ALFE-H-general

Release 0.30.0

X.M.

Jun 13, 2026

CONTENTS:

1 Introduction **3**
- 1.1 Theoretical backgrounds . . . 3
- 1.2 Using this package . . . 4

2 Installation **7**
- 2.1 The `Heff` part . . . 7
  - 2.1.1 Requirements . . . 7
  - 2.1.2 Build with `CMake` . . . 9
- 2.2 The `heff_gen` part . . . 12
- 2.3 The `heffpy` part . . . 12

3 Running Simulations **15**
- 3.1 Overview . . . 15
- 3.2 Controlling the simulations . . . 16
  - 3.2.1 Minimal example . . . 16
  - 3.2.2 A more complex example . . . 18
  - 3.2.3 Tables . . . 19
  - 3.2.4 Structure specification . . . 19
  - 3.2.5 Physical environment . . . 21
  - 3.2.6 Simulation tasks . . . 23
  - 3.2.7 Initial structure . . . 26
  - 3.2.8 Energy model types . . . 26
  - 3.2.9 Integrator types . . . 27
  - 3.2.10 Controlling the energy components . . . 27
  - 3.2.11 Controlling the DoFs . . . 28
- 3.3 Parallel simulations . . . 28
  - 3.3.1 `OpenMP` parallelism . . . 28
  - 3.3.2 `MPI` parallelism . . . 29
- 3.4 Preparing input files . . . 31
  - 3.4.1 Specifying input files . . . 31
  - 3.4.2 Ewald summation file . . . 32
- 3.5 Explaining the results . . . 34
  - 3.5.1 Defining the physical quantities . . . 35

4 Optimization and single-point calculation **41**
- 4.1 Optimization procedures . . . 41
  - 4.1.1 Structural relaxation . . . 42
  - 4.1.2 Structural relaxation workflow . . . 43
  - 4.1.3 Nudged elastic band (NEB) . . . 43
  - 4.1.4 Optimization algorithms . . . 45

4.2 Single-point calculation . . . 46

5 Building a model **47**
5.1 Select reference structure . . . 47
5.2 Build the LWF basis . . . 48
5.2.1 Reciprocal-space method . . . 49
5.2.2 Real-space method . . . 60
5.2.3 Further details for LWF building . . . 61
5.3 Prepare unit cell metadata . . . 62
5.3.1 Unit cell specification file . . . 63
5.3.2 Generate unit cell YAML file . . . 66
5.3.3 Check selected local mode basis . . . 66
5.4 Generate interactions . . . 69
5.4.1 Basic concepts and hierarchy of interactions . . . 70
5.4.2 Alloy model level . . . 71
5.4.3 Bounding-only interactions . . . 72
5.4.4 Defining the cutoffs . . . 73
5.4.5 Interaction generation outputs . . . 78
5.5 Fit the parameters . . . 79
5.5.1 Precalculation . . . 80
5.5.2 On-the-fly learning . . . 83

6 Advanced techniques in model fitting **93**
6.1 Detailed configurations of OTF fittings . . . 93
6.1.1 Reference energy . . . 93
6.1.2 The long-range dipolar interaction in fitting . . . 94
6.1.3 Selecting parameters for fitting . . . 94
6.1.4 Controlling FP threshold . . . 95
6.2 Fitting with existing data . . . 97
6.2.1 `fit_from_data` . . . 98
6.2.2 `merge_otf_fit` . . . 98
6.2.3 `reuse_data_fit` . . . 99
6.2.4 Fitting with multiple data directories . . . 100
6.3 Multi-OTF fitting . . . 101
6.3.1 Reciprocal-space traversing strategy . . . 102
6.4 Multi-simulation-OTF iteration . . . 103
6.4.1 Test simulation configurations . . . 104
6.5 Interaction selection and compression . . . 105
6.5.1 Machine-learning-based model construction (MLCons) . . . 105
6.6 Energy boundedness . . . 107
6.6.1 Relations between SATs . . . 108
6.6.2 Boundedness condition . . . 109
6.6.3 Closed bounding procedure . . . 110
6.6.4 Open bounding procedure . . . 114
6.7 Practical fitting workflows . . . 117
6.7.1 OTF-bounding workflows (Nov2025) . . . 117

7 Theoretical Background **121**
7.1 Alloy model . . . 121
7.1.1 Interaction generation with mixed sites . . . 121
7.1.2 Computation of occupation components . . . 122
7.2 Electrical conditions . . . 125
7.2.1 `FixE` condition . . . 125
7.2.2 `FixD` condition . . . 126

7.2.3 FixRedD condition . . . 126
7.2.4 FixRedEbar condition . . . 126
7.2.5 FixRedMix condition . . . 127
7.3 Structural manipulations . . . 127
7.3.1 Approximately periodic sub-cell . . . 127

8 Utility programs **129**
8.1 C++ utility programs . . . 129
8.1.1 heff_build_info . . . 129
8.1.2 heff_dof_split . . . 129
8.1.3 heff_dipole_split . . . 130
8.1.4 heff_dof_average . . . 130
8.1.5 heff_dipole_average . . . 130
8.1.6 heff_make_param_index_map . . . 130
8.1.7 heff_diel_response . . . 131
8.1.8 heff_approx_periodic_sub_cell . . . 131
8.2 Python executable modules . . . 132
8.2.1 heff_gen.cell.extract_disp . . . 132
8.2.2 heff_gen.cell.expand_cell . . . 132
8.2.3 heff_gen.cell.gen_unitcell_yaml . . . 133
8.2.4 heff_gen.cell.build_atomic_struct . . . 133
8.2.5 heff_gen.cell.lwf_basic_params . . . 134
8.2.6 heff_gen.lwf.cui.gen_lwf_phonon . . . 134
8.2.7 heff_gen.lwf.cui.gen_multiple_lwf_phonon . . . 134
8.2.8 heff_gen.lwf.cui.lwf_dynamic_mass . . . 134
8.2.9 heff_gen.lwf.cui.mode_atomic_disp . . . 134
8.2.10 heff_gen.lwf.cui.plot_shell_lwf_band . . . 135
8.2.11 heff_gen.lwf.cui.show_lwf_info . . . 137
8.2.12 heff_gen.lwf.lwf_bec_tensor . . . 137
8.2.13 heff_gen.fit.cui.extract_occupation_history . . . 137
8.2.14 heff_gen.fit.cui.fit_from_data . . . 137
8.2.15 heff_gen.fit.cui.merge_otf_fit . . . 137
8.2.16 heff_gen.fit.cui.otf_fit . . . 137
8.2.17 heff_gen.fit.cui.multi_otf_fit . . . 137
8.2.18 heff_gen.fit.cui.param_null_space . . . 137
8.2.19 heff_gen.fit.cui.reuse_data_fit . . . 137
8.2.20 heff_gen.fit.cui.collect_otf_cells . . . 137
8.2.21 heff_gen.fit.cui.merge_otf_count_fps . . . 138
8.2.22 heff_gen.fit.plot.otf_plot . . . 138
8.2.23 heff_gen.fit.plot.plot_param_null_space . . . 138
8.2.24 heff_gen.fit.plot.reuse_plot . . . 138

9 Input reference **139**
9.1 Atomic structure files . . . 139
9.1.1 ASE structure files . . . 139
9.1.2 Phonopy structure file . . . 139
9.2 Simulation configuration file . . . 139
9.2.1 file_path table . . . 139
9.2.2 cell table . . . 140
9.2.3 ewald table . . . 142
9.2.4 physical table . . . 144
9.2.5 energy table . . . 152
9.2.6 init table . . . 156
9.2.7 simulation table . . . 161

9.2.8 `flow` table . . . 173
9.2.9 `optimization` table . . . 178
9.2.10 Physical quantity specification reference . . . 183
9.3 Fitting configuration files . . . 185
9.3.1 `struct_proj` table . . . 185
9.3.2 `fit_general` table . . . 185
9.3.3 `pader` table . . . 186
9.3.4 `regression` table . . . 187
9.3.5 `linear` table . . . 188
9.3.6 `bayesian` table . . . 189
9.3.7 `ridge` table . . . 192
9.3.8 `active` table . . . 192
9.3.9 `otf` table . . . 200
9.3.10 `intrusive_md` table . . . 203
9.3.11 `fp` table . . . 204
9.3.12 `merge_otf` table . . . 207
9.3.13 `reuse_data` table . . . 209
9.3.14 `mlcons` table . . . 211
9.3.15 `boundness` table . . . 217
9.3.16 `bounding_closed` table . . . 218
9.3.17 `bounding_open` table . . . 222

10 Output reference **225**
10.1 DoF output files . . . 225
10.1.1 Single-frame DoF file . . . 225
10.1.2 Multi-frame DoF file . . . 225
10.2 Site text file . . . 226
10.3 Local mode and strain text files . . . 226

11 Frequently Asked Questions **227**
11.1 Installation . . . 227
11.1.1 Why is the installation so “complex”? . . . 227
11.1.2 Why do we need such dependencies? . . . 227
11.1.3 Why cannot I use GCC 4.8.5 for building this package? . . . 228
11.2 usage . . . 228
11.2.1 Why are there so many different file formats? . . . 228
11.2.2 What is “undefined behavior”? . . . 228

12 Change log summaries **229**
12.1 Jun. 13, 2026 . . . 229
12.2 May 13, 2026 . . . 229

13 Development Notes **231**
13.1 Checklist for adding new physical quantities . . . 231
13.2 Checklist for adding new condition variables . . . 231

14 References **233**
14.1 Effective Hamiltonian related . . . 233
14.2 Symmetry, LWF building, and related . . . 233
14.3 Simulations and related . . . 233
14.4 Fitting algorithms and related . . . 233
14.5 Optimization algorithms and related . . . 233
14.6 Tools . . . 233
14.7 Other resources . . . 233

Bibliography 235

Welcome to the documentation of ALFE-H general package. The documentation is still under development.

CHAPTER

ONE

INTRODUCTION

This package provides the construction and simulation of the effective Hamiltonian for lattice dynamics problems. This effective Hamiltonian is an Taylor expansion of potential energy surface on a set of considered degrees of freedom (DoFs) around the reference structure, with expansion parameters fitted from first-principles (FP) calculations. This method is generally applicable for the lattice dynamics or atomic simulation problems where the considered structures contain only relatively small distortions about the reference structure. For example, the ferroelectric or relax ferroelectric phase transition, the lattice vibration, etc. The electronic structure and magnetic DoFs are *not* considered directly in current version. There is no limitation on the reference structure, except that it must be a single crystal structure. Solid solutions are also supported (see below for more details).

The effective Hamiltonian method (specifically, implemented in this package) has the following highlighted features:

- Very fast and applicable to super-large-scale atomic structures (more than 10^7 atoms).
- Highly automatic construction.
- Specific physical meanings of the expansion parameters.

The effective Hamiltonian could be applied to compute:

- structure phase transition simulation, for example, ferroelectric phase transition;
- response to external conditions, for example, temperature, electric field (static or dynamic), stress;
- structural optimization;
- phonon calculation;
- and so on.

For more applications, please see the publications on ALFE-H homepage.

1.1 Theoretical backgrounds

The effective Hamiltonian is built up on the Taylor expansion on the selected DoFs around the reference structure. Here, the reference structure is a crystal structure. The reference structure should be chosen such that the considered structures could be built with relatively small distortions (i.e., sufficiently close to the considered structures) and should have as high symmetry as possible. The reference structure defines a *unit cell* for the effective Hamiltonian simulation, including the unit cell basis and the *sites*. The *sites* are the positions in the unit cell that occupied by atoms.

The effective Hamiltonian simulation is conducted on a *supercell*, which includes a number of unit cells, and defined by a supercell matrix, forming a parallelepiped lattice (where each unit cell is considered as a lattice point here). The supercell may have different dimensionalities, ranging from 3d (bulk) to 2d (film), 1d (rod) and 0d (dot).

For solid solutions, the reference structure may contain some "mixed sites", where multiple elements could appear at one site in different unit cells. These mixed sites should be considered as *one single site* in the reference structure, and (conceptually) occupied by a virtual mixed atom. For example, for the complex perovskite solid solution $Ba(Zr,Ti)O_3$ (BZT), the Zr and Ti could be considered as a mixed site, occupying the same site in different unit cells. In the reference

structure, it could be conceptually understood as a mixed virtual atom $\langle B \rangle$, and the reference structure is Ba$\langle B \rangle$O$_3$, sharing the same symmetry with the corresponding simple perovskite structure.

The DoFs considered in the effective Hamiltonian include:

- Local mode distortions. The localized atomic displacement of some pre-determined form. Each "local mode" typically includes the collective motions of some atoms around a *center site*, but in a limited range (indicating the atoms far from the center site do not move).
- Strain distortion. The strain tensor of the supercell.
- Occupation index, represents the actual element occupying each of the mixed sites. For example, in the BZT model, indexing 1 or 2 represents Zr and Ti atoms, respectively.

Note

The expansion on local mode distortions and strain distortions are directly Taylor expansions, defined in usual ways. The "expansion" on the occupation index are somewhat different, where the *value* of the occupation indexes (1 or 2 in the example above) are not used directly in the expressions. For more details, see [ALFE-H-2025] and references therein.

The expansion coefficients in the Taylor expansion (introduced above) should be fitted by first-principles (FP) calculations. There are often referred to as "parameters" of the effective Hamiltonian. Conventionally, these parameters are fitted by constructing some structures with pre-determined special atomic displacement patterns and performing FP calculations on them. This fitting strategy could be rather complex and the result may be unreliable for complex interactions. Recently, we proposed an active-learning-based fitting process to fit these parameters, which is implemented in this package. For more information about this fitting strategy, please refer to [ALFE-H-2025].

This package provides a generalized implementation to the effective Hamiltonian for any possible reference structures, in contrast to some previous works (where the proposed effective Hamiltonian is only applicable to perovskite structures). The local mode basis and interaction expressions (Taylor expansion) are found according to the symmetry of the reference structure, and the precision of the effective Hamiltonian (number of terms included) are controlled by various cutoff conditions. The parameters are fitted using the active-learning approach, making it relatively easy to build a model for a specified material.

1.2 Using this package

If an effective Hamiltonian model for the material system to be studied is available (including the unit cell metadata, interactions and parameters), the simulations could be performed directly with this model, please refer to *Running Simulations*. Otherwise, if the model is not available, one could build this model using this package, please refer to *Building a model*. We have provided some models as examples, available at `tests/testdata/models` directory in the `Heff` package source root. These models are provided for technical purposes only, and are not verified physically (unless otherwise specified). Extra attentions should be paid if one would like to use these models to do some physical predictions. Here we introduce some technical issues that shared commonly by most of the programs provided by this package.

Most of the programs provided by this package are controlled by one or more input files. These input files may be generated by another program, or be hand-edited. Several different file formats are selected for different purposes (see *Why are there so many different file formats?* for more information). Specifically, most of input configuration files that are expected to be edited by the user are in TOML format, which is a configuration file format that is rather simple and easy to read. Please see their homepage for more information. We will use the standard terms defined by TOML, like "table", "array", representing the TOML types, throughout this documentation.

Most of the TOML configuration files in this package has its "echoed" file after running the program, for verifying whether the input by the user is parsed appropriately. For example, for running simulations, the input configuration file is typically `heff_config.toml`, and an "echoed" file `out_heff_config.toml` will be generated after running

`heff_run_sim`. This echoed file contains almost all possible keys in the input file. If the input contains some keys that are not parsed by the program (typically caused by wrong spelling), the package may provide warning or error message (depending on different programs). Please check the warning messages carefully to ensure the input was parsed as predicted.

The outputs from the programs are messages on standard output, the output files, and possibly figures in some programs. Some output files may be later used by the programs in this package, and some of them (in general file formats, like TOML, YAML or HDF5) may also be readable by other tools.

Important

This package is still under active development. At this stage, we do not provide *any* guarantee about *any* compatibility between different versions, either forward or backward, including but not limited to the APIs, ABIs, input and output files. The input requirement, default values of input variables, and interpretation of the input variables may be changed at any release version in any form. Besides, using different versions for different part of this package results in **undefined behavior**. Do *not* try to do a partial updating unless you know exactly what will happen.

Important

For any input, strictly follow the required format and value ranges as required by this document, or required by physical meanings. For input values or configurations related to physical quantities, it is the user's responsibility to make sure the values are reasonable from the physical point of view. Any violation to the file format and/or range requirement for the values may result in undefined behavior. Any unreasonable input values for the physical quantities may result in unreasonable behavior or output of this program.

CHAPTER

TWO

INSTALLATION

The `ALFE-H-general` project could be divided into three main parts:

- The pure C++ part, to be called `Heff`, implementing the simulation with effective Hamiltonian, and supports to the parameter fitting;
- The pure Python part, mainly the `heff_gen` package, implementing the algorithms for building up an effective Hamiltonian for specified material;
- The C++-Python binding part, i.e., the `heffpy` package.

We suggest all users to install all of these three parts. For the users need only to perform simulations with existing models, the minimal requirement is only the `Heff` part, but there are still some utility tools requiring the other two parts. For the users that would like to build a new effective Hamiltonian model, all of the three parts are needed.

Important

Always make sure the three parts have consistent version (as obtained from the same release package). Do *not* try to mix different versions, or do a partial update to any part or any individual file, unless you know exactly what will happen. Any mixing of different versions results in undefined behavior.

2.1 The `Heff` part

2.1.1 Requirements

The `Heff` part is a pure C++ package and should be installed with `CMake`. The following requirement should be satisfied:

- A C++ compiler that supports (at least part of) C++17 standard and supported by the `Boost.Describe` library and `magic_enum` library (since version 0.27).
- CMake version 3.23 or later.

Important

Due to the dependency on `Boost.Describe` library, not all C++ compilers are supported. Please check Boost.Describe documentation for more information.

Important

Due to the dependency on `magic_enum` library since version 0.27, some non-standard features of the compilers are required. Please check magic_enum repository for more information.

Tip

The default compiler on some commonly used systems (for example, GCC 4.8.5) does **not** meet the requirement of C++17 and thus cannot be used. Meanwhile, according to our test, the Intel (R) C++ Compiler Classic (`icc` and `icpc` compilers) is also not suitable, as it is not supported by `Boost.Describe`. Please see *Specifying compiler* for a guide of specifying compilers.

The following environments are tested:

- Windows 10, MSVC 2022
- Linux CentOS 7, GCC 15.2.0, 14.1.0, 12.3.0, 8.3.0
- Linux CentOS 7, LLVM Clang 19.1.5 (using libc++)
- Linux CentOS 7, Intel LLVM ICX compiler 2022

The following libraries are required and should be installed in advance, and make sure they could be found by CMake:

- Boost version 1.77 or later. The following components are needed:
  - `headers`
  - `unit_test_framework` (optional but recommended)
  - `program_options` (required if `AVOID_BOOST_PROGRAM_OPTION` set to `OFF`, which is the default setting)
- FFTW. FFTW3 is required. Version 3.3.10 is used in development.
- HDF5. Version 1.14.4 is used in development.

The following libraries are required and bundled in the source of this project. Typically, it is convenient to use the bundled source. However it is also possible to install them elsewhere (in advance) and let CMake find it by specifying corresponding CMake variable.

- Eigen3. Version 3.4.90 is used in development. Pass `-DUSE_EXTERNAL_EIGEN=ON` to CMake if you would like to use externally installed version.
- TOML11 version 4.0 or later. Header-only library for parsing and writing TOML files. Pass `-DUSE_EXTERNAL_TOML11=ON` to CMake if you would like to use externally installed version.
- HighFive header-only library for parsing and writing the HDF5 files, as a wrapper to HDF5 C-APIs. Version 3.1.1 is used in development. Pass `-DUSE_EXTERNAL_HIGHFIVE=ON` to CMake to use externally installed version.
- yaml-cpp library for parsing and writing YAML files. Pass `-DUSE_EXTERNAL_YAML_CPP=ON` to CMake to use externally installed version.
- nlohmann-json library for parsing and writing JSON files. Pass `-DUSE_EXTERNAL_JSON=ON` to CMake to use externally installed version.
- spdlog library for logging. Pass `-DUSE_EXTERNAL_SPDLOG=ON` to CMake to use externally installed version.
- magic_enum library for the reflection of enumerators. Pass `-DUSE_EXTERNAL_MAGIC_ENUM` to CMake to use externally installed version. (Since version 0.27)

Known issues with compilers

Here are some issues with some compilers that we found in our tests. They are listed only for reference. We are not sure whether these problems comes from other detailed configuration issues, etc.

If Clang 19 compiler is used, the custom exceptions raised in C++ codes sometimes could not be converted correctly to Python side through Pybind11. This may be an issue related to pybind11. This causes the OTF fitting stops directly when FP calculation fails.

If LLVM OpenMP library is used (in our tests, Clang and ICX compilers used), the OpenMP reduction has initialization issues for some special cases. This is automatically fixed (at least in our tests) with the CMake option `HEFF_OMP_REDUCE_INIT_NO_ORIG`. Typically the user should not worry about this.

2.1.2 Build with `CMake`

To build the `Heff` part with CMake, first create and enter a building directory, typically as a subdirectory of source code directory (which contains the `CMakeLists.txt` file).

```
mkdir build
cd build
```

Then configure at this directory

```
cmake .. -DCMAKE_INSTALL_PREFIX=/path/to/your/install/directory
```

Here, `..` indicates the root directory of the `Heff` source code that containing `CMakeLists.txt` file. `/path/to/your/install/directory` indicates the directory you would like to install this project to, which should be chosen according to your system. Extra parameters and/or environment variables could be set. For more details, see *Frequently used configuration arguments*.

After configuration completed successfully, build the project with

```
cmake --build .
```

Then one can optionally (but strongly recommended to) run the test with

```
ctest
```

Then install with

```
cmake --install .
```

If every thing goes well, the `Heff` part should now be installed successfully. Let `HEFF_INSTALL_ROOT` represents the path in which the `Heff` is installed (i.e., that previously set as `CMAKE_INSTALL_PREFIX`).

Setup environment

To use the package, one should set up the environment for the shell to find the module. There are several different means to do this, and one could choose either one suitable for the system.

Use ``module`` commands. On Linux with `module` available, it is recommended to use the `module load` command to load the environment settings. To do this, first add the path `${HEFF_INSTALL_ROOT}/etc/modulefiles` to environment variable `MODULEPATH` as

```
# assume that the HEFF_INSTALL_ROOT variable is set in advance
export MODULEPATH=$HEFF_INSTALL_ROOT/etc/modulefiles:$MODULEPATH
```

Or alternatively, copy the contents in `${HEFF_INSTALL_ROOT}/etc/modulefiles` to some path that is included in `MODULEPATH`. Then the `Heff` package could be loaded with commands like `module load heff-general/v0.1.0_GNU` (here the specific module name depends on the version and the compiler you have used in building).

Use the ``set_heff_env.sh`` script. On Linux, is is also possible to use the script provided in the install directory directly to set up the environment:

```
# Please set HEFF_INSTALL_ROOT to your install directory in advance
source ${HEFF_INSTALL_ROOT}/etc/set_heff_env.sh
```

Set the environment variables manually. On Linux, add the path `${HEFF_INSTALL_ROOT}/bin` to environment variable `PATH`, and add the path `${HEFF_INSTALL_ROOT}/lib64` to environment variable `LD_LIBRARY_PATH` with the commands like this:

```
export PATH=${HEFF_INSTALL_ROOT}/bin:$PATH
export LD_LIBRARY_PATH=${HEFF_INSTALL_ROOT}/lib64:$LD_LIBRARY_PATH
```

On windows, add the paths `${HEFF_INSTALL_ROOT}/bin` to environment variable `PATH`.

After setting up the environment, one could check it by typing

```
heff_build_info
```

It is supposed to print some information about the version and building environment of this package if the installation is successful and the environment is properly set up.

Frequently used configuration arguments

In the CMake configuration step, it is possible to pass some variables to CMake or set some environment variables that affect the configuration behavior. Here we introduce some frequently used arguments/variables of this package. For some commonly used arguments/variables, please check the cmake documentation. For all options used in this package, please see the root `CMakeLists.txt` file of this package.

The arguments passed to `CMake` could be specified using the syntax `-D<argument_name>=<value>` in the `cmake` command line. For example, the following command will set the `CMAKE_INSTALL_PREFIX` variable in CMake to `/opt/Heff`.

```
cmake .. -DCMAKE_INSTALL_PREFIX=/opt/Heff
```

Specifying compiler

Typically, `CMake` would find the C++ compiler automatically. This automatically found compiler may not be suitable for building this package (for example, in some commonly used Linux systems, the default compiler GCC 4.8.5 is **not** able to compile this package). In this case, one needs to specify the compiler by either environment variables or `CMake` variables. The CMake variables `CMAKE_CXX_COMPILER` and `CMAKE_C_COMPILER` could be used to set the compilers. For example, if one would like to use Intel LLVM compiler (`icx` and `icpx`), the following command could be used:

```
cmake .. -DCMAKE_CXX_COMPILER=icpx -DCMAKE_C_COMPILER=icx
```

The compilers could also be specified with the environment variable `CC` and `CXX`. For example, on Linux,

```
CC=clang CXX=clang++ cmake ..
```

on Windows command line prompt,

```
set CC=gcc
set CXX=g++
cmake ..
```

Optional features

The following `CMake` variables controls some optional features/functionalities of this package:

- `ENABLE_OPENMP` enables the OpenMP parallelism on shared memory architectures. Default: `ON`.

> **Note**
>
> OpenMP library 4.0 or later is required. If your OpenMP version is too old, the configuration would fail. Consider using a newer C++ compiler or set `ENABLE_OPENMP` to `OFF`.

Performance tuning

The following `CMake` variables controls the building options of this package, which may affect the performance.

- `HEFF_NATIVE_ARCH`: Try to use the `-march=native` compiler flag (if available) to optimize. If the compiler does not support this flag, nothing will be done. Default: `OFF`

> **Warning**
>
> Binaries built with this option is only able to run on the platforms with same architecture. Some other compilation issues may arise.

- `EIGEN_USE_MKL EIGEN_USE_BLAS EIGEN_DONT_PARALLELIZE`: Defines the corresponding macros for the `Eigen` library. Default: `OFF`. Please see the `Eigen` document: https://eigen.tuxfamily.org/dox/TopicUsingBlasLapack.html https://eigen.tuxfamily.org/dox/TopicUsingIntelMKL.html https://eigen.tuxfamily.org/dox/TopicMultiThreading.html
- `HEFF_USE_MKL`: Use the MKL library for fast Fourier transformation (FFT) implementation. By default, FFTW library is used for FFT. If this option is set to `ON`, MKL will be used instead. Default: `OFF`.

> **Warning**
>
> If you would like to use these options, it is strongly recommended to do performance test (on your problems to solve) first. The performance can vary from case to case.

Dependency control

The following `CMake` variables are typically not needed, but they are provided for the cases where one encounters some dependency issues. Please see the explanations below.

- `NO_OMP_FFTW`: Do not link to `FFTW` library with `OpenMP` support, even if the codes from the package itself is compiled with `OpenMP` support. This is useful when one has some issues with `FFTW` library with `OpenMP` support (not available or unable to link against, etc). Default: `OFF`.

> **Warning**
>
> Turning on this option is likely to reduce the performance of `FFTW` library in parallel case.

- `HEFF_TEST_WITH_SHARED_LIBS`: The `Boost.Test` could be used within a header-only manner or a dynamically-linked-library (DLL) manner. If this option is set to `ON` (the default), the package would prefer to select the DLL manner (unless the `Boost::unit_test_framework` could not be found by `CMake`). Otherwise, the package would switch to the header-only manner directly. Both the header-only and DLL manners work with the same effect, however using the DLL may make the compiling process faster. This option is only defined for some special cases, for example, the `Boost::unit_test_framework` could be found by `CMake` but cannot be linked to, possibly caused by mismatched compiler version etc.
- `AVOID_BOOST_PROGRAM_OPTIONS`: Avoid the dependency on the `Boost::program_options` library. Default: `OFF`. `Boost::program_options` is a binary library. As people may encounter problems building and installing the Boost binaries, we provide this option to avoid such dependency, at the cost of skipping some executables of this package (including `heff_dof_potential`, etc), by setting this variable to `ON`.

> **Warning**
>
> If this is set to `ON`, some executables will not be built.

2.2 The `heff_gen` part

The `heff_gen` part is a pure Python package, which should be very easy to be installed. This package requires Python 3.10 or later to be installed. Which is available at Python.

> **Tip**
>
> We have tested our installation workflow using the CPython implementation (available at Python). We do **not** recommend to use anaconda because we have encountered some problems while testing with it, though it is still applicable in principle.

With suitable version of Python and `pip` available, switch to the directory of `heff_gen` that containing the `pyproject.toml` file, and type the following command

```
python3 -m pip install .
```

to install this package. If everything goes well, the package should be installed successfully.

This package requires various third-party Python packages, with some of them depending on others, and possibly require some specified versions. We try to keep maximum compatibility when and where possible while developing our codes, and provide known version requirement in `pyproject.toml`. However, it is still possible to encounter version conflicts, both during installation and running. Here is a possible list of installed packages, on which the `heff_gen` package works well, which could be a reference for preparing the environment. The Python version is 3.12.4.

2.3 The `heffpy` part

The `heffpy` part provides the Python binding for the C++ code, making Python codes available to the `Heff` part written in C++. To install this package, one should have the `Heff` part properly installed and its environment properly set up. Please refer to *Setup environment* for more information. The key is to make `CMake` be able to find the `Heff` part installation, namely the `Heff_general` package defined in CMake. This could be done by setting the `CMAKE_PREFIX_PATH`

environment variable (this is done if one has used the environment setting up script or loaded the `module`) or passing the `Heff_general_DIR` variable to `CMake`.

To install, switch to the root of `heffpy` package (the `python` directory under the root of `Heff`, which contains the `pyproject.toml` andd `CMakeLists.txt` files), and type the following command:

```
python3 -m pip install . -v
```

It is also possible to select different compilers by setting environment variables, for example, on Linux:

```
CC=gcc CXX=g++ python3 -m pip install . -v
```

It is strongly recommended to use the same compiler as that have been used in `Heff` installation.

For Windows users, if dynamically linked libraries (DLLs) are used, it may be necessary to copy these DLL files to the installation directory. Here we provide a script for reference (please set the variables according to your installation directories on your computer before using it):

```
rem ---------------- Manual configure area --------------------
rem !!!!!!!! Set these variables according to directories on your computer !!!!!!!!!
set PYTHON_ROOT=D:\Python\Python312
set HEFF_INSTALL_ROOT=E:\Cpp\Heff_general\Heff_general\out\install
set CMAKE_PREFIX_PATH=E:\Git\linalg\eigen_install\share\eigen3\cmake;E:\Git\Boost\boost_
↪1_84_0_install\lib\cmake\Boost-1.84.0;%HEFF_INSTALL_ROOT%/lib/cmake/Heff_general
rem ------------ End of manual configure area -----------------

set PYTHON_EXEC=%PYTHON_ROOT%/python.exe
set PYTHON_HEFFPY_INSTALL_PATH=%PYTHON_ROOT%\Lib\site-packages\heffpy

pip install . -v

copy %HEFF_INSTALL_ROOT%\bin\heff_general.dll %PYTHON_HEFFPY_INSTALL_PATH%
copy %HEFF_INSTALL_ROOT%\bin\spdlog.dll %PYTHON_HEFFPY_INSTALL_PATH%
```

CHAPTER

THREE

RUNNING SIMULATIONS

This document introduces how to run effective Hamiltonian simulation **with existing models** for one specified material (including interactions and parameters). For building a model for specified material, please refer to *Building a model*.

3.1 Overview

To run a simulation, one typically chooses a working directory (folder) including all the input files. The following input files are required:

- Unitcell file, by default `heff_unitcell.yaml`;
- Interaction file, by default `heff_interactions.json`;
- Parameter file, by default `heff_params.yaml`;
- Simulation configuration file, by default `heff_config.toml`;
- Ewald summation file (optional but strongly recommended to be prepared in advance), with filename starts with `EW_DD` and ends with `.h5`.

The unitcell file and interaction file should be obtained from *model building* process, while the configuration file should be (typically) hand edited. Please see *Controlling the simulations* for more information about this control file. Some examples would also be given therein.

Important

Please make sure all the input files are consistent with each other. The program will do some checks upon start up, but not all inconsistent cases could be found by this checking process.

Tip

Some of the input files could be assigned to other filename or directory. Please see *Specifying input files* in this document.

With all files prepared, run the simulation by typing

```
heff_run_sim
```

If `OpenMP` is enabled in building process, the number of threads may be specified by (the standard way of using `OpenMP`) setting the environment variable `OMP_NUM_THREADS`, for example, on Linux:

```
OMP_NUM_THREADS=4 heff_run_sim
```

or on Windows cmd:

```
set OMP_NUM_THREADS=4
heff_run_sim
```

The line printed on the screen with following format confirms the number of threads is set correctly:

```
[info] OpenMP is enabled with 4 threads
```

After some initializations, there will be some logs printing on the standard output `stdout` (typically just the screen) like

```
[2025-01-04 11:42:41.975] [info] STEP      0, Htot=0.078427, Hpot=0.000000, Hkin=0.
↪078427, T=303.074292
[2025-01-04 11:42:43.115] [info] STEP    200, Htot=0.078440, Hpot=0.016781, Hkin=0.
↪061659, T=238.275516
[2025-01-04 11:42:44.082] [info] STEP    400, Htot=0.078443, Hpot=0.023738, Hkin=0.
↪054705, T=211.400786
```

These output lines are provided with a pre-defined interval, showing some basic information in the simulation. This will be called "snap" output. For this example case, the molecular dynamics (MD) simulation, the step number, total enthalpy `Htot` and its potential part `Hpot` and kinetic part `Hkin` (in eV per unit cell) as well as the temperature (in Kelvin) are provided. A more detailed snap output is given in the `snap.txt` file (by default) with the same interval, providing more information, including the energy contribution from each part.

Each time the simulation program is called, one or more *simulation runs* will be conducted. Typically, each simulation run is performed in a specified physical condition (including temperature, electric field, pressure, etc). These simulation runs could be identified by the `sim_id` column in the snap output file.

After the simulation, there will be some output files, containing the result of the simulation and some other related information. The most frequently used output file is the thermodynamics statistics output file, by default `heff_thermo.txt`, including the thermodynamics average of the physical quantities (e.g., enthalpy, polarization, etc) collected in the simulation, where each line corresponds to one single simulation run. Please see *Explaining the results* for more information.

3.2 Controlling the simulations

This section mainly introduces the simulation configuration input file (`heff_config.toml`), which is the most basic input files for the simulation. This file is in TOML format. For more information about the TOML format, please visit TOML official webpage. This document will not cover this part.

3.2.1 Minimal example

Here is the minimal example of the configuration file for running the simulation, which defines only a small amount of keys that are most basic for the simulation. Many other keys are left as default.

```
# heff_config.toml
# The lines starts with "#" are comments, and will not be processed by the program.
[cell]
supercell_diag = [8, 8, 8]   # Supercell matrix (diagonal form for this example)

[flow]
```

(continues on next page)

(continued from previous page)

```
task = "Single"     # Single simulation
alg = "MD"          # Molecular dynamic simulation

[physical]
temperature = 300   # Kelvin
```

In the input example above, three TOML *tables* are defined. The `[cell]` table defines the supercell for this simulation. The supercell matrix, is defined in line with that used in the phonopy library. More specifically, let a_s, b_s, c_s be the supercell basis vectors, and a_u, b_u, c_u be the unit cell basis vectors, the supercell matrix $P \in \mathbb{Z}^{3\times3}$ is defined so that

$$(a_s, b_s, c_s) = (a_u, b_u, c_u)P.$$

Important

The lattice vectors a_s, b_s, c_s and a_u, b_u, c_u are defined as *column* vectors.

In the input file, the supercell matrix P could be specified in two different ways. The first is to provide the full matrix using the `supercell_matrix` key, and the second is to provide only the diagonal part (as in this example) using the `supercell_diag` key, if the supercell matrix is actually a diagonal matrix.

The `[flow]` table defines the workflow that the program will run. A *workflow* is a series of simulation runs organized in some pre-defined manners, typically with varying physical conditions. There are several types of manners provided by this package, which could be specified with different values for the `task` key. For example,

- `task = "Single"`: Single simulation run with provided conditions.
- `task = "Repeat"`: Repeated simulation for several times (set by the `flow.repetitions` key) with the same condition.
- `task = "Tsweep"`: Temperature sweep with a series of equal-spacing temperature values (controlled by the `flow.temp_start`, `flow.temp_end`, `flow.temp_step` keys).
- `task = "Tcycle"`: Temperature cycle, with a sweep from `flow.temp_start` to `flow.temp_end` and then back to `flow.temp_start`.
- `task = "Esweep"`: Electric field sweep.
- `task = "Ecycle"`: Electric field cycle.
- `task = "Eloop"`: Electric field loop, consistent with typical dielectric loop simulation.

Note that some extra keys may be required for some of the `task` values.

The algorithm set by the `flow.alg` key defines the simulation algorithm. In this case, it is `MD`, indicating molecular dynamics (MD) simulation. Some extra keys or tables may bee able to/required to set for some of the `alg` values.

The `[physical]` table defines the physical conditions for the simulation, including the temperature, electric field, epitaxial strain, etc. In this case, only the temperature is specified.

Note

Some of the physical environment settings in the `[physical]` table may be overwritten by the settings in `[flow]` table.

3.2.2 A more complex example

Here we provide a slightly more complex example for the simulation configuration input file.

```
[cell]
supercell_matrix = [
    [ 6, 6, 0],
    [-6, 6, 0],
    [ 0, 0, 6],
]

[physical]
pressure = -2   # GPa

[simulation]
sim_steps = 20000
snap_freq = 500

[md]
ensemble = "NPT"
adjust_temp_begin = true

[flow]
task = "Tsweep"
alg = "MD"

temp_start = 400
temp_end = 10
temp_step = -10
```

In this configuration file, we use the `Tsweep` task to conduct a series of simulation runs at different temperatures. The temperatures are (in order) 400 K, 390 K, ..., 20 K, 10 K, as determined by the starting temperature `temp_start`, end temperature `temp_end`, and the step for each change `temp_step`. The negative value of `temp_step` indicates the temperature is decreased step-by-step in the simulation.

In the `[physical]` table, the external pressure is set to -2 GPa by the `pressure` key. The `temperature` key need not to be set, as the temperature is controlled by the `Tsweep` task and will be set automatically for each simulation run.

In the `[simulation]` table, the `sim_steps` key set the number of steps (MD steps in this case) to be performed for each single simulation run. The `snap_freq` key determines the interval (in MD steps) between two snap outputs during the simulation.

The `[md]` table is provided for `alg="MD"` case for some detailed settings. Here, the ensemble is set to `NPT` among the two available options `NPT` and `NPH`, enabling the Evans-Hoover thermostat (see [HeffMD2008]) to control the temperature. The key `adjust_temp_begin` tells the simulator to adjust the temperature (by rescaling momenta) at the beginning of each single simulation run, which is required for the `Tsweep` task.

> **Tip**
>
> Changed since Jul. 21, 2025: The `simulation.md.adjust_temp_begin` will be automatically set to `true` if `flow.task = "Tsweep"` is selected in `heff_run_sim`.

3.2.3 Tables

The simulation configuration keys are organized as several TOML tables at top level. Each table includes some configuration keys for some related aspects. Some tables may include nested TOML tables and/or arrays. The top-level tables are as follows:

- `[file_path]` Controls the base paths for resolving the relative paths.
- `[cell]` Controls the supercell for the simulation, including the supercell matrix, the unit cell specification file path, etc.
- `[physical]` Controls the physical environment for running the simulation, including the temperature, electric field, pressure, and epitaxial strain, etc.
- `[energy]` Controls the file paths for loading the interaction/parameter information, and the components of the energy (enable or disable some part of the energy, etc).
- `[init]` Controls the initialization for the simulation, including (optionally) load the site distribution and/or mode configuration, etc.
- `[simulation]` Common configurations for the simulation that shared by all (or most) of the simulation algorithms, including the number of steps, snap frequency, and output file control, etc.
- `[flow]` Controls the simulation workflow, including the simulation task, algorithm, energy computation type (model type), etc.
- `[ewald]` Controls the Ewald summation process, which will be used only if such calculation is conducted. See *Ewald summation file* for more information.

This document will be organized based on "topics" of controlling the simulation. For a more detailed reference of input variables, please refer to *Input reference*.

3.2.4 Structure specification

The structure, or the cell, of the Heff simulations, includes the atomic structure of the reference structure and the LWF information (including LWF basis, symmetry information, etc). During the simulations, the unitcell structure of reference structure is specified by the unitcell input file (`heff_unticell.yaml` by default), and the supercell information is specified by the `[cell]` table in the config file (`heff_config.toml`). The former is typically generated in the model building stage (see *Building a model*) and is human-readable, though typically not hand-edited. During the simulation, only some basic information of the LWFs are required, and the detailed basis information (in the LWF data files, as used in model building stage) is not needed.

The unitcell basis (lattice vectors) are specified by the `basis` key in the unitcell file, in a row-vector format (the three lines correspond to a_s, b_s, c_s vectors, respectively). Such basis is defined in the Cartesian coordinate with the unit of Angstrom. The supercell matrix (P) is specified by the `supercell_matrix` or `supercell_diag` key in config file, as described in *Minimal example*.

By default, the simulated system is a bulk system with periodic boundary condition (PBC) applied to all of the three axes. Nevertheless, all possible dimensionalities are supported by this package, including 3d (bulk), 2d (film), 1d (rod) and 0d (dot), with some restrictions to the low-dimensional systems. To specify the dimensionality, there are two possible ways, by setting either the `periodic` or the `vacuum_bound` in the `cell` table.

The first way for specifying the dimensionality is to set the `periodic` key that directly specify whether PBC is applicable on each of the axes, by providing an array containing three boolean values. For example, the following setting specifies a supercell of $12 \times 12 \times 12$ with PBC applied on a and b axes but not on c axis, which is a film sample growing along the c axis. In this case, the c axis is finite, with surfaces appeared on both side of the sample.

```
[cell]
supercell_diag = [12, 12, 12]
periodic = [true, true, false]
```

Another way for specifying the dimensionality with more flexibility (but more complexity as well) is to set the `vacuum_bound` key with an array containing three floating-point numbers. Such vector v specified in the *fractional coordinate* about the *unitcell* basis defines a point x_v in the supercell such that

$$x_v = a_s + b_s + c_s - v,$$

where a_s, b_s, c_s are the supercell basis vectors (see *Minimal example*). Any point with at least one component projected on the *supercell basis* larger than the x_v point is considered in the vacuum layer. The axes with vacuum layer inserted (i.e., the corresponding component of the fractional coordinate about the supercell basis of x_v vector *less than* 1.0) are finite and do not have PBC.

Important

The user input of the `vacuum_bound` must ensure that all the components of x_v vector are in the $[0, 1]$ range. Otherwise, the program will exit with error message.

Tip

The `vacuum_bound` could be specified simultaneously with the `periodic` key. In this case, the axes determined to be non-periodic by *any* of the two criteria are treated as non-periodic ones.

The `vacuum_bound` key provides more functionality than the simple `periodic` key, as one can specify not only the non-periodic axis, but also the vacuum *sites*. In fact, it is possible to include only a part of the unitcells on the edge (i.e., only some of the sites, but not all sites) in the vacuum layer. This is especially useful for the models where the termination layer is specified, for example, for the perovskite ABX_3 films, one may want to specify that both surfaces terminate with A site.

Example: the following example defines a film model with non-periodic c axis. Suppose the unitcell is a perovskite unitcell with A site at [0, 0, 0] and B site at [0.5, 0.5, 0.5] (both in unitcell fractional coordinate), then the both sides of the sample terminate with A site. [Actually, any value ranging from 0.5 (not included) to 1.0 (included) for the third component of `vacuum_bound` yield the same result. We choose the value 0.6 that is away from both limits to avoid the influence of numeric errors.]

```
[cell]
supercell_diag = [12, 12, 12]
vacuum_bound = [0, 0, 0.6]
```

Tip

Different from previous perovskite implementation, to define a low-dimensional model, there is generally no need to insert vacuum layers. For simplicity and computational efficiency, it is strongly *discouraged* to insert vacuum layers containing only *complete* unitcells, as this has no effect at all.

For the 2d (respectively, 1d) case, although it is free to choose the non-periodic (respectively, periodic) axis to be any of the three crystal axes, it is strongly recommended to choose the c (respectively, a) axis as the non-periodic (respectively, periodic) axis, as such configuration is consistent with the requirement for running fast Fourier transformation (FFT) (where possible and required), which may greatly speed up the simulation.

It should be noted that, if long-range dipolar interaction is required, some requirement are imposed for 2d and 1d models, see *Ewald summation file* for more details.

3.2.5 Physical environment

Basically, the physical environment could be controlled by using the `[physical]` table.

Temperature

The temperature could simply be set by using the `temperature` key, which accepts a floating-point number in Kelvin, for example, `temperature = 300` set the temperature to 300 K (near room temperature).

Strain and misfit strain

Physically, the strain is defined as the tensor ϵ that deforms the supercell as

$$(a, b, c) = (\epsilon + \mathbb{1})(a_0, b_0, c_0), \tag{3.1}$$

where $\mathbb{1}$ is the 3×3 identical matrix, a, b, c are the lattice vectors (in column vector form and in Cartesian coordinate), and a_0, b_0, c_0 are the lattice vectors of the reference structure within the same form. In this project, the strain is expressed using the well-known Voigt notation, where a six-component array η is used to express the strain, with $\eta_1 = \epsilon_{11}$, $\eta_4 = \epsilon_{23} + \epsilon_{32}$, etc.

The epitaxial strain (or any other type of imposed static strain) could be set by the `[physical.misfit]` table. Technically, a misfit strain condition indicates that one or more component of the (possibly transformed) strain tensor to be fixed at some given value. The settings are only enabled if the `enabled` key is set to `true` (the default value is `false`).

The `free_components` key set the components that are free to be relaxed, accepting 6 boolean values in Voigt index, where `true` indicates the corresponding component is free while `false` indicates the corresponding component is fixed to the initial value. The `strain` key set the initial strain, accepting an array of 6 floating-point values in Voigt index (the default value is all zero). Among them, the values corresponding to the fixed components will be fixed in the simulation.

By default, the `free_components` and `strain` keys directly control the components of the components of η. It is possible to impose a rotation transformation R so that such two keys control another strain tensor $\tilde{\eta}$ which is transformed into η *after* applying the rotation transformation expressed by R. The transformation matrix R could be defined in several different ways, as controlled by the `transform_mode` key:

- `transform_mode = "Direct"`: The transformation matrix $R = \mathbb{1}$, i.e., no transformation. This is the default behavior.
- `transform_mode = "AlignX"`: The transformation matrix R will transform the first lattice vector a from a vector parallel to the first Cartesian coordinate axis x.
- `transform_mode = "AlignZ"`: The transformation matrix R will transform the first lattice vector c from a vector parallel to the first Cartesian coordinate axis z.
- `transform_mode = "Manual"`: The transformation matrix is manually supplied using the `transform_matrix` key, which accept a three-by-three matrix. The input matrix must be a orthogonal matrix.

> **Tip**
>
> The `AlignZ` modes could be useful for the misfit strain for the epitaxial film case if the misfit strain is isotropy in the in-plane direction, especially for the cases where the film is grown along the c axis which is not parallel to the z axis. But be aware, the algorithm only ensures the direction of the c axis, but not of the a, b axes. This is similar for the `AlignX` mode.

Example 1: simple expand epitaxial strain 2% for film growing along the z axis.

```
[physical.misfit]
enabled = true
free_components = [false, false, true, true, true, false]
strain = [0.02, 0.02, 0.0, 0.0, 0.0, 0.0]
transform_mode = "Direct"   # Optional
```

Example 2: compress epitaxial strain of 1% for film growing along the c axis (which is not parallel to the z axis)

```
[physical.misfit]
enabled = true
free_components = [false, false, true, true, true, false]
strain = [-0.01, -0.01, 0.0, 0.0, 0.0, 0.0]
transform_mode = "AlignZ"
```

Pressure and stress

The `pressure` key sets the hydrostatic pressure, which accepts a floating-point number in GPa. The pressure takes part in the simulation by adding the following term in the enthalpy

$$H_{\text{pressure}} = p(V - V_0) = p[(\eta_1 + 1)(\eta_2 + 1)(\eta_3 + 1) - 1]V_0,$$

where p is the pressure, V is the volume of the *supercell*, V_0 is the volume of the supercell with reference structure, and η is the strain. Such definition indicates that a possible pressure tends to *compress* the sample while a negative one tends to inflate the sample.

The `stress` key sets the anisotropy external stress, which accepts an array of six floating-point numbers, in Voigt index and in the unit of GPa. The stress σ_l takes part in the simulation by adding the following term in the enthalpy

$$H_{\text{stress}} = V_0 \sum_l \eta_l \sigma_l.$$

If the `pressure` and `stress` keys are set simultaneously, effects from both setting will be applied.

Electric field

The (homogeneous) electric field could be set by the `elec_field_homo` and `elec_field_scale` keys. The `elec_field_homo` accepts an array of three floating-point nubmers, expressing the electric field along the three Cartesian coordinate axes, in the unit of V/Angstrom. The `elec_field_scale` is a scalar factor multiplying on `elec_field_homo` (the default value is 1).

> **Note**
>
> The unit of electric field is V/Angstrom, which is different from our previous perovskite implementation (where V/m is used), and is a very large unit.

Since version 0.26, it is also possible to specify the "reduced" electric field, instead of the macroscopic electric field defined in usual way, by using the `red_elec_field_bar` field, with the unit `V`. See *Electrical conditions* for more information.

Displacement field

Available as of version 0.26 (Mar. 16, 2026).

The displacement field could be specified using the `disp_field_homo` and `disp_field_scale` keys, in a similar way to the electric field as described above. Also, the reduced displacement field is controlled using the `red_disp_field` key. See *Electrical conditions* for more information.

3.2.6 Simulation tasks

The simulation task defines way that the conditions are controlled for a series of simulation runs. There are several task types, each corresponds to a pre-defined workflow type, with possibly some parameters controlling the details of the task. The task must be defined by the `task` key in `[flow]` table for all simulations.

> **Note**
>
> For some task types, the physical conditions set in `[physical]` table may be overwritten.

The pre-defined task types are as follows.

Single simulation and repeating simulation

The simplest task, `task = "Single"`, corresponds to a single simulation with a fixed condition, defined by the `[physical]` table. The repeated simulation, `task = "Repeat"`, refers to a series of repeated simulation (the repetition times is defined by the `repetition` key) within the same condition. The default `repetition` value is 2. If `repetition` is set to 1, it is equivalent to `task = "Single"`.

Changing temperature

The `Tsweep` task changes the temperature within a monotonic (i.e., either decreasing or increasing) and linearly-spaced list of temperature values. The temperatures are defined by the starting value `temp_start`, the termination value `temp_end`, and the step size `temp_step`. For the simulation indexed by `i` (starting from 0), the temperature value is `temp_start + i * temp_step`. If the temperature value exceeds `temp_end`, the simulation is ended.

> **Note**
>
> The value of `physical.temperature` has no effect in these tasks.

> **Warning**
>
> The values of `temp_start`, `temp_step` and `temp_step` must define a *finite* series of temperature.

Example: The following setting will trigger a series of simulation with decreasing temperatures 400 K, 390 K, …, 20 K, 10 K, in total 40 simulation runs.

```
# ...

[flow]
task = "Tsweep"
temp_start = 400
temp_end = 10
temp_step = -10

# ...
```

The `Tcycle` task is a `Tsweep` task followed by another series of temperature that varies from `temp_end` to `temp_start` with the step `- temp_step` (i.e., reverse of the former series). Specifically, this could be a heating process followed by a cooling process, or a cooling process followed by a heating process. For example, for the example above, if the `task` is now changed to `Tcycle`, the temperatures will be 400 K, 390 K, …, 20 K, 10 K, 20 K, …, 390 K, 400 K.

Changing electric field

For current implementation, all tasks that changes the electric field during the simulation is done in a *colinear* way, i.e., only a *scalar* scale factor of the electric field is changed, with its direction unchanged. The homogeneous electric field must be set through the `physical.elec_field_homo` key (see *Electric field*) or the `physical.red_elec_field_bar` key (since version 0.26, see *Electrical conditions*).

Important

Remember to set the `physical.elec_field_homo` (or `physical.red_elec_field_bar`) value to a non-zero vector!

Note

The `physical.elec_field_scale` value has no effect in these tasks.

The `task = "Esweep"` set the electric field scale to a series of monotonic (either increasing or decreasing) values, each is fixed for a single simulation run. Such series is controlled by the starting scale `Eamp_start` (default: 0.0), the terminal scale `Eamp_end` (default: 1.0) and the number of steps `Eamp_steps` (default: 11). Note that the number set by `Eamp_steps` *includes* both the first and last simulation run.

Example: the following configuration will trigger a series of simulation with electric field scale 0.0, 0.1, ..., 1.0; or equivalently, the electric field values of 0.01, 0.02, ..., 0.09, 0.1 V/Angstrom, along the z axis of the Cartesian coordinate system.

```
# ...
[physical]
elec_field_homo = [0.0, 0.0, 0.1]   # V/Angstrom

[flow]
task = "Esweep"
Eamp_start = 0.0   # The same as default
Eamp_end = 1.0     # The same as default
Eamp_steps = 11    # The same as default

# ...
```

The `task = "Ecycle"` set the electric field scale to a series of monotonic values followed by another series of monotonic values but with reversed order. For example, in the example above, if the `task` is set to `Ecycle`, the electric field scale will be 0.0, 0.1, ..., 0.9, 1.0, 0.9, ..., 0.1, 0.0.

The `task = "EsweepInc"` (respectively, `task = "EcycleInc"`) is similar to `task = "Esweep"` (respectively, `task = "Ecycle"`), but with the electric field changes progressively during the first part of each simulation runs, instead of suddenly at the first step of each simulation run (as `Eweeep` and `Ecycle` does). The details of the electric field changing behavior could be controlled by the `[physical.inc_elec_field]` table. With `task` set to `EsweepInc` or `EcycleInc`, the `physical.inc_elec_field.enabled` is automatically set to `true`, and the `inc_amp` key is automatically computed. Usually, only the `steps_rate` could be modified, which determines the number of steps that the electric field scale is changing. For example, with default setting `steps_rate = 0.4` and in total 10000 steps in each simulation run, the electric field scale will change linearly during the first 4000 steps. Comparing to `Esweep` and `Ecycle`, this smooth changing of electric field is more close to the thermodynamics reversible condition and may be useful for some specific simulation tasks, for example, evaluating the adiabatic temperature change when applying the electric field.

Important

Usually, the `physical.inc_elec_field.steps_rate` should be set to a value *no larger than* that of `simulation.equiv_rate`. Otherwise, the statistic average values are collected with changed electric field, which may lead to error results.

Example: In the following example, the electric field *scale* will be 0 for the first simulation run, and increases linearly to 0.1 during the first 4000 steps of the second run, and keeps 0.1 during the next 6000 steps; then increases to 0.2 during the first 4000 steps of the third run, etc.

```
# ...
[physical]
elec_field_homo = [0.0, 0.0, 0.1]

[simulation]
sim_steps = 10000

[physical.inc_elec_field]
steps_rate = 0.4   # Same as the default value

[flow]
alg = "MD"
task = "EsweepInc"
Eamp_start = 0.0
Eamp_end = 1.0
Eamp_steps = 11

# ...
```

The `task = "Eloop"` set the electric field scale to a series of values that including both positive and negative values, consistent with common dielectric loop diagram. The electric field scale starts from 0, then change to `Eamp_end` with linearly spaced steps (in total `Eamp_steps` simulation runs, including both ends), and then change reversely to `-Eamp_end` with same step size, finally goes back to 0 with same step size. For example, for the default configuration `Eamp_end = 1.0` and `Eamp_steps = 11`, the electric field scales would be 0.0, 0.1, ..., 0.9, 1.0, 0.9, ..., 0.1, 0.0, -0.1, ..., -0.9, -1.0, -0.9, ..., -0.1, 0.0.

