## CONTENTS

1 Installation 3  
  1.1 Installing from conda 3  
  1.2 Building from source 3  
  1.3 Python extension module 6  

2 Comparison with other DFT-D3 implementations 9  
  2.1 Users of this library 9  

3 Public API documentation 11  
  3.1 Fortran API 11  
  3.2 C API 14  
  3.3 Python API 16  

Python Module Index 29  

Index 31
This page describes the usage and functionality of s-dftd3 reimplementations of the D3 dispersion correction. The s-dftd3 project aims to provide a user-friendly and uniform interface to the D3 dispersion model and for the calculation of DFT-D3 dispersion corrections.
The preferred method for installing this package is to use a binary release distributed with a package manager. If your preferred package manager is not available, you can also download the source code and build it yourself.

1.1 Installing from conda

This project is packaged for the conda package manager and available on the conda-forge channel. To install the conda package manager we recommend the miniforge installer. If the conda-forge channel is not yet enabled, add it to your channels with

```
conda config --add channels conda-forge
```

Once the conda-forge channel has been enabled, this project can be installed with:

```
conda install simple-dftd3
```

If you want to enable the Python API as well install

```
conda install dftd3-python
```

It is possible to list all of the versions available on your platform with:

```
conda search simple-dftd3 --channel conda-forge
```

Now you are ready to use the s-dftd3 executable, find the dftd3.h header or import dftd3 in your Python projects.

1.2 Building from source

This library depends on few Fortran modules to provide the desired functionality

- mctc-lib: Modular computation tool chain library
- mstore: Molecular structure store (testing only)
1.2.1 Meson based build

The primary build system of this project is meson. For the full feature-complete build it is highly recommended to perform the build and development with meson. To setup a build the following software is required:

- A Fortran 2008 compliant compiler, like GCC Fortran and Intel Fortran classic
- meson, version 0.55 or newer
- ninja, version 1.7 or newer

Optional dependencies are:

- a linear algebra backend, like MKL or OpenBLAS
- asciidoctor, to build the manual pages
- A C compiler to test the C API and compile the Python extension module
- Python 3.6 or newer with the CFFI package installed to build the Python API

To setup a new build run

```
meson setup _build --prefix=$HOME/.local
```

The Fortran and C compiler can be selected with the FC and CC environment variable, respectively. The installation location is selected using the --prefix option. The required Fortran modules will be fetched automatically from the upstream repositories and checked out in the subprojects directory.

**Note:** By default a redistributed version of the reference BLAS routines are used for matrix vector operations. To use a different BLAS library, the -Dblas option can be set.

For example, to use Intel MKL the custom option should be used and the libraries explicitly provided as in

```
meson setup _build --prefix=$HOME/.local -Dblas=custom -Dblas_libs="mkl_rt"
```

**Tip:** To produce statically linked binaries set --default-library=static and add -Dfortran_link_args=--static as well. To link statically against the MKL library, use the -Dblas_libs option to specify the required library names (-Dblas_libs=mkl_intel_l64,mkl_intel_thread,mkl_core). If meson cannot find the mkl_intel_thread library, append -qopenmp to the -Dfortran_link_args option.

To compile the project run

```
meson compile -C _build
```

Verify that the resulting projects is working correctly by running the testsuite with

```
meson test -C _build --print-errorlogs
```

In case meson is spawning too many concurrent test jobs limit the number of processes with the --num-processes option when running the test command. By default the whole library and its subprojects are tested, to limit the testing to the project itself add --suite s-dftd3 as option.

To verify the included parametrizations are working correctly run the extra testsuite by passing the --benchmark argument
meson test -C _build --print-errorlogs --benchmark

Finally, you can make the project available by installation with

meson install -C _build

### 1.2.2 CMake based build

This project also provides support for CMake to give projects using it as build system an easier way to interface. The CMake build files usually do not provide a feature-complete build, but contributions are more than welcome. To setup a build the following software is required:

- A Fortran 2008 compliant compiler, like GCC Fortran and Intel Fortran classic
- cmake, version 3.14 or newer
- ninja, version 1.10 or newer

Configure a new build with

```
cmake -B _build -G Ninja -DCMAKE_INSTALL_PREFIX=$HOME/.local
```

You can set the Fortran compiler in the FC environment variable. The installation location can be selected with the CMAKE_INSTALL_PREFIX, GNU install directories are supported by default. CMake will automatically fetch the required Fortran modules, you can provide specific version in the `subprojects` directory which will be used instead.

To run a build use

```
cmake --build _build
```

After a successful build make sure the testsuite passes

```
pushd _build && ctest --output-on-failure && popd
```

To make the project available install it with

```
cmake --install _build
```

### 1.2.3 Fpm based build

This projects supports building with the Fortran package manager (fpm). Create a new build by running

```
fpm build
```

You can adjust the Fortran compiler with the `--compiler` option and select the compilation profile with `--profile` release. To test the resulting build run the testsuite with

```
fpm test
```

The command line driver can be directly used from fpm with

```
fpm run --profile release -- --help
```

To make the installation accessible install the project with
The Python API is available as Python extension module. The easiest way to setup is to add `-Dpython=true` to a meson tree build and follow the meson installation instructions. The extension module will become available once the project is installed.

