_dihedrals
Alias for field number 8

interfrag_bonds
Alias for field number 2

linear_bend_complements
Alias for field number 6

linear_bends
Alias for field number 5

proper_dihedrals
Alias for field number 7

rotation_inds
Alias for field number 10

translation_inds
Alias for field number 9

typed_prims
Alias for field number 12

pysisyphus.intcoords.setup.**connect_fragments**(*cdm*, *fragments*, *max_aux=3.78*, *aux_factor=1.3*, *logger=None*)

Determine the smallest interfragment bond for a list of fragments and a condensed distance matrix. For more than a few fragments this function performs poorly, as each fragment is connected to all reamaining fragments, leading to an explosion of bonds, bends and dihedrals.

pysisyphus.intcoords.setup.**connect_fragments_ahlrichs**(*cdm*, *fragments*, *atoms*, *min_dist_scale=1.1*, *scale=1.2*, *avoid_h=False*, *logger=None*)

pysisyphus.intcoords.setup.**connect_fragments_kmeans**(*cdm*, *fragments*, *atoms*, *aux_below_thresh=3.7807*, *aux_add_dist=2.8356*, *aux_keep=5*, *aux_no_hh=True*, *min_dist_thresh=5.0*, *min_scale=1.2*, *logger=None*)

Generate (auxiliary) interfragment bonds.

In the a first step, minimum distance interfragment bonds (IFBs) are determined between all possible fragment pairs. Similarly, possible auxiliary IFBs are determined. Candidates for auxiliary IFBs are:

IFB <= aux_below_thresh, default 2 Å IFB <= (minimum distance IFB + aux_add_dist), default 1.5 Å

By default, only the first aux_keep (default = 5) auxiliary IFBs are kept.

Connecting all fragments can lead to bonds between very distant atoms. If more than two fragments are present we cluster the minimum distance IFB distances using KMeans, to determine a reasonable length for valid IFBs. We start out with two clusters and increase the number of cluster until the center of one cluster is around the scaled global minimum distance between the fragments. The center of this cluster is then used as a cutoff vor valid IFBs.

After pruning all possible IFBs we can determine the fragment pairs, that are actually connected. This information is then used to also prune possible interfragment bonds. Only auxiliary IFBs between fragments that are actually connected via IFBs are kept.

pysisyphus.intcoords.setup.**get_bend_inds**(*coords3d*, *bond_inds*, *min_deg*, *max_deg*, *logger=None*)

pysisyphus.intcoords.setup.**get_bond_mat**(*geom*, *bond_factor=1.3*)

pysisyphus.intcoords.setup.**get_bond_sets**(*atoms*, *coords3d*, *bond_factor=1.3*, *return_cdm=False*, *return_cbm=False*)

I'm sorry, but this function does not return sets, but an int ndarray.

pysisyphus.intcoords.setup.**get_dihedral_inds**(*coords3d*, *bond_inds*, *bend_inds*, *max_deg*, *logger=None*)

pysisyphus.intcoords.setup.**get_fragments**(*atoms*, *coords*, *bond_inds=None*, *bond_factor=1.3*)

This misses unconnected single atoms!

pysisyphus.intcoords.setup.**get_hydrogen_bond_inds**(*atoms*, *coords3d*, *bond_inds*, *logger=None*)

pysisyphus.intcoords.setup.**get_hydrogen_bond_inds_v2**(*atoms*, *coords3d*, *bond_inds*, *logger=None*)

pysisyphus.intcoords.setup.**get_linear_bend_inds**(*coords3d*, *cbm*, *bends*, *min_deg=175*, *max_bonds=4*, *logger=None*)

pysisyphus.intcoords.setup.**get_pair_covalent_radii**(*atoms*)

pysisyphus.intcoords.setup.**get_primitives**(*coords3d*, *typed_prims*, *logger=None*)

pysisyphus.intcoords.setup.**setup_redundant**(*atoms*, *coords3d*, *factor=1.3*, *define_prims=None*, *min_deg=15*, *dihed_max_deg=175*, *lb_min_deg=None*, *lb_max_bonds=4*, *min_weight=None*, *tric=False*, *hybrid=False*, *interfrag_hbonds=True*, *hbond_angles=False*, *freeze_atoms=None*, *define_for=None*, *internals_with_frozen=False*, *rm_for_frag=None*, *logger=None*)

pysisyphus.intcoords.setup.**setup_redundant_from_geom**(*geom*, **args*, ***kwargs*)

pysisyphus.intcoords.setup.**sort_by_prim_type**(*to_sort=None*)

pysisyphus.intcoords.setup_fast module

pysisyphus.intcoords.setup_fast.**find_bends**(*coords3d*, *bonds*, *min_deg*, *max_deg*, *logger=None*)

pysisyphus.intcoords.setup_fast.**find_bonds**(*atoms*, *coords3d*, *covalent_radii=None*, *bond_factor=1.3*, *min_dist=0.1*)

pysisyphus.intcoords.setup_fast.**find_bonds_bends**(*geom*, *bond_factor=1.3*, *min_deg=15*, *max_deg=175*)

pysisyphus.intcoords.setup_fast.**find_bonds_for_geom**(*geom*, *bond_factor=1.3*)

pysisyphus.intcoords.setup_fast.**find_dihedrals**(*coords3d*, *bonds*, *bends*, *max_deg*, *logger=None*)

pysisyphus.intcoords.setup_fast.**get_bend_candidates**(*bonds*)

Also yields duplicates [a, b, c] and [c, b, a].

pysisyphus.intcoords.setup_fast.**get_bond_vec_getter**(*atoms*, *covalent_radii*, *bonds_for_inds*, *no_bonds_with=None*, *bond_factor=1.3*)

pysisyphus.intcoords.setup_fast.**get_max_bond_dists**(*atoms*, *bond_factor*, *covalent_radii=None*)

pysisyphus.intcoords.update module

pysisyphus.intcoords.update.**correct_dihedrals**(*new_dihedrals*, *old_dihedrals*)

Dihedrals are periodic. Going from -179° to 179° is not a step of 358°, but a step of 2°. By considering the actual distance of the dihedrals from the correct step can be calculated.

dihedral step length = abs(abs(new_dihedral) -) + abs(abs(old_dihedral) -)

or put differently

dihedral step length = abs(abs(new_dihedral - old_dihedral) - 2*)

The sign is left to be determined. Going from -179° to 179° (roughly - - = 2) is a counter clockwise rotation and the dihedral has to decrease below -. Going from 179° to -179° (roughly - - = -2) is a clockwise rotation and the dihedral increases abvove . So the correct sign corresponds to the negative sign of the original difference.

original difference 2 -> dihedral must decrease -> sign = -1 original difference -2 -> dihedral must increase -> sign = +1

Overall, the old dihedral is modified by the actual step length with the correct sign.

pysisyphus.intcoords.update.**inds_from_prim_types**(*typed_prims*, *prim_types*)

pysisyphus.intcoords.update.**transform_int_step**(*int_step*, *old_cart_coords*, *cur_internals*, *Bt_inv_prim*, *primitives*, *typed_prims=None*, *dihedral_inds=None*, *rotation_inds=None*, *bend_inds=None*, *check_dihedrals=False*, *check_bends=False*, *bend_min_deg=15*, *bend_max_deg=175*, *freeze_atoms=None*, *constrained_inds=None*, *update_constraints=False*, *cart_rms_thresh=1e-06*, *Bt_inv_prim_getter=None*, *max_cycles=25*, *logger=None*)

Transformation is done in primitive internals, so int_step must be given in primitive internals and not in DLC!

pysisyphus.intcoords.update.**update_internals**(*new_coords3d*, *old_internals*, *primitives*, *dihedral_inds*, *rotation_inds*, *bend_inds*, *check_dihedrals=False*, *check_bends=False*, *bend_min_deg=15*, *bend_max_deg=175*, *rotation_thresh=0.9*, *logger=None*)

pysisyphus.intcoords.valid module

pysisyphus.intcoords.valid.**are_collinear**(*vec1*, *vec2*, *deg_thresh=179.5*)

pysisyphus.intcoords.valid.**bend_valid**(*coords3d*, *indices*, *min_deg*, *max_deg*)

pysisyphus.intcoords.valid.**check_typed_prims**(*coords3d*, *typed_prims*, *bend_min_deg*, *dihed_max_deg*, *lb_min_deg*, *logger=None*, *check_bends=True*)

pysisyphus.intcoords.valid.**dihedral_valid**(*coords3d*, *inds*, *deg_thresh=177.5*)

pysisyphus.intcoords.valid.**dihedrals_are_valid**(*coords3d*, *dihedral_inds*, *logger=None*)

Module contents

class pysisyphus.intcoords.**Bend**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Primitive*

class pysisyphus.intcoords.**Bend2**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Bend*

class pysisyphus.intcoords.**CartesianCoords**(*atoms*, *coords3d*, *masses*, *freeze_atoms=None*, ***, *mass_weighted=False*, ***kwargs*)

Bases: *CoordSys*

property coords: ndarray[Any, dtype[_ScalarType_co]]

Getter for coordinates in this coordinate system.

property coords3d: ndarray[Any, dtype[_ScalarType_co]]

Getter for 3d Cartesian coordinates.

property inv_masses_rep_sqrt: ndarray[Any, dtype[_ScalarType_co]]

property masses: ndarray[Any, dtype[_ScalarType_co]]

property masses_sqrt: ndarray[Any, dtype[_ScalarType_co]]

project_hessian(*hessian*)

Project Hessian in the current coordinate system.

transform_forces(*cart_forces*)

Transform Cartesian forces to this coordinate system.

Return type
ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]

transform_hessian(*cart_hessian*, *int_gradient=None*)

Transform Cartesian Hessian to this coordinate system.

transform_int_step(*step*, *update_constraints=False*, *pure=False*)

Transform step in this coordinate system to Cartesians.

Return type
ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]

property typed_prims: List

List of (primitive) internal coordinates.

May be empty, e.g., when the coordinate system is Cartesian.

class pysisyphus.intcoords.CartesianX(**args*, ***kwargs*)

Bases: *Cartesian*

cart_axis = 0

class pysisyphus.intcoords.CartesianY(**args*, ***kwargs*)

Bases: *Cartesian*

cart_axis = 1

class pysisyphus.intcoords.CartesianZ(**args*, ***kwargs*)

Bases: *Cartesian*

cart_axis = 2

class pysisyphus.intcoords.DLC(**args*, *full_set=True*, ***kwargs*)

Bases: *RedundantCoords*

property B

Wilson B-Matrix in the non-redundant subspace.

property U

backtransform_hessian(**args*, ***kwargs*)

Transform Hessian in internal coordinates to Cartesians.

property constraints

property coords

freeze_primitives(*typed_prims*)

Freeze primitive internal coordinates.

Parameters
typed_prims (*iterable of typed primitives*) -- Iterable containing typed_primitives, starting with a PrimType and followed by atom indices.

get_active_set(*B*, *inv_thresh=None*)

See [5] between Eq. (7) and Eq. (8) for advice regarding the threshold.

get_constrained_U(*constraint_vecs*)

project_hessian(*H*)

As we already work in the non-redundant subspace we don't have to project/shift the hessian as we do it for simple redundant internal coordinates.

project_primitive_on_active_set(*prim_ind*)

reset_constraints()

set_active_set()

transform_hessian(*cart_hessian*, *int_gradient=None*)

Transform Cartesian Hessian to DLC.

transform_int_step(*step*, **args*, ***kwargs*)

As the transformation is done in primitive internal coordinates we convert the DLC back to primitive coordinates.

class pysisyphus.intcoords.**DistanceFunction**(*indices*, *coeff=-1*, ***kwargs*)

Bases: *Primitive*

class pysisyphus.intcoords.**DummyImproper**(*indices*, **args*, *fix_inner=True*, ***kwargs*)

Bases: *Torsion*

static get_coords3d_and_indices_ext(*coords3d*, *indices*)

class pysisyphus.intcoords.**DummyTorsion**(*indices*, **args*, *fix_inner=True*, ***kwargs*)

Bases: *Torsion*

static get_coords3d_and_indices_ext(*coords3d*, *indices*)

static get_fourth_coords(*coords3d*, *indices*, *r=1.889*, *theta=90*)

M N <- add

/

u v

/

P

m, o, p, n = indices

class pysisyphus.intcoords.**HDLC**(**args*, ***kwargs*)

Bases: *DLC*

class pysisyphus.intcoords.**HybridRedundantCoords**(**args*, ***kwargs*)

Bases: *RedundantCoords*

class pysisyphus.intcoords.**LinearBend**(**args*, *complement=False*, ***kwargs*)

Bases: *Primitive*

calculate(*coords3d*, *indices=None*, *gradient=False*)

jacobian(*coords3d*, *indices=None*)

class pysisyphus.intcoords.**LinearDisplacement**(**args*, *complement=False*, ***kwargs*)

Bases: *Primitive*

calculate(*coords3d*, *indices=None*, *gradient=False*)

jacobian(*coords3d*, *indices=None*)

class pysisyphus.intcoords.**MWCartesianCoords**(**args*, ***kwargs*)

Bases: *CartesianCoords*

class pysisyphus.intcoords.**OutOfPlane**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Primitive*

[1] https://doi.org/10.1002/(SICI)1096-987X(19990730)20:10<1067::AID-JCC9>3.0.CO;2-V
Lee, 1999

exception pysisyphus.intcoords.**PrimitiveNotDefinedException**(*typed_prim*, **args*, ***kwargs*)

Bases: Exception

class pysisyphus.intcoords.**RedundantCoords**(*atoms*, *coords3d*, *masses=None*, *bond_factor=1.3*, *typed_prims=None*, *define_prims=None*, *constrain_prims=None*, *freeze_atoms=None*, *freeze_atoms_exclude=False*, *internals_with_frozen=False*, *define_for=None*, *bonds_only=False*, *check_bends=True*, *rebuild=True*, *bend_min_deg=15*, *dihed_max_deg=175*, *lb_min_deg=175*, *weighted=False*, *min_weight=0.3*, *svd_inv_thresh=0.000316*, *recalc_B=False*, *tric=False*, *hybrid=False*, *hbond_angles=False*, *rm_for_frag=None*)

Bases: object

property B

Wilson B-Matrix

property B_inv

Generalized inverse of the Wilson B-Matrix.

property B_inv_prim

Generalized inverse of the primitive Wilson B-Matrix.

property B_prim

Wilson B-Matrix

property Bt_inv

Transposed generalized inverse of the Wilson B-Matrix.

property Bt_inv_prim

Transposed generalized inverse of the primitive Wilson B-Matrix.

property C

Diagonal matrix. Entries for constraints are set to one.

property P

Projection matrix onto B. See [1] Eq. (4).

backtransform_hessian(*redund_hessian*, *int_gradient=None*)

Transform Hessian in internal coordinates to Cartesians.

property bend_atom_indices

property bend_indices

property bond_atom_indices

property bond_indices

property bond_typed_prims

property cartesian_indices

clear()

property constrained_indices

property coords

property coords3d

property dihedral_atom_indices

property dihedral_indices

eval(*coords3d*, *attr=None*)

get_K_matrix(*int_gradient=None*)

get_index_of_typed_prim(*typed_prim*)

Index in self.typed_prims for the supplied typed_prim.

get_prim_internals_by_indices(*indices*)

inv_B(*B*)

inv_Bt(*B*)

property linear_bend_indices

log(*message*)

log_int_grad_msg(*int_gradient*)

property outofplane_indices

property prim_coords

property prim_indices_set

property prim_internals

property primitives

print_typed_prims()

project_hessian(*H*, *shift=1000*)

Expects a hessian in internal coordinates. See Eq. (11) in [1].

project_vector(*vector*)

Project supplied vector onto range of B.

return_inds(*slice_*)

property rotation_indices

set_inds_from_typed_prims(*typed_prims*)

set_primitive_indices(*atoms*, *coords3d*)

transform_forces(*cart_forces*)

Combination of Eq. (9) and (11) in [1].

transform_hessian(*cart_hessian*, *int_gradient=None*)

Transform Cartesian Hessian to internal coordinates.

transform_int_step(*int_step*, *update_constraints=False*, *pure=False*)

property translation_indices

property typed_prims

class pysisyphus.intcoords.**RobustTorsion1**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Primitive*

class pysisyphus.intcoords.**RobustTorsion2**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Primitive*

class pysisyphus.intcoords.**RotationA**(*indices*, **args*, *ref_coords3d*, ***kwargs*)

Bases: *Rotation*

index = 0

class pysisyphus.intcoords.**RotationB**(*indices*, **args*, *ref_coords3d*, ***kwargs*)

Bases: *Rotation*

index = 1

class pysisyphus.intcoords.**RotationC**(*indices*, **args*, *ref_coords3d*, ***kwargs*)

Bases: *Rotation*

index = 2

class pysisyphus.intcoords.**Stretch**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Primitive*

class pysisyphus.intcoords.**TMTRIC**(*atoms*, **args*, ***kwargs*)

Bases: *TRIC*

class pysisyphus.intcoords.**TRIC**(**args*, ***kwargs*)

Bases: *RedundantCoords*

class pysisyphus.intcoords.**Torsion**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Primitive*

class pysisyphus.intcoords.**Torsion2**(*indices*, *periodic=False*, *calc_kwargs=None*, *cache=False*)

Bases: *Torsion*

class pysisyphus.intcoords.**TranslationX**(**args*, ***kwargs*)

Bases: *Translation*

cart_axis = 0

class pysisyphus.intcoords.**TranslationY**(**args*, ***kwargs*)

Bases: *Translation*

cart_axis = 1

class pysisyphus.intcoords.**TranslationZ**(**args*, ***kwargs*)

Bases: *Translation*

cart_axis = 2

17.1.1.8 pysisyphus.interpolate package

Submodules

pysisyphus.interpolate.IDPP module

class pysisyphus.interpolate.IDPP.**IDPP**(*geoms*, *between*, *extrapolate=0*, *extrapolate_before=0*, *extrapolate_after=0*, *extrapolate_damp=1.0*, *align=False*)

Bases: *Interpolator*

interpolate(*initial_geom*, *final_geom*, ***kwargs*)

pysisyphus.interpolate.Interpolator module

class pysisyphus.interpolate.Interpolator.**Interpolator**(*geoms*, *between*, *extrapolate=0*, *extrapolate_before=0*, *extrapolate_after=0*, *extrapolate_damp=1.0*, *align=False*)

Bases: object

all_geoms_to_trj(*trj_fn*)

interpolate(*initial_geom*, *final_geom*, *interpolate_only=0*, *extrapolate=False*)

interpolate_all()

pysisyphus.interpolate.LST module

class pysisyphus.interpolate.LST.**LST**(**args*, *align=True*, *gtol=0.0001*, *silent=False*, ***kwargs*)

Bases: *Interpolator*

cost_function(*wa_c*, *f*, *rab*, *wab*)

interpolate(*initial_geom*, *final_geom*, ***kwargs*)

pysisyphus.interpolate.Redund module

class pysisyphus.interpolate.Redund.**Redund**(**args*, *align=True*, *rebuild_geoms=True*, ***kwargs*)