Note

The `Eamp_start` is not used in `Eloop`; the starting point is always 0.

Changing displacement field

Available as of version 0.26.

The work flows for changing the displacement field (`disp_field_homo` or `red_disp_field`) are designed in the same manner as those for the electric field (*Changing electric field*), with the `E` in the names of the keys and tasks changed into `D` (e.g., `Esweep` is changed to `Dsweep`, `Eamp_steps` is changed to `Damp_steps`), and the controlled scale `elec_field_scale` changed into `disp_field_scale`.

3.2.7 Initial structure

By default, the initial structure of the simulation is the reference structure (see also *Structure specification*), with all DoFs zero. If the model contains mixed sites, the atoms of the mixed sites are distributed randomly according to the concentrations set in the unit cell metadata file (see also *Select reference structure*). These behaviors could be changed through the `init` table.

The atomic distributions of mixed sites could be specified explicitly using the site text input file. To do this, use the `init.read_sites_txt` key to specify the input file name prefix (see *Input reference*).

The initial DoF values could be loaded through either text input files, or HDF5 input files. To do this, use the `init.read_modes_txt`, `init.read_strain_txt` and `init.load_dof_h5` keys. See *Input reference* for more details.

Note

The DoF HDF5 file contains both local mode and strain data. Thus, the DoFs could be loaded by setting `load_dof_h5` *alone*, without `read_strain_txt` and `read_modes_txt`. If the latter `txt` files are specified, they take precedence over the HDF5 file for the corresponding part.

Important

The DoF HDF5 file **does not** contain atom distribution file. The most common setting for continuing a previous simulation run is to use `read_sites_txt` along with `load_dof_h5`.

3.2.8 Energy model types

In this program, the "energy models" are responsible for computing the potential energy and its derivatives (i.e., forces and stresses). In most of the simulation tasks, computing such quantities takes most of the computational time, highlighting the vital importance of the *computational efficiency* of the energy models. In current implementation, the energy model consists of two parts: the dipole model, which is responsible for computing the potential energy, force and stress directly related to the dipole moment (including the long-range dipole interaction); and the local model, for all other contributions. We provide several types of models for both part. Each of these model types should yield exactly the same result, but they may have different performance (efficiency) for different tasks.

For the dipole model, there are two types, which could be specified by the `dipole_model` key in `flow` table: `Real` and `Reci`. Typically we recommend the `Reci` model (which is also the default), where the potential energy and force of the dipole moment are computed in the reciprocal space using the fast Fourier transformation (FFT) trick, as it is expected to yield better performance for most of the cases.

For the local model, there are several options for the `local_model` key: `Proto`, `Real`, `DP`, `Reci`, and `ConditionalReci`. The `Proto` model is only provided for development aims and is not expected to provide good efficiency in any conditions. The `Real` model computes the potential energy and gradient directly from the metadata, with minimal status data stored. This is the default (currently not able to change) for the on-the-fly MD (used in parameter fitting). The `DP` model makes use of the direct product (DP) of local modes and converts the mode interaction to matrix-vector multiplication tasks. The `Reci` model uses the same DP strategy, with a further FFT performed to (possibly) accelerate the computation with available supercells (see *Structure specification*). For the supercells not supporting FFT, the model produces error message and exit. The latter behavior is different for the `ConditionalReci` model, which falls back automatically to `Real` model if FFT is not available.

According our simple efficiency test for some models, the `Reci` model performs the best for most of the simulation tasks, among all the local models. However, the `Real` model generally yields better parallel efficiency. It is strongly recommended to do efficiency test between different models before doing production runs.

Tip

The dipole model and local model types are independent from each other. The choice of dipole model does not (theoretically) affect the local model efficiency and vice versa.

Note

By default, the local model type `Proto` and dipole model type `Real` are not available as they rarely yield better performance than other models. If you do wish to use these models, pass the argument `-DHEFF_ENABLE_PROTO_MODELS=ON` to CMake during building process.

3.2.9 Integrator types

For MD-based simulators (for example, MD and HMC), the time domain integrator may affect its performance. The default integrator is the well-known velocity Verlet integrator, which evaluates the force (and stress) only once in each MD step (with the time step length set in `simulation.md.timestep`, with default value 1 fs). The integrator type could also be modified by setting `flow.integrator` key.

An alternative integrator is the so-called Omelyan integrator, introduced by [Omelyan2002], which is a second-order two-stage integrator, evaluating the force twice in each MD step. This could be selected by setting `flow.integrator = "Omelyan"`. The ξ coefficient is set to the "optimized" value (Eq. 13 of the paper) $\xi = 0.193...$ by default, but could also be modified by setting `simulation.md.omelyan_xi`. Note that by definition of this package, (on average) one time step includes two force evaluations in for the Omelyan integrator.

3.2.10 Controlling the energy components

Sometimes it is useful to turn off some of the interactions (i.e., remove these terms from the potential energy expression). In addition to editing the parameter file to set the corresponding parameters to zero, it is also possible to realize this using the configuration file. This is mainly about the `energy` table in the configuration TOML file (`heff_config.toml`).

To disable an INT completely (for the concept of INT, SGRP and SAT, please refer to *Generate interactions*), the simplest method is to specify its index (see the interaction report LaTeX file) in the `disable_interactions` field, which accepts an array of integers. In addition, it is also possible to control the SGRPs (SAT groups) in each INT, using the `interact_controls` field, which accepts an array of tables. In each element of the `interact_controls`, the INT index must be specified. The maximal group index could be given by the `max_group` key. If a non-negative integer is given, the SGRP with index larger than the given value is disabled.

For example, using the following configuration, the INTs indexed 0 and 1 are disabled completely. The SGRPs with indexes larger than 2 in the INT indexed 4 are also disabled. The disabled terms never contribute to the energy, force or stress, regardless of the value of parameters provided for them.

```
[energy]
disable_interactions = [0, 1]

[[energy.interact_controls]]
index = 4
max_group = 2
```

In addition, the long-range dipolar interaction could be disabled by setting `disable_long_range = true`.

3.2.11 Controlling the DoFs

The DoFs in the effective Hamiltonian (the local modes and the strain) have three possible statuses:

- `Enabled`: The DoF could contribute to the potential energy, and is free in dynamic simulations.
- `Frozen`: The DoF could contribute to the potential energy, but its value is fixed to the initial value (may be zero by default, or input value from file).
- `Disabled`: The DoF is completely disabled and has no contribution to the potential energy, as if it does not exist in the effective Hamiltonian. All the interactions including this DoF is also disabled. Physically it is equivalent to the `Frozen` status with initial value being zero, however the computational complexity is reduced with the `Disabled` status.

By default, all the DoFs have the `Enabled` status. To control the status of strain DoF, use the `strain_status` key. To control the statuses of local modes, use the `lwf_controls.<lwf_name>.status` key, where `<lwf_name>` should be replaced by the name of the LWF, which defines a table in the TOML input file.

For example, in the following configuration, the strain DoF is set to `Enabled` status, the local mode named `u` is set to `Frozen` status, and the local mode named `v` is set to `Disabled` status. Note that these two different setting formats for `u` and `v` defines equivalent TOML file structure, see TOML homepage for more details.

```
[energy]
strain_status = "Enabled"
energy.lwf_controls.u.status = "Frozen"

[energy.lwf_controls.v]
status = "Disabled"
```

3.3 Parallel simulations

There are two different means of parallelism employed in this program, both are optional and requires corresponding support in the building stage:

- `OpenMP` parallelism, requires `-DENABLE_OPENMP=ON` CMake variable on build (this is the default), and
- `MPI` parallelism, requires `-DENABLE_MPI=ON` CMake variable on build (this is the default).

Generally, `OpenMP` parallelism is applied to share work loads to multiple cores and increasing the overall speed, like many other programs that also use OpenMP. On the other hand, the `MPI` parallelism in this program is only applicable to several special cases to run some tasks in parallel, which is different from some commonly-used softwares.

3.3.1 `OpenMP` parallelism

`OpenMP` supports parallelism on **shared-memory architectures** (usually, on the same node in HPC system). In this program, the `OpenMP` parallelism is applied in many functions to speed up the computation. Because of some applied `OpenMP` APIs in this program, we require the `OpenMP` library version 4.0 or later. Usually the `OpenMP` library is provided along with the C++ compiler. Thus, if the `OpenMP` library is too old, consider changing to another C++ compiler.

> **Warning**
>
> As mentioned, the `OpenMP` parallelism is only applicable on **shared-memory architectures**. DO NOT try to run programs on non-memory-shared architectures with only `OpenMP` parallelism (for example, running the program on multiple nodes in HPC system).

If the program is built successfully with `OpenMP` support, running the `heff_run_sim` executable will automatically invoke `OpenMP` parallelism. It is important to properly choose the number of threads for `OpenMP`. Generally, this is

achieved by the standard way: setting the `OMP_NUM_THREADS` environment variable before running the program. For example, on Linux:

```
OMP_NUM_THREADS=4 heff_run_sim
```

The command will execute the simulation with 4 parallel threads. The following output confirms the setting is actually applied:

```
[info] OpenMP is enabled with 4 threads
```

Important

The number of `OpenMP` threads could affect the parallel efficiency. Too many threads may not be beneficial, sometimes even causes longer execution time. It is strongly recommend to do efficiency test on the number of threads before running a long-time simulation.

3.3.2 `MPI` parallelism

Important

In this program, the `MPI` is NOT generally used for speeding up, but instead, only applicable to some specially designed parallel tasks, as introduced below. If you do not understand this, DO NOT USE MPI!!!

Currently, the `MPI` parallelism is applicable to the following two conditions:

- The parallel tempering Monte Carlo (PTMC) simulation, and
- Running multiple (serial) simulation tasks independently in parallel.

If `MPI` is enabled and multiple (more than one) processes are started, most of the output (that presented in the standard output or `stdout`) will be redirected to the files called `stdout_<mpi_rank>.txt`, where `<mpi_rank>` is the rank of the `MPI` process. Running the program with `MPI` support but using only one process is equivalent to running without the `MPI` support.

Parallel tempering

The parallel tempering Monte Carlo (PTMC) simulation is implemented according to Ref. [PTMC2005]. The temperature distribution is generated through some pre-defined algorithms (specified in the configuration file by `simulation.ptmc.temp_dist`), and could be adjusted after each iteration [PTMC1996]. The number of iterations performed is specified by `simulation.ptmc.num_iters`, which is default to 1, with no temperature adjustment applied. After each iteration, the temperature distribution is written in an YAML file with default filename `temp_dist_iter<iter_id>.yaml`, where `<iter_id>` is the iteration index starting from 1. For the simulations with `num_iters` set to n, the YAML files with ids $1, 2, \cdots, n+1$ will be generated. The file indexed $n+1$ represents the temperature distribution that designed to be used in the next iteration (but the iteration is actually not performed).

In each PTMC iteration, a *replica* of the simulated cell is generated for each selected temperature. The simulation runs for several *blocks* (controlled by `simulation.sim_steps`). Within each block, the simulation of each replica (i.e., on each temperature) is performed *independently* with the selected simulator (which is specified by `flow.alg`). The simulator for the blocks must be an `NPT` simulator and is transparent to temperature change (currently, just `alg = "CanonicalHMC"`). Within each block, the simulation runs for several steps (specified by `simulation.ptmc.block_steps`). At the end of each block (except for the last block), a trial to *exchange* neighboring replicas is performed. The neighbor replicas may be exchanged or not. At the end of each PTMC iteration, the thermodynamics variables are computed for each of the temperatures, and written to output files. By default, the thermodynamics variable output file is named `heff_thermo_iter<iter_index>.txt`. The final configuration of the lowest temperature

replica is output by default. And the configurations of other replicas could also be written by optionally specify some keys in `simulation.ptmc`.

Here is an entire example of the configuration file for running the PTMC simulation.

```
[cell]
supercell_diag = [8, 8, 8]

[simulation]
sim_steps = 400

[simulation.md]
ensemble = "NPT"
adjust_temp_begin = true

[simulation.ptmc]
temp_low = 10
temp_high = 500
num_replicas = 24
num_iters = 2
block_steps = 200
temp_dist = "Inverse"

[flow]
task = "ParallelTempering"
alg = "CanonicalHMC"
local_model = "Reci"
```

In the example above, the PTMC simulation is enabled by using `task = "ParallelTempering"` in `[flow]`. The Canonical HMC simulator is selected by `task = "ParallelTempering"`, for running the simulation in each block. The `simulation.ptmc` table controls the detail of the PTMC simulation. The lowest and highest temperature are specified by `temp_low` and `temp_high` keys (note that for PTMC, `flow.temp_start` and `flow.temp_end` has no effect). The number of replicas is selected 24 (`num_replicas = 24`), which means 24 different temperature values will be selected (including the lowest and highest one). Two iterations will be performed (`num_iters = 2`), and at the end of the first iteration, the temperature distribution will be adjusted according to the acceptance data in the first iteration. In each block, 200 HMC steps will be performed (`block_steps = 200`). This means, in total, $400 \times 200 = 80000$ HMC steps will be performed (see also `simulation.sim_steps`). The initial temperature distribution algorithm is selected to be `Inverse`, indicates that the replica temperatures will be generated such that the *inverse* of the temperature values ($1/T$) are distributed uniformly between the highest and lowest values.

Once the input files are prepared, the simulation could be executed by typing the command like

```
mpiexec -np <number_of_processes> heff_run_sim
```

where `<number_of_processes>` is the number of `MPI` processes to be generated. The number of processes must be a divisor of the number of replicas (24 in this case). In this example, the number of processes could be 1, 2, 4, ..., 24. The `OpenMP` parallelism could also be applied (if the program is built with `OpenMP` support).

Parallel simulation for independent tasks

Another usage of `MPI` parallelism is to run several simulations *independently* in parallel. These simulations could have different random seeds (which generates different atomic distribution for multi-component models), electric fields, or temperatures. To enable this and select the distribution rule of the MPI processes, set the `mpi_dist` key in the `[flow]` table with following values:

- `mpi_dist = "Seed"`: Each `MPI` process has a different random seed, ranging from 1 to the total number of `MPI`

processes. The value of `init.seed` has no effect in this case.

- `mpi_dist = "Eamp"`: Each `MPI` process has a different scale of electric field (`physical.elec_field_scale`), distributed uniformly from 0 to 1. The value of `physical.elec_field_scale` has no effect in this case.
- `mpi_dist = "Temp"`: Each `MPI` process has a different temperature. For the `MPI` rank indexes `i` (this index starts from 0), the temperature is `mpi_temp_start + i * mpi_temp_step`, where `mpi_temp_start` and `mpi_temp_step` are defined in the `flow` table. The value of `physical.temperature` has no effect in this case.

To run the simulation, use the command like

```
mpiexec -np <number_of_processes> heff_run_sim
```

The number of parallel processes (this is also the number of parallel tasks) is determined by the number of processes started by `mpiexec`.

3.4 Preparing input files

3.4.1 Specifying input files

As mentioned above, the paths or filenames of most of the input files could be specified by the configuration file. These paths could be given by either relative or absolute paths. If relative path is used, it will be resolved based on current directory for running this package.

> **Note**
>
> Changed since version 0.12 (Jul. 1, 2025): The file paths for the model specification files could now be resolved based on the path given by `file_path.model_path_base`, which is just current working directory by default. See corresponding part in *Input reference*.

The following keys controls the filenames (paths) of the mandatory input files:

- `cell.unitcell_yaml`: the unit cell file. Default: `heff_unitcell.yaml`.
- `energy.files.interactions_json`: the interaction data file. Default: `heff_interactions.json`.
- `energy.files.params_yaml`: the parameter input file. Default: `heff_params.yaml`.
- `energy.files.ewald_path`: Directory for loading Ewald data file (see *Ewald summation file*). Default: `.` (current directory).

In addition, several other keys control the optional input files, with default value to empty string, indicating the files are not loaded. For example, in `init` table, the `read_modes_txt`, `read_strain_txt` and `read_sites_txt` keys specify the files for loading the mode, strain and site occupation data from text files.

The filename (path) of the configuration file itself, could be controlled by the first argument to the `heff_run_sim` executable. For example, using the following command:

```
heff_run_sim config.toml
```

the program will load the `config.toml` file at current directory as the configuration file, instead of `heff_config.toml` (the default).

3.4.2 Ewald summation file

The Ewald summation file includes the data required for computing the long-range dipolar interactions, which is stored after first computation (since the computation of the data contained in this file is rather time-consuming for large systems). The files are named in the format `EW_DD<dimension>_<P11>_<P12>_<P13>_<P11>_<P22>_<P23>_<P31>_<P32>_<P33>_N<total_uc>.h5`, where `EW` stands for "Ewald", `DD` stands for dipole-dipole interaction, `<dimension>` marks the dimensionality of the system (for example, `3` stands for the bulk system, `2z` stands for the film with *non-periodic* axis along the z axis, and `1x` stands for the nanotube with *periodic* axis along the x axis), `<P11>` to `<P33>` are the components of the supercell matrix P, and `<total_uc>` is the number of unit cells in the supercell. All the simulations require the corresponding Ewald file, *unless* there is no modes with long-range dipolar interaction (`dipole="LongRange"`) or `energy.disable_long_range` is set to `true`.

> **Note**
>
> The Ewald summation file format has changed from `EW_DD<dimension>_<P11>_<P22>_<P33>_N<total_uc>.h5` as of Jun. 19, 2025. All newly generated files will be named using the new format (as introduced above). The file with old filename format will be read if it exists and required file with new filename format is not found.

The Ewald file could be acquired with the following two ways: (1) compute manually, and (2) compute implicitly at the beginning of the simulation.

(1) Compute manually. This is the standard way to obtain the required Ewald summation file. To do this, one use the `heff_cal_ewald` executable. Typically, the input set that used for the simulation is enough. Additionaly, the table `[ewald]` may be supplied into the config file to specify more details of the computation. The output path of the Ewald summation file could be specified by the `output_path` key (the default is `.`, meaning the current directory for running this command). If the required output file exists in that directory and could be loaded, the computation is skipped (unless `overwrite_existing` is set `ture`, for which the existing file will be overwritten).

(2) Compute implicitly at the beginning of the simulation. This is a convenient way for obtaining the required Ewald summation file, especially useful for small supercells. To do this, set `auto_compute_ewald` to `true` in the `[energy]` table (this is the default), and the program would compute and save the Ewald summation file automatically if the required one is not found in the path specified by `energy.files.ewald_path` (in this case, the `ewald.output_path` is ignored). If the required file is found but could not be loaded (for example, has different supercell matrix, or unitcell basis not compatible with current cell, etc), the program would exit with error message, unless `ewald.overwrite_existing` is set to `true` (for which case the existing file will be overwritten).

> **Tip**
>
> For large supercells, the Ewald summation costs much time, especially for bulk case. It is recommended to use more OpenMP threads (if OpenMP is available), typically much more than the nubmer of threads used for the simulations. In this case, we recommend to compute the Ewald summation file manually to speed up.

Restrictions for the low-dimensional cases

In this project, the dipole interactions of the low-dimensional systems are treated using the method described in [LowDimHeff]. As a precondition of such method, the special crystal axis (the non-periodic axis for the 2d case and the periodic axis for the 1d case) must be perpendicular to the surface spanned by the other two axes. For example, for the 2d case with c axis has no PBC, there must be $a \perp c$ and $b \perp c$.

Moreover, during the calculation, as the algorithm required, the special crystal axis must parallel to the corresponding Cartesian axis. For example, for the 1d case with a axis has PBC, the a crystal axis must coincidence with the x Cartesian axis. This could, however, be solved automatically by the Ewald summation program. More precisely, the program will find the coordinate transformation matrix that send the special crystal axis to the corresponding

Cartesian axis at the beginning of the computation, and transform the resulting Ewald summation matrix back to the original coordinate system at the end of the computation. It is important to note that another similar coordinate transformation is also performed, in order to diagonalize the optic dielectric tensor, as required by the method used to treat the dipole-dipole interaction [AnistroDip]. For 1d and 2d systems, these two coordinate transformation operations must be compatible with each other. In other words, in order to compute the Ewald summation file, we must find a single coordinate transformation matrix that send the special crystal axis to the corresponding Cartesian axis, and *simultaneously* diagonalize the optic dielectric tensor. This is, unfortunately, not always possible. The program would try to find such transformation, and exit with error message if it is not possible.

In summary, for 1d and 2d cases, the system must meet the following two requirements: (1) the special crystal axis must be perpendicular to the other two crystal axes; and (2) it is possible to find a unitary coordinate transformation that send the special crystal axis to the corresponding Cartesian axis and diagonalize the optic dielectric tensor at the same time. There are no such requirements for the 3d and 0d cases.

Symmetry-adapted Ewald summation

Available as of version 0.25.0 (Mar. 1, 2026).

The crystal symmetry is utilized to reduce the number of required Ewald summation matrix elements, significantly increasing the computational efficiency, especially for the systems with high symmetry (the reduction of computational load could be **more than 90%** in some cases). This feature could be enabled by setting `ewald.crystal_symmetry` to true. This is enabled by default. If problems encountered, one may try to set this key to false to fall back to normal algorithm which computes all Ewald summation matrix elements.

The symmetry property of the Ewald summation matrix is determined by the symmetry of the unit cell, as well as that of the supercell. Thus, a high-symmetry supercell matrix (the supercell matrix that preserves as many symmetry operations of the unit cell as possible) may lead to higher efficiency. Generally, the supercell matrix form with the highest possible symmetry is the diagonal matrix with equal values (for example, $8 \times 8 \times 8$ supercell).

Checkpoint file in Ewald summation

Available as of version 0.25.3 (Mar. 10, 2026).

During the Ewald summation calculation, the checkpoint file, with name starts with `EWckpt_DD`, will be created automatically (if enabled). The result of each point will be written in this file, in YAML format. At the start of every Ewald summation calculation task, the program will check whether such checkpoint file exists. If so, and the calculation setup is consistent with currently setting, the checkpoint will be loaded, and the calculated results will be used in the new calculation, without the need of calculating again. This is extremely useful when the Ewald summation task is interrupted unexpected (due to some unexpected external reason, e.g., shut down of the machine, etc).

To load the checkpoint file, the unit cell configuration and Ewald summation configuration related to the precision must be exactly consistent in current run and the previous run generating the checkpoint file. The program will check the Ewald summation configuration, the supercell matrix (this is distinguished using the filename directly), and some of the unit cell metadata. If the provided unit cell metadata is inconsistent with previous run, the behavior is undefined (the program may fail to find some inconsistency in the unit cell metadata). If the loaded checkpoint file is found to be inconsistent with current setting, the program will exit with error messages. In this case, delete the checkpoint file and rerun the program. This will be needed if the Ewald summation precision settings is changed.

The checkpoint file will always be written in, and loaded from, the current directory for running the program, no matter it coincidences with the `ewald_path` or not.

The checkpoint writing is enabled when the number of unit cells in the supercell is no less than `ewald.write_checkpoint_min_size` (default: 16). Setting this field to zero will disable checkpoint writing. The checkpoint loading is enabled when `ewald.load_checkpoint` is set to true.

Note

Currently, the checkpoint loading is not supported if `ewald.crystal_symmetry = false` and `ewald.use_component_loop = false`.

3.5 Explaining the results

The outputs of the simulations are typically text files (`*.txt`) and/or *hdf5 <https://www.hdfgroup.org/>* files (`*.h5`). For serial simulations (i.e., without `MPI` parallelism), some abstract information are also presented in standard output (`stdout`, typically printed on the screen). For MPI parallel simulations, most of these abstract information are redirected to the `stdout_<number>.txt` files. By default, only part of the outputs are enabled; the others could be turned on by setting some tags in the configuration file. For all the filenames specified in the configuration file (including input and output), the suffix (extension, like `.txt`) should NOT be given; those suffixes are automatically appended by the program.

The following types of output files may be generated:

- The thermodynamics variable file, containing the statistical results of the selected physical quantities (for example, potential energy, temperature, polarization, etc). Default filename is `heff_thermo.txt` for serial simulations, `heff_thermo_iter<iter_number>.txt` for PTMC simulations, or `heff_thermo<mpi_rank>.txt` for other MPI simulations. This file may be the most frequently used output file. The output filename could be controlled by the `simulation.thermo_file` key. The variables selected to be computed is specified by the `simulation.thermo_vars` key.
- (Since 0.14.3, Sep. 12, 2025) The thermodynamics data YAML file, containing the same results as the thermodynamics text file above, but in YAML format. The default filename is `heff_thermo_data.yaml` for serial simulations, and similar extensions for parallel cases. This file could be loaded directly using other languages with YAML support, or loaded with the `FlowThermoData` class in this package (possibly through Python binding). The file contains some comments, making it sometimes more readable than the text output file.
- The snapshot text file, containing the snapshots (instantaneous values, without thermodynamics average) of some selected physical quantities. The default filename is `snap.txt` for serial simulations, `snap_rep<replica_index>.txt` for PTMC simulations, or `snap<mpi_rank>.txt` for other MPI simulations. By default, the contribution (from each *interaction*) of the potential energy is also given. By default, the physical quantities are those used for computing the thermodynamics average values, but could also be given separately using `simulation.snap_vars` along with `simulation.independent_snap_vars = true`. The frequency of generating these snapshots are given by the `simulation.snap_freq` key (default value is 200, meaning the snapshots will be given for each 200 simulation steps).
- The background snapshot text file, containing snapshots (instantaneous values) of some selected physical quantities. This is different from the snapshot text file above in the sense that the production of the snapshots in this file does *not* come along with log information on `stdout`, and no contribution information of potential energy is given. These differences make the overhead of writing this type snapshot lighter. This is typically used for a "denser" snapshot than the above snapshot output, for observing some step- or time-dependent variables. This is especially useful for the cases where time-dependent electric fields are applied. To enable the snapshot of this type, set `simulation.bg_snap.enabled` to `true`, and specify variables in `simulation.bg_snap.vars`. Note that different from the above type of snapshot, there is *no* variables output by default (except the condition variables).
- The simulation status file, containing the instantaneous values of some selected physical quantities. The default filename is `status-steps.h5` for serial run, and `status-steps<mpi_rank>.h5` for MPI parallel simulation runs. By default this output file is not enabled. This could be enabled by setting a positive integer value to `simulation.write_status_freq`, which also determines the frequency (per simulation step) for writing the status. The variables in the status are the same as those in snapshot output, as defined by `simulation.snap_vars`.
- The degrees of freedom (DoF) file for each simulation run, containing all the DoFs of the simulated cell at the end of each simulation run. The default filename is `dof-runs.h5` for serial simulations and `dof-runs<mpi_rank>.`

h5 for MPI parallel simulations. The output filename (prefix) is controlled by the `simulation.dof_runs_file`, and an empty string would disable this output.

- The DoF file for the final status in HDF5 format, containing a full set of all DoFs for the final status of the simulated cell (after *all* the required simulation runs). For serial simulation, the default filename is `dof-last.h5`. For PTMC simulations, by default the file is only generated for the replica with the lowest temperature with name `dof-lowest-iter<iter_id>.h5`, and the files of same format for *all* replicas could be generated by specifying a non-empty filename prefix to `simulation.ptmc.dof_replica_file`. For other MPI simulations, the default filename is `dof-last<mpi_rank>.h5`.

> **Note**
>
> In current implementation, the file formats of `dof-runs.h5` and `dof-last.h5` are different. The former contains multiple frames (set of DoFs) in a single file, while the latter contains only one. The frames in the former format could be selected to generate a file that compatible with the latter format, using the `heff_dof_split` executable tool.

- The DoF file for the final status in plain text format, including local mode variables and strain variables, written in two separate files. The default names are `final-modes.txt` and `final-strain.txt` for serial simulations, and `final-modes<mpi_rank>.txt` and `final-strain<mpi_rank>.txt` for MPI parallel simulation runs. For PTMC simulations, these files are automatically generated for the replica with the lowest temperature, and could also be generated for all replicas by specifying `simulation.ptmc.modes_replica_file` and `simulation.ptmc.strain_replica_file`, as similar to the HDF5 DoF file above.These files are given in plain text format, and could be read directly by commonly-used text editors, also easily processed by other programs.
- The sites output file, containing the coordinates for all of the unit cells and the atomic occupation information for models with mixed sites, in plain text format. The default filename is `out_sites.txt` for serial runs, and `out_sites<mpi_rank>` for MPI parallel runs.The files of the same format could be used as the input to specify the atomic distribution by setting `init.read_sites_txt` or `init.read_sites_txt_mpi`.

3.5.1 Defining the physical quantities

This program supports computing several physical quantities (variables) during simulations. These physical quantities may be computed as thermodynamics statistical quantities (i.e., thermodynamics average applied), or output as snapshot quantities (i.e., output the *instantaneous* values directly). Each type of simulator has several default quantities for thermodynamics statistical quantities and snapshot quantities, and other quantities could also be computed/output by specifying them in the input configuration file.

The physical quantities are specified using a simple domain-specific embedded language (DSEL) (detailed below) in the configuration file. Technically, this DSEL is parsed using the Boost.Spirit library. The thermodynamics statistical quantities could be added by setting `simulation.thermo_vars` in the configuration file, using an *array of strings*, where each element of the array should be a physical quantities defined in the DSEL. The specified physical quantities would be computed and written in thermodynamics statistical result file (by default `heff_thermo.txt`). The default physical quantities (defined by each type of simulators) could be *disabled* by setting `simulation.disable_built_in_thermo_vars` to `true`. Similarly, the snapshot quantities could be added by setting `simulation.snap_vars`, also with an array of string. The default snapshot quantities could also be disabled by setting `disable_built_in_snap_vars` to `true`. Moreover, if `simulation.independent_snap_vars` is set to `true`, the snapshot quantities will be set to the quantities that are required for the thermodynamics statistical quantities, and the `simulation.snap_vars` settings will be ignored. The physical quantities for background snapshot (`bg_snap`) could be specified using `simulation.bg_snap.vars`, also with an array of strings. Note that there is no default quantities for the background snapshot output.

Besides the physical quantities as output of the program mentioned above, there are also some physical quantities serving as input variable to the program, which are those controlled by the simulated process. For example, the temperature in canonical HMC simulations. These variables are called condition quantities. Some condition quantities

are written by default by the program, while other quantities could also be written as required by the user, using the `simulation.extra_condition_vars` key.

It should be noted that there are some intersection between the thermodynamics statistical quantities and condition quantities. For example, the temperature, electric field, displacement field, etc. They have the same input keys (but present in different places), an always have different output keys, to make them easy to be distinguished in the output files.

The required physical quantities, both the thermodynamics and condition ones, must be valid from the physical point of view. The required conditional quantities must be the controlled variables in the simulation process, while the thermodynamics statistical quantities must be quantities computed from the simulation task, not directly from controlled input quantities. For example, in MD simulations, the temperature is always computed (even in `NPT` ensemble), and is a valid thermodynamics statistical quantity. However, for canonical HMC simulation, the temperature is purely an input controlled quantity, and is thus not a valid thermodynamics statistical quantity. Another example, if the electrical condition is set to `FixE`, the electric field (`ElecField`) is a valid condition quantity but not a valid thermodynamics statistical quantity, while other field quantities, like `RedElecFieldBar`, `DispField`, are valid statistical quantity. If, on the other hand, the electrical condition is set to `FixRedD`, then the reduced displacement field (`RedDispField`) is a valid condition quantity, while the electric field `ElecField` is now a valid thermodynamics statistical quantity.

> **Warning**
>
> If a required quantity (either thermodynamics statistical one or condition one) does not meet the physical requirement above, the output value may be meaningless.

Physical quantity types

Roughly, all the physical quantities supported by this program could be divided into two categories:

1. **Status quantities**, that could be collected directly during the simulation, and corresponding average values and standard deviations could be computed as the statistical result. These quantities could be used as both snapshot quantities and thermodynamics statistical quantities. For example, potential energy, and strain.
2. **Final statistical quantities**, that could only be computed *after* the simulation run is completed, relying on several status quantities collected during the simulation run. These quantities could only be used as thermodynamics statistical quantities, but not as snapshot quantities. Typically, the standard deviation for these statistical quantities are not available. For example, heat capacity, and dielectric tensor.

The DSEL for the physical quantities consists of the quantity name specifier and an optional set of parameters. Some physical quantities accepts parameters, which are given in square brackets `[]`; while other quantities do not accept parameters. The following example shows some valid physical quantity specifications:

```
[simulation]
thermo_vars = [ "Diel", "Polar", "ModeAverage[u]" ]
```

Here, the `Diel` and `Polar` are final statistical quantities, referring to dielectric tensor and polarization, respectively, which do not accept extra parameters in the specification; while `ModeAverage` refers to the average local mode movement (over all the unit cells and over time/simulation steps), which takes an extra parameter (here, `u`), specifying the name of the corresponding mode (LWF).

Some important status quantities are summarized in the table below. Here, the column "Input key" provides the key in the physical quantity specification that used in the configuration file; the column "Output key" provides the key (prefix) used in the header row of output thermodynamics text file (`heff_thermo.txt`); the column `Dimension` provides the dimension (number of coefficients) for the physical quantity. Besides the listed quantities, some quantities with suffix `Abs` to the input key, and suffix `A` to the output key, are also supported. These quantities refer to the absolute values (in a proper way) for the corresponding variables. For example, the physical quantity `PolarInsAbs` is the coefficient-wise absolute value of `PolarIns`. This absolute value strategy is provided due to the symmetry properties of some

quantities, which is especially useful (and is the default behavior) of the PTMC simulations, where replicas with some quantities having inverse (negative or positive) but symmetrically equivalent values may be exchanged.

Note

The absolute value processing for strain and related variables are only applied on the non-diagonal elements (indexed 4, 5, 6, or yz, zx, xy, respectively), but not on the diagonal elements (indexed 1, 2, 3, or xx, yy, zz, respectively), because the positive and negative values of diagonal elements in the strain tensor (corresponding to expansion or compression, respectively) are not equivalent.

For the physical quantities support deviation values, the deviation values may be written into the thermodynamics output file `heff_thermo.txt` if `simulation.write_thermo_deviations` is set to `true` (this is the default). The keys for the deviation values start by the character `~`, followed by the key of corresponding physical quantities. For example, `~potential` refers to the standard deviation values of `potential`.

In the table below, the mark `(C)` in the `dimension` column indicates the current quantity is defined under the Cartesian coordinate, the mark `(D)` indicates the quantity is defined under the direct coordinate.

Note

The multi-dimensional physical quantities are listed in the output with all of their components one by one, with the component marked after an underscore (_) in the format of `<output_key>_<component>`. For quantities defined under Cartesian coordinate, the three-dimensional ones are marked with `x y z`, the six-dimensional ones with Voigt notation are marked with `xx`, `yy`, `zz`, `yz`, `zx`, `xy`. For quantities defined under direct coordinate, the three-dimensional ones are marked with `1`, `2`, `3`, the six-dimensional ones with Voigt notation are marked with `11`, `22`, `33`, `23`, `31`, `12`. For quantities defined for each mode, the components are marked with one-based indexes `1`, `2`, ...

Note

For some extensive properties proportional to the number of unit cells in the supercell, we here provide two versions of physical quantities: the "total" one with suffix `Tot` in input key and `_t` in output key, and the unit cell averaged one (i.e., with the number of unit cells divided) with suffix `UC` in input key and `_u` in output key. Available since version 0.28.1.

Warning

The physical quantity values are only available if it is not the physically controlled parameter during the executed simulation process. For example, with reduced displacement field controlled (`electrical_condition = "FixRedD"`, see *Electrical conditions*), the corresponding quantity `RedDispField` is not available.

Note

Some variables have been changed since version 0.28. We do not list the differences between the old and new versions here.

Input key	Output key	Dimension	Unit	Parameters	Description
`Temperature`	`temperature`	1	K	none	[1]
`EnthalpyTot`	`enthalpy_t`	1	eV	none	Total enthalpy[2]
`EnthalpyUC`	`enthalpy_u`	1	eV	none	Enthalpy per unit cell
`PotentialTot`	`potential_t`	1	eV	none	Total potential[3]
`PotentialUC`	`potential_u`	1	eV	none	Potential per unit cell
`InternalPotentialTot`	`in_pot_t`	1	eV	none	[4]
`InternalPotentialUC`	`in_pot_u`	1	eV	none	
`KineticTot`	`kinetic_t`	1	eV	none	[5]
`KineticUC`	`kinetic_u`	1	eV	none	
`SupercellVolume`	`sc_volume`	1	Å^3	none	Unit cell volume[6]
`UnitcellVolume`	`uc_volume`	1	Å^3	none	Unit cell volume[7]
`Strain`	`strain`	6 (C)	1	none	
`DipoleMoment`	`dipole`	3 (C)	e Å	none	
`DipoleMomentAmp`	`dipole_amp`	1	e Å	none	[8]
`PolarIns`	`polarI`	3 (C)	e / Å^2	none	Instantaneous polarization
`PolarInsLocAmp`	`polarI_Lamp`	1	e / Å^2	none	[9]
`ElecField`	`E`	3 (C)	V/Å	none	Electric field
`DispField`	`D`	3 (C)	e/Å^2	none	Displacement field
`RedElecField`	`RedE`	3 (D)	V Å	none	Reduced electric field[10]
`RedElecFieldBar`	`RedEb`	3 (D)	V	none	Reduced electric field[11]
`RedDispField`	`RedD`	3 (D)	e	none	Reduced displacement field[12]
`RedPolar`	`RedP`	3 (D)	e	none	Reduced polarization[13]
`ModeAverage`	mode name	LWF dimension	Å	mode name	[14]
`QModifiedModeAverage`	[15]	LWF dimension	Å	[16]	[17]
`ModeAmp`	[18]	1	Å	mode name	[19]

Some import final statistical quantities are summarized in the table below. Here, some quantities are only available for

[1] Currently only available for MD simulations.
[2] Total enthalpy provided for each unit cell.
[3] Potential part of the total enthalpy (per unit cell).
[4] Potential energy excluding the external contributions (electric field, stress, etc) per unit cell.
[5] Kinetic energy. Not available for NPT simulators.
[6] Supercell volume with strain considered.
[7] Unit cell volume with strain considered.
[8] `DipoleMomentAmp`: Summation of amplitude (norm) of dipole moment for each dipole. This is not the amplitude (norm) of the `DipoleMoment`. Available as of May 12, 2026.
[9] `PolarInsLocAmp`: Average of the amplitude (norm) of polarization of the local dipoles. This is not the amplitude (norm) of the `PolarIns`. Available as of May 12, 2026.
[10] Reduced electric field $e_i = \Omega \vec{b_i} \cdot \vec{E}$
[11] Reduced electric field (bar) $\bar{e}_i = \vec{a}_i \cdot \vec{E}$
[12] Reduced displacement field $d_i = \Omega \vec{b_i} \cdot \vec{D}$
[13] Reduced polarization $p_i = \Omega \vec{b_i} \cdot \vec{P}$
[14] `ModeAverage`: Average movement of a specified type of local mode (specified by parameter at input) over the whole supercell.
[15] In format: `<mode_name>@<q_spec>`
[16] mode name; q-point
[17] `QModifiedModeAverage`: Average movement of a specified type of local mode (specified by parameter at input) modified by a given q-point in the reciprocal space Brillouin zone (specified by the second parameter at input).
[18] In format: `<mode_name>_amp`
[19] `ModeAmp`: Average *amplitude* of the selected mode. The amplitude of each mode is computed in each unit cell and then averaged over all unit cells.

some specified ensemble types and/or simulator types, as indicated in the `ensemble` column. The meanings of other columns are the same as that in the status quantity table. Similarly, there are also some quantities with `Abs` suffixes, which are computed from status variables with absolute value operation properly applied. For example, the absolute values of dipole moment are used for the computation of `DielAbs`.

Input Key	Output Key	Dimension	Unit	Parameters	Ensemble	Description
`HeatCapTot`	`C_tot`	1	eV/K	none	NPT[20]	Heat capacity[21]
`HeatCapPot`	`C_pot`	1	eV/K	none	NPT	Heat capacity[22]
`Diel`	`diel`	6 (C)	1	none	NPT	Dielectric susceptibility
`ECcoefPot`	`ECcoefp`	3 (C)	(10^{-6} K m) / V	none	NPT	Electrocaloric effect constant[23]
`Piezo`	`piezo`	18	pC/N	none	NPT	Piezoelectric coefficient[24]

Simulation condition quantities

The simulation condition quantities are similar to the physical quantities, but they should be the "input" to the simulation, instead of the "output" of the simulation (as the physical quantities, see above). The condition quantities are mainly the physical conditions controlled and fixed by the simulation process during the simulation. For example, the temperature and the pressure in the canonical HMC simulations. The values of these conditions are also written into the thermodynamics output file and the snap output file, before the physical quantities output.

The program writes some physical conditions by default, as determined by the task and algorithm of the simulation. Extra condition variables could be specified using the `simulation.extra_condition_vars` key by specifying their names. The valid condition variable names are listed in the following table. The output keys of the condition variables are always distinct from the calculated physical quantities to avoid ambiguity.

The multi-dimensional components are marked in the same format as the physical quantities (see above).

Warning

If the condition variables selected by the user (through `simulation.extra_condition_vars`) are not fixed during the simulation (for example, temperature for microcanonical MD simulations), the output values may be incorrect.

Note

Some condition variable keys have been changed since version 0.26 (Mar. 16, 2026).

[20] Only available with MD simulation using NPT ensemble for now.
[21] `HeatCapTot`: heat capacity computed using the fluctuation of the total energy (enthalpy).
[22] `HeatCapPot`: heat capacity computed using the fluctuation of the potential energy part.
[23] Computed only with the (non-trivial) potential energy part.
[24] Piezoelectric coefficient with indices `ij`, where i and j are the indexes for polarization (from 1 to 3) and strain (from 1 to 6), respectively.

Input Key	Output Key	Dimension	Unit	Description
`SimID`	`sim_id`	1	dimensionless	Simulation ID[25]
`Seed`	`seed`	1	dimensionless	Pseudo random number seed[26]
`Temperature`	`T`	1	K	
`ElecField`	`Efield`	3 (C)	V/Å	Electric field
`DispField`	`Dfield`	3 (C)	e/Å^2	Displacement field
`RedElecFieldBar`	`RedEfieldB`	3 (D)	V	Reduced electric field[27]
`RedDispField`	`RedDfield`	3 (D)	e	Reduced displacement field[28]
`Pressure`	`P`	1	GPa	Hydrostatic pressure
`Stress`	`stress`	6 (C)	GPa	Anisotropic stress
`Beta`	`beta`	1	dimensionless	β value[29]

[25] Simulation index of current simulation run. Indexed from 1.
[26] Initial pseudo random number generation seed used for the simulation.
[27] Reduced electric field defined as $\bar{e}_i = \vec{a}_i \cdot \vec{E}$.
[28] Reduced displacement field defined as $d_i = \Omega \vec{b}_i \cdot \vec{E}$.
[29] The β value used for electrostatic screen for low-dimensional systems [LowDimHeff].

CHAPTER

FOUR

OPTIMIZATION AND SINGLE-POINT CALCULATION

In *Running Simulations*, we have introduced how to run simulations with an available Heff model. Here in this tutorial, we will introduce how to run structural optimizations and single-point calculations with an Heff model, which is rather similar to running simulations. Different from the simulations where thermodynamics distribution of the DoFs according to statistical physics is desired, the optimization procedures simply find the energy (local) minima of the system (the supercell or some generalized system, like the elastic band in NEB method, see later) from initial state. Such optimization procedures are similar to those usually performed with first-principles softwares, however with the potential energy and its gradient (force and stress) provided by the Heff model here. The single-point calculation, as indicated by its name, just compute the energy, force, stress and other additional properties (if required) of a single provided structure.

Note

The implementations of all the techniques introduced in this tutorial are highly experimental, please use with special care.

The required input files are the same as those for the simulation runs (*Running Simulations*), but with some different parts required in the configuration file `heff_config.toml`, detailed below.

4.1 Optimization procedures

To run the optimizations, the `optimization` table is required in the configuration file. The optimization task must be specified by the `task` key using the following options:

- `task = "Relax"`: Structural relaxation. Find the local energy minima from the given structure. See *Structural relaxation*.
- `task = "FlowRelax"`: Available as of version 0.28.0 (Mar. 26, 2026). Equivalent to running a series of `Relax` tasks with a workflow that changing the external conditions.
- `task = "NEB"`: Nudged elastic band (NEB). Find the minimal energy path (MEP) for some transformation process. See *Nudged elastic band (NEB)*.

There are several optional algorithms available for these optimization tasks, which must be selected using the `alg` key. The options includes:

- `alg = "SteepestDescent"`: Steepest descent algorithm. See [NEB_opt_2008] (Eq. 5).
- `alg = "AdamGD"`: Adam gradient descent algorithm. See [Adam2017].
- `alg = "FIRE2"`: Fast inertial relaxation engine (FIRE) version 2.0. Available as of version 0.24.0 (Feb. 5, 2026). See [FIRE2_2020]; see also [FIRE2006].
- `alg = "QuickMin"`: Quick-Min algorithm. Introduced in [NEB_opt_2008].

- `alg = "CG"`: Conjugate-gradient (CG) algorithm. The form introduced in [NEB_opt_2008] is used.

There are also other options available in the `optimization` table, as well as the sub-tables for the tasks and the algorithms (see below). The supercell information must be specified using the `cell` table, and the initial structure is typically specified by the `init` table, similar to the simulation runs.

Note

Note on version 0.24.0: a number of changes have been applied for these algorithms, including some of input parameters and implementation details. Besides, the `FIRE` algorithm is now deprecated, we suggest to use the `FIRE2` algorithm instead.

After all the input files prepared, the optimization could be executed by calling

```
heff_run_opt
```

Note that MPI parallelism is supported for the NEB task, but not supported for structural relaxation task. For NEB tasks with MPI support, the program may be executed with a command like

```
mpirun -np 4 heff_run_opt
```

During the execution, the snapshot of the system will be produced in some steps on the standard output (for MPI runs, this is redirected to the `stdout_<mpi_rank>.txt` file), for example:

```
[2026-01-28 16:22:29.132] [info] STEP      0, Hpot=-0.049345, force_max=0.321072, strain_
↪max=0.211469
[2026-01-28 16:22:29.225] [info] STEP     20, Hpot=-0.064746, force_max=0.010917, strain_
↪max=0.008991
```

The optimization procedure is finished when the maximal magnitude of force (on the modes) is less than the value configured by `optimization.force_tolerance` (unit: eV/Angstrom), *and* the magnitude of stress per unit cell is less than the value configured by `optimization.stress_tolerance` (unit: eV, note that here the stress is defined as the derivative of the potential energy over the strain). The following information will be produced to the standard output:

```
[2026-01-28 16:22:29.236] [info] converged at step 22  max force=0.005528 eV/Ang;  max␣
↪stress=0.007583 eV/uc
[2026-01-28 16:22:29.237] [info] Final potential energy (total) : -4.143909709129037;␣
↪internal part: -0.3795509294436621
[2026-01-28 16:22:29.237] [info] Final potential energy (per uc): -0.0647485892051412;␣
↪internal part: -0.00593048327255722
```

The snapshot file (`snap.txt` or `snap<mpi_rank>.txt`) and optimization log file (`opt_log.txt` or `opt_log<mpi_rank>.txt`) will also be produced, providing more information. The physical status (polarization, etc) is also available during the calculation, by specifying the `status_vars` key. See *Defining the physical quantities* for more information.

4.1.1 Structural relaxation

Warning

MPI is *not* supported in structural relaxation tasks. DO NOT attempt to run this task with MPI parallelism!

To run the structural relaxation task, the initial structure is typically provided with the `init` table (see *Initial structure*, see also *Input reference*). If no related configuration provided, the reference structure is used as initial structure, which is typically undesired.

Important

Always remember to specify and check the initial structure!

After the optimization finished, the final structure is written to (by default) the `dof-optimized.h5` file in HDF5 format, and `final-modes.txt`, `final-strain.txt` in text format, as well as the `final-dipoles.yaml` for the local dipole information in YAML format.

4.1.2 Structural relaxation workflow

Note

This part is available only as of version 0.28.0 (Mar. 26, 2026).

If task is set to `FlowRelax`, a series of structural optimization is executed, as controlled by the top-level `flow` table that used in the simulations (*Running Simulations*). This process could be understood as running a simulation workflow at zero temperature.

It should be noted that not all tasks in the `flow` table are available. Specifically:

1. As temperature is meaningless for optimization tasks, all temperature-changing tasks like `Tsweep` and `Tcycle` are not available.
2. As the time step or simulation step is meaningless in optimization tasks, all increasing-field tasks such as `EsweepInc`, `EcycleInc` are not available.

During the workflow, the structure is initialized only once at the beginning, as specified by the `cell` and `init` tables. The structural relaxation is performed at the first step. Then, at each step, the structural optimization starts from the final (optimized) structure in the last step. The summary of each optimized structure, including the selected physical quantities, are written into the optimization result file (by default, `heff_opt_result.txt`) and related data file (by default, `heff_opt_result_data.yaml`). The optimized structure of each step is written into (by default) the `dof-runs.h5` file.

4.1.3 Nudged elastic band (NEB)

The NEB method (along with some its variants) is widely used for finding the minimal energy path (MEP) between the initial and final states, with several images provided. In this package, the climbing image NEB (CI-NEB) and solid state NEB (SS-NEB, but with a slightly different format) are supported. See [CI-NEB-2000] and [G-SSNEB-2012] and references therein for theoretical aspects.

Technically, the NEB method optimizes a so-called *elastic band* made up by several *images* of the studied system, with interactions between each other. Each image is a *replica* of the studied system (i.e., a supercell with some distortions). In the Heff context, all of these replicas must be different from each other, but must with the same occupations for the mixed sites. The elastic band is optimized using the selected algorithm until the force (and stress, if SS-NEB is enabled) for each NEB image (consists of part of the physical force/stress on the replica, and some forces/stress arisen by the elastic band) meets the convergency requirement.

Warning

Please ensure the same atomic occupations for each image if there are mixed sites in the model. Otherwise the behavior is undefined. The program does not guarantee to detect this inconsistency.

The NEB calculation starts from initial structure *for each image (replica)*. Typically, the `init.load_dof_h5_rep` key is used for loading the different configurations for different replica. The number of images must be specified using `optimization.neb.num_images` (denoted by n here). The program will load $n + 2$ images from the input files, where the index 0 is the initial structure, the index $n + 1$ is the final structure, and the indexes from 1 to n are the intermediate structures. The initial and final structures are not changed during the optimization, while the intermediate structures are optimized.

For example, with the following configuration:

```
[init]
load_dof_h5_rep = "dof-interp"

[optimization]
alg = "FIRE"
task = "NEB"
[optimization.neb]
num_images = 5
```

The file `dof-interp0.h5` will be the initial image, the file `dof-interp6.h5` will be the final image, and the files from `dof-interp1.h5` to `dof-interp5.h5` will be the input structure for the intermediate structures.

The input structures may be obtained from linear interpolation of the initial and final images of the transition process, or any other approaches. For the former, we provide a utility program called `heff_dof_linear_interp`, which could be called using command like

```
heff_dof_linear_interp -i dof-init -f dof-final -o dof-interp -n 5
```

with this command, the files `dof-init.h5` and `dof-final.h5` are required as inputs, for the initial and final structures of the transition process, and 5 images will be inserted between them with equal spacing; the output files will be `dof-interp0.h5` to `dof-interp6.h5`.

The NEB task could be executed either with or without MPI parallelism. If MPI parallelism is enabled, the number of MPI processes must be a *divisor* of the number of (movable) NEB images. For example, if `optimization.neb.num_images = 12`, the number of MPI processes may be 1, 2, 3, 4, 6, 12, but not 5.

After the optimization, the minimal energy path (MEP) is described by the output file `mep.txt`, including the potential and tangent force information as well as other required variables. The file may be visualized using the command like

```
heffplt_mep_interp.py mep.txt
```

Where the potential energy is shown as a function of reaction coordinate [root mean square (RMS) distance from the initial structure], with interpolation curve obtained from Hermite spine interpolation utilizing the information of tangent force.

Warning

The implementation of NEB is highly experimental and not extensively tested. Use with special care!

Tip

We find the NEB calculation is rather challenge, especially for relatively large cells. The convergency may be difficult to be obtained even for cells with just several tens of unit cells. If the NEB calculation does not converge, try to use a smaller cell, change the optimization algorithm, adjust the parameters for the optimization algorithms, or use a looser convergency requirement.

4.1.4 Optimization algorithms

There are different optimization algorithms implemented in this package. These algorithms may be suitable for different cases, possibly with several adjustable parameters. If the selected optimization algorithm suffers from convergence or stability issues, it is suggested to try other algorithms.

Here are some suggestions from our preliminary tests, which may be suitable for some cases, but also possible to fail in other cases. For structural relaxation tasks, which is typically easier to be converged, we suggest to use the algorithms like `CG`, `FIRE2`, and `AdamGD`, which can lead to fast convergence. For NEB tasks, the convergence could be considerably challenging, and stability issues may arise for some algorithms like `FIRE2` and `CG`. Nevertheless, we still suggest to try `FIRE2` first because it may be faster. If it encounters some stability issue, try to use some algorithms that are less aggressive, like `QuickMin` and `SteepestDescent`.

Here we introduce some algorithms with their adjustable parameters.

`SteepestDescent`

The most direct gradient descent method. In each step, the modes are moved along the negative gradient direction with fixed step size, see Eq. 5 in [NEB_opt_2008].

In this package, the step sizes are controlled independently for local modes and strain, with the keys `mode_step_size` (in the unit of Angstrom) and `strain_step_size` (dimensionless) in `[optimization]` table.

Note

The step size for strain is used directly for the strain variables. As the energy of strain (i.e., elastic energy) is scaled linearly with respect to the supercell size, the step size for strain may need to be *decreased* for large supercells to avoid instability.

`QuickMin`

The Quick-Min algorithm is a damped dynamics routine with fixed time step. Here, the time step is a dimensionless variable that links the local mode variables and their gradient (negative force), velocity directly, without any extra unit conversion. The strain variables are scaled by the Jacobian factor as used in NEB (see [G-SSNEB-2012]). The algorithm is described in [NEB_opt_2008].

The adjustable variables in the `[optimization]` table are:

- The timestep Δt: `quick_min.timestep`.
- The maximal allowed movement of a single step `max_move`.

`FIRE2`

The `FIRE2` option switches the FIRE (fast inertial relaxation engine) V2.0 which is published in [FIRE2_2020]. The similar damped dynamics routine is applied as in `QuickMin`, with the scaling of the Jacobian factor used. The maximal allowed movement `max_move` is also applied. Besides, other adjustable parameters are included in the `optimization.fire` table (for the symbols used here, please refer to the cited article)

- `fire.timestep`: The initial time step Δt_{start}
- `fire.time_max`: The *scale* of the maximal timestep Δt_{max} with respect to `fire.timestep`.

- `fire.time_min`: The *scale* of the minimal timestep Δt_{min} with respect to `fire.timestep`.
- `fire.n_min`: The parameter $N_{N>0}$
- `fire.time_inc`: The parameter f_{inc}.
- `fire.time_dec`: The parameter f_{dec}.
- `fire.alpha_start`: The parameter α_{start}
- `fire.alpha_dec`: The parameter f_α
- `fire.alpha_min`: The parameter α_{min}

CG

The conjugate gradient (CG) algorithm takes the form introduced in [NEB_opt_2008]. The maximal movement every step is controlled by `max_move` as in `QuickMin`. The adjustable parameter is the `cg.trial_step` which controls the step size for the trial step for computing the step size in line search.

AdamGD

The Adam algorithm takes the form of [Adam2017]. The adjustable variables for Adam algorithm are listed in the `adam` table:

- `adam.stepsize`: step size α.
- `adam.decay_b1`: parameter β_1.
- `adam.decay_b2`: parameter β_2.
- `adam.epsilon`: parameter ϵ.