This section describes alternative ways to build the Python API

### 1.3.1 Using pip

This project support installation with pip as an easy way to build the Python API.

- C compiler to build the C-API and compile the extension module (the compiler name should be exported in the `CC` environment variable)
- Python 3.6 or newer
- The following Python packages are required additionally
  - `cffi`
  - `numpy`
  - `pkgconfig` (setup only)

Make sure to have your C compiler set to the CC environment variable

```bash
export CC=gcc
```

Install the project with pip

```bash
pip install .
```

To install extra dependencies as well use

```bash
pip install '.[ase]'
```

### 1.3.2 Using meson

The Python extension module can be built on-top of an existing installation, either provided by meson or CMake.

Building requires against an existing `s-dftd3` installation requires

- C compiler to build the C-API and compile the extension module
- meson version 0.55 or newer
- a build-system backend, *i.e.* `ninja` version 1.7 or newer
- Python 3.6 or newer with the `CFI` package installed

Setup a build with

```bash
meson setup _build_python python -Dpython_version=3
```
The Python version can be used to select a different Python version, it defaults to '3'. Python 2 is not supported with this project, the Python version key is meant to select between several local Python 3 versions.

Compile the project with

```
meson compile -C _build
```

The extension module is now available in `_build_python/dftd3/_libdftd3.*.so`. You can install as usual with

```
meson configure _build --prefix=/path/to/install
meson install -C _build
```
COMPARISON WITH OTHER DFT-D3 IMPLEMENTATIONS

This DFT-D3 reimplementation was created as a spin-off from the dfd4 and xtb project, to provide an easier to use API, improve the parallel performance and get a fast build of the DFT-D3 project. It is however not the only project providing an implementation of DFT-D3, many forks of the original reference implementation and some reimplementations are currently available.

A non-comprehensive list of DFT-D3 implementations is provided here:

<table>
<thead>
<tr>
<th>repository</th>
<th>license</th>
<th>APIs</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>dftd3</td>
<td>GPL-1.0</td>
<td>Fortran</td>
<td>reference implementation</td>
</tr>
<tr>
<td>dftd3/simple-dftd3</td>
<td>LGPL-3.0</td>
<td>Fortran, C, Python</td>
<td></td>
</tr>
<tr>
<td>dftbplus/libdftd3</td>
<td>GPL-1.0</td>
<td>Fortran</td>
<td>patched fork</td>
</tr>
<tr>
<td>ehermes/ased3</td>
<td>LGPL-3.0</td>
<td>Python</td>
<td>f2py, ASE</td>
</tr>
<tr>
<td>pfnet-research/torch-dftd</td>
<td>MIT</td>
<td>Python</td>
<td>torch</td>
</tr>
<tr>
<td>cuanto/libdftd3</td>
<td>GPL-3.0</td>
<td>Fortran, Python</td>
<td>ctypes, pyscf</td>
</tr>
<tr>
<td>cresset-group/dftd3</td>
<td>GPL-1.0</td>
<td>Fortran</td>
<td>patched fork</td>
</tr>
<tr>
<td>loriab/dftd3</td>
<td>GPL-1.0</td>
<td>Fortran</td>
<td>patched fork, Windows</td>
</tr>
<tr>
<td>f3rmion/dftd3</td>
<td>GPL-1.0</td>
<td>Fortran</td>
<td>patched fork</td>
</tr>
<tr>
<td>bobbypaton/pydftd3</td>
<td>MIT</td>
<td>Python</td>
<td>Gaussian</td>
</tr>
</tbody>
</table>

Many more versions are probably around or redistributed in various quantum chemistry programs.

2.1 Users of this library

A list of projects currently using this DFT-D3 implementation is given here.

**tblite:** *(since 0.1.0)*
Light-weight tight-binding framework

**DFTB+:** *(since 21.2)*
General package for performing fast atomistic calculations

**DFT-FE:**
Real-space DFT calculations using Finite Elements

**QCEngine:** *(WIP)*
Quantum chemistry program executor and IO standardizer. For current status see qcegine#343

**Siesta:** *(WIP)*
A first-principles materials simulation code using DFT. For current status see siesta!70

If your project is using s-dftd3 feel free to add your project to this list.
This DFT-D3 implementation provides first class API support Fortran, C and Python. Other programming languages should try to interface via one of those three APIs. To provide first class API support for a new language the interface specification should be available as meson build files.

3.1 Fortran API

The s-dftd3 library seamlessly integrates with other Fortran projects via module interfaces,

Note: Generally, all quantities used in the library are stored in atomic units.

Contents

• Fortran API
  – Handling of geometries and structure
  – Error handling
  – Performing calculations

3.1.1 Handling of geometries and structure

The basic infrastructure to handle molecular and periodic structures is provided by the modular computation tool chain library. The library provides a structure type which is used to represent all geometry related informations in s-dftd3. A structure type can be constructed from arrays or read from a file.