Bases: *Interpolator*

dump_progress(*geoms*, *out_fn='redund_interpol_fail.trj'*)

interpolate(*initial_geom*, *final_geom*, *interpolate_only=0*, *extrapolate=False*, *typed_prims=None*)

restart_interpolate(*geoms*, *new_geom*, *interpolate_kwargs*)

step_along_tangent(*geom*, *prim_tangent*, *step_size*)

pysisyphus.interpolate.helpers module

pysisyphus.interpolate.helpers.**interpolate**(*initial_geom*, *final_geom*, *between*, *kind='linear'*, *only_between=False*, *align=False*, *interpol_kwargs=None*)

pysisyphus.interpolate.helpers.**interpolate_all**(*geoms*, *between*, *kind='linear'*, *align=False*, ***interpol_kwargs*)

Module contents

17.1.1.9 pysisyphus.io package

Submodules

pysisyphus.io.cjson module

pysisyphus.io.cjson.**geom_from_cjson**(*fn*, ***kwargs*)

pysisyphus.io.cjson.**parse_cjson**(*fn*)

pysisyphus.io.crd module

pysisyphus.io.crd.**a8**(*str_*)

pysisyphus.io.crd.**atoms_coords_to_crd_str**(*atoms*, *coords*, *resno=1*, *res='UNL1'*, *segid=None*, *resid=1*, *ref_atoms=None*, *del_atoms=None*)

pysisyphus.io.crd.**f20**(*float_*)

pysisyphus.io.crd.**geom_from_crd**(*fn*, ***kwargs*)

pysisyphus.io.crd.**geom_to_crd_str**(*geom*, ***kwargs*)

pysisyphus.io.crd.**i10**(*int_*)

pysisyphus.io.hdf5 module

pysisyphus.io.hdf5.**get_h5_group**(*fn*, *group_name*, *data_model=None*, *reset=False*)
Return (and create if neccesary) group with given name and data model.

pysisyphus.io.hdf5.**init_h5_group**(*f*, *group_name*, *data_model*)
Create group with given name and data model.

pysisyphus.io.hdf5.**resize_h5_group**(*group*, *max_cycles*)
Increase size of first dimension of datasets in the given group.

pysisyphus.io.hessian module

pysisyphus.io.hessian.**geom_from_hessian**(*h5_fn*, ***geom_kwargs*)

pysisyphus.io.hessian.**save_hessian**(*h5_fn*, *geom*, *cart_hessian=None*, *energy=None*, *mult=None*)

pysisyphus.io.hessian.**save_third_deriv**(*h5_fn*, *geom*, *third_deriv_result*, *H_mw*, *H_proj*)

pysisyphus.io.mol2 module

pysisyphus.io.mol2.**dict_to_mol2_string**(*as_dict*)

pysisyphus.io.mol2.**geom_from_mol2**(*text*, ***kwargs*)

pysisyphus.io.mol2.**render_xyz**(*xyz*)

pysisyphus.io.molden module

pysisyphus.io.molden.**geoms_from_molden**(*fn*, ***kwargs*)

pysisyphus.io.molden.**parse_mo**(*mo_lines*)

pysisyphus.io.molden.**parse_molden_atoms**(*data*)

pysisyphus.io.pdb module

pysisyphus.io.pdb.**atoms_coords_to_pdb_str**(*atoms*, *coords*, *fragments=None*, *resname=''*, *conect=True*)

pysisyphus.io.pdb.**geom_from_pdb**(*fn*, ***kwargs*)

pysisyphus.io.pdb.**geom_to_pdb_str**(*geom*, *detect_fragments=False*, ***kwargs*)

pysisyphus.io.pdb.**get_conect_lines**(*atoms*, *coords*)

pysisyphus.io.pdb.**get_parser**(*widths*)

pysisyphus.io.pdb.**parse_atom_name**(*name*)

pysisyphus.io.pubchem module

pysisyphus.io.pubchem.**cid_from_name**(*name*)

pysisyphus.io.pubchem.**geom_from_pubchem_name**(*name*, ***kwargs*)

pysisyphus.io.pubchem.**sdf_from_cid**(*cid*)

pysisyphus.io.sdf module

pysisyphus.io.sdf.**geom_from_sdf**(*text*, ***kwargs*)

pysisyphus.io.sdf.**parse_coord_line**(*line*)

pysisyphus.io.sdf.**parse_sdf**(*text*)

pysisyphus.io.xyz module

pysisyphus.io.xyz.**geom_from_xyz**(*fn*, ***kwargs*)

pysisyphus.io.xyz.**geoms_from_inline_xyz**(*inline_xyz*, ***kwargs*)

pysisyphus.io.xyz.**geoms_from_xyz**(*fn*, ***kwargs*)

pysisyphus.io.zmat module

class pysisyphus.io.zmat.**ZLine**(*atom*, *rind*, *r*, *aind*, *a*, *dind*, *d*)

Bases: tuple

a

Alias for field number 4

aind

Alias for field number 3

atom

Alias for field number 0

d

Alias for field number 6

dind

Alias for field number 5

r

Alias for field number 2

rind

Alias for field number 1

pysisyphus.io.zmat.**geom_from_zmat**(*zmat*, *atoms=None*, *coords3d=None*, *geom=None*, *start_at=None*, *drop_dummy=True*, ***geom_kwargs*)

Adapted from https://github.com/robashaw/geomConvert by Robert Shaw.

pysisyphus.io.zmat.**geom_from_zmat_fn**(*fn*, ***geom_kwargs*)

pysisyphus.io.zmat.**geom_from_zmat_str**(*text*, *coord_type='cart'*, *coord_kwargs=None*)

pysisyphus.io.zmat.**zmat_from_fn**(*fn*)

pysisyphus.io.zmat.**zmat_from_str**(*text*)

Module contents

pysisyphus.io.**geom_from_cjson**(*fn*, ***kwargs*)

pysisyphus.io.**geom_from_crd**(*fn*, ***kwargs*)

pysisyphus.io.**geom_from_cube**(*inp*, **args*, ***kwargs*)

pysisyphus.io.**geom_from_fchk**(*inp*, **args*, ***kwargs*)

pysisyphus.io.**geom_from_hessian**(*h5_fn*, ***geom_kwargs*)

pysisyphus.io.**geom_from_mol2**(*text*, ***kwargs*)

pysisyphus.io.**geom_from_pdb**(*fn*, ***kwargs*)

pysisyphus.io.**geom_from_pubchem_name**(*name*, ***kwargs*)

pysisyphus.io.**geom_from_qcschema**(*qcschema*, ***geom_kwargs*)

pysisyphus.io.**geom_from_zmat**(*zmat*, *atoms=None*, *coords3d=None*, *geom=None*, *start_at=None*, *drop_dummy=True*, ***geom_kwargs*)

Adapted from https://github.com/robashaw/geomConvert by Robert Shaw.

pysisyphus.io.**geoms_from_molden**(*fn*, ***kwargs*)

pysisyphus.io.**geoms_from_xyz**(*fn*, ***kwargs*)

pysisyphus.io.**save_hessian**(*h5_fn*, *geom*, *cart_hessian=None*, *energy=None*, *mult=None*)

pysisyphus.io.**save_third_deriv**(*h5_fn*, *geom*, *third_deriv_result*, *H_mw*, *H_proj*)

17.1.1.10 pysisyphus.irc package

Submodules

pysisyphus.irc.DWI module

class pysisyphus.irc.DWI.**DWI**(*n=4*, *maxlen=2*)

Bases: object

__init__(*n=4*, *maxlen=2*)

Distance weighted interpolation.

dump(*fn*)

static from_h5(*fn*)

interpolate(*at_coords*, *gradient=False*)

See [1] Eq. (25) - (29)

update(*coords*, *energy*, *gradient*, *hessian*)

pysisyphus.irc.DWI.**taylor**(*energy*, *gradient*, *hessian*, *step*)

Taylor series expansion of the energy to second order.

pysisyphus.irc.DWI.**taylor_grad**(*gradient*, *hessian*, *step*)

Gradient of a Taylor series expansion of the energy to second order.

pysisyphus.irc.DampedVelocityVerlet module

class pysisyphus.irc.DampedVelocityVerlet.**DampedVelocityVerlet**(*geometry*, *v0=0.04*, *dt0=0.5*, *error_tol=0.003*, *max_cycles=150*, ***kwargs*)

Bases: *IRC*

damp_velocity(*velocity*)

estimate_error(*new_mw_coords*)

mw_grad_to_acc(*mw_grad*)

Takes care of the units for the mass-weighted gradient. Converts units of a mass-weighted gradient [Hartree/(Bohr*amu)] to units of acceleration [sqrt(amu)*Bohr/fs^2]. The 1e30 comes from converting second^2 to femto second^2.

prepare(*direction*)

step()

pysisyphus.irc.Euler module

class pysisyphus.irc.Euler.**Euler**(*geometry*, *step_length=0.01*, ***kwargs*)

Bases: *IRC*

step()

pysisyphus.irc.EulerPC module

class pysisyphus.irc.EulerPC.**EulerPC**(**args*, *hessian_recalc=None*, *hessian_update='bofill'*, *hessian_init='calc'*, *max_pred_steps=500*, *loose_cycles=3*, *dump_dwi=False*, *scipy_method=None*, *corr_func='mbs'*, ***kwargs*)

Bases: *IRC*

corrector_step(*init_mw_coords*, *step_length*, *dwi*)

get_integration_length_func(*init_mw_coords*)

prepare(**args*, ***kwargs*)

scipy_corrector_step(*init_mw_coords*, *step_length*, *dwi*)

Solve IRC equation dx/ds = -g/|g| on DWI PES in mass-weighted coordinates. Integration done until self.step_length in unweighted coordinates is achieved.

step()

pysisyphus.irc.GonzalezSchlegel module

class pysisyphus.irc.GonzalezSchlegel.**GonzalezSchlegel**(*geometry*, *max_micro_cycles=20*, *micro_step_thresh=0.001*, *hessian_recalc=None*, *line_search=False*, ***kwargs*)

Bases: *IRC*

micro_step(*counter*)

Constrained optimization on a hypersphere.

perp_component(*vec*, *perp_to*)

postprocess()

step()

pysisyphus.irc.IMKMod module

class pysisyphus.irc.IMKMod.**IMKMod**(*geometry*, *corr_first=True*, *corr_first_size=0.5*, *corr_first_energy=True*, *corr_bisec_size=0.0025*, *corr_bisec_energy=True*, ***kwargs*)

Bases: *IRC*

fit_grad_and_energies(*energy0*, *grad0*, *energy1*, *direction*)

fit_parabola(*x*, *y*)

step()

pysisyphus.irc.IRC module

class pysisyphus.irc.IRC.**IRC**(*geometry*, *step_length=0.1*, *max_cycles=125*, *downhill=False*, *forward=True*, *backward=True*, *root=0*, *hessian_init=None*, *displ='energy'*, *displ_energy=0.001*, *displ_length=0.1*, *displ_third_h5=None*, *rms_grad_thresh=0.001*, *hard_rms_grad_thresh=None*, *energy_thresh=1e-06*, *imag_below=0.0*, *force_inflection=True*, *check_bonds=True*, *out_dir='.'*, *prefix=''*, *dump_fn='irc_data.h5'*, *dump_every=5*)

Bases: `object`

__init__(*geometry*, *step_length=0.1*, *max_cycles=125*, *downhill=False*, *forward=True*, *backward=True*, *root=0*, *hessian_init=None*, *displ='energy'*, *displ_energy=0.001*, *displ_length=0.1*, *displ_third_h5=None*, *rms_grad_thresh=0.001*, *hard_rms_grad_thresh=None*, *energy_thresh=1e-06*, *imag_below=0.0*, *force_inflection=True*, *check_bonds=True*, *out_dir='.'*, *prefix=''*, *dump_fn='irc_data.h5'*, *dump_every=5*)

Base class for IRC calculations.

Parameters

- **geometry** (*Geometry*) -- Transtion state geometry, or initial geometry for downhill run.
- **step_length** (*float, optional*) -- Step length in unweighted coordinates.
- **max_cycles** (*int, optional*) -- Positive integer, controlloing the maximum number of IRC steps taken in a direction (forward/backward/downhill).

- **downhill** (*bool, default=False*) -- Downhill run from a non-stationary point with non-vanishing gradient. Disables forward and backward runs.
- **forward** (*bool, default=True*) -- Integrate IRC in positive s direction.
- **backward** (*bool, default=True*) -- Integrate IRC in negative s direction.
- **root** (*int, default=0*) -- Use n-th root for initial displacement from TS.
- **hessian_init** (*str, default=None*) -- Path to Hessian HDF5 file, e.g., from a previous TS calculation.
- **displ** (*str, one of ("energy", "length", "energy_cubic")*) -- Controlls initial displacement from the TS. 'energy' assumes a quadratic model, from which a step length for a given energy lowering (see 'displ_energy') is determined. 'length' corresponds to a displacement along the transition vector. 'energy_cubic' considers 3rd derivatives of the energy along the transition vector.
- **displ_energy** (*float, default=1e-3*) -- Required energy lowering from the TS in au (Hartree). Used with 'displ: energy|energy_cubic'.
- **displ_length** (*float, default=0.1*) -- Step length along the transition vector. Used only with 'displ: length'.
- **displ_third_h5** (*str, optional*) -- Path to HDF5 file containing 3rd derivative information. Used with 'displ: energy_cubic'.
- **rms_grad_thresh** (*float, default=1e-3,*) -- Convergence is signalled when to root mean square of the unweighted gradient is less than or equal to this value
- **energy_thresh** (*float, default=1e-6,*) -- Signal convergence when the energy difference between two points is equal to or less than 'energy_thresh'.
- **imag_below** (*float, default=0.0*) -- Require the wavenumber of the imaginary mode to be below the given threshold. If given, it should be a negative number.
- **force_inflection** (*bool, optional*) -- Don't indicate convergence before passing an inflection point.
- **check_bonds** (*bool, optional, default=True*) -- Report whether bonds are formed/broken along the IRC, w.r.t the TS.
- **out_dir** (*str, optional*) -- Dump everything into 'out_dir' directory instead of the CWD.
- **prefix** (*str, optional*) -- Short string that is prepended to all files that are created by this class, e.g., trajectories and HDF5 dumps.
- **dump_fn** (*str, optional*) -- Base name for the HDF5 files.
- **dump_every** (*int, optional*) -- Dump to HDF5 every n-th cycle.

property coords

dump_data(*dump_fn=None*, *full=False*)

dump_ends(*path*, *prefix*, *coords=None*, *trj=False*)

property energy

get_conv_fact(*mw_grad*, *min_fact=2.0*)

get_endpoint_and_ts_geoms()

get_full_irc_data()

get_irc_data()

get_path_for_fn(*fn*)

property gradient

initial_displacement()

Returns non-mass-weighted steps in +s and -s direction for initial displacement from the TS. Earlier version only returned one step, that was later multiplied by either 1 or -1, depending on the desired IRC direction (forward/backward). The current implementation directly returns two steps for forward and backward direction. Whereas for plus and minus steps for displ 'length' and displ 'energy'

step_plus = -step_minus

is valid, it is not valid for dipsl 'energy_cubic' anymore. The latter step is formed as

x(ds) = ds * v0 + ds**2 * v1

so

x(ds) != -x(ds)

as

ds * v0 + ds**2 * v1 != -ds * v0 - ds**2 * v1 .

So, all required step are formed directly and later used as appropriate.

See

https://aip.scitation.org/doi/pdf/10.1063/1.454172 https://pubs.acs.org/doi/10.1021/j100338a027 https://aip.scitation.org/doi/pdf/10.1063/1.459634

irc(*direction*)

log(*msg*)

property m_sqrt

mass_weigh_hessian(*hessian*)

property mw_coords

property mw_gradient

postprocess()

prepare(*direction*)

report_bonds(*prefix*, *bonds*)

report_conv_thresholds()

run()

set_data(*prefix*)

unweight_vec(*vec*)

valid_displs = ('energy', 'length', 'energy_cubic')

pysisyphus.irc.IRCDummy module

class pysisyphus.irc.IRCDummy.**IRCDummy**(*all_coords*, *atoms*, *forward=True*, *backward=True*, *downhill=False*)

Bases: object

all_coords: list

atoms: tuple

backward: bool = True

downhill: bool = False

forward: bool = True

pysisyphus.irc.Instanton module

class pysisyphus.irc.Instanton.**Instanton**(*images*, *calc_getter*, *T*)

Bases: object

property P

P_bh

the first image is connected to the last image. At k=0 the index k-1 will be -1, which points to the last image.

Below we pre-calculate some indices (assuming N images).

unshifted indices ks: k = {0, 1, .. , N-1} shifted indices ksm1: k-1 = {-1, 0, 1, .. , N-2} shifted indices ksp1: k+1 = {1, 2, .. , N-1, 0}

Type

The Instanton is periodic

action()

Action in au / fs, Hartree per femtosecond.

action_gradient()

kin_grad corresponds to the gradient of S_0 in (Eq. 1 in [1], or first term in Eq. 6 in [2].) It boils down to the derivative of a sum of vector norms

d sum_k (||y_k - y_(k-1)||_2)^2 --- d y_k

The derivative of a norm of a vector difference is quite simple, but care has to be taken to recognize, that y_k appears two times in the sum. It appears in the first summand for k and in the second summand for k+1.

sum_k (||y_k - y_(k-1)||_2)^2

1. term 2. term

= (||y_k - y_(k-1)||_2)^2 + (||y_(k+1) - y_k||_2)^2 + ... and so on

The derivative of the first term is

2 * (y_k - y_(k-1))

and the derivative of the second term is

-2 * (y_(k+1) - y_k))

which is equal to

2 * (y_k - y_(k+1)) .