4.2 Single-point calculation

Available as of version 0.23.3, Feb. 1, 2026.

The so-called single-point calculation is an evaluation to the potential energy and its gradient (force and stress; optional) for a given structure, which is a single point in the phase space. The single-point calculation uses the structure from input, exactly the same as the initial structure for simulation and optimization runs (see *Initial structure*). The required input files are the same as the optimization and simulation runs. The input configuration file (default: `heff_config.toml`) is basically the same as the simulation input file, but with an optional extra table `single_point`, controlling several details. A snapshot which is similar to the simulation snapshot is conducted. The physical quantities for the snapshot could be specified using the `status_vars` key, using the syntax introduced in *Defining the physical quantities*.

With input file prepared, the single-point calculation could be executed by

```
heff_single_point
```

The potential energy information is printed to stdout. The gradient of mode and strain (by default) are written to files, with default names `grad-modes.txt` and `grad-strain.txt`, using the same formats as the text files for DoFs.

MPI is supported by this calculation, for processing different input structures in parallel. If MPI is enabled with more than one process, different input structures may be specified by keys like `init.load_dof_h5_rep` (see *Input reference* for more information). Output for different structures are given in files with MPI rank suffixes.

CHAPTER

FIVE

BUILDING A MODEL

This document introduces how to build an effective Hamiltonian model for a specified material, including finding the basis, the interaction forms, and fitting the parameters.

Generally, the task of building an effective Hamiltonian (Heff) model for a specified compound consists of the following steps:

1. Select the reference structure.
2. Build the LWF basis.
3. Prepare unit cell metadata.
4. Generate the interactions.
5. Fit the parameters.

In this document, we will introduce the building process following these steps. For simplicity and clarity, we here only introduce the most basic techniques and operations in this tutorial. More advanced techniques and operations will be introduced in *Advanced techniques in model fitting*.

5.1 Select reference structure

By the definition of effective Hamiltonian [Zhong1995] [ALFE-H-2025], all the structural distortion (including local modes and strain) are defined with respect to the *reference structure*. The reference structure is a single crystal structure defined by the *unit cell* basis and the *sites* inside the unit cell. Typically, the reference structure should have relatively high symmetry (according to the studied distorted structures), making the effective Hamiltonian form simpler with fewer parameters to be fitted. Meanwhile, the reference structure should also be chosen so that the studied distorted structures have relatively *small* distortions, as the effective Hamiltonian is built using Taylor expansion, and smaller distortion means fewer terms and parameters needed for the parametrization. For example, for perovskites like $BaTiO_3$, the cubic structure with $Fm\bar{3}m$ space group is typically chosen as the reference structure.

For multi-component models, the reference structure is somewhat ambiguous. Conceptually, the sites with multiple components (**mixed site** henceforth) should be treated as a *single* "average" site, which behave like sites of the same type under symmetry operations. Such mixed site is just similar to the virtual atom in the virtual crystal approximation (VCA) [VCA2000]. For example, in $PbZr_{0.5}Ti_{0.5}O_3$ (PZT), the Ti and Zr sites are *conceptually* treated as a virtual element, denoted as $\langle B \rangle$. Thus, the reference structure is treated as $Pb\langle B \rangle O_3$, which has exactly the same symmetry properties as any simple perovskites, like $PbTiO_3$.

In this program, however, such virtual mixed atom only exists conceptually. The actual first-principles (FP) calculations do not use VCA. The properties of the reference structure should be computed using the supercells with some specified atomic arrangement (typically a somewhat "ordered" arrangement).

5.2 Build the LWF basis

In our formalism, the effective Hamiltonian is the Taylor expansion of the potential energy surface with respect to the strain and *local modes*. The local modes are localized collective movement of some atoms, in a specified form. The basis of such movements is sometimes called **lattice Wannier function (LWF)** [LWF1995] [LWF2000] in this program and this document.

> **Note**
>
> Here the concept of LWF is somewhat generalized. Conventionally, LWF typically denotes the real-space localized basis for the *phonons*, which are obtained from the normal modes of some selected phonon branches in the Brillouin zone. However, in this program, the LWF could be obtained from, either the phonon dispersions, or atomic distortions of some considered low-energy structures, or even other sources. We emphasize that, as least in its mathematical form, the local mode basis, or the LWF, is *just a linear combination of atomic movements*. Specifically, the *local mode* that extensively used in previous perovskite works [Zhong1995], could also be considered as a special type of LWF (where, only the unstable dipolar mode at Γ point is considered in the building process).

Each LWF has specified symmetry properties, as specified by their corresponding *band representations* [LWF2000] [band_rep_2018] [band_rep_1988], labelled by its *center site* and a irreducible representation (irrep) label of the site group. For a center site with Wyckoff multiplicity d_w and a d_r-dimension irreducible representation, the labelled LWF has the dimension of $d = d_w d_r$. The local mode with d-dimension basis LWF has d *order parameters* in each unit cell (to which the center sites of the LWF belongs). For supercell with N unit cells, such local mode has Nd order parameters. Such order parameters are used as the arguments for the Taylor expansion of the potential energy surface. For example, for the $BaTiO_3$ model [Zhong1995], the local dipolar mode basis is centered at Ti site, which has the site group $m\bar{3}m$, and the LWF is labelled by the T1u irrep, which has the dimension $d_r = 3$. The Wyckoff multiplicity of the Ti site is $d_w = 1$, thus the LWF dimension is 3.

> **Note**
>
> In this program, we use *symmetry-adapted* LWF. Any LWF generated from methods that do not consider the symmetry, for example, [LWF_opt_2005], could not be used directly.

In general, the LWFs should be linearly independent from each other (but not necessarily orthogonal to each other). The total dimension of selected LWFs in each unit cell should be equal to or less than the total atomic degrees of freedom (DoFs) in each unit cell (this is obvious, because the local modes are just linear combinations of atomic displacements). For a unit cell with n atoms, the number of atomic DoFs is $3n$, and the summation of LWF dimensions in the unit cell is at most $3n$. The LWFs should be chosen and fitted such that the important distorted *structures* (about the considered physical problem) could be well fitted by the basis. This requirement should be considered in, and reached by, both the selection of LWFs (the center and symmetry of the LWFs) and the fitting of LWF (selection of the free parameters in LWFs, see below). For example, for perovskite structures, the local dipolar mode u and acoustic mode v should be included (see [Zhong1995]).

After selecting the LWF centers and irreps, the next task is to build the LWF, or equivalently, fit the free parameters in the LWF. Building the LWF basis is generally a complex task. In this program, although several automatic tools are provided, the LWF basis should still be built with great care. Currently, we provide two different methods for building the LWF: (1) the reciprocal-space method, where the LWF is first built in the reciprocal space from phonon normal modes and then Fourier transformed to the real space; and (2) the real-space method, where the LWF parameters in the real space are directly fitted to phonon or atomic displacements. The two methods are introduced in the following sections.

Warning

The LWF building process requires the precision of the structure (in the sense of symmetry) to te sufficiently high. This means the numerical error of the structure yield by the symmetry operations must be sufficiently small. This may be checked by using the `--tolerance` argument of `phonopy --symmetry` command, for example `phonopy --symmetry --tolerance 1e-7`. The numerical errors from the structure may *strongly affect* the LWF building process and may cause serious problems. We strongly recommend to refine the structure according to desired crystal symmetry before building LWF.

5.2.1 Reciprocal-space method

In the reciprocal space method, the LWF is first built at each k-point in the reciprocal space as the linear combination of the normal modes, and then transformed to the real space through Fourier transformation. We mainly adopt the method described in [LWF2000], but with some modifications.

The reciprocal-space build supports two different building procedures: the first build one LWF at each time, and the second build multiple LWFs at a single run. The main difference is that the latter ensures that there is no intersection between the band subspaces of different LWFs. In this section, we first introduce single LWF build procedure, and then multiple LWF build procedure.

The input for each LWF includes the atomic structure of the reference structure, the phonon calculation from Phonopy, the configuration file [including the selected center of the LWF, the irrep labels for the LWFs band representation (of the site group of the selected center), and other configurations].

Important

For multi-component models, in current implementation, if the reciprocal-space method is used, the atomic structure used for the phonon calculation and LWF generation must have only a single element at the mixed site. For example, for the PZT model, either $PbTiO_3$ or $PbZrO_3$ should be used in phonon calculation and LWF generation. Otherwise the symmetry of the reference structure is broken.

The phonon calculation data should contain the unit cell structure file, the phonopy configuration file for plotting band structure (by default `band.conf`), and the force constant or force set file (`FORCE_CONSTANTS` or `FORCE_SET`) generated by Phonopy. The files should be given such that running the command like

```
phonopy -p band.conf
```

could get the proper phonon band structure. For more information, please refer to Phonopy document. By default, the unit cell structure file should be given in vasp format with a file named `POSCAR`.

Important

The unit cell structure file should provide the *primitive cell* structure. If the given structure is not a primitive cell (i.e., is a supercell of the primitive cell that contains more atoms), the result is undefined.

Single LWF build

Firstly, we would like to introduce the single LWF build procedure, which builds only one LWF at each time, because we find this procedure is relatively simpler, and the multiple LWF build procedure shares some common steps with this procedure.

In this package, the reciprocal-space generation of LWF includes the following steps:

1. For each k-point in the Brillouin zone, select a set of *irreps* that is consistent with the LWF band representation. This is accomplished using the algorithm described in [band_rep_2018].
2. (Optional) For each k-point in the Brillouin zone, choose a set of phonon *bands* to form the *band subspace* for the LWF. The selected bands must meet the symmetry requirement as specified by the irreps in the last step.
3. (Optional) Optimize the band subspace by minimizing the spillage functional as previously used in symmetry-adapted maximal localized Wannier function (SA-MLWF) for electrons [SA-MLWF-2013]. Note that if this step is applied, the resulting band subspace is formed, generally, not by a set of normal modes, but by the *linear combination* of a set of normal modes. The linear combination may include non-degenerated normal modes. Thus, the resulting basis vectors of the optimized band subspace are in general not the eigenvectors of the dynamical matrices.
4. Perform the gauge transformation [LWF2000] in the selected band subspace in order to make the resulting LWF more localized.
5. Perform Fourier transformation to get the real-space LWF.
6. Get the *shelled-LWF* by extracting the atomic displacements in each of the "shells" around the center site. Each shell may include multiple atoms of the same type, sharing the same distance from the center site. The shelled-LWF form is directly used in later parametrization of the effective Hamiltonian.

Note

Both step 2 and 3 are optional, but at least one of them must be performed. To build LWFs for Heff, we recommend to use step 2.

The LWF generation task is fully controlled by the configuration file, which is provided in TOML format. The default filename is `lwf_gen.toml`. Here is a simple example of the file.

```
[control]
phonon_dir = '.'
unitcell_filename = 'POSCAR'
band_conf_file = 'band.conf'
plot_regen_lwf_band = true

[lwf_config]
q_mesh = [2, 2, 2]
center_atom_id = 1
wyckoff_irrep_label = "T1u"

anchor_qpoint = [0, 0, 0]
start_degenerate_set_ids = [0]

output_prefix = "lwf_u"

optimize_subspace_basis = false

branch_select_strategy = "spillage_iter"

classic_transform_method = "coherent_svd"

auto_resort_axes = true
print_level = 2
```

(continues on next page)

(continued from previous page)

```
[lwf_config.spillage_config]
random_retries = 20
max_iters = 500
```

The configuration file contains two root TOML tables, `[control]` and `[lwf_config]`. In the `[control]` table, the `phonon_dir`, `unitcell_filename`, and `band_conf_file` keys defines the required data for Phonopy calculation. The presented values are default values, for which the phonon data will be loaded from current working directory. Other keys are also available for configuring the building process.

The details of the LWF building process is controlled by the `[lwf_config]` table.

The q-point mesh in the reciprocal space should be given by the `q_mesh` key. The phonon branches and normal modes are computed for each of the q-points in the mesh. The real-space LWF will be given in the supercell that consistent with this q-mesh, as a result of Fourier transformation. Thus, the q-point mesh could determine not only the accuracy of the calculation, but also the maximal possible spread of the resulting LWF.

The center site of the LWF should be specified by the `center_atom_id` key with the center atom index (zero-based index in the unit cell structure file). The irrep label of the LWF band representation should be given with the `wyckoff_irrep_label` (using an irrep label of the site group of the center site). To find possible irrep labels using this program, see *Find possible Irreps*.

The `output_prefix` key sets the prefix for the output files. Two important output files will be generated: `<output_prefix>_data.json` containing the LWF data (atomic movement directions and amplitudes), and `<output_prefix>_symmetry.yaml`, containing the symmetry information of the LWF.

Other important keys are introduced in the following sections, following the building stages introduced above.

Find band subspace discretely

This section introduces the keys controlling the step 2 as introduced above, where the band subspace is build *discretely*. Here, we use the word "discretely" to indicate that the band subspace built from this step is spanned by several selected modes, but not their arbitrary linear combinations, at each q-point, which has only finite number of possible results. As introduced in the literatures (see, e.g., [LWF2000]), the band subspace is a subspace of the atomic displacement space, with the dimension equal to the dimension of the LWF that is on building. This step is by default performed, but could be skipped if `skip_discrete_search` is set to `true`.

The band subspace selection is done by selecting *degenerate set* at each q-point. A degenerate set is formed by one or more normal modes at a given q-point, sharing same frequency, and transform according to an irrep at the little group of the q-point. The degenerate sets for each q-point are indexed from zero in ascending order of the frequency (the degenerate set indexed 0 corresponding to the lowest frequency, etc).

Important

In this package, we mainly use the normal modes and degenerated sets calculated from phonopy package. However, we find that the phonopy (at least up to version 2.38.0) determines the degenerate sets simply by comparing the frequency values. The "Irrep" found for these degenerate sets (using the *IRREP* tag in phonopy configuration file) are sometimes actually *reducible*. In these cases, the degenerate set could be further split into several degenerate sets, and each of them corresponds to an irrep of the little group. This decomposition is done by our package (when necessary). If this happens, the supplied degenerate set index should be the index of the decomposed one, not the original one by phonopy. In this case, the output would contain lines like this:

```
INFO:root:Reducible mode representation found at qpt=array([0., 0., 0.]) ideg=0;␣
↪decomposed to 2 actual irreps
INFO:root:Decomposition of degenerate set performed for qpt=[0. 0. 0.], now␣
↪degenerated sets: [[0], [1, 2], [3], [4, 5], [6], [7], [8, 9], [10, 11]]
```

There are several choices of algorithms for selecting the band subspace, each with different standard, and may produce different results. The algorithm is selected by the `branch_select_strategy` key, with following choices: `min_energy`, `max_projection`, `spillage_iter`, and `load`. Meanwhile, several optional keys also control the selecting process.

The `branch_select_strategy = "min_energy"` algorithm simply selects the degenerate sets with lowest possible energy (frequency) at each q-point, provided that the symmetry requirement is satisfied.

The `branch_select_strategy = "max_projection"` algorithm selects the branches with maximal projection on a selected "anchor" mode, which means the modes (at each q-point) most "similar" to the anchor mode are selected at each q-point, provided that the symmetry requirement is satisfied. The anchor mode is defined by the `anchor_qpoint` key which should be supplied with an array of 3 elements (by default, `[0, 0, 0]`, i.e., the Gamma point) and the `start_degenerate_set_ids` key which accepts an array of integer indexes (zero-based), defining the degenerate set at the selected anchor point. In the example above, we have `start_degenerate_set_ids = [0]`, which means the first degenerate set (i.e., the degenerate set with lowest energy or frequency) at Gamma point is used as the anchor mode. For each of the q-points in the mesh, the eigenvectors of all degenerated sets are projected onto the anchor mode, and the maximal possible combination is selected.

The `branch_select_strategy = "spillage_iter"` algorithm selects the degenerate sets at each q-point, such that the "spillage functional" (see [MLWF-2001], Eq. 8) is minimized. This iteration is performed using a greedy strategy. The algorithm try to find best combination of branches that maximizes the overlapping between neighboring q-points, and is considered to be converged if no updates of selected branches take place for all the considered q-points. Some detailed configurations are provided in the `[spillage_config]` table. We numerically find that the results of this procedure dependent drastically on the initial choice of branches. To increase the possibility of finding better choices, the procedure may be repeated for several times (configured by `spillage_config.random_retires`), each with a different random initial choice of branches. The program would automatically choose the result with maximal spillage functional. If `spillage_config.force_unstable` is set to `true`, the program will enforce the unstable modes (with negative frequencies) to be chosen for the q-points where possible (i.e., the unstable modes could expand the required subspace). If `spillage_config.fix_anchor` is set to `true` (this is the default), the pre-selected modes (by `start_degenerate_set_ids`) will be fixed during the iterations. In this case, the selected degenerated set ids for the `anchor_qpoint` must meet the symmetry requirements.

The `branch_select_strategy = "load"` does not try to find a new set of degenerate sets, but instead just load it from provided file. The files are typically generated in the multiple-LWF build run, or subspace splitting run (see *Multiple LWF build*). The filename is specified by `load_subspace_file` key (default: `lwf_subspaces.json`), and the index of current LWF must be specified by `load_subspace_lwf_index` key (zero-based index). In this case, the loaded subspace file must be generated with consistent configurations with current run. Specifically, the q-point mesh must be consistent, and the loaded eigenvectors and frequencies must be consistent with currently built dynamical matrices (by phonopy package).

Optimize subspace continuously

The *continuous* subspace optimization could be turned on by setting `optimize_subspace_basis = true`. Unlike the previous step, where the band subspace is spanned by the normal modes, in the optimization, the band subspace is spanned by the *linear combination* of the normal modes with the constrain of symmetry. As the normal modes take part in the linear combinations are not necessarily from the same degenerate set, these basis vectors of the band subspace are generally *not* eigenvectors of the dynamical matrix. Such optimization is conducted using the algorithm provided in [SA-MLWF-2013].

In practice, the detailed configurations are provided in the `[spillage_config]` table. The `max_iters` key sets the maximal number of iterations for the subspace optimization. The `freq_window` key sets the "window" of the considered frequency by providing minimal and maximal frequency values (in the unit of THz). If given, only the modes with frequencies inside this window are considered for the subspace basis. Note that in this case, the initial modes of the optimization must belong to the frequency window.

If the previously introduced discrete search of subspace is performed, the branches selected by such step serve as initial choice for the subspace optimization here. If the such discrete search is skipped (`skip_discrete_search = true`),

the initial choice will be built according to the `branch_select_strategy` selection (see previous section), but without symmetry constraint. Note that in this case, `branch_select_strategy = "spillage_iter"` is not supported.

Gauge transformation

Theoretically, the gauge transformation step finds a non-singular transformation matrix M at each q-point, such that

$$\tilde{u}_j(q) = \sum_i M_{ji}(q) u_i(q),$$

where u is the basis vector of the band subspace at the q-point, non-singular matrix M is the gauge transformation matrix. There are several method provided for the transformation, but we strongly recommend the `classic_transform_method = "coherent_svd"` option. Such option uses the so-called "coherent" condition that introduced in [LWF2000], in which a set of orthonormal vectors (we call them "coherent axes") $\{x_t\}$ are generated according to the band representation, and the M matrix makes the transformed basis vector $\tilde{u}_s$ orthogonal to x_t if $s \neq t$, and "parallel" to it if $s = t$. The difference from the original implementation is that a Lowdin transformation to the matrix M is followed to make the matrix M orthonormal [MLWF-2001]. The Fourier transformation is then applied to the $\tilde{u}_j(q)$ basis to get the real-space LWF.

Fourier transformation

Note

This part is done automatically by the program, typically do not need to be controlled by the user. We write this part only to introduce the underlying formalism.

After the gauge transformation, the real-space LWF is obtained by running Fourier transformation

$$u_k(R_l) = \sum_q u_k(q) \exp(iq \cdot R_l),$$

where R_l is the real-space position of the unit cell, k has $d = d_w d_r$ different indexes. We will refer to u_k (or w_k later) as LWF "axis" from then on.

Note that by construction, the LWF obtained here is in "reduced coordinate", where the value of the LWF is

$$u_k(R_l, j, a) = w_k(R_l, j, a)\sqrt{m_j},$$

where w_k is the atomic movement of atoms (in length unit), m_j is the mass of the atom indexed by j, and a is Cartesian coordinate index. The program then convert the LWF into the "direct coordinate" (w_k), and normalize the coefficients in w_k.

Build shelled-LWF

Till now, the LWF has been built in a *periodic supercell*. The output of the previous section is the (normalized) atomic displacement of each atom in a periodic supercell, with a center site for this LWF. We emphasize there is periodicity in the *supercell*, making the atomic position relative to the center site somewhat ambiguous. For example, suppose we have a $2 \times 2 \times 2$ supercell, the center site of the LWF is `[0, 0, 0]` (expressed in fractional coordinate of the unit cell). Then, for a atom located at `[1, 1, 1]`, its position related to the center could be `[1, 1, 1]`, or `[-1, -1, -1]`, or even `[3, 3, 3]`, according to the periodicity of the supercell.

To avoid such ambiguity, in the later parameter fitting of the effective Hamiltonian, we use another form of LWF, called *shelled-LWF*, where the atomic displacement are expressed by their position (unit cell position and site inside the unit cell) about the center site, and the atomic displacement. We call it "shelled" because the atoms around the center site could form several "shells", each with some atoms (of the same type) with the same atomic displacement *amplitude*

(but possibly different directions). Such shelled structure is a consequence of the symmetry. In principle, the shelled LWF could have infinity spread. In practice (and obviously), it consists only finite shells, spreading only a finite range, with respect to the center site.

The program automatically converts the LWF in periodic supercell to shelled-LWF if `control.build_shell_lwf` is set to `true` (this is the default). During the conversion, for each atom, the program simply choose the shortest vectors with respect to the center site among all the possible vectors (arisen from the periodicity of supercell). If there are multiple choices (i.e., we have more than one vectors that have the same shortest length for a given atom in the supercell), the atomic displacement is divided equally to each of the choices. In other words, the atomic displacement of this atom in the supercell is considered to be equally contributed by multiple atoms in the same shell with respect to the center site, and all of these atoms in the shell is linked by the supercell translation vector. Note that such division is only performed if `control.shell_allow_pbc_split` is set to `true` (this is the default); otherwise, such atoms (with multiple choices of vectors) in the supercell would be discarded, and there atomic displacement will *not* be included in the shelled-LWF.

After the shelled-LWF is built, the program checks whether the desired "transformation property" is satisfied by the LWF. This is done by *numerically* checking that

$$\hat{R}w_k(R_l, j, a) = \sum_j w_j(R_l, j, a)[\Gamma(\hat{R})]_{jk},$$

for any of the space group symmetry operation $\hat{R}$. Here, $\Gamma(\hat{R})$ is the band representation matrix of operation $\hat{R}$ at Γ point. Note that, technically, the left side of the above equation, i.e., the transformed LWF, may be translated so that the center atom (of each LWF axis) locates in the unit cell with $R_l = 0$.

Such transformation check is conducted if `control.check_lwf_transform` is set to `true` (the default). If the transformation check fails, the program would exit with error message. Such failure may be caused by internal error of this program or numeric errors. For the latter case, first ensure the input structure is precise enough (in the sense of symmetry). Try to refine the structure according to the desired crystal symmetry. If this does not solve the problem, try to set larger value for `lwf_config.symprec`. This will set a looser numerical requirement (i.e., allow larger numerical errors) for such transformation check.

Run single LWF building

After the configuration file and other source data are prepared, type a command like

```
python3 -m heff_gen.lwf.cui.gen_lwf_phonon lwf_gen_u.toml
```

to execute the LWF building process.

Here, we use the example of $BaTiO_3$ to demonstrate the building process. The input files could be found in the `lwf/phonon/BaTiO3` directory of the [Examples]. The input file `lwf_gen_u.toml` is the same as the example TOML file shown above.

The program will automatically show the figure below (if the running platform supports GUI) for the phonon dispersion. Here, the red curves are phonon dispersion from DFT, calculated by Phonopy; and the blue curves are phonon dispersions from the LWF, known as Wannier interpolation (see [MLWF-2001]). The negative frequency mode (known as "soft mode") is well-fitted by the built LWF.

Note

In the LWF building process, only some q-points in the mesh are considered (i.e., their eigenvectors are explicitly used while building LWF), while the dispersion at other q-points are from Wannier interpolation. Meanwhile, the DFT phonon dispersion calculation itself also considers only some q-points in another mesh (compliant to the supercell for computing the force constant matrix; this mesh may differ from that used for LWF building), while

the dispersion at other q-points are also from interpolation. The differences of the dispersion at those q-points are most likely from different interpolation schemes.

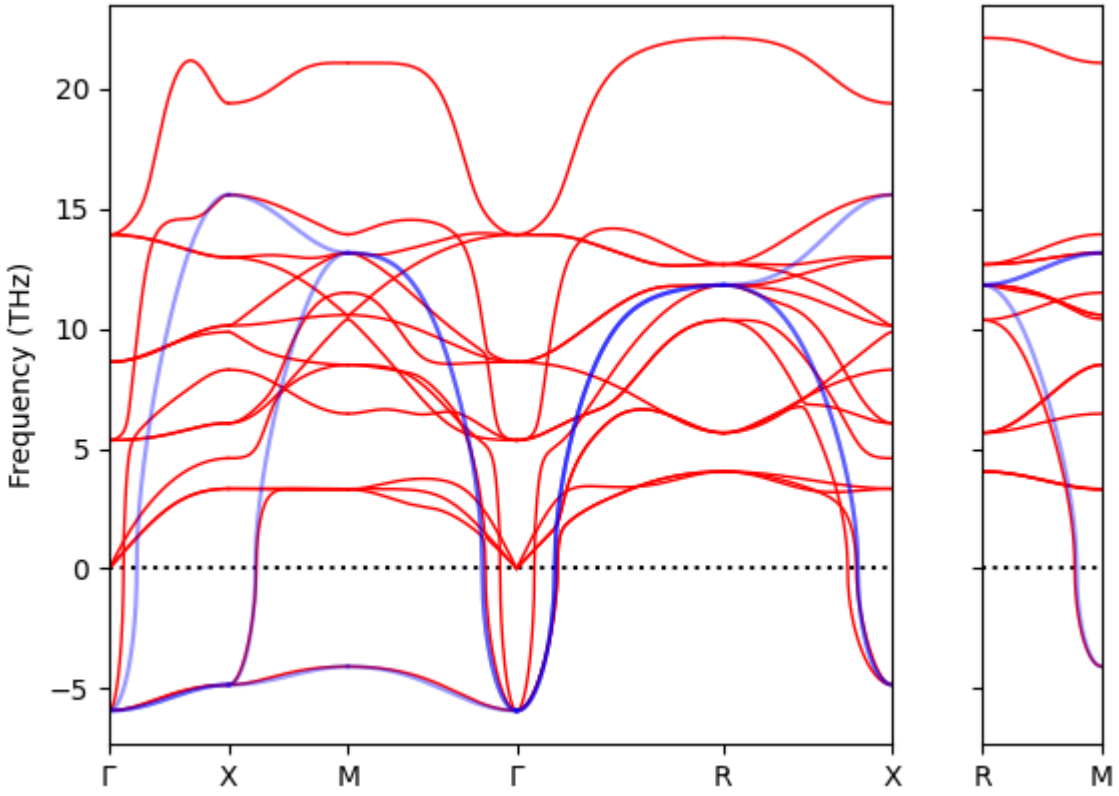


Fig. 1: Phonon dispersion from DFT calculation (red) and LWF (blue) of $BaTiO_3$ from the program

The program continues after closing the plotting window above. The next present figure is the amplitudes of atomic movement in the built shelled-LWF as a function of distance from the center site (see the following figure). Here, each bar shows the atomic displacement amplitude of a different set of atoms in the shell (about the center site). The position at horizontal axis shows its distance from the center site. The circle linked by dotted line shows the accumulated amplitude at each geometry shell (with same distance from the center site). The amplitudes are normalized, so that the accumulated amplitude is exactly one for the furthest shell considered in this LWF. The desired "localization" property of the LWF could be directly demonstrated by this figure. An LWF is well-localized if its atomic displacement mainly concentrate near the center site, and the amplitudes of atoms far from the center site are close to zero. Here in this example, the atomic displacement are mainly contributed by the first three shells (up to distance 1.992 Angstrom to the center), indicating this LWF is well-localized.

The program then shows the "regenerated" phonon dispersion figure. If `control.shell_allow_pbc_split` is set to `true` (the default), this dispersion should be exactly the same as the dispersion shown above. Here, the "regenerated" means the phonon dispersion is obtained from the data regenerated from shelled-LWF.

The main output of this program are the `lwf_u_data.json` and `lwf_u_symmetry.yaml` files. These files could be used in later effective Hamiltonian building process. The displacement of the LWF could also be visualized (in periodic supercell form) using the `lwf_u_ax0.xsf` to `lwf_u_ax2.xsf` files, each XSF file contains the atomic displacements of one LWF axis, and could be visualized by softwares like XCrySDen and VESTA. The XSF files are generated if `control.dump_xsf` is set to `true` (the default).

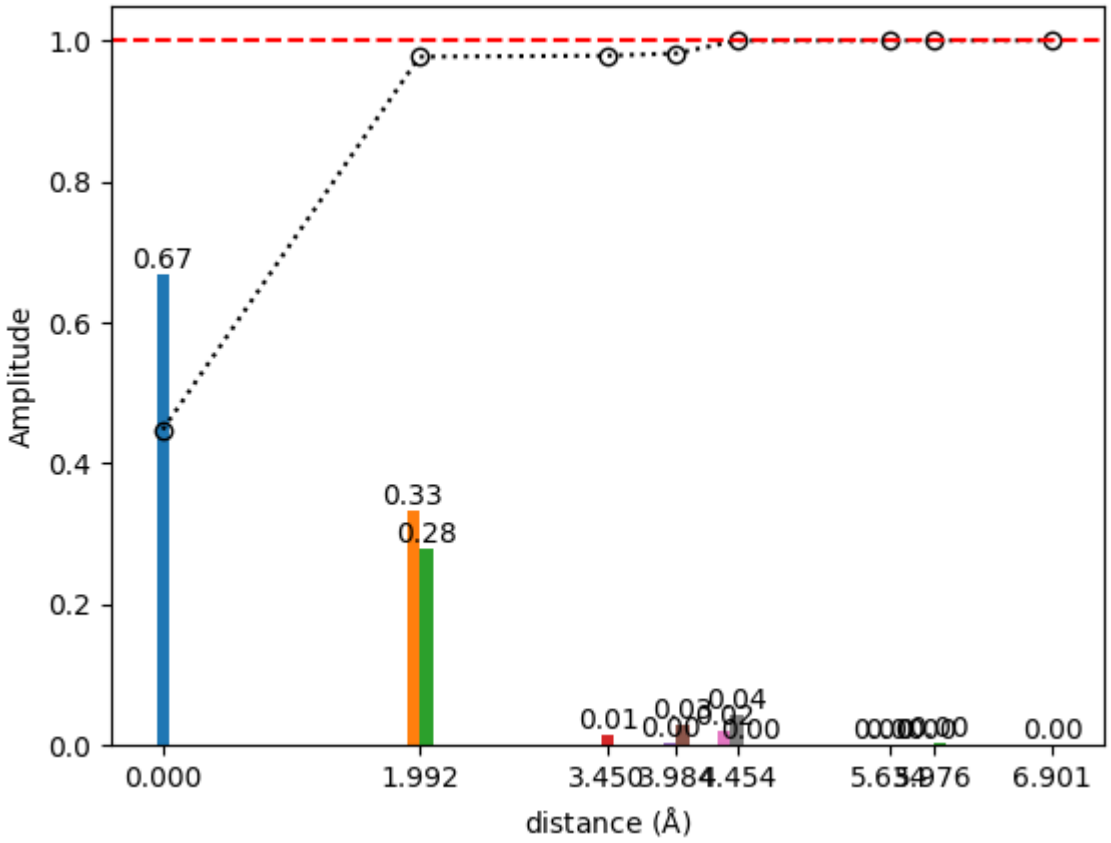


Fig. 2: Amplitudes of atomic movement in the built shelled-LWF

Find possible Irreps

As introduced above, an Irrep label should be assigned to each LWF. This Irrep label denotes an irreducible representation of the point group of the center site, which is a subgroup of the crystal space group.

It should be noted that not all Irreps are valid for a phonon band representation (see [phonon_band_rep_1995]). Practically in this package, such invalid choice of center site and Irrep would produce subduced band representation at some q-points (in the first Brillouin zone) that could *not* be reduced to Irreps of the normal modes. In this package, the provided combination of center site and Irrep label must be valid, otherwise the program will generate error message and exit in the subspace finding stage.

To find possible Irrep labels for a chosen center site of a given reference structure, we provide a simple method. With the `lwf_gen.toml` configuration file provided (including the `center_site` field), use the command like

```
python3 -m heff_gen.lwf.cui.gen_lwf_phonon lwf_gen.toml -s
```

to generate all possible Irrep labels for the chosen center site. The `irrep_label` field in the configuration file is ignored for this usage. This program simply enumerates all possible Irreps for the given center (generated using the `spgrep` package), and check whether they could be reduced to the Irreps of the normal modes at each q-point. The output may includes:

```
site symmetry group: m-3m  irrep_labels dict_keys(['A1g', 'A2g', 'Eg', 'T1g', 'T2g', 'A1u
↪', 'A2u', 'Eu', 'T1u', 'T2u'])
possible_labels: ['T1u']
```

Here, the center site has site symmetry group `m-3m`, with possible Irreps `A1g`, `A2g`, ... Among them, only the `T1u` Irrep is valid.

In general, the center site and Irrep for LWFs should be chosen according to the physical problems to be studied. The chosen LWF basis set should properly expand the atomic displacements of the desired structures (see *Check selected local mode basis*). The provided LWF phonon dispersion and atomic displacement XSF files may also provide useful information on choosing the LWFs.

Multiple LWF build

In realistic effective Hamiltonian building tasks, multiple LWFs may be needed. For example, in the $BaTiO_3$ model built in 1995 [Zhong1995], the basis set includes the local dipolar mode (or soft mode) u and the inhomogeneous strain mode or acoustic mode v. To build multiple LWFs for a unit cell, using the single LWF building process (see above) for multiple times is a straightforward solution. However, for the basis of the effective Hamiltonian, we typically require the subspaces of different LWFs to be exclusive from each other. This means if one normal mode at a q-point is selected by the subspace of one LWF, it should never enter the subspace of another LWF. This, however, is generally not guaranteed by running the single LWF building process independently for multiple times.

To solve this problem, a combined LWF building process is provided. In this process, multiple LWFs could be built, and the subspaces for each of the LWFs are guaranteed to have no intersection.

Similar to single LWF building process, the input set contains unit cell structure file, Phonopy calculation data, and a TOML configuration file. The TOML configuration file, however, has a different structure from that used in single LWF building process. Here is an example of the configuration file, which could also be found in [Examples].

```
[subspace_split]
q_mesh = [2, 2, 2]
permutation_repeat_times = 4

[[lwfs]]
center_atom_id = 1
wyckoff_irrep_label = "T1u"
```

(continues on next page)

(continued from previous page)

```
output_prefix = "lwf_u"
branch_select_strategy = "spillage_iter"

[[lwfs]]
center_atom_id = 0
wyckoff_irrep_label = "T1u"
output_prefix = "lwf_v"
branch_select_strategy = "spillage_iter"

[[lwfs]]
center_atom_id = 2
wyckoff_irrep_label = "A2u"
output_prefix = "lwf_q"
branch_select_strategy = "spillage_iter"

[[lwfs]]
center_atom_id = 2
wyckoff_irrep_label = "Eu"
output_prefix = "lwf_w"
branch_select_strategy = "spillage_iter"
```

Here, a new table called `[subspace_split]` must present. This table provides configurations for splitting the band subspaces into different parts (each for an LWF). The q-point mesh is specified by `q_mesh` and such mesh is shared by all the LWFs to be built in this run. The array `lwfs` contains the configurations for each LWF, each element has the same format as the `lwf_config` table in single LWF build process.

Briefly, the multiple LWF building process contains the following steps:

1. Split the subspaces for LWFs.
2. Build each LWF in its own subspace.

Subspace splitting

The task of subspace splitting is to split the band space into several non-intersect subspaces, each for an LWF and meet the symmetry requirement, on which the LWF will be built. The "subspace" here are defined in "discrete" sense, spanned by eigenvectors (not their general linear combinations) of the dynamical matrices at each q-point. This is similar to *Find band subspace discretely*.

Presently, the subspace splitting is done by a simple greedy strategy, containing several "sweeps". In each sweep, the subspaces for the LWFs are built one by one in some specified order. For each LWF, the normal modes already chosen by another LWF subspace in this sweep cannot be chosen again. During this process, a target function is computed to evaluate such choice (split) of subspaces. After all sweeps, the subspace choice with minimal target function is selected as final output.

Clearly, the order of the sweeps should not affect the result, however different LWF order inside each sweep may result in different selection. There are two strategies for controlling the orders, specified by `subspace_split.lwf_order_sweep_strategy`. If it is set to `"full_permutation"` (the default), all the possible permutations of the LWFs are considered (in total $n!$ possibilities, where n is the number of LWFs to be built). In each sweep, one possible permutation of the LWFs is considered. This process could be repeated several times to avoid possible random effects, using the `permutation_repeat_times` key (default is 1). If `subspace_split.lwf_order_sweep_strategy` is set to `"random"`, several random permutations of LWF ordering are considered. The number of such permutations is set by `lwf_sweep_random_count` (default: 5).

The type of target function is determined by the `branch_select_strategy` selected by each LWF (in the `lwfs` array). The target function values for each case are defined as follows:

- For `branch_select_strategy = "min_energy"`, the target function is defined as the summation of energy (frequency, in THz) of each selected normal modes at each q-point.
- For `branch_select_strategy = "max_projection"`, the target function is the negative value of the summation of projection of each selected normal modes on the anchor modes.
- For `branch_select_strategy = "spillage_iter"`, the target function is the negative value of the spillage functional.

The total target function value is simply the summation of target function values of each LWF subspace. An optional weight could be assigned to each of the LWFs by the `subspace_split.target_function_weights` key, for example, `[1, 2.5]` indicates the weight 1 and 2.5 are applied to the first and second considered LWF, respectively.

Note

It is possible to select different `branch_select_strategy` for each of the LWFs. However, it is not recommended, as the unit of the target function values may be different. In this case, the weight need to be carefully chosen.

Once the subspace splitting is done, the output files `lwf_subspaces.json` and `lwf_subspaces.pickle` are automatically generated. These two files should contain exactly the same amount of information, i.e., the subspace selection for each LWF at each of the q-points. These files could be used to provide subspace information in single LWF building process. More precisely, set `branch_select_strategy = "load"` and specify the index of the desired LWF in the file (i.e., in the `lwfs` array when generating this file) to load the subspace selection, and continue the single LWF building process. See *Find band subspace discretely* for more information.

Note

Two different files are provided for the subspace splitting data file due to technical reasons. The `json` file is generated by the `dataclass-json` Python package, which may encounter some problems when the data format is complex. In such cases, the `json` file is not produced. The `pickle` file is generated by the Python standard library `pickle`, which should not encounter any problem (as far as our tests covered) in generation, but loading this file may require the same interpreter and library versions. By default, the `branch_select_strategy = "load"` process loads the `json` file.

LWF building

After the subspace for each LWF is determined, the next task is to build the LWF in such selected subspace. This process is exactly the same as that in single LWF building process. The details of each LWF is controlled by corresponding element in the `lwfs` array in the input file, which has the same format as the `lwf_config` table in the single LWF building configuration file.

However, there are still some differences; some configurations have no effect in multiple LWF building process. For example, the `q_mesh` for each LWF must be the same as that defined in `subspace_split`, this is automatically ensured by this program. Moreover, the `optimize_subspace_basis` option is ignored, and the continuous optimization of subspace basis is never performed.

Run multiple LWF building

To run the multiple LWF building process, use the command like

```
python3 -m heff_gen.lwf.cui.gen_multiple_lwf_phonon
```

with optional argument of the configuration file name (by default, `lwf_gen_multiple.toml`). The example here could be found in the `lwf/phonon/BaTiO3` directory of [Examples].

The program will present the LWF phonon dispersion as follows:

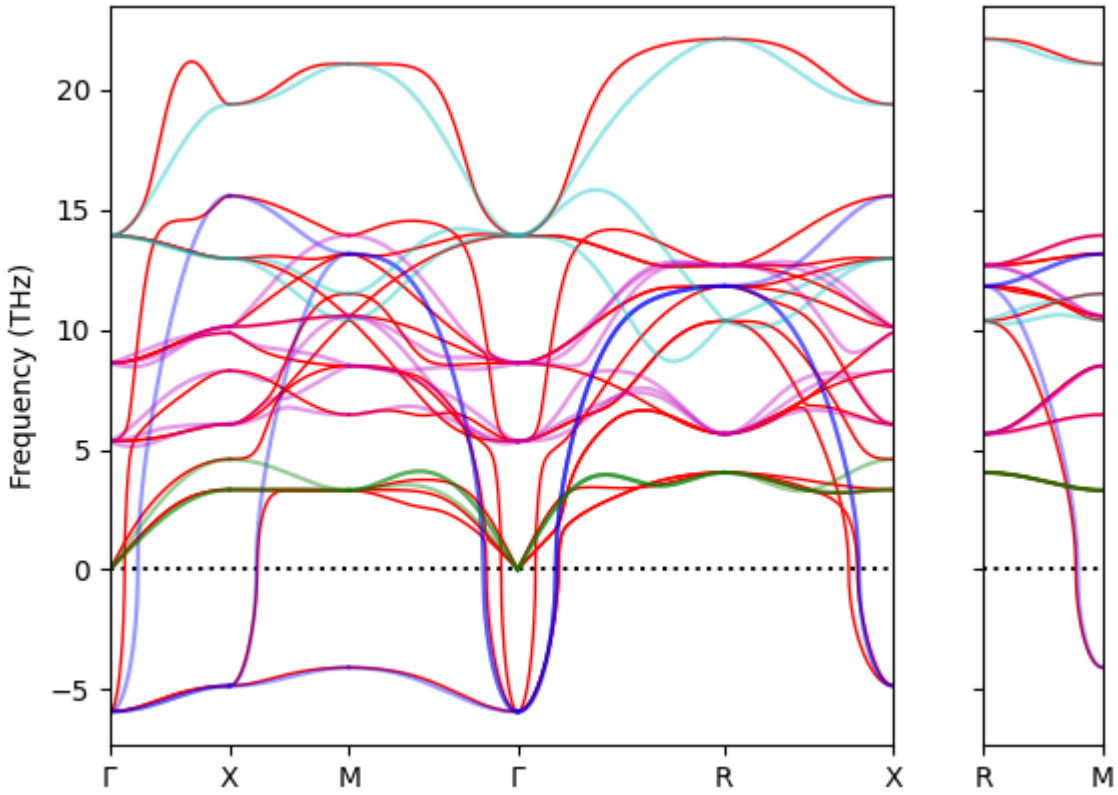


Fig. 3: Phonon dispersion of $BaTiO_3$ from DFT (red curves) and LWFs (other colors)

Here in this example, four different LWFs are built for $BaTiO_3$, containing all the dynamical DoFs in the unit cell, thus all the normal modes (at each high symmetry q-point) are covered by the LWFs. The LWFs include the Ba-centered mode with `T1u` irrep, Ti-centered mode with `T1u` mode, O-centered mode with `Eu` and `A2u` irreps.

The program then produces the amplitude bar plot of each LWF, and then the regenerated LWF dispersion, like the single LWF building process.

5.2.2 Real-space method

> **Note**
>
> We emphasize again that the term "LWF" used in this package (and also this document) is actually *generalized*. It is especially this case for the real-space method to be introduced here, where the LWF may be not (only) fitted from phonon dispersion information, but also from the atomic displacement, etc.

The real-space method for building the LWF, in contrast to the reciprocal-space method, does not use the reciprocal space phonon normal modes at the beginning. Instead, a set of "basis" for the LWF is found in real-space. Each basis is actually a basis function for the selected irrep of the center site point group (for the selected band representation of the LWF), typically containing atomic displacements in one or several shells (with respect to the center site). The linear combination of these basis modes forms the desired LWF, and the scales for each of the basis modes are the parameters to be fitted. The fitting process may consider the atomic displacements of some selected structures, the phonon normal modes, and the localization of the LWF. Building the LWF with real-space method takes the following steps:

1. Build the real-space symmetry basis modes for the LWF.
2. Fit the parameters (the amplitudes of each basis mode) for the LWF.

Build symmetry basis

The basis modes of the LWF could be built using the projector operator, with the help of `spgrep` Python package. More precisely, the atomic displacement representation (for the given atomic structure) in a finite (periodic) supercell is projected onto the chosen irrep of the center site group (which defines the band representation of the LWF). The input includes the unit cell atomic structure file, the center site index (zero-based) for the structure, the irrep label, and the supercell dimension. Since the projection is done only in a finite supercell, only the basis modes containing only atoms *without* supercell periodic ambiguity are kept (the same as *Build shelled-LWF*; actually the basis is converted from supercell format to shelled format with the same routine).

To build the symmetry basis, use the command like:

```
python3 -m heff_gen.lwf.cui.gen_symm_basis -d <supercell_dim> -n <center_site_id> -l
↪<irrep_label> -o <output_prefix>
```

where `<supercell_dim>` specifies the supercell dimension, `<center_site_id>` is the (zero-based) index of the center atom in the input unit cell, `<irrep_label>` is the chosen irrep label of the center site point group for the LWF to be built, and `<output_prefix>` is the prefix to the output filenames. For example, the following command builds the symmetry basis for perovskite structure with B site as center, and `T1u` irrep label, using a $3 \times 3 \times 3$ supercell:

```
python3 -m heff_gen.lwf.cui.gen_symm_basis -d 3 3 3 -n 1 -l T1u -o basis_u
```

More possible options could be seen by using the `--help` option, or `-h` in short.

The main outputs are the `<output_prefix>_data.json` and `<output_prefix>_symmetry.yaml` files, in same format as the shelled-LWF files. Note that the amplitudes in the shelled-LWF file are meaningless; the actual amplitudes should be fitted (see below).

TODO: LWF fitting

5.2.3 Further details for LWF building

Acoustic LWF

In the phonon dispersion, the acoustic modes are somewhat special because their frequencies are approaching to zero as the wave vector q is approaching to Gamma point. Such zero frequency modes are responsible for the translation of the whole supercell. In the effective Hamiltonian, the LWFs with subspace including the acoustic modes (which we shall call *acoustic LWF*) are also somewhat special, and should be treated specially in some cases.

According to the behavior at Gamma point, all LWFs could be divided into three types: full-acoustic LWF, partial-acoustic LWF, and non-acoustic LWF. If the phonon dispersion of *all* branches in the LWF have zero frequency at Gamma point, it is a full-acoustic LWF. If only some of its branches have zero frequency at Gamma point, it is a partial-acoustic LWF. If none of its branches have zero frequency at Gamma, it is a non-acoustic LWF. The acoustic mode (inhomogeneous strain variable) v in the $BaTiO_3$ model (see [Zhong1995]) is an example of full-acoustic LWF; the dipolar mode (soft mode) u in the same model is an example of non-acoustic LWF.

The formal definition of the acoustic LWF is as follows. Let $w_k(R_l, j, a)$ denote the LWF atomic displacement, where R_l is the unit cell position about the center site, k is the LWF axis index, j is the atom index inside the unit cell, and a is the Cartesian direction index. The Fourier transform of LWF is (extra scalar factor neglected)

$$w_k(q, j, a) = \sum_l w_k(R_l, j, a) \exp(-iq \cdot R_l).$$

Define the *acoustic transform matrix* A such that the ϕ_t

$$\phi_t(j, a) = \sum_k A_{kt} w_k(\Gamma, j, a)$$

is a simple translation mode (all atoms in the unit cell move along the same direction with the same amount), where $w_k(\Gamma, j, a)$ is the LWF at Gamma point ($q = 0$ point). In other words, the acoustic transformation matrix A converts $w_k(\Gamma)$ to a set of vectors $\phi_t(j, a)$ such that $\phi_t(j, a) = \psi_t(a)$ for any j. If there is no such transformation (impossible to find such A), the LWF is non-acoustic. If the number of ϕ_t vectors is equal to the LWF dimension (the A matrix is a squared matrix), the LWF is full-acoustic. Otherwise, the LWF is partial-acoustic.

When the reciprocal-space method is used for building LWF, the program tries to find the acoustic transformation matrix if `lwf_config.acoustic` is *not* set to `"No"`. The `acoustic` attribute will be set to `"Full"`, `"Partial"`, or `"No"` by finding the eigenvectors associated with the zero eigenvalues for the LWF dynamical matrix at Gamma point. If the LWF is non-acoustic, the program will warn if `lwf_config.acoustic` is set to `"Yes"`, while the program does nothing if `lwf_config.acoustic` is set to `"Auto"` (the default).

When the real-space method is used for building LWF, the acoustic property of each LWF to be built should be specified in the input `lwf_configs` array, using the `constrain_acoustic` key, using `"No"` (the default), `"Full"`, or `"Partial"`. If `"Partial"` is specified, the directions of the supercell translation must be specified either by `acoustic_axis_ids` (using zero-based indexes of Cartesian directions) or by `acoustic_directions` (using 3-dim vectors in Cartesian coordinate directly). This will impose constrains when the LWF parameters are optimized. Please remember to check whether this constrain is satisfied when the program finishes (by verifying whether the acoustic target function value is sufficiently small).

In principle, the effective Hamiltonian should satisfy the so-called acoustic summation rule (ASR) (see, e.g., [SecondPrinciples2017]), which states that the translation of the whole supercell should result in no energy change and no force. However, in the effective Hamiltonian, though this ASR should (in principle) be automatically satisfied by the fitting process (up to some numerical errors), no *exact* guarantee is provided for the ASR, as the effective Hamiltonian is typically highly anharmonic, and imposing this ASR exactly could be inconvenient. In this program, we provides some special treatments for the acoustic LWFs, which are enabled by default:

- For simulation and optimization, the initial momenta and calculated forces are constrained such that the projection on the supercell translation is zero. This guarantees dynamical process never causes a supercell translation. This could be otherwise disabled by setting `simulation.disable_sum_rules` or `optimization.disable_sum_rules` to `true`.
- For effective Hamiltonian phonon dispersion, the ASR is imposed to the periodic dynamical matrix. This could be disabled by setting `phonon.disable_sum_rules` to `true`.

5.3 Prepare unit cell metadata

The unit cell metadata includes the reference atomic structure, LWF information, mixed site information (if any) and other related information. It forms an important part for the model. Before building the interactions and fitting the parameters, the unit cell metadata must be prepared. The unit cell metadata are stored in two sets of files:

- The `heff_unitcell.yaml` file;
- The `unitcell_spec.toml` file, reference atomic structure file, and LWF file (including `<prefix>_data.json` and `<prefix>_symmetry.yaml`).

The latter is used only in model building, while the former is used in both model building and simulation. The former should typically be generated from the latter.

The unit cell metadata mainly includes the following parts:

- The basis vectors of the unit cell.
- The *sites* in the unit cell. Each *site* is a position where an atom located in.

- The LWF information.
- The mixed site information.

Here, the sites with LWF centered on or with mixed site is called *non-trivial sites*, while others are trivial sites. By default, only the non-trivial sites enter the effective Hamiltonian model. Each site, mixed site and LWF is assigned with a name. The name may be given by the user (recommended) or generated automatically by the program, and will be used later in the interaction generation step. These names must be distinct from each other, and special characters are not recommended.

5.3.1 Unit cell specification file

The unit cell specification file (typically `unitcell_spec.toml`) should typically be edited by the user. The following information should be included:

- Reference structure (unit cell) atomic structure filename;
- LWF filename (prefix) and information;
- Mixed site information (if any);
- Additional unit cell metadata.

Tip

In the unit cell specification file, you will need to write the *selected* LWF information. The *selected* LWFs are those served as basis for expanding the atomic displacements in the Heff model. The selected LWFs forms a subset of the full set of LWFs for the reference structure. In many cases, this is a *proper* subset (meaning that some possible LWFs may not be included in the selected subset). The more LWFs selected as basis, the more precision the *structural fitting* will be, but the more interactions will need to be generated (meaning much more computational cost later). The selection of LWFs varies for different models. One basic rule is that the selected LWFs should be able to expand the ground state or low energy structure that to be studied properly (see *Check selected local mode basis*). **We suggest to adjust the selected LWF subset and check it. This process may be repeated for several times util the necessary LWF subset is selected.**

Here is a simple example of the unit cell metadata file for $BaTiO_3$, which could also be found in the directory `unitcell_meta/BaTiO3` in [Examples]:

```
[unitcell]
model_name = "BaTiO3"
eps_inf = 6.864493

[[unitcell.lwfs]]
name = "u"
file_prefix = "lwf_u"
dipolar = "LongRange"

[[unitcell.lwfs]]
name = "v"
file_prefix = "lwf_v"
dipolar = "NonPolar"
```

Most of data are given in the `[unitcell]` table. The `model_name` is an optional name for the model to be built. The optic dielectric constant ϵ_∞ should be given with the `eps_inf` key, which accepts a scalar, a 3×3 matrix or three-dimensional vector (the diagonal elements of a diagonal matrix). The values of the optic dielectric constant should be found by FP calculations, and this is required for models with at least one long-range dipolar LWF (see below). The site

names could be automatically given by the `site_names` key, with an array of string with length equals to the number of atoms in the unit cell. If not given, the site names will be automatically assigned by the program, using the format `<element><index>`, for example `Ba1`.

The LWF information should be given in the `unitcell.lwfs` array, with each element specifies the information for each LWF (local mode basis). For each LWF, a unique name should be given by `name` key, the `file_prefix` key specifies the prefix to the LWF filenames (i.e., `<file_prefix>_data.json` and `<file_prefix>_symmetry.yaml`). The `dipolar` key specifies the dipolar attribute of the corresponding local mode, which accepts three possible values:

- `dipolar = "LongRange"`: The long-range dipolar interaction should be considered for this type of local mode. Born effective charge should be given (in the parameter file, see below).
- `dipolar = "NonPolar"`: The dipolar interaction is not considered for this type of local mode, and the local mode is considered to be non-polar, without contribution to the polarization.
- `dipolar = "ShortRange"`: The long-range dipolar interaction is *not* considered, but it contributes to the polarization. This option should be considered as an approximation for the cases where the long-range interaction is negligible.

Warning

The `dipolar = "ShortRange"` option is experimental. We are not sure whether this is a proper treatment on physical level. Problems may be encountered in implementations. Use with special care!

Here is another (slightly more complex) example of the unit cell specification file, which could also be found in the directory `unitcell_meta/AlScN` in [Examples]:

```
[unitcell]

model_name = "AlScN-hex"

unitcell_filename = 'POSCAR'
unitcell_interface = 'vasp'

eps_inf = [4.51932650, 4.51932650, 4.96092600]   # Symmetrized

[unitcell.config]
symprec = 1e-5

[[unitcell.lwfs]]
file_prefix = "lwf_Al1"
name = "v"
dipolar = "NonPolar"
irrep_label = "E'"

[[unitcell.lwfs]]
file_prefix = "lwf_Al2"
name = "q"
irrep_label = "A2''"
dipolar = "NonPolar"

[[unitcell.lwfs]]
file_prefix = "lwf_N1"
name = "u"
```

(continues on next page)

(continued from previous page)

```
dipolar = "LongRange"
irrep_label = "E'"

[[unitcell.lwfs]]
file_prefix = "lwf_N2"
name = "p"
dipolar = "LongRange"
irrep_label = "A2''"

[[unitcell.mixed_sites]]
concentrations = [1.0, 0.0]
elements = ["Al", "Sc"]
generator_site = 0
name = "A"
variable_concentration = true
```

Here, the unit cell atomic structure filename is explicitly specified to `POSCAR`, and the interface type is explicitly specified to `vasp`. The unit cell atomic structure file is loaded using the Python package ASE, any file readable by `ase.io.read` could be specified here. Here the irrep labels of the LWFs are also given. They are only provided for reference in present version.