The constructor is provided with the generic interface new and takes an array of atomic numbers (integer) or element symbols (character(len=*)) as well as the cartesian coordinates in Bohr. Additionally, the molecular charge and the number of unpaired electrons can be provided the charge and uhf keyword, respectively. To create a periodic structure the lattice parameters can be passed as 3 by 3 matrix with the lattice keyword.

An example for using the constructor is given here

```fortran
subroutine example
  use mctc_env, only : wp
  use mctc_io, only : structure_type, new
  implicit none
```

(continues on next page)
To interact with common input file formats for structures the `read_structure` procedure is available. The file type is inferred from the name of the file automatically or if a file type hint is provided directly from the enumerator of available file types. The `read_structure` routine can also use an already opened unit, but in this case the file type hint is mandatory to select the correct format to read from.

```fortran
use mctc_env, only  : error_type
use mctc_io, only   : structure_type, read_structure, file_type
implicit none

type(structure_type) :: mol

type(error_type), allocatable :: error
character(len=:), allocatable :: input

input = "struc.xyz"

call read_structure(mol, input, error, file_type%xyz)
if (allocated(error)) then
   print '(a)', error%message
   stop 1
end if
! ...
end subroutine example
```

The structure type as well as the error type are using only allocatable members and can therefore be used without requiring explicit deconstruction.

Certain members of the structure type should be considered immutable, like the number of atoms (`nat`), the identifiers for unique atoms (`id`) and the boundary conditions (`periodic`). To change those specific structure parameters the structure type and all dependent objects should be reconstructed to ensure a consistent setup. Other properties, like the geometry (`xyz`), molecular charge (`charge`), number of unpaired electrons (`uhf`) and lattice parameters (`lattice`) can be changed without requiring to reconstruct dependent objects like calculators or restart data.
3.1.2 Error handling

The basic error handler is an allocatable derived type, available from mctc_env as error_type, which signals an error by its allocation status.

```fortran
use mctc_env, only : error_type, fatal_error
implicit none
type(error_type), allocatable :: error

call always_ok(error)
if (allocated(error)) then
  print '(a)', "Unexpected failure: ", error%message
end if

call always_failed(error)
if (allocated(error)) then
  print '(a)', "Error: ", error%message
end if
contains
  subroutine always_ok(error)
    type(error_type), allocatable, intent(out) :: error
  end subroutine always_ok

  subroutine always_failed(error)
    type(error_type), allocatable, intent(out) :: error
    call fatal_error(error, "Message associated with this error")
  end subroutine always_failed
end
```

An unhandled error might get dropped by the next procedure call.

3.1.3 Performing calculations

An example for performing a calculation with DFT-D3(BJ)-ATM is shown below

```fortran
subroutine calc_dftd3(mol, method, energy, gradient, sigma, error)
  use mctc_env
  use mctc_io
  use dftd3
  type(structure_type), intent(in) :: mol
  character(len=__), intent(in) :: method
  real(wp), intent(out) :: energy
  real(wp), intent(out) :: gradient(:, :)
  real(wp), intent(out) :: sigma(:, :)
  type(error_type), allocatable, intent(out) :: error
  type(d3_model) :: disp
  type(d3_param) :: inp
  class(damping_param), allocatable :: param
  call get_rational_damping(inp, method, error, s9=1.0_wp)
  if (allocated(error)) return
end subroutine calc_dftd3
```

(continues on next page)
call new_rational_damping(param, inp, mol)
call new_d3_model(disp, mol)
call get_dispersion(mol, disp, param, realspace_cutoff(), energy, &
              & gradient, sigma)
end subroutine calc_dftd3

### 3.2 C API

The C API bindings are provided by using the `iso_c_binding` intrinsic module. Generally, objects are exported as opaque pointers and can only be manipulated within the library. The API user is required delete all objects created in the library by using the provided deconstructor functions to avoid memory leaks.

Overall four classes of objects are provided by the library

- error handlers (`dftd3_error`), used to communicate exceptional conditions and errors from the library to the user
- structure containers (`dftd3_structure`), used to represent the system specific information and geometry data, only the latter are mutable for the user
- dispersion model objects (`dftd3_model`), general model for calculating dispersion related properties
- damping function objects (`dftd3_param`) polymorphic objects to represent the actual method parametrisation

**Note:** Generally, all quantities provided to the library are assumed to be in atomic units.

### Contents

- **C API**
  - Error handling
  - Structure data
  - Performing calculations

### 3.2.1 Error handling

The library provides a light error handle type (`dftd3_error`) for storing error information. The error handle requires only small overhead to construct and can only contain a single error.

The handler is represented by an opaque pointer and can only be manipulated by call from the library. The user of those objects is required to delete the handlers again using the library provided deconstructors to avoid memory leaks.
3.2.2 Structure data

The structure data is used to represent the system of interest in the library. It contains immutable system specific information like the number of atoms, the unique atom groups and the boundary conditions as well as mutable geometry data like cartesian coordinates and lattice parameters.