To summarize:

d sum_k(||y_k - y_(k-1)||_2)^2 --- d y_k

= 2 * (2 * y_k - y_(k-1) - y_(k+1)) .

action_hessian()

as_xyz()

property cart_coords

property cart_forces

property cart_hessian

property coords

property energy

property forces

classmethod from_instanton(*other*, ***kwargs*)

classmethod from_ts(*ts_geom*, *P*, *dr=0.4*, *delta_T=25*, *cart_hessian=None*, ***kwargs*)

get_additional_print()

property gradient

property hessian

is_analytical_2d()

property path_length

pysisyphus.irc.Instanton.**T_crossover_from_eigval**(*eigval*)

pysisyphus.irc.Instanton.**T_crossover_from_ts**(*ts_geom*)

pysisyphus.irc.Instanton.**log_progress**(*val*, *key*, *i*)

pysisyphus.irc.LQA module

class pysisyphus.irc.LQA.**LQA**(*geometry*, *N_euler=5000*, ***kwargs*)

Bases: *IRC*

step()

pysisyphus.irc.ModeKill module

class pysisyphus.irc.ModeKill.**ModeKill**(*geometry*, *kill_inds*, *nu_thresh=-5.0*, *do_hess=True*, *hessian_update=True*, ***kwargs*)

Bases: *IRC*

get_additional_print()

postprocess()

prepare(**args*, ***kwargs*)

step()

update_mw_down_step()

pysisyphus.irc.ParamPlot module

class pysisyphus.irc.ParamPlot.**ParamPlot**(*coords_list*, *param1_inds*, *param2_inds*)

Bases: object

calc_angle(*coords*, *ind1*, *ind2*, *ind3*)

calc_bond(*coords*, *ind1*, *ind2*)

get_param(*coords*, *param_inds*)

plot()

save(*path*, *prefix*)

save_coords(*path*, *prefix*)

show()

pysisyphus.irc.RK4 module

class pysisyphus.irc.RK4.**RK4**(*geometry*, *step_length=0.1*, *max_cycles=125*, *downhill=False*, *forward=True*, *backward=True*, *root=0*, *hessian_init=None*, *displ='energy'*, *displ_energy=0.001*, *displ_length=0.1*, *displ_third_h5=None*, *rms_grad_thresh=0.001*, *hard_rms_grad_thresh=None*, *energy_thresh=1e-06*, *imag_below=0.0*, *force_inflection=True*, *check_bonds=True*, *out_dir='.'*, *prefix=''*, *dump_fn='irc_data.h5'*, *dump_every=5*)

Bases: *IRC*

get_k(*mw_coords*)

step()

pysisyphus.irc.initial_displ module

class pysisyphus.irc.initial_displ.**ThirdDerivResult**(*coords_plus*, *energy_plus*, *H_plus*, *coords_minus*, *energy_minus*, *H_minus*, *G_vec*, *vec*, *ds*)

Bases: tuple

G_vec

Alias for field number 6

H_minus

Alias for field number 5

H_plus

Alias for field number 2

coords_minus

Alias for field number 3

coords_plus

Alias for field number 0

ds

Alias for field number 8

energy_minus

Alias for field number 4

energy_plus

Alias for field number 1

vec

Alias for field number 7

pysisyphus.irc.initial_displ.**cubic_displ**(*H*, *v0*, *w0*, *Gv*, *dE*)

According to Eq. (26) in [2] v1 does not depend on the sign of v0.
v1 = (F0 - 2v0^T F0 v0 I)[1] x ([G0v0] - v0^T [G0v0] v0 I) v0

The first term is obviously independent of v0's sign. Using v0' = -v0 the second term becomes

([G0v0'] - v0'^T [G0v0'] v0' I) v0' (-[G0v0] - v0^T [G0v0'] v0 I) v0' (-[G0v0] + v0^T [G0v0] v0 I) v0' -(-[G0v0] + v0^T [G0v0] v0 I) v0 ([G0v0] - v0^T [G0v0] v0 I) v0

Strictly speaking the contraction [G0v0] should depend on the sign of v0. In the current implementation, the direction of v0 is not taken into account, but

get_curv_vec(H, Gv, v0, w0) == get_curv_vec(H, -Gv, -v0, w0) .

But somehow the Taylor expansion gives bogus results when called with -Gv and -v0...

pysisyphus.irc.initial_displ.**cubic_displ_for_geom**(*geom*, *dE=-0.0005*)

pysisyphus.irc.initial_displ.**cubic_displ_for_h5**(*h5_fn='third_deriv.h5'*, *dE=-0.0005*)

pysisyphus.irc.initial_displ.**get_curv_vec**(*H*, *Gv*, *v0*, *w0*)

pysisyphus.irc.initial_displ.**taylor_closure**(*H*, *Gv*, *v0*, *v1*, *w0*)

Taylor expansion of energy to 3rd order.

dx(ds) = ds*v0 + ds**2*v1 / 2 dE(ds) = dx^T H dx / 2 + dx^T [Gv] dx / 6

H = Hessian Gv = 3rd derivative of energy along v0 v0 = Transition vector v1 = Curvature vector w0 = Eigenvalue belonging to v0

pysisyphus.irc.initial_displ.**third_deriv_fd**(*geom*, *vec*, *ds=0.001*)

Third derivative of the energy in direction 'vec'.

Module contents

class pysisyphus.irc.**DampedVelocityVerlet**(*geometry*, *v0=0.04*, *dt0=0.5*, *error_tol=0.003*, *max_cycles=150*, ***kwargs*)

Bases: *IRC*

damp_velocity(*velocity*)

estimate_error(*new_mw_coords*)

mw_grad_to_acc(*mw_grad*)

Takes care of the units for the mass-weighted gradient. Converts units of a mass-weighted gradient [Hartree/(Bohr*amu)] to units of acceleration [sqrt(amu)*Bohr/fs^2]. The 1e30 comes from converting second^2 to femto second^2.

prepare(*direction*)

step()

class pysisyphus.irc.**Euler**(*geometry*, *step_length=0.01*, ***kwargs*)

Bases: *IRC*

step()

class pysisyphus.irc.**EulerPC**(**args*, *hessian_recalc=None*, *hessian_update='bofill'*, *hessian_init='calc'*, *max_pred_steps=500*, *loose_cycles=3*, *dump_dwi=False*, *scipy_method=None*, *corr_func='mbs'*, ***kwargs*)

Bases: *IRC*

corrector_step(*init_mw_coords*, *step_length*, *dwi*)

get_integration_length_func(*init_mw_coords*)

prepare(**args*, ***kwargs*)

scipy_corrector_step(*init_mw_coords*, *step_length*, *dwi*)

Solve IRC equation dx/ds = -g/|g| on DWI PES in mass-weighted coordinates. Integration done until self.step_length in unweighted coordinates is achieved.

step()

class pysisyphus.irc.**GonzalezSchlegel**(*geometry*, *max_micro_cycles=20*, *micro_step_thresh=0.001*, *hessian_recalc=None*, *line_search=False*, ***kwargs*)

Bases: *IRC*

micro_step(*counter*)

Constrained optimization on a hypersphere.

perp_component(*vec*, *perp_to*)

postprocess()

step()

class pysisyphus.irc.**IMKMod**(*geometry*, *corr_first=True*, *corr_first_size=0.5*, *corr_first_energy=True*, *corr_bisec_size=0.0025*, *corr_bisec_energy=True*, ***kwargs*)

Bases: *IRC*

fit_grad_and_energies(*energy0*, *grad0*, *energy1*, *direction*)

fit_parabola(*x*, *y*)

step()

class pysisyphus.irc.**LQA**(*geometry*, *N_euler=5000*, ***kwargs*)

Bases: *IRC*

step()

class pysisyphus.irc.**ModeKill**(*geometry*, *kill_inds*, *nu_thresh=-5.0*, *do_hess=True*, *hessian_update=True*, ***kwargs*)

Bases: *IRC*

get_additional_print()

postprocess()

prepare(**args*, ***kwargs*)

step()

update_mw_down_step()

class pysisyphus.irc.**RK4**(*geometry*, *step_length=0.1*, *max_cycles=125*, *downhill=False*, *forward=True*, *backward=True*, *root=0*, *hessian_init=None*, *displ='energy'*, *displ_energy=0.001*, *displ_length=0.1*, *displ_third_h5=None*, *rms_grad_thresh=0.001*, *hard_rms_grad_thresh=None*, *energy_thresh=1e-06*, *imag_below=0.0*, *force_inflection=True*, *check_bonds=True*, *out_dir='.'*, *prefix=''*, *dump_fn='irc_data.h5'*, *dump_every=5*)

Bases: *IRC*

get_k(*mw_coords*)

step()

17.1.1.11 pysisyphus.line_searches package

Submodules

pysisyphus.line_searches.Backtracking module

class pysisyphus.line_searches.Backtracking.**Backtracking**(**args*, *rho_lo=0.05*, *rho_hi=0.9*, *use_grad=False*, ***kwargs*)

Bases: *LineSearch*

__init__(**args*, *rho_lo=0.05*, *rho_hi=0.9*, *use_grad=False*, ***kwargs*)

Backtracking line search enforcing Armijo conditions.

Uses only energy evaluations.

See [1], Chapter 3, Line Search methods, Section 3.1 p. 31 and Section 3.5 p. 56.

alpha_new_from_phi(*cycle*, *phi0*, *dphi0*, *alpha*, *alpha_prev*)

alpha_new_from_phi_dphi(*cycle*, *phi0*, *dphi0*, *alpha*)

run_line_search()

pysisyphus.line_searches.HagerZhang module

class pysisyphus.line_searches.HagerZhang.**HagerZhang**(**args*, *alpha_prev=None*, *f_prev=None*, *dphi0_prev=None*, *quad_step=False*, *eps=1e-06*, *theta=0.5*, *gamma=0.5*, *rho=5*, *psi_0=0.01*, *psi_1=0.1*, *psi_2=2.0*, *psi_low=0.1*, *psi_hi=10*, *Delta=0.7*, *omega=0.001*, *max_bisects=10*, ***kwargs*)

Bases: *LineSearch*

bisect(*a*, *b*)

Bisect interval [a, b].

bracket(*c*)

Generate initial interval [a, b] that satisfies the opposite slope condition (dphi(a) < 0, dphi(b) > 0).

double_secant(*a*, *b*)

Take secant^2 step.

initial()

Get an initial guess for alpha.

interval_update(*a*, *b*, *c*)

Narrows down the bracketing interval.

norm_inf(*arr*)

Returns infinity norm of given array.

prepare_line_search()

run_line_search()

secant(*a*, *b*)

Take secant step.

take_quad_step(*alpha*, *g0_*)

Try to get alpha for minimum step from quadratic interpolation.

pysisyphus.line_searches.LineSearch module

class pysisyphus.line_searches.LineSearch.**LineSearch**(*p*, *cond='armijo'*, *x0=None*, *geometry=None*, *f=None*, *df=None*, *alpha_init=None*, *f0=None*, *g0=None*, *c1=0.1*, *c2=0.9*, *max_cycles=10*, *alpha_min=1e-06*)

Bases: object

check_alpha(*alpha*)

curvature_condition(*alpha*)

get_fg(*what*, *alpha*)

Lookup raw function/gradient values for a given alpha.

get_phi_dphi(*what*, *alpha*, *check=True*)

Wrapper that handles function/gradient evaluations.

got_alpha_phi_dphi(*alpha*)

log(*message*)

prepare_line_search()

run()

abstract run_line_search()

strong_curvature_condition(*alpha*)

strong_wolfe_condition(*alpha*)

Strong wolfe condition.

sufficiently_decreased(*alpha*)

Sufficient decrease/Armijo condition.

wolfe_condition(*alpha*)

Normal, not strong, Wolfe condition.

exception pysisyphus.line_searches.LineSearch.**LineSearchConverged**(*alpha*)

Bases: Exception

exception pysisyphus.line_searches.LineSearch.**LineSearchNotConverged**

Bases: Exception

class pysisyphus.line_searches.LineSearch.**LineSearchResult**(*converged*, *alpha*, *f_new*, *g_new*, *f_evals*, *df_evals*, *dphi0*)

Bases: tuple

alpha

Alias for field number 1

converged
Alias for field number 0

df_evals
Alias for field number 5

dphi0
Alias for field number 6

f_evals
Alias for field number 4

f_new
Alias for field number 2

g_new
Alias for field number 3

pysisyphus.line_searches.StrongWolfe module

class pysisyphus.line_searches.StrongWolfe.**StrongWolfe**(**args*, *alpha_max=10.0*, *fac=2*, ***kwargs*)
Bases: *LineSearch*

__init__(**args*, *alpha_max=10.0*, *fac=2*, ***kwargs*)
Wolfe line search.

Uses only energy & gradient evaluations.

See [1], Chapter 3, Line Search methods, Section 3.5 p. 60.

run_line_search()

zoom(*alpha_lo*, *alpha_hi*, *phi_lo*, *phi_alpha_=None*, *alpha_0_=None*, *max_cycles=10*)

pysisyphus.line_searches.interpol module

pysisyphus.line_searches.interpol.**interpol_alpha_cubic**(*f_0*, *df_0*, *f_alpha_0*, *f_alpha_1*, *alpha_0*, *alpha_1*)

pysisyphus.line_searches.interpol.**interpol_alpha_quad**(*f_0*, *df_0*, *f_alpha_0*, *alpha_0*)

Module contents

class pysisyphus.line_searches.**Backtracking**(**args*, *rho_lo=0.05*, *rho_hi=0.9*, *use_grad=False*, ***kwargs*)
Bases: *LineSearch*

__init__(**args*, *rho_lo=0.05*, *rho_hi=0.9*, *use_grad=False*, ***kwargs*)
Backtracking line search enforcing Armijo conditions.

Uses only energy evaluations.

See [1], Chapter 3, Line Search methods, Section 3.1 p. 31 and Section 3.5 p. 56.

alpha_new_from_phi(*cycle*, *phi0*, *dphi0*, *alpha*, *alpha_prev*)

alpha_new_from_phi_dphi(*cycle*, *phi0*, *dphi0*, *alpha*)

run_line_search()

class pysisyphus.line_searches.**HagerZhang**(**args*, *alpha_prev=None*, *f_prev=None*, *dphi0_prev=None*, *quad_step=False*, *eps=1e-06*, *theta=0.5*, *gamma=0.5*, *rho=5*, *psi_0=0.01*, *psi_1=0.1*, *psi_2=2.0*, *psi_low=0.1*, *psi_hi=10*, *Delta=0.7*, *omega=0.001*, *max_bisects=10*, ***kwargs*)

Bases: *LineSearch*

bisect(*a*, *b*)

Bisect interval [a, b].

bracket(*c*)

Generate initial interval [a, b] that satisfies the opposite slope condition (dphi(a) < 0, dphi(b) > 0).

double_secant(*a*, *b*)

Take secant^2 step.

initial()

Get an initial guess for alpha.

interval_update(*a*, *b*, *c*)

Narrows down the bracketing interval.

norm_inf(*arr*)

Returns infinity norm of given array.

prepare_line_search()

run_line_search()

secant(*a*, *b*)

Take secant step.

take_quad_step(*alpha*, *g0_*)

Try to get alpha for minimum step from quadratic interpolation.

class pysisyphus.line_searches.**StrongWolfe**(**args*, *alpha_max=10.0*, *fac=2*, ***kwargs*)

Bases: *LineSearch*

__init__(**args*, *alpha_max=10.0*, *fac=2*, ***kwargs*)

Wolfe line search.

Uses only energy & gradient evaluations.

See [1], Chapter 3, Line Search methods, Section 3.5 p. 60.

run_line_search()

zoom(*alpha_lo*, *alpha_hi*, *phi_lo*, *phi_alpha_=None*, *alpha_0_=None*, *max_cycles=10*)

17.1.1.12 pysisyphus.modefollow package

Submodules

pysisyphus.modefollow.NormalMode module

class pysisyphus.modefollow.NormalMode.**NormalMode**(*l*, *masses*)

Bases: object

See http://gaussian.com/vib/

__init__(*l*, *masses*)

NormalMode class.

Cartesian displacements are normalized to 1.

Parameters

- **l** (*np.array*) -- Cartesian, non-mass-weighted displacements.
- **masses** (*np.array*) -- Atomic masses.

property l_mw

mw_norm_for_norm(*norm=0.01*)

property red_mass

pysisyphus.modefollow.davidson module

class pysisyphus.modefollow.davidson.**DavidsonResult**(*cur_cycle*, *converged*, *final_modes*, *qs*, *nus*, *mode_inds*, *res_rms*)

Bases: tuple

converged

Alias for field number 1

cur_cycle

Alias for field number 0

final_modes

Alias for field number 2

mode_inds

Alias for field number 5

nus

Alias for field number 4

qs

Alias for field number 3

res_rms

Alias for field number 6

pysisyphus.modefollow.davidson.**block_davidson**(*cart_coords*, *masses*, *forces_getter*, *guess_modes*, *lowest=None*, *trial_step_size=0.01*, *hessian_precon=None*, *max_cycles=25*, *res_rms_thresh=0.0001*, *start_precon=5*, *remove_trans_rot=True*, *print_level=1*)

pysisyphus.modefollow.davidson.**forces_fin_diff**(*forces_getter*, *coords*, *b*, *step_size*)

pysisyphus.modefollow.davidson.**geom_davidson**(*geom*, **args*, ***kwargs*)

pysisyphus.modefollow.lanczos module

pysisyphus.modefollow.lanczos.**geom_lanczos**(*geom*, **args*, ***kwargs*)

Wraps Lanczos algorithm for use with Geometry objects.

pysisyphus.modefollow.lanczos.**lanczos**(*coords*, *grad_getter*, *dx=0.005*, *dl=0.01*, *guess=None*, *max_cycles=25*, *reortho=True*, *logger=None*)

Lanczos method to determine smallest eigenvalue & -vector.

See [1] for description of algorithm.

Module contents

class pysisyphus.modefollow.**NormalMode**(*l*, *masses*)

Bases: `object`

See http://gaussian.com/vib/

__init__(*l*, *masses*)

NormalMode class.

Cartesian displacements are normalized to 1.

Parameters

- **l** (*np.array*) -- Cartesian, non-mass-weighted displacements.
- **masses** (*np.array*) -- Atomic masses.

property l_mw

mw_norm_for_norm(*norm=0.01*)

property red_mass

pysisyphus.modefollow.**geom_davidson**(*geom*, **args*, ***kwargs*)

pysisyphus.modefollow.**geom_lanczos**(*geom*, **args*, ***kwargs*)

Wraps Lanczos algorithm for use with Geometry objects.

17.1.1.13 pysisyphus.optimizers package

Submodules

pysisyphus.optimizers.BFGS module

class pysisyphus.optimizers.BFGS.**BFGS**(*geometry*, **args*, *update='bfgs'*, ***kwargs*)

Bases: *Optimizer*

bfgs_update(*s*, *y*)

damped_bfgs_update(*s*, *y*, *mu_1=0.2*)

Damped BFGS update of inverse Hessian.

Potentially updates s. See Section 3.2 of [2], Eq. (30) - (33). There is a typo ;) It should be

H_{k+1} = V_k H_k V_k^T + ...

instead of
H_{k+1} = V_k^T H_k V_k + ...

double_damped_bfgs_update(*s*, *y*, *mu_1=0.2*, *mu_2=0.2*)

Double damped BFGS update of inverse Hessian.