Here, the mixed site (see *Select reference structure*) is specified with the `unitcell.mixed_sites` array, with each element specifies a type of mixed site. Here in the (Al,Sc)N example, we have only one type of mixed site called `A`, which may be actually occupied by either Al or Sc element at each site. The concentrations of each element should be given with `concentrations`, which accepts an array of floating-point values. The concentration of each element must be in range $[0, 1]$, and the total concentration of all elements must be 1.0. The location of the mixed site must be given by either `generator_site` (using a single integer index) or `site_ids` (using an array of integer indexes, of *all* the sites) with the site index (zero-based) in the input atomic structure file. The provided `site_ids` (if present) must be consistent with the symmetry; in other words, the given sites must form an orbit of the space group symmetry. The `variable_concentration = true` indicates the concentrations of elements in this mixed site may be variable, that is, the same model (defined by the unit cell metadata, interactions and parameters) could be applied to different concentrations (may be limited in a certain range). For example, the same model could be applied to (Al,Sc)N with 5% Sc or 10% Sc, etc.

Tip

Some TOML implementations does not accept arrays mixed with integers and floating-point numbers. Thus, use `concentrations = [0.0, 0.5, 0.5]` instead of `concentrations = [0, 0.5, 0.5]`.

Important

Once the unit cell metadata is prepared, they should not be changed under any circumstances. Particularly, the unit cell atomic structure file should not be changed in any form. If you do need to change the unit cell metadata (including the unit cell atomic structure file, the LWFs, the mixed site information, etc), you will need to restart the *whole* model building process from here (or even along with LWF building process)! In this sense, double check the unit cell metadata before going further.

5.3.2 Generate unit cell YAML file

Once the LWF files and the unit cell specification file is prepared, the unit cell YAML file `heff_unitcell.yaml` could be generated using the command

```
python3 -m heff_gen.cell.gen_unitcell_yaml
```

The obtained `heff_unitcell.yaml` will be used in later interaction generation, parameter fitting, and simulations.

There are also several options for the generation process, given in `unitcell.config` table. An important one is `symmetrize_eps_inf`. If it is set to `true` (default is `false`), the program will symmetrize the given `eps_inf` tensor according to symmetry. More precisely, as the optic dielectric tensor is a tensor property of the crystal, it should obey the Neumann's principle, and takes a "symmetry-adapted" form. In this case, the output will contain the symmetrized tensor, like

```
INFO:root:eps_inf after  symmetrize:
[[ 4.51932650e+00 -3.09533014e-16  0.00000000e+00]
[-3.09533014e-16  4.51932650e+00  0.00000000e+00]
[ 0.00000000e+00  0.00000000e+00  4.96092600e+00]]
```

5.3.3 Check selected local mode basis

Before building the interactions and parameters (for the interactions), it is of vital importance to verify the LWFs (local mode basis) are chosen properly. For many cases, the chosen LWFs span only a subspace of the full atomic displacement space. For example, for $BaTiO_3$ (the model built in [Zhong1995]), only u and v modes are included in the effective Hamiltonian, covering 6 DoFs in each unit cell, comparing to 15 atomic DoFs in each unit cell (i.e., five atoms in each perovskite unit cell, each with three displacement DoFs). In these cases, one should verify that the most important DoFs could be included in the chosen subspace. In other words, the omitted DoFs should be unimportant, or at least irrelevant to the studied problem. Typically, this requires the desired distorted structures could be well fitted by the chosen local modes.

Important

Here in this section, we talks about fitting the *structures*, not energies. Obviously, one atomic structure must first be well fitted by the effective Hamiltonian DoFs (to some precision) before its energy could be well fitted by the effect Hamiltonian.

To describe the fitting precision of a distorted structure, we use a quantity $\bar{\Delta x}$, to be called "displacement norm", defined between two atomic structures as

$$\bar{\Delta x} = \sqrt{\sum_{ia} (\Delta x_{ia})^2},$$

where the indexes i, a loop over all atoms in the supercell and the three Cartesian axes, respectively; Δx_{ia} is the *relative atomic displacement* between the two structures.

Note

The displacement norm quantity between two atomic structures is valid *only if* one structure could be treated as another structure with (relatively small) atomic distortion (including lattice strain) and vice versa. This indicates the two structures must have same number of atoms, chemical species.

To analyze the structure fitting precision, an utility command line user interface (CUI) program called `extract_disp` is

provided. Though optional, it is strongly recommended to check the structure fitting precision (of the desired structures) before going further, using this CUI program.

The `extract_disp` CUI program provides the following functions:

- For a given distorted atomic structure, find the atomic displacements with respect to the reference structure.
- Project the atomic displacements into the local mode space, i.e., get the local mode variables for the structure.
- Check the atomic distortion fitting precision.
- Compute Born effective charges (BECs) for the LWFs.

An optional input file (default name is `disp_analysis.toml`) could be supplied for this program. This file includes three TOML tables: `energy`, `cell`, and `disp_analysis`. In addition, the unit cell metadata (`unitcell_spec.toml`, unit cell atomic structure file and LWF files) should present in the same directory. The distorted atomic structure file (default: `CONTCAR`) should also present in the same directory.

Note

It is a normal case to extracting the displacement of a supercell. Thus, it is normal that the distorted atomic structure file have different number of atoms from the unit cell structure file. However, the distorted structure file must be a *slightly* distorted structure (comparing to the unit cell lattice constant) of the supercell of the unit cell *defining under the same Cartesian coordinate*. No rotation operation is allowed between the reference structure and distorted structure. And the translation should be avoided to avoid ambiguity in structure identification.

Warning

We emphasize again that the unit cell structure file should never be changed, otherwise you will need to restart the whole building process. If the unit cell and supercell structure files are not consistent with each other, always change the distorted structure file to make it align with the unit cell structure.

To run this program, use the command

```
python3 -m heff_gen.cell.extract_disp
```

An example could be found in the `unitcell_meta/BaTiO3` directory in [Examples]. Some important information are presented in the output. First is the atomic displacement norms. There are three displacement norm values. The first value is the displacement norm between the reference structure and the input distorted structure file (`CONTCAR`). The second is the displacement norm between the reference structure and the distorted structure regenerated from the local mode basis set. These amount of atomic displacement could be properly spanned by the effective Hamiltonian basis (the local mode basis, or LWFs). The third, "structure fitting error", is the displacement norm between the original input distorted structure and the regenerated distorted structure. Obviously, this value should be small if the input distorted structure could be well fitted by the current effective Hamiltonian basis. And it is exactly zero if the local mode basis set covers all atomic DoFs. In this case, the fitting error is 0.01487 Angstrom, which is small comparing to the displacement norm of input structure 0.171627 Angstrom, indicating the input structure could be well fitted by the local mode basis set.

```
INFO:root:original struct disp norm:     0.171627 uc-wise: 0.171627
INFO:root:regenerated struct disp norm:  0.170983 uc-wise: 0.170983
INFO:root:structure fitting error:       0.014847 uc-wise: 0.014847
```

Tip

If `StructureExtractFailed` exception is encountered, check whether the given distorted structure is consistent with the reference structure (unit cell structure) file. They must be defined under the same Cartesian coordinate, and we recommend to avoid lattice translations, that is, make the atoms of the same sub-lattice sit at the origin point. In some cases, especially when the strain (with respect to the reference structure) is relatively large, the automatically guessed supercell matrix may be incorrect. In this case, provide the supercell matrix of the distorted structure in the `cell` table of the input file (by default, `disp_analysis.toml`) may be helpful.

If `disp_analysis.analysis_polar` is set to `true` (default is `false`), the following information will also be presented (this requires an additional input file, the BEC file, default name is `BORNCHG`):

```
INFO:root:Total polarization (atomic): [0.25118198 0.25118198 0.25118198]
INFO:root:Total polarization (mode):   [0.25026953 0.25026953 0.25026953]
INFO:root:Polarization mode contribution:
[[2.50269463e-01 2.50269463e-01 2.50269463e-01]
[7.17189592e-08 7.17189591e-08 7.17189591e-08]]
INFO:root:Mode Born effective charge tensors:
[[ 1.08107246e+01 -1.71764038e-15 -2.86685055e-15]
[ 3.86075683e-15  1.08107246e+01 -4.16377540e-15]
[-1.27779459e-15  2.09267828e-15  1.08107246e+01]
[ 6.81051473e-06 -2.54151160e-15  7.12401100e-16]
[ 2.42786708e-15  6.81051472e-06  3.03247921e-15]
[-5.08382087e-15 -1.85613462e-15  6.81051472e-06]]
INFO:root:Projected BEC for mode u: [10.81072456]
INFO:root:Projected BEC for mode v: [6.81051472e-06]
```

The input file, (by default) `BORNCHG`, contains the Born effect charges (BECs) of the *atoms* in the unit cell. If `VASP` is used for first-principles calculations, this could be obtained from a part of `OUTCAR` file. If other first-principles packages are used, the input file should be converted to such format. Example of such input file:

```
ion    1
    1     2.75275     0.00000     0.00000
    2    -0.00000     2.75275     0.00000
    3     0.00000     0.00000     2.75275
ion    2
    1     7.46441    -0.00000    -0.00000
    2    -0.00000     7.46441     0.00000
    3     0.00000     0.00000     7.46441
ion    3
    1    -5.90501     0.00000     0.00000
    2     0.00000    -2.15607    -0.00000
    3    -0.00000     0.00000    -2.15607
ion    4
    1    -2.15607     0.00000     0.00000
    2     0.00000    -5.90501    -0.00000
    3     0.00000    -0.00000    -2.15607
ion    5
    1    -2.15607     0.00000    -0.00000
    2     0.00000    -2.15607    -0.00000
    3     0.00000     0.00000    -5.90501
```

Where, the BECs of each atom are given in the order of their presence in the unit cell atomic structure file (default:

`POSCAR`). The BECs of each atom starts from one title line which is not actually read, followed by three lines, each with a starting index (1, 2, or 3) followed by three floating-point values. Such three lines, in total 9 values, provide the BEC values (in tensor format) for the corresponding atom. All values should be split by whitespace characters (space, or tabs). No extra lines allowed before all the input content finished.

The first two lines shows the polarization computed using the Born effective charge, from both atomic displacements and mode distortions, respectively. The polarization from each mode contribution is then presented. The BECs (in tensor form) for all LWFs are then presented, where each row corresponds to the BEC for one LWF axis. Finally, the "projected" BECs for LWFs are shown. These projected BECs are symmetry-adapted BEC values, defined with respect to the "directions" of the LWF axes, which could be used for input later in the parameter file. Specifically, let d_{kja} be a set of vectors, called "polar directions", associated with the LWF index k, where j is the index of polar directions for this axis, and a is the Cartesian direction index. These vectors could be found by symmetry (which has been done in the LWF generation step) and are stored in the LWF data JSON files. The BEC tensor for the LWF, Z_{ka}, is then computed from

$$Z_{ka} = \sum_j Z_j d_{kja},$$

where Z_j is the "projected" BEC introduced above. The program here projects the Z_{ka} tensors to get the Z_j values.

Several output files are also generated. The regenerated atomic structure, which is generated from the extracted local mode displacement, is written to (by default) `POSCAR_regen`. This file could be used to check the structure fitting precision with FP calculations. For example, do FP calculations on the original distorted structure and the regenerated structure, and compare their energies. The extracted local mode displacements, strain and site occupations are written to (by default) `extracted-modes.txt`, `extracted-strain.txt`, `extracted-sites.txt`, all in plain text format.

5.4 Generate interactions

Important

At this stage, you should have chosen your set of LWFs (local modes) for this model. All the following steps rely on *fixed* LWF set. If you would like to change the set of LWF later, you will need to re-build the unit cell metadata, then re-generate the interactions before going further. Thus, please carefully check your chosen LWFs before going down.

The interactions in the effective Hamiltonian are the energy coupling terms involving the local mode variables, strain variables, and mixed site occupations. The interactions are formally terms in Taylor expansion (within some range) of the potential energy landscape around the reference structure that meet the symmetry requirements. For more theoretical background information, please refer to our article [ALFE-H-2025] and references therein. Here in this document, we only focus on the technical and practical aspects.

The interactions could be divided into long-range dipolar interactions and other interactions (called "local" interactions). The long-range dipolar interactions take fixed form, while the local interactions are different according to selected reference structure, local mode basis set, and certain cutoff conditions. The interaction generation is controlled by a TOML input file, default name is `interact_config.toml`. The file mainly contains a table called `interact_config`. The simplest example for this configuration file is just an empty file (but this file must present), leaving everything default. More examples of this file could be found in [HeffCXXTestData] (see `unitcells` directory).

Warning

Generation of the interactions is of *vital importance* for building an effective Hamiltonian model. Although default values have been provided for all of these generation configurations, they cannot be suitable for all the cases. **Please**

check the generation configurations carefully before going further!

To begin, we first introduce some basic concepts or terms used in this program and this document.

5.4.1 Basic concepts and hierarchy of interactions

The local interactions are arranged in a hierarchical framework. From bottom to up, there are following levels:

Note

To avoid misunderstandings, here (and henceforth) we use abbreviations with capital letters to indicate the terms (level names).

1. Component (COMP). A COMP is the local mode or strain along a certain direction, or a site occupation variable. For example, u_{i1}, the local mode called u along its first axis at unit cell i, is a COMP. The local mode u along a non-axial direction, for example the $[110]$ direction, is also a COMP. There are three possible types of COMP: the local mode COMP, indicating the local mode variable at one unit cell along certain direction; the strain COMP, indicating the strain along certain direction; and alloy COMP, or occupation COMP, indicating the atomic occupation at a mixed site in one unit cell (the σ variable in [ALFE-H-2025]).
2. Product (PROD). A PROD is the production of one or more COMPs. For example, $u_x u_y$ is a PROD containing two COMPs.
3. Symmetry-adapted term (SAT). An SAT is a combination of a set of PRODs that are linked with symmetry operations. In other words, an SAT includes a set of symmetrically-equivalent PRODs. Due to such equivalence, all PRODs in one SAT have equivalent contribution to the potential energy. Such contribution is measured by an coefficient, which is the parameter to be fitted from FP. Formally, the local interaction energy could be written as

$$E = \sum_{\lambda} w_\lambda t_\lambda = \sum_{\lambda} w_\lambda \sum_{j} p_{\lambda j},$$

where t_λ is the SAT, w_λ is the corresponding parameter, and $p_{\lambda j}$ is the PROD belonging to the SAT.

4. SAT group (SGRP). An SAT group is a set of SATs that shares similar or same distances between the components inside each PROD.
5. Interaction (INT). An INT contains several SGRPs, and includes *all* considered interactions that consists of the same local modes with same order. For example, an INT may contain *all* interactions with linear contribution from local mode u and square contribution from mode v, called $u^1 - v^2$ INT (similar term is also used in [ALFE-H-2025]); all considered interactions with square contribution from u and linear contribution from v forms another INT, namely, $u^2 - v^1$ INT.
6. Interaction category (ICAT). An ICAT contains *all* considered INTs that consist of same *type* of variables (local mode, strain, or occupation). For example, both $u^1 - v^2$ INT and $u^2 - v^1$ INT belong to the $u - v$ ICAT.

In the hierarchy above, the SGRP and ICAT levels are auxiliary levels, that may be skipped in some scenes. For example, the parameter file (`heff_params.yaml`) is arranged using the INT-SAT hierarchy; the interaction file (`heff_interactions.json`) is arranged using the INT-SAT-PROD-COMP hierarchy, etc.

Currently, the local interactions could be divided into 8 types (ITYPs):

1. Mode intersite (ModeInter) interaction, or mode-mode (MM) interaction. Each term contains several COMPs that do not locate at the same site. Currently, this interaction is limited to two-body interaction, i.e., up to two COMPs may take part in one PROD.
2. Mode-strain interaction, or mode-elastic (ME) interaction, containing the interactions between strain variable and local mode variables. Currently, each PROD may include local interactions on at most two sites. Note that

ME interaction with two sites in one PROD may cause great computational effort, and the support to this is experimental.

3. Mode-Alloy (MA), known as "spring" interaction (see [ALFE-H-2025]), describing the "alloying effect" on the local modes. Currently, only onsite local mode interactions are considered, and at most two site occupation variables (at different sites) are considered.
4. Onsite (ON) interaction, including only local modes at one site for each term.
5. Alloy intersite (AlloyInter) interaction, or alloy-alloy (AA) interaction. Describes the *direct* contribution from atomic occupations to energy, including only alloy COMPs. Currently, this is considered up to 2 occupation sites.
6. Strain-alloy interaction, or elastic-alloy (EA) interaction. Describes the coupling between strain and atomic occupations. Currently, this is considered up to 2 occupation sites.
7. Strain-spring interaction, or elastic-spring (ES) interaction. Including all of the three possible types of COMPs, namely, mode, strain and alloy COMPs, in each PROD. Currently, this is considered up to mode COMPs at one site and alloy COMPs at two sites. Available as of version 0.15 (Oct. 7, 2025). This type is currently only experimental and not enabled by default. To use this, specify `enable_strain_spring = true` in input file.
8. Elastic (EL) interaction, including only strain variables.

5.4.2 Alloy model level

For effective Hamiltonian models containing mixed sites (alloy models), there are several different "fitting levels", indicating the model will be built for different scopes. The different choice of such alloy model level mainly affect some low-order interaction terms. Such special treatment on these low-order terms comes from the fact that for some cases they could not be fitted from the data.

The following keys in `interact_config` table controls the related interaction generation:

- `include_elastic_first_order`, default is `true`. If `true`, the linear elastic INT (with each PROD consists of only one strain COMP) and strain-alloy INT with linear strain contribution (with each PROD includes only one strain COMP) are included; otherwise not included. Note that for non-alloy models (models *without* mixed sites), the energy contribution of this INT is in principle zero, because the reference structure is typically a (at least local) minima of potential energy with optimized lattice constants. Nevertheless, it is also possible to include this term in this case, and parameters close to zero are expected to be obtained from fitting.
- `disable_alloy_inter`, default is `false`. If `true`, the AlloyInter (or AA) interaction is disabled.
- `disable_strain_alloy`, default is `false`. If `true`, the strain-alloy (EA) interaction is disabled.
- `include_alloy_nonlocal_first_order`, default is `false`. If `true`, the "non-local" alloy interactions with first order contribution from occupation are included. Here the "non-local" refers to strain-alloy and AlloyInter interactions, because they do not dependent on local modes. Extra cares must be taken on these terms because (it is easy to show that) these terms are dependent only on the *concentration* of the mixed site elements, but not their arrangement. Thus, if the model is only fitted at a fixed concentration [for example, in (Al,Sc)N, only fitted with 25% Sc concentration], the contributions from these terms are constant, and cannot be distinguished from the reference energy.

In addition to the (somewhat detailed) controlling keys above, there is another key, `alloy_model_level`, that automatically set the values for some of the keys above. It is strongly recommended to use this key. It may take the following values:

- `alloy_model_level = "NotSet"` (the default), nothing to do.
- `alloy_model_level = "FixedConcRelative"`. The model will be fitted only for a fixed concentration, and the energy between different atomic ordering will not be compared. In this case, `disable_alloy_inter` is

set to `true`, `include_alloy_nonlocal_first_order` is set to `false`, other keys are not changed. This configuration is consistent with most of the previous effective Hamiltonian implementations for perovskite solid solutions, like PZT, BST, etc. See [ALFE-H-2025] and references therein.

- `alloy_model_level = "FixedConcAbsolute"`. The model will be fitted only for a fixed concentration, but the energy between different atomic ordering should be comparable. Such comparability is enabled by incorporating AA interactions and possibly EA interactions. In this case, `disable_alloy_inter` is set to `false`, `include_alloy_nonlocal_first_order` is set to `false`.
- `alloy_model_level = "VariableConc"`. The model will fitted for a variable range of concentration, and the energy between different atomic ordering and concentrations should be comparable. In this case, the program will set `disable_alloy_inter = false`, `disable_strain_alloy = false`, `include_alloy_nonlocal_first_order = true`, and `include_elastic_first_order = true`.

Warning

When `alloy_model_level` is given with a value other than `"NotSet"`, the keys mentioned above will be overwritten as described, even if they are otherwise specified. Such overwriting is *not* reflected in the output `out_interact_config.toml` file.

5.4.3 Bounding-only interactions

Note

This part is available only as of version 0.19 (Oct. 23, 2025).

This part is provided mainly for closed-bounding procedure (see *Closed bounding procedure*). In the closed-bounding procedure, the bounding SATs will be added to the effective Hamiltonian when it is needed. These bounding SATs may have high order (up to 6th, 8th or even higher order). In closed-bounding procedure, these added SATs must be generated in advance, through the interaction generation introduced in this section. During this procedure, it is possible (and also strongly recommended) to specify, **the INTs with order higher than certain threshold are only provided for bounding purpose**. The INTs satisfying this condition is called bounding-only INT, and all SATs belonging to such INTs are considered as bounding-only SATs. The *bounding-only* property has two important consequences:

1. If an INT is a bounding-only INT, only SATs with all-even-order COMPs are kept. Here, SAT with all-even-order COMPs is defined as the SATs with at least one PROD containing only even-order COMPs (i.e., no odd-order COMP). For example, u^2v^2 is an all-even PROD, and u^2v^3 is not. This can greatly reduce the number of high-order SATs generated.
2. If an SAT is a bounding-only SAT, it will be disabled (by default) during the ML-based construction (MLCons) (see *Machine-learning-based model construction (MLCons)*) procedure. This guarantees such bounding-only SATs are only used in the bounding procedure.

Note

In current implementation, the OTF fitting and merge-otf fitting does not consider the bounding-only property. The bounding-only SATs are (by default) used in OTF fitting and merge-otf fittings. Thus, we recommend to generate these high-order bounding-only SATs only after OTF fitting, and use the data from DFT with merge-otf fitting, followed by MLCons and closed-bounding procedure.

By the definition of controlling SATs (see *Relations between SATs*), only INTs *without* alloy COMPs are considered as controlling SATs. Thus, the bounding-only INTs could have only the following types: MM, ME, ON, and EL.

In practice, there are some configuration fields in the `interact_config.bound_order` table. Each configuration field accepts an integer, specifying the maximal order of specified ITYP (or some part of the ITYP) that is *not* considered as bounding-only. The default value 0 specifies unlimited (no INT will be determined as bounding-only using this criterion). The configuration keys includes:

- `mm_order`: Maximal order of mode-inter (MM) interaction, corresponding to `mm_order_sum`.
- `me_strain_order`: Maximal order of strain part in mode-strain (ME) interaction.
- `me_mode_order`: Maximal order of mode part in mode-strain (ME) interaction. Corresponding to `me_mode_order_sum`.
- `elastic_order`: Maximal order of elastic (EL) interaction.
- `onsite_order`: Maximal order of onsite (ON) interaction.

For example, if the following configuration is specified:

```
[interact_config.bound_orders]
mm_order = 3
me_strain_order = 1
me_mode_order = 2
elastic_order = 2
onsite_order = 4
```

Then, the SAT containing the PROD $u_1^2(Ba1,[0,0,0])u_1^2(Ba1,[0,0,1])$ is a bounding-only SAT, because its order 4 exceeds `mm_order = 3`. The SAT containing PROD $u_1^4(Ba1,[0,0,0])$ is not a bounding-only SAT, because its order 4 is equal to (does not exceed) `onsite_order = 4`.

5.4.4 Defining the cutoffs

As we already known, the effective Hamiltonian is built up based on the Taylor expansion of the potential energy surface within certain cutoff. The most important task for the (local) interaction generation step is to define such cutoffs. Most of the keys in the `interact_config.toml` file are just provided for this task.

In general, the cutoffs are defined for each ITYP (the seven interaction types, i.e., MM, MA, etc). For a certain ICAT, *several* cutoff conditions may be defined. If an SAT satisfies *any one* of these cutoff conditions, it is a valid SAT. If no cutoff conditions defined for the ICAT, the global cutoff condition for this ITYP is applied.

The cutoff conditions may be defined for each ITYP using the `<ITYP_ABBR>_specs` arrays, with table elements of different format for each ITYP. The global cutoff conditions are defined using the keys with names starting with `<ITYP_ABBR>`. Here, `<ITYP_ABBR>` is the abbreviation of the ITYP name, in lowercase, for example, `mm`, `ma`, `me`, etc. Each ITYP cutoff specification (element in the `<ITYP_ABBR>_specs` array) must first select the controlled ICAT, and then define the cutoff conditions. The selection of ICAT is done by specifying the site names, or mixed site names, etc. The cutoff conditions are optional. If not defined, the global cutoff condition setting for corresponding key is used.

For the cutoff tables, the `site_names` and `mixed_site_names` keys (if any) are mandatory. Such fields select the ICAT under control by the current cutoff table. These fields should be filled with site names or mixed site names (as configured in the unit cell metadata files), with exactly the same name (they are matched by simple string comparison) or wildcards. Here, the wildcards is * meaning that all possible sites (mixed sites) are matched. The wildcard is only available as of version 0.16 (Oct. 12, 2025). It is important to know that the cutoff tables do not generate the ICATs, they only control the ICATs that already generated. If the provided cutoff table does not match any generated ICAT, it will be ignored.

The cutoff conditions mainly include distance and order. The distance specifies the maximal possible distance between the local COMPs participating each PROD in the desired interactions, for example, the distance between the local mode COMPs in MM PRODs. The order defines the maximal power COMPs for each PROD, and may be specified for each site, each type of LWF, or the total summed power (for the PROD). For alloy occupation COMPs, the order defines the maximal number of COMPs that may appear in each PROD.

In the following subsections, we will introduce the definition of cutoffs for each of the ITYPs. The possible configuration keys could also be seen in the `out_interact_config.toml` file generated by the program after running `heff_gen_interact`.

MM cutoffs

> **Note**
>
> Currently, the MM interactions are limited to interactions between at most two sites.

The mode intersite interaction or MM interaction has the following *global* cutoff conditions:

- `mm_distance`: maximal distances between the COMPs in each PROD, in Angstrom. Default: 5.0
- `mm_order_sum`: maximal total order (power) of each term. Default: 4
- `mm_order_site`: maximal total order (power) for COMPs at the same site. Default: 4
- `mm_order_single`: maximal order (power) for one single mode (LWF) at a single site. Default: 4

By default, the three order conditions must be satisfied at the same time. In other words, the PRODs with *any* of the three conditions violated are excluded.

Each MM cutoff specification (in `mm_specs`) includes the following configurations:

- `site_names`: Mandatory. Select the site names for the ICAT to be controlled. Array of strings (currently, only two elements).
- `distance`: Overwrites `mm_distance`.
- `order_sum`: Overwrites `mm_order_sum`.
- `order_single`: Overwrites `mm_order_single`.
- `site_orders`: Array of integers, specifying the maximal order for each of the sites. Must have same length as `site_names`.
- `lwf_orders`: Table, with LWF name as key, maximal order for that LWF as value. Defines the maximal order for each specified LWF. For LWFs not given, the `order_single` is used.

ME cutoffs

> **Note**
>
> Currently, the ME interactions are limited to strain interactions with local modes at up to *two* sites. Meanwhile, the ME interactions with local modes at two sites are experimental.

The mode-strain interaction or ME interaction has the following *global* cutoff conditions:

- `me_max_site_count`: Maximal number of sites in one ME PROD. Options are 1 and 2. Default: 1. Note that the support of 2 is experimental.
- `me_distance`: The maximal distance between two local mode COMPs in one PROD. Not used when `me_max_site_count = 1`. Default: 5.0
- `me_strain_order`: Maximal strain order (power) in each PROD. Default: 1
- `me_mode_order_sum`: Maximal local mode order (summation over all included sites). Default: 2
- `me_mode_order_site`: Maximal local mode order for each site. Default: 2

- `me_mode_order_single`: Maximal local mode order for each type of LWF. Default: 2

Note: all the order conditions must be satisfied at the same time for a given PROD.

Each ME cutoff specification (in `me_specs`) includes the following configurations:

- `site_names`: Mandatory. Specify the site names for the ICAT to be controlled. Array of strings (currently, only one element).
- `strain_order`: Overwrites `me_strain_order`.
- `mode_order_sum`: Overwrites `me_mode_order_sum`.
- `mode_order_single`: Overwrites `me_mode_order_single`.
- `distance`: Overwrites `me_distance`.
- `site_orders`: Array of integers, specifying the maximal order for each of the sites. Must have same length as `site_names`.
- `lwf_orders`: Table, with LWF name as key, maximal order for that LWF as value. Defines the maximal order for each specified LWF. For LWFs not given, the `order_single` is used.

MA cutoffs

> **Note**
>
> Currently, the MA interactions are limited to only one local mode COMP and up to two occupation COMPs in a single PROD.

In each of the MA interaction, the local mode COMP (its number is limited to one in current version) is considered as the "center". The distances of the occupation COMPs are measured with respect to such center.

The mode-alloy interaction or MA interaction has the following *global* cutoff conditions:

- `ma_distance`: Maximal distance between the occupation COMP and the center local mode COMP in Angstrom. Default: 5
- `ma_mode_order_site`: Maximal order (power) of local mode at the center site. Default: 1
- `ma_mode_order_single`: Maximal order (power) of the local mode for each single LWF type. Default: 3
- `ma_alloy_order`: Maximal number of occupation COMPs in one PROD. Default: 2

Note: all the three order conditions must be satisfied at the same time for any valid PRODs.

Each MA cutoff specification (in `ma_specs`) includes the following configurations:

- `site_names`: The site names for the local mode COMP. Currently, limited to array with only one element. Mandatory, for selecting the ICAT.
- `mixed_site_names`: The mixed site names for the occupation COMPs. Mandatory, for selecting the ICAT along with `site_names`.
- `distance`: Overwrites `ma_distance`.
- `mode_order_single`: Overwrites `ma_mode_order_single`.
- `site_orders`: Array of integers, specifying the maximal order for each of the sites. Must have same length as `site_names`.
- `lwf_orders`: Table, with LWF name as key, maximal order for that LWF as value. Defines the maximal order for each specified LWF. For LWFs not given, the `order_single` is used.

ON cutoffs

The onsite (ON) interaction has the following *global* cutoff conditions:

- `onsite_order`: The maximal order (power) of the local mode COMPs in one PROD. Default: 4
- `onsite_order_single`: The maximal order (power) of a single LWF in one PROD. Default: 4

Note: the two order requirements must be satisfied at the same time for all valid PRODs.

Each ON cutoff specification (in `onsite_specs`) includes the following configurations:

- `site_name`: The site name for the local mode, selecting the ICAT under control. Mandatory.
- `order_sum`: Overwrites `onsite_order`.
- `order_single`: Overwrites `onsite_order_single`.
- `lwf_orders`: Table, with LWF name as key, maximal order for that LWF as value. Defines the maximal order for each specified LWF. For LWFs not given, the `order_single` is used.

AA cutoffs

> **Note**
>
> Currently, the Alloy-intersite (AA) interactions are limited to PRODs with at most two occupation COMPs in each PROD.

The alloy-intersite or AA interaction has the following *global* cutoff conditions:

- `aa_distance`: The maximal distance between the occupation COMPs in Angstrom. Default: 5
- `aa_alloy_order`: The maximal number of occupation COMPs in one PROD. Default: 2

Each AA cutoff specification (in `aa_specs`) includes the following configurations:

- `mixed_site_names`: The names of the mixed sites for the ICAT under control. Mandatory.
- `distance`: Overwrites `aa_distance`.

> **Note**
>
> The `disable_alloy_inter` and `include_alloy_nonlocal_first_order` keys also control the generation of AA interaction. See *Alloy model level*.

EA cutoffs

> **Note**
>
> Currently, the strain-alloy or EA interactions are limited to PRODs with at most two occupation COMPs in each PROD.

The strain-alloy interaction or EA interaction has the following *global* cutoff conditions:

- `ea_distance`: Maximal distance between the occupation COMPs included in one EA PROD, in Angstrom. Default: 5
- `ea_strain_order`: Maximal order (power) for the strain COMPs in each PROD. Default: 1

- `ea_alloy_order`: Maximal number of occupation COMPs in each PROD. Default: 2

Each EA cutoff specification (in `ea_specs`) includes the following configurations:

- `mixed_site_names`: The names of the mixed sites for the ICAT under control. Mandatory.
- `distance`: Overwrites `ea_distance`.
- `strain_order`: Overwrites `ea_strain_order`.

Note

The `disable_strain_alloy` and `include_elastic_first_order` keys also control the generation of EA interaction. See *Alloy model level*.

ES cutoffs

Available as of version 0.15 (Oct. 7, 2025).

Note

Currently, the strain-spring or ES interactions are limited to PRODs with at most mode COMPs in one site and alloy COMPs in two sites.

The strain-spring interaction or ES interaction has the following *global* cutoff conditions:

- `enable_strain_spring`: Global switch for this type. Default: false (ES interactions are disabled).
- `es_distance`: Maximal distance (Angstrom) between the alloy COMPs and center mode PROD. Default: 5.0
- `es_strain_order`: Maximal strain order (power) in each PROD. Default: 1
- `es_alloy_order`: Maximal number of alloy COMPs in one PROD. Default: 1
- `me_mode_order_site`: Maximal local mode order for each site. Default: 2
- `me_mode_order_single`: Maximal local mode order for each type of LWF. Default: 3

Note: all the order conditions must be satisfied at the same time for a given PROD.

Each ES cutoff specification (in `es_specs`) includes the following configurations:

- `site_names`: Mandatory. Specify the site names for the ICAT to be controlled. Array of strings (currently, only one element).
- `mixed_site_names`: Mandatory. Specify the mixed site names for the ICAT to be controlled. Array of strings (currently, only one element).
- `strain_order`: Overwrites `es_strain_order`.
- `distance`: Overwrites `es_distance`.
- `mode_order_single`: Overwrites `es_mode_order_single`.
- `site_orders`: Array of integers, specifying the maximal order for each of the sites. Must have same length as `site_names`.
- `lwf_orders`: Table, with LWF name as key, maximal order for that LWF as value. Defines the maximal order for each specified LWF. For LWFs not given, the `order_single` is used.

EL cutoffs

The elastic (EL) interaction has the following *global* cutoff conditions:

- `elastic_order`: Maximal order (power) of the strain COMP in one PROD. Default: 2

There is distinct cutoff specification for EL interaction.

> **Note**
>
> The `include_elastic_first_order` key also controls the generation of EL interaction. See *Alloy model level*.

5.4.5 Interaction generation outputs

When everything is prepared, run the interaction generation with the following command:

```
heff_gen_interact
```

Note that this is a parallel program with OpenMP support, so it is strongly recommended to set the `OMP_NUM_THREADS` environment variable properly. The program should run very fast, taking typically only several seconds for simple models.

The output from this program includes:

- `heff_interactions.json`: The interaction data file, used in later parameter fitting and simulations.
- `heff_params_example.yaml`: The example or template for the parameter file. It has the same format as the parameter file `heff_params.yaml` that is used later. But the file itself contains no actual parameter information (they will be fitted later).
- `heff_interactions_topology.yaml`: The topology relation (controlling and perturbation relations) between SATs. This is useful for boundedness checking and bounding procedures. Available as of version 0.18 (Oct. 16, 2025); previously named `interactions_topology.yaml` (before version 0.20).
- `interactions_report.tex`: The details for the interactions generated by the program. It is in LaTeX source file format, and is able to be compiled to PDF documents through, e.g., `pdflatex` command from TexLive. The file could be very large, usually the compiled PDF varies from several tens of pages to thousands of pages, depending on the complexity of the model.
- `interactions_brief.tex`: The brief output information for the interactions, including the interaction/SAT indexes, their first PROD information, and their controlling/perturbation relation, etc. This is available as of version 0.19 (Oct. 23, 2025).

Format of the LaTeX report file

To generate the PDF file from LaTeX report file written by this program, use the command like the following (here, `pdflatex` from TexLive is taken as an example)

```
pdflatex interactions_report.tex
pdflatex interactions_report.tex
```

The command should be executed for at least two times to ensure the correct generation of table-of-contents.

The report file is ordered in hierarchical format. Each top section is dedicated to an ICAT (interaction category), with each subsection dedicated to an INT. The INTs are indexed with global indexes (starting from 0) and shown in the subsection title. The information for each SGRP (SAT group) is given in one sub-subsection. Each sub-subsection are further divided into two parts. The first, heading with **Original symmetry-adapted SATs**, shows the SATs generated by the program, with symmetry-adapted form, each with a paragraph leading with **SAT**. These SATs are independent in the colinear level, but some of them may still be linearly dependent on others if the unit cell basis is not orthogonal.

Next, the part heading with **Linearly independent SATs** shows the linearly independent SATs, each with a paragraph leading with **AxisSAT**. Note that for orthogonal unit cell basis, the number of SATs shown in these two parts should be the same. The **AxisSAT**s are the SATs projected on the LWF axes (see below for more information), and are SATs used in energy computation, later in parameter fitting and simulations. Each Axis-SAT is assigned with an index (numbered under the same INT) and a local index (numbered under the same SGRP).

For each paragraph for SAT, the PRODs are given in an ordered list, with each item represents a PROD. Each PROD includes a sign (positive or negative), an optional scalar factor, and several COMPs, for example:

$$+v_3(Ba1;[0,0,0])\ v_3(Ba1;[0,0,1])$$

is a PROD.

The COMPs, as introduced above, may be one of the three types: local mode COMP, strain COMP, or occupation COMP.

The local mode COMP may be expressed in two different forms. In both forms, the following necessary information is provided: the LWF name, the direction of the displacement, the center site name, and the unit cell location that includes the center site. The first form, with the direction of the COMP along one of the axes of the LWF (i.e., the displacement includes only the movement described by one LWF axis, but not others), sometimes called *axis COMP*, is marked with subscript LWF axis index (the axis index starts from 1). For example, $v_3(Ba1;[0,0,0])$ represents the local mode with LWF named v, displacing along its third axis, centering on the site called $Ba1$, belonging to the unit cell at $[0,0,0]$. The location of the unit cell is given in the fractional coordinate of the unit cell. Due to the translation symmetry, adding an arbitrary integer vector on *all* the COMPs in the PROD would yield a PROD that is exactly equivalent to the original one. The second form, with the direction of the COMP along a general direction (not along any of the LWF axes), sometimes called *lean COMP*, is marked with the (normalized) displacement direction. For example, $v[-0,-0,0.5,-0.8660254](Al2;[-1,1,1])$ represents the local mode with LWF called v, displacing along a linear combination of the third and fourth axes (this is a four-dimensional LWF), centering at site $Al2$ belonging to the unit cell located at $[-1,1,1]$. Such lean COMPs typically present only in models with non-orthogonal unit cell basis. For the linearly independent SATs, the Axis-SATs, by construction, they include only axis COMPs, but not lean COMPs.

The strain COMP, similar to local mode COMP, has two formats. The axis COMP form uses axis index (in Voigt index, ranging from 1 to 6) as subscript, for example, η_3. The lean COMP form marks the direction as a vector, for example $\eta[0.25,0.75,0,0,0,-0.8660254]$.

The occupation COMP is marked with mixed site name, site name and the unit cell location. For example, $A(Al2;[-1,0,0])$ represents the mixed site named A, at the site called $Al2$, belonging to the unit cell located at $[-1,0,0]$. Note that although the occupation COMP is written in a production form here, it is not computed directly as this. The occupations take part in the effective Hamiltonian by selecting the parameter. For example, in the following PROD:

$$q_1(Al1;[0,0,0])A(Al2;[-1,0,0]),$$

where q is a local mode and A is a mixed site, the parameter (for the corresponding SAT) is a *set* of values (instead of a single scalar). The actual value for the interaction during the energy evaluation depends on the actual occupation at the selected $Al2$ site. For example, if the occupation index is 1, the first parameter is selected; if the occupation index is 2, then the second parameter, etc.

5.5 Fit the parameters

The Taylor expansion coefficients, or the parameters of the effective Hamiltonian, should be fitted according to FP calculations. In this package, a state-of-the-art fitting strategy is provided, mainly based on active learning scheme. Nevertheless, there are still some parameters that must be determined *before* the performing the active-learning process. The whole parameter fitting could be divided into two steps:

1. Precalculation: calculate some specific parameters (mainly about the long-range dipolar interaction).
2. Active-learning or on-the-fly learning: compute all the other parameters using on-the-fly active learning strategy.

5.5.1 Precalculation

In the precalculation step, the following parameters should be determined from FP calculations:

- The effective masses of the local mode and strain DoFs.
- The Born effective charges (BECs) of the dipolar local modes (if any).
- The optical dielectric constant (required if there is at least one local mode with long-range dipolar interaction).

Important

All the FP calculations used for building the model should be exactly compatible (in the sense of the choice of xc functional/pseudo potential, precision, etc). Otherwise, the model building may be hard to be converged, and the resulting model may be physically unreasonable.

Some basic ideas of dipolar interactions have been introduced in *Unit cell specification file*. The calculation of optical dielectric constant has also been introduced there. The effective masses and BECs could be computed with the assistance of some CUI programs provided by this package (see below), or computed using any other methods. The only outcome is several values, to be filled in the input file later used in on-the-fly learning.

These calculated values should be written into the pre-calculated parameter file, which is an YAML file (default name is `heff_params_precalc.yaml`), for later parameter fitting. The template of this file could be obtained by copying the `heff_params_example.yaml` file generated in interaction generation step. In this file, only the first several lines (the `globals` and `lwfs` sequences) should be fitted, while others (mainly the `interactions` sequence) need not to be modified (in fact, they could even not present). The format of the `globals` and `lwfs` must be exactly the same as the `heff_params_example.yaml` file to ensure it could be properly loaded later. Here is an example of this file:

```yaml
global:
    eps_inf_inv_factor: 1
    mass_strain: [1, 1, 1, 1, 1, 1]
    mass_strain_factor: 233.19219999999999
lwfs:
-  # LWF name: u; irrep_label: T1u
    mass: 30.55784139
    charge: [10.81072456]
-  # LWF name: v; irrep_label: T1u
    mass: 107.27040745
-  # LWF name: q; irrep_label: A2u
    mass: 16.31743758
    charge: [-1.25959883]
```

Note

One may get puzzled with the two input files, `heff_params.yaml` (see *Configuring the first OTF MD run*) and `heff_params_precalc.yaml`. Actually, the former need only to be copied from the example file (`heff_params_example.yaml`) that generated along with the interaction JSON file (`heff_interactions.json`) *without any further edition*, and the latter needs only the `global` and `lwfs` parts (as shown in the example above). The reason why we need these two input files is that this makes the change of interaction set much more easily. More precisely, if we change the interaction set, the `heff_params.yaml` file need to be updated along with the interaction file, while the `heff_params_precalc.yaml` file need not to be modified (provided the LWFs and their properties are not changed).

In the following sections, we will introduce how to compute these values and fill them into proper places.

Non-alloy models

For non-alloy models (models without any mixed sites), the precalculation step is typically straightforward. This could be done for each LWF with a simple command, with unit cell specification files and BEC data file (provided for each atoms in the *unit cell* and named, by default, `BORNCHG`, see *Check selected local mode basis* for format) available at the running directory:

```
python3 -m heff_gen.lwf.lwf_bec_tensor -l <lwf_json_file>
```

Here, the `<lwf_json_file>` should be replaced by the name of the LWF data file generated earlier. Use `-h` or `--help` to see more possible arguments to this program. The possible output is:

```
Born effective charge of LWF:
[[ 1.08107246e+01 -1.71764038e-15 -2.86685055e-15]
[ 3.86075683e-15  1.08107246e+01 -4.16377540e-15]
[-1.27779459e-15  2.09267828e-15  1.08107246e+01]]

Born effective charge (symmetry-adapted) of LWF:
[10.81072456]

Effective mass of LWF (c.f. PRB.78.104104):
[30.55784139 30.55784139 30.55784139]
```

Here, the tensor form of the LWF BEC is first shown, followed by symmetry-adapted BEC values. The symmetry-adapted BEC values should be written in the `charge` field of corresponding LWF in the `lwfs` sequence of pre-calculated parameter file, if the LWF is dipolar (`dipolar` is *not* `NO` in `unitcell_spec.toml`).

The effective masses of each axis in the LWF is then presented. Typically, the mass values should be identical, because of the symmetry transformation properties between different LWF axes. The mass value (only one scalar value) should be written in the `mass` field for corresponding LWF in the `lwf` sequence.

> **Note**
>
> There may be different methods for determining the effective masses of local modes (LWFs). For the method used here, and related discussions, please refer to [ALFE-H-2025] and references therein.

The effective mass of the strain DoF is defined by the `mass_strain` and `mass_strain_factor` fields. The actual value is the multiplication of these two inputs. Note, that here we let the effective mass of strain proportional to the number of unit cells in the supercell. That is, the input values will be multiplied by the number of unit cell in the supercell.

> **Note**
>
> There seems to be no standard method for determining the effective mass of strain DoF, we are not even sure about whether such scaling with number of unit cells is a proper treatment. Pay special care if your result rely on these values.

Solid solution models

The solid solutions models are models with at least one mixed sites defined in its unit cell metadata. See *Unit cell specification file* for more information. For these solid solution models, there may be something need to be discuss in the precalculation stage. It is not difficult to understand that LWFs with different chemical environment around the center (including the center site itself) may have different properties, by definition. For example, suppose one LWF mainly contains the displacement of its center site, which is a mixed site. Suppose the mixed site contains two different

elements that are rather different from each other, with very large atomic mass difference. It is then clear that, by definition, the center occupied with different elements could have different effective masses for the LWF.

To take this difference into consideration, occupation-dependent (occ-dep) parameters may be applied to these LWFs. Currently, this is only limited to the center site, that is, the different occupation at the center site (but not other sites) could result in different pre-calculated parameters for the local mode.

If the center site of one LWF is a mixed site, the automatically generated YAML parameter example file would contain more information than normal, these information should also be used in the pre-calculated parameter file. Here is an example of input file (only LWF part), where the LWF named `u` is centered on a mixed site containing two different elements. If the occ-dep mass is desired (i.e., one would like local mode centering on different type of elements have different mass values), the `occ_dep_mass` should be set to `true`, and the mass values for each type of the center occupation should be given in `mass_alloy` field (which must have the length equal to the number of possible element types of that mixed site). In this case, the `mass` field is optional and useless. Similarly, if the occ-dep BECs are desired, set `occ_dep_charge` to `true`, and fill the `charge_alloy` with (symmetry-adapted) BECs for different center occupation element (each row for one occupation possibility). The `charge` field is optional and useless.

```
lwfs:
-  # LWF name: u; irrep_label: T1u
    occ_dep_mass: true
    mass_alloy: [89.10816709022129, 62.45705001276345]
    mass: 30.64
    occ_dep_charge: true
    charge_alloy:
    - [9.29669393]
    - [8.05900256]
    charge: [9.96870784]
-  # LWF name: v; irrep_label: T1u
    mass: 110.24758646
-  # LWF name: q; irrep_label: A2u
    mass: 16.61107005
    charge: [-0.87170103]
```

Here we provide a possible approach for the precalculation of alloy models. First prepare a supercell with some atomic arrangement of the mixed sites (must include all possible elements). Do first principle calculations on such supercell and obtain the `BORNCHG` file (see *Check selected local mode basis* for the format of this file). Then, use the following command in a directory containing the unit cell metadata, the *supercell* structure and the BEC file for the *supercell*:

```
python3 -m heff_gen.cell.lwf_basic_params <supercell_filename>
```

Remember to replace `<supercell_filename>` by actual supercell structure filename. The standard output will contain some information about occupation-dependent or occupation-independent BECs and masses, for example:

```
INFO:root:LWF u  simple averaged BECs:
[[ 9.57911746e+00 -1.54119982e-15 -2.53128037e-15]
[ 3.41610469e-15  9.57649878e+00 -3.84501098e-15]
[-1.10171615e-15  1.84125228e-15  9.61339636e+00]]
INFO:root:LWF u  simple averaged masses: [44.3563008 44.3563008 44.3563008]  44.
↪35630079766008
WARNING:root:extract_lwf_born_charges: large projection error occurred 0.
↪02372549820575287
INFO:root:LWF u  symmetry-adapted BEC values: [8.58967087]
WARNING:root:extract_lwf_born_charges_site: large projection error occurred 0.
↪47866692829969715
WARNING:root:extract_lwf_born_charges_site: large projection error occurred 1.
```

(continues on next page)

(continued from previous page)

```
↪5286827198943254
INFO:root:LWF u  occupation-resolved averages:
INFO:root:occ 1  BEC [9.29669393]  mass [89.145399   89.14558259 89.03351969] 89.
↪10816709022129
INFO:root:occ 2  BEC [8.05900256]  mass [62.54878669 62.48020248 62.34216088] 62.
↪45705001276345
INFO:root:LWF v  simple averaged BECs:
[[ 3.40525737e-06 -2.25167107e-15  7.25114413e-16]
 [ 2.15799600e-15  1.70262869e-06  2.76081241e-15]
 [-4.45476989e-15 -1.61849700e-15 -5.10788602e-06]]
INFO:root:LWF v  simple averaged masses: [110.24758646 110.24758646 110.24758646]  110.
↪24758645803537
WARNING:root:extract_lwf_born_charges: large projection error occurred 5.
↪107886035460423e-06
INFO:root:LWF v  symmetry-adapted BEC values: [1.37314928e-14]
```

This program just simply compute the BECs and masses for each center site (i.e., center sites in different unit cells of the supercell), and compute average values to obtain the overall result. For local modes with mixed center site, `u` mode above, the output contain the "simple average" that disregard the center site occupation (averaged over all center sites), and the "occupation-resolved" average that take averages for different type of center site atom. In the example above, the simple average BEC for `u` is `[8.58967087]`, and occupation-resolved one is `[9.29669393]` and `[8.05900256]` for different center occupation possibilities. The average masses are provided similarly. For local mode with non-mixed center site, only simple average values are presented.

It is up to the users' decision to select whether occupation-dependent values should be adapted or not.

> **Note**
>
> The WARNINGs in above example comes from the projection of BECs on the symmetry-adapted basis (the `polar_directions`, see above). These warnings are quite normal for alloy models, because the local symmetry *is* broken due to different occupations.

5.5.2 On-the-fly learning

Here we introduce the on-the-fly (OTF) learning, or active learning step, for obtaining the parameters of the effective Hamiltonian. The term "on-the-fly" means the data for the learning task is (mostly) not prepared in advance, but is found during the learning process itself. Briefly, the learning process is performed with a period of molecular dynamics (MD) simulation mainly using the effective Hamiltonian, and FP calculations are triggered when the predictions of energy, force, and stress by the effective Hamiltonian are considered unreliable. Typically, the FP calculations are called only a few times during the long MD simulation stage (typically, less than 1% of the MD steps). For more details of the OTF learning, please refer to our article [ALFE-H-2025]. In this document, we focus mainly on technical and practical aspects.

Each OTF MD run is conducted on a fixed supercell size under (almost) fixed physical conditions. Typically, this OTF MD simulation should be conducted for several times, with different supercell size (and shape) and different physical conditions, to explore the potential energy surface sufficiently and efficiently. To make this process efficient, we recommend to start the first OTF MD run with small supercell size and at low temperature, then increase the supercell size and temperature gradually in each OTF MD run, and add other external fields (electric field, pressure, etc) if required.

In this document, we introduce some basic configuration of OTF fitting. For some more advanced (and more detailed) configurations, please refer to *Detailed configurations of OTF fittings*.

Configuring the first OTF MD run

The first OTF MD run starts with all the parameters (except those from precalculation) unknown. Typically the MD simulation starts from reference structure, with small supercell, and low temperature. To run the first OTF MD fitting, prepare the following files:

- Unit cell specification files (`unitcell_spec.toml`, unit cell atomic structure file, LWF data and symmetry files);
- Interaction file (`heff_interactions.json`), obtained from *Generate interactions* step.
- Parameter template file (`heff_params.yaml`), copied from the `heff_params_example.yaml` obtained from *Generate interactions* step.
- Pre-calculated parameter file (`heff_params_precalc.yaml`), which should have been filled with pre-calculated parameters, see *Precalculation*.
- Fitting configuration file, the default filename is `fit_config.toml`.

The main input file that controls the whole fitting procedure, typically edited manually, is the fitting configuration file `fit_config.toml`. Here, we provide an example of the full input set, which could be found in the `otf-fit/BaTiO3/work-20` directory of [Examples].

Here is an example of the `fit_config.toml` file:

```
[cell]
supercell_matrix = [
    [1, 1, 0],
    [-1, 1, 0],
    [0, 0, 2]
]

[simulation]
sim_steps = 200000

[physical]
temperature = 30

[simulation.md]
ensemble = "NPT"
adjust_temp_begin = true

[regression]
method = "BayesianLinear"
variable_ref_energy = "No"

[bayesian]
inf_sigw = false

[active]
fp_distance = 20
thres_update_scale = 0.99
thres_average_count = 40
fp_criterion = "energy"

[otf]
restart_interval = 20000
```

(continues on next page)

(continued from previous page)

```
[fp]
# see: https://wiki.fysik.dtu.dk/ase/ase/calculators/vasp.html#module-ase.calculators.
↪vasp
calculator = "Vasp"

[fp.kwargs]
#command = "mpirun /data/home/mxy/vasp-build/vasp6.3.2/vasp.6.3.2-cpu/bin/vasp_std"
xc = "pbe"
gga = "PS"
encut = 550
ediff = 1e-6
ismear = 0
sigma = 0.05
isif = 2
prec = "Accuracy"
lreal = "Auto"
kspacing = 0.3
symprec = 1e-8

npar = 4
kpar = 2

[fp.kwargs.setups]
Ba = "_sv"
Ti = "_sv"
O = ""
```

Here, the `cell`, `simulation` and `physical` tables are exactly the same as those used in simulation (see *Running Simulations*). Similarly, the `init` table could also be defined (though omitted here). These tables mainly define the condition, configuration and external environments of the MD simulation. The supercell here contains 20 atoms (the unit cell is five-atom perovskite unit cell).

The `regression` table defines the regression method `BayesianLinear`, which selects the Bayesian linear regression (BLR) method, as described in [ALFE-H-2025]. The `variable_ref_energy` key defines whether the reference energy in the fitting is variable, accepts three possible values: `"Yes"`, `"No"`, and `"Auto"` (the default). Here the "reference energy" is the energy value to be *removed* from the potential energy calculated from FP, in order to avoid too large energy values. The reference energy is collected at the first step with reference structure. Here we selected `"No"`, indicating such reference energy is never updated in the simulation. For more detailed information, please refer to *Reference energy*.

The `bayesian` table defines some details for the BLR algorithm. Here, the `inf_sigw` key specifies whether the σ_w hyperparameter is to be estimated during the fitting process using the standard evidence approximation (with `inf_sigw = false`), or it is assumed to approach infinity. See [ALFE-H-2025] and references therein for more information.

The `active` table defines some details for the active-learning-based methods. Here, `fp_distance` defines the minimal MD steps between two FP calculations if the uncertainty does not exceed the "high threshold" (otherwise FP is executed immediately). The "high threshold" is defined by `high_thres_scale` times the "threshold" σ_t, with the latter being updated dynamically throughout the fitting process, based on historical observations.

The `fp_criterion` defines which type of value is used as the uncertainty for determining FP calculations (also as the threshold). Here we use `"energy"`, meaning that the energy uncertainty per unit cell is used.

The keys `thres_update_scale` and `thres_average_count` controls the threshold for determining FP calculations. See *Controlling FP threshold* for more information.

The `otf` table defines details for on-the-fly (OTF) fitting. It mainly controls the details for the OTF MD simulation that do not apply to ordinary MD simulation. Here, `restart_interval = 20000` specifies the MD simulation is "restarted" (from reference structure) every 20000 steps (if set to 0, no such restarting). Similarly, `reset_momenta_interval` configures the frequency for resetting the momenta (but not the values of the DoFs); `reset_occupations_interval` defines the frequency for resetting the atomic occupations for mixed sites (which is meaningless for non-alloy models).