3.2.3 Performing calculations

An example wrapper to perform a DFT-D3(BJ)-ATM calculation is shown below.

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include "dftd3.h"

static const buffersize = 512;

int calc_dftd3(int natoms, int* numbers, double* positions,
                double* lattice, bool* periodic, char* method,
                double* energy, double* gradient, double* sigma)
{
   // Local API objects from the s-dftd3 library
   dftd3_error error = dftd3_new_error();
   dftd3_structure mol = NULL;
   dftd3_model disp = NULL;
   dftd3_param param = NULL;
   int stat = EXIT_SUCCESS;

   // Create a new geometry for the library to work with
   mol = dftd3_new_structure(error, natoms, numbers, positions, lattice, periodic);
   stat = dftd3_check_error(error);

   if (stat) {
      // Initialize the D3 dispersion model for the given structure,
      // this step depends on the atomic numbers, but not on the actual geometry
      disp = dftd3_new_d3_model(error, mol);
      stat = dftd3_check_error(error);
   }

   if (stat) {
      // Load D3(BJ)-ATM parameters for the given method from internal storage,
      // this step depends on the atomic numbers, but not on the actual geometry
      param = dftd3_load_rational_damping(error, mol, method, true);
      stat = dftd3_check_error(error);
   }

   if (stat) {
      // Evaluate the dispersion energy, gradient and virial,
      // the gradient and virial are optional and can be replaced by NULL
      dftd3_get_dispersion(error, mol, disp, param, &energy, gradient, sigma);
  }
}
```

(continues on next page)
```c
stat = dftd3_check_error(error);
}
if (!stat) {
    char buffer[buffersize];
    dftd3_get_error(error, buffer, buffersize);
    printf("[Error] %s\n", buffer);
}
```

// Always free the used memory
dftd3_delete(error);
dftd3_delete(mol);
dftd3_delete(disp);
dftd3_delete(param);

return stat;
```

3.3 Python API

Python API for the DFT-D3 dispersion model

3.3.1 ASE Support

ASE calculator implementation for the s-dftd3 program.

This module provides a basic single point calculator implementations to integrate the s-dftd3 API into existing ASE workflows. To use DFTD3 as dispersion correction the ase.calculators.mixing module can be used to combine DFTD3 with a DFT calculator using the SumCalculator.

Supported properties by this calculator are:

- energy (free_energy)
- forces
- stress

Supported keywords are

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>None</td>
<td>Method to calculate dispersion for</td>
</tr>
<tr>
<td>damping</td>
<td>None</td>
<td>Damping function to use</td>
</tr>
<tr>
<td>params_tweaks</td>
<td>None</td>
<td>Optional dict with the damping parameters</td>
</tr>
<tr>
<td>cache_api</td>
<td>True</td>
<td>Reuse generate API objects (recommended)</td>
</tr>
</tbody>
</table>

The params_tweaks dict contains the damping parameters, at least s8, a1 and a2 must be provided
<table>
<thead>
<tr>
<th>Tweakable parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s6</td>
<td>1.0</td>
<td>Scaling of the dipole-dipole dispersion</td>
</tr>
<tr>
<td>s8</td>
<td>None</td>
<td>Scaling of the dipole-quadrupole dispersion</td>
</tr>
<tr>
<td>s9</td>
<td>1.0</td>
<td>Scaling of the three-body dispersion energy</td>
</tr>
<tr>
<td>a1</td>
<td>None</td>
<td>Scaling of the critical radii</td>
</tr>
<tr>
<td>a2</td>
<td>None</td>
<td>Offset of the critical radii</td>
</tr>
<tr>
<td>alp</td>
<td>16.0</td>
<td>Exponent of the zero damping (ATM only)</td>
</tr>
</tbody>
</table>

Either method or s8, a1 and a2 must be provided. s9 can be used to overwrite the ATM scaling if the method is provided in the model. Disabling the three-body dispersion (s9=0.0) changes the internal selection rules for damping parameters of a given method and prefers special two-body only damping parameters if available!

**Example**

```python
>>> from ase.build import molecule
>>> from dftd3.ase import DFTD3
>>> atoms = molecule('H2O')
>>> atoms.calc = DFTD3(method="TPSS", damping="d3bj")
>>> atoms.get_potential_energy()
-0.0114416338147162
>>> atoms.calc.set(method="PBE")
{'method': 'PBE'}
>>> atoms.get_potential_energy()
-0.005358475432239303
>>> atoms.get_forces()
array([[ 0.   ,  0.   ,  0.00296845],
       [-0.00119152, -0.00148423],
       [-0.00119152, -0.00148423]])
```

### 3.3.2 QCSchema Support

Integration with the QCArchive infrastructure.

This module provides a way to translate QCSchema or QCElemental Atomic Input into a format understandable by the dftd3 API which in turn provides the calculation results in a QCSchema compatible format.