See [3]. Potentially updates s and y.

property eye

optimize()

prepare_opt()

pysisyphus.optimizers.BacktrackingOptimizer module

class pysisyphus.optimizers.BacktrackingOptimizer.**BacktrackingOptimizer**(*geometry*, *alpha*, *bt_force=5*, *dont_skip_after=2*, *bt_max_scale=4*, *bt_disable=False*, ***kwargs*)

Bases: *Optimizer*

backtrack(*cur_forces*, *prev_forces*, *reset_hessian=None*)

Accelerated backtracking line search.

reset()

pysisyphus.optimizers.ConjugateGradient module

class pysisyphus.optimizers.ConjugateGradient.**ConjugateGradient**(*geometry*, *alpha=0.1*, *formula='FR'*, *dont_skip=True*, ***kwargs*)

Bases: *BacktrackingOptimizer*

get_beta(*cur_forces*, *prev_forces*)

optimize()

prepare_opt()

reset()

pysisyphus.optimizers.CubicNewton module

class pysisyphus.optimizers.CubicNewton.**CubicNewton**(*geometry*, ***kwargs*)

Bases: *HessianOptimizer*

optimize()

postprocess_opt()

pysisyphus.optimizers.FIRE module

class pysisyphus.optimizers.FIRE.**FIRE**(*geometry*, *dt=0.1*, *dt_max=1*, *N_acc=2*, *f_inc=1.1*, *f_acc=0.99*, *f_dec=0.5*, *n_reset=0*, *a_start=0.1*, ***kwargs*)

Bases: *Optimizer*

optimize()

reset()

pysisyphus.optimizers.HessianOptimizer module

class pysisyphus.optimizers.HessianOptimizer.**HessianOptimizer**(*geometry*, *trust_radius=0.5*, *trust_update=True*, *trust_min=0.1*, *trust_max=1*, *max_energy_incr=None*, *hessian_update='bfgs'*, *hessian_init='fischer'*, *hessian_recalc=None*, *hessian_recalc_adapt=None*, *hessian_xtb=False*, *hessian_recalc_reset=False*, *small_eigval_thresh=1e-08*, *line_search=False*, *alpha0=1.0*, *max_micro_cycles=25*, *rfo_overlaps=False*, ***kwargs*)

Bases: *Optimizer*

__init__(*geometry*, *trust_radius=0.5*, *trust_update=True*, *trust_min=0.1*, *trust_max=1*, *max_energy_incr=None*, *hessian_update='bfgs'*, *hessian_init='fischer'*, *hessian_recalc=None*, *hessian_recalc_adapt=None*, *hessian_xtb=False*, *hessian_recalc_reset=False*, *small_eigval_thresh=1e-08*, *line_search=False*, *alpha0=1.0*, *max_micro_cycles=25*, *rfo_overlaps=False*, ***kwargs*)

Baseclass for optimizers utilizing Hessian information.

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.
- **trust_radius** (float, default: 0.5) -- Initial trust radius in whatever unit the optimization is carried out.
- **trust_update** (bool, default: True) -- Whether to update the trust radius throughout the optimization.
- **trust_min** (float, default: 0.1) -- Minimum trust radius.
- **trust_max** (float, default: 1) -- Maximum trust radius.
- **max_energy_incr** (Optional[float], default: None) -- Maximum allowed energy increased after a faulty step. Optimization is aborted when the threshold is exceeded.
- **hessian_update** (Literal['none', None, False, 'bfgs', 'damped_bfgs', 'flowchart', 'bofill', 'ts_bfgs', 'ts_bfgs_org', 'ts_bfgs_rev'], default: 'bfgs') -- Type of Hessian update. Defaults to BFGS for minimizations and Bofill for saddle point searches.
- **hessian_init** (Literal['calc', 'unit', 'fischer', 'lindh', 'simple', 'swart', 'xtb', 'xtb1', 'xtbff'], default: 'fischer') -- Type of initial model Hessian.
- **hessian_recalc** (Optional[int], default: None) -- Recalculate exact Hessian every n-th cycle instead of updating it.
- **hessian_recalc_adapt** (Optional[float], default: None) -- Use a more flexible scheme to determine Hessian recalculation. Undocumented.
- **hessian_xtb** (bool, default: False) -- Recalculate the Hessian at the GFN2-XTB level of theory.
- **hessian_recalc_reset** (bool, default: False) -- Whether to skip Hessian recalculation after reset. Undocumented.
- **small_eigval_thresh** (float, default: 1e-08) -- Threshold for small eigenvalues. Eigenvectors belonging to eigenvalues below this threshold are discardewd.
- **line_search** (bool, default: False) -- Whether to carry out a line search. Not implemented by a subclassing optimizers.
- **alpha0** (float, default: 1.0) -- Initial alpha for restricted-step (RS) procedure.
- **max_micro_cycles** (int, default: 25) -- Maximum number of RS iterations.
- **rfo_overlaps** (bool, default: False) -- Enable mode-following in RS procedure.
- ****kwargs** -- Keyword arguments passed to the Optimizer baseclass.

filter_small_eigvals(*eigvals*, *eigvecs*, *mask=False*)

get_alpha_step(*cur_alpha*, *rfo_eigval*, *step_norm*, *eigvals*, *gradient*)

get_augmented_hessian(*eigvals*, *gradient*, *alpha=1.0*)

static get_newton_step(*eigvals*, *eigvecs*, *gradient*)

get_newton_step_on_trust(*eigvals*, *eigvecs*, *gradient*, *transform=True*)

Step on trust-radius.

See Nocedal 4.3 Iterative solutions of the subproblem

get_rs_step(*eigvals*, *eigvecs*, *gradient*, *name='RS'*)

static get_shifted_step_trans(*eigvals*, *gradient_trans*, *shift*)

get_step_func(*eigvals*, *gradient*, *grad_rms_thresh=0.01*)

housekeeping()

Calculate gradient and energy. Update trust radius and hessian if needed. Return energy, gradient and hessian for the current cycle.

log_negative_eigenvalues(*eigvals*, *pre_str=''*)

prepare_opt(*hessian_init=None*)

property prev_eigvec_max

property prev_eigvec_min

static quadratic_model(*gradient*, *hessian*, *step*)

reset()

rfo_dict = {'max': (-1, 'max'), 'min': (0, 'min')}

static rfo_model(*gradient*, *hessian*, *step*)

save_hessian()

set_new_trust_radius(*coeff*, *last_step_norm*)

solve_rfo(*rfo_mat*, *kind='min'*, *prev_eigvec=None*)

update_hessian()

update_trust_radius()

pysisyphus.optimizers.HessianOptimizer.**dummy_hessian_update**(*H*, *dx*, *dg*)

pysisyphus.optimizers.LBFGS module

class pysisyphus.optimizers.LBFGS.**LBFGS**(*geometry*, *keep_last=7*, *beta=1*, *max_step=0.2*, *double_damp=True*, *gamma_mult=False*, *line_search=False*, *mu_reg=None*, *max_mu_reg_adaptions=10*, *control_step=True*, ***kwargs*)

Bases: *Optimizer*

__init__(*geometry*, *keep_last=7*, *beta=1*, *max_step=0.2*, *double_damp=True*, *gamma_mult=False*, *line_search=False*, *mu_reg=None*, *max_mu_reg_adaptions=10*, *control_step=True*, ***kwargs*)

Limited-memory BFGS optimizer.

See [1] Nocedal, Wright - Numerical Optimization, 2006 for a general discussion of LBFGS. See pysisyphus.optimizers.hessian_updates for the references related to double damping and pysisyphus.optimizers.closures for references related to regularized LBFGS.

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.
- **keep_last** (int, default: 7) -- History size. Keep last 'keep_last' steps and gradient differences.
- **beta** (float, default: 1) -- Force constant in -(H + I)^{1}g.
- **max_step** (float, default: 0.2) -- Upper limit for the absolute component of the step vector in whatever unit the optimization is carried out.
- **double_damp** (bool, default: True) -- Use double damping procedure to modify steps s and gradient differences y to ensure sy > 0.
- **gamma_mult** (bool, default: False) -- Estimate from previous cycle. Eq. (7.20) in [1]. See 'beta' argument.
- **line_search** (bool, default: False) -- Enable implicit linesearches.
- **mu_reg** (Optional[float], default: None) -- Initial guess for regularization constant in regularized LBFGS.
- **max_mu_reg_adaptions** (int, default: 10) -- Maximum number of trial steps in regularized LBFGS.
- **control_step** (bool, default: True) -- Wheter to scale down the proposed step its biggest absolute component is equal to or below 'max_step'
- ****kwargs** -- Keyword arguments passed to the Optimizer baseclass.

get_lbfgs_step(*forces*)

optimize()

postprocess_opt()

reset()

pysisyphus.optimizers.LayerOpt module

class pysisyphus.optimizers.LayerOpt.**LayerOpt**(*geometry*, *layers=None*, ***kwargs*)

Bases: *Optimizer*

check_convergence(**args*, ***kwargs*)

Check if we must use the model optimizer to signal convergence.

property layer_num: int

property model_opt

Return the persistent optimizer belonging to the model system. We don't have to supply any coordinates to the optimizer of the most expensive layer, as it is persistent, as well as the associated geometry.

optimize()

Return type
None

postprocess_opt()

Return type
None

print_opt_progress(**args*, ***kwargs*)

Pick the correct method to report opt_progress.

When the model optimizer decides convergence we also report optimization progress using its data and not the data from LayerOpt, where the total ONIOM gradient is stored.

class pysisyphus.optimizers.LayerOpt.**Layers**(*geometry*, *opt_thresh*, *layers=None*)

Bases: object

classmethod **from_oniom_calculator**(*geometry*, *oniom_calc=None*, *layers=None*, ***kwargs*)

pysisyphus.optimizers.LayerOpt.**get_geom_kwargs**(*layer_ind*, *layer_mask*)

pysisyphus.optimizers.LayerOpt.**get_opt_kwargs**(*opt_key*, *layer_ind*, *thresh*)

pysisyphus.optimizers.MicroOptimizer module

class pysisyphus.optimizers.MicroOptimizer.**MicroOptimizer**(*geom*, *step='lbfgs'*, *line_search=True*, *max_cycles=100000000*, *max_step=0.2*, *keep_last=10*, *rms_force=None*, *double_damp=True*, *dump=False*, ***kwargs*)

Bases: object

cg_step(*forces*)

lbfgs_step(*forces*)

log(*msg*)

optimize()

run()

sd_step(*forces*)

take_step(*energy*, *forces*, *return_step=False*)

pysisyphus.optimizers.NCOptimizer module

class pysisyphus.optimizers.NCOptimizer.**NCOptimizer**(*geometry*, **args*, *freeze_modes=None*, ***kwargs*)

Bases: *HessianOptimizer*

optimize()

pysisyphus.optimizers.Optimizer module

class pysisyphus.optimizers.Optimizer.**ConvInfo**(*cur_cycle*, *energy_converged*, *max_force_converged*, *rms_force_converged*, *max_step_converged*, *rms_step_converged*, *desired_eigval_structure*)

Bases: object

cur_cycle: int

desired_eigval_structure: bool

energy_converged: bool

get_convergence()

is_converged()

max_force_converged: bool

max_step_converged: bool

rms_force_converged: bool

rms_step_converged: bool

class pysisyphus.optimizers.Optimizer.**Optimizer**(*geometry*, *thresh='gau_loose'*, *max_step=0.04*, *max_cycles=150*, *min_step_norm=1e-08*, *assert_min_step=True*, *rms_force=None*, *rms_force_only=False*, *max_force_only=False*, *force_only=False*, *converge_to_geom_rms_thresh=0.05*, *align=False*, *align_factor=1.0*, *dump=False*, *dump_restart=False*, *print_every=1*, *prefix=''*, *reparam_thresh=0.001*, *reparam_check_rms=True*, *reparam_when='after'*, *overachieve_factor=0.0*, *check_eigval_structure=False*, *restart_info=None*, *check_coord_diffs=True*, *coord_diff_thresh=0.01*, *fragments=None*, *monitor_frag_dists=0*, *out_dir='.'*, *h5_fn='optimization.h5'*, *h5_group_name='opt'*)

Bases: object

__init__(*geometry*, *thresh='gau_loose'*, *max_step=0.04*, *max_cycles=150*, *min_step_norm=1e-08*, *assert_min_step=True*, *rms_force=None*, *rms_force_only=False*, *max_force_only=False*, *force_only=False*, *converge_to_geom_rms_thresh=0.05*, *align=False*, *align_factor=1.0*, *dump=False*, *dump_restart=False*, *print_every=1*, *prefix=''*, *reparam_thresh=0.001*, *reparam_check_rms=True*, *reparam_when='after'*, *overachieve_factor=0.0*, *check_eigval_structure=False*, *restart_info=None*, *check_coord_diffs=True*, *coord_diff_thresh=0.01*, *fragments=None*, *monitor_frag_dists=0*, *out_dir='.'*, *h5_fn='optimization.h5'*, *h5_group_name='opt'*)

Optimizer baseclass. Meant to be subclassed.

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.
- **thresh** (Literal['gau_loose', 'gau', 'gau_tight', 'gau_vtight', 'baker', 'never'], default: 'gau_loose') -- Convergence threshold.
- **max_step** (float, default: 0.04) -- Maximum absolute component of the allowed step vector. Utilized in optimizers that don't support a trust region or line search.
- **max_cycles** (int, default: 150) -- Maximum number of allowed optimization cycles.
- **min_step_norm** (float, default: 1e-08) -- Minimum norm of an allowed step. If the step norm drops below this value a ZeroStepLength-exception is raised. The unit depends on the coordinate system of the supplied geometry.
- **assert_min_step** (bool, default: True) -- Flag that controls whether the norm of the proposed step is check for being too small.
- **rms_force** (Optional[float], default: None) -- Root-mean-square of the force from which user-defined thresholds are derived. When 'rms_force' is given 'thresh' is ignored.
- **rms_force_only** (bool, default: False) -- When set, convergence is signalled only based on rms(forces).
- **max_force_only** (bool, default: False) -- When set, convergence is signalled only based on max(**|forces|**).
- **force_only** (bool, default: False) -- When set, convergence is signalled only based on max(**|forces|**) and rms(forces).
- **converge_to_geom_rms_thresh** (float, default: 0.05) -- Threshold for the RMSD with another geometry. When the RMSD drops below this threshold convergence is signalled. Only used with Growing Newton trajectories.
- **align** (bool, default: False) -- Flag that controls whether the geometry is aligned in every step onto the coordinates of the previous step. Must not be used with internal coordinates.
- **align_factor** (float, default: 1.0) -- Factor that controls the strength of the alignment. 1.0 means full alignment, 0.0 means no alignment. The factor mixes the rotation matrix of the alignment with the identity matrix.
- **dump** (bool, default: False) -- Flag to control dumping/writing of optimization progress to the filesystem
- **dump_restart** (bool, default: False) -- Flag to control whether restart information is dumped to the filesystem.
- **print_every** (int, default: 1) -- Report optimization progress every nth cycle.
- **prefix** (str, default: '') -- Short string that is prepended to several files created by the optimizer. Allows distinguishing several optimizations carried out in the same directory.
- **reparam_thresh** (float, default: 0.001) -- Controls the minimal allowed similarity between coordinates after two successive reparametrizations. Convergence is signalled if the coordinates did not change significantly.
- **reparam_check_rms** (bool, default: True) -- Whether to check for (too) similar coordinates after reparametrization.

- **reparam_when** (Optional[Literal['before', 'after']], default: 'after') -- Reparametrize before or after calculating the step. Can also be turned off by setting it to None.
- **overachieve_factor** (float, default: 0.0) -- Signal convergence when max(forces) and rms(forces) fall below the chosen threshold, divided by this factor. Convergence of max(step) and rms(step) is ignored.
- **check_eigval_structure** (bool, default: False) -- Check the eigenvalues of the modes we maximize along. Convergence requires them to be negative. Useful if TS searches are started from geometries close to a minimum.
- **restart_info** (default: None) -- Restart information. Undocumented.
- **check_coord_diffs** (bool, default: True) -- Whether coordinates of chain-of-sates images are checked for being too similar.
- **coord_diff_thresh** (float, default: 0.01) -- Unitless threshold for similary checking of COS image coordinates. The first image is assigned 0, the last image is assigned to 1.
- **fragments** (Optional[Tuple], default: None) -- Tuple of lists containing atom indices, defining two fragments.
- **monitor_frag_dists** (int, default: 0) -- Monitor fragment distances for N cycles. The optimization is terminated when the interfragment distances falls below the initial value after N cycles.
- **out_dir** (str, default: '.') -- String poiting to a directory where optimization progress is dumped.
- **h5_fn** (str, default: 'optimization.h5') -- Basename of the HDF5 file used for dumping.
- **h5_group_name** (str, default: 'opt') -- Groupname used for dumping of this optimization.

check_convergence(*step=None*, *multiple=1.0*, *overachieve_factor=None*)

Check if the current convergence of the optimization is equal to or below the required thresholds, or a multiple thereof. The latter may be used in initiating the climbing image.

dump_restart_info()

final_summary()

fit_rigid(*, *vectors=None*, *vector_lists=None*, *hessian=None*)

get_path_for_fn(*fn*, *with_prefix=True*)

get_restart_info()

log(*message*, *level=50*)

make_conv_dict(*key*, *rms_force=None*, *rms_force_only=False*, *max_force_only=False*, *force_only=False*)

abstract optimize()

postprocess_opt()

prepare_opt()

print_opt_progress(*conv_info*)

procrustes()

Wrapper for procrustes that passes additional arguments along.

report_conv_thresholds()

run()

scale_by_max_step(*steps*)

set_restart_info(*restart_info*)

write_cycle_to_file()

write_image_trjs()

write_results()

write_to_out_dir(*out_fn*, *content*, *mode='w'*)

pysisyphus.optimizers.Optimizer.**get_data_model**(*geometry*, *is_cos*, *max_cycles*)

pysisyphus.optimizers.PreconLBFGS module

class pysisyphus.optimizers.PreconLBFGS.**PreconLBFGS**(*geometry*, *alpha_init=1.0*, *history=7*, *precon=True*, *precon_update=1*, *precon_getter_update=None*, *precon_kind='full'*, *max_step_element=None*, *line_search='armijo'*, *c_stab=None*, ***kwargs*)

Bases: *Optimizer*

__init__(*geometry*, *alpha_init=1.0*, *history=7*, *precon=True*, *precon_update=1*, *precon_getter_update=None*, *precon_kind='full'*, *max_step_element=None*, *line_search='armijo'*, *c_stab=None*, ***kwargs*)

Preconditioned limited-memory BFGS optimizer.

See pysisyphus.optimizers.precon for related references.

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.
- **alpha_init** (`float`, default: `1.0`) -- Initial scaling factor for the first trial step in the excplicit line search.
- **history** (`int`, default: `7`) -- History size. Keep last 'history' steps and gradient differences.
- **precon** (`bool`, default: `True`) -- Wheter to use preconditioning or not.
- **precon_update** (`int`, default: `1`) -- Recalculate preconditioner P in every n-th cycle with the same topology.
- **precon_getter_update** (`Optional[int]`, default: `None`) -- Recalculate topology for preconditioner P in every n-th cycle. It is usually sufficient to only determine the topology once at the beginning.
- **precon_kind** (`Literal['full', 'full_fast', 'bonds', 'bonds_bends']`, default: `'full'`) -- What types of primitive internal coordinates to consider in the preconditioner.
- **max_step_element** (`Optional[float]`, default: `None`) -- Maximum component of the absolute step vector when no line search is carried out.