Note

In some of the literatures (even in some part of this document), the terms "active learning" and "on-the-fly learning" are used in a mixed way. Here we note that to our understanding, and at least in our code, the "on-the-fly learning" scheme is a special case of "active learning", where (almost) all the data used is generated in the fitting process itself. The general "active learning", in contrast, may just *select* data from a given range (though it may be very large).

The first-principles (FP) calculation is controlled by the `fp` table. A calculator must be provided using the `calculator` key. In this package, we use the ASE package to execute the FP calculation runs. Any calculator with a ASE `Calculator` interface supported, provided in any available Python package, is supported. The *Python class name* for the ASE calculator should be filled in the `calculator` field. For example, for Vasp calculation, the ASE calculator class is (provided internally in ASE package) `ase.calculators.vasp.Vasp`, thus `calculator = "Vasp"` (instead of `"vasp"`, note the input file is case sensitive) should be used. The module that contain the calculator class should be provided using the `module` key, except for calculators defined in submodule of `ase.calculators` package, this `module` field could be omitted.

Important

The FP calculation configuration must be exactly compatible with the calculations for optimizing the reference structure, computing phonon dispersion, and precalculation. The inconsistency of the FP configurations may lead to slow convergence and physically unreasonable results.

Important

The FP calculation configuration provided here must specify a self-consistent field (SCF) single-point calculation (no movement of ionic positions and lattice vectors) with force and stress calculated. Otherwise, the behavior is undefined.

Note that we here also provided some extra features for some specified calculators. For example, for the `Vasp` calculator, we have implemented a special treatment of the wave function file `WAVECAR`. For calculations that not too far away from the previous calculation, the `WAVECAR` is kept (if written by the previous run) since it may help to accelerate the convergence. If the steps between current calculation and last calculation exceeds certain value (`fp.remove_wave_function_steps`), the wave function file `WAVECAR` is removed before running FP calculation. For more information about the FP calculators, see *Customize the FP calculators*.

The arguments passed to the calculator should be defined in `fp.kwargs` table, and possibly its sub-tables. All these arguments are passed to the ASE calculator class constructor (`__init__` method). For available keyword arguments, please refer to ASE calculator document and links therein. Note that some calculators may require the user to set some environment variables, for example, the `ASE_VASP_COMMAND` environment variable for `Vasp` calculator.

Tip

There are some configurations provided only for development purposes. We do *not* recommend you to adjust the configurations not mentioned by this document (including this page and other pages) if you do not understand exactly what you are doing.

Running the first OTF MD fitting

Once all the input files are prepared, the OTF fitting could be triggered by the following command:

```
python3 -m heff_gen.fit.cui.otf_fit
```

Warning

Only one active fitting process could be run in one working directory at the same time. *Do not* try to run another fitting process at the same directory before the previous process at the same working directory ends. Otherwise serious data conflict may be caused.

This OTF fitting process typically takes long time, because FP calculations are triggered when and where desired, and the process cannot be continued util the FP calculation finishes. For high performance computing (HPC) clusters, it may be necessary to submit this job to the work load manager system (like PBS, Slurm, LSF, etc), like any other FP calculation tasks. The allocation of resources for these OTF fitting jobs should take the FP calculation into account. Here is an example submission script for the Slurm working load manager system:

```
#!/usr/bin/sh
#SBATCH -p cpu
#SBATCH -J gOTF_BTO
#SBATCH -N 1
#SBATCH -n 32


# Load the environment variables, modules, according to your system
module load gcc/14.1.0
module load compiler mpi mkl
source ~/util/init_general_Heff_gcc14.sh

# The ASE_VASP_COMMAND environment variable required by ASE Vasp calculator. Set␣
↪according to your system path.
export ASE_VASP_COMMAND="mpirun  /data/home/vasp/bin/vasp_std"

# Optional but recommended: show the version information in output
heff_build_info

# Run the OTF fitting
python3 -m heff_gen.fit.cui.otf_fit
```

Once the fitting begins, some information is presented in the standard output (see screen, or output file of the job created by the work load manager system). These outputs on standard output present the most basis information about the fitting process, for example the current step, whether FP is called, etc. Here is a part of possible outputs:

```
INFO:root:TRAINING STEP 125
INFO:root:Uncertainty at step 125 0.0017684489027820175
INFO:root:Skipping FP for step 125
```

(continues on next page)

(continued from previous page)

```
[2025-05-08 10:54:47.906] [info] MD STEP    125, Htot=0.263458, Hpot=0.105112, Hkin=0.
↪158346, T=400.006662
INFO:root:TRAINING STEP 126
INFO:root:Uncertainty at step 126 0.0017760928127237966
INFO:root:Uncertainty threshold updated to 0.002832397691702434 with std_err=0.
↪0004767650520880307 5 ratio=0.16664234791246876
INFO:root:Skipping FP for step 126
[2025-05-08 10:54:47.919] [info] MD STEP    126, Htot=0.264325, Hpot=0.105979, Hkin=0.
↪158346, T=400.006662
```

The FP calculations are performed under `fp_dir` sub-directory. Typically, all the input files needed for the FP calculation is generated by the program. The output files, including the redirected standard output, are also stored in this directory, which may show the calculation progress. After each FP calculation, the data from FP is stored in the checkpoint file (using ASE `Checkpoint`). For next OTF MD run at the same directory (if any), the checkpoint file of the same index would be checked before FP calculation, and the data is loaded from the FP calculation (and actual FP calculation skipped) if the checkpoint (including structure and configurations) matches the FP calculation to be performed. The stored checkpoint files may also be used later in other fitting processes, for example, reuse data fitting.

Besides, some output files are also generated. These files may be used in later analysis, or continuing fitting (see below). The final fitted parameters (updated each time after the parameters are fitted) are given in the `heff_params_fit.yaml` file, which could be used for thermodynamics simulation (see *Running Simulations*).

During the fitting process, the status could be modified through the output files. We provide a convenient plotting tool for this purpose. To use it, typing the following command under the working directory:

```
python3 -m heff_gen.fit.plot.otf_plot .
```

Here is an example of the figure obtained from such program:

Here, the top left panel shows the uncertainty evaluated for each step; the two dashed lines show the uncertainly threshold and the high threshold, which are updated during the fitting process; the filled circles indicate the steps where FP calculations are called and parameters are fitted; the empty circles (not presented in this example) indicate the steps where FP calculations are called but parameters not updated (one will encounter this case if `fit_distance` is larger than 0). The left middle panel shows the maximal (blue) and root-mean-square (orange) force error in each step where FP is called. These force errors are obtained between the FP data and the prediction from the effective Hamiltonian *before* FP. The bottom left panel shows the temperature during the fitting. The top right panel shows the potential energy predicted by the effective Hamiltonian (blue line) and from FP calculations (empty circles). The middle right and bottom right panels shows the polarization and strain evolution during the fitting process, respectively.

After the OTF MD run finishes, one could optionally (but recommended) to get the potential energy of each step using the final set of parameters (obtained in this fitting run) by running

```
heff_dof_potential
```

The result is stored in the `pot.txt` file. This "final potential" data could be plotted along with other data by adding `-p` argument to the `otf_plot` program:

```
python3 -m heff_gen.fit.plot.otf_plot . -p
```

The final potential data would be shown in another line (typically green) on the top right panel, along with the potential obtained in the fitting and from FP.

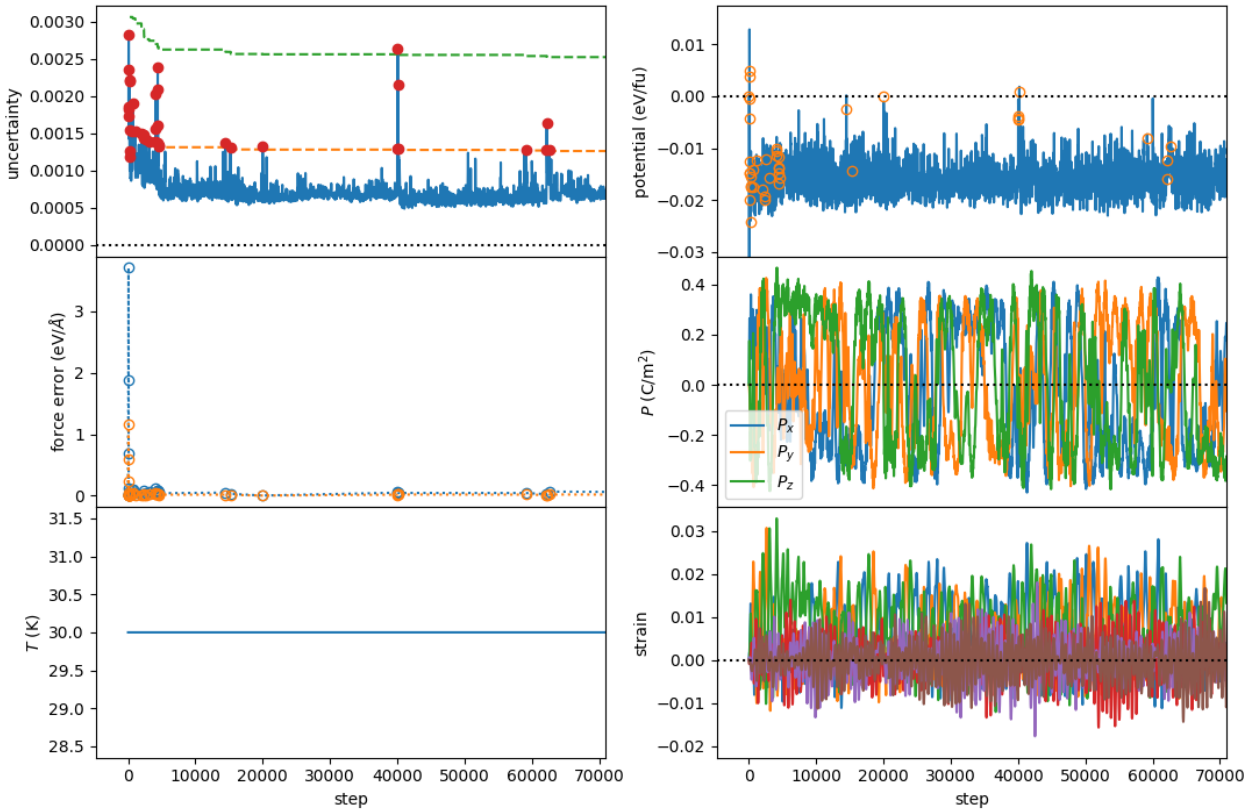


Fig. 4: OTF-plot example

Continuing the OTF fitting

Warning

The metadata of the model [including the unit cell metadata files, interaction file (`heff_interactions.json`) and parameter file (`heff_params.yaml`)] must be compatible with those provided in the first OTF MD run throughout the whole fitting process, as well as in later simulation runs. Otherwise the behavior of the program is undefined. Note that some possible inconsistency may be detected by the program, but not all. The program does *not* guarantee these inconsistency are found and reported. It is the users' responsibility to ensure this consistency.

Important

Except the configurations controlling the MD process (in the tables also used in simulation) and those mentioned below, we recommend that all the configurations in the continuing OTF MD run should be consistent with that of the first OTF MD run. Although changes to some of them may be allowed and make some sense, some changes may result in unpredictable results.

As introduced above, the fitting of effective Hamiltonian for a specified material typically consists of OTF MD runs on several different supercells, with possibly different conditions. Each of these OTF MD runs (except the first one) takes all the information (the data collected from FP calculations, the history of uncertainties, etc) from the previous run, and the data from FP calculations are accumulated during the whole fitting process. These OTF MD runs taking information from previous runs are called *continuing* OTF MD fitting runs.

The input file set of continuing OTF MD runs are basically the same as that of the first OTF MD run, but with some extra input files and settings. The data from previous OTF MD runs should be provided in a JSON file (by default, `active_fit_in.json`). This JSON file should typically be copied from the `active_fit_out.json` file from previous OTF MD run. Besides, the `continuing` key in `active` table must be set to `true` to trigger the continuing fitting.

Important

Remember to set `continuing = true` for all continuing OTF fitting. Otherwise the previously collected data are discarded.

For the first OTF MD run, as the supercell is typically rather small, not all the parameters could be fitted due to periodicity employed by the supercell (see [SecondPrinciples2017] for some discussions to similar problem). In fact, a large amount of parameters would have zero values in the resulted parameter file (see `heff_params_fit.yaml`). As the OTF fitting continues with large enough supercell sizes, all parameters should be able to captured by the fitting process. It is recommended to set `inf_sigw = false` at the first OTF MD run, and all OTF MD runs before all parameters could be captured. After large enough supercell is applied, it is recommended to switch to `inf_sigw = true` for better performance and for the reason discussed in [ALFE-H-2025].

The execution of continuing OTF fitting and the output files are basically the same as that of the first OTF fitting.

Interaction selection and bounding

In principle, it is possible to get a well-fitted model by simply repeating the continuing OTF fitting process on different supercells and conditions (temperature, pressure, etc). However, we practically find such process is not efficient, especially for complex models. We thus provide the Machine-learning-based construction (MLCons) process for selecting the interactions (*Machine-learning-based model construction (MLCons)*), and bounding procedures for making a bounded model (*Open bounding procedure*). For the whole practical fitting workflow, please refer to *Practical fitting workflows*.

Termination of OTF MD fitting

For some reasons you may want to terminate the OTF MD fitting before reaching the configured MD steps. For example, the FP calculations have not been called for a very large amount of steps, or the FP calculations are called too frequently, etc. To do this, we provide a way to safely terminate the OTF MD simulation while it is still running through the input file `stop.toml`.

> **Warning**
>
> We do not recommend to terminate the fitting process directly by killing the process, or cancel the task on the work load manager system, etc, because you will *not* be able to continuing the OTF fitting with all collected FP data this way, due to the continuing data file (`active_fit_out.json` by default) is not generated if the program is not terminated normally.

The `stop.toml` file accepts two possible keys, `stop` and `stop_after`. If `stop = true` is given in `stop.toml` file, the OTF fitting will be terminated safely once the program have read this file. If `stop_after` is assigned to an integer (representing the step index that the termination will be required), the OTF fitting will be terminated safely if the program have read this file, and the current step index is no less than the given integer. In either cases, the `stop.toml` file will be deleted before the program exists.

The `stop.toml` will be checked before any FP calculations performed, or with fixed interval. The interval is defined by `active.check_stop_interval` key, defining the number of MD steps between the two checks for the `stop.toml` file.

Customize the FP calculators

> **Note**
>
> This part is only valid as of Jul. 29, 2025.

The FP calculations required by the OTF fitting are mainly conducted with the ASE calculators (see *Configuring the first OTF MD run*). However, as some features required by this package are not provided by the original ASE calculators, we actually use *extended* calculators in this package. An FP calculator framework is provided by this package. The extension to some type of the FP calculators are implemented in this package. Nevertheless, the user could also provide their own extensions, from external packages. Here we introduce this FP calculator framework.

All the FP calculators actually used in this package should be inherited both from one ASE calculator and the `HeffFPCalculatorBase` class (defined in `heff_gen.fit.fp_interface.calculator_base` module). The latter provides some interfaces used by the OTF fitter and their default implementations, for example, the function for removing the wave function file, and the function for checking the consistency between two ASE calculators (using their `todict()` methods). To customize one FP calculator for OTF fitting, one should write a class inheriting from both the corresponds ASE calculator and `HefFPCalculatorBase`, for example:

```
from ase.calculators.vasp import Vasp as AseVasp
from heff_gen.fit.fp_interface.calculator_base import HeffFPCalculatorBase

class Vasp(AseVasp, HeffFPCalculatorBase):
    ...
```

A full example could be found from the source code of `heff_gen.fit.fp_interface.calculators.vasp.Vasp`.

In the OTF fitting, the FP calculator will be loaded based on the `fp.calculator` and `fp.module` configuration fields. As introduced above, the `fp.calculator` field should be set to the *class name* of the calculator class. The optional `fp.module` should be set to the module that could be imported by the running OTF fitter and containing the desired

FP calculator class (with name specified by `fp.calculator` key). The specified calculator class could be either the wrapped Heff FP calculator (that inheriting both ASE calculator and `HeffFPCalculatorBase`), or simply an ASE calculator. If the calculator class is simply an ASE calculator, the program will create a class inheriting from the specified calculator and `HeffFPCalculatorBase` dynamically.

If `fp.module` is not specified, the program will try to find the calculator from following locations, in order:

1. The wrapped Heff FP calculators, provided by this package, in `heff_gen.fit.fp_interface.calculators` package.
2. The standard ASE calculators, in `ase.calculators` package.

CHAPTER

SIX

ADVANCED TECHNIQUES IN MODEL FITTING

In the page *Building a model*, we have introduced the whole procedure for building an effective Hamiltonian model of a specified material. The operations introduced there should, in principle, be enough to obtain a model. However, sometimes we need some more advanced operations in order to adjust fitting configurations, analyze the fitting result, etc. Here in this page, we will introduce some advanced techniques for model building and analysis, mainly for the parameter fitting step.

6.1 Detailed configurations of OTF fittings

In this section, we will introduce some configuration methods for the OTF fittings, using a topic-based structure. These configurations may be useful for controlling the OTF fitting processes, but we find it too tedious to include these topics in the *Building a model* tutorial, so they are now provided in this (more advanced) document. There is generally no specified order between these topics.

6.1.1 Reference energy

It is clear that adding any constant to the effective Hamiltonian potential energy will not affect the thermodynamics properties. The potential energy value from FP typically has large magnitude comparing to the energy differences between the considered structures. For example, for perovskites, the potential energy from Vasp is typically some value around -39 eV per unit cell (containing 5 atoms), while the difference between cubic reference structure and ferroelectric structure is less than 0.1 eV per unit cell. Such large value of potential energy can cause inconvenience. Moreover, it may affect the BLR and ridge regression process. Thus, in the fitting process, we typically subtract a constant potential energy (per unit cell) from the potential energy provided by FP calculator. Such constant potential energy value is called *reference energy*. This makes the effective Hamiltonian potential energy value of the reference structure close to zero.

Typically, this reference energy is collected after the first FP calculation triggered in the first OTF MD run (this is why we recommend the first OTF MD simulation starts from reference structure). The reference energy value *may* be updated during later OTF fitting runs, controlled by the `regression.variable_ref_energy` field, which could be set to `Yes`, `No`, or `Auto`. If the reference energy is variable (`Yes` or `Auto` and judged to be variable by the program) and `collect_ref_energy` is *not* set to `No`, the reference energy is updated at the first step of each OTF MD run. Otherwise, the reference energy value is kept during the whole fitting process (even in different OTF MD runs).

This field should be set according to the model type:

- For alloy model with `alloy_model_level = "FixedConcRelative"` (see *Alloy model level*), the reference energy *must* be variable, because the reference energies of different atomic arrangements are different by definition. In this case, `variable_ref_energy` *cannot* be set to `No`.
- For alloy model with `alloy_model_level = "FixedConcAbsolute"` or `"VariableConc"`, the reference energy must be *invariable*, because the energies of different atomic arrangements must be compared under the same reference energy. In this case, `variable_ref_energy` *cannot* be set to `Yes`.

- For non-alloy models, the reference energy could be either variable or invariable. If set to `Auto`, it is equivalent to `No` in this case. For these non-alloy models, the reference energy values collected in different OTF MD runs (with possibly different supercells) should be exactly the same in principle, however there are typically some numerical differences between them.

6.1.2 The long-range dipolar interaction in fitting

The parameter fitting for long-range dipolar interaction is somewhat different from that for local interactions. The BLR fitting algorithm is only valid if the energy, force and stress are *linearly* dependent on the *parameters* (not the DoFs), which is this case for all the *local* interaction parameters. However, the parameters that control the long-range dipolar interaction, including the BECs of modes, optic dielectric tensor (defined in `unitcell_spec.toml`), have some different features. By definition, the energy and force arisen by the long-range dipolar interaction dependent on these parameters in a *nonlinear* manner (see Eqs. 7, 8 in [Zhong1995]). Moreover, for general tensor-form optic dielectric constant, its effect should be taken into account during Ewald summation, which is a quite time-consuming process for large supercell. For these reasons, these parameters for long-range dipolar interactions are *not* fitted in the OTF process, but must be given in advance to OTF fitting, by direct FP calculations.

Nevertheless, an overall factor of the dipolar interaction energy in the total potential energy, could be treated as a parameter, which has linear contribution to the energy, force and stress. Such parameter is simply a scalar multiplied to the dipolar interaction energy, and is equivalent to a factor to the reciprocal of optic dielectric tensor. Such parameter is named `eps_inf_inv_factor` in the `heff_params.yaml` file. By default, the parameter should have the value 1 if the calculated dipolar parameters above are exactly suitable for the model.

By default, the `eps_inf_inv_factor` parameter is *not* fitted during OTF fitting, and is fixed to the value 1. Fitting of the parameter could be enabled by setting `fix_eps_inf = false` in the `pader` table.

6.1.3 Selecting parameters for fitting

In addition to the `eps_inf_inv_factor` for long-range dipolar interaction, all the *local* parameters could also be chosen to be fitted or not during the OTF fitting process. By default, all the *local* parameters are fitted (their values could be changed according to the FP data).

To fix some of the parameters to their initial values (those read from `heff_params.yaml` file) during the fitting process, use the following approaches:

- Set the `fix_interactions` key in `pader` table. This key accepts an array of integers, representing the indexes of the INTs, whose parameters should be fixed during the OTF fitting. The indexes of the INTs are consistent with those provided in the interaction report file, see *Generate interactions*.
- Set the `interact_controls` key in `pader` table. This key accepts an array of tables, each includes the index of INT under control and the `fix_sats`, accepting an array of SAT indexes (under this INT) to be fixed.

For example, in the following configuration file (part), all the parameters associated with the INTs indexed 0 and 1 are fixed. In the INT indexed 4, the parameters associated with SATs indexed 0, 4, and 5 are fixed, while others are not fixed (by default, fitted by the program).

```
[pader]
fix_interactions = [0, 1]

[[pader.interact_controls]]
index = 4
fix_sats = [0, 4, 5]
```

Note

In addition to fixing the parameters using the methods described above, it is also to *disable* some interactions completely during the fitting process, as in the simulations. See *Controlling the energy components* for more details.

6.1.4 Controlling FP threshold

During the OTF fitting process, the FP calculation is determined at each step by comparing the predicted uncertainty to the thresholds. There are two threshold values controlling such determination: the (normal) FP threshold and the high threshold. The latter is precisely the former one multiplying a certain scale (defined by `active.high_thres_rate`, default is 2.0). The FP calculation is determined using an algorithm similar to [VaspMLFF2019]:

1. If the threshold exceeds the high threshold, the FP calculation is executed immediately, followed by parameter fitting.
2. If the threshold exceeds the normal FP threshold but does not exceed high threshold, the FP calculation will be triggered if and only if the number of steps passed since last FP calculation is no less than that configured by `active.fp_distance`. The parameter fitting is triggered if the number of FP calculations since last parameter fitting is no less than that configured by `active.fit_distance` (default is 0, indicating parameter will always be fitted after each FP calculation).

The FP threshold may be determined by different strategies, as controlled by `active.thres_strategy` key. Options are:

1. `active.thres_strategy = "Fixed"`: The FP threshold is fixed to a given value. In this case, the threshold value must be specified in the input file using the key `active.thres_value`.
2. `active.thres_strategy = "Dynamic"`: The default. The FP threshold will be updated automatically during the fitting process, see details below.

When the `active.thres_strategy` is set to `Dynamic`, the uncertainty threshold σ_t is updated automatically during the OTF MD simulation. There are two different modes for determining the threshold.

(2a) In the first (default) mode, the program logs the uncertainty values σ_i each time *after* the parameter is updated. The average value $\bar{\sigma}$ and standard deviation D_σ of the most recent n (defined by `thres_average_count`) values are computed. If the ratio $D_\sigma/\bar{\sigma}$ (the standard deviation divided by the average value) does not exceed certain threshold p_σ (defined by `thres_update_std_rate`), the uncertainty threshold is updated to $a\bar{\sigma}$, where a is defined by `thres_update_scale`. In other words, the threshold is updated if

$$\frac{D_\sigma}{\bar{\sigma}} \leq p_\sigma,$$

where

$$\bar{\sigma} = \frac{1}{n}\sum_{i=1}^{n}\sigma_i,$$

and

$$D_\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\sigma_i - \bar{\sigma})}.$$

Before the threshold is setup at the very beginning of the OTF fitting, the FP calculation is triggered every `fp_distance` steps. The FP threshold will tried to be set up after the number of collected σ_i values is no less than `thres_init_count`.

(2b) In the second mode which is enabled by setting `active.update_thres_without_fp=true`, the program logs the uncertainty values σ_j for each step where FP calculation is *not* triggered. Similar to the (2a) mode, the average value $\bar{\sigma}$ and standard deviation D_σ of the most recent n (but here defined by

`thres_average_count_without_fp`) values are computed. If the ratio $D_\sigma/\bar{\sigma}$ does not exceed the threshold p_σ (defined by `thres_update_std_rate_without_fp`), the uncertainty threshold is updated to $a\bar{\sigma}$, here a is defined by `thres_update_scale_without_fp`.

Warning

The (2b) mode is highly experimental. The configuration keys as well as their value interpretation and default values, and the program behaviors, may be changed in future versions without any special notice.

Tips for controlling the threshold

Important

The strategy introduced in this section is highly experimental and only provided for reference. We do not try to claim the introduced strategies as a somehow "standard" practice.

It is clear from above that the FP threshold is controlled by many parameters. Although the FP determination are controlled by these parameters, and the number of triggered FP calculations are closely related to these parameters, most of these parameters have *no* direct explicit relation to the number of triggered FP calculations (that means you cannot generally say, increasing some parameter will increase the number of FP calculations, or something like this, qualitatively or quantitatively, for most of the parameters). Here we provide some comments and tips for adjusting these parameters.

Generally, by our preliminary tests, we can use two fitting modes, as determined by several parameters, which will be used in *Practical fitting workflows*:

1. Normal fitting or dense fitting. This is typically used in the initial stage of the fitting, with low temperature. In this mode, the threshold is updated dynamically (`thres_strategy = "Dynamic"`), with `thres_update_scale` not too large. This will provide a sufficient sampling of the potential energy landscape at the low energy range.
2. Sparse fitting. This is typically used in high temperature, for solving the energy boundedness problem. In this mode, the threshold strategy may be set to fixed value (`thres_strategy = "Fixed"`), with a threshold value (`thres_value`) two to three times of the average uncertainty values observed in the fittings, to reduce the number of FP calculations required. Several trials may be needed for determining a proper threshold value. Alternatively, the dynamic threshold strategy may also be used, along with a relatively large `thres_update_scale` (typically around 2.0 to 3.0). The core purpose of these settings is just to reduce the number of FP calculations.

We also have some other comments, on several cases which may be encountered during the fitting process.

Important

We make these comments with the assumption that all other configurations of OTF fitting are proper. We emphasize that any improper configurations could lead to poor performance of OTF fitting, even if such improper configuration is not directly related to the threshold control at all (for example, FP calculation configurations, interaction generation configurations, etc). Thus, if you find the performance of OTF fitting is low (too much FP calculations are triggered, or the uncertainty cannot be reduced after FP), first check other configurations.

(1) Sometimes difficulties may be encountered during the setting up of threshold of the first several OTF fitting runs (mostly the first one). In this case, the FP threshold changes drastically (typically increases) at the beginning of the fitting, and too many FP calculations are triggered, making the fitting efficiency rather low. A simple solution to this problem is to use a fixed threshold (`thres_strategy = "Fixed"`) with a proper value for the first (and possibly several following) OTF runs. After sufficient uncertainty history data is collected, it is suggested to switch to normal dense fitting mode to ensure the proper sampling.

(2) The exceeding of high threshold should be considered as abnormal cases, and this should not happen frequently. If the FP calculations are continuously triggered due to the exceeding of high threshold, it is suggested to increase `high_thres_rate` to increase the high threshold and decrease the number of useless calculations on very similar structures.

(3) If the uncertainty may change drastically during the fitting process (including continuing fitting), for example, at the beginning of changing supercell, or after reset alloy occupations, setting `thres_average_count` could make the threshold more sensitive to such change, and may be useful for reducing the number of continuous calculations after such changes. This is extremely useful when the uncertainty could not drop below the threshold after several FP calculations after such changes.

(4) If `thres_strategy = "Fixed"`, the threshold value must be chosen properly. Although the small change of such threshold value is typically not important (the user typically need not pay much attention to the *precise* determination of such threshold, such as the choice between 0.01 and 0.0101), an improper setting of threshold could still affect the fitting performance and result. Specifically, if the threshold is set to a value that is too large, the structures triggering FP calculations may be too much deviated from "normal" structure (i.e., the structure itself is too "strange", or with too high energy), which will drastically increase the fitting RMSE value, leading to the reduction of accuracy to the fitting of the (important) low-energy range, or even cause the failure of FP calculation. On the other hand, if the threshold is too low, too much FP calculations may be triggered. Thus, it is strongly suggested to watch the running of OTF fitting carefully, and adjust the threshold value (by stopping and restarting the OTF fitting) if the threshold is improper.

6.2 Fitting with existing data

In *On-the-fly learning*, we have introduced the OTF MD approach for fitting the parameters of the effective Hamiltonian. This OTF approach is typically the most commonly used approach for parameter fitting. As we have already introduced, the OTF fitting calls FP calculations when necessary. Such FP calculations are the most time-consuming part in the fitting process. The result of the FP calculations (useful for OTF fitting) are the potential energy, force and stress for given atomic structure, and are stored as the ASE checkpoint files. Using these stored FP data, it is possible to obtain the parameters (without actually performing OTF fit), as all the data required for the regression process are available.

For some purposes, we may want to fit the parameters from existing data. This fitting process should be much faster than the actual OTF fitting because no actual FP calculations performed, making it suitable for some fast testing purposes. A typical application scenario is to test and compare various fitting configurations and/or interaction sets with the same data, to choose an optimized configuration.

In this section, we will introduce some techniques for fitting from existing data, provided for different conditions. It should be noted that some of these techniques are *not* active fitting, but just fit based on existed, pre-constructed data set (the data set are typically obtained from previous OTF runs). The validity of the parameters obtained from these methods should be carefully tested if one means to use it for production runs.

Basically, according to the configuration differences from the previous runs (the runs that generated the FP data) and the required further operations, the following techniques are available. Note that the costs of these techniques are in ascending order. Thus, the method arranging at the front are preferred if multiple choices are available.

1. If the model configurations (including interaction set, enabled interactions and DoFs) are not changed, and the `active_fit_out.json` file is available, then the `fit_from_data` is applicable. This technique simply loads the `active_fit_out.json` file (or other files with the same format) and fit the parameters based on these data. This is useful for fitting parameters with *regression* configurations (for example, the weights for energy or stress part, etc) changed.
2. If the model configurations have been changed (mostly, a new interaction set is generated), the data in `active_fit_out.json` is not consistent with the program, and cannot be loaded. In this case, `merge_otf_fit` is applicable. This technique goes through all the FP data stored in previous OTF runs, analyzes the structure and energy/force/stress contributions based on the newly provided interaction set, and fit the parameters. The parameters are fitted *only once* during each run. Finally, the collected data is written into `merge_otf_data.json`,

in a format consistent with `active_fit_out.json`, which could be used for later `fit_from_data` operation if required. This is useful for fitting parameters with changed interaction set.

3. If continued OTF fitting from the data generated (`active_fit_out.json` format) is desired, the `reuse_data_fit` should be applied. The model configurations may be changed comparing to the original OTF runs which have generated the FP data. This method goes through all the available data that generated in *one* previous OTF run, load the FP data if required (determined using the uncertainty threshold, as in the OTF runs), and fit the parameters when required (as configured by the input file). This method is different from the previous two methods because it provides the uncertainty *history* information which is useful for threshold determination (which is required by OTF fitting). This technique is typically much *slower* than `merge_otf_fit` because it requires the parameters to be fitted for multiple times (as in OTF fitting).

In the following sections, we will introduce these methods in more detailed manner.

6.2.1 `fit_from_data`

In this program, the parameters are simply fitted from the data available in `active_fit_out.json` file or files with consistent format. FP data are *not* parsed directly. The model configuration (interaction set, disabled interactions or SATs) must be exactly the same as the original run generating the dataset. The program simply produces a set of parameters fitted from given data. No uncertainty history for the collected data is produced.

To run this program, use the command like

```
python3 -m heff_gen.fit.cui.fit_from_data -i <data_file>
```

Here, <data_file> is the filename of the input data, for example `active_fit_out.json`.

The unit cell metadata and model configuration metadata (interaction set, parameter file template, pre-calculated parameters etc) must be available at the working directory. The parameters will be written into the `heff_params_fit.yaml` file (by default).

> **Warning**
>
> To use this program, the model configuration must be exactly the same as the original run. Otherwise the result is undefined. The program does not guarantee to detect such inconsistency if occurred.

6.2.2 `merge_otf_fit`

In this program, the OTF data from *a set of* working directories are parsed in order, all the FP data (generated in previous FP runs) are collected and parsed. The atomic structure stored in the FP data are parsed and projected into the local mode space, and the potential energy contributions are analyzed based on the newly provided model metadata (in the working directory of `merge_otf_fit`, to be called "current working directory").

To run this program, a working directory should be prepared. This could be obtained simply by copying one of the original OTF working directory, with new model metadata (if needed). The configuration file `fit_config.toml` must provide a new table called `merge_otf`, with an array of string `data_dirs`, setting the directories from which the OTF data will be loaded ("data directories" hence). For example:

```
[merge_otf]
data_dirs = [
    "../work-20",
    "../work-40cub",
]
```

Each item in the `data_dirs` array specifies a data directory, where both absolute and relative paths are allowed.

The regression configurations (`fit_general`, `regression`, `linear`, `bayesian` tables) could be given similar to the OTF configuration, controlling the current fitting task.

To run this program, use the command like

```
python3 -m heff_gen.fit.cui.merge_otf_fit
```

The program will parse all the given data directories in `data_dirs` in order, loading all FP data (without any data selection), and perform a *single* parameter fitting at the end of the program. The data will be written into a file called (by default) `merge_otf_data.json`, which has the format consistent with `active_fit_out.json`, which could be used in `fit_from_data` program. However, as the parameters are fitted only once and no uncertainty history available, this file should typically *not* be used to continue an OTF fit (for that purpose, use the `reuse_data_fit` program).

It is worth noting that some configurations are loaded from the data directories, while others are loaded from the current working directory (overwriting those from data directories). More specifically:

- The unit cell metadata (unit cell specification file and atomic structure file, LWF data etc) are loaded from current working directory.
- The supercell configuration (`cell` table) and energy contribution and analyzing configurations (`energy`, `ewald`, `pader` tables, etc) are loaded from the data directories.
- The model metadata files (interaction set, parameters, etc) are loaded according to the *configured paths* in data directories. Note, if the configured paths are relative paths (the default paths are just `.`, which are relative paths), they are resolved based on *current working directory*, not the data directory.
- The configurations for regression are loaded from current working directory. This includes the `regression`, `linear`, `bayesian` and `ridge` tables.
- The configurations for fitting procedure control are in principle loaded from data directory, including the following tables in the configuration file: `cell`, `physical`, `ewald`, `init`, `energy`. However, the `pader` and `struct_proj` tables are loaded from current working directory. Note that some of these tables may not be used in this merge-otf fitting process.
- The Ewald summation file, loaded from `energy.files.ewald_path`. If the path is given by relative path, it is resolved based on the *data directories*. This is different from other model metadata files.
- The `continuing` configuration is ignored; the `active_fit_in.json` files in data directories are ignored, except for the first data directory, whose continuing data file could be loaded by setting `merge_otf.continuing_first = true`.

> **Warning**
>
> Please pay special attention to the model metadata resolving algorithm if you have specified their paths manually. Typically we recommend to use relative paths, as the absolute path may result in unexpected results.

6.2.3 `reuse_data_fit`

In this program, all FP data generated by *one* previous OTF run are parsed. The atomic displacements in the stored structures are projected into the local mode space, and the energy contributions are analyzed based on provided model metadata. The energy, force, stress and their uncertainties are predicted for each of these FP structures, and parameters are fitted when required, as controlled by active fitting framework. The FP data may be selected based on the uncertainty threshold or not (meaning all data are used). This workflow is similar to the OTF fitting, with exceptions that MD is not performed (but stored structures are loaded), and FP calculations are not *actually* performed (but stored FP results are loaded). In this sense, it is sometimes referred to as a type of *pseudo-OTF* (pOTF) fitting method. We also call this fitting method "reuse data fitting" because it reuses all the FP data generated by previous OTF run. Obviously, this reuse data fitting is much faster than actual OTF fitting, but it takes more efforts than the `merge_otf` fit, because the parameters are fitted for *multiple* times during this process.

To run this program, the user should first prepare a *working directory*. The working directory should contain the unit cell metadata and model metadata (interaction set, parameter template file, pre-calculated parameters, etc), similar to the OTF working directories. The path to the original OTF directory (*data directory* hence) should be specified by `data_dir` field in the `reuse_data` table of `fit_config.toml` file. The provided metadata could be different from original OTF run.

The data directory should be a directory where an OTF fitting run has been finished. The following files in the data directory are required:

- The calculation log file `cal_log.csv`;
- The occupation history file `occupations_history.yaml`;
- The FP calculation working directory `fp_dir/` with ASE checkpoint files.

These files should have been automatically generated during the finished FP run. All the files in the data directory are not modified.

Tip

The working directory could be simply obtained by copying necessary files from the data directory, with new model metadata files if needed.

The fitting configuration file `fit_config.toml` in the working directory (but not the data directory) controls the whole fitting process. In addition to the `reuse_data` table, other tables controlling the active fitting or the regression, like `active`, `fit_general`, `regression` are also used.

After all input files prepared, this program could be executed by running the following command at *working directory*:

```
python3 -m heff_gen.fit.cui.reuse_data_fit
```

The outputs are given in both standard output and output files. Most of the output files are in similar formats to corresponding files generated by OTF fitting. Similar to the `otf_plot` program, the running status could be visualized by the following command, both during and after the fitting:

```
python3 -m heff_gen.fit.plot.reuse_plot .
```

Also similar to OTF fitting, the final potential could be obtained by the `heff_dof_potential` program, and the `-p` option in `reuse_plot` is also supported.

6.2.4 Fitting with multiple data directories

As introduced before, the OTF fitting for a model typically includes multiple OTF MD runs, each in its own working directory. For `reuse_data_fit`, sometimes (often) it is also necessary to fit the parameters with the data in some of (or all of) these working directories (serving as data directories in `reuse_data_fit`). As the `reuse_data_fit` could process only one data directory for a single run (see above), we provide some scripts for convenience, to prepare the working directory for `reuse_data_fit` and run the fittings. The scripts are located in the `related_scripts/reuse_data_fit/` directory of `heff_gen` source root. Here we introduce the usage of such scripts, and the common practice for running the `reuse_data_fit`.

Note

These scripts are only applicable to Linux or systems with shell that could parse these bash scripts. We have not extensively tested them. We strongly recommend to read and understand these scripts before using them.

Assume all the OTF directories (data directories) to be used are located in a same directory, called, for example, `otf-runs`. To prepare for `reuse_data_fit`, create a sibling directory of `otf-runs`, for example, `reuse-data`. Then copy the scripts from `related_scripts/reuse_data_fit/` directory of `heff_gen` to this directory. The file structure may look like

```
├─otf-runs
│  ├─work-20
│  ├─work-40
│  └─work-80
└─reuse-data
   ├─make_reuse_dirs.sh
   ├─run_reuse_fits.sh
   └─...
```

Then edit the `make_reuse_dirs.sh` script, ensuring that the `SOURCE_DATA` points to the OTF data root directory (`otf-runs` here), and `SUB_DIRS` array contains the sub-directory names for the OTF data directories. For example, in this case:

```
SOURCE_DATA=`pwd`/../otf-runs
SUB_DIRS=( work-20 work-40 work-80 )
```

Then run this script

```
sh make_reuse_dirs.sh
```

The working directories corresponding to the data directories with the same directory names should be created. The data are copied from the data directories, with `data_dir` in `[reuse_data]` table of `fit_config.toml` pointing to the original data directory. In addition, another directory called `merge-otf` is created, containing data copied from the last OTF data directory, and `merge_otf.data_dirs` including *all* data directories. To run `merge_otf_fit`, enter the `merge-otf` directory and follow the instruction in *merge_otf_fit*. To run `reuse_data_fit` for one working directory, enter such directory and follow the instruction in *reuse_data_fit*. To run `reuse_data_fit` for *all* working directories, directly execute the `run_reuse_fits.sh` script.

If the configurations need to be modified, the `update_config.py` script could be used. This is a Python script reading and modifying the `fit_config.toml` file, and should be edited according to the configurations as needed. The `update_config_all.sh` script simply apply such Python script in all working directories.

Tip

We strongly recommend to read and understand these scripts before using them. These scripts are not complex.

6.3 Multi-OTF fitting

Tip

This feature is highly experimental. The configurations may be changed in future versions.

The parameter fitting for a material model typically needs several OTF fitting runs to get converged and sufficiently accurate parameter set. In most cases, different supercells are needed. The basic method for realizing this is to run several different continuing OTF fittings in order with different supercell matrices (`cell.supercell_diag` or `cell.supercell_matrix`). This is valid, but the repeated continuing run may be somehow inconvenient, and the output and input of continuing data file (`active_fit_out.json`) may take much time when the model is complex or the

data set is large. To make this easier, we provide another fitting method called multi-OTF fitting, in which a series of supercells are used in sequence to fit the parameters.

In multi-OTF fitting, a sequence of supercell matrices are specified, using an array called `cells` (each has the same format as the `cell` table) in the `multi_otf` table of the configuration file. These cells are used in sequence by the program, to fit the parameters. The fitting process is almost equivalent to running an OTF fitting for each of these cells in sequence, with the other configurations specified in the input file, and the `cell` table replaced by the corresponding element in `cells` array. These OTF fittings are called sub-OTF fittings. Each sub-OTF fitting runs in a subdirectory of current working directory (default name is `otf_<index>`, for example `otf_0`). After each sub-OTF fitting, the continuing data JSON file is written *synchronously* to its subdirectory. The program continues while the output is in progress, and reading the continuing data file before each sub-OTF is not needed (except for the first, of course), making the program faster than running several independent OTF fittings in turn.

To run multi-OTF fitting, the fitting configuration file (`fit_config.toml`) should be prepared as normal OTF fitting (*On-the-fly learning*), with a table called `multi_otf` containing an `cells` array required. For example:

```
[multi_otf]
[[multi_otf.cells]]
supercell_matrix = [ [ 1, 0, 0,], [ 0, 1, 0,], [ 0, 0, 1,],]

[[multi_otf.cells]]
supercell_matrix = [ [ 2, 0, 0,], [ 0, 1, 0,], [ 0, 0, 1,],]
```

In this example, two different supercells ($1\times1\times1$ and $2\times1\times1$) will be used in turn. The table `cell` in the configuration file has no effect, although it should be provided.

The multi-OTF fitting could be executed by using command like

```
python3 -m heff_gen.fit.cui.multi_otf_fit
```

6.3.1 Reciprocal-space traversing strategy

In this section, we will introduce a strategy and program for generating supercell matrices used for the multi-OTF fitting, to be called reciprocal-space traversing strategy. In this method, the supercell matrices are generated so that a pre-determined reciprocal-space q-point mesh is fully sampled.

> **Note**
>
> The method introduced in this section is only one possible method for generating the supercells used in multi-OTF fitting. The supercells in multi-OTF could be generated in different (in principle, arbitrary) ways. We do *not* try to claim this method as a somewhat "standard" method.

This strategy is actually inspired by a special method for generating supercells (especially non-diagonal ones) in the first-principles computation of phonon dispersion, where calculating the force constant matrix of some supercells could sample specified some reciprocal-space q-points. For example, calculating the force constant matrix of a $2\times2\times2$ supercell could sample q-points like $(0,0,0)$, $(1/2,0,0)$, $(1/2,1/2,1/2)$ etc (in fractional coordinate in the first Brillouin zone). Bartomeu Monserrat et al have shown ([PhononSupercell]) that any q-point in the form $(m_1/n_1, m_2/n_2, m_3/n_3)$ (where $m_1, m_2, m_3, n_1, n_2, n_3$ are integers) could be sampled by constructing a supercell of with size (times of the unit cell) *at most* $LCM(n_1, n_2, n_3)$ (where LCM is least common multiplier). The authors have also provided a Fortran program for generating supercells from given q-points, which is available as a supplementary file of their article.

In our package, we have provided some helper programs (with the help of `Phonopy`) as pre-processor and post-processor to make the supercell generation more easily, utilizing the code provided by Bartomeu Monserrat et al.

In practice, first create a new directory containing the unit cell atomic structure file (the same as that in the unit cell metadata files). The default file name is `POSCAR`. Then run the following command in that directory:

```
python3 -m heff_gen.util.cui.prepare_supercell_generation <mesh_number>
```

where `<mesh_number>` should be replaced by the q-point mesh number to be generated (with three integers). For example, `4 4 4` indicates a $4 \times 4 \times 4$ q-point mesh. Other available arguments could be seen by the `--help` argument.

The above command creates the input files used by the program provided by Bartomeu Monserrat et al. Then compile their program with a Fortran compiler. For example:

```
gfortran generate_supercells.f90 -o generate_supercells.x -O2
```

Then run such program in the directory above and run the post-processing program:

```
./generate_supercells.x   # NOTE: replace it with the actual executable path on your␣
↪system
python3 -m heff_gen.util.cui.supercell_matrices_to_toml
```

The result is stored in a file called `supercell_configs.toml`, containing the supercell matrices generated by the program in TOML format that is readable by the multi-OTF fitting program. The content of such file could be *appended* to the fitting configuration file using command like:

```
cat supercell_configs.toml >> fit_config.toml
```

6.4 Multi-simulation-OTF iteration

Note

This feature is available only as of version 0.30 (Jun. 5, 2026).

Note

This feature is highly experimental. The configurations may be changed in future versions.

In practical model building processes, typically, there will be several continuing OTF fittings aiming at solving the *convergence* problem in certain circumstances (including the supercell shape, temperature, pressure, electric field, etc), see, e.g., *Practical fitting workflows*. In these processes, an efficient scheme is to find some supercell configuration (that is compatible with FP calculations) along with some MD simulation conditions (temperature, pressure, electric field, pseudorandom number generator seed, etc) that lead to divergent simulation result (unreasonable DoF configurations and/or unreasonable energy predictions), and then use exactly the same configuration to run the continuing OTF fitting to refine the model under such circumstance. Here, for convenience, we provide a new tool called *multi-simulation-OTF iteration*, which does the procedure above in a more automatic way. The whole procedure is made up by several *iterations* consist of two steps: simulation and OTF fitting. In each iteration, a *test simulation* is first executed to check the convergency (detailed below). If the model is considered to be converged, the OTF fitting is skipped. Otherwise, OTF fitting is executed with given condition. Technically, the whole procedure could be understood as a specialized multi-OTF fitting (see *Multi-OTF fitting*), but with the following major differences:

1. For each given cell, a *test simulation* run is first executed to check the convergency. If the model is considered to be converged with this set of configuration, the OTF fitting with this cell is skipped. Otherwise, OTF fitting is executed with this given cell.
2. All the cells (`multi_otf.cells`) could be iterated in sequence for more than one time (`sim_otf_iter.repeat_times`). If no OTF simulation is executed for all cells during one sweep of all cells, the whole procedure is ended.

Like the multi-OTF fitting, all the configurations need to be the same for all iterations, except the supercell configuration. The test simulations are executed in a sub-directory of the working directory (default: `test_simulation`) with configurations mostly come directly from the fitting configuration file (`fit_config.toml`). The configurations for the test simulations need to satisfy certain requirements to make the procedure valid and efficient, see later.

> **Tip**
>
> The `heff_gen.fit.cui.collect_otf_cells` program collects all the cells used in a range of OTF fittings, which may be useful for preparing the input file.

The procedure is executed using the command like

```
python3 -m heff_gen.fit.cui.multi_sim_otf_iter
```

6.4.1 Test simulation configurations

> **Important**
>
> Read this section carefully. If the configurations do not satisfy the requirements, the procedure may cause error or low efficiency.

The *test simulation* embed in the multi-simulation-OTF iteration means to find the possible unbounded result from the simulations. Controlled by the `flow` table and `test_simulation` table in `fit_config.toml` file, the simulation is executed like normal simulations (see *Running Simulations*), with almost all configurations come directly from accepted tables (e.g., `cell`, `simulation`, `physical`, etc) in the fitting configuration file (`fit_config.toml`), with several exceptions:

- The `cell` table in the configuration comes from the element in `multi_otf.cells` array, not from the `cell` table in the top of the `fit_config.toml` file.
- The `simulation.sim_steps` comes from `test_simulation.sim_steps`. The `simulation.sim_steps` key controls the number of steps of the OTF fitting, not the test simulation.

> **Important**
>
> Check the executable path and MPI support carefully.

The test simulation is executed using *external* executable `heff_run_sim` of this package, not the Python binding. MPI must be enabled for the executable (see *MPI parallelism* and *Installation*). The version and the building configurations of the package providing such executable must be consistent with that running current fitting procedure, otherwise the result is undefined. In the test simulations, MPI parallelism is used for several independent tasks with different pseudorandom number generator seeds (`flow.mpi_dist = "Seed"`, see *Parallel simulation for independent tasks*), meaning to find possibly unbounded configurations under different initial conditions. The number of tasks (MPI processes) is determined by the `test_simulation.mpi_size` key, for which a positive integer value must be given. The number of OpenMP threads is given by `test_simulation.omp_num_threads` key (see also *OpenMP parallelism*). The test simulation is equivalent to executing the following command (taking `test_simulation.mpi_size = 4` as an example):

```
mpirun -np 4 heff_run_sim
```

If the command is not proper, `mpi_command`, `mpi_process_param`, and `run_sim_command` may be changed in the `test_simulation` table.

Likewise, before the simulation, and Ewald summation progress is first executed using the `heff_cal_ewald` command, also with the external executable. This command can be changed by the `cal_ewald_command` key.

For compatible with OTF MD simulation, the test simulation must be an MD simulation (`flow.alg = "MD"`), and the workflow task (`flow.task`) must be `Single`. These configurations will be automatically changed if not properly specified in the input file.

For the efficiency of finding unbounded structures, the OTF fitting should use exactly the same configuration of the test simulation, resulting the following (and possibly more) requirements:

1. The MD integrator must be `Verlet`.
2. In OTF MD fittings, there are several configurations for improving the DoF-space searching efficiency that may break the MD trajectory before the first FP calculation, including `reset_momenta_interval`, `reset_occupations_interval`, and `restart_interval`. If these reset or restart strategies are used, their interval should be no less than the number of MD steps executed in the test simulation (`test_simulation.sim_steps`).

After the test simulation, the program try to check the consistency of each MPI process. If the simulation of an MPI process is considered to be divergent, the OTF fitting will be executed with the same supercell configuration *along with the pseudorandom number generator used in that MPI process*. If all MPI processes are considered to be converged, the OTF fitting will be skipped.

The convergency of each MPI process is checked using the following criteria:

1. If an exception is thrown during the simulation (with the generation of `mpi_error<mpi_rank>.yaml` file), the simulation is considered to be divergent.
2. Otherwise, if `test_simulation.predict_unbound_by_energy` is set to `true`, and the *average* potential energy is lower than `test_simulation.unbound_energy`, the simulation is considered to be divergent.

Warning

Any exception thrown during the simulation is considered as divergent. If the input causes other exception thrown during the simulation, the behavior is undefined.

6.5 Interaction selection and compression

In all the model building procedures introduced till now, the interaction used for the effective Hamiltonian are just those generated in the *Generate interactions* step. The interactions generated in such step could be complex and may include many SATs (up to thousands of SATs for complex models). However, most of these SATs may be not important for the effective Hamiltonian; only a small amount of them are significant to the potential energy landscape. In most cases, we do not have enough knowledge to determine which part of the SATs are significant. One possible solution is first generate a relatively large interaction set that could describe the model well (which may lead to over fitting), and then try to select a part of SATs (and their corresponding parameters) that are significant to the potential energy landscape according to the training data. Such selection process makes the final model contain fewer SATs, providing more clear physical impression, and reducing the computational efforts.

6.5.1 Machine-learning-based model construction (MLCons)

In this package, we implement a machine-learning (ML) based model construction procedure, to be denoted as MLCons in this document and in the codes. This method selects significant SATs from a given interaction set generated in advance. This method is similar to the MLMCH method proposed by Hongjun Xiang et al (see [MLMCH2020]), with several modifications. For more details of this method, we strongly recommend to read their article [MLMCH2020].

The default input file is still `fit_config.toml`. For MLCons procedure, there is a table named `mlcons`, determining several detailed configurations.

In the MLCons procedure, the pre-generated interaction set and FP data collected during OTF fitting (use `active_fit_out.json`) or from merge-otf fitting (use `merge_otf_data.json`) are used as inputs. The FP data is typically loaded from the file named `linear_data.json`. During this procedure, the FP data are divided into train set, validation set (optional, enabled by default), and test set. The model (i.e., the constructed effective Hamiltonian) are evaluated mainly by the variance

$$\sigma^2 = \frac{|Y - \tilde{Y}|^2}{N - p},$$

where Y is the energy, force and stress predicted by the model, $\tilde{Y}$ is such data from DFT calculation, N is the number of coefficients in Y, and p is the number of parameters in the model. The data in Y and $\tilde{Y}$ are normalized to get dimensionless values (see more details in [ALFE-H-2025]). For some cases, following [MLMCH2020], we also use the regularized variance

$$\lambda^p \sigma^2,$$

where $\lambda \geq 1$ is a hyperparameter.

The MLCons procedure includes several iterations on a pre-defined decreasing λ values (`mlcons.pow_lambdas`). With each iteration, a λ is fixed, and the model is updated using greedy strategy. At the very beginning, the effective Hamiltonian contain only the reference energy term (constant term) unless initial selection given (`mlcons.continuing_search`, available as of Oct. 11, 2025). In each step, the program tries to remove an existing SAT from the model, or add a new SAT to the model, or replace one SAT with another. In all cases, the program selects the best possible model at this step, according to the (regularized) variance. After the iterations of λ, sequential back search (SBS) is performed. Such SBS process finds the model with minimal test set variance and also determine the relative importance of the SATs (parameters) in the selected models. More details for the algorithm are given in [MLMCH2020]. At the end of the program, two "best" models are given. The first is the "safe" model, which is obtained directly from the first local minima of the test set variance in the decreasing λ sequence. The second is the "best predict" model, which is the model find in the SBS processes with the minimal test set variance.

In practice, the program could be executed by the command like

```
python3 -m heff_gen.fit.cui.run_mlcons
```

In the MLCons process, by default, the bounding-only SATs (*Bounding-only interactions*) are disabled. This require the interactions topology file `heff_interactions_topology.yaml` to be provided. See also `disable_bounding_only` in *Input reference*.

During the MLCons process, the program will provide the variance information. At the beginning, the program produces the variances from the initial model (by default, model with only reference energy term, see also `continuing_search` in *Input reference*), maximal model (with all *enabled* SATs added), and full model (with all SATs added, including disabled ones):

```
INFO:mlcons:Initial variance   : 7.779667063458995  10.30131902469301 9.13750740935197
INFO:mlcons:Max  model variance: 0.0804056339164163  0.1733003539838405 0.
↪24295055385147343
INFO:mlcons:Full model variance: 0.0804056339164163  0.1733003539838405 0.
↪24295055385147343
```

In each line, the three values correspond to the variance in the training set, validation set (if enabled), and the test set. The variance of the maximal model (second line above) is typically the best (minimal) variance that is possible to be reached during the MLCons process. If the variance of validation set and test set is significantly larger than that of training set for the maximal and full models, significant over-fitting may occur in these models. As the MLCons processes, the variance is shown each time a new parameter is added, removed, or substituted, for example:

```
INFO:mlcons:Add term indexed 381 (search rank 737); best variance 0.24649726419599785  0.
↪24030584432916166 predicted: 0.24649726419599804  0.24030584432499455
INFO:mlcons:Delete term indexes 30 (search rank 0); best variance 0.24678233029735722  0.
↪2377109604717556  predicted 0.24678233029735722  0.23771096047409201
```

Here, the two values after `best variance` are the variances for the training set and validation set (if enabled), respectively. The `predicted` variances are shown only if `incremental_adding` (see *Input reference*) is enabled, and these values should be close to the best variances shown before.