Supported keywords are

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>level_hint</td>
<td>None</td>
<td>Dispersion correction level</td>
</tr>
<tr>
<td>params_tweaks</td>
<td>None</td>
<td>Optional dict with the damping parameters</td>
</tr>
<tr>
<td>pair_resolved</td>
<td>False</td>
<td>Enable pairwise resolved dispersion energy</td>
</tr>
</tbody>
</table>

Allowed level hints are "d3bj", "d3zero", "d3bjm"/"d3mbj", "d3mzero"/"d3zerom", and "d3op".

The params_tweaks dict contains the damping parameters, at least s8, a1 and a2 must be provided for rational damping, while s8 and rs6 are required in case of zero damping.

Parameters for (modified) rational damping are:
### Tweakable parameter details

<table>
<thead>
<tr>
<th>Tweakable parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s6</td>
<td>1.0</td>
<td>Scaling of the dipole-dipole dispersion</td>
</tr>
<tr>
<td>s8</td>
<td>None</td>
<td>Scaling of the dipole-quadrupole dispersion</td>
</tr>
<tr>
<td>s9</td>
<td>1.0</td>
<td>Scaling of the three-body dispersion energy</td>
</tr>
<tr>
<td>a1</td>
<td>None</td>
<td>Scaling of the critical radii</td>
</tr>
<tr>
<td>a2</td>
<td>None</td>
<td>Offset of the critical radii</td>
</tr>
<tr>
<td>alp</td>
<td>14.0</td>
<td>Exponent of the zero damping (ATM only)</td>
</tr>
</tbody>
</table>

Parameters for (modified) zero damping are:

<table>
<thead>
<tr>
<th>Tweakable parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s6</td>
<td>1.0</td>
<td>Scaling of the dipole-dipole dispersion</td>
</tr>
<tr>
<td>s8</td>
<td>None</td>
<td>Scaling of the dipole-quadrupole dispersion</td>
</tr>
<tr>
<td>s9</td>
<td>1.0</td>
<td>Scaling of the three-body dispersion energy</td>
</tr>
<tr>
<td>rs6</td>
<td>None</td>
<td>Scaling of the dipole-dipole damping</td>
</tr>
<tr>
<td>rs8</td>
<td>1.0</td>
<td>Scaling of the dipole-quadrupole damping</td>
</tr>
<tr>
<td>alp</td>
<td>14.0</td>
<td>Exponent of the zero damping</td>
</tr>
<tr>
<td>bet</td>
<td>None</td>
<td>Offset for damping radius (modified zero damping)</td>
</tr>
</tbody>
</table>

Parameters for optimized power damping are:

<table>
<thead>
<tr>
<th>Tweakable parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s6</td>
<td>1.0</td>
<td>Scaling of the dipole-dipole dispersion</td>
</tr>
<tr>
<td>s8</td>
<td>None</td>
<td>Scaling of the dipole-quadrupole dispersion</td>
</tr>
<tr>
<td>s9</td>
<td>1.0</td>
<td>Scaling of the three-body dispersion energy</td>
</tr>
<tr>
<td>a1</td>
<td>None</td>
<td>Scaling of the critical radii</td>
</tr>
<tr>
<td>a2</td>
<td>None</td>
<td>Offset of the critical radii</td>
</tr>
<tr>
<td>alp</td>
<td>14.0</td>
<td>Exponent of the zero damping (ATM only)</td>
</tr>
<tr>
<td>bet</td>
<td>None</td>
<td>Power for the zero-damping component</td>
</tr>
</tbody>
</table>

**Note:** input_data.model.method with a full method name and input_data.keywords["params_tweaks"] cannot be provided at the same time. It is an error to provide both options at the same time.

### Example

```python
>>> from dftd3.qcschema import run_qcschema
>>> import qcelemental as qcel

>>> atomic_input = qcel.models.AtomicInput(...
...   molecule = qcel.models.Molecule(...
...     symbols = ["O", "H", "H"],
...     geometry = [...
...       0.00000000000000, 0.00000000000000, -0.73578586109551,
...       1.44183152868459, 0.00000000000000, 0.36789293054775,
...       -1.44183152868459, 0.00000000000000, 0.36789293054775,...
...     ],
...     driver = "energy",
...   )
```

(continues on next page)
... model = {
...     "method": "TPSS-D3(BJ)",
... },
... keywords = {},
... )

>>> atomic_result = run_qcschema(atomic_input)
>>> atomic_result.return_result
-0.00042042440936212056

3.3.3 PySCF Support

Compatibility layer for supporting DFT-D3 in pyscf.

```python
class dftd3.pyscf.DFTD3Dispersion(*args: Any, **kwargs: Any)

Implementation of the interface for using DFT-D3 in pyscf. The `xc` functional can be provided in the constructor together with the `version` of the DFT-D3 damping function to use. Possible damping functions are

"d3bj": (default)
   For rational damping function

"d3zero"
   For zero damping function

"d3mbj"
   Modified damping parameters for the rational damping function

"d3mzero"
   Modified version of the zero damping function

"d3op"
   Optimized power damping function
```

Custom parameters can be provided with the `param` dictionary. The `param` dict contains the damping parameters, at least `s8`, `a1` and `a2` must be provided for rational damping, while `s8` and `rs6` are required in case of zero damping.