- **line_search** (Literal['armijo', 'armijo_fg', 'strong_wolfe', 'hz', None, False], default: 'armijo') -- Whether to use explicit line searches and if so, which kind of line search.
- **c_stab** (Optional[float], default: None) -- Regularization constant c in (H + cI)[1] in atomic units.
- ****kwargs** -- Keyword arguments passed to the Optimizer baseclass.

get_precon_getter()

optimize()

prepare_opt()

scale_max_element(*step*, *max_step_element*)

pysisyphus.optimizers.PreconSteepestDescent module

class pysisyphus.optimizers.PreconSteepestDescent.**PreconSteepestDescent**(*geometry*, *alpha_init=0.5*, ***kwargs*)

Bases: *PreconLBFGS*

pysisyphus.optimizers.QuickMin module

class pysisyphus.optimizers.QuickMin.**QuickMin**(*geometry*, *dt=0.35*, ***kwargs*)

Bases: *Optimizer*

optimize()

prepare_opt()

reset()

pysisyphus.optimizers.RFOptimizer module

class pysisyphus.optimizers.RFOptimizer.**RFOptimizer**(*geometry*, *line_search=True*, *gediis=False*, *gdiis=True*, *gdiis_thresh=0.0025*, *gediis_thresh=0.01*, *gdiis_test_direction=True*, *max_micro_cycles=25*, *adapt_step_func=False*, ***kwargs*)

Bases: *HessianOptimizer*

__init__(*geometry*, *line_search=True*, *gediis=False*, *gdiis=True*, *gdiis_thresh=0.0025*, *gediis_thresh=0.01*, *gdiis_test_direction=True*, *max_micro_cycles=25*, *adapt_step_func=False*, ***kwargs*)

Rational function Optimizer.

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.
- **line_search** (bool, default: True) -- Whether to carry out implicit line searches.
- **gediis** (bool, default: False) -- Whether to enable GEDIIS.

- **gdiis** (bool, default: True) -- Whether to enable GDIIS.
- **gdiis_thresh** (float, default: 0.0025) -- Threshold for rms(forces) to enable GDIIS.
- **gediis_thresh** (float, default: 0.01) -- Threshold for rms(step) to enable GEDIIS.
- **gdiis_test_direction** (bool, default: True) -- Whether to the overlap of the RFO step and the GDIIS step.
- **max_micro_cycles** (int, default: 25) -- Number of restricted-step microcycles. Disabled by default.
- **adapt_step_func** (bool, default: False) -- Whether to switch between shifted Newton and RFO-steps.
- ****kwargs** -- Keyword arguments passed to the Optimizer/HessianOptimizer baseclass.

optimize()

postprocess_opt()

pysisyphus.optimizers.RSA module

class pysisyphus.optimizers.RSA.**RSA**(*geometry*, *trust_radius=0.5*, *trust_update=True*, *trust_min=0.1*, *trust_max=1*, *max_energy_incr=None*, *hessian_update='bfgs'*, *hessian_init='fischer'*, *hessian_recalc=None*, *hessian_recalc_adapt=None*, *hessian_xtb=False*, *hessian_recalc_reset=False*, *small_eigval_thresh=1e-08*, *line_search=False*, *alpha0=1.0*, *max_micro_cycles=25*, *rfo_overlaps=False*, ***kwargs*)

Bases: *HessianOptimizer*

The Importance of Step Control in Optimization Methods, del Campo, 2009.

optimize()

pysisyphus.optimizers.StabilizedQNMethod module

class pysisyphus.optimizers.StabilizedQNMethod.**StabilizedQNMethod**(*geometry*, *alpha=0.5*, *alpha_max=1*, *alpha_stretch=0.5*, *alpha_stretch_max=1*, *eps=0.0001*, *hist_max=10*, *E_thresh=1e-06*, *bio=True*, *trust_radius=0.1*, *linesearch=True*, ***kwargs*)

Bases: *Optimizer*

adjust_alpha(*gradient*, *precon_gradient*)

adjust_alpha_stretch()

bio_mode(*gradient*)

property n_hist

optimize()

precondition_gradient(*gradient*, *steps*, *grad_diffs*, *eps*)

prepare_opt()

pysisyphus.optimizers.SteepestDescent module

class pysisyphus.optimizers.SteepestDescent.**SteepestDescent**(*geometry*, *alpha=0.1*, ***kwargs*)

Bases: *BacktrackingOptimizer*

optimize()

prepare_opt()

pysisyphus.optimizers.StringOptimizer module

class pysisyphus.optimizers.StringOptimizer.**StringOptimizer**(*geometry*, *max_step=0.1*, *stop_in_when_full=-1*, *keep_last=10*, *lbfgs_when_full=True*, *gamma_mult=False*, *double_damp=True*, *scale_step='global'*, ***kwargs*)

Bases: *Optimizer*

check_convergence(**args*, ***kwargs*)

Check if the current convergence of the optimization is equal to or below the required thresholds, or a multiple thereof. The latter may be used in initiating the climbing image.

optimize()

prepare_opt()

reset()

restrict_step_components(*steps*)

pysisyphus.optimizers.closures module

pysisyphus.optimizers.closures.**bfgs_multiply**(*s_list*, *y_list*, *vector*, *beta=1*, *P=None*, *logger=None*, *gamma_mult=True*, *mu_reg=None*, *inds=None*, *cur_size=None*)

Matrix-vector product H·v.

Multiplies given vector with inverse Hessian, obtained from repeated BFGS updates calculated from steps in 's_list' and gradient differences in 'y_list'.

Based on algorithm 7.4 Nocedal, Num. Opt., p. 178.

pysisyphus.optimizers.closures.**get_update_mu_reg**(*mu_min=0.001*, *gamma_1=0.1*, *gamma_2=5.0*, *eta_1=0.01*, *eta_2=0.9*, *logger=None*)

See 5.1 in [1]

pysisyphus.optimizers.closures.**lbfgs_closure**(*force_getter*, *M=10*, *beta=1*, *restrict_step=None*)

pysisyphus.optimizers.closures.**modified_broyden_closure**(*force_getter*, *M=5*, *beta=1*, *restrict_step=None*)

https://doi.org/10.1006/jcph.1996.0059 F corresponds to the residual gradient, so we after calling force_getter we multiply the force by -1 to get the gradient.

pysisyphus.optimizers.closures.**small_lbfgs_closure**(*history=5*, *gamma_mult=True*)

Compact LBFGS closure.

The returned function takes two arguments: forces and prev_step. forces are the forces at the current iterate and prev_step is the previous step that lead us to the current iterate. In this way step restriction/line search can be done outisde of the lbfgs function.

pysisyphus.optimizers.cls_map module

pysisyphus.optimizers.cls_map.**get_opt_cls**(*opt_key*)

pysisyphus.optimizers.cls_map.**key_is_tsopt**(*opt_key*)

pysisyphus.optimizers.exceptions module

exception pysisyphus.optimizers.exceptions.**OptimizationError**

Bases: Exception

exception pysisyphus.optimizers.exceptions.**ZeroStepLength**

Bases: Exception

pysisyphus.optimizers.gdiis module

class pysisyphus.optimizers.gdiis.**DIISResult**(*coeffs*, *coords*, *forces*, *energy*, *N*, *type*)

Bases: tuple

N

Alias for field number 4

coeffs

Alias for field number 0

coords

Alias for field number 1

energy

Alias for field number 3

forces

Alias for field number 2

type

Alias for field number 5

pysisyphus.optimizers.gdiis.**diis_result**(*coeffs*, *coords*, *forces*, *energy=None*, *prefix=''*)

pysisyphus.optimizers.gdiis.**from_coeffs**(*vec*, *coeffs*)

pysisyphus.optimizers.gdiis.**gdiis**(*err_vecs*, *coords*, *forces*, *ref_step*, *max_vecs=5*, *test_direction=True*)

pysisyphus.optimizers.gdiis.**gediis**(*coords*, *energies*, *forces*, *hessian=None*, *max_vecs=3*)

pysisyphus.optimizers.gdiis.**log**(*msg*)

pysisyphus.optimizers.gdiis.**valid_diis_direction**(*diis_step*, *ref_step*, *use*)

pysisyphus.optimizers.guess_hessians module

pysisyphus.optimizers.guess_hessians.**fischer_guess**(*geom*)

pysisyphus.optimizers.guess_hessians.**get_guess_hessian**(*geometry*, *hessian_init*, *int_gradient=None*, *cart_gradient=None*, *h5_fn=None*)

Obtain/calculate (model) Hessian.

For hessian_init="calc" the Hessian will be in the coord_type of the geometry, otherwise a Hessian in primitive internals will be returned.

pysisyphus.optimizers.guess_hessians.**get_lindh_alpha**(*atom1*, *atom2*)

pysisyphus.optimizers.guess_hessians.**improved_guess**(*geom*, *bond_func*, *bend_func*, *dihedral_func*)

pysisyphus.optimizers.guess_hessians.**lindh_guess**(*geom*)

Slightly modified Lindh model hessian as described in [1].

Instead of using the tabulated r_ref,ij values from [1] we will use the 'true' covalent radii as pyberny. The tabulated r_ref,ij value for two carbons (2nd period) is 2.87 Bohr. Carbons covalent radius is ~ 1.44 Bohr, so 2*1.44 Bohr = 2.88 Bohr which fits nicely with the tabulate value. If values for elements > 3rd are requested the alpha values for the 3rd period will be (re)used.

pysisyphus.optimizers.guess_hessians.**lindh_style_guess**(*geom*, *ks*, *rhos*)

Approximate force constants according to Lindh.[1]

Bonds: k_ij = k_r * rho_ij Bends: k_ijk = k_b * rho_ij * rho_jk Dihedrals: k_ijkl = k_d * rho_ij * rho_jk * rho_kl

pysisyphus.optimizers.guess_hessians.**simple_guess**(*geom*)

Default force constants.

pysisyphus.optimizers.guess_hessians.**swart_guess**(*geom*)

pysisyphus.optimizers.guess_hessians.**ts_hessian**(*hessian*, *coord_inds*, *damp=0.25*)

According to [3]

pysisyphus.optimizers.guess_hessians.**xtb_hessian**(*geom*, *gfn=None*)

pysisyphus.optimizers.hessian_updates module

pysisyphus.optimizers.hessian_updates.**bfgs_update**(*H*, *dx*, *dg*)

pysisyphus.optimizers.hessian_updates.**bofill_update**(*H*, *dx*, *dg*)

pysisyphus.optimizers.hessian_updates.**damped_bfgs_update**(*H*, *dx*, *dg*)

See [5]

pysisyphus.optimizers.hessian_updates.**double_damp**(*s*, *y*, *H=None*, *s_list=None*, *y_list=None*, *mu_1=0.2*, *mu_2=0.2*, *logger=None*)

Double damped step 's' and gradient differences 'y'.

H is the inverse Hessian! See [6]. Potentially updates s and y. y is only updated if mu_2 is not None.

Parameters

- **s** (*np.array, shape (N,), floats*) -- Coordiante differences/step.
- **y** (*np.array, shape (N,), floats*) -- Gradient differences
- **H** (*np.array, shape (N, N), floats, optional*) -- Inverse Hessian.
- **s_list** (*list of nd.array, shape (K, N), optional*) -- List of K previous steps. If no H is supplied and prev_ys is given the matrix-vector product Hy will be calculated through the two-loop LBFGS-recursion.
- **y_list** (*list of nd.array, shape (K, N), optional*) -- List of K previous gradient differences. See s_list.
- **mu_1** (*float, optional*) -- Parameter for 's' damping.
- **mu_2** (*float, optional*) -- Parameter for 'y' damping.
- **logger** (*logging.Logger, optional*) -- Logger to be used.

Returns

- **s** (*np.array, shape (N,), floats*) -- Damped coordiante differences/step.
- **y** (*np.array, shape (N,), floats*) -- Damped gradient differences

pysisyphus.optimizers.hessian_updates.**flowchart_update**(*H*, *dx*, *dg*)

pysisyphus.optimizers.hessian_updates.**mod_flowchart_update**(*H*, *dx*, *dg*)

pysisyphus.optimizers.hessian_updates.**psb_update**(*z*, *dx*)

pysisyphus.optimizers.hessian_updates.**sr1_update**(*z*, *dx*)

pysisyphus.optimizers.hessian_updates.**ts_bfgs_update**(*H*, *dx*, *dg*)

As described in [7]

pysisyphus.optimizers.hessian_updates.**ts_bfgs_update_org**(*H*, *dx*, *dg*)

Do not use! Implemented as described in the 1998 bofill paper [8].

This does not seem to work too well.

pysisyphus.optimizers.hessian_updates.**ts_bfgs_update_revised**(*H*, *dx*, *dg*)

TS-BFGS update as described in [9].

Better than the original formula of Bofill, worse than the implementation in [7]. a is caluclated as described in the footnote 1 on page 38. Eq. (8) looks suspicious as it contains the inverse of a vector?! As also outlined in the paper abs(a) is used (**|a|** in the paper).

pysisyphus.optimizers.poly_fit module

class pysisyphus.optimizers.poly_fit.**FitResult**(*x*, *y*, *polys*)

Bases: tuple

polys

Alias for field number 2

x

Alias for field number 0

y

Alias for field number 1

pysisyphus.optimizers.poly_fit.**cubic_fit**(*e0*, *e1*, *g0*, *g1*)

pysisyphus.optimizers.poly_fit.**gen_solutions**()

Given two energies (e0, e1) and corresponding gradients (g0, g1) we can (try to) fit a quartic polynomial

f(x) = a0 + a1*x + a2*x**2 + a3*x**3 + a4*x**4

s.t. the constraint f''(x) >= 0, with the equality being fullfilled at only one point. There are five unknowns (a0 - a4) to be determined. Four equations can be derived from f(x) and its first derivative

f'(x) = a1 + 2*a2*x + 3*a3*x**2 + 4*a4*x**3 .

With (e0, g0) being given at x=0 and (e1, g1) being given at x=1 we can setup the following equations:

f (0) = a0 (1) f'(0) = a1 (2)

using e0 and g0 at x=0, and
f (1) = a0 + a1 + a2 + a3 + a4 (3) f'(1) = a1 + 2*a2 + 3*a3 + 4*a4 . (4)

The missing last equation can be derived from the constraint. The second derivative of f(x) is

f''(x) = 2*a2 + 6*a3*x + 12*a4*x**2

and shall be positive except at one point where it is allowed to be 0, that its two roots (f''(x) = 0) must be degenerate. This is fullfilled when the discriminant D of the quadratic polynomial a*x**2 + b*x + c is zero.

D = b**2 – 4*a*c = 0

With
a = 12*a4 b = 6*a3 c = 2*a2

we get
0 = (6*a3)**2 - 4*12*a4*2*a2 0 = 36*a3**2 - 96*a4*a2 0 = 3*a3**2 - 8*a4*a2 (5) or a4 = 3/8 * a3**2 / a2

Using (1) - (5) we can solve the set of equations for a0 - a4.

pysisyphus.optimizers.poly_fit.**get_maximum**(*poly*)

pysisyphus.optimizers.poly_fit.**get_minimum**(*poly*)

pysisyphus.optimizers.poly_fit.**poly_line_search**(*cur_energy*, *prev_energy*, *cur_grad*, *prev_grad*, *prev_step*, *cubic_max_x=2.0*, *quartic_max_x=4.0*, *logger=None*)

Generate directional gradients by projecting them on the previous step.

pysisyphus.optimizers.poly_fit.**quartic_fit**(*e0*, *e1*, *g0*, *g1*, *maximize=False*)

See gen_solutions() for derivation.

pysisyphus.optimizers.poly_fit.**quintic_fit**(*e0*, *e1*, *g0*, *g1*, *H0*, *H1*)

pysisyphus.optimizers.precon module

pysisyphus.optimizers.precon.**get_lindh_k**(*atoms*, *coords3d*, *bonds=None*, *angles=None*, *torsions=None*)

pysisyphus.optimizers.precon.**get_lindh_precon**(*atoms*, *coords*, *bonds=None*, *bends=None*, *dihedrals=None*, *c_stab=0.0103*, *logger=None*)

c_stab = 0.00103 hartree/bohr^2 corresponds to 0.1 eV/Å^2 as given in the paper.

pysisyphus.optimizers.precon.**precon_getter**(*geom*, *c_stab=0.0103*, *kind='full'*, *logger=None*)

pysisyphus.optimizers.restrict_step module

pysisyphus.optimizers.restrict_step.**get_scale_max**(*max_element*)

pysisyphus.optimizers.restrict_step.**restrict_step**(*steps*, *max_step*)

pysisyphus.optimizers.restrict_step.**scale_by_max_step**(*steps*, *max_step*)

Module contents

class pysisyphus.optimizers.**CubicNewton**(*geometry*, ***kwargs*)

Bases: *HessianOptimizer*

optimize()

postprocess_opt()

class pysisyphus.optimizers.**MicroOptimizer**(*geom*, *step='lbfgs'*, *line_search=True*, *max_cycles=100000000*, *max_step=0.2*, *keep_last=10*, *rms_force=None*, *double_damp=True*, *dump=False*, ***kwargs*)

Bases: `object`

cg_step(*forces*)

lbfgs_step(*forces*)

log(*msg*)

optimize()

run()

sd_step(*forces*)

take_step(*energy*, *forces*, *return_step=False*)

17.1.1.14 pysisyphus.plotters package

Submodules

pysisyphus.plotters.AnimPlot module

class pysisyphus.plotters.AnimPlot.**AnimPlot**(*calculator*, *optimizer*, *xlim=None*, *ylim=None*, *levels=None*, *num=100*, *figsize=(8, 8)*, *interval=250*, *energy_profile=True*, *colorbar=True*, *save=None*, *title=True*, *tight_layout=False*)

Bases: object

animate()

as_html5(*out_fn*)

func(*frame*)

on_keypress(*event*)

Pause on SPACE press.

Module contents

17.1.1.15 pysisyphus.stocastic package

Submodules

pysisyphus.stocastic.FragmentKick module

class pysisyphus.stocastic.FragmentKick.**FragmentKick**(*geom*, *fragments*, *fix_fragments=None*, *displace_from=None*, *random_displacement=False*, ***kwargs*)

Bases: *Kick*

get_fragments(*fragments*)

get_input_geom(*geom*)

get_origin()

kick_fragment(*frag_coords*)

print_fragments()

pysisyphus.stocastic.Kick module

class pysisyphus.stocastic.Kick.**Kick**(*geom*, *radius=0.5*, ***kwargs*)

Bases: *Pipeline*

get_input_geom(*geom*)

get_kick()

run_cycle(*geom*)

pysisyphus.stocastic.Pipeline module

class pysisyphus.stocastic.Pipeline.**Pipeline**(*geom*, *calc_getter=None*, *seed=None*, *max_cycles=5*, *cycle_size=15*, *rmsd_thresh=0.1*, *energy_thresh=0.001*, *energy_range=0.125*, *compare_num=25*, *break_after=2*, *calc_kwargs=None*)

Bases: object

atoms_are_too_close(*geom*, *factor=0.7*)

Determine if atoms are too close.

geom_is_close_in_energy(*geom*)

geom_is_new(*geom*)

Determine if geometry is not already known.

geom_is_valid(*geom*)

Filter out geometries that are None, or were the atoms are too close or when they are already known.

get_input_geom(*geom*)

get_unique_geometries(*geoms*)

get_valid_index_set(*to_intersect*)

log(*message*)

Write a log message.