Important

It is important to watch the variance values during the MLCons process. If the variance is too large, the result may be unreliable. For this case, one may try to rerun the program, because the data processing depends on some non-deterministic algorithms.

The time consumed by this program dependents on the model complexity and the size of FP data. At the end of the program, the two "best" models are output to files. The best safe model is written into `heff_params_best_safe.yaml` and `param_selection_best_safe.yaml`; similarly, the best predict model is written into `heff_params_best_predict.yaml` and `param_selection_best_predict.yaml`. For each model, the first output file is consistent with the parameter input file for the simulations (`heff_params.yaml`), and the second file may be used for later OTF fitting, continuing MLCons, and bounding procedure, etc. These two parameter files contain the summary of number of parameters selected at the beginning as comments. For example:

```
# Parameter file output by Heff program at 2025-11-02 12:53:54.293560431
# Additional info: Parameters selected by MLConstructor  selected/non-zero/all: 120 /␣
↪4027 / 4409
api_version: 3
    # ... (omitted) ...
```

In this example file, the MLCons procedure has chosen 120 parameters from the input 4409 possible parameters.

The choice between the two models have been discussed in [MLMCH2020]. We just note here that for some cases, where the test set variance may be somewhat fluctuating, the "best safe" model may contain much fewer parameters (SATs) than the best predict model. In this case, we recommend to use the best predict model instead, because the "best safe" model may lead to great under fitting.

6.6 Energy boundedness

Note

The closed bounding procedure is available as of version 0.19 (Oct. 23, 2025), and the open bounding procedure is available as of version 0.20 (Nov. 1, 2025).

From physical view, it is clear that the effective Hamiltonian must be bounded from below, which means the potential energy should *not* approach negative infinity ($-\infty$) for any configuration (the potential energy is possible to approach positive infinity for some unreasonable configurations). Such energy boundedness should be guaranteed by the model building (parameter fitting) process. In principle, this purpose should be naturally reached if the training set (including the structures with FP calculations) is properly built. However, we found such ideal condition may be hard to satisfy. For some complex models, the energy boundedness is difficult to be reached. Furthermore, for some cases, if the interaction set is not properly generated, the effective Hamiltonian may even impossible to be bounded from below.

To try to solve this problem, we have implemented some strategies to bound the effective Hamiltonian from the mathematical view, as also proposed by the second-principles model building [SecondPrinciples2017]. As the effective Hamiltonian is formed by SATs with polynomial forms, it is possible to approach negative infinity ($-\infty$) only when some of the degrees of freedom (DoFs) approach infinity at some direction. The behavior of the polynomial at infinity is determined by the term with the highest order. The low order terms are *controlled* by the high order terms (using the terminology from [SecondPrinciples2017]). In the one dimensional case:

- If the highest order term has an odd order (e.g., ax^3), the polynomial is not bounded, because it always approaches negative infinity as the variable x approach infinity at one direction (positive or negative, depending on the parameter a).
- If the highest order term has an even order (e.g., bx^4), the polynomial is bounded if and only if the parameter $b > 0$.

The basic idea of the bounding procedure is just to add new high-order terms controlling the terms that may lead to unboundedness. For the realistic effective Hamiltonian, this is much more complex, because the local mode DoFs dependent on the unit cell position, thus the effective Hamiltonian has a very large variable space. More details will be introduced in *Relations between SATs* and *Boundedness condition*.

Technically, there are two different bounding procedures implemented in this package: the closed bounding procedure and open bounding procedure, depending on whether the interaction set is allowed to be updated during the bounding procedure:

- **Closed bounding procedure**. The interaction set is generated in advance (using `heff_gen_interaction`, see *Generate interactions*), and the SATs are selected from such interaction set to bound the effective Hamiltonian. This bounding procedure is relatively simple and fast, but typically requires a large interaction set with some high-order SATs to be generated in advance. The *close* indicates the SATs are selected in a closed range, i.e., the pre-generated interaction set.
- **Open bounding procedure**. The interaction set is updated during the bounding procedure. The new interactions (SATs) controlling the problematic SATs are generated dynamically. This procedure does not require a large interaction set at the beginning (just a normal interaction set is enough), but it is more complex than the closed bounding procedure, including interaction set updating and FP data loading, etc. Fortunately, most of these procedures are done in a completely automatic way, the user just need to prepare the input data set properly.

In the following sections, we will first introduce the basic concepts and theories about the bounding procedure, and then introduce the two bounding procedures.

> **Tip**
>
> In practice, we always recommend to use open bounding procedure instead of closed bounding procedure for most cases. However, to understand open bounding procedure, it is still necessary to introduce the closed bounding procedure, because it is a part of the open bounding procedure.

6.6.1 Relations between SATs

There are two possible types of relations (about the energy boundedness problem) between SATs:

- **Controlling relation**. Briefly, if an SAT A has "higher even order" than another SAT B, we say A controls B, or B is controlled by A. If an SAT is controlled by any other SATs, it will not be considered as problematic SATs that may lead to unboundedness. For detailed definition or the "higher even order" condition, see below.
- **Perturbation relation**. If an SAT A contains one more alloy COMP in each PROD than B, we say A is a perturbation to B, or B is perturbated by A. If A is a perturbation to any other SAT, we call A a perturbative SAT. An SAT containing alloy COMPs which is neither perturbative nor controlled by other SAT is always considered as a problematic SAT that may lead to unboundedness.

The controlling and perturbation relations form a *graph* data structure between SATs, in which the SATs are considered as the *vertices* of the graph, and the controlling and perturbation relations are represented by the *directed edges* in the graph. These two graphs, along with other related data, form the *topology* between the SATs (interactions). Such controlling and perturbation relations (topology) are detected after the interaction set is generated (*Generate interactions*) or updated (during open bounding procedure), and the data is written into the topology YAML file (default file name is `heff_interactions_topology.yaml`, or `interactions_topology.yaml` in version 0.19). These topology data provides essential supports for the bounding (both open and closed) procedures, and should be provided for these programs.

To visualize the controlling and perturbation relations easily, the controlling and perturbation graphs are written into graphviz format, with default names `control_graph.gv` and `perturbation_graph.gv`. Figures could be generated from these files using the `dot` command like

```
dot -Tpdf control_graph.gv -o control_graph.pdf
```

In the output graphs, the vertices denote the SATs, with indexes consistent with the SAT indexes presented in the `interactions_brief.tex` file. The `A -> B` directed edges denotes that "A is controlled by B" in the controlling graph, and that "A is a perturbation to B" in the perturbation graph.

Determining controlling relations

TODO

Determining perturbation relations

TODO

6.6.2 Boundedness condition

The purpose of the bounding procedure is to find a *mathematically bounded* effective Hamiltonian. Here, the *mathematically* bounded means the effective Hamiltonian will never provide potential energy values approaching the *negative infinity* ($-\infty$) when the DoFs approach the *infinity* at any direction. We emphasize that this *mathematical* boundedness is only strictly valid when the DoFs approach the infinity. There are some cases that the potential energy does not approach negative infinity, but has some very large negative (unreasonable) value when the DoFs take some values, which may be called *physically unbounded* because these large negative energy predictions have no physical meanings. Such physical boundedness will *not* be considered by our bounding procedure. The physical boundedness problem may be solved by constructing the training set properly.

Furthermore, we would like to emphasize that we mean to find a *sufficient but not necessary condition* for the mathematical boundedness condition above. It will be very hard to find the exact necessary condition for mathematical boundedness because of the position-dependent nature of the local mode DoFs and the complexity of the SAT forms, as (maybe) already seen in the section above.

Benefited from its polynomial form, the (mathematical) boundedness of the effective Hamiltonian could be determined directly from the forms of the SATs, and the topology relations (controlling and perturbation relations) between the SATs. The effective Hamiltonian is considered to be (potentially) unbounded if there are any *uncontrolled* SATs (SATs not controlled by any other SATs) that are either:

1. has any COMPs with odd order (e.g., u_1^3, $u_1^2 v_2^3$, etc),
2. has no COMP with odd order, but has negative parameter value, or has the possibility to get negative total perturbated parameter values (if this is a perturbated SAT) by any possible occupation combinations, or
3. has any alloy COMP, but is not a perturbative SAT.

The conditions for non-alloy SATs are easy to be understood. Special notes may be needed for the alloy-dependent SATs (SATs including alloy COMPs). As we have applied (by default) summation rules for the parameters of alloy-dependent SATs (see *Alloy parameter summation rule*), any non-perturbative alloy-dependent SAT with non-zero parameter value

always includes at least one negative component for some atomic occupation combinations. If such SAT is uncontrolled, this negative component of parameters may lead to unboundedness. On the other hand, if an alloy-dependent SAT is a perturbation to another SAT, it may be considered as an adjustment to the parameter for the latter SAT according to the atomic occupations. Such adjustment may lead to negative parameter values for some possible occupation combinations, depending on the parameters of both SATs. If this happens, the perturbated SAT may lead to unboundedness if it is uncontrolled, even if it has no odd-order COMPs and has positive parameter.

We emphasize again that we only mean to find a sufficient but not necessary condition for mathematical boundedness. Some conditions defined above and related implementations may actually determine some models to be unbounded even if it is actually bounded. We consider this acceptable.

Following the conditions above, we can not classify the *selected* SATs (the SATs included in the fitting process) into the following categories (note that the SATs not selected by the fitting process are considered as if they do not exist):

1. `Bounded`. The SAT is uncontrolled, but has no odd-order COMP, and has positive parameter.
2. `Controlled`. The SAT is controlled (directly or indirectly) by at least one other SATs.
3. `PerturbativeAlloy`. The alloy-dependent SAT is a (direct or indirect) perturbation to at least one other SATs.
4. `ConstantType`. The SAT only has constant contribution to the potential energy, and does not have any local mode or strain DoFs. This is always true for the AlloyInter (AA) type SATs.
5. `NegativeTop`. The SAT is uncontrolled, has no odd-order COMP, but has negative parameter.
6. `NegativePerturbativeTop`. The SAT is uncontrolled, has no odd-order COMP, and has positive parameter for itself, but may have negative perturbated parameter for some possible occupation combination, yielded by the perturbative SATs for this SAT.
7. `OddOrderTop`. The SAT is uncontrolled and has odd-order COMP.
8. `NonPerturbativeAlloy`. The alloy-dependent SAT is neither controlled nor perturbative.

The first 4 categories are considered to be safe that do not lead to energy unboundedness. The last 4 categories, in contrast, may lead to energy unboundedness. The effective Hamiltonian with any SAT having these 4 types of boundedness types are considered unbounded. These categories may be reported during the bounding procedure for the unbounded SATs.

6.6.3 Closed bounding procedure

In this section, we introduce the close bounding procedure. The "closed" is named in the sense that the SATs are chosen and added in a *closed* set of pre-generated SATs (interaction set). Given such interaction set and the controlling/perturbation relations between the SATs, this closed bounding procedure itself is considerably easy to be implemented.

The closed bounding procedure requires the following inputs:

1. The interaction set and the SAT topology files (default: `heff_interactions.json` and `heff_interactions_topology.yaml`).
2. The initial parameter selection file (default: `param_selection_init.yaml`).
3. The linear fitting data file containing all FP data (default: `linear_data.json`).
4. The unit cell metadata file set (default: `heff_unitcell.yaml`, `unitcell_spec.toml`, and related LWF files).
5. The parameter template file and pre-calculated parameter file as used in the OTF fitting (default: `heff_params.yaml` and `heff_params_precalc.yaml`).
6. Fitting configuration file (default: `fit_config.toml`).

In the list above, the item 1 are obtained from interaction generation process (see *Generate interactions*). The item 2 is typically obtained from the MLCons process (see *Machine-learning-based model construction (MLCons)*). The item

3 is typically obtained from OTF fitting or merge-otf-fit, and must be the same as the input file with the same name for the MLCons procedure that generated the file in item 2. The items 4 and 5 are the same as the OTF fitting.

Important

All the input files must be consistent with each other. For example, the interaction generation process for the files in item 1 must use the same unit cell metadata as provided by the files in item 4 and 5; the MLCons must use the interaction set defined by the files in item 1 and the files in items 4 and 5; the linear fitting data file (item 3) must be consistent in the MLCons procedure and current procedure, etc. Any violation of such consistency rules will cause undefined behavior, and the program does not guarantee to report all possible violations.

The fitting configuration file may optionally include two tables called `boundness` and `bounding_closed`, determining the details for the boundedness detection and closed bounding procedure.

The closed bounding procedure consists of several *steps*. The maximal number of steps to be performed is determined by `bounding_closed.max_control_steps` (default: 1000). In each step, the boundedness of the effective Hamiltonian is checked, and the unbounded SATs with minimal order defined by `boundness.min_unbound_order` (default: 0) will be detected and tried to be solved. Before the first step, the input data set is split to the train set and the validation set (optional but recommended). The SAT adding, deletion and constrained fitting operations are performed based on the fitting and validation on these datasets. At the beginning, a relatively large interaction set is provided as input, and typically among them only a (small) part of SATs are selected (as determined by `param_selection_init.yaml` input file). During the bounding procedure, new SATs will be "added" to the effective Hamiltonian actually by "select" them. We emphasize that in the closed bounding procedure, no new SATs are *generated*; they are only selected from a closed range, that is the pre-generated interaction set.

Note

In principle, any unbounded SATs should be solved. However, in many cases, the low order SATs (for example, harmonic ones) may rarely cause real (physical) unboundedness problem. Thus, we provide the key `boundness.min_unbound_order` to ignore the low-order unbounded SATs that may help to reduce the computational efforts. If non-zero value is given, only SATs with order *greater than or equal to* this value will be considered as unbounded.

In each step, one of the following operations may be taken, to try to solve the unboundedness problem, until the model is considered to be bounded, or no further operation could be taken:

1. Try to do a constrained fitting if all *uncontrolled* SATs have no odd-order COMP and are not perturbated. Here, the "uncontrolled SATs" may be both unbounded or bounded (i.e., all-even-order SATs with positive parameters). In this constrained fitting process, the parameters for these uncontrolled SATs are constrained to have a positive parameter no less than a certain threshold, and the other parameters are solved by linear fitting or constrained optimization. For more information see *Constrained parameter fitting*.
2. Try to remove one unbounded SAT if `bounding_closed.allow_removing` is set to true. The removing is only accepted if the variance criterion (the same as in the MLCons, with the λ factor set by `bounding_closed.pow_lambda_deleting`) is met. Note that this deletion is an experimental feature and may cause problems like died loops.
3. Add one SAT according to controlling graph or perturbation graph. This is the most important and the most frequently used strategy in the bounding procedure. For more details, see *Adding bounding SATs*.

To run the closed bounding procedure directly, use the command like

```
python3 -m heff_gen.fit.cui.run_bounding_closed
```

When the bounding procedure finishes, the result will be reported on the `stdout`. If the bounding procedure finishes successfully with all problematic SATs solved, the following line will be shown:

```
INFO:cl-bnd:Closed-bounding successfully finished
```

Otherwise, if there are unsolvable SATs, the output will show a line similar to the following, followed by the final boundness report:

```
WARNING:cl-bnd:Bounding finished, but with unsolvable SATs: [67, 79, 80, 125, 237, 274,␣
↪299, 403, 472, 490, 510, 511, 720, 722, 723, 724, 725, 735, 765, 766, 768, 782, 817,␣
↪826]
```

The fitting RMSE for the final parameters in training set and validation set will also be shown, no matter there are unsolvable SATs or not.

After the bounding procedure, the resulting parameter will be written into the file (by default) `heff_params_bounded.yaml`, and the parameter selection file (in the same format as that output by MLCons) will be written into `param_selection_bounded.yaml`.

Constrained parameter fitting

In principle, we would like to obtain a set of well-bounded parameter from direct parameter fitting process by choosing SATs properly, which is just the purpose of adding bounding SATs in the bounding procedure. However, in practice, this condition may be rather difficult to be satisfied. In practice, the parameter values of some high-order SATs may be not so important to the fitting RMSE (i.e., changing their values does not lead to significant change of fitting RMSE). In this case, we may try to impose some extra constraint to the uncontrolled SATs to obtain a bounded model. Note that the parameter adjustment strategy based on similar idea was also proposed by second-principles method [SecondPrinciples2017].

In the bounding procedure, the constrained fitting is enabled if `allow_constrained_fitting` is set to true (this is the default). The constrained fitting will be tried if all the uncontrolled SATs are bounded or have the type of `NegativeTop`. Such condition is required because this is the only case that the constrained fitting will safely produce bounded model if it successes.

> **Note**
>
> By default, the perturbated SATs (with all-even-order COMPs) are not considered as unbounded, unless there are some occupation combinations lead to negative total parameter (see *Boundedness condition*). However, if SATs with this condition appear, the constrained fitting may be disabled. To solve this, it is possible to force the program to consider *all* the perturbated SATs as unbounded (and thus try to add SATs to control them), by setting `boundness.perturbated_as_unbound` to `true`.

The constrained fitting is currently performed by two possible implementations, as configured by `constrained_fitting_impl`:

- `scipy` (the default): run constrained fitting with `scipy.optimize` package. This requires `scipy` package (Python third-party package) to be installed. The parameter fitting is implemented by running optimization method configured by `constrained_fitting_scipy_method` (default: `SLSQP`), with extra options passed by `constrained_fitting_scipy_options` (default: empty). It is possible to use the result from the `fixed_linear` implementation (see below) as the initial solution, which may increase the convergence speed.
- `fixed_linear`: fix the parameter values of the uncontrolled SATs to their initial values, and run simple (free) linear regression for the rest of the parameters. This is typically much faster but may lead to larger RMSE.

In the `scipy` implementation, the lower bound of all constrained parameters are x_a times the absolute values of their fitted values in the last (linear) parameter fitting, where x_a is a factor defined by `constrained_fitting_param_lower_bound_scale` (default: 0.1).

In the `fixed_linear` implementation, the parameters for uncontrolled SATs will be fixed to its fitted value in the last fitting if such value is positive, or y_a times the absolute values of such value, where y_a is determined by `adjustment_param_scale` (default: 0.1).

The constrained fitting will be evaluated by the resulting variance change to determine whether it is acceptable or not. If the variance scales (the ratio between the new variance after constrained fitting and the old variance before the constrained fitting) in the training set and the validation set (if enabled) are *below* the threshold set by `adjustment_max_variance_scale` (default: 1.5), the result is accepted, and the `stdout` will show some line like:

```
INFO:cl-bnd:Accepted constrained-fitted parameters: variance: train 0.1279324765291714␣
↪validate 0.13275001566254785; scale: train 1.0615889697130538 validate 1.04830001088662
```

Otherwise, if the variance scales exceed the threshold, the adjustment is rejected, and `stdout` will show some line like:

```
INFO:cl-bnd:Rejected constrained-fitted parameters: variance: train 0.2942446960170942␣
↪validate 0.30532503602386; scale: train 2.4416546303400235 validate 2.411090025039226
```

If the constrained fitting is accepted, typically the bounding procedure will soon finish successfully. If it is rejected, another constrained fitting will be tried in later steps (typically after one bounding SAT is added).

Adding bounding SATs

As mentioned, adding SAT is the most important strategy for the bounding procedure. In the bounding procedure, only one SAT is added in one bounding step. The added SAT is selected from the SATs controlling the currently unbounded SATs, or being perturbated by the currently unbounded SATs (`NonPerturbativeAlloy` type).

The SAT for adding is selected in a two-step manner. In the first step, a subset of SATs is selected, mainly based on the order and relations with unbounded SATs. In the second step, an SAT is chosen from this subset, mainly based on fitting variance. We always try to add a new SAT if required, unless there is no optional SAT, or the addition of new SAT causes too much increasing of variance. The latter is determined by the ratio between the variance after and before the adding (in validation set if available, otherwise training set), and the addition is rejected if such ratio exceeds `adding_max_variance_scale` (default: 1.3).

In the first step, the subset is chosen from all optional SATs such that:

- all SATs already removed are not considered;
- all SATs leading to linearly dependent model are not considered (unless `reject_all_linearly_dependent_model` is set to false);
- if `constrain_lowest_order` is set to true (the default), only the SATs with the lowest possible order are chosen.

Furthermore, if `faster_adjustment` is set to true (the default), the following filtering is conducted, in order to reach the accepted constrained-fitting as fast as possible (from a somewhat "greedy strategy" point of view):

- if there are any unbounded SATs with types *other than* `NegativeTop`, only the SATs that controlling these unbounded SATs are considered;
- if all the unbounded SATs have the `NegativeTop` type, the program tries to *adjust* the parameters into positive one by one, and compute the resulting variance increasing. The unbounded SATs with the maximal resulting variance are identified, and SATs controlling these unbounded SATs are considered first.

In the second step, the optional SATs in the selected subset in the first step are evaluated by trying to add them to the selected SATs one by one and compute the resulting variance (in both training set and validation set if available). If the resulting regularized variance is reduced (the same as the condition for adding SATs in MLCons process, see *Machine-learning-based model construction (MLCons)*, where the λ factor is defined by `pow_lambda_adding`, default to 1.0), the SAT is accepted immediately and added to the interaction set. If there is no optional SATs satisfying this condition, the SAT with the lowest resulting variance (in validation set if available, otherwise training set) is selected. If such variance does not exceed the threshold set by `adding_max_variance_scale`, the corresponding SAT is accepted and added to the interaction set.

Once a new SAT is added, the added SAT information is presented in `stdout`, along with the variance data:

```
INFO:cl-bnd:Add SAT (minimal variance): ControllingSATInfo[sat_id=188, order=6,␣
↪controlled_sats=[52] (count=1)]
INFO:cl-bnd:Variance: train 0.12227382723306668 validate 0.12052840725229087
```

It is strongly suggested to watch the change of variance during the bounding procedure.

If there is not suitable SAT for adding (all optional SATs lead to linearly dependent model, or the SAT with the minimal updated variance exceeds the maximal allowed variance threshold), no SAT will be added in this step. If such circumstance repeated for two steps continuously, the program will determine there is not optional SAT for adding and stop.

In practice, the SAT adding may be conducted using an "incremental" approach, if `incremental_adding` is set to true (the default). In this case, the variance after adding a new SAT is estimated using an incremental algorithm, without solving the whole model, which is faster. It should be noted that, with this incremental adding approach, all SATs leading to linearly dependent model are rejected (equivalent to set `reject_all_linearly_dependent_model` to true).

6.6.4 Open bounding procedure

As mentioned, the open bounding procedure will generate new SATs to solve the unbounded SATs when needed. To implement this, the open bounding procedure could be seen as a combination of incremental interaction generation process, merge-otf fitting process, and embed closed bounding procedure. As a result of this implementation strategy, the `bounding_closed`, `boundness`, and `merge_otf` tables in the input configuration file are used, along with the `bounding_open` table dedicated for this procedure.

The open bounding procedure is made up of several *iterations*. In each iteration, the program determine whether new SATs are needed. If so, new SATs are generated, and `merge_otf_fit` is run to load the new data. After that, an embed closed bounding procedure with limited order and steps (configured by `bounding_open.max_closed_steps`) is executed. The following figure shows the flowchart of the open bounding procedure. It is clear to see that the bounding problem is actually solved by the embed closed bounding procedure, using the same algorithm as introduced in *Closed bounding procedure*, but with limited order (for the bounding SATs to be added) and number of steps.

The open bounding procedure always tries to find SATs to solve the problematic SATs with *minimal possible orders*. For this purpose, the program estimates the minimal possible controlling SAT order at the beginning of each iteration, and (by default) limit that all SATs added in this iteration must have an order no greater than this estimated minimal order. Note that one could also turn off such limitation by setting `bounding_open.iter_by_order` to false. We should note that such *estimation* to the minimal possible controlling order is only a prediction and may be incorrect or improper. At first, the program estimate the minimal possible controlling order for each problematic SAT by analyzing its form. This result may be incorrect, as the controlling SATs with such order may be invalid, or cannot control the given problematic SAT. The program will fix such incorrect predictions after generating new controlling SATs. However, in some cases, some SATs may lead to linearly dependent model, or have zero contribution to the Heff, and should be discarded. In such cases, the estimated order cannot be correctly fixed in current implementation. To solve this problem, we choose to increase the minimal order forcefully if there are two continuous iterations with no accepted updating.

The input to the open bounding procedure includes:

1. All the input data for the closed bounding procedure (see *Closed bounding procedure*), including but not limited to the linear fitting data, the model metadata, initial selection, etc.
2. Configuration to the merge-otf fitting, especially the data directories specification.

Important

The given data directories specification to the merge-otf fitting must be consistent with the given linear fitting data (`linear_data.json`). Typically, the linear data is directly generated by the `merge_otf_fit` program

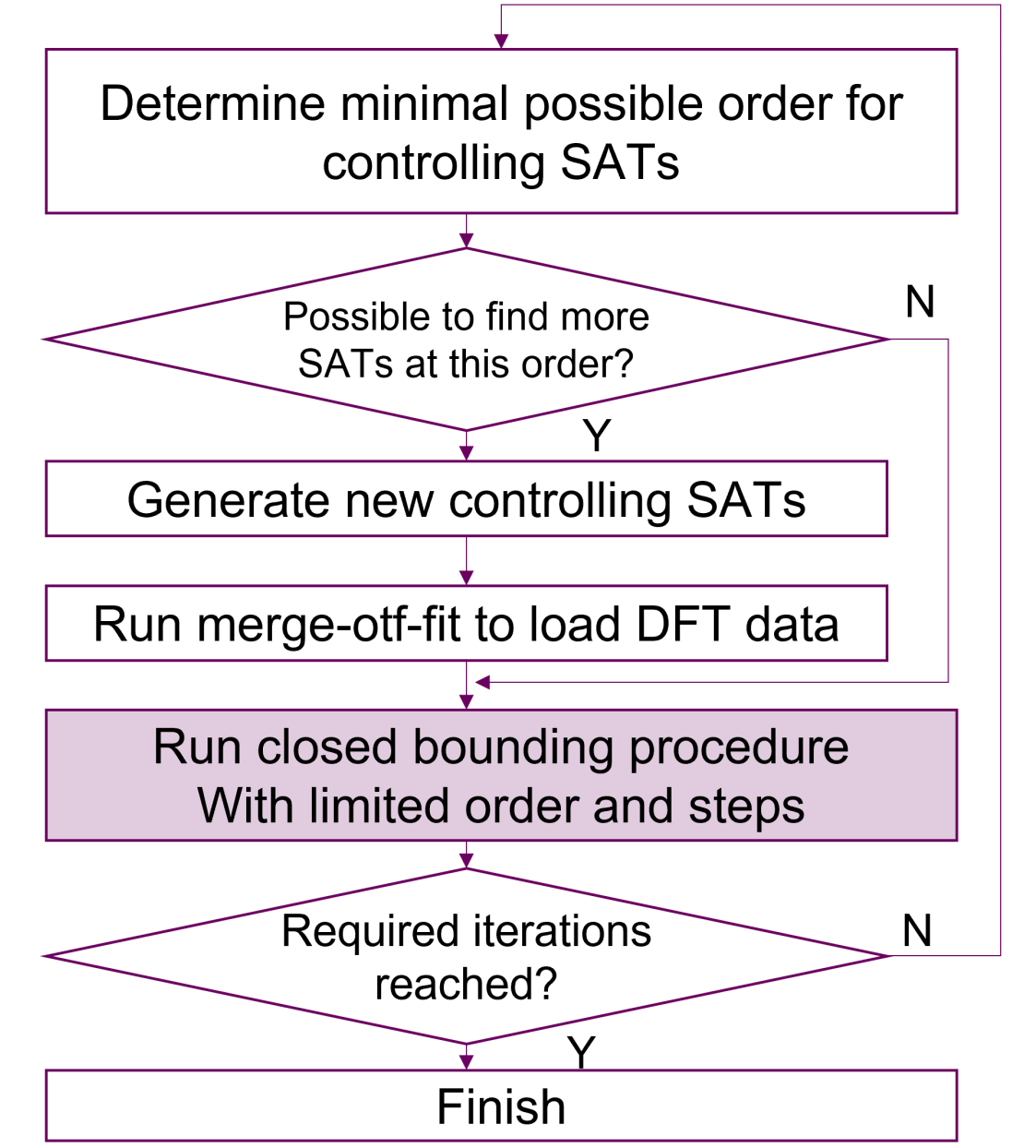


Fig. 1: Open bounding procedure flowchart

(*merge_otf_fit*) with the same configuration, or copied from the active fitting output data `active_fit_out.json` with the source data of merge-otf directories specification and in exactly the same order.

Important

We emphasize again the importance of the input data consistency, including but not limited to the consistency between interaction data file, interaction topology file, parameter file, unit cell specification files, linear fitting data file. Any violation to this consistency rule will cause **undefined behavior** and the program does not guarantee to detect it.

With all the input data properly prepared, run the command like

```
python3 -m heff_gen.fit.cui.run_bounding_open
```

To execute the open bounding procedure.

During the execution, the program generates some logging information, most of them come from the merge-otf fitting and the embed closed bounding procedure. These two parts also consume most of the running time of this program. In each iteration, if the program determines new SATs are needed, it will first try to compress the interaction set by removing all unused SATs, if the proportion of unused SATs is larger than `max_unused_param_proportion` (default: 0.2). After that, new SATs are added, and the interaction set is updated. Once interaction set is updated, a new directory called `model_iter_<iter_id>` (by default) will be generated, where `<iter_id>` is the iteration index (0-based). The interaction data file, topology file, and related LaTeX report files are written into this directory. The merge-otf data file `merge_otf_data.json` is also written in this directory (in an asynchronous manner). At the end of each iteration, the parameter files are also written into the latest model directory, along with the parameter selection file.

Note

If interaction set is not updated in one iteration, no new model directory is generated. As a result, the `model_iter_<iter_id>` directory may not be generated for *all* iterations.

The open bounding procedure is finished when:

1. the maximal allowed number of iterations (`bounding_open.max_iters`) is finished,
2. the embed closed bounding procedure report a successful status, or
3. the SATs with higher order than the maximal allowed order (`bounding_open.max_controlling_order`) is required.

In the second case, the open bounding procedure is successfully finished, with all problematic SATs solved. In the other two cases, the bounding procedure is not successfully finished, with some SATs unsolved. Related information is provided in the log to `stdout`.

As the open bounding procedure finishes, two extra directories are also created, they are named (by default) `model_final` and `model_compressed`, containing the information for the final model (interaction set data and their topology, parameter, and parameter selection, etc). The difference between the two directory is that the data in `model_final` is exactly the same as the data used in the last iteration, while the data in `model_compressed` is further compressed with unused SATs removed. Furthermore, merge-otf data is also computed and written for the compressed final model. For further simulation or continuing OTF fitting purposes, we recommend to use the compressed data, which may have better performance.

Each model directory (including the `model_iter_<iter_id>` and the `model_final`, `model_compressed` directories) contains the following output files:

- Interaction data file `heff_interactions.json`, and topology file `heff_interactions_topology.yaml`. In the same format as the files generated by `heff_gen_interact` (*Generate interactions*).
- Topology graphviz file `control_graph.gv` and `perturbation_graph.gv`, also in the same format as the files generated by `heff_gen_interact`.
- `interactions_brief.tex` and `interactions_full.tex`. LaTeX report file for the interaction set in this model. The difference between the two files is that the former contains only the first PROD in each SAT while the latter contains all PRODs. The formats have some differences from the files generated by `heff_gen_interact` because of different generation implementation.
- Linear fitting data file `merge_otf_data.json` (not available for the `model_final` directory). This is equivalent to the output file generated by `merge_otf_fit` with the current model as input. This could be used as active fitting input file (`active_fit_in.json`) in OTF fitting. But be aware, as in normal `merge_otf_fit` (*merge_otf_fit*), this file contains *no* uncertainty history information, and may cause problem if `thres_strategy` is set to `Dynamic` in the OTF fitting loading this file.
- Parameter selection file `param_selection_bounded.yaml`. In the same format as the file generated by the MLCons process (*Machine-learning-based model construction (MLCons)*).
- Bounded parameter file `heff_params_bounded.yaml` and direct fitted parameter file `heff_params_direct_fit.yaml`. Both files have the same format and could be used for further fitting or simulation. The difference is that, if the constrained-fitting is performed (*Constrained parameter fitting*), the `heff_params_bounded.yaml` will contain the parameter produced by the constrained fitting, while `heff_params_direct_fit.yaml` will contain the parameter produced by direct least-square fitting.

Tip

In practice, we find that sometimes the direct-fitted parameter file has better performance than the bounded file. We suggest to test both files for simulation.

6.7 Practical fitting workflows

Here we try to introduce some *practical* fitting workflows for building an Heff model. The workflows starts *after* pre-calculations (*Precalculation*), and is a combination of OTF fitting (*On-the-fly learning*), MLCons (*Machine-learning-based model construction (MLCons)*), and open bounding procedures (*Open bounding procedure*).

Important

The part introduced here is highly experimental. They are just summarized from our preliminary practices for building models. They are provided *only for reference*. We do *not* mean to declare these strategies as any *standard* practice. And the document about these approaches may be changed at any time.

6.7.1 OTF-bounding workflows (Nov2025)

In this presented workflow, the OTF fitting and open bounding procedure are used in combination. This workflow was first proposed in November, 2025. The following figure briefly shows the workflow:

The workflow starts *after* the pre-calculation (*Precalculation*) and interaction generation (*Generate interactions*).

Step 1. The workflow starts with *a series of* OTF fittings, mainly with low temperatures. The first OTF fitting (*Configuring the first OTF MD run*) typically uses a small supercell, and the following continuing OTF fittings (*Continuing the OTF fitting*) may use different supercell sizes and shapes, in order to fit interactions with different distances. Multi-OTF fitting (*Multi-OTF fitting*) may also be used.

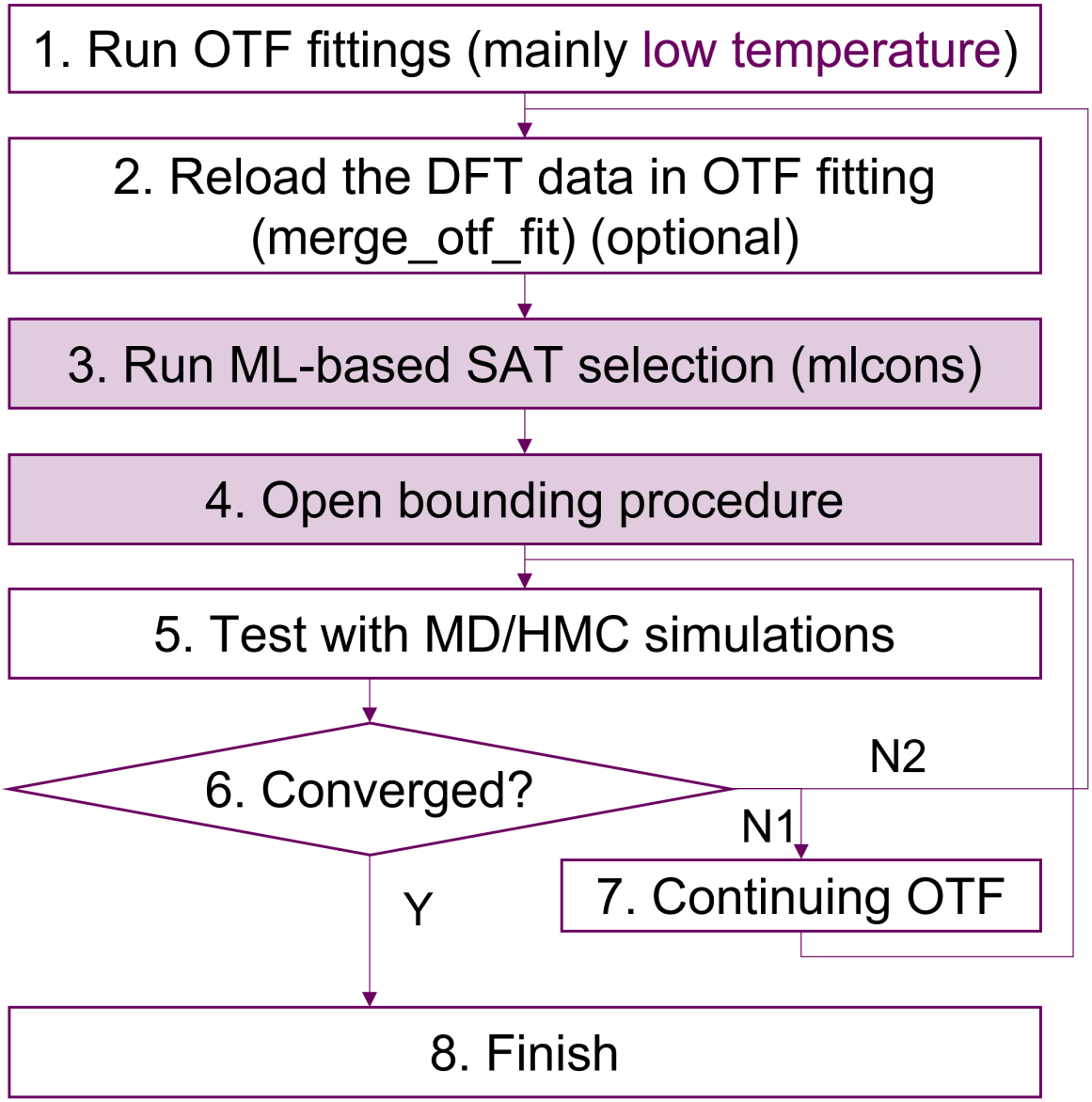


Fig. 2: Schematics for OTF-bounding workflow (Nov2025)

This series of OTF fitting could be completed when all the parameters in the interaction set are properly fitted. This could be checked by running a continuing fitting or `fit_form_data` (*fit_from_data*) with `inf_sigw = true`. If there is *no* warning telling that the model is not invertible, it could be determined that all parameters are fitted. Another method is to simply check the output file `heff_params_fit.yaml` and find if there are zero parameters. If there are parameters with exactly zero values, it typically indicate that these parameters are not properly fitted, and different or larger supercells should be used to fit them.

Note

We recommend to mainly use low temperature in this stage, because we find that sometimes using too much data with high temperature will reduce the accuracy for the fitting of some parameters, especially at low temperature range. Such reduction of accuracy in low temperature is undesired. This low-temperature-only fitting in this stage could likely cause divergent model at higher temperature. This will be solved later.

Step 2. Reload all data with `merge_otf_fit` (*merge_otf_fit*). This step is completely optional but we recommend to do so, in order to keep ensure the data consistency in later open-bounding procedure. In this step, provide all data generated in the previous step.

Step 3. Run MLCons (*Machine-learning-based model construction (MLCons)*). In this step, several SATs (parameters) with significant contribution to the Hamiltonian are selected out of the originally generated ones.

Step 4. Run the open bounding procedure (*Open bounding procedure*) for the first time. If this finishes successfully, a mathematically bounded model will be generated.

Step 5 and 6. Test with MD or HMC simulations with the model generated from step 4, and check convergency. Remember to copy *all* the related files in the output model directory (default: `model_compressed`). In this step, we suggest to run several different tests with different settings, including different supercell sizes, temperatures, initial pseudo random number generator seed (`init.seed`) etc. Here the "convergency" includes not only the mathematical convergency, but also the physical convergency. Any unreasonable energy values (e.g., very large positive or negative values), or unreasonable DoF configurations, should be considered as not converged. If this happens, go to step 7 (path N1) or step 2 (path N2). Otherwise, go to step 8 (finish).

The choice between path N1 and path N2 is determined by whether the training data has changed significantly (i.e., whether a relatively large amount of data is collected) since the last open bounding process. If so, we suggest to go to step 2 and rerun merge-otf, MLCons, and open bounding procedure, to update the interaction set. Otherwise, one can directly go to step 7. Clearly, for the first time this point is reached after last open bounding (step 4), one always choose path N1 (go to step 7).

Step 7. Continuing OTF fitting with new model. Choose a *small* supercell (that is processable by FP calculations) and simulation configurations that lead to the unreasonable results, and run continuing OTF fitting with this configuration. This OTF fitting could be executed with low temperature or high temperature, depending on the tested configurations. This OTF fitting should use continuing OTF fitting, loading all the data generated by the previous OTF fittings (see also *Continuing the OTF fitting*). The new model generated in the bounding procedure (typically in `model_compressed`) should be used. Remember to copy all the related metadata files (interaction data file, parameter file, parameter selection file, etc) to the new OTF fitting directory, and set the `regression.param_selection_yaml` field to the file name of the parameter selection file from the bounding procedure (`param_selection_bounded.yaml` by default). The continuing fitting data file (`active_fit_in.json` by default) could be copied from the merge-otf data file (`merge_otf_data.json`).

Warning

Remember to copy all the metadata files above and set the related setting tags. Make sure the interaction set data file (`heff_interactions.json`), parameter file (`heff_params.yaml`), parameter selection file (`param_selection_bounded.yaml`), and linear fitting data file (`active_fit_in.json`) are all consistent with

each other. Otherwise the behavior is undefined!

In this OTF fitting, we suggest to use a (so-called) **sparse** strategy to reduce the number of required FP calculations, especially when relatively high temperature is applied. Generally, the sparse strategy could be realized using different settings:

- Use a fixed threshold (`thres_strategy = "Fixed"`) with a relatively large threshold value (`thres_value`). This threshold value could be chosen according to the uncertainty history in previous OTF runs. We suggest to use a value that is about two or three times of the stable uncertainty values in previous OTF fittings (use `otf_fit` to visualize this).
- Alternatively, use a dynamic threshold (`thres_strategy = "Dynamic"`) but with a large updating scale (`thres_update_scale`), up to 2 or 3.

This OTF fitting should trigger at least one FP calculation and improve the convergency of the model about the given configuration.

Tip

If the FP calculation is not triggered in the OTF fitting, try to provide initial structure using `init.load_dof_h5`. The initial structure could be obtained from previous simulation tests. We suggest to take the structures where unreasonable structure or energy is *nearly* reached.

After this continuing OTF fitting, one can test again the model convergence (go to step 5). It should be noted that the parameters directly fitted from continuing OTF fitting are typically not mathematically converged. However, we sometimes find such parameters provide good *physical* result (i.e., physically converged or nearly converged), and could be used in the simulations. One could also do another bounding procedure (step 4) after the OTF fitting to get a mathematically converged parameter set. But that parameter set may not be suitable, so always do tests before using it in the productive simulations.

The workflow above seems to be complex. But it may be understood in another way that may be more clear. At the beginning (just after interaction generation and precalculation), one run the OTF fitting with low temperature, run merge-otf fitting, MLCons, and open bounding (steps 1 to 4), to get the first set of interaction that is *possible* to be *mathematically* converged. After that, one choose the configurations (supercell, temperature, initial seed, etc) leading to unreasonable result and run continuing OTF fittings with these configurations iteratively, until the parameter set is considered to be converged. During this stage, the interaction set may be updated by doing another series of merge-otf fitting, MLCons, and open bounding procedures (steps 2 to 4) when the data set changed significantly since last open bounding procedure.

CHAPTER

SEVEN

THEORETICAL BACKGROUND

In this document we will introduce some theoretical background about the effective Hamiltonian and implementation in this package. We do not mean to provide a complete explanation of the whole building and simulation process (because this may be too verbose), but just provide some aspects that are related to the practical usage of this package. We think they have close connection to the usage and understanding of this package. For more information, we recommend to read the article [ALFE-H-2025] and references therein.

This document will be organized based *topics*. These topics may be related to the model building or simulation, and the connection between different topics may be not significant.

7.1 Alloy model

The effective Hamiltonian with *mixed sites* are called *alloy model* in this package and this document. These models typically represent the solid solutions, like (Ba,Sr)TiO_3(BST), (Al,Sc)N, etc. Here, the *mixed sites* contain multiple possible elements sitting at the same site in the atomic structure (crystal structure). For example, in the BST model, the mixed site, which shall be named `A` (the name could be assigned by the user in practice), may be occupied by Ba or Sr atoms, in different unit cells of the simulated supercell. Such occupation may be completely random, or have some pre-defined order (like super-lattice etc). For some practical setting of the mixed sites, please refer to *Select reference structure*.

Inside the program, the *occupation* of the mixed sites are represented by a set of integer-valued variables, which is often called σ in previous articles (see [ALFE-H-2025] and references therein). In this package, the occupation of different elements are represented by integers starting from 1. For example, in BST, the Ba and Sr may be represented by 1 and 2, respectively. The value 0 is reserved for vacuum sites, and the value -1 is reserved for non-alloy sites (the sites with only one possible element, like the Ti site in BST). Such integer variable are assigned for *every* mixed site in the supercell, and are sometimes called *occupation variables*. These occupation variables are written in the output file in simulations (default name: `out_sites.txt`), and are also used in the input file while using `init.read_sites_txt`.

7.1.1 Interaction generation with mixed sites

During the interaction generation stage, the occupation variables take part in the products (PRODs) as components (COMPs), as introduced in *Generate interactions*. This type of alloy COMP or occupation COMP holds the information of the mixed site index (name) and the actual position of the site to be described (unit cell position and site index inside the unit cell). During interaction generation, the alloy COMPs are included in the PRODs like multipliers. However, as we will introduce below, the actual energy (and force, stress) computation of these PRODs including alloy COMPs are *not* performed by directly multiply the occupation variable with other COMPs (multiplying these integer occupation values 1, 2, 3, … with other components is obviously not a reasonable way). As will be shown below, multiple alloy COMPs at the same site in one PROD is meaningless, and this case is not allowed in the interaction generation stage.

In the LaTeX output files (`interactions_report.tex`, `interactions_brief.tex` etc), the alloy COMPs are written as if they are normal multipliers of the PRODs, although it is not computed in this way. For example, if u is a local mode name centering on the site called `Ti1`, A is an occupation variable centering on the site called `Ba1`, we may get

a PROD in the LaTeX output file like

$$u_1(Ti1;[0,0,0])A(Ba1;[0,0,1]),$$

which is formed by the first axis of the local mode variable u, centering at the $Ti1$ site at the unit cell `[0, 0, 0]`, and the mixed occupation of A, at the site called `Ba1`, and the unit cell position `[0, 0, 1]` (i.e., one unit cell away from the original point along the third basis vector of the unit cell).

7.1.2 Computation of occupation components

To understand the energy computation of PRODs with alloy COMPs, we need to trace back to the starting point, that is, the virtual crystal approximation (VCA) [VCA2000]. In VCA, the different elements at the same site will be treated as average atoms ("virtual" atoms) of the same type, containing the (somehow) averaged influence from different elements. In first-principles (FP) level, this is accomplished by averaging their pseudo-potentials. In classic effective Hamiltonian works with alloy models (see, e.g., [PZT_Heff_2000], [PST_Heff_2022], etc), the effective Hamiltonian is split into two parts:

$$E_{\text{tot}} = E_{\text{ave}}(\{s\},\eta) + E_{\text{loc}}(\{s\},\{\sigma\}),$$

where E_{ave} contains the "averaged" contribution of the mixed site atoms, which takes the same form as the Heff models without mixed sites, and E_{loc} contains the local perturbation to the former part due to the atomic occupation of the mixed sites. Here, s represents all the local mode variables, η is the (homogeneous) strain tensor, and σ represents all the occupation variables. As the latter part is only a perturbation, it typically includes only some simple terms.

In these classic works, both parts are parametrized explicitly with the FP calculations applying VCA. This way is rather straightforward, but sometimes the accuracy of VCA may be doubtable, especially for the mixed elements with great differences.

In this package, we use the same perturbation approach only *conceptually*. We also use the idea of VCA to derive the forms of the interactions, but VCA is not actually used in the FP calculations. It can be seen as the E_{ave} is not explicitly parameterized, but the whole E_{tot} is directly parameterized using the OTF fitting with several atomic occupations.

The energy contribution of the PRODs without alloy components are computed directly by multiplying the parameter value with the components. For example, if we have a product

$$P_1 = u_1^2(Ti1;[0,0,0]),$$

with parameter α_1, the energy contribution will just be

$$E(P_1) = \alpha_1 u_1^2(Ti1;[0,0,0]).$$

If the parameter $\alpha_1 = 0.15$, the local mode u at the `Ti1` site of `[0, 0, 0]` unit cell has the value 0.08, the energy will simply be $E(P_1) = 0.15 \times 0.08^2$.

For energy contribution of PRODs with alloy components, things are different. As derived from the perturbation idea, and also consistent with the previous works, we use a strategy that may be called "parameter selection". Suppose we have a product

$$P_2 = u_1(Ti1;[0,0,0])A(Ba1,[0,0,1]),$$

the parameter α_2 will **not be a scalar**, but have several values, for different occupations. The energy will be computed as

$$E(P_2) = \begin{cases} \alpha_2(1)u_1(Ti1;[0,0,0]), & A(Ba1,[0,0,1]) = 1, \\ \alpha_2(2)u_1(Ti1;[0,0,0]), & A(Ba1,[0,0,1]) = 2. \end{cases}$$

Here, $\alpha_2(1)$ is the value of the parameter α_2 for the occupation value 1 (i.e., the corresponding site occupied by Ba atom in this case), and $\alpha_2(2)$ is such value for occupation value 2 (the corresponding site occupied by Sr atom in this case).

The $\alpha_2(1)$ and $\alpha_2(2)$ are the parameters needed to be obtained from the parameter fitting process with FP calculations. Clearly, the *dimension* of the parameter α_2 is just the number of different possible elements for the mixed site A.

For PRODs with more than one alloy COMPs (at different sites), the computation is similar, but more complex. For example, for the PROD:

$$P_3 = u_1(Ti1; [0,0,0])A(Ba1, [0,0,0])A(Ba1, [0,0,1])$$

having two alloy COMPs at two different sites, the parameter α_3 will be a rank-2 tensor indexed by the two occupation indexes. The corresponding energy computation is implemented as

$$E(P_3) = \begin{cases} \alpha_3(1,1)u_1(Ti1; [0,0,0]), & A(Ba1, [0,0,0]) = 1 \text{ and } A(Ba1, [0,0,1]) = 1, \\ \alpha_3(1,2)u_1(Ti1; [0,0,0]), & A(Ba1, [0,0,0]) = 1 \text{ and } A(Ba1, [0,0,1]) = 2, \\ \alpha_3(2,1)u_1(Ti1; [0,0,0]), & A(Ba1, [0,0,0]) = 2 \text{ and } A(Ba1, [0,0,1]) = 1, \\ \alpha_3(2,2)u_1(Ti1; [0,0,0]), & A(Ba1, [0,0,0]) = 2 \text{ and } A(Ba1, [0,0,1]) = 2. \end{cases}$$

Now, for this case, there are four different parameters $\alpha_3(1,1)$ etc to be fitted.

Alloy parameter summation rule

We have introduced that the parameters for the alloy-dependent SATs have multiple components, as shown by the examples of P_2 and P_3 above. However, not all of these parameters are independent. Actually, for an alloy-dependent SAT with alloy COMPs $A_1, A_2, ...$ having $c_1, c_2, ...$ possible different element types, the independent parameter count is $(c_1 - 1)(c_2 - 1)....$ Thus, for the P_2 and P_3 examples above, the number of independent parameters are both only 1. There are several *summation rules* imposed between the parameters of the same alloy-dependent SAT to accomplish these reduced count of independent parameters. For SATs with only one alloy COMP in each PROD, the summation rule is

$$\sum_c \alpha(c) = 0,$$

where α is the parameter, c is the possible occupation values (1, 2, 3, ...). For deviation of this summation rule, please see remarks below. For SATs with more than one alloy COMPs in each PROD, similar summation rule is applied to each of the parameter index positions. For example, for the SATs with two possible occupation COMPs, the summation rules are

$$\sum_{c_1} \alpha(c_1, c_2) = 0, \ (\forall c_2),$$
$$\sum_{c_2} \alpha(c_1, c_2) = 0, \ (\forall c_1).$$

During the parameter fitting procedure, the program will apply similar summation rules to the energy, force and stress contributed by these alloy-dependent SATs to ensure the fulfillment of these summation rules, and only independent parameters take part in the parameter fitting regression process.

Deviation of the summation rules

For simplicity, we will only show the deviation of the summation rules for the SATs with only one alloy COMP in one PROD. For SATs with more alloy COMPs, the deviation is straightforward by considering another SAT (possibly valid or not) with one alloy COMP removed as the perturbated SAT (see below).

For an alloy-dependent SAT S_1 with one alloy COMP in each PROD, we consider another SAT (possibly valid or not) S_0 with all alloy COMPs removed from the PRODs. Here, S_0 is called the *perturbated SAT* by S_1, using the terminology defined in *Relations between SATs*. It should be stressed that the "SAT" S_0 may be a valid SAT (actual SAT permitted by symmetry, and could also be derived by the program), or an invalid SAT (strictly speaking this is not an SAT because it does not meet the symmetry requirement) that is forbidden by symmetry. An invalid SAT always provide zero values to the energy contribution by definition.

The energy contribution by S_1 could be written as (see above)

$$E(S_1) = \sum_i \alpha(A_i) P_i,$$

where P_i is the PROD inside the SAT S_1, A_i is the occupation variable for the PROD P_i, and $\alpha(A_i)$ is the parameter for this PROD.

Let us now rewrite the parameter $\alpha(A)$ as

$$\alpha(A) = \alpha_a + \tilde{\alpha}(A),$$

where α_a is a constant that does not dependent on occupation value, $\tilde{\alpha}(A)$ is the part that satisfy the summation rule defined above:

$$\sum_c \tilde{\alpha}(c) = 0.$$

Note that such decomposition is always possible because it includes only linear computation with scalars. We can now rewrite the energy contribution as

$$E(S_1) = \sum_i \alpha_a P_i + \sum_i \tilde{\alpha}(A_i) P_i.$$

It is clear to find that the first term of the right side in the equation above is just the energy contribution from the perturbated SAT S_0.

Case 1, the SAT S_0 is invalid. In this case, its energy contribution must be zero, and the parameter must also be zero, thus $\alpha_a = 0$.

Case 2, the SAT S_0 is valid. Then the term led by α_a is the same as the contribution of SAT S_0. In this case, we assume the program will find the SAT S_0 from other initial configurations, and it is no need to be covered by the SAT S_1, and we can safely set $\alpha_a = 0$.

Note

From the perturbation point of view, as the S_1 is a perturbation to S_0, it is physically unreasonable to exclude S_0 from the interaction set while include only S_1, if S_0 is a valid SAT. If the initial configuration of interaction generation lead to this result, the configuration itself should be considered unreasonable. If open bounding procedure is used, the program may add the SAT S_0 in this case if needed.

Thus we have shown that in both cases we can set the non-occupation-dependent part of the parameter $\alpha_a = 0$. This is exactly equivalent to force the parameters to satisfy the summation rules.

Summation rules in the old implementations

It is worth noting that in the old implementation of the effective Hamiltonian method, including our previous work for perovskites and earlier works (see [ALFE-H-2025] and references therein), a different form of summation rule is used. Such form was introduced in the Sec. V of Supplementary Information of [ALFE-H-2025]. Using the symbols in this document, such summation rule could be written as

$$\sum_c \alpha(c) x(c) = 0,$$

where c is the occupation integer value, $x(c)$ is the concentration of component c, and obviously we have $\sum_c x(c) = 1$. Such summation rule is derived from the perturbation of the DFT functional.

In this package, we recommend to use the summation rule with the new form as introduced above, because this form does not dependent on the concentration of elements. If one would like to use the old form of summation rule, set `pader.concentration_dependent_spring_folding` to `true` in the configuration file while fitting parameters.

7.2 Electrical conditions

Note

The features described in this section is available as of version 0.26.0 (Mar. 16, 2026).