Parameters for (modified) rational damping are:

<table>
<thead>
<tr>
<th>Tweakable parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s6</td>
<td>1.0</td>
<td>Scaling of the dipole-dipole dispersion</td>
</tr>
<tr>
<td>s8</td>
<td>None</td>
<td>Scaling of the dipole-quadrupole dispersion</td>
</tr>
<tr>
<td>s9</td>
<td>1.0</td>
<td>Scaling of the three-body dispersion energy</td>
</tr>
<tr>
<td>a1</td>
<td>None</td>
<td>Scaling of the critical radii</td>
</tr>
<tr>
<td>a2</td>
<td>None</td>
<td>Offset of the critical radii</td>
</tr>
<tr>
<td>alp</td>
<td>14.0</td>
<td>Exponent of the zero damping (ATM only)</td>
</tr>
</tbody>
</table>

Parameters for (modified) zero damping are:
### Parameters for optimized power damping are:

<table>
<thead>
<tr>
<th>Tweakable parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s6</td>
<td>1.0</td>
<td>Scaling of the dipole-dipole dispersion</td>
</tr>
<tr>
<td>s8</td>
<td>None</td>
<td>Scaling of the dipole-quadrupole dispersion</td>
</tr>
<tr>
<td>s9</td>
<td>1.0</td>
<td>Scaling of the three-body dispersion energy</td>
</tr>
<tr>
<td>a1</td>
<td>None</td>
<td>Scaling of the critical radii</td>
</tr>
<tr>
<td>a2</td>
<td>None</td>
<td>Offset of the critical radii</td>
</tr>
<tr>
<td>alp</td>
<td>14.0</td>
<td>Exponent of the zero damping (ATM only)</td>
</tr>
<tr>
<td>bet</td>
<td>None</td>
<td>Offset for damping radius (modified zero damping)</td>
</tr>
</tbody>
</table>

The version of the damping can be changed after constructing the dispersion correction. With the `atm` boolean the three-body dispersion energy can be enabled, which is generally recommended.

### Examples

```python
>>> from pyscf import gto
>>> import dftd3.pyscf as disp

>>> mol = gto.M(...
...  atom=''',
...    C -0.189833176 -0.645396435 0.069807761  
...    C 1.121636324 0.354065576 0.439096514  
...    C 1.486520953 0.962572632 0.712107225  
...    C 0.549329390 1.989209324 0.61768956  
...    C -0.757627135 1.681862630 0.246856908  
...    C -1.138190460 0.370551816 -0.028582325  
...    Br -2.038462778 3.070459841 0.115165429  
...    H 1.852935245 -1.146434699 0.514119204  
...    H 0.825048723 3.012176989 0.829385472  
...    H 2.502259769 1.196433556 1.000317333  
...    H -2.157140187 0.151608161 -0.313181471  
...    H -0.480820487 -1.664983631 -0.142918416  
...    S -4.157443472 5.729584377 -0.878761129  
...    H -4.823791426 4.796089466 -1.563433338  
...    C -2.828338520 5.970593053 -2.091189515  
...    H -2.167577293 6.722356639 -1.668621815  
...    H -2.264954814 5.054835899 -2.240198499  
...    H -3.218524904 6.337447714 -3.035087058  
...    ),
...>

>>> d3 = disp.DFTD3Dispersion(mol, xc="PW6B95", version="d3bj")
```

(continues on next page)
>>> d3.kernel()[0]
array(-0.01009386)
>>> d3.version = "d3zero"  # Change to zero damping
>>> d3.kernel()[0]
array(-0.00574098)
>>> d3.atm = True       # Activate three-body dispersion
>>> d3.kernel()[0]
array(-0.00574289)

dump_flags(verbosr: Optional[bool] = None)
    Show options used for the DFT-D3 dispersion correction.

kernel() → Tuple[float, numpy.ndarray]
    Compute the DFT-D3 dispersion correction.

    The dispersion model as well as the parameters are created locally and not part of the state of the instance.

    Returns
    The energy and gradient of the DFT-D4 dispersion correction.