Wraps the logger variable.

Parameters

message (*str*) -- Message to be logged.

run()

run_geom_opt(*geom*)

write_geoms_to_trj(*geoms*, *fn*, *comments=None*)

pysisyphus.stocastic.align module

pysisyphus.stocastic.align.**apply_transform**(*coords3d*, *swap*, *reflection*)

Apply axis swaps and reflections to a coordinate array.

pysisyphus.stocastic.align.**match_geom_atoms**(*ref_geom*, *geom_to_match*, *hydrogen=True*)

Apply the hungarian method to geom_to_match.

Uses the scipy implementation of the Hungarian method/ Kuhn-Munkres algorithm in scipy.optimize.linear_sum_assignment.

See

[1] 10.1021/ci400534h [2] 10.1021/acs.jcim.6b00516

pysisyphus.stocastic.align.**matched_rmsd**(*geom1*, *geom2*, *thresh=0.05*)

RMSD for optimally aligned and matched geometries.

Returns

- **matched_rmsd** (*float*) -- RMSD of optimally aligned and matched geometries.
- **matched_geoms** (*tuple(Geometry, Geometry)*) -- Tuple of the optimally aligned and matched geometries.

Module contents

class pysisyphus.stocastic.**FragmentKick**(*geom*, *fragments*, *fix_fragments=None*, *displace_from=None*, *random_displacement=False*, ***kwargs*)

Bases: *Kick*

get_fragments(*fragments*)

get_input_geom(*geom*)

get_origin()

kick_fragment(*frag_coords*)

print_fragments()

class pysisyphus.stocastic.**Kick**(*geom*, *radius=0.5*, ***kwargs*)

Bases: *Pipeline*

get_input_geom(*geom*)

get_kick()

run_cycle(*geom*)

17.1.1.16 pysisyphus.tests package

Submodules

pysisyphus.tests.test_calculators module

Module contents

17.1.1.17 pysisyphus.tsoptimizers package

Submodules

pysisyphus.tsoptimizers.RSIRFOptimizer module

class pysisyphus.tsoptimizers.RSIRFOptimizer.**RSIRFOptimizer**(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Bases: *TSHessianOptimizer*

optimize()

pysisyphus.tsoptimizers.RSPRFOptimizer module

class pysisyphus.tsoptimizers.RSPRFOptimizer.**RSPRFOptimizer**(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Bases: *TSHessianOptimizer*

optimize()

pysisyphus.tsoptimizers.TRIM module

class pysisyphus.tsoptimizers.TRIM.**TRIM**(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Bases: *TSHessianOptimizer*

optimize()

pysisyphus.tsoptimizers.TSHessianOptimizer module

class pysisyphus.tsoptimizers.TSHessianOptimizer.**TSHessianOptimizer**(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Bases: *HessianOptimizer*

Optimizer to find first-order saddle points.

__init__(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Baseclass for transition state optimizers utilizing Hessian information.

Several arguments expect a typed primitive or an iterable of typed primitives. A typed primitive is specified as (PrimType, int, int, ...), e.g., for a bond between atoms 0 and 1: (BOND, 0, 1) or for a bend between the atom triple 0, 1, 2 as (BEND, 0, 1, 2).

Parameters

- **geometry** (*Geometry*) -- Geometry to be optimized.
- **roots** (Optional[List[int]], default: None) -- Indices of modes to maximize along, e.g., to optimize saddle points of 2nd order. Overrides 'root'.
- **root** (int, default: 0) -- Index of imaginary mode to maximize along. Shortcut for 'roots' with only one root.

- **hessian_ref** (Optional[str], default: None) -- Filename pointing to a pysisyphus HDF5 Hessian.
- **rx_modes** (*iterable of (iterable of (typed_prim, phase_factor))*) -- Select initial root(s) by overlap with a modes constructed from the given typed primitives with respective phase factors.
- **prim_coord** (*typed_prim*) -- Select initial root/imaginary mode by overlap with this internal coordinate. Shortcut for 'rx_modes' with only one internal coordinate.
- **rx_coords** (*iterable of (typed_prim)*) -- Construct imaginary mode comprising the given typed prims by modifying a model Hessian.
- **hessian_init** (Literal['calc', 'unit', 'fischer', 'lindh', 'simple', 'swart', 'xtb', 'xtb1', 'xtbff'], default: 'calc') -- Type of initial model Hessian.
- **hessian_update** (Literal['none', None, False, 'bfgs', 'damped_bfgs', 'flowchart', 'bofill', 'ts_bfgs', 'ts_bfgs_org', 'ts_bfgs_rev'], default: 'bofill') -- Type of Hessian update. Defaults to BFGS for minimizations and Bofill for saddle point searches.
- **hessian_recalc_reset** (bool, default: True) -- Whether to skip Hessian recalculation after reset. Undocumented.
- **max_micro_cycles** (int, default: 50) -- Maximum number of RS iterations.
- **trust_radius** (float, default: 0.3) -- Initial trust radius in whatever unit the optimization is carried out.
- **trust_max** (float, default: 0.5) -- Maximum trust radius.
- **augment_bonds** (bool, default: False) -- Try to derive additional streching coordinates from the imaginary mode.
- **min_line_search** (bool, default: False) -- Carry out line search along the imaginary mode.
- **max_line_search** (bool, default: False) -- Carry out line search in the subspace that is minimized.
- **assert_neg_eigval** (bool, default: False) -- Check for the existences for at least one significant negative eigenvalue. If enabled and no negative eigenvalue is present the optimization will be aborted.
- ****kwargs** -- Keyword arguments passed to the HessianOptimizer/Optimizer baseclass.

static do_line_search(*e0*, *e1*, *g0*, *g1*, *prev_step*, *maximize*, *logger=None*)

prepare_opt(**args*, ***kwargs*)

property root

property roots

step_and_grad_from_line_search(*energy*, *gradient_trans*, *eigvecs*, *min_indices*, *max_indices*)

update_ts_mode(*eigvals*, *eigvecs*)

valid_updates = ('bofill', 'ts_bfgs', 'ts_bfgs_org', 'ts_bfgs_rev')

Module contents

class pysisyphus.tsoptimizers.**RSIRFOptimizer**(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Bases: *TSHessianOptimizer*

optimize()

class pysisyphus.tsoptimizers.**RSPRFOptimizer**(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Bases: *TSHessianOptimizer*

optimize()

class pysisyphus.tsoptimizers.**TRIM**(*geometry*, *roots=None*, *root=0*, *hessian_ref=None*, *rx_modes=None*, *prim_coord=None*, *rx_coords=None*, *hessian_init='calc'*, *hessian_update='bofill'*, *hessian_recalc_reset=True*, *max_micro_cycles=50*, *trust_radius=0.3*, *trust_max=0.5*, *augment_bonds=False*, *min_line_search=False*, *max_line_search=False*, *assert_neg_eigval=False*, ***kwargs*)

Bases: *TSHessianOptimizer*

optimize()

17.1.1.18 pysisyphus.wrapper package

Submodules

pysisyphus.wrapper.jmol module

pysisyphus.wrapper.jmol.**call_jmol**(*spt_str*, *show=False*)

pysisyphus.wrapper.jmol.**render_cdd_cube**(*cdd_cube*, *isoval=0.001*, *orient=''*)

pysisyphus.wrapper.jmol.**render_geom_and_charges**(*geom*, *point_charges*)

pysisyphus.wrapper.jmol.**view_cdd_cube**(*cdd_cube*, *isoval=0.001*, *orient=''*)

pysisyphus.wrapper.mwfn module

pysisyphus.wrapper.mwfn.**call_mwfn**(*inp_fn*, *stdin*, *cwd=None*)

pysisyphus.wrapper.mwfn.**get_mwfn_exc_str**(*energies*, *Xa*, *Ya=None*, *Xb=None*, *Yb=None*, *thresh=0.001*)

Write plain text input for MWFN according to 3.21.

As of version 3.8 MWFN does not seem to handle this file when spin labels are present, even though it is given as an example in the manual.

pysisyphus.wrapper.mwfn.**log**(*msg*)

pysisyphus.wrapper.mwfn.**make_cdd**(*inp_fn*, *state*, *log_fn*, *cwd=None*, *keep=False*, *quality=2*, *prefix='S'*)

Create CDD cube in cwd.

Parameters

- **inp_fn** (*str*) -- Filename of a .molden/.fchk file.
- **state** (*int*) -- CDD cubes will be generated up to this state.
- **log_fn** (*str*) -- Filename of the .log file.
- **cwd** (*str or Path, optional*) -- If a different cwd should be used.
- **keep** (*bool*) -- Wether to keep electron.cub and hole.cub, default is False.
- **quality** (*int*) -- Quality of the cube. (1=low, 2=medium, 3=high).

pysisyphus.wrapper.mwfn.**wrap_stdin**(*stdin*)

pysisyphus.wrapper.packmol module

pysisyphus.wrapper.packmol.**call_packmol**(*inp*)

pysisyphus.wrapper.packmol.**make_input**(*output_fn*, *solvent_fn*, *solvent_num*, *sphere_radius*, *solute_fn=None*, *solute_num=None*, *tolerance=2.0*)

Module contents

17.1.2 Submodules

17.1.3 pysisyphus.Geometry module

class pysisyphus.Geometry.**Geometry**(*atoms*, *coords*, *fragments=None*, *coord_type='cart'*, *coord_kwargs=None*, *isotopes=None*, *freeze_atoms=None*, *comment=''*, *name=''*)

Bases: object

__init__(*atoms*, *coords*, *fragments=None*, *coord_type='cart'*, *coord_kwargs=None*, *isotopes=None*, *freeze_atoms=None*, *comment=''*, *name=''*)

Object representing atoms in a coordinate system.

The Geometry represents atoms and their positions in coordinate system. By default cartesian coordinates are used, but internal coordinates are also possible.

Parameters

- **atoms** (*iterable*) -- Iterable of length N, containing element symbols.
- **coords** (*1d iterable*) -- 1d iterable of length 3N, containing the cartesian coordinates of N atoms.
- **fragments** (*dict, optional*) -- Dict with different keys denoting different fragments. The values contain lists of atom indices.
- **coord_type** (*{"cart", "redund"}, optional*) -- Type of coordinate system to use. Right now cartesian (cart) and redundand (redund) are supported.
- **coord_kwargs** (*dict, optional*) -- Dictionary containing additional arguments that get passed to the constructor of the internal coordinate class.
- **isotopes** (*iterable of pairs, optional*) -- Iterable of pairs consisting of 0-based atom index and either an integer or a float. If an integer is given the closest isotope mass will be selected. Given a float, this float will be directly used as mass.
- **freeze_atoms** (*iterable of integers*) -- Specifies which atoms should remain fixed at their initial positions.
- **comment** (*str, optional*) -- Comment string.
- **name** (*str, optional*) -- Verbose name of the geometry, e.g. methanal or water. Used for printing

align_principal_axes()

Align the principal axes to the cartesian axes.

https://math.stackexchange.com/questions/145023

property all_energies

Return energies of all states that were calculated.

This will also set self.energy, which may NOT be the ground state, but the state correspondig to the 'root' attribute of the calculator.

approximate_radius()

Approximate molecule radius from the biggest atom distance along an axis.

as_ase_atoms()

as_g98_list()

Returns data for fake Gaussian98 standard orientation output.

Returns

g98_list -- List with one row per atom. Every row contains [center number, atomic number, atomic type (always 0 for now), X Y Z coordinates in Angstrom.

Return type

list

as_xyz(*comment=''*, *atoms=None*, *cart_coords=None*)

Current geometry as a string in XYZ-format.

Parameters

- **comment** (*str, optional*) -- Will be written in the second line (comment line) of the XYZ-string.
- **cart_coords** (*np.array, 1d, shape (3 * atoms.size,)*) -- Cartesians for dumping instead of self._coords.

Returns
xyz_str -- Current geometry as string in XYZ-format.

Return type
str

`assert_cart_coords`(*coords*)

`assert_compatibility`(*other*)

Assert that two Geometries can be substracted from each other.

Parameters
other (Geometry) -- Geometry for comparison.

`atom_indices`()

Dict with atom types as key and corresponding indices as values.

Returns
inds_dict -- Unique atom types as keys, corresponding indices as values.

Return type
dict

`property atom_types`

`atom_xyz_iter`()

`property atomic_numbers`

`property bond_sets`

`calc_double_ao_overlap`(*geom2*)

`calc_energy_and_forces`()

Force a calculation of the current energy and forces.

`property cart_coords`

`property cart_forces`

`property cart_gradient`

`property cart_hessian`

`center`()

`property center_of_mass`

Returns the center of mass.

Returns
R -- Center of mass.

Return type
np.array, shape(3,)

`center_of_mass_at`(*coords3d*)

Returns the center of mass at given coords3d.

Parameters
coords3d (*np.array, shape(N, 3)*) -- Cartesian coordiantes.

Returns
R -- Center of mass.

Return type
np.array, shape(3,)

property centroid

Geometric center of the Geometry.

Returns
R -- Geometric center of the Geometry.

Return type
np.array, shape(3,)

clear()

Reset the object state.

property comment

coord_types = {'cart': None, 'cartesian': <class 'pysisyphus.intcoords.CartesianCoords.CartesianCoords'>, 'dlc': <class 'pysisyphus.intcoords.DLC.DLC'>, 'hdlc': <class 'pysisyphus.intcoords.DLC.HDLC'>, 'hredund': <class 'pysisyphus.intcoords.RedundantCoords.HybridRedundantCoords'>, 'mwcartesian': <class 'pysisyphus.intcoords.CartesianCoords.MWCartesianCoords'>, 'redund': <class 'pysisyphus.intcoords.RedundantCoords.RedundantCoords'>, 'tmtric': <class 'pysisyphus.intcoords.RedundantCoords.TMTRIC'>, 'tric': <class 'pysisyphus.intcoords.RedundantCoords.TRIC'>}

property coords

1d vector of atomic coordinates.

Returns
coords -- 1d array holding the current coordinates.

Return type
np.array

property coords3d

Coordinates in 3d.

Returns
coords3d -- Coordinates of the Geometry as 2D array.

Return type
np.array

property coords_by_type

Coordinates in 3d by atom type and their corresponding indices.

Returns

- **cbt** (*dict*) -- Dictionary with the unique atom types of the Geometry as keys. It's values are the 3d coordinates of the corresponding atom type.
- **inds** (*dict*) -- Dictionary with the unique atom types of the Geometry as keys. It's values are the original indices of the 3d coordinates in the whole coords3d array.

copy(*coord_type=None*, *coord_kwargs=None*)

Returns a new Geometry object with same atoms and coordinates.

Parameters

- **coord_type** (*str*) -- Desired coord_type, defaults to current coord_type.
- **coord_kwargs** (*dict, optional*) -- Any desired coord_kwargs that will be passed to the RedundantCoords object.

Returns
geom -- New Geometry object with the same atoms and coordinates.

Return type
Geometry

copy_all(*coord_type=None*, *coord_kwargs=None*)

property covalent_radii

del_atoms(*inds*, ***kwargs*)

describe()

dump_xyz(*fn*, *cart_coords=None*, ***kwargs*)

eckart_projection(*mw_hessian*, *return_P=False*, *full=False*)

property energy

Energy of the current atomic configuration.

Returns
energy -- Energy of the current atomic configuration.

Return type
float

fd_coords3d_gen(*step_size=0.001*)

Iterator returning 3d Cartesians for finite-differences.

property forces

Energy of the current atomic configuration.

Returns
force -- 1d array containing the forces acting on the atoms. Negative of the gradient.

Return type
np.array

get_energy_and_cart_forces_at(*cart_coords*)

get_energy_and_cart_hessian_at(*cart_coords*)

get_energy_and_forces_at(*coords*)

Calculate forces and energies at the given coordinates.

The results are not saved in the Geometry object.

get_energy_at(*coords*)

get_energy_at_cart_coords(*cart_coords*)

get_fragments(*regex*)

get_imag_frequencies(*hessian=None*, *thresh=1e-06*)

get_normal_modes(*cart_hessian=None*, *full=False*)

Normal mode wavenumbers, eigenvalues and Cartesian displacements Hessian.

get_restart_info()

get_sphere_radius(*offset=4*)

get_subgeom(*indices*, *coord_type='cart'*, *sort=False*)

Return a Geometry containing a subset of the current Geometry.

Parameters

- **indices** (*iterable of ints*) -- Atomic indices that the define the subset of the current Geometry.
- **coord_type** (*str, ("cart", "redund"), optional*) -- Coordinate system of the new Geometry.

Returns

sub_geom -- Subset of the current Geometry.

Return type

Geometry

get_subgeom_without(*indices*, ***kwargs*)

get_temporary_coords(*coords*)

get_thermoanalysis(*energy=None*, *cart_hessian=None*, *T=298.15*, *p=101325*, *point_group='c1'*)

get_trans_rot_projector(*full=False*)

property gradient

Negative of the force.

Returns

gradient -- 1d array containing the negative of the current forces.

Return type

np.array

property hessian

Matrix of second derivatives of the energy in respect to atomic displacements.

Returns

hessian -- 2d array containing the second derivatives of the energy with respect to atomic/coordinate displacements depending on the type of coordiante system.

Return type

np.array

property inertia_tensor

property is_analytical_2d

jmol(*atoms=None*, *cart_coords=None*)

Show geometry in jmol.

property layers

mass_weigh_hessian(*hessian*)

property masses

property masses_rep

property mm_inv

Inverted mass matrix.

Returns a diagonal matrix containing the inverted atomic masses.

property mm_sqrt_inv

Inverted square root of the mass matrix.

modes3d()

property moving_atoms

moving_atoms_jmol()

property mw_coords

Mass-weighted coordinates.

Returns
mw_coords -- 1d array containing the mass-weighted cartesian coordiantes.

Return type
np.array

property mw_gradient

Mass-weighted gradient.

Returns
mw_gradient -- Returns the mass-weighted gradient.

Return type
np.array

property mw_hessian

Mass-weighted hessian.

Returns
mw_hessian -- 2d array containing the mass-weighted hessian M^(-1/2) H M^(-1/2).

Return type
np.array

principal_axes_are_aligned()

Check if the principal axes are aligned with the cartesian axes.

Returns
aligned -- Wether the principal axes are aligned or not.

Return type
bool

reparametrize()

reset_coords(*new_typed_prims=None*)

rmsd(*geom*)

rotate(*copy=False*, *rng=None*)

set_calculator(*calculator*, *clear=True*)

Reset the object and set a calculator.

set_coord(*ind*, *coord*)

Set a coordinate by index.

Parameters

- **ind** (*int*) -- Index in of the coordinate to set in the self.coords array.
- **coord** (*float*) -- Coordinate value.

set_coords(*coords*, *cartesian=False*, *update_constraints=False*)

set_h5_hessian(*fn*)

set_restart_info(*restart_info*)

set_results(*results*)

Save the results from a dictionary.