Warning

All the features described by this section are highly experimental.

In the previous effective Hamiltonian works (see, e.g., [PST_Heff_2022]), the external electric field could be imposed by adding an extra force on the dipolar local modes. This is equivalent to adding an electric-field-related term in the potential function. Physically, the *enthalpy* that determining the thermodynamics distribution is computed as

$$H = E_{\text{eff}} - \Omega \vec{P} \cdot \vec{E},$$

where E_{eff} is the energy defined by the effective Hamiltonian, including both long-range and short-range interactions (and also kinetic energy when required); Ω is the supercell volume; $\vec{P}$ is the polarization of the simulated supercell; and $\vec{E}$ is the electric field.

Such definition and treatment, while rather natural and simple, is not the only possible choice. As detailed in [DispField2009], the authors proposed electrical displacement field $\vec{D}$ along with its *reduced* version as another possible variable to be controlled. In contrast, the definition above corresponds to the condition where the electric field $\vec{E}$ is fixed. In our implementation, both fixing electric field or displacement field or their reduced versions, as well as mixing reduced displacement field and electric field in different surfaces, are supported. The input setting is somehow similar to that Abinit package (see, e.g., variables `berryopt`, `dfield`, `red_dfield`, `jfielddir`, etc, in Abinit), making it easier for users familiar with Abinit. The choice between these controlled variables will be referred to as *electrical condition* hence. Here, we introduce these conditions in more detail, especially with more implementation details.

We have implemented five possible electrical conditions. Such conditions are selected using the `electrical_condition` key in the `physical` table of the input configuration file, using the following options:

- `electrical_condition = "FixE"`. Fixed electric field $\vec{E}$ condition. This is the default, and the condition used by most of previous effective Hamiltonian works.
- `electrical_condition = "FixD"`. Fixed displacement field $\vec{D}$ condition. Corresponding to open circuit boundary condition.
- `electrical_condition = "FixRedEbar"`. Fixed reduced electric field $\bar{e}$ ($\bar{\varepsilon}$ in [DispField2009]).
- `electrical_condition = "FixRedD"`. Fixed reduced displacement field d.
- `electrical_condition = "FixRedMix"`. Fixed reduced field with mixed boundary conditions. The directions are specified using the `red_field_direction` field, in which the element 1 specifies reduced electric field ($\bar{e}$) is controlled in the corresponding surface, and 2 for reduce displacement field (d).

7.2.1 `FixE` condition

In this condition, the macroscopic electric field $\vec{E}$ is controlled as external condition. The enthalpy defined as

$$H = E_{\text{eff}} - \Omega \vec{P} \cdot \vec{E}$$

will be used in the thermodynamics formulas. This is consistent with most of previous effective Hamiltonian works.

7.2.2 `FixD` condition

In this condition, the macroscopic displacement field $\vec{D}$ is controlled as external condition. The displacement field is defined as

$$\vec{D} = \epsilon_0 \vec{E} + \vec{P},$$

where ϵ_0 is the vacuum permittivity. In this package, the displacement field is specified using the `physical.disp_field_homo` key with the unit `e / Angstrom^2`.

In this condition, the internal energy defined as

$$U = E_{\text{eff}} + \frac{1}{2\epsilon_0}\Omega(\vec{D} - \vec{P})^2$$

is used in the thermodynamics formulas.

> **Warning**
>
> In current implementation, the derivative of the supercell volume Ω over the strain tensor η is ignored when computing the energy derivative over the strain (i.e., the stress). Such approximation may affect accuracy if the strain changes significantly. We always recommend to use the reduced version (`FixRedD`) based on this implementation issue, as well as the reason described in detail in [DispField2009].

7.2.3 `FixRedD` condition

The *reduced* displacement field is defines as

$$d_i = \Omega \vec{b}_i \cdot \vec{D},$$

where $i = 1, 2, 3$ is the crystal axis index, the vector $\vec{b}_i$ satisfies $\vec{a}_i \cdot \vec{b}_j = \delta_{ij}$ where $\vec{a}_i$ is the supercell lattice vector. As discussed in [DispField2009], the fixed reduced displacement field corresponds to the condition where the surface free charge is rigorously conserved under strain (where fixing the macroscopic displacement field $\vec{D}$ does not guarantee exactly). Moreover, as for the implementation, the derivative of the energy over the strain is exactly considered, including Maxwell contribution and the augmented part (the latter is arisen by the dependency of the vector $\vec{b}_i$ on the strain), making the implementation rather reliable. We thus recommend to use this condition rather than `FixD` condition.

In practice, the d_i is provided in the input by the `red_disp_field` key with an array of three floating-point numbers. The unit is e ($\approx 1.602 \times 10^{-19}$ C). The `disp_field_scale` is applied to `red_disp_field` in this case.

7.2.4 `FixRedEbar` condition

In this condition, the *reduced* electric field defined as

$$\bar{e}_i = \vec{a}_i \cdot \vec{E}$$

is controlled as external condition. In this condition, the potential drop on the surfaces is conserved. The enthalpy, which shall be noted as H_1, is used in the thermodynamics formula:

$$H_1 = E_{\text{eff}} - \Omega \vec{P} \cdot \vec{E} + \frac{1}{2}\epsilon_0 \Omega E^2,$$

where the macroscopic electric field is computed from the reduced version

$$\vec{E} = \sum_i \bar{e}_i \vec{b}_i.$$

Such enthalpy H_1 is the Legendre transformation of the internal energy U used in `FixRedD` condition. Comparing to the enthalpy H used in `FixE` condition, another $\frac{1}{2}\epsilon_0\Omega E^2$ term is added. The derivative of the energy over the strain is also rigorously computed, as in the `FixRedD` condition.

In practice, the field is provided using the `red_elec_field_bar` key with an array consists of three floating-point numbers, in the unit of `V`. The `elec_field_scale` field scales such input value.

7.2.5 `FixRedMix` condition

In this condition, mixed electrical conditions could be applied to different surfaces, as specified in the input by the field `red_field_direction`. For a surface orthogonal to $\vec{b}_i$, it is open-circuit with reduced displacement field controlled if `red_field_direction[i]` is set to 2, or closed-circuit with reduced electric field ($\bar{e}$) controlled if `red_field_direction[i]` is set to is set to 1. Any other values in `red_field_direction` is invalid.

The "enthalpy" (we just call it "enthalpy" for convenience), to be called H_M (M=mixed), which is a partial Legendre transformation from U, is used in the thermodynamics formulas:

$$H_M = U - \sum_k \bar{e}_k d_k,$$

where the summation of k runs over the directions where closed-circuit condition are required.

In practice, the reduced fields are set by the `red_disp_field` and `red_elec_field_bar` fields, with `disp_field_scale` and `elec_field_scale` as scaling factors. For the direction indexed i, the value in `red_disp_field` will be used if `red_field_direction[i]` is set to 2, and the value in `red_elec_field_bar` will be used if `red_field_direction[i]` is set to 1. Other values are ignored.

7.3 Structural manipulations

7.3.1 Approximately periodic sub-cell

> **Note**
>
> The feature described in this section is available as of version 0.30 (Jun. 5, 2026).

In this section, we describe the theoretical background and algorithm for obtaining an *approximately periodic sub-cell* (APS-cell) for a given supercell. The purpose for developing the APS-cell is to build a smaller (possibly much smaller) supercell (sub-cell) for a given (original) supercell, so that the original supercell could be approximately considered as periodic repetition of the sub-cell. In current implementation (see the `heff_approx_periodic_sub_cell` utility program), the supercell matrix for the sub-cell is given by the user.

This is useful for, somehow, "compress" a supercell. When the original supercell is "almost" periodic repetition of some basic structure, for example, the single-domain ferroelectric phase, the APS-cell could describe the original supercell to a good accuracy. We note that it is not possible to compress a general structure; here the "compression" of structure is only based on the assumed periodicity, with direct and rigorous mathematical form.

Generally, the supercell lattice vectors $\mathbf{A}_s$ are linked with the unit cell lattice vectors $\mathbf{A}_u$ by

$$\mathbf{A}_s = \mathbf{A}_u\mathbf{P}, \quad \mathbf{P} \in \mathbb{Z}^{3\times 3},$$

where $\mathbf{A}_s$ is a matrix made by supercell lattice vectors as *columns*, and similarly for $\mathbf{A}_u$. The *supercell matrix* $\mathbf{P}$ must be an non-degenerate integer matrix.

The sub-cell is defined by another integer matrix, say $\mathbf{M}$, so that

$$\mathbf{A}_t = \mathbf{A}_u\mathbf{M}, \quad \mathbf{M} \in \mathbb{Z}^{3\times 3},$$

with the requirement

$$\mathbf{P} = \mathbf{MD}, \;\; \mathbf{D} \in \mathbb{Z}^{3\times 3},$$

which comes naturally from the periodic requirement.

The reciprocal space lattice vectors are defined for the unit cell and supercells in normal way, satisfying, e.g.,

$$\mathbf{A}_u^T \mathbf{B}_u = \mathbf{I},$$

where $\mathbf{I}$ is identity matrix. It is easy to show that

$$\mathbf{B}_u = \mathbf{B}_s \mathbf{P}^T,$$

and similarly for other lattice vectors. For an reciprocal space vector $\vec{q}$, the Born von-Karman (BvK) boundary condition requires

$$\vec{q} \cdot \vec{a}_s = 2\pi n, \;\; \exists n \in \mathbb{Z},$$

where $\vec{a}_s$ is an *arbitrary* translation vector of the supercell. It is easy to show that such requirement is equivalent to

$$\vec{q} = \mathbf{B}_s \vec{h}_s, \;\; \exists \vec{h}_s \in \mathbb{Z}^3.$$

Similar relationship is also valid for the sub-cell defined by $\mathbf{B}_t$. With the relationship $\mathbf{P} = \mathbf{MD}$ given, it is easy to show that the $\vec{q}$-points in the first Brillouin zone of the sub-cell and satisfying its BvK boundary condition form a *subset* of those $\vec{q}$-points for the original supercell, denoted by

$$BZ_t \subseteq BZ_s.$$

According to Fourier transformation (FT), any function defined on the real-space unit cells $F(\vec{r})$ could be equivalently represented in the reciprocal-space vectors

$$F(\vec{q}) = \frac{1}{N} \sum_{\vec{q}} F(\vec{r}) \exp(-i\vec{q} \cdot \vec{r}),$$

where N is the number of unit cells in the supercell, and, correspondingly,

$$F(\vec{r}) = \sum_{\vec{q}} F(\vec{q}) \exp(i\vec{q} \cdot \vec{r}).$$

The local mode variables, defined in each unit cell, naturally satisfy this relationship. Thus, with the help of FT, the local mode variables $F(\vec{r})$ in the sub-cell could be obtained from its reciprocal-space representation $F(\vec{q})$, which is obtained by choosing the $\vec{q}$-points in BZ_t from the FT of $F(\vec{r})$ defined in the original supercell. For alloy models, the mixed site occupation variables are first converted to concentrations, and the relationship above could then be used.

Clearly, such process includes approximation, by discarding the $\vec{q}$-points that does not belong to BZ_t. Define the summation of squared norm of the local modes defined in the reciprocal space

$$A_s = \sum_{\vec{q} \in BZ_s} |F(\vec{q})|^2,$$

and similarly for A_t. It is natural that the following variable

$$\phi \equiv \frac{A_t}{A_s}$$

measures the proportion of local mode movements in the original supercell captured by the sub-cell, and thus indicates the ability of the sub-cell for representing the original supercell as periodic repetition.

The derivation above assumes the supercell to be periodic along all of the three crystal axes. For partly periodic models (films and rods), the periodicity is broken in some axes. Currently, this is solved by building *intermediate* cells that are fully periodic, and apply the algorithm above on such intermediate cells, followed by applying the original periodic boundary conditions. Such process is done automatically inside the program. For the axes where the periodicity is broken, the lattice vectors in the sub-cell must be exactly the same as those in the original cell. This is clear because we could not expect periodicity on an axis that has no periodic boundary condition at all.

CHAPTER

EIGHT

UTILITY PROGRAMS

This documents introduces some utility programs provided by this package. The utility programs may be standalone C++ executable (typically linked against to the `heff_general` library, `libheff_general.so` on Linux) or Python CUI modules. Note that some important programs have been introduced in *Running Simulations* or *Building a model* (for example, `heff_run_sim` and `heff_gen.fit.cui.otf_fit`, etc), and will not be covered by this document.

8.1 C++ utility programs

The C++ utility programs are provided along with the `Heff_general` part of this package, and are installed automatically along with the main program. The executables are installed into the binary directory (`bin` on Linux) in the installation root.

8.1.1 `heff_build_info`

This program simply provides the version and building environment information for current `Heff_general` package, including the version of the program, macros defined during building process, compiler version and supported C++ features, etc.

8.1.2 `heff_dof_split`

This utility is provided for processing the DoF HDF5 file generated and used by this package. For the DoF HDF5 file formats, see *DoF output files*.

The main purpose of this program is to pick up one frame from a given multi-frame structure DoF HDF5 file, and write into a single-frame structure DoF HDF5 file, as well as text files.

> **Note**
>
> The text file output is only available as of Dec. 4, 2025.

The main options of this program includes:

- `-d <prefix>` or `--dof-file-prefix <prefix>`: specify the DoF input file (multi-frame structure) prefix as `<prefix>`. The prefix contains *no* suffix. Default: `dof-runs`.
- `-o <output>` or `--output-prefix <output>` specify the output prefix of the single-frame DoF file. Default: `dof-split`.
- `-t <prefix>` or `--text-prefix <prefix>` specify the prefix of text output files (available as of Dec. 4, 2025). Default: `split-`.
- `-s <id>` or `--sim-id <id>`: specify the simulation run index (`sim_id`) to be picked. This value is consistent with the `sim_id` data in the thermodynamics output file. Default: 1

- `-f <id>` or `--frame-id <id>`: specify the frame index (0-based) inside the selected simulation run. Default: 0.
- `-c <file>` or `--config-file <file>`: specify the configuration file. Default: `heff_config.toml`.

This program also supports processing MPI output (from MPI simulations with independent tasks, see *Parallel simulation for independent tasks*). For this case, specify the `-m` or `--mpi-size` with argument (greater than 1) specifying the number of processes. The files with names `<prefix>0.h5`, `<prefix>1.h5` etc will be processes.

8.1.3 heff_dipole_split

Available as of version 0.25.3 (Mar. 10, 2026).

This program splits the dipole HDF5 file and provide an output file in the final dipole YAML file format. Similar to `heff_dof_split`.

8.1.4 heff_dof_average

This utility is provided for computing the average value of DoFs in the same simulation, which is typically from the `tape` output files.

Some important command line arguments:

- `-d` or `--dof-file-prefix`: Prefix to the DoF file (without file type extension). Default: `dof-runs`.
- `-o` or `--output`: Output filename prefix. Default: `dof-average`.
- `-s` or `--sim-id`: Specify the simulation index. An integer is expected. Default: 1.
- `-b` or `--begin-frame-id`: Specify the beginning frame index (inside the specified simulation index). Default: 0, indicating the first frame.
- `-e` or `--end-frame-id`: Specify the end frame index (inside the specified simulation index). This specified index is *not* included in the average. Default: -1, indicating the last frame is included.

8.1.5 heff_dipole_average

Available as of version 0.25.3 (Mar. 10, 2026).

This utility is provided for computing the average value of dipoles in the same simulation, which is typically from the `tape` output files.

Some important command line arguments:

- `-d` or `--dipole-file-prefix`: Prefix to the DoF file (without file type extension). Default: `dipole-tape`.
- `-o` or `--output`: Output filename prefix. Default: `dipole-average`.
- `-s` or `--sim-id`: Specify the simulation index. An integer is expected. Default: 1.
- `-b` or `--begin-frame-id`: Specify the beginning frame index (inside the specified simulation index). Default: 0, indicating the first frame.
- `-e` or `--end-frame-id`: Specify the end frame index (inside the specified simulation index). This specified index is *not* included in the average. Default: -1, indicating the last frame is included.

8.1.6 heff_make_param_index_map

This utility program provides the parameter index map for the given Heff model, which is useful for analyzing the parameter fitting data, especially the regression data.

This program requires unit cell YAML file (`heff_unitcell.yaml` by default) and parameter file (`heff_params.yaml` by default) as input. The output data is written into (by default) `heff_params_index.yaml`. In the output

file, the position for each parameter is filled with its index (zero-based index). Such index is numbered globally (all parameters, including dipole part and local part, including all INTs, are numbered one by one), and is consistent with the parameter array during parameter fitting regression process.

Usage:

```
heff_make_param_index_map
```

Type `heff_make_param_index_map -h` to see the optional command line arguments for modifying the input or output file paths.

8.1.7 heff_diel_response

This program computes the frequency-dependent complex dielectric response using the algorithm in [Dielectric2017]. The computation of the dielectric response is based on the status output file (see *Explaining the results*). The file should be specified using the `--input-file` or `-i` flag (default: `status-steps.h5`), which is typically generated by a previous MD run. The MD time step (`simulation.md.timestep`) and `simulation.md.equiv_rate` are loaded from the configuration file, which must be consistent with the MD run that generated the input status file. The program picks the variable with specified key (defined in `diel_response.source_var`) to compute response of this key. The default variable corresponds to the dipole moment `DipoleMoment`, which is just the standard dielectric response. Note that this program could also be used to compute "dielectric" response of other variables, like some type of local modes. The variable specified by the key `source_var` must be included in the data status file during the calculation from which the file is generated (set by `simulation.snap_vars` key).

Warning

By default the `source_var` key uses the `DipoleMoment` quantity, which is *not* generated by default in simulations. To enable this output, add `DipoleMoment` to `simulation.snap_vars` before running the simulation.

The output file is a text file, the default file name is `diel_response.txt`. If `--voigt` or `-v` flag is set, the program also generates another output file (default: `diel_response_voigt.txt`) with Voigt-type compressed indexes. This is only available for three-dimensional input variables.

8.1.8 heff_approx_periodic_sub_cell

Note

This is available as of version 0.30.0 (Jun. 5, 2026).

Note

This feature is highly experimental.

Build the approximately periodic sub-cell (APS-cell) with given original cell and given sub-cell supercell matrix. For theoretical background, see *Approximately periodic sub-cell*.

Practically, the program reads the `heff_config.toml` as configuration file. The original cell must be initialized as in simulations. The supercell matrix of the sub-cell must be given by the `aps_cell.sub_cell` table, which has exactly the same structure as the `cell` table in the simulation configuration file.

During the running process, the ϕ variable (see *Approximately periodic sub-cell*) is computed and written into output, for all local mode variable components (in each unit cell) and total local mode variable. After running the program,

the resulting DoFs of the APS-cell are written into (by default) the `aps-modes.txt`, `aps-strain.txt`, `dof-aps.h5` files. For alloy models, the occupation *concentration* of each site is written in the `occupations_restored.yaml` file. The occupation of mixed sites are obtained by finding the element with maximal concentration in each mixed site and written in the `site-aps.txt` file. The validity of such occupation must be carefully verified.

Warning

Please carefully read the theoretical background (*Approximately periodic sub-cell*) before using this program. The choice of sub-cell supercell matrix is critical for the validity of the final result and must be considered carefully. The results must be carefully verified.

8.2 Python executable modules

Most of the Python utility programs provided by this package are presented as an executable module of (some subpackage) of the `heff_gen` Python package. They could be executed using the shell commands like

```
python3 -m heff_gen.<executable_module_name> <arguments_to_module>
```

after the `heff_gen` package has been properly installed. For example,

```
python3 -m heff_gen.cell.extract_disp
```

These executable modules may accept some command-line arguments. To see the possible arguments, pass the `-h` or `--help` argument to the corresponding executable module, for example

```
python3 -m heff_gen.cell.extract_disp -h
```

In the following sections, these executable modules will be introduced one by one. The titles starting with `heff_gen.` are just the module names.

8.2.1 `heff_gen.cell.extract_disp`

This program supports analyzing the atomic displacements of distorted structures. To use this program, the unit cell metadata file set must be provided, and an input structure file (by default, `CONTCAR`) must be provided. For more information and typical usages, see *Check selected local mode basis*.

8.2.2 `heff_gen.cell.expand_cell`

This program generates expanded cell (supercell) of the input supercell. The input supercell must be specified as the initialization structure for simulations (see *Initial structure*). The supercell matrix (for *expanding* the input structure, not the supercell matrix of the *input structure*) should be given with the command-line argument, using one, three, or nine integer values, corresponding to a scalar, diagonal matrix, and full matrix form of the expanding matrix, respectively. The expanding matrix definition is the same as that of the supercell in the simulation (see *Structure specification*). The configuration file for the input structure should be specified by `--config` or `-c` flag (the default is `heff_config.toml`).

The program will generate the DoF HDF5 file and the site text representing the expanded supercell. The default output filenames are `expanded-dof.h5` and `expanded-sites.txt`. If `--text` or `-t` flag is set, the text file for strain and local modes will also be written. The output files could be used for the initial structure for simulations.

Important

Remember to set the initial structure in the configuration file. If nothing is specified in `init` table, the structure will be the reference structure, which is typically not expected.

Example: suppose we have the supercell specified by `dof-in.h5` and `sites-in.txt` (they may be generated from a previous simulation, as `dof-last.h5` and `out_sites.txt` by default). To create a $2 \times 2 \times 4$ supercell expanding of this cell, prepare the following input file, named `expand_cell_config.toml`

```
[cell]

# IMPORTANT: set the supercell matrix according to the input structure.
# This is only an example!!
supercell_matrix = [
    [2, 2, 0],
    [-2, 2, 0],
    [0, 0, 2]
]

[init]
read_sites_txt = "sites-in"
load_dof_h5 = "dof-in"
```

Then use the following command:

```
python3 -m heff_gen.cell.expand_cell -c expand_cell_config.toml  2 2 4
```

The output files `expanded-dof.h5` and `expanded-sites.txt` fully specifies the resulted expanded supercell.

8.2.3 heff_gen.cell.gen_unitcell_yaml

This program generates the unit cell YAML specification file `heff_unitcell.yaml` from specification files (`unitcell_spec.toml`, unit cell structure file and LWF files). See *Generate unit cell YAML file*.

8.2.4 heff_gen.cell.build_atomic_struct

This program constructs the atomic structure file (compatible with DFT calculations) from the Heff supercell data. The input structure should be specified as for the initial structure for simulations (see *Initial structure*). The unit cell specification file should be specified using `--unitcell-spec` or `-u` (default: `unitcell_spec.toml`). The configuration file, describing the initial structure information, should be provided by `--config-file` or `-c` (default: `heff_config.toml`). The element order of the output file may be optionally specified using `--element-order`. If provided, all the chemical elements in the structure must be presented exactly once in the arguments.

Example:

Suppose we have structures described by `dof-in.h5` and `sites-in.txt` (typically generated by previous simulation runs). To generate the atomic structure, edit the following configuration file `heff_config.toml`:

```
[cell]

# IMPORTANT: set this according to the input file. This is only an example!!
supercell_diag = [8, 8, 8]

[init]
load_dof_h5 = "dof-in"
read_sites_txt = "sites-in"
```

Then use the following command:

```
python3 -m heff_gen.cell.build_atomic_struct
```

The output file is by default `POSCAR_modes`. To change the output file name, use `--output-file` or `-o`. To change output format, use `--output-format`. The atomic structure is represented by `ase.atoms.Atoms` object in this program. The available formats could be found at ASE IO formats document.

Important

Remember to set the initial structure keys (like `load_dof_h5`). The program will *not* load any input file by default and build a reference structure in that case, which is typically not what expected!

8.2.5 heff_gen.cell.lwf_basic_params

This program is provided for computing the basic parameters associated with the LWFs. See *Precalculation* for more information.

8.2.6 heff_gen.lwf.cui.gen_lwf_phonon

This is the main program for building single LWF from phonon dispersion. See *Run single LWF building*.

8.2.7 heff_gen.lwf.cui.gen_multiple_lwf_phonon

This is the main program for building multiple LWFs from phonon dispersion. Seee *Run multiple LWF building*.

8.2.8 heff_gen.lwf.cui.lwf_dynamic_mass

Compute the so-called "dynamical" masses of LWFs. To be documented later.

8.2.9 heff_gen.lwf.cui.mode_atomic_disp

Available as of Sep. 1, 2025.

This program computes the atomic displacements of a given local mode motion of specified type of mode (LWF). Specifically, this program assumes a infinite cell (i.e., no periodic boundary condition applied), with the center of the LWF in the center unit cell (indexed `[0, 0, 0]`). The atomic displacement of all atoms (involved by this LWF at the center unit cell), arisen by the given local mode motion (through the argument to the program), are presented at output.

The LWF should be specified by its data file (`XXX_data.json` file), using the `--shell-lwf-file` or `-s` argument (required). The displacement of such LWF, with dimension (i.e., number of elements) consistent with the LWF dimension, must be provided using the positional argument of this program.

Example:

```
python3 -m heff_gen.lwf.cui.mode_atomic_disp -s lwf_u_data.json 0.2 0.1 -0.3
```

Here, the LWF is specified by the `lwf_u_data.json` file, with displacement `0.2 0.1 -0.3` (indicating this is a three-dimensional LWF). The output may look like

```
Atom 0  uc: [0 0 0]  site: [-0. -0. -0.]  displacement: [ 0.00209787  0.00104893 -0.
↪0031468 ]
...
Atom 4  uc: [0 0 0]  site: [0.5 0.5 0.5]  displacement: [ 0.14021803  0.07010901 -0.
↪21032704]
```

(continues on next page)

(continued from previous page)

```
Atom 7  uc: [-1  0  0]  site: [0.5 0.5 0.5]  displacement: [ 0.0006533  -0.00297918  0.
↪00893755]
...
```

Here, each line is provided for one displaced atom. In each line, the `Atom 0` is the displaced atom index (involved by this LWF). `uc: [0 0 0]` indicates the unit cell position (in unit cell fractional coordinate, meaning that only integers). `site: [0.5 0.5 0.5]` is the atom site inside the unit cell (in unit cell fractional coordinate). The displacement is given in Cartesian coordinate in the unit of Angstrom.

Note

As no structure information required, the atom sites are not refined according to the possible atom sites inside the unit cell. If floating point precision problem arises, try to change the `--symprec` argument.

8.2.10 heff_gen.lwf.cui.plot_shell_lwf_band

This program plots the phonon dispersion of the LWF (i.e., Wannier interpolation) along with DFT phonon dispersion (provided by Phonopy package). The required position argument specifies the directory containing phonon data (required by Phonopy, containing `FORCE_SETS` or `FORCE_CONSTANT`, `band.conf` and unit cell structure file). The phonon data directory is the same as that for building LWF with reciprocal-space method (*Reciprocal-space method*).

The LWF data should be provided by the `--shell-lwf` or `-s` argument, with the `XXX_data.json` files. Multiple LWF files may be provided for a single run.

Other available parameters could be seen by using the `-h` or `--help` argument.

Technically, this program first converts the shelled-LWF to the periodic supercell format (the inverse operation of that used in shelled-LWF building, see *Build shelled-LWF*), and then build the dynamical matrix based on the DFT phonon data. The supercell matrix for the periodic supercell is by default loaded from `band.conf` file. However, if the supercell matrix loaded this way is a non-diagonal matrix, the program fails because this is not consistent with the q-point mesh. To solve this problem, provide the supercell dimension (diagonal elements of the diagonal supercell matrix) with the `--supercell-dim` flag (with three integers).

If `--fp-mesh-points` or `-m` flag (available as of Sep. 6, 2025) is specified, the points on the dispersion, with q-point coherent with the q-mesh (under Born-Von Karman boundary condition), are plotted. If `--no-Bloch-interpolation` or `-B` flag is specified, the Bloch interpolation of the phonon dispersion (i.e., the default DFT phonon dispersion, by Phonopy) will not be shown.

Example:

To the following figure shows the result of this program for one LWF of $BaTiO_3$. The command is like

```
python3 -m heff_gen.lwf.cui.plot_shell_lwf_band -s lwf_u_data.json ./phonon/ -m
```

Here, the red lines are the DFT phonon dispersion (with Bloch interpolation, by Phonopy). The red dots are the points that coherent with the q-point mesh (here, $2 \times 2 \times 2$ mesh is used), as required by the `-m` flag. The blue lines are the phonon dispersion of the LWF (Wannier interpolation).

Important

This program plots the *phonon dispersion of the LWF*, or equivalently, the Wannier interpolation of phonon bands using the provided LWF. The force constant data comes directly from the underlined DFT data. This is **not** the phonon dispersion of the effective Hamiltonian (the effective Hamiltonian interactions/parameters are not even used by this program)!

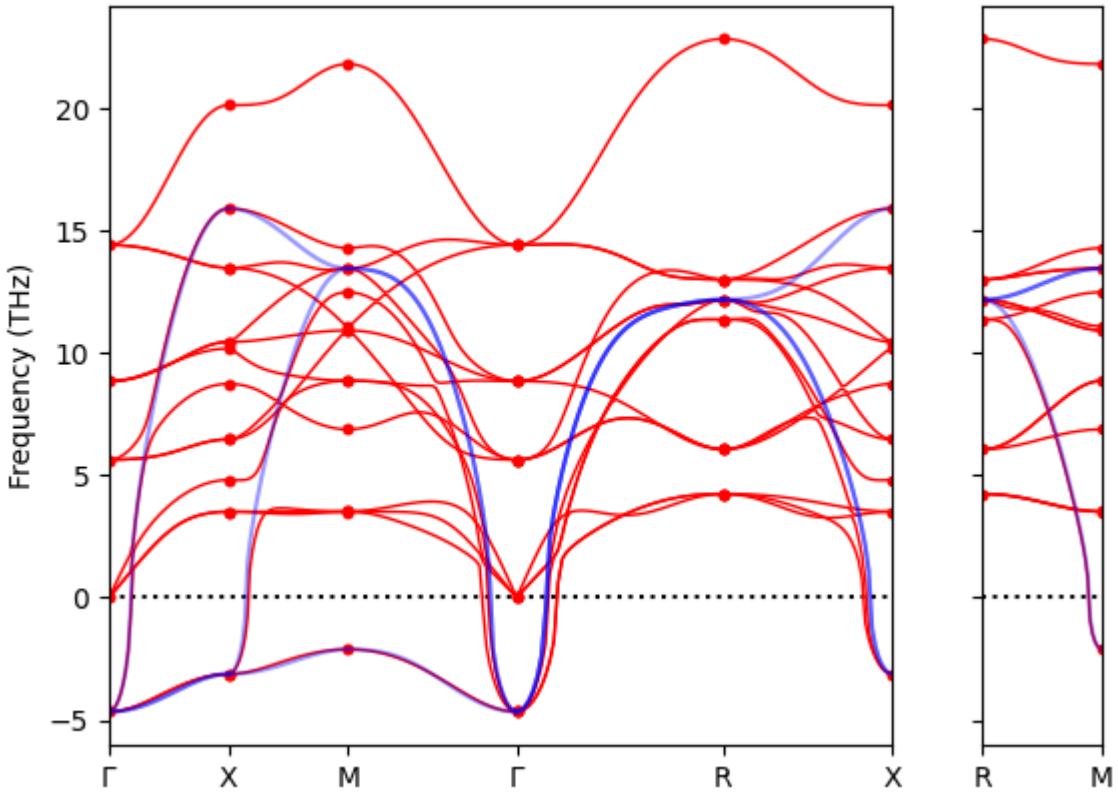


Fig. 1: Example of shell LWF band

8.2.11 heff_gen.lwf.cui.show_lwf_info

This program shows some basic information for the provided LWF, the same as part of the output while building the LWF with some configurations. The LWF data (`XX_data.json` file) should be specified with the positional argument.

8.2.12 heff_gen.lwf.lwf_bec_tensor

This program computes the Born effective charge (BEC) of given LWF. See *Precalculation*.

8.2.13 heff_gen.fit.cui.extract_occupation_history

This program extracts the occupation history YAML file (generated in OTF fitting), pick one set of site occupations (of selected step), and write into site text file, for possible later usage (e.g., as input of simulation, or continuing OTF fitting, etc). The occupation history file (default file name: `occupations_history.yaml`) could be set via the positional argument of this program. The step index should be provided with the `--step` or `-s` flag.

See also *On-the-fly learning*, `init.read_sites_txt` in *Input reference*.

8.2.14 heff_gen.fit.cui.fit_from_data

See *fit_from_data*.

8.2.15 heff_gen.fit.cui.merge_otf_fit

See *merge_otf_fit*.

8.2.16 heff_gen.fit.cui.otf_fit

See *On-the-fly learning*.

8.2.17 heff_gen.fit.cui.multi_otf_fit

Running several OTF fittings, on different cells, at the same run. This module is experimental.

8.2.18 heff_gen.fit.cui.param_null_space

This program computes the *null space* of the parameters in the design matrix, i.e. the combinations of columns in the design matrix that yield null vector. This is helpful for finding the linearly dependent parameters in the data.

8.2.19 heff_gen.fit.cui.reuse_data_fit

See *reuse_data_fit*.

8.2.20 heff_gen.fit.cui.collect_otf_cells

Available as of version 0.30.0 (Jun. 5, 2026). This program collects the supercell matrices from a range of work directories where OTF (including multi-OTF) fitting have been performed. Example usage:

```
python3 -m heff_gen.fit.cui.collect_otf_cells -r . work* multiotf*
```

The command collects all the *direct* sub-directories of the root directory (`.`, current directory here) with names starting with `work` or `multiotf`. The output file by default is `cell_configs.toml`, providing the cell configurations in TOML format using the structure of multi-OTF input file. The format of the sub-directory names (`work*` and `multiotf*`) is parsed using the `glob` library in Python.

In this program, the multi-OTF (including multi-simulation-OTF iteration) directories are automatically detected by finding the `multi_otf.cells` array in the `fit_config.toml` file. To disable all multi-OTF data, pass `--no-multi` argument.

8.2.21 heff_gen.fit.cui.merge_otf_count_fps

Available as of version 0.30 (Jun. 5, 2026).

This program counts the number of FP calculations performed during in the selected OTF fittings. The input configuration file is the same as that of the `merge_otf_fit`, with the `merge_otf.data_dirs` field listing all required directories. Merge-otf fittings are automatically detected.

The number of FP calculations and OTF MD steps in each OTF run (or multi-OTF sub-OTF run) are provided in the `otf_fp_statistics.csv` file. The summed number of FP calculations and OTF MD steps grouped by different supercell sizes (number of unit cells) is given in the `otf_fp_summary.csv` file.

8.2.22 heff_gen.fit.plot.otf_plot

Plots the basic information for on-the-fly (OTF) learning. See *On-the-fly learning*.

8.2.23 heff_gen.fit.plot.plot_param_null_space

This program visualizes the parameter null space. See also `heff_gen.fit.cui.param_null_space`.

8.2.24 heff_gen.fit.plot.reuse_plot

This program plots the information for reuse data fit. See also *reuse_data_fit*.

CHAPTER

NINE

INPUT REFERENCE

9.1 Atomic structure files

In this package, for portability considerations, all the *atomic* structure input/output and most of their internal representations are handled by the commonly used Python packages like `ase` and `phonopy`.

9.1.1 ASE structure files

Most of the atomic structure input/output in fitting part are supported by the ASE package. The file input and output are handled by `ase.io.read` and `ase.io.write`, respectively. The internal representation of the atomic structures is realized by the class `ase.atoms.Atoms`.

For most of the cases, the input file format could be detected by ASE package and need not to be specified by the user. For some special cases and for output, the format may need to be specified explicitly. For the available formats, please refer to ASE IO document.

9.1.2 Phonopy structure file

In some parts of this package (mainly in LWF building part), the atomic structures are read and written with the support of Phonopy package, with internally represented by the `PhonopyAtoms` class (see PhonopyAtoms document). The available interface modes and corresponding structure file formats are available at Phonopy interface document.

9.2 Simulation configuration file

This section lists the input keys in the simulation configuration file (by default, `heff_config.toml`). Possible keys are given for each accepted TOML tables. This configuration file is mainly used in simulations (see *Running Simulations*). OTF fitting (see *On-the-fly learning*) and some other programs also use some of the tables listed below.

This input file is in TOML format. We here introduces the available keys in this configuration file, organized by top-level tables. The key names are given as sub-section titles, followed by their types and default values. If encapsuled tables appears, we include the names of parent arrays or tables to avoid ambiguity (except the top-level table name).

9.2.1 `file_path` table

The `file_path` table controls the base path for resolving the relative paths (provided elsewhere in the configuration file). This table is available as of version 0.12.0.

`model_path_base`

type
String (path)

default
. (current directory)

This field controls the base path for model data files (unit cell YAML file, interaction JSON file, parameter file). Must be a path to an existing directory. The default value `.` represents current path for running this program. If the filenames of these model files are provided with relative paths (this is the default case), these relative paths are resolved based on the path provided by this field. If the filenames are absolute paths, this base path is not used.

9.2.2 `cell` table

The `cell` table controls the *supercell* configuration for the simulation. It is mandatory for running simulations (`heff_run_sim`) and some other programs. See also *Structure specification* for more information.

`supercell_matrix`

type
Array (2d array of integers)

default
none

The `supercell_matrix` field defines the matrix P for the supercell. The value for this key must be consistent with 3×3 matrix of integers. See *Minimal example* for more information.

Example:

```
[cell]
supercell_matrix = [
    [1, 1, 0],
    [-1, 1, 0],
    [0, 0, 2]
]
```

Note

Exactly one of `supercell_matrix` and `supercell_diag` must be given for this table.

Tip

Changed since Jul. 11 2025: the determinant of supercell matrix may be positive or negative. However, it is strongly recommended to use a matrix with positive determinant.

`supercell_diag`

type
Array (of integers)

default
none

The `supercell_diag` is provided for convenience, for the cases where the supercell matrix P is only a diagonal matrix. This field accepts an array of exactly three integers, with positive product (i.e., the corresponding supercell matrix must have positive determinant). See *Minimal example* for more information.

Example:

```
supercell_diag = [2, 2, 2]
```

> **Note**
>
> Exactly one of `supercell_matrix` and `supercell_diag` must be given for this table.

> **Tip**
>
> Changed since Jul. 11 2025: the determinant of supercell matrix may be positive or negative. However, it is strongly recommended to use a matrix with positive determinant.

`periodic`

type
Array (of Booleans)

default
`[true, true, true]`

This field defines the periodicity of the simulated supercell, along with `vacuum_bound`. See *Structure specification* for more information.

`vacuum_bound`

type
Array (of Floats)

default
`[0.0, 0.0, 0.0]`

This field defines the periodicity of the simulated supercell in a more detailed manner than `periodic`. See *Structure specification* for more information.

`symprec`

type
Float

default
1e-7

The `symprec` field defines the tolerance when comparing floating-point numbers. When the difference is smaller than this value, two floating-point numbers are considered to be equal. This is mainly used in symmetry transformations.

`unitcell_yaml`

type
String (file path)

default
`heff_unitcell.yaml`

This field specifies the file path from which the unit cell file `heff_unitcell.yaml` will be read. The default file path is just `heff_unticell.yaml` at current directory. Both relative and absolute paths are allowed. If relative path provided, it will be resolved based on `file_path.model_path_base`, see *model_path_base*.

9.2.3 `ewald` table

The `ewald` table controls the Ewald summation calculation. See *Ewald summation file* for more information.

`delta`

type
Float

default
1e-12

Defines the parameter δ controlling the Ewald summation precision. See [EwaldSum2019] for more information.

`gcut_factor`

type
Float

default
1.0

Defines the factor for the reciprocal-space vector cutoff length.

`output_path`

type
String (directory path)

default
`.` (current directory)

Defines the path for writing output HDF5 file of the Ewald summation result.

> **Note**
>
> This is only used for the `heff_cal_ewald` executable. If the Ewald summation is triggered from the `heff_run_sim` executable, the `energy.files.ewald_path` will be used instead.

`overwrite_existing`

type
Boolean

default
false

This field defines the behavior when the expected output Ewald HDF5 file already exists. Generally, if this is set to `true`, such file is overwritten. Otherwise (the default), the file is not overwritten. If the loaded Ewald file is not consistent with current configuration and this key is set to `false`, error message will be generated. In this case, try set this key to `true`, or delete the existing Ewald data HDF5 file.

`compute_3d`

type
Boolean

default
true

If set to `true`, the 3d Ewald summation file (which will also be used in the simulation) will be computed along with the low-dimensional one. Only applicable to the low dimensional (2d, 1d or 0d) models.

`crystal_symmetry`

type
Boolean

default
true

Available as of version 0.25.0 (Mar. 1, 2026).

If true, the crystal symmetry information is utilized to compute the Ewald summation matrix, which may significantly reduce the computational load. See also *Symmetry-adapted Ewald summation*. Note that enabling this field requires the symmetry information to be provided in the unit cell YAML file, which is naturally satisfied if such file is created using this package in recommended way.

`load_checkpoint`

type
Boolean

default
true

Available as of version 0.25.3 (Mar. 10, 2026).

If true, the checkpoint file will tried to be loaded, and any computed data will be loaded if possible. See *Checkpoint file in Ewald summation*.

`write_checkpoint_min_size`

type
Integer

default
16

The Ewald checkpoint file (*Checkpoint file in Ewald summation*) will be written during Ewald summation if the number of unit cells in the supercell is no less than the given value.

If set to 0, the Ewald checkpoint file will not be written.

`lr_symmetry` table

The `lr_symmetry` table controls the options used for analyzing the symmetry property of the Ewald matrix, where LR stands for long-range. This table is only used if `crystal_symmetry = true`, available as of version 0.25.0 (Mar. 1, 2026).

`lr_symmetry.distance_group_sep`

type
Float

default
0.05

This key controls the separation of distance between different groups while generating LR-SATs. The SATs are only searched in one group.

lr_symmetry.sat_report_file

type
String (file path)

default
`LR_sat_report.txt`

This key specifies the filename for writing a brief LR-SAT report file.

9.2.4 physical table

The `physical` table defines the physical environment for the simulation. See *Physical environment* for more information.

temperature

type
Float (physical quantity)

unit
Kelvin

default
-1

This key defines the temperature for the simulation. This is not used in some workflows where the temperature is controlled by the workflow.

pressure

type
Float (physical quantity)

unit
GPa

default
0.0

This key defines the hydrostatic pressure for the simulation. See *Pressure and stress* for more information.

stress

type
Array (of Floats)

unit
GPa

default
`[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]`

This key defines the (possibly) anisotropic external stress for the simulation. See *Pressure and stress* for more information.

`electrical_condition`

type
String (Literals)

options
`FixE`, `FixD`, `FixRedEbar`, `FixRedD`, `FixRedMix`

default
`FixE`

Available as of version 0.28.

Control the type of electrical condition applied. See *Electrical conditions*.

`red_field_direction`

type
Array (of Integers)

default
[0, 0, 0]

Available as of version 0.28.

Control the type of electrical boundary conditions used in each direction. Each of the components may take the following values:

- 1: reduced electric field $\bar{e}$ condition is applied to corresponding surface.
- 2: reduced displacement field d condition is applied to corresponding surface.

Any other values are invalid and will cause the program to be terminated.

This field is only used, and must be supplied, if and only if `electrical_condition = "FixRedMix"`.

`elec_field_homo`

type
Array (of Floats)

unit
V/

default
`[0.0, 0.0, 0.0]`

The `elec_field_homo` key controls the homogenous electric field along with `elec_field_scale`. The field must be specified with an array containing three floating-point numbers, defining the electric field in the Cartesian coordinate in the unit of Volt per Angstrom.

Only used when `electrical_condition = "FixE"`

Note

The unit V/ is a very large unit comparing to SI unit V/m ($1\mathrm{V}/ = 10^{10}\mathrm{V}/\mathrm{m}$). The values between 0.01 and 0.1 are sufficient for most cases.

red_elec_field_bar

type
Array (of Floats)

unit
V

default
`[0.0, 0.0, 0.0]`

Available as of version 0.28.

This field controls the reduced electric field $\bar{e}$. See *Electrical conditions*.

This is only used when `electrical_condition = "FixRedEbar"` or `electrical_condition = "FixRedMix"`.

elec_field_scale

type
Float

unit
dimensionless

default
1.0

The `elec_field_scale` key defines a scalar factor on `elec_field_homo` or `red_elec_field_bar`, determining the actual magnitude of electric field applied on the system. The actual applied electric field is `elec_field_homo` or `red_elec_field_bar` multiplied by `elec_field_scale`. This key is ignored in some workflows where the electric field scale is set by the workflow.

disp_field_homo

type
Array (of Floats)

unit
e/^{2}

default
[0.0, 0.0, 0.0]

Available as of version 0.28. This field controls the macroscopic displacement field when `electrical_condition = "FixD"`. The scale `disp_field_scale` is multiplied. See also *Electrical conditions*.

red_disp_field

type
Array (of Floats)

unit
e

default
[0.0, 0.0, 0.0]

Available as of version 0.28. This field controls the reduced displacement field d when `electrical_condition = "FixRedD"` or `electrical_condition = "FixRedMix"`. The scale `disp_field_scale` is multiplied. See also *Electrical conditions*.

disp_field_scale

type
Float

default
1.0

Available as of version 0.28.

The scale applied to `disp_field_homo` or `red_disp_field`. Similar to `elec_field_scale`.

beta

type
Float

unit
dimensionless

default
0.0

The `beta` key defines the electric screening factor β used in low-dimensional models. See [LowDimHeff] for more details. This is not used for 3d model (see *Structure specification*).

misfit table

The `misfit` table is a sub-table to `physical`, defining the misfit strain, usually induced by misfit between the film and the substrate, known as epitaxial strain. See *Strain and misfit strain* for the physical definition and example.

misfit.enabled

type
Boolean

default
false

The misfit strain is imposed if `misfit.enabled` is set to `true`.

misfit.free_components

type
Array (of Booleans)

default
`[true, true, true, true, true, true]`

The `misfit.free_components` field determines whether each of the strain DoFs is variable. The given array must have exactly 6 components, corresponding to each of the strain components. The strain component is variable if the corresponding element in this array is set to `true`, or fixed to initial value if set to `false`. If `transform_mode` is *not* set to `"Direct"`, the `misfit.free_component` key controls the strain components *before* such transformation (see *Strain and misfit strain*).

misfit.strain

type
Array (of Floats)

default
`[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]`

unit
dimensionless

THe `misfit.strain` field determines the initial strain. The given array must have exactly 6 components. The given strain component will be maintained during the simulation if the corresponding element in `free_components` is `false`, or free to be changed if it is `true`. See *Strain and misfit strain* for physical definition and examples.

Note

This strain is only applied if `misfit.enabled` is set to `true`.

Note

This strain is not used if initial strain is given by other files (see `init.read_strain_txt` and `init.load_dof_h5`).

misfit.transform_mode

type
String (literals)

options
`Direct`, `AlignX`, `AlignZ`, `Manual`

default
`Direct`

The `misfit.transform_mode` defines the method for determining the transformation matrix for initial strain. The given values must be one of the provided options (see above). See *Strain and misfit strain*.

misfit.transform_matrix

type
Array (2d 3×3 array of Floats)

default
All zero

The `misfit.transform_matrix` defines the transformation matrix when `misfit.transform_mode` is set to `Manual`. The input array must be consistent with 3×3 unitary (orthogonal) matrix. See *Strain and misfit strain* for more information.

Example:

```
[physical.misfit]
transform_mode = "Manual"
transform_matrix = [
```

(continues on next page)

(continued from previous page)

```
    [1.0, 0.0, 0.0],
    [0.0, 1.0, 0.0],
    [0.0, 0.0, 1.0]
]
```

Note: this example is actually equivalent to `transform_mode = "Direct"` as an identity matrix is given.

`inc_elec_field` table

The `inc_elec_field` table is a sub-table of `physical` table, controls the gradually changed ("increased") electric field scale during the simulation. See *Changing electric field* (`EsweepInc` part). Usually, this table need *not* to be defined manually; using the `EsweepInc` or `EcycleInc` tasks is enough.

`inc_elec_field.enabled`

type
Boolean

default
false

The gradual electric field changing feature is enabled if `inc_elec_field.enabled` is set to `true`. This field would be automatically set to `true` if `flow.task` is set to `EsweepInc` or `EcycleInc` in the `heff_run_sim` executable.

`inc_elec_field.inc_scale`

type
Float

unit
dimensionless

default
1.0

Note

The default value of this key was changed from 0.0 to 1.0 since May 7, 2026.

The total change of electric field scale (equivalent to `elec_field_scale`) during one simulation run. This field would be automatically set if `flow.task` is set to `EsweepInc` and `EcycleInc` in the `heff_run_sim` executable.

`inc_elec_field.steps_rate`

type
Float

unit
dimensionless

default
0.4

This value determines the portion (1.0 for all, 0.5 for half, etc) of steps in each simulation run in which the electric field scale would be changed. The electric field scale (equivalent to `elec_field_scale`) would be changed linearly during these steps. See *Changing electric field* for more information.

Important

Usually, the `physical.inc_elec_field.steps_rate` should be set to a value *no larger than* that of `simulation.equiv_rate`. Otherwise, the statistic average values are collected with changed electric field, which may lead to error results.

`inc_disp_field` table

The `inc_disp_field` controls the increasing of displacement field scale. The table has exactly the same structure (including keys and default values) as the `inc_elec_field` table.

`td_elec_field` table

The `td_elec_field` configures the time-dependent (TD) homogeneous electric field. A scalar function may be defined by this table and used as the electric field scale (`elec_field_scale`). This table is only available for real-time simulators (where the real *time* could be defined, for example, MD simulator, while HMC simulator is *not* a real-time simulator).

`td_elec_field.enabled`

type
Boolean

default
false

Set this value to `true` to enable the TD field controlled by this table.

`td_elec_field.type`

type
String (literals)

options
`NoTDField`, `Sine`, `LaserPulses`

default
`NoTDField`

This field defines the model of the TD field. Each model defines an analytical expression (as a function of time) for the electric field scale, with possibly some parameters to be filled.

`type = "NoTDField"`. No TD electric field is applied. Equivalent to `td_elec_field.enabled = false`.

`type = "Sine"`. Use a sine function for the TD field. The electric field scale A is defined by:

$$A(t) = \sin(2\pi f(t - t_0) + \phi),$$

Here, f is the frequency, determined by `frequency` key; `t_0` is defined by `start_time`; `\phi` is determined by `phase`.

`type = "LaserPulses"`. Use a laser pulse model defined by [LaserPulse2022]. The electric field scale is defined by:

$$A(t) = \sum_{n=0}^{N-1} \Big\{ \exp\big[-4\big(\frac{t - t_n}{\tau}\big)^2 \ln 2\big] \cos(2\pi f(t - t_n)) \Big\},$$

where t_n is the center of n-th pulse, calculated by

$$t_n = t_0 + n\Delta t.$$

Here, N is defined by `num_pulses`, Δt is defined by `pulse_interval`, τ is defined by `tau`.

td_elec_field.frequency

type
Float (physical quantity)

unit
THz

default
1.0

Defines the frequency of field in sine and laser pulse models. See *td_elec_field.type*.

td_elec_field.phase

type
Float

unit
dimensionless

default
0.0

Defines the initial phase in sine model. See *td_elec_field.type*.

td_elec_field.start_time

type
Float (physical quantity)

unit
fs

default
0.0

Defines the start time of electric field in sine and laser pulse models. See *td_elec_field.type*.

td_elec_field.tau

type
Float (physical quantity)

unit
fs

default
100.0

Defines the τ parameter in laser pulse model. See *td_elec_field.type*.

td_elec_field.pulse_interval

type
Float (physical quantity)

unit
fs

default
1000.0

Defines the interval time between two laser pulse centers. See *td_elec_field.type*.

td_elec_field.num_pulses

type
Integer

default
1

Defines the total number of pulses. See *td_elec_field.type*.

td_disp_field table

This table is similar to the `td_elec_field` table, but controls the displacement field `disp_field_scale`. The table has exactly the same structure as the `td_elec_field` table.

9.2.5 energy table

The `energy` table controls the energy (including force and stress) evaluation process and options, including the Hamiltonian contributions, DoF status, etc.

disable_long_range

type
Boolean

default
false

Disable the long-range dipole interaction. This is mainly provided for test purposes.

strain_status

type
String (literals)

options
`Enabled`, `Frozen`, `Disabled`

default
`Enabled`

This key controls the strain DoF status. See *Controlling the DoFs*.

auto_compute_ewald

type
Boolean

default
true

This key controls whether the Ewald summation data file (HDF5 file) will be computed automatically if it is not found. See *Ewald summation file*.

disable_interactions

type
Array (of integers)

default
[] (Empty)

The INTs with indexes listed in this field will be disabled during the simulation. The disabled INTs provide no contribution to the energy, force and stress. See *Controlling the energy components*.

lwf_controls table

The lwf_controls table controls the status for each type of local mode (LWF). The data for each LWF is specified in a sub-table named by its name. For example, if we have an LWF called u, the table controlling its behavior is lwf_controls.u.

Example:

```
[energy.lwf_controls.u]
status = "Enabled"

[energy.lwf_controls.v]
status = "Disabled"
```

In the following, we use <lwf_name> to indicate the LWF name in the sub-table name.

lwf_controls.<lwf_name>.status

type
String (literals)

option
Enabled, Frozen, Disabled

default
Enabled

This key controls the status for the selected local mode (LWF). See *Controlling the DoFs*.

lwf_controls.<lwf_name>.free_components

type
Array (of Integers)

default
empty

Available as of version 0.23.3 (Jan. 28, 2026). This is an experimental feature.

Set the free components of the specified local mode. If provided, this must be an array with the number of elements equal to the dimensionality of the LWF, containing only values of 0 and 1, where 0 specifies the corresponding element of the local mode will be frozen to initial value in all unit cells, and 1 specifies the corresponding element of the local mode is free.

If not provided, all the components are free.

> **Warning**
>
> If the field is not set as the requirement above, the behavior is undefined.

`interact_controls` (array of tables)

type
Array (of tables)

default
Empty

This is an array of tables, each of them controls an INT in the Hamiltonian. In TOML, this is typically specified using the double square bracket syntax, for example:

```
[energy]
[[energy.interact_controls]]
index = 12
max_group = 2

[[energy.interact_controls]]
index = 13
disabled = true
```

`interact_controls.index`

type
Integer

default
-1

Select the index of INT that is under control by current table. Must be filled to make this table effective.

`interact_controls.max_group`

type
Integer

default
-1

Select the maximal SGRP index in the current INT to be included. The SATs with group index larger than this value will be disabled. The default value -1 indicates no such limitation. See *Controlling the energy components*.

`interact_controls.disabled`

type
Boolean

default
false

If `true`, the current INT would be disabled. This is equivalent to add the index of current INT into `disable_interactions`.

`files` table

The `files` table controls the file paths related to the energy model. All the paths here must be provided with a non-empty string, pointing to the desired input file. The input paths (*except* `ewald_path`, see below) are resolved based on `file_path.model_path_base` (see *model_path_base*) if provided with relative paths.

`files.ewald_path`

type
String (directory path)

default
`.`

This field specifies the directory path from which the Ewald summation data file would be loaded or written (see also *Ewald summation file*). An existing path to a directory must be provided. The default path is just current working directory.

> **Note**
>
> If relative path given, the path will be resolved based on *current working directory*, not controlled by `file_path.model_path_base`.

`files.interactions_json`

type
String (file path)

default
`heff_interactions.json`

This field specifies the file path from which the interactions data JSON file would be loaded. The default is just the `heff_interactions.json` file from current directory.

`files.params_yaml`

type
String (file path)

default
`heff_params.yaml`

This field specifies the file path from which the parameters are loaded from. The default is just the `heff_params.yaml` file from current directory.

`files.free_local_dof_txt`

type
String (file path)

default
empty

If provided, the file with specified name will be read for specifying the status of each DoF (in each unit cell). The input file must have the same format as the local mode text file, with the value for each DoF must be 0 (indicating the DoF is frozen) or 1 (indicating the DoF is free). Input file with any other format or any other values will result in undefined behavior.