    Return type
    float, ndarray

Examples

>>> from pyscf import gto
>>> import dftd3.pyscf as disp
>>> mol = gto.M(...
... atom='''
... Br 0.000000 0.000000 1.919978
... Br 0.000000 0.000000 -0.367147
... N 0.000000 0.000000 -3.235006
... C 0.000000 0.000000 -4.376626
... H 0.000000 0.000000 -5.444276
... ''',
... )
>>> d4 = disp.DFTD3Dispersion(mol, xc="PBE0")
>>> energy, gradient = d4.kernel()
>>> energy
array(-0.00303589)
>>> gradient
array([[ 0.00000000e+00, 0.00000000e+00, 9.66197638e-05],
[ 0.00000000e+00, 0.00000000e+00, 2.36000434e-04],
[ 0.00000000e+00, 0.00000000e+00, -1.16718302e-04],
[ 0.00000000e+00, 0.00000000e+00, -1.84332770e-04],
[ 0.00000000e+00, 0.00000000e+00, -3.15691249e-05]])

reset(mol: pyscf.gto.Mole)
    Reset mol and clean up relevant attributes for scanner mode

dftd3.pyscf.energy(mf: pyscf.scf.hf.SCF, **kwargs) → pyscf.scf.hf.SCF
    Apply DFT-D3 corrections to SCF or MCSCF methods by returning an instance of a new class built from the original instances class. The dispersion correction is stored in the with_dftd3 attribute of the class.
Parameters

- \textbf{mf (scf.hf.SCF)} – The method to which DFT-D3 corrections will be applied.
- \textbf{**kwargs} – Keyword arguments passed to the \textit{DFTD3Dispersion} class.

Return type
The method with DFT-D3 corrections applied.

Examples

```python
>>> from pyscf import gto, scf
>>> import dftd3.pyscf as disp
>>> mol = gto.M(
...     atom='''
...     N -1.57871857 -0.04661102 0.00000000
...     N  1.57871857  0.04661102 0.00000000
...     H -2.15862174  0.13639605 0.80956529
...     H -0.84947130  0.65819321 0.00000000
...     H -2.15862174  0.13639605 -0.80956529
...     H  2.15862174 -0.13639605 -0.80956529
...     H  0.84947130 -0.65819321 0.00000000
...     H  2.15862174 -0.13639605 0.80956529
...     ''),
...     ...)
>>> mf = disp.energy(scf.RHF(mol)).run()
converged SCF energy = -110.932603617026
>>> mf.kernel()
-110.93260361702605
```

dftd3.pyscf.grad(scf_grad: pyscf.grad.rhf.Gradients, **kwargs)

Apply DFT-D3 corrections to SCF or MCSCF nuclear gradients methods by returning an instance of a new class built from the original class. The dispersion correction is stored in the \textit{with_dftd3} attribute of the class.

Parameters

- \textbf{scf_grad (rhf.grad.Gradients)} – The method to which DFT-D3 corrections will be applied.
- \textbf{**kwargs} – Keyword arguments passed to the \textit{DFTD3Dispersion} class.

Return type
The method with DFT-D3 corrections applied.

Examples

```python
>>> from pyscf import gto, scf
>>> import dftd3.pyscf as disp
>>> mol = gto.M(
...     atom='''
...     O -1.65542061 -0.12330038 0.00000000
...     O  1.24621244  0.10268870 0.00000000
...     H -0.70409026  0.03193167 0.00000000
...     H  2.03867273  0.75372294 0.00000000
...     '),
...     ...)
>>> mf = disp.energy(scf.RHF(mol)).run()
converged SCF energy = -110.932603617026
>>> mf.kernel()
-110.93260361702605
```

(continues on next page)
3.3.4 Library interface

Wrapper around the C-API of the s-dftd3 shared library. It provides the definition the basic interface to the library for most further integration in other Python frameworks.

The classes defined here allow a more Pythonic usage of the API object provided by the library in actual workflows than the low-level access provided in the CFFI generated wrappers.

**Structure**

```python
```

Represents a wrapped structure object in s-dftd3. The molecular structure data object has a fixed number of atoms and immutable atomic identifiers.

**Example**

```python
>>> from dftd3.interface import Structure
>>> import numpy as np
>>> mol = Structure(
...     positions=np.array([
...         [+0.00000000000000, +0.00000000000000, -0.73578586109551],
...         [+1.44183152868459, +0.00000000000000, +0.36789293054775],
...         [-1.44183152868459, +0.00000000000000, +0.36789293054775],
...     ]),
...     numbers = np.array([8, 1, 1]),
... )
>>> len(mol)
3
```
 Raises

**ValueError** – on invalid input, like incorrect shape / type of the passed arrays

**update** *(positions: numpy.ndarray, lattice: Optional[numpy.ndarray] = None) → None*

Update coordinates and lattice parameters, both provided in atomic units (Bohr). The lattice update is optional also for periodic structures.

Generally, only the cartesian coordinates and the lattice parameters can be updated, every other modification, boundary condition, atomic types or number of atoms requires the complete reconstruction of the object.

 Raises

**ValueError** – on invalid input, like incorrect shape / type of the passed arrays

**DispersionModel**


Contains the required information to evaluate all dispersion related properties, like C6 coefficients. It also manages an instance of the geometry it was constructed for to ensure that the dispersion model is always used with the correct structure input.

**get_dispersion** *(param: DampingParam, grad: bool) → dict*

Perform actual evaluation of the dispersion correction

**get_pairwise_dispersion** *(param: DampingParam) → dict*

Evaluate pairwise representation of the dispersion energy

**set_realspace_cutoff** *(disp2: float, disp3: float, cn: float)*

Set realspace cutoff for evaluation of interactions

**DampingParam**

class dftd3.interface.DampingParam(**kwargs)**

Abstract base class for damping parameters, representing a parametrization of a DFT-D3 method.