Parameters

results (*dict*) -- The keys in this dict will be set as attributes in the current object, with the corresponding item as value.

standard_orientation()

property sum_formula

tmp_xyz_handle(*atoms=None*, *cart_coords=None*)

property total_mass

unweight_mw_hessian(*mw_hessian*)

Unweight a mass-weighted hessian.

Parameters

mw_hessian (*np.array*) -- Mass-weighted hessian to be unweighted.

Returns

hessian -- 2d array containing the hessian.

Return type

np.array

property vdw_radii

vdw_volume(***kwargs*)

without_hydrogens()

zero_frozen_forces(*cart_forces*)

pysisyphus.Geometry.**get_trans_rot_projector**(*cart_coords*, *masses*, *full=False*)

pysisyphus.Geometry.**get_trans_rot_vectors**(*cart_coords*, *masses*, *rot_thresh=1e-06*)

Vectors describing translation and rotation.

These vectors are used for the Eckart projection by constructing a projector from them.

See Martin J. Field - A Pratcial Introduction to the simulation of Molecular Systems, 2007, Cambridge University Press, Eq. (8.23), (8.24) and (8.26) for the actual projection.

See also https://chemistry.stackexchange.com/a/74923.

Parameters

- **cart_coords** (*np.array, 1d, shape (3 * atoms.size,)*) -- Atomic masses in amu.
- **masses** (*iterable, 1d, shape (atoms.size,)*) -- Atomic masses in amu.

Returns
ortho_vecs -- 2d array containing row vectors describing translations and rotations.

Return type
np.array(6, 3*atoms.size)

pysisyphus.Geometry.**inertia_tensor**(*coords3d*, *masses*)

Inertita tensor.

x^2 xy xz |

(x y z)^T . (x y z) = | xy y^2 yz |

xz yz z^2 |

17.1.4 pysisyphus.TableFormatter module

class pysisyphus.TableFormatter.**TableFormatter**(*header*, *fmts*, *min_width=7*, *space=3*)

Bases: object

property header

join_format(*fmts*, *lst*)

Format a given iterable lst with formats given in the iterable lst and return the joined items of the formatted list.

line(**args*)

min_width_fmts(*raw_fmts=None*)

pysisyphus.TableFormatter.**run**()

17.1.5 pysisyphus.TablePrinter module

class pysisyphus.TablePrinter.**TablePrinter**(*header*, *col_fmts*, *width=12*, *sub_underline=True*, *shift_left=0*, *fmts_update=None*, *mark='*'*)

Bases: object

print(**args*, ***kwargs*)

print_header(*with_sep=True*)

print_row(*args*, *marks=None*)

print_sep()

17.1.6 pysisyphus.color module

pysisyphus.color.**bool_color**(*bool_*, *str_=None*)

pysisyphus.color.**green**(*str_*)

pysisyphus.color.**red**(*str_*)

17.1.7 pysisyphus.config module

pysisyphus.config.**detect_paths**()

pysisyphus.config.**get_cmd**(*section*, *key='cmd'*, *use_defaults=True*)

pysisyphus.config.**parse_args**(*args*)

pysisyphus.config.**run_detect_paths**()

17.1.8 pysisyphus.constants module

17.1.9 pysisyphus.elem_data module

pysisyphus.elem_data.**get_tm_indices**(*atoms*)

pysisyphus.elem_data.**nuc_charges_for_atoms**(*atoms*)

17.1.10 pysisyphus.exceptions module

exception pysisyphus.exceptions.**HEIIsFirstOrLastException**

Bases: Exception

17.1.11 pysisyphus.filtertrj module

pysisyphus.filtertrj.**get_unique_internals**(*geom*)

pysisyphus.filtertrj.**parse_args**(*args*)

pysisyphus.filtertrj.**parse_filter**(*raw_filter*)

pysisyphus.filtertrj.**run**()

17.1.12 pysisyphus.helpers module

class pysisyphus.helpers.**FinalHessianResult**(*neg_eigvals*, *eigvals*, *nus*, *imag_fns*, *thermo*)

Bases: tuple

eigvals

Alias for field number 1

imag_fns

Alias for field number 3

neg_eigvals

Alias for field number 0

nus

Alias for field number 2

thermo

Alias for field number 4

pysisyphus.helpers.**align_coords**(*coords_list*)

pysisyphus.helpers.**align_geoms**(*geoms*)

pysisyphus.helpers.**check_for_end_sign**(*check_user=True*, *cwd='.'*, *add_signs=None*)

pysisyphus.helpers.**complete_fragments**(*atoms*, *fragments*)

pysisyphus.helpers.**confirm_input**(*message*)

pysisyphus.helpers.**do_final_hessian**(*geom*, *save_hessian=True*, *write_imag_modes=False*, *is_ts=False*, *prefix=''*, *T=298.15*, *p=101325*, *ev_thresh=-1e-06*, *print_thermo=False*, *out_dir=None*)

pysisyphus.helpers.**fit_rigid**(*geometry*, *vectors=None*, *vector_lists=None*, *hessian=None*, *align_factor=1.0*)

pysisyphus.helpers.**geom_from_xyz_file**(*xyz_fn*, *coord_type='cart'*, ***coord_kwargs*)

pysisyphus.helpers.**geom_loader**(*fn*, *coord_type='cart'*, *iterable=False*, ***coord_kwargs*)

After introducing the pubchem functionality I don't like this function anymore :) Too complicated.

pysisyphus.helpers.**geoms_from_trj**(*trj_fn*, *first=None*, *coord_type='cart'*, ***coord_kwargs*)

pysisyphus.helpers.**get_coords_diffs**(*coords*, *align=False*, *normalize=True*)

pysisyphus.helpers.**get_geom_getter**(*ref_geom*, *calc_setter*)

pysisyphus.helpers.**get_tangent_trj_str**(*atoms*, *coords*, *tangent*, *comment=None*, *points=10*, *displ=None*)

pysisyphus.helpers.**imag_modes_from_geom**(*geom*, *freq_thresh=-10*, *points=10*, *displ=None*)

pysisyphus.helpers.**index_array_from_overlaps**(*overlaps*, *axis=1*)

It is assumed that the overlaps between two points with indices i and j with (j > i) are computed and that i changes along the first axis (axis=0) and j changes along the second axis (axis=1).

So the first row of the overlap matrix (overlaps[0]) should contain the overlaps between state 0 at index i and all states at index j.

argmax along axis 1 returns the indices of the most overlapping states at index j with the states at index i, given by the item index in the indices array. E.g.:

[0 1 3 2] indicates a root flip in a system with four states when going from index i to index j. Root 2 at i became root 3 at j and vice versa.

pysisyphus.helpers.**match_geoms**(*ref_geom*, *geom_to_match*, *hydrogen=False*)

See

[1] 10.1021/ci400534h [2] 10.1021/acs.jcim.6b00516

pysisyphus.helpers.**norm_max_rms**(*arr*)

pysisyphus.helpers.**np_print**(*func*, *precision=2*, *suppress=True*, *linewidth=120*)

pysisyphus.helpers.**pick_image_inds**(*cart_coords*, *images*)

Pick approx. evenly distributed images from given Cartesian coordinates.

pysisyphus.helpers.**print_barrier**(*ref_energy*, *comp_energy*, *ref_str*, *comp_str*)

pysisyphus.helpers.**procrustes**(*geometry*, *align_factor=1.0*)

pysisyphus.helpers.**shake_coords**(*coords*, *scale=0.1*, *seed=None*)

pysisyphus.helpers.**simple_peaks**(*data*)

pysisyphus.helpers.**slugify_worker**(*dask_worker*)

17.1.13 pysisyphus.helpers_pure module

class pysisyphus.helpers_pure.**OrderedEnum**(*value*)

Bases: Enum

An enumeration.

pysisyphus.helpers_pure.**approx_float**(*num*, *expected*, *abs_tol=1e-06*, *rel_tol=1e-12*)

Return type
bool

pysisyphus.helpers_pure.**cache_arrays**(*sources*, *dest*)

pysisyphus.helpers_pure.**check_mem**(*mem*, *pal*, *avail_frac=0.85*, *logger=None*)

pysisyphus.helpers_pure.**chunks**(*l*, *n*)

Yield successive n-sized chunks from l. https://stackoverflow.com/a/312464

pysisyphus.helpers_pure.**describe**(*arr*)

pysisyphus.helpers_pure.**eigval_to_wavenumber**(*ev*)

pysisyphus.helpers_pure.**estimate**(*gen*, *elems*)

pysisyphus.helpers_pure.**expand**(*to_expand*)

pysisyphus.helpers_pure.**file_or_str**(**args*, *method=False*, *mode='r'*, *exact=False*)

pysisyphus.helpers_pure.**filter_fixture_store**(*test_name*)

pysisyphus.helpers_pure.**find_closest_sequence**(*str_*, *comp_strs*)

Return type
Tuple[str, float]

pysisyphus.helpers_pure.**full_expand**(*to_expand*)

pysisyphus.helpers_pure.**get_box**(*coords3d*, *offset=0.0*)

pysisyphus.helpers_pure.**get_clock**()

pysisyphus.helpers_pure.**get_cubic_crystal**(*l*, *n=2*, *atom='Na'*)

pysisyphus.helpers_pure.**get_input**(*data*, *prompt*, *lbl_func=None*)

pysisyphus.helpers_pure.**get_molecular_radius**(*coords3d*, *min_offset=0.9452*)

pysisyphus.helpers_pure.**get_random_path**(*stem=''*)

pysisyphus.helpers_pure.**get_ratio**(*str_*, *comp_str*)

See https://stackoverflow.com/a/17388505

Return type
str

pysisyphus.helpers_pure.**hash_args**(**args*, *precision=4*)

pysisyphus.helpers_pure.**hash_arr**(*arr*, *precision=4*)

pysisyphus.helpers_pure.**highlight_text**(*text*, *width=80*, *level=0*)

pysisyphus.helpers_pure.**increment_fn**(*org_fn*, *suffix=None*)

Append, or increase a suffixed counter on a given filename. If no counter is present it will be set to zero. Otherwise it is incremented by one.

```
>>> increment_fn("opt", "rebuilt")
'opt_rebuilt_000'
>>> increment_fn("opt")
'opt_000'
>>> increment_fn("opt_rebuilt_000", "rebuilt")
'opt_rebuilt_001'
```

Parameters

- **org_fn** (str) -- The original, unaltered filename.
- **suffix** (Optional[str], default: None) -- Optional suffix to be append.

Returns
Modified filename with optional suffix and incremented counter.

Return type
incr_fn

pysisyphus.helpers_pure.**interpolate_colors**(*values*, *c1*, *c2*, *num=32*)

Expects two RGB colors c1 and c2.

pysisyphus.helpers_pure.**json_to_results**(*as_json*)

pysisyphus.helpers_pure.**kill_dir**(*path*)

Innocent function remove a directory.

It must contain only files and no other directories. So this won't do too much damage hopefully.

pysisyphus.helpers_pure.**log**(*logger*, *msg*, *level=10*)

pysisyphus.helpers_pure.**merge_sets**(*fragments*)

Merge a list of iterables.

pysisyphus.helpers_pure.**molecular_volume**(*coords3d*, *vdw_radii*, *n_trial=10000*, *offset=1.0*)

Monte-Carlo estimate of molecular volume using Van der Waals spheres. Cartesian coordinates and VdW-radii are expected in Bohr!

Return type
Tuple[float, float, float]

pysisyphus.helpers_pure.**recursive_update**(*d*, *u*)

Recursive update of d with keys/values from u.

From https://stackoverflow.com/questions/3232943

pysisyphus.helpers_pure.**remove_duplicates**(*seq*)

pysisyphus.helpers_pure.**report_frozen_atoms**(*geom*)

pysisyphus.helpers_pure.**report_isotopes**(*geom*, *affect_str*)

pysisyphus.helpers_pure.**results_to_json**(*results*)

pysisyphus.helpers_pure.**rms**(*arr*)

pysisyphus.helpers_pure.**sort_by_central**(*set1*, *set2*)

Determines a common index in two sets and returns a length 3 tuple with the central index at the middle position and the two terminal indices as first and last indices.

pysisyphus.helpers_pure.**standard_state_corr**(*T=298.15*, *p=101325*, *n=1*)

dG for change of standard state from gas to solution of 1 mol/l

pysisyphus.helpers_pure.**timed**(*logger=None*)

pysisyphus.helpers_pure.**to_sets**(*iterable*)

pysisyphus.helpers_pure.**to_subscript_num**(*num*)

pysisyphus.helpers_pure.**touch**(*fn*)

17.1.14 pysisyphus.init_logging module

pysisyphus.init_logging.**get_fh_logger**(*name*, *log_fn*)

Initialize a logger with 'name', level DEBUG and a FileHandler.

pysisyphus.init_logging.**init_logging**(*log_dir='./'*, *scheduler=None*)

Prepare the logger in log_path. When called with scheduler loggers for every worker are prepared.

pysisyphus.init_logging.**init_logging_base**(*dask_worker*, *log_path*)

Prepare individual loggers for one dask_worker.

17.1.15 pysisyphus.linalg module

pysisyphus.linalg.**cross3**(*a*, *b*)

10x as fast as np.cross for two 1d arrays of size 3.

pysisyphus.linalg.**eigvec_grad**(*w*, *v*, *ind*, *mat_grad*)

Gradient of 'ind'-th eigenvector.

dv_i / dx_i = (w_i*I - mat)[1] dmat/dx_i v_i

pysisyphus.linalg.**finite_difference_hessian**(*coords*, *grad_func*, *step_size=0.01*, *acc=2*, *callback=None*)

Numerical Hessian from central finite gradient differences.

Return type
ndarray[Any, dtype[float]]

See central differences in
https://en.wikipedia.org/wiki/Finite_difference_coefficient

for the different accuracies.

pysisyphus.linalg.**get_rot_mat**(*abc=None*)

pysisyphus.linalg.**get_rot_mat_for_coords**(*coords3d_1*, *coords3d_2*)

pysisyphus.linalg.**gram_schmidt**(*vecs*, *thresh=1e-08*)

pysisyphus.linalg.**make_unit_vec**(*vec1*, *vec2*)

Return unit vector pointing from vec2 to vec1.

pysisyphus.linalg.**matrix_power**(*mat*, *p*, *thresh=1e-12*, *strict=True*)

pysisyphus.linalg.**multi_component_sym_mat**(*arr*, *dim*)

pysisyphus.linalg.**norm3**(*a*)

5x as fas as np.linalg.norm for a 1d array of size 3.

pysisyphus.linalg.**orthogonalize_against**(*mat*, *vecs*, *max_cycles=5*, *thresh=1e-10*)

Orthogonalize rows of 'mat' against rows in 'vecs'

Returns a (modified) copy of mat.

pysisyphus.linalg.**perp_comp**(*vec*, *along*)

Return the perpendicular component of vec along along.

pysisyphus.linalg.**pivoted_cholesky**(*A*, *tol=-1.0*)

Cholesky factorization a real symmetric positive semidefinite matrix. Cholesky factorization is carried out with full pivoting.

Adapated from PySCF.

P.T * A * P = L * L.T

Parameters

- **A** (ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]) -- Matrix to be factorized.
- **tol** (float, default: -1.0) -- User defined tolerance, as outlined in the LAPACK docs.

Returns

- *L* -- Lower or upper triangular matrix.
- *piv* -- Pivot vectors, starting at 0.
- *rank* -- Rank of the factoirzed matrix.

pysisyphus.linalg.**pivoted_cholesky2**(*A*, *tol=None*, *m_max=0*)

https://doi.org/10.1016/j.apnum.2011.10.001

pysisyphus.linalg.**quaternion_to_rot_mat**(*q*)

pysisyphus.linalg.**rmsd_grad**(*coords3d*, *ref_coords3d*, *offset=1e-09*)

RMSD and gradient between two sets of coordinates from quaternions.

The gradient is given w.r.t. the coordinates of 'coords3d'.

Python adaption of
ls_rmsd.f90

from the xtb repository of the Grimme group, which in turn implements
[1] https://doi.org/10.1002/jcc.20110

.

Parameters

- **coords3d** (ndarray[Any, dtype[float]]) -- Coordinate array of shape (N, 3) with N denoting the number of atoms.
- **ref_coords3d** (ndarray[Any, dtype[float]]) -- Reference coordinates.
- **offset** (float, default: 1e-09) -- Small floating-point number that is added to the RMSD, to avoid division by zero.

Return type
Tuple[float, ndarray[Any, dtype[float]]]

Returns

- *rmsd* -- RMSD value.
- *rmsd_grad* -- Gradient of the RMSD value w.r.t. to 'coords3d'.

pysisyphus.linalg.**rot_quaternion**(*coords3d*, *ref_coords3d*)

pysisyphus.linalg.**svd_inv**(*array*, *thresh*, *hermitian=False*)

pysisyphus.linalg.**sym_mat_from_tril**(*arr*, *data*)

pysisyphus.linalg.**sym_mat_from_triu**(*arr*, *data*)

17.1.16 pysisyphus.pack module

pysisyphus.pack.**as_pdb**(*fn*)

pysisyphus.pack.**parse_args**(*args*)

pysisyphus.pack.**print_info**(*title*, *geom*)

pysisyphus.pack.**run**()

pysisyphus.pack.**sphere_radius_from_volume**(*volume*)

pysisyphus.pack.**volume_for_density**(*molecule_num*, *mol_mass*, *density*)

17.1.17 pysisyphus.peakdetect module

pysisyphus.peakdetect.**peakdetect**(*y_axis*, *x_axis=None*, *lookahead=200*, *delta=0*)

Converted from/based on a MATLAB script at: http://billauer.co.il/peakdet.html

function for detecting local maxima and minima in a signal. Discovers peaks by searching for values which are surrounded by lower or larger values for maxima and minima respectively

keyword arguments: y_axis -- A list containing the signal over which to find peaks

x_axis -- A x-axis whose values correspond to the y_axis list and is used
in the return to specify the position of the peaks. If omitted an index of the y_axis is used. (default: None)

lookahead -- distance to look ahead from a peak candidate to determine if
it is the actual peak (default: 200) '(samples / period) / f' where '4 >= f >= 1.25' might be a good value

delta -- this specifies a minimum difference between a peak and
the following points, before a peak may be considered a peak. Useful to hinder the function from picking up false peaks towards to end of the signal. To work well delta should be set to delta >= RMSnoise * 5. (default: 0)

When omitted delta function causes a 20% decrease in speed. When used Correctly it can double the speed of the function

return: two lists [max_peaks, min_peaks] containing the positive and
negative peaks respectively. Each cell of the lists contains a tuple of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*max_peaks)

pysisyphus.peakdetect.**peakdetect_fft**(*y_axis*, *x_axis*, *pad_len=20*)

Performs a FFT calculation on the data and zero-pads the results to increase the time domain resolution after performing the inverse fft and send the data to the 'peakdetect' function for peak detection.

Omitting the x_axis is forbidden as it would make the resulting x_axis value silly if it was returned as the index 50.234 or similar.

Will find at least 1 less peak then the 'peakdetect_zero_crossing' function, but should result in a more precise value of the peak as resolution has been increased. Some peaks are lost in an attempt to minimize spectral leakage by calculating the fft between two zero crossings for n amount of signal periods.