If this input file is used simultaneously with the `energy.lwf_controls.<lwf_name>.free_components`, the configuration by the file takes advantage.

The example of this file may be obtained by setting `files.write_local_free_dof` to true and run the simulation program.

`files.write_local_free_dof_file`

type
String (file path)

default
`free_local_modes.txt`

Output file for the status of the local mode DoFs. Enabled if `write_local_free_dof` is true. Each value is 1 for a free DoF or 0 for a frozen DoF. The value of the frozen DoFs will be fixed to their initial values, which may be 0 if reference structure used as initial structure, or other values if initial structure provided otherwise.

`files.write_local_free_dof`

type
Boolean

default
false

If set to true, the text file containing the information of DoF status (free or not) is written.

`files.fftw_export_wisdom`

type
Boolean

default
true

If `true`, the FFTW wisdom will be exported to a file called `FFTW_wisdom` before the program finishes. The FFTW wisdom may make the FFT planning faster in later calculations. See FFTW document: http://fftw.org/fftw3_doc/Words-of-Wisdom_002dSaving-Plans.html#Words-of-Wisdom_002dSaving-Plans

`files.fftw_import_wisdom`

type
Boolean

default
true

If `true`, the program will try to load the FFTW wisdom (previously exported, if any, see `files.fftw_export_wisdom`) before planning FFTs. The program will report whether the loading process is successful. The program should not be affected even if the wisdom is not loaded or loaded failed (except the possible slowing down in FFT planning).

9.2.6 `init` table

The `init` table controls the initialization of the simulated cell. The initialization of the simulated cell including the distribution of elements (atoms) for the mixed sites (if any) and the initialization of all DoFs. The atom distribution may be initialized randomly (the default) or read from provided text file. The DoFs may be initialized to zero (the default; except that some of them may be initialized elsewhere) or read from input files (text files or HDF5 files). The

input file path configurations in this table should provide the required file names (relative or absolute paths, *without* suffixes). If empty (the default), the corresponding input file is not read.

Note

The file paths controlled by fields in this table are resolved based on current working directory (for relative paths), not controlled by `file_path.model_path_base`.

Important

The provided filenames should contain *no* file type suffixes (like `.txt`). Such suffixes are added by the program. For example, if `read_modes_txt = "modes-in"`, the actual file to be read is `modes-in.txt`. The actual suffixes are listed for each field.

Input files for MPI parallelism

For MPI parallelism case (with the program compiled with MPI support and number of processes more than 1, see *MPI parallelism*), there may be two different strategies for the input files:

- All the MPI processes initialize with the same input file. This guarantees the initial structure equivalence for different MPI ranks, and is suitable for the simulations with different physical conditions (electric field, etc). To use this strategy, use the input keys as if no MPI enabled. For example, if `read_modes_txt = "modes-in"` (see below for more information), all MPI processes loads the same file `modes-in.txt` to initialize the simulated cell.
- Each MPI process reads its own input file. This allows more detailed controls for the initial structures, and is necessary for simulations with different initial structures (including different mixed site atom distributions). Specifically, this is equivalent to the previous strategy if all input files for the MPI processes are exactly the same. To use this strategy, use the input keys with `_rep` (short for "replica") suffix. The MPI rank index (zero-based) will be inserted into the input file name, right before the file type suffix. For example, if `read_modes_txt_rep = "modes-in"` (see below for more information), the MPI process ranked 0 will load the file called `modes-in0.txt`, the MPI process ranked 4 will load the file called `modes-in4.txt`, etc.

Important

The two keys with same prefix, with and without `_rep` suffix, could not be assigned to non-empty values simultaneously. For example, if `read_modes_txt = "modes-in"`, one *cannot* set `read_modes_txt_rep` to non-empty string at the same time.

seed

type
Integer (unsigned, C++ type `std::uint32_fast_t`)

default
0

This field controls the initial seed of the pseudo random number generator. In this program, the C++11 `std::mt19937` pseudo random number generator is used, initialized with the given seed. Different seed may result in different atom distribution (for mixed sites), different momenta initialization (if required), etc, and thus result in different simulation paths.

read_modes_txt

type
String (file path prefix)

suffix
`.txt`

default
empty

This field specifies the filename for reading the mode text file (if provided). The mode text file provides the local mode values for each unit cell. This text file should contains one header line (which is actually not parsed by the program) followed by N rows, where N is the number of unit cells in the simulated supercell. Each row contains an index (zero-based, for the unit cell) followed by n floating-point numbers, specifying the local mode values for the corresponding DoF. The values in the same row should be separated by space characters (spaces or Tabs). The local mode values for each unit cell (i.e., in each row) should be sorted according to the order of the LWFs listed in the unit cell metadata (`unitcell_spec.toml` or `heff_unitcell.yaml`). The order of the unit cells should be consistent with that in the site input/output file (see `read_sites_txt`, `out_sites_txt`).

Example for the input file:

```
num     u_1     u_2     u_3     v_1     v_2     v_3
0       0.0528888       0.104989        -0.0281194      0.0354708       0.0805966       ␣
↪0.0543007
1       -0.0306055      -0.00990871     0.0748127       0.0150759       0.0579834       -
↪0.0383334
```

This example is provided for the supercell with two unit cells. The first row is the header. In each of the two following rows, the six floating-point numbers following the index are local mode values for the unit cell, with the first three values provided for the components of local mode named `u`, and last three values provided for the local mode `v`.

Tip

The sample of this input file is also given by the output file (default name: `final-modes.txt`) after running the simulation with the same supercell configuration.

Important

The number of unit cells and number of local mode DoFs in each unit cell must be consistent with the current configuration of the simulated cell, otherwise unexpected result may occur.

read_strain_txt

type
String (file path prefix)

suffix
`.txt`

default
empty

This field specifies the filename from which the initial strain will be loaded (if provided). The input file simply contains six floating-point numbers, for the six strain components. Example:

```
0.00578224      0.0042256       0.00577026      0.000305214     -3.47599e-05    0.
↪000382657
```

Note

If this field is used along with misfit strain (`physical.misfit.enabled = true`), the strain provided by `physical.misfit.strain` will *not* be used.

`read_modes_txt_rep`

type
String (file path prefix)

suffix
`.txt`

default
empty

Changed since version 0.23: from `read_modes_txt_mpi` to `read_modes_txt_rep`.

This field is similar to the `read_modes_txt` field, but the MPI rank is inserted into the filename for the MPI parallel cases (see *MPI parallelism*). This allows different MPI ranks to load different initial structure if required. The actual filename to be loaded is `<read_modes_txt_rep><mpi_rank>.txt`, for example, if `read_modes_txt_rep = "modes-in"`, the MPI process ranked 3 would load the file called `modes-in3.txt`. See also *Input files for MPI parallelism*.

Warning

This key cannot be used along with `read_modes_txt`.

`read_strain_txt_rep`

type
String (file path prefix)

suffix
`.txt`

default
empty

Changed since version 0.23: from `read_strain_txt_mpi` to `read_strain_txt_rep`.

This field is similar to the `read_strain_txt` field, but with MPI rank inserted into the filename. See *MPI parallelism* and `read_modes_txt_rep`. See also *Input files for MPI parallelism*.

`read_sites_txt`

type
String (file path prefix)

suffix
`.txt`

default
empty

This field specifies the filename for reading the site occupation information (atom distribution of mixed sites) if provided. The file is a text file with one header row (which is not actually parsed by this program) followed by N rows (where N is the number of unit cells in the simulated supercell). In each followed row, the unit cell index is provided, followed by the coordinate of the unit cell (in unit cell fractional coordinate, expressed by three integers) and the occupation indexes (the σ variable as described in [ALFE-H-2025]) for each *site* of the unit cell. The occupation indexes should be 0 if the site is vacuum or not valid, positive if the site is a mixed site (each value corresponds to a type of element occupying this site), or -1 if the site is not a mixed site. The order of the sites should be consistent with that specified in unit cell metadata (`heff_unitcell.yaml`). The values in the same row are separated by space characters (spaces or Tab).

Example: suppose we have a supercell with 2 unit cells. Each unit cell contains two sites, where the second is a mixed site. The text file with following content may be a valid input file for initializing the site occupations.

```
num     a       b       c       sigs
0       0       0       0       -1      1
1       0       0       1       -1      2
```

In this example, suppose the mixed site is defined in `unitcell_spec.toml` as follows (see *Building a model*):

```
[[unitcell.mixed_sites]]
name = "B"
concentrations = [0.5, 0.5]
elements = ["Ti", "Zr"]
generator_site_id = 1
```

Then, the first unit cell contains a Ti atom (indexed 1, here the index is one-based), and the second unit cell contains a Zr atom (indexed 2).

Tip

This input file has the same format as the file generated by the simulation program as specified by `write_sites_txt`.

`read_sites_txt_rep`

type
String (file path prefix)

suffix
`.txt`

default
empty

Changed since version 0.23: from `read_sites_txt_mpi` to `read_sites_txt_rep`.

See `read_sites_txt` and *Input files for MPI parallelism*.

`write_sites_txt`

type
String (file path prefix)

suffix
`.txt`

default
`out_sites`

If non-empty value given, the site occupations after initialization (either initialized randomly or loaded from file) would be written to the file. If MPI parallelism enabled (*MPI parallelism*), the MPI rank is inserted to the filename. This output file has the same format as the input file required by `read_sites_txt`.

`load_dof_h5`

type
String (file path prefix)

suffix
`.h5`

default
empty

This field specifies the filename of the DoF HDF5 file to be loaded (if provided) for initializing the DoFs (including local modes and strain). The input file is in HDF5 format and is typically generated by the simulation program itself after another run (default filename: `dof-last.h5`), or generated by the `heff_dof_split` utility (see *heff_dof_split*).

Note

If this field is used along with `read_modes_txt`, the modes from the latter is used, overwriting the data provided by the HDF5 file. Similarly for `read_strain_txt`.

Note

If this field is used along with misfit strain (`physical.misfit.enabled = true`), the provided initial strain by `physical.misfit.strain` is ignored.

`load_dof_h5_rep`

type
String (file path prefix)

suffix
`.h5`

default
empty

Changed since version 0.23: from `load_dof_h5_mpi` to `load_dof_h5_rep`.

See `load_dof_h5` and *Input files for MPI parallelism*.

9.2.7 `simulation` table

This table controls simulation process. The variables (fields) defined directly in the `simulation` table are the configurations shared by most of the simulation algorithms. The algorithm-specified configurations are given in some sub-tables of the `simulations`. Note that the program always accept all sub-tables without any warnings, even if some of them may not be used by the algorithm performed. For example, it is valid to write an `simulation.hmc` sub-table while running MD simulation, and the program will *not* produce any warning messages about this, even if this table is not used by the program at all.

The output file control variables in this table accepts *prefixes* to the output file, meaning that the file type suffix should *not* be provided by the user, and would be added by the program instead. For serial runs, the output file name is directly the configured prefix following the file type suffix. For MPI runs, the MPI process rank will be inserted before the file

type suffix (see *MPI parallelism*). For example, if `snap_file = "snap"` (see below), the output file name for serial run will be `snap.txt`, the output for MPI run will be `snap0.txt` for process ranked 0, and `snap5.txt` for process ranked 5, etc.

sim_steps

type
Integer

default
1000

The number of simulation steps to be performed in each simulation run. The "step" is defined by the actual simulation algorithm applied. For example, for MD simulation, one "step" is one MD step containing the integration of time for a time step (with length `simulation.md.timestep`). For HMC simulation, one "step" is one HMC sweep containing several MD steps (`simulation.hmc.sweep_md_steps`).

equiv_rate

type
Float

default
0.5

valid range
[0.0, 1.0]

Defines the proportion of steps that will be treated as equilibrium process, and the thermodynamics statistics data will be collected only *after* these steps. For example, if `equiv_rate` is set tp 0.6, and `sim_steps` is set to 1000. Then, the first 600 steps are treated as equilibrium process and the thermodynamics statistics data is collected in the last 400 steps.

snap_freq

type
Integer

default
200

Defines the frequency (steps) for the snap output. The default, 200, indicates the snap output will be given every 200 simulation steps. The snap output contains some log information in the standard output (`stdout`) as well as some data in the snap output file (the output file is determined by `snap_file`, by default, `snap.txt`).

Warning

If the frequency is too high (the `snap_freq` value is too small), the snap output may cost significant simulation time and reduce the overall simulation efficiency.

stat_samp_freq

type
Integer

default
1

Defines the frequency for collecting thermodynamics statistics samples *after* the `equiv_rate` part. The default value, 1, indicates the thermodynamics sample will be collected every step.

write_status_freq

type
Integer

default
0

This field defines the frequency for writing the status into file. If set to 0 (the default), the status output is not generated. The output file is in HDF5 format with filename defined by `status_output_file`.

snap_potential_parts

type
Boolean

default
true

This field defines whether the potential contributions would be calculated and written during the snap process. This includes the potential contributions from each part (strictly, from each INT) at the step where the snap output is performed.

disable_sum_rules

type
Boolean

default
false

If set to `true`, the `sum_rules` defined in the unit cell metadata (including the sum rule imposed by the acoustic LWFs) will be disabled during the simulation. Only for test purposes.

thermo_vars

type
Array (of Strings)

default
empty

This field defines the thermodynamics variables to be calculated and written during the simulation, accepting an array of strings, each of them must be a valid thermodynamics variable specification. See *Defining the physical quantities*, *Physical quantity specification reference*. The default is an empty array, indicating only some default physical quantities will be calculated and written, including the potential energy, polarization, etc.

snap_vars

type
Array (of Strings)

default
empty

This field defines the physical quantities to be written into the snap output file during the simulation, accepting an array of strings with the same format of `thermo_vars`. See *Defining the physical quantities*, *Physical quantity specification reference*.

This field is only useful if `independent_snap_vars` is set to `true`. Otherwise, the snap output variables are just the variables used by the thermodynamics statistics, as determined by `thermo_vars`.

`independent_snap_vars`

type
Boolean

default
true

This field defines whether the snap output variables will be defined independently from the thermodynamics statistic variables. If set to `true`, the physical quantities for the snap output is defined by `snap_vars`. Otherwise, the quantities are those used during the thermodynamics statistics, as determined by `thermo_vars`.

`extra_condition_vars`

type
Array (of Strings)

default
empty

This field could be used to add extra condition variables (that not added by default). The condition variables will be written in the output thermodynamics file. This field accepts an array of strings, each of them must be a valid condition variable name (Input key), see *Simulation condition quantities*.

> **Warning**
>
> If the selected condition variable is not fixed during the simulation (for example, temperature for microcanonical ensemble MD simulation), the output result may be incorrect.

`disable_built_in_snap_vars`

type
Boolean

default
false

Set this key to `true` to disable the built-in snap output variables. Has no effect if `independent_snap_vars` set to `false`.

`disable_built_in_thermo_vars`

type
Boolean

default
false

Set this key to `true` to disable the built-in thermodynamics statistical variables.

snap_file

type
String (file path prefix)

suffix
`.txt`

default
`snap`

see also
`snap_freq`

Controls the output file name for the snap output. The frequency of such output is determined by `snap_freq`.

thermo_file

type
String (file path prefix)

suffix
`.txt`

default
`heff_thermo`

see also
`thermo_vars`

Controls the output file name for the thermodynamics statistical output file.

final_modes_file

type
String (file path prefix)

suffix
`.txt`

default
`final-modes`

Controls the filename of the final mode data file. The file will be a text file containing the values of all local modes at the end of the simulation. Use empty string as input to disable this output. The output file has exactly the same format as the input file required by `init.read_modes_txt`.

final_strain_file

type
String (file path prefix)

suffix
`.txt`

default
`final-strain`

Controls the filename of the final strain data file. The file will be a text file containing the values of the strain at the end of the simulation (in total 6 values). Use empty string as input to disable this output. The output file has exactly the same format as the file required by `init.read_strain_txt`.

dof_runs_file

type
String (file path prefix)

suffix
`.h5`

default
`dof-runs`

Controls the filename for writing the DoF (including local modes and strain) at the end of each run. This will be an HDF5 file. Use empty string to disable this output.

dof_last_file

type
String (file path prefix)

suffix
`.h5`

default
`dof-last`

Controls the filename for writing the final DoF data (including local modes and strain) at the end of the final run. This will be an HDF5 filename. Use empty string to disable this output. This output file has exactly the same format as the input file required by `init.load_dof_h5` and `init.load_dof_h5_rep`.

status_steps_file

type
String (file path prefix)

suffix
`.h5`

default
`status-steps`

see also
`write_status_freq`

This key controls the output filename for the status output file. The output file will be an HDF5 file containing the status information of the selected steps (see `write_status_freq`). Use empty string to disable this output.

bg_snap table

The `bg_snap` table controls a type of output file which will be called as "background snap". Similar to the snap output controlled by `snap_freq`, the background snap write some information based on the *instantaneous* status where the snap is performed (i.e., no history information considered, no statistical average performed). However, there is no "foreground" information written to the `stdout`, no potential contributions calculated, and the snapped variables must be specially defined in this sub-table, making this background-snap lighter and costing less computational efficiency, and thus could be called more frequently. This is mainly designed for watching the time-dependent evolution during one single simulation run, especially along with the time-dependent electric field.

bg_snap.enabled

type
Boolean

default
false

The overall switch for the background-snap. The background-snap is enabled only if this key is set to `true`.

bg_snap.freq

type
Integer

default
10

The frequency for the background-snap. The snap will be performed every configured steps. Must be positive.

bg_snap.filename

type
String (file path prefix)

suffix
`.txt`

default
`bg_snap`

The output filename for the background-snap. Must be a non-empty string.

bg_snap.vars

type
Array (of Strings)

default
empty

This field defines the physical quantities to be snapped by the background-snap. Only the quantities specified here will be written into the output file. See *Defining the physical quantities*.

> **Note**
>
> Unlike the default snap output, there are *no* built-in variables for background-snap. Thus, all physical quantities to be snapped must be specified in this field.

md table

The `md` table controls the details for molecular dynamics (MD) simulations. Note that some of the keys in this sub-table also controls the MD simulation paths inside the HMC sweeps.

md.timestep

type
Float (physical quantity)

unit
fs (10^{-15} s)

default
1.0

This key controls the time step size in the MD simulation. Each MD simulation step integrates along the time axis for the given length.

md.ensemble

type
String (literals)

options
`NPT`, `NPH`

default
`NPH`

This key controls the ensemble used in the MD simulation. Currently, there are only two available ensemble types. The first is the `NPH` ensemble, where "N", "P", "H" state that the number of atoms, pressure and enthalpy are fixed, respectively. This is similar to microcanonical ensemble. The second is the `NPT` ensemble, where "T" states the temperature is fixed. This is similar to the canonical ensemble.

Note that in the effective Hamiltonian, the strain of the simulated cell is explicitly imposed as a type of DoF, which couples directly to other DoFs and external pressure (stress). Thus, the most natural choice is to fix the external pressure (stress) and let the strain to relax. If you do want to fix the basis vectors of the cell (and thus the volume), you could disable the strain DoF (see `energy.strain_status`) or use the misfit strain table (`physical.misfit`) to impose extra constraints.

For MD simulations, the default ensemble is the `NPH` ensemble, where the equations of motion are solved directly (using, e.g., velocity Verlet algorithm), and the total enthalpy are conserved naturally without any extra constraints. For the `NPT` ensemble, the thermostat (currently, just Evans-Hoover thermostat) is applied to conserve the temperature.

md.adjust_temp_begin

type
Boolean

default
false

If set to `true`, the temperature will be adjusted (by rescaling momenta) at the begin of each MD simulation run.

Changed since Jul. 21, 2025: In `heff_run_sim`, if `flow.alg` is set to `MD` and `flow.task` is set to `Tsweep` or `Tcycle`, this key will be automatically set to `true`, although this would not be reflected in the echoed output configuration file.

md.adjust_temp_interval

type
Integer (non-negative)

default
0

This field determines the interval for adjusting the temperature (by rescaling the momenta). A non-negative value must be given. If positive value given, the momenta will be rescaled every specified MD steps. If the value is 0, such feature is disabled.

md.omelyan_xi

type
Float

default
0.1931833275037836

This field determines the ξ coefficient in the Omelyan integrator (see [Omelyan2002]). Only valid if `flow.integrator = "Omelyan"`.

hmc table

The `hmc` sub-table controls some details in hybrid Monte Carlo (HMC) simulations. Note that the HMC simulations is also controlled by some keys in the `md` sub-table, especially `timestep` and `omelyan_xi`.

hmc.sweep_md_steps

type
Integer

default
40

This field determines the number of MD steps to be conducted for each HMC sweep. See [HMC2018] for more details.

ptmc table

The `ptmc` table controls some details of the parallel-tempering Monte Carlo (PTMC) simulations. In the `heff_run_sim` executable, the PTMC simulation must be turned on by using `task = "ParallelTempering"`, see *Parallel tempering* for more details. As the PTMC simulation has a rather different framework from other simulations, some configurations in the `simulation` table are not used.

ptmc.temp_dist

type
String (literals)

options
`InvalidTempDist`, `Read`, `Linear`, `Geometry`, `Inverse`

default
`InvalidTempDist`

This key defines the temperature distribution strategy for the first iteration of PTMC simulation (see also `num_iters`). For PTMC simulation, the distribution must be specified with an value *other than* `InvalidTempDist`. Let T_L and T_H be the low and high temperatures for the PTMC simulation (`ptmc.temp_low` and `ptmc.temp_high`, respectively), and N be the total number of replicas created by the PTMC simulation (`ptmc.num_replicas`). The temperature $T(n)$ for the replica indexed n (starting from 0) are defined as follows.

For `temp_dist = "Linear"`, the temperature is uniformly distributed between T_L and T_H. Specifically, we have

$$T_{\text{Linear}}(n) = T_L + n\frac{T_H - T_L}{N - 1}.$$

For `temp_dist = "Inverse"`, the reciprocal value of temperature ($1/T$) is uniformly distributed. Specifically,

$$\frac{1}{T_{\text{Inverse}}(n)} = \frac{1}{T_L} + n\frac{T_H^{-1} - T_L^{-1}}{N-1}.$$

For `temp_dist = "Geometry"`, the temperature is determined by

$$T_{\text{Geometry}}(n) = T_L \left(\frac{T_H}{T_L}\right)^{\frac{n}{N-1}}.$$

For `temp_dist = "Read"`, the temperature distribution is read from given file (see `ptmc.read_temp_dist_yaml`).

ptmc.num_replicas

type
Integer

default
1

Number of replicas to be created for the PTMC simulation. Each replica represents a unique temperature to be simulated by the system. See *Parallel tempering*.

The number of replicas must be a multiple of the number of MPI processes created for the PTMC simulation.

> **Warning**
>
> The number of replicas should be set by the user for all PTMC simulations. The default value 1 is meaningless.

ptmc.num_iters

type
Integer

default
1

Number of PTMC iterations. At the end of each iterations, the temperature distribution will be updated. See *Parallel tempering*.

ptmc.block_steps

type
Integer

default
200

Number of simulation steps to be performed in one PTMC *block step*. Note that the number of PTMC block steps to be performed in each PTMC iteration is determined by `simulation.sim_steps`.

ptmc.temp_high

type
Float (physical quantity)

unit
Kelvin

default
-1.0 (Invalid value)

The highest temperature of the replicas to be generated in the PTMC simulation. Must be larger than `temp_low`. The user must provide this field for PTMC simulations, as the default value -1.0 is an invalid value. The `flow.temp_start`, `flow.temp_end`, `flow.temp_step` fields are *not* used in PTMC simulations.

ptmc.temp_low

type
Float (physical quantity)

unit
Kelvin

default
-1.0 (Invalid value)

The highest temperature of the replicas to be generated in the PTMC simulation. Must be less than `temp_high`. The user must provide this field for PTMC simulations, as the default value -1.0 is an invalid value. The `flow.temp_start`, `flow.temp_end`, `flow.temp_step` fields are *not* used in PTMC simulations.

ptmc.dof_replica_file

type
String (file path prefix)

suffix
`.h5`

default
empty

This field specifies the filename prefix for the replica-specified DoF file. If a non-empty string is given, the final DoF at the end of each PTMC iteration for each replica will be written to a file named `<prefix>-rep<rep_id>-iter<iter_id>.h5`, where `<prefix>` is the value given by this key, `<rep_id>` is the replica index, and `<iter_id>` is the iteration index.

ptmc.modes_replica_file

type
String (file path prefix)

suffix
`.txt`

default
empty

Similar to `ptmc.dof_replica_file`, this field specifies the filename prefix for the replica-specified local mode value file. If a non-empty string is given, the final local mode data will be written to a file called `<prefix>-rep<rep_id>-iter<iter_id>.txt`.

ptmc.strain_replica_file

type
String (file path prefix)

suffix
`.txt`

default
empty

Similar to `ptmc.dof_replica_file`, this field specifies the filename prefix for the replica-specified strain value file. If a non-empty string is given, the final strain data will be written to a file called `<prefix>-rep<rep_id>-iter<iter_id>.txt`.

ptmc.dof_lowest_file

type
String (file path prefix)

suffix
`.h5`

default
`dof-lowest`

This field specifies the filename for writing the DoF of the replica with lowest temperature at the end of each PTMC iteration. If non-empty value given, the file will be `<prefix>-iter<iter_id>.h5`, where `<prefix>` is the value given by this key, `<iter_id>` is the iteration index.

ptmc.modes_lowest_file

type
String (file path prefix)

suffix
`.txt`

default
`modes-lowest`

This field specifies the filename for writing the local mode values of the replica with lowest temperature at the end of each PTMC iteration. If non-empty value given, the file will be `<prefix>-iter<iter_id>.txt`, where `<prefix>` is the value given by this key, `<iter_id>` is the iteration index.

ptmc.strain_lowest_file

type
String (file path prefix)

suffix
`.txt`

default
`strain-lowest`

This field specifies the filename for writing the strain values of the replica with lowest temperature at the end of each PTMC iteration. If non-empty value given, the file will be `<prefix>-iter<iter_id>.txt`, where `<prefix>` is the value given by this key, `<iter_id>` is the iteration index.

ptmc.write_temp_dist_yaml

type
String (file path prefix)

suffix
`.yaml`

default
`temp_dist`

This field specifies the filename for writing the temperature distribution file. If the value is not empty, the output filename will be <prefix>_iter<iter_id>.yaml, where <prefix> is the value given by this key, <iter_id> is the iteration index.

ptmc.read_yaml_temp_dist_yaml

type
String (file path prefix)

suffix
`.yaml`

default
`temp_dist_in`

This field specifies the filename for reading the initial temperature distribution, if the `temp_dist` is set to `Read`. The input filename should be <prefix>.yaml, with the same file format as the files output by this program, controlled by `write_temp_dist_yaml`.

9.2.8 `flow` table

This table configures the workflow for running the simulations, including all configurations *above* the single simulation run level and some configurations for building the simulator objects. Note that as we already mentioned above, some of the configurations in this table, in some specified configured task types, will cover corresponding configurations in other tables.

alg

type
String (literals)

options
`NoAlg`, `MD`, `CanonicalHMC`

default
`NoAlg`

This key specifies the algorithm of the simulation. Accepted options are:

- `NoAlg`: The default value which is invalid for running a simulation. To run a simulation, the user must provide a value other than this default one.
- `MD`: Molecular dynamics (MD) simulation.
- `CanonicalHMC`: Standard hybrid Monte Carlo (HMC) simulation, designed for canonical ensemble (`NPT` ensemble).

task

type
String (literals)

options
`NoTask`, `Single`, `Repeat`, `Tsweep`, `Tcycle`, `Esweep`, `Ecycle`, `EsweepInc`, `EcycleInc`, `Eloop`, `Dsweep`, `Dcycle`, `DsweepInc`, `DcycleInc`, `ParallelTempering`

default
`NoTask`

This key specifies the task of the simulation. The default value `NoTask` is invalid for running a simulation, and the user must provide a value other than this default value to run a simulation. The `ParallelTempering` task would trigger the PTMC simulation, see *Parallel tempering* for more information. For other tasks, see *Simulation tasks* for more information.

`local_model`

type
String (literals)

options
`Proto`, `Real`, `DP`, `Reci`, `ConditionalReci`

default
`ConditionalReci`

This key specifies the local model type, which is responsible for calculating the energy, force and stress arisen from the *local* part of the effective Hamiltonian (i.e., other than the long-range dipolar interaction part). See *Energy model types*.

`dipole_model`

type
String (literals)

options
`Real`, `Reci`

default
`Reci`

This key specifies the dipole model type for computing the energy, force arisen from long-range dipolar interactions. See *Energy model types*.

`mpi_dist`

type
String (literals)

options
`NoDist`, `Eamp`, `Temp`

default
`NoDist`

This key defines the mpi parallel task distribution when running the program with MPI parallelism for multiple independent tasks. See *Parallel simulation for independent tasks* for more information.

Note that the default value `NoDist` is invalid for running this type of simulation. More specifically, if MPI is enabled with multiple (more than one) processes, and `task` is not set to `ParallelTempering`, one valid value (other than the default one) must be given for this field.

Important

We emphasize again that the `MPI` parallelism in this package is NOT generally used for speeding up, but instead, only applicable to parallel tempering and running multiple independent tasks in parallel. If you do not understand this, DO NOT USE MPI!!!

integrator

type
String (literals)

options
`Verlet`, `Omelyan`

default
`Verlet`

This field determines the time integrator type for the MD simulations (including the MD inside HMC sweeps). See *Integrator types*.

temp_start

type
Float (physical quantity)

unit
Kelvin

default
-1.0

This key defines the start temperature for `Tsweep` and `Tcycle` tasks. If such tasks selected, a positive value must be given to this key. See also *Simulation tasks*.

temp_end

type
Float (physical quantity)

unit
Kelvin

default
-1.0

This key defines the end temperature for `Tsweep` and `Tcycle` tasks (for `Tcycle` task, this is actually the temperature value before the temperature sweeps for another direction). If such tasks selected, a positive value must be given to this key. See also *Simulation tasks*.

temp_step

type
Float (physical quantity)

unit
Kelvin

default
0.0

This key defines the step size to change the temperature in `Tsweep` and `Tcycle` tasks. If such tasks selected, a non-zero value must be given. The configuration of `temp_start`, `temp_end` and `temp_step` must define a finite temperature value list. See also *Simulation tasks*.

Eamp_start

type
Float

unit
dimensionless

default
0.0

This key defines the start value of electric field scale (`elec_field_scale`) for `Esweep`, `Ecycle`, `EsweepInc`, `EcycleInc` tasks. See also *Simulation tasks*. Note that this field has no effect for the `Eloop` task, because the scale always starts from 0 for an `Eloop` task.

Eamp_end

type
Float

unit
dimensionless

default
1.0

This key defines the end value of electric field scale (`elec_field_scale`) for `Esweep`, `Ecycle`, `EsweepInc`, `EcycleInc` and `Eloop` tasks. For `Ecycle` and `EcycleInc`, this is actually the scale before it sweeping another direction. See also *Simulation tasks*.

Eamp_steps

type
Integer

default
11

This key defines the number of steps (simulation runs) to be performed in one field sweep (from `Eamp_start` to `Eamp_end`) in `Esweep`, `Ecycle`, `EsweepInc`, `EcycleInc` and `Eloop` tasks (for `Eloop`, the `Eamp_start` should be replaced with 0). Both ending sides are included. For example, with default values `Eamp_start=0.0, Eamp_end=1.0, Eamp_steps=11`, the `Esweep` task will perform 11 simulation runs with `elec_field_scale` values 0.0, 0.1, 0.2, . . . , 0.9, 1.0.

Damp_start

type
Float

unit
dimensionless

default
0.0

Available as of version 0.26. Similar to `Eamp_start`, but for displacement field.

`Damp_end`

type
Float

unit
dimensionless

default
1.0

Available as of version 0.26. Similar to `Eamp_end`, but for displacement field.

`Damp_steps`

type
Integer

default
11

Available as of version 0.26. Similar to `Eamp_steps`, but for displacement field.

`mpi_temp_start`

type
Float (physical quantity)

unit
Kelvin

default
-1.0

This key determines the start temperature for MPI independent task with `mpi_dist = "Temp"`. See *Parallel simulation for independent tasks*. To run this task, a positive value must be provided.

`mpi_temp_step`

type
Float (physical quantity)

unit
Kelvin

default
0.0

This key determines the temperature step for MPI independent task with `mpi_dist = "Temp"`. See *Parallel simulation for independent tasks*. To run this task, a non-zero must be provided.

`mpi_seed_start`

type
Integer

default
1

This key defines the start seed for MPI independent task with `mpi_dist = "Seed"`. See *Parallel simulation for independent tasks*.

`repetitions`

type
Integer

default
2

This key determines the number of simulation runs to be performed with `task = "Repeat"`. See *Simulation tasks*.

9.2.9 `optimization` table

The `optimization` table controls the optimization tasks (see *Optimization procedures*).

`task`

type
String (Literals)

options
`NoTask`, `Relax`, `FlowRelax`, `NEB`

default
`NoTask`

For any optimization runs, the task must be specified using this key. The valid options are:

- `Relax`: structural relaxation, finding the local energy minima from the input structure.
- `FlowRelax`: a serial of structural relaxation runs with possibly changed physical conditions. Available as of version 0.28. If provided, the `flow` table must be provided at the same time.
- `NEB`: nudged elastic band, finding the energy path between the provided initial and final structures, from a set of initial intermediate states.

`alg`

type
String (Literals)

options
`NoAlg`, `NaiveGradientDescent`, `AdamGD`, `FIRE`, `QuickMin`, `CG`

For any optimization runs, the employed algorithm must be specified by this key. The valid options are:

- `NaiveGradientDescent`: the naive (simple) gradient descent with fixed step, also called steepest descent. This is not recommended, and is provided only for tests and comparison.
- `AdamGD`: Adam (adaptive moment estimation) algorithm. See [Adam2017].
- `FIRE`: FIRE (fast inertial relaxation engine). See [FIRE2006].
- `QuickMin`: Quick-Min algorithm. The form of Ref. [NEB_opt_2008] is applied, with velocity-Verlet integrator.
- `CG`: CG (conjugate gradient) algorithm. The form of Ref. [NEB_opt_2008] is applied.

These optimization methods may be tested for their performances, especially for the NEB tasks. Some algorithms have their specified sub-table for detailed configurations.

max_steps

type
Integer

default
10000

The maximal number of steps allowed for the optimization task.

snap_freq

type
Integer

default
20

The snapshot frequency, in number of steps.

force_tolerance

type
Float

default
0.01

unit
eV/Angstrom

The tolerance value of the maximal magnitude of force (negative gradient of the potential energy over the local modes) for the convergency.

stress_tolerance

type
Float

default
0.01

unit
eV (per unit cell)

The tolerance value of the maximal magnitude of stress (negative gradient of the potential energy over the strain) for the convergency.

mode_step_size

type
Float

default
0.001

unit
Angstrom

The move step for the modes. Interpreted by the optimization algorithms. Currently, this is used by the following algorithms: `NaiveGradientDescent`, `AdamGD`.

strain_step_size

type
Float

default
0.0001

unit
dimensionless

The move step for the strain. Interpreted by the optimization algorithms. Currently, this is used by the following algorithms: `NaiveGradientDescent`, `AdamGD`.

max_move

type
Float

default
0.2

unit
Angstrom

The maximal allowed movement of each DoF in one step. This is applied to part of the algorithms. Currently, this is used by the following algorithms: `CG`.

disable_sum_rules

type
Boolean

default
false

Disable the summation rules.

disable_built_in_status_vars

type
Boolean

default
false

Disable the built-in status variables (physical quantities).

status_vars

type
Array (of strings)

default
empty

The variables (physical quantities) collected in the snapshots. See *Defining the physical quantities*.

neb table

The neb table controls the details for nudged elastic band (NEB) calculation.

neb.num_images

type
Integer

default
0

The number of NEB images. Mandatory. A positive integer must be supplied.

neb.neb_cell

type
Boolean

default
true

Whether to enable the solid-state NEB. See [G-SSNEB-2012]. Note: currently, a different definition of stress from the cited article is used.

neb.spring

type
Float

default
5

Spring factor in `eV/Angstrom^2`.

> **Note**
>
> Different from the VTST implementation, here, a positive value should be supplied for this spring factor.

neb.spring_delta

type
Float

default
1

The Δk parameter if variable spring factor is enabled ([CI-NEB-2000], Eq. 6).

neb.variable_spring

type
Boolean

default
true

Whether to enable the variable spring factor scheme ([CI-NEB-2000], Eq. 6).

neb.jacobian_scale

type
Float

default
1

The extra scale multiplied to the Jacobian factor ([G-SSNEB-2012]).

> **Note**
>
> This parameter is also used by some optimization algorithms, even in the optimization tasks other than NEB.

neb.climb

type
Boolean

default
true

Whether to enable the climbing image NEB (CI-NEB). See [CI-NEB-2000].

neb.climb_local_maxima

type
Boolean

default
true

This is an experimental feature. If set to true, the climbing image is enabled for all potential energy *local* maxima (see also [CI-NEB-2000]). Otherwise, only the image with maximal potential energy across all NEB images is set to the climbing image, as originally proposed.

neb.dneb

type
Boolean

default
false

Experimental feature. Available as of version 0.24.0 (Feb. 6, 2026). If set to true, the doubly nudged NEB (DNEB) is enabled. See [NEB_opt_2008] (Sec. VI).

neb.dneb_switched

type
Boolean

default
true

Experimental feature. Available as of version 0.24.0 (Feb. 6, 2026). Whether to enable the switching factor of DNEB (Eq. 15 of [NEB_opt_2008]). No effect only if `neb.dneb = false`.

neb.old_spring

type
Boolean

default
false

Experimental feature. Available as of version 0.24.0 (Feb. 6, 2026). If set to true, the old-fashioned spring force is applied (Eq. 5 in [NEB2000]). The default form is Eq. 12 in [NEB2000].

Note: the tangent vector is still the new form in [NEB2000], Eq. 8.

neb.old_tangent

type
Boolean

default
false

Experimental feature. Available as of version 0.24.0 (Feb. 7, 2026). If set to true, the old-fashioned tangent vector is applied (Eq. 1 in [NEB2000]).

neb.real_spring

type
Boolean

default
false

Experimental feature. Available as of version 0.24.0 (Feb. 7, 2026). If set to true, the real-spring implementation of spring force is conducted (Appendix C, [NEB-real-spring-2016]).

TODO: This part is not finished.

9.2.10 Physical quantity specification reference

The physical quantities should be specified using the DSEL as described in *Defining the physical quantities*. The specification of each physical quantity in the input contains the key for the quantity and optional parameters. The keys of quantities are listed in the `Input key` column in the tables in *Defining the physical quantities*.

For the physical quantities that does not accept any extra parameter (the `Parameters` column contains only `none`; this is the case for most of the quantities), the input contains only the key of the quantities, for example, `Polar`, or `Potential`, etc. For the quantities accepting parameters, the parameters should be placed in the square brackets (`[]`), and multiple parameters should be split by semicolons (`;`).

Most of the available physical quantities have already listed in *Defining the physical quantities*. The physical quantities that need extra explanation, or take extra parameters, are introduced below.

ModeAverage

The `ModeAverage` physical quantity takes one parameter: the mode name. The mode name is the name specified in `unitcell_spec.toml` (or `heff_unitcell.yaml`) for each LWF. Here, the given parameter should be exactly the same as the required mode name, in order to find such mode correctly. For example, `ModeAverage[u]` is the mode average of mode named `u`.

The `ModeAverage` quantity is defined as the average mode value (in vector form) over all unit cells. Specifically,

$$\bar{X} = \frac{1}{N}\sum_i X_i,$$

where $\bar{X}$ is the `ModeAverage` for mode X, i is the unit cell index, and N is the number of *real* unit cells in the supercell.

> **Note**
>
> The unit cells containing *only* vacuum sites are not considered as a real unit cell.

`ModeAmp`

The `ModeAmp` physical quantity takes one parameter: the mode name, as similar to `ModeAverage`.

This quantity is defined as the average mode amplitude over all unit cells:

$$\bar{|X|} = \frac{1}{N}\sum_i |X_i|,$$

where $|X_i|$ is the amplitude (norm) of mode X in unit cell indexed i.

`QModifiedModeAverage`

The `QModifiedModeAverage` physical quantity takes two parameters: the mode name (as similar to `ModeAverage`), and the q-point specification. The latter could be given in two different forms: (1) provide the q-point coordinate explicitly; and (2) use special q-point symbols.

For the first form, the fractional coordinate of the q-point in the first Brillouin zone (of the unit cell) should be provided with three floating-point numbers, corresponding to the fractional coordinate along the three basis vectors of the reciprocal space, split by colons (`,`), and enclosed by brackets (`()`). For example, `(0.0, 0.0, 0.0)` is the center of the Brillouin zone (usually marked as Γ point), and `(0.0, 0.0, 0.5)` is a point at the edge of the Brillouin zone.

For the second form, several pre-defined symbols could be used, including:

- `GM`, equivalent to `(0.0, 0.0, 0.0)`.
- `R`, equivalent to `(0.5, 0.5, 0.5)`.
- `M1`, equivalent to `(0.0, 0.5, 0.5)`.
- `M2`, equivalent to `(0.5, 0.0, 0.5)`.
- `M3`, equivalent to `(0.5, 0.5, 0.0)`.
- `X1`, equivalent to `(0.5, 0.0, 0.0)`.
- `X2`, equivalent to `(0.0, 0.5, 0.0)`.
- `X3`, equivalent to `(0.5, 0.0, 0.5)`.

Examples of the physical quantity specification: `QModifiedModeAverage[u;R]`; `QModifiedModeAverage[w;(0.5, 0.25, 0.25)]`.

The `QModifiedModeAverage` is calculated as:

$$\bar{X}(q) = \mathrm{Re}\frac{1}{N}\sum_i X_i \exp(-2\pi i q\cdot R_i) = \frac{1}{N}\sum_i X_i \cos(2\pi q\cdot R_i),$$

where q is the q-point (in fractional coordinate in the first Brillouin zone), and R_i is the coordinate of the unit cell indexed i (in fractional coordinate of the *unit cell*, thus the values of R_i are only integers). The second equal sign comes from the fact that all mode values are real numbers.

9.3 Fitting configuration files

Here we introduce the configurations used in parameter fitting, including OTF fitting and other advanced fitting techniques (like `merge_otf_fit`, `reuse_data_fit`, etc). Similar to the simulation configuration part, this section is also organized by the top-level TOML tables. Each fitting method uses a subset of these tables, along with some tables used in simulation (introduced in the previous section). The following tables, used in simulations, are also used for most of the fitting tasks:

- `file_path`
- `cell`
- `ewald`
- `energy`
- `physical`
- `init`

These tables should be configured as in the simulations (i.e., with the same set of available keys). However, for some tables, not all the keys are used by the fitting process; the actual meaning of the keys are interpreted by the fitting process.

9.3.1 `struct_proj` table

The `struct_proj` table configures the structural projection process that links the atomic structure and the local mode/strain DoF motions, see Fig. 1 of [ALFE-H-2025].

The keys in this table are mainly internally used and experimental. We do not recommend to change their values.

9.3.2 `fit_general` table

The `fit_general` table controls some basic settings of the fitting process, shared by most of the fitting processes.

`unitcell_spec`

type
String (file path)

default
`unitcell_spec.toml`

This key defines the file path for loading the unit cell specification file. See *Prepare unit cell metadata*.

`output_filename`

type
String (file path)

default
`heff_params_fit.yaml`

This key defines the file path for the fitted parameter file, containing the fitting result. This file could be used in later simulation after the fitting is finished.

params_precalc_yaml

type
String (file path)

default
`heff_params_precalc.yaml`

This key defines the file path for loading the pre-calculated parameters (see *Precalculation*).

9.3.3 pader table

This table controls the process for obtaining the derivatives of the potential energy and its gradient over the DoFs (local modes and strain), which is one essential step in the parameter fitting process.

fix_eps_inf

type
Boolean

default
`true`

This key determines whether the `eps_inf_inv_factor` parameter (the scale of optical dielectric constant) is considered to be fixed during the fitting. If `false`, the `eps_inf_inv_factor` parameter is fixed to its initial value (typically just 1) during the fitting, and the long-range dipolar part of the effective Hamiltonian is completely fixed. Otherwise, such factor will be fitted during the fitting, as a free parameter.

> **Warning**
>
> This key must be kept to the same value for the whole fitting process (including continuing fittings). In other words, the continuing fitting (mainly OTF fitting) **cannot** change this value. Otherwise, the result may be incorrect.

fix_interactions

type
Array (of Integers)

default
empty

This key specifies the indexes for the local INTs whose parameters will be fixed (to their initial values) during the fitting process. The default empty value means all parameters will be fitted. See also *Generate interactions*.

interact_controls (Array of Tables)

type
Array (of tables)

default
empty

This key provides a more precise control of the fixed parameters (than `fix_interactions`). Each element of this array should be a table with the following introduced keys. See also *Generate interactions*.

Example:

```
[pader]

[[pader.interact_controls]]
index = 0
fix_sats = [0, 1, 2]

[pader.interact_controls]
index = 1
fix_sats = [4]
```

interact_controls.index

type
Integer

default
-1

This key specifies the index of the INT to be controlled by this sub-table. This key must be provided by the user, as the default value -1 is invalid.

interact_controls.fix_sats

type
Array (of integers)

default
empty

This key specifies the indexes of the SATs inside the controlled INT (see `interact_controls.index`) whose parameter will be fixed during the fitting.

9.3.4 `regression` table

The `regression` table controls the configuration of regression method that shared by most of the regression algorithms.

method

type
String (of Literals)

options
`LeastSquare`, `BayesianLinear`, `RidgeLinear`

default
None

This key selects the algorithm of the regression. For all fitting tasks, this key must be specified by the user. The available options includes:

- `LeastSquare` Simple linear least-square fitting. Provides no uncertainty prediction and cannot be used in active fittings.
- `BayesianLinear` Bayesian linear regression (BLR). The main fitting method for OTF fitting.
- `RidgeLinear` Ridge linear regression. Experimental.

param_selection_yaml

type
String (file path)

default
empty

If specified, the parameter selection YAML file will be loaded and applied. This will disable some selected parameters. This feature is experimental.

variable_ref_energy

type
String (Literals)

options
`Yes`, `No`, `Auto`

default
`Auto`

This key specifies whether the reference energy during the parameter fitting process is variable. See also *Configuring the first OTF MD run*.

9.3.5 `linear` table

The `linear` table controls the linear fitting regression algorithms. These configurations are shared by most of the linear methods, including least-square, Bayesian linear, and Ridge regressions.

weight_energy

type
Float

default
1.0

This key defines the weight for the potential energy (per unit cell) in the regression process. The default 1.0 means no extra weight imposed. Note that another weight corresponding to the number of unit cells in the supercell is automatically imposed in the regression process and not included here. For the form of the weights, see [ALFE-H-2025].

weight_strain

type
Float

default
1.0

This key defines the weight for the derivative over strain (i.e., stress) in the regression process. The default 1.0 means no extra weight imposed. See also [ALFE-H-2025].

weight_modes

type
Array (of Floats)

default
none

This key determines the weight for the derivatives over local modes (i.e., force on the modes) in the regression process. If provided, the length of the array must be equal to the number of LWFs inside the unit cell, with each element corresponds to one LWF in the unit cell. The default value is None, meaning no extra weight imposed.

9.3.6 `bayesian` table

The `bayesian` table defines some details while using the Bayesian linear regression (BLR) algorithm (`regression.method = "BayesianLinear"`).

`inf_sigw`

type
Boolean

default
true

This key determines whether the σ_w hyperparameter is treated as infinity ($\sigma_w \to \infty$) during the regression process. For a discussion, ses [ALFE-H-2025].

`iter_eps_v`

type
Float

default
1e-6

This key determines the maximal allowed difference of σ_v^2 value between two iterations upon convergence.

`iter_eps_w`

type
Float

default
1e-4

This key determines the maximal allowed difference of σ_w^2 value between two iterations (absolute or relative values) upon convergence.

`iter_steps`

type
Integer

default
500

This key determines the maximal allowed self-consistent iteration steps in BLR. If `inf_sigw = true`, this key has no effect.

`iter_mix`

type
Boolean

default
false

This key determines whether mixing strategy is applied in BLR, which may be helpful for convergence in some cases. If enabled, the mixing will be enabled after the step indexed `iter_mix_step` (see later), and the new value for next iteration of σ_v^2 will be `iter_mix_scale` (see later) times the computed new value in this step plus (1 - `iter_mix_scale`) times the value in last step. Similar mixing also applied for σ_w^2. If the mixing is not enabled, it is equivalent to `iter_mix_scale = 1`.

> **Warning**
>
> The feature of mixing is experimental.

iter_mix_step

type
Integer

default
30

See `iter_mix`.

min_iter_steps

type
Integer

default
2

This key determines the minimal steps for the iteration when `inf_sigw = false`.

iter_mix_scale

type
Float

default
0.2

See `iter_mix`.

eigen_low

type
Float

default
1e-10

This key determines the threshold of the eigenvalue of the $\Phi^T\Phi$ matrix. Below which the eigenvalue is considered to be zero. See [ALFE-H-2025].

init_v2

type
Float

default
0.1

This key determines the initial value of σ_v^2 for BLR iteration. The value is only used in the first BLR regression during the fitting process and when the BLR has failed. For other BLR regressions, the σ_v^2 and σ_w^2 in last regression will be used.

init_w2

type
Float

default
10.0

This key determines the initial value of σ_w^2 for BLR iteration. Similar to `init_v2`.

verbose_iter

type
Boolean

default
false

Controls whether extra (verbose) output will be generated during the iterations.

allow_fall_back_to_inf_sigw

type
Boolean

default
true

This key controls whether the BLR iteration with `inf_sigw = false` could be fallen back to `inf_sigw = true` when the estimated σ_w^2 during the iteration exceeds certain threshold (`max_sigw2`).

max_sigw2

type
Float

default
1e10

See `allow_fall_back_to_inf_sigw`.

fall_back_to_inf_sigw_if_failed

type
Boolean

default
true

This key controls whether the BLR iteration with `inf_sigw = false` could be fallen back to `inf_sigw = true` when the BLR fails to converge. This includes two cases: whether the σ_w^2 or σ_v^2 is NaN, or the BLR does not converge in the `max_iters` steps.

9.3.7 `ridge` table

This table controls details for Ridge linear regression (`regression.method = "RidgeLinear"`). This feature is experimental.

9.3.8 `active` table

The `active` table controls the configurations for active-learning-based fitting processes, including on-the-fly (OTF) fitting, reuse data fitting, etc.

`use_realistic_uncertainty`

type
Boolean

default
false

In the BLR framework, the predicted uncertainty of energy, force and stress contain two terms (see Eq. 14 of [ALFE-H-2025]):

$$\sigma = \sigma_v^2 I + \phi \Sigma \phi^T,$$

where I is the identity matrix, σ_v is a hyperparameter, ϕ is the so-called design matrix (of the new structure to be predicted), and Σ holds the information of uncertainty for the Heff parameters. As suggested by [VaspMLFF2019] (see Eq. 30 therein and the followed discussions), we typically use only the second term as the uncertainty prediction. This is called (in our codes and document) *predictive* uncertainty. In contrast, it is also possible to use the summation of both terms anyway, as the definition above. This is called *realistic* uncertainty.

The default behavior of this package is to use the predictive uncertainty. To use realistic uncertainty, set this key to true.

`fp_criterion`

type
String (Literals)

options
`direct`, `relative`, `energy`, `force`, `stress`

default
`direct`

This key determines the type of uncertainty to be used as the criterion for determining whether first-principles (FP) calculations are needed. As indicated by the definition (see Eq. 14 of [ALFE-H-2025]), the predicted uncertainty has different values for energy, force, and stress. For simplicity, currently we choose to use only one *scalar* value as the uncertainty (sometimes also called "Bayesian error") to determine FP calculations. This key determines how this scalar value is selected or calculated. The threshold for determining the FP calculations is consistent with the choice here (including the threshold value and unit).

This key has the following options:

- `direct`. In this case, the scalar uncertainty is the maximal value of the "direct uncertainty" which is directly computed in the regression process, which is dimensionless, but with the weights removed (including the weight imposed by the number of unit cells and by the user). For the dimensionless value definition and the weight, see the discussions in [ALFE-H-2025] (between Eqs. 9 and 10). In this case, the uncertainty value (and also the threshold) is dimensionless.
- `relative`. In this case, the scalar uncertainty is the maximal value of the "relative uncertainty" which is obtained by dividing the direct uncertainty by the $\sigma_v 2$ hyperparameter. This is similar to some other MLFF implementations. In this case, the uncertainty value and the threshold are dimensionless.

- `energy`. In this case, the scalar uncertainty is the uncertainty of the potential energy per unit cell, in the unit of eV.
- `force`. In this case, the scalar uncertainty is the maximal uncertainty of the force (more strictly, the gradient of the potential energy over the local modes), in the unit of eV/Angstrom.
- `stress`. In this case, the scalar uncertainty is the maximal uncertainty of the stress (more strictly, the gradient of the potential energy over the strain, divided by the number of unit cells), in the unit of eV.

In most of our tests, we found the `energy` criterion suitable.

Important

If the uncertainty threshold is determined dynamically (`thres_strategy = "Dynamic"`), this key must be kept as long as the uncertainty history is *not* cleared (between continuing fittings). Otherwise the calculated threshold will not be well-defined.

`high_thres_rate`

type
Float

default
2.0

In the active fitting process, if the predicted uncertainty exceed the so-called *high threshold*, FP calculation and parameter fitting will be triggered immediately (regardless of how many steps have passed since last FP and fit). This *high threshold* is defined as the FP threshold (either dynamically updated or determined by the input, see `thres_strategy`) multiplies a factor. The factor is set by this key.

`collect_ref_energy`

type
String (Literals)

options
`Auto`, `Yes`, `No`

default
`Auto`

Determines whether the reference energy during the fitting process will try to be collected at the first step. See also *Reference energy*.

`fp_distance`

type
Integer

default
20

Determines the minimum distance (in unit of steps) between two FP calculations. To avoid running FP calculations with very similar atomic structures (especially in OTF fitting), the FP calculations are typically separated by at least the number of steps set by this key. For example, if the value is set to 20, the FP calculations will not be executed, even the predicted uncertainty exceeds the FP threshold, if the steps since last FP calculation is less than 20.

There is an exception of this rule. If the predicted uncertainty exceeds the high threshold (see `high_thres_scale`), the FP calculation will be executed immediately, regardless of the number of steps passed since last FP.

fit_distance

type
Integer

default
0

This key determines the distance (in unit of FP calculations) between two parameter fitting calls during the active fitting. The default value 0 means the parameter will be fitted each time after the FP calculation.

min_fit_fps

type
Integer

default
3

This key sets the minimal number of FP calculations required for the first parameter fitting in the active fitting process.

allow_failed_fits

type
Integer

default
3

During the active fitting process, the parameter fitting may sometimes fail. This field determines the maximal allowed *continuous* fittings. If the parameter fitting failed *continuously* for the configured times, the whole active fitting process will be stopped.

thres_strategy

type
String (Literals)

options
`Dynamic`, `Fixed`

default
`Dynamic`

This key determines the strategy for updating the FP threshold during the active fitting process. Currently, there are two options.

- `Dynamic`. The FP threshold will be updated based on the uncertainty history. See *Configuring the first OTF MD run*.
- `Fixed`. The FP threshold is kept to the value set by `thres_value`.

thres_value

type
Float

default
-1

This field sets the FP threshold value. If the value is negative (as default), the value will be ignored, as if it is not set. If `thres_strategy = "Dynamic"`, this field is optional, providing the initial threshold value. If `thres_strategy = "Fixed"`, this field is mandatory. One positive value must be given, as the threshold value to be used.

The unit of the threshold is the same as the scalar uncertainty (Bayesian error). See `fp_criterion`.

thres_init_count

type
Integer

default
10

This field determines the minimal number of sampled uncertainty value after fitting (σ_i, see *Configuring the first OTF MD run*) for initializing the FP threshold (σ_