The damping parameters contained in the object are immutable. To change the parametrization, a new object must be created. Furthermore, the object is opaque to the user and the contained data cannot be accessed directly.

There are two main ways provided to generate a new damping parameter object:

1. a method name is passed to the constructor, the library will load the required data from the *s-dftd3* shared library.

2. all required parameters are passed to the constructor and the library will generate an object from the given parameters.

**Note:** Mixing of the two methods is not allowed to avoid partial initialization of any created objects. Users who need full control over the creation of the object should use the second method.
**RationalDampingParam**

```python
class dftd3.interface.RationalDampingParam(**kwargs)
```

Rational damping function for DFT-D3. The original scheme was proposed by Becke and Johnson\(^1\)\(^2\) and implemented in a slightly adjusted form using only the C8/C6 ratio in the critical for DFT-D3.\(^3\) The rational damping scheme has the advantage of damping the dispersion energy to finite value, rather than removing it at short distances.

**Note:** The zero damping function is retained for the three-body contributions from the ATM term.

---

**ZeroDampingParam**

```python
class dftd3.interface.ZeroDampingParam(**kwargs)
```

Original DFT-D3 damping function,\(^4\) based on a variant proposed by Chai and Head-Gordon.\(^5\) Since it is damping the dispersion energy to zero at short distances it is usually called zero damping scheme for simplicity. However, due to this short-range limit of the dispersion energy a repulsive contribution to the gradient can arise, which is considered artificial.\(^6\)

---

**ModifiedRationalDampingParam**

```python
class dftd3.interface.ModifiedRationalDampingParam(**kwargs)
```

Modified version of the rational damping parameters. The functional form of the damping function is *unmodified* with respect to the original rational damping scheme. However, for a number of functionals new parameters were introduced.\(^7\)

This constructor allows to automatically load the reparameterized damping function from the library rather than the original one. Providing a full parameter set is functionally equivalent to using the `RationalDampingParam` constructor.

---


**ModifiedZeroDampingParam**

```python
class dftd3.interface.ModifiedZeroDampingParam(**kwargs)
```

Modified zero damping function for DFT-D3. This scheme adds an additional offset parameter to the zero damping scheme of the original DFT-D3.

*Note:* This damping function is identical to zero damping for \( \text{bet}=0.0 \).

**OptimizedPowerDampingParam**

```python
class dftd3.interface.OptimizedPowerDampingParam(**kwargs)
```

Optimized power version of the rational damping parameters. The functional form of the damping function is modified by adding an additional zero-damping like power function. This constructor allows to automatically load the reparameterized damping function from the library rather than the original one. Providing the parameter \( \text{bet}=0 \) is equivalent to using rational the `RationalDampingParam` constructor.

### 3.3.5 QCSchema support

*Note:* For more information on using the DFT-D3 with QCSchema input see `dftd3.qcschema`.

### 3.3.6 ASE support

*Note:* For more information on using the DFT-D3 as ASE calculator see `dftd3.ase`.

### 3.3.7 PySCF support

*Note:* For more information on integrating DFT-D3 with PySCF see `dftd3.pyscf`.

---


3.3.8 Literature
d

dftd3, 16

dftd3.ase, 16

dftd3.interface, 23

dftd3.pyscf, 19

dftd3.qcschema, 17

PYTHON MODULE INDEX
D
DampingParam (class in dftd3.interface), 24
dftd3
    module, 16
dftd3.ase
    module, 16
dftd3.interface
    module, 23
dftd3.pyscf
    module, 19
dftd3.qcschema
    module, 17
DFTD3Dispersion (class in dftd3.pyscf), 19
DispersionModel (class in dftd3.interface), 24
dump_flags() (dftd3.pyscf.DFTD3Dispersion method), 21

E
energy() (in module dftd3.pyscf), 21

G
get_dispersion() (dftd3.interface.DispersionModel method), 24
get_pairwise_dispersion()
    (dftd3.interface.DispersionModel method), 24
grad() (in module dftd3.pyscf), 22

K
kernel() (dftd3.pyscf.DFTD3Dispersion method), 21

M
ModifiedRationalDampingParam (class in dftd3.interface), 25
ModifiedZeroDampingParam (class in dftd3.interface), 26
module
dftd3, 16
dftd3.ase, 16
dftd3.interface, 23
dftd3.pyscf, 19

O
OptimizedPowerDampingParam (class in dftd3.interface), 26

R
RationalDampingParam (class in dftd3.interface), 25
reset() (dftd3.pyscf.DFTD3Dispersion method), 21

S
set_realspace_cutoff()
    (dftd3.interface.DispersionModel method), 24
Structure (class in dftd3.interface), 23

U
update() (dftd3.interface.Structure method), 24

Z
ZeroDampingParam (class in dftd3.interface), 25