The biggest time eater in this function is the ifft and thereafter it's the 'peakdetect' function which takes only half the time of the ifft. Speed improvements could include to check if 2**n points could be used for fft and ifft or change the 'peakdetect' to the 'peakdetect_zero_crossing', which is maybe 10 times faster than 'peakdetct'. The pro of 'peakdetect' is that it results in one less lost peak. It should also be noted that the time used by the ifft function can change greatly depending on the input.

keyword arguments: y_axis -- A list containing the signal over which to find peaks

x_axis -- A x-axis whose values correspond to the y_axis list and is used
in the return to specify the position of the peaks.

pad_len -- By how many times the time resolution should be
increased by, e.g. 1 doubles the resolution. The amount is rounded up to the nearest 2**n amount (default: 20)

return: two lists [max_peaks, min_peaks] containing the positive and
negative peaks respectively. Each cell of the lists contains a tuple of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*max_peaks)

pysisyphus.peakdetect.**peakdetect_parabola**(*y_axis*, *x_axis*, *points=31*)

Function for detecting local maxima and minima in a signal. Discovers peaks by fitting the model function: y = k (x - tau) ** 2 + m to the peaks. The amount of points used in the fitting is set by the points argument.

Omitting the x_axis is forbidden as it would make the resulting x_axis value silly, if it was returned as index 50.234 or similar.

will find the same amount of peaks as the 'peakdetect_zero_crossing' function, but might result in a more precise value of the peak.

keyword arguments: y_axis -- A list containing the signal over which to find peaks

x_axis -- A x-axis whose values correspond to the y_axis list and is used
in the return to specify the position of the peaks.

points -- How many points around the peak should be used during curve
fitting (default: 31)

return: two lists [max_peaks, min_peaks] containing the positive and
negative peaks respectively. Each cell of the lists contains a tuple of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*max_peaks)

pysisyphus.peakdetect.**peakdetect_sine**(*y_axis*, *x_axis*, *points=31*, *lock_frequency=False*)

Function for detecting local maxima and minima in a signal. Discovers peaks by fitting the model function: y = A * sin(2 * pi * f * (x - tau)) to the peaks. The amount of points used in the fitting is set by the points argument.

Omitting the x_axis is forbidden as it would make the resulting x_axis value silly if it was returned as index 50.234 or similar.

will find the same amount of peaks as the 'peakdetect_zero_crossing' function, but might result in a more precise value of the peak.

The function might have some problems if the sine wave has a non-negligible total angle i.e. a k*x component, as this messes with the internal offset calculation of the peaks, might be fixed by fitting a y = k * x + m function to the peaks for offset calculation.

keyword arguments: y_axis -- A list containing the signal over which to find peaks

x_axis -- A x-axis whose values correspond to the y_axis list and is used
in the return to specify the position of the peaks.

points -- How many points around the peak should be used during curve
fitting (default: 31)

lock_frequency -- Specifies if the frequency argument of the model
function should be locked to the value calculated from the raw peaks or if optimization process may tinker with it. (default: False)

return: two lists [max_peaks, min_peaks] containing the positive and
negative peaks respectively. Each cell of the lists contains a tuple of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*max_peaks)

pysisyphus.peakdetect.**peakdetect_sine_locked**(*y_axis*, *x_axis*, *points=31*)

Convenience function for calling the 'peakdetect_sine' function with the lock_frequency argument as True.

keyword arguments: y_axis -- A list containing the signal over which to find peaks x_axis -- A x-axis whose values correspond to the y_axis list and is used

in the return to specify the position of the peaks.

points -- How many points around the peak should be used during curve
fitting (default: 31)

return: see the function 'peakdetect_sine'

pysisyphus.peakdetect.**peakdetect_spline**(*y_axis*, *x_axis*, *pad_len=20*)

Performs a b-spline interpolation on the data to increase resolution and send the data to the 'peakdetect_zero_crossing' function for peak detection.

Omitting the x_axis is forbidden as it would make the resulting x_axis value silly if it was returned as the index 50.234 or similar.

will find the same amount of peaks as the 'peakdetect_zero_crossing' function, but might result in a more precise value of the peak.

keyword arguments: y_axis -- A list containing the signal over which to find peaks

x_axis -- A x-axis whose values correspond to the y_axis list and is used
in the return to specify the position of the peaks. x-axis must be equally spaced.

pad_len -- By how many times the time resolution should be increased by,
e.g. 1 doubles the resolution. (default: 20)

return: two lists [max_peaks, min_peaks] containing the positive and
negative peaks respectively. Each cell of the lists contains a tuple of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*max_peaks)

pysisyphus.peakdetect.**peakdetect_zero_crossing**(*y_axis*, *x_axis=None*, *window=11*)

Function for detecting local maxima and minima in a signal. Discovers peaks by dividing the signal into bins and retrieving the maximum and minimum value of each the even and odd bins respectively. Division into bins is performed by smoothing the curve and finding the zero crossings.

Suitable for repeatable signals, where some noise is tolerated. Executes faster than 'peakdetect', although this function will break if the offset of the signal is too large. It should also be noted that the first and last peak will probably not be found, as this function only can find peaks between the first and last zero crossing.

keyword arguments: y_axis -- A list containing the signal over which to find peaks

x_axis -- A x-axis whose values correspond to the y_axis list
and is used in the return to specify the position of the peaks. If omitted an index of the y_axis is used. (default: None)

window -- the dimension of the smoothing window; should be an odd integer
(default: 11)

return: two lists [max_peaks, min_peaks] containing the positive and
negative peaks respectively. Each cell of the lists contains a tuple of: (position, peak_value) to get the average peak value do: np.mean(max_peaks, 0)[1] on the results to unpack one of the lists into x, y coordinates do: x, y = zip(*max_peaks)

pysisyphus.peakdetect.**zero_crossings**(*y_axis*, *window_len=11*, *window_f='hanning'*, *offset_corrected=False*)

Algorithm to find zero crossings. Smooths the curve and finds the zero-crossings by looking for a sign change.

keyword arguments: y_axis -- A list containing the signal over which to find zero-crossings

window_len -- the dimension of the smoothing window; should be an odd
integer (default: 11)

window_f -- the type of window from 'flat', 'hanning', 'hamming',
'bartlett', 'blackman' (default: 'hanning')

offset_corrected -- Used for recursive calling to remove offset when needed

return: the index for each zero-crossing

pysisyphus.peakdetect.**zero_crossings_sine_fit**(*y_axis*, *x_axis*, *fit_window=None*, *smooth_window=11*)

Detects the zero crossings of a signal by fitting a sine model function around the zero crossings: y = A * sin(2 * pi * Hz * (x - tau)) + k * x + m Only tau (the zero crossing) is varied during fitting.

Offset and a linear drift of offset is accounted for by fitting a linear function the negative respective positive raw peaks of the wave-shape and the amplitude is calculated using data from the offset calculation i.e. the 'm' constant from the negative peaks is subtracted from the positive one to obtain amplitude.

Frequency is calculated using the mean time between raw peaks.

Algorithm seems to be sensitive to first guess e.g. a large smooth_window will give an error in the results.

keyword arguments: y_axis -- A list containing the signal over which to find peaks

x_axis -- A x-axis whose values correspond to the y_axis list
and is used in the return to specify the position of the peaks. If omitted an index of the y_axis is used. (default: None)

fit_window -- Number of points around the approximate zero crossing that
should be used when fitting the sine wave. Must be small enough that no other zero crossing will be seen. If set to none then the mean distance between zero crossings will be used (default: None)

smooth_window -- the dimension of the smoothing window; should be an odd
integer (default: 11)

return: A list containing the positions of all the zero crossings.

17.1.18 pysisyphus.plot module

pysisyphus.plot.**CDD_PNG_FNS = 'cdd_png_fns'**

Default to kJ mol[1]. DE_LABEL and get_en_conv will be overwritten when run() is executed. Both are still defined here, to make the functions from plot.py importable.

pysisyphus.plot.**anim_cos**(*cart_coords*, *energies*)

pysisyphus.plot.**get_en_conv**()

pysisyphus.plot.**get_force_unit**(*coord_type*)

pysisyphus.plot.**list_h5_groups**(*h5_fn*)

pysisyphus.plot.**load_h5**(*h5_fn*, *h5_group*, *datasets=None*, *attrs=None*)

pysisyphus.plot.**parse_args**(*args*)

pysisyphus.plot.**plot_afir**(*h5_fn='afir.h5'*, *h5_group='afir'*)

pysisyphus.plot.**plot_all_energies**(*h5*)

pysisyphus.plot.**plot_cos_energies**(*h5_fn='optimization.h5'*, *h5_group='opt'*)

pysisyphus.plot.**plot_cos_forces**(*h5_fn='optimization.h5'*, *h5_group='opt'*, *last=15*)

pysisyphus.plot.**plot_cycle**(*cart_coords*, *energies*)

pysisyphus.plot.**plot_gau**(*gau_fns*, *num=50*)

pysisyphus.plot.**plot_irc**()

pysisyphus.plot.**plot_irc_h5**(*h5*, *title=None*)

pysisyphus.plot.**plot_md**(*h5_group='run'*)

pysisyphus.plot.**plot_opt**(*h5_fn='optimization.h5'*, *h5_group='opt'*)

pysisyphus.plot.**plot_overlaps**(*h5*, *thresh=0.1*)

pysisyphus.plot.**plot_scan**(*h5_fn='scan.h5'*)

pysisyphus.plot.**plot_trj_energies**(*trj*)

Parse comments of .xyz/.trj as energies and plot.

pysisyphus.plot.**render_cdds**(*h5*)

pysisyphus.plot.**run**()

pysisyphus.plot.**spline_plot_cycles**(*cart_coords*, *energies*)

17.1.19 pysisyphus.run module

class pysisyphus.run.**RunResult**(*preopt_first_geom*, *preopt_last_geom*, *cos*, *cos_opt*, *ts_geom*, *ts_opt*, *end_geoms*, *irc*, *irc_geom*, *mdp_result*, *opt_geom*, *opt*, *calced_geoms*, *calced_results*, *stocastic*, *calc_getter*, *scan_geoms*, *scan_vals*, *scan_energies*, *perf_results*)

Bases: `tuple`

calc_getter

Alias for field number 15

calced_geoms

Alias for field number 12

calced_results

Alias for field number 13

cos

Alias for field number 2

cos_opt
Alias for field number 3

end_geoms
Alias for field number 6

irc
Alias for field number 7

irc_geom
Alias for field number 8

mdp_result
Alias for field number 9

opt
Alias for field number 11

opt_geom
Alias for field number 10

perf_results
Alias for field number 19

preopt_first_geom
Alias for field number 0

preopt_last_geom
Alias for field number 1

scan_energies
Alias for field number 18

scan_geoms
Alias for field number 16

scan_vals
Alias for field number 17

stocastic
Alias for field number 14

ts_geom
Alias for field number 4

ts_opt
Alias for field number 5

pysisyphus.run.**check_asserts**(*results*, *run_dict*)

pysisyphus.run.**copy_yaml_and_geometries**(*run_dict*, *yaml_fn*, *dest_and_add_cp*, *new_yaml_fn=None*)

pysisyphus.run.**do_clean**(*force=False*)
Deletes files from previous runs in the cwd. A similar function could be used to store everything ...

pysisyphus.run.**do_rmsds**(*xyz*, *geoms*, *end_fns*, *end_geoms*, *preopt_map=None*, *similar_thresh=0.25*)

pysisyphus.run.**get_calc_closure**(*base_name*, *calc_key*, *calc_kwargs*, *iter_dict=None*, *index=None*)

pysisyphus.run.**get_defaults**(*conf_dict*, *T_default=298.15*, *p_default=101325*)

pysisyphus.run.**get_last_calc_cycle**()

pysisyphus.run.**load_run_dict**(*yaml_fn*)

pysisyphus.run.**main**(*run_dict*, *restart=False*, *yaml_dir='./'*, *scheduler=None*)

pysisyphus.run.**parse_args**(*args*)

pysisyphus.run.**print_bibtex**()

pysisyphus.run.**print_header**()

Generated from https://asciiartgen.now.sh/?s=pysisyphus&style=colossal

pysisyphus.run.**run**()

pysisyphus.run.**run_calculations**(*geoms*, *calc_getter*, *scheduler=None*, *assert_track=False*, *run_func=None*)

pysisyphus.run.**run_endopt**(*irc*, *endopt_key*, *endopt_kwargs*, *calc_getter*)

pysisyphus.run.**run_from_dict**(*run_dict*, *cwd=None*, *set_defaults=True*, *yaml_fn=None*, *cp=None*, *scheduler=None*, *clean=False*, *fclean=False*, *version=False*, *restart=False*)

pysisyphus.run.**run_irc**(*geom*, *irc_key*, *irc_kwargs*, *calc_getter*)

pysisyphus.run.**run_md**(*geom*, *calc_getter*, *md_kwargs*)

pysisyphus.run.**run_mdp**(*geom*, *calc_getter*, *mdp_kwargs*)

pysisyphus.run.**run_preopt**(*first_geom*, *last_geom*, *calc_getter*, *preopt_key*, *preopt_kwargs*)

pysisyphus.run.**run_scan**(*geom*, *calc_getter*, *scan_kwargs*, *callback=None*)

pysisyphus.run.**run_stocastic**(*stoc*)

pysisyphus.run.**run_tsopt_from_cos**(*cos*, *tsopt_key*, *tsopt_kwargs*, *calc_getter=None*, *ovlp_thresh=0.4*, *coordinate_union='bonds'*)

pysisyphus.run.**setup_run_dict**(*run_dict*)

17.1.20 pysisyphus.socket_helper module

pysisyphus.socket_helper.**get_fmts**(*cartesians*)

pysisyphus.socket_helper.**recv_closure**(*sock*, *hdrlen*, *fmts*, *verbose=False*)

pysisyphus.socket_helper.**send_closure**(*sock*, *hdrlen*, *fmts*, *verbose=False*)

17.1.21 pysisyphus.testing module

class pysisyphus.testing.**DummyMark**(*available*)

Bases: object

pysisyphus.testing.**available**(*calculator*, ***kwargs*)

pysisyphus.testing.**module_available**(*calculator*)

pysisyphus.testing.**using**(*calculator*, *set_pytest_mark=True*)

Calling disabling set_pytest_mark avoids a runtime dependency on pytest.

17.1.22 pysisyphus.thermo module

pysisyphus.thermo.**get_thermoanalysis_from_hess_h5**(*h5_fn*, *T=298.15*, *p=101325*, *point_group='c1'*, *return_geom=False*)

pysisyphus.thermo.**print_thermoanalysis**(*thermo*, *geom=None*, *is_ts=False*, *unit='joule'*, *level=0*, *title=None*)

Print thermochemical analysis.

17.1.23 pysisyphus.trj module

exception pysisyphus.trj.**GotNoGeometryException**

Bases: Exception

pysisyphus.trj.**align**(*geoms*)

Align all geometries onto the first using partical procrustes.

pysisyphus.trj.**append**(*geoms*)

pysisyphus.trj.**center**(*geoms*)

pysisyphus.trj.**centerm**(*geoms*)

pysisyphus.trj.**dump_geoms**(*geoms*, *fn_base*, *trj_infix=''*, *dump_trj=True*, *dump_xyz=True*, *dump_pdb=False*, *ang=False*)

pysisyphus.trj.**every**(*geoms*, *every_nth*)

pysisyphus.trj.**frag_sort**(*geoms*)

pysisyphus.trj.**get**(*geoms*, *index*)

pysisyphus.trj.**get_geoms**(*xyz_fns*, *coord_type='cart'*, *geom_kwargs=None*, *union=False*, *same_prims=True*, *quiet=False*, *interpol_kwargs=None*)

Returns a list of Geometry objects in the given coordinate system and interpolates if necessary.

pysisyphus.trj.**get_interactively**(*geoms*)

pysisyphus.trj.**hardsphere_merge**(*geoms*)

pysisyphus.trj.**match**(*ref_geom*, *geom_to_match*)

pysisyphus.trj.**origin**(*geoms*)

pysisyphus.trj.**parse_args**(*args*)

pysisyphus.trj.**print_internal_vals**(*geoms*, *typed_prim*)

pysisyphus.trj.**print_internals**(*geoms*, *filter_atoms=None*, *add_prims=''*)

pysisyphus.trj.**read_geoms**(*xyz_fns*, *coord_type='cart'*, *geom_kwargs=None*)

pysisyphus.trj.**run**()

pysisyphus.trj.**shake**(*geoms*, *scale=0.1*, *seed=None*)

pysisyphus.trj.**spline_redistribute**(*geoms*)

pysisyphus.trj.**standard_orientation**(*geoms*)

pysisyphus.trj.**standardize_geoms**(*geoms*, *coord_type*, *geom_kwargs*, *same_prims=True*, *union=False*)

pysisyphus.trj.**translate**(*geoms*, *trans*)

17.1.24 pysisyphus.version module

17.1.25 pysisyphus.xyzloader module

pysisyphus.xyzloader.**coords_to_trj**(*trj_fn*, *atoms*, *coords_list*, *comments=None*)

pysisyphus.xyzloader.**make_trj_str**(*atoms*, *coords_list*, *comments=None*)

pysisyphus.xyzloader.**make_trj_str_from_geoms**(*geoms*, *comments=None*, *energy_comments=False*)

pysisyphus.xyzloader.**make_xyz_str**(*atoms*, *coords*, *comment=''*)

pysisyphus.xyzloader.**parse_trj_file**(*trj_fn*, *with_comments=False*)

pysisyphus.xyzloader.**parse_trj_str**(*trj_str*, *with_comments=False*)

pysisyphus.xyzloader.**parse_xyz_file**(*xyz_fn*, *with_comment=False*)

pysisyphus.xyzloader.**parse_xyz_str**(*xyz_str*, *with_comment*)

Parse a xyz string.

17.1.25.1 Paramters

xyz_str
[str] The contents of a .xyz file.

with_comment
[bool] Return comment line if True.

returns

- **atoms** (*list*) -- List of length N (N = number of atoms) holding the element symbols.
- **coords** (*np.array*) -- An array of shape (N, 3) holding the xyz coordinates.
- **comment_line** (*str, optional*) -- Comment line if with_comment argument was True.

pysisyphus.xyzloader.**split_xyz_str**(*xyz_str*)

Example:

xyz:

1

X -1.2 1.4 0.0 1

X 2.0 4.0 0.0

pysisyphus.xyzloader.**write_geoms_to_trj**(*geoms*, *fn*, *comments=None*)

17.1.26 pysisyphus.yaml_mods module

pysisyphus.yaml_mods.**get_constructor**(*unit*)

pysisyphus.yaml_mods.**get_loader**(*units=['Eh', 'kJpermol', 'kcalpermol', 'eV', 'fs', 'ps', 'ns', 'a0', 'angstrom', 'nm', 'deg']*)

17.1.27 Module contents

CHAPTER

EIGHTEEN

INDICES AND TABLES

- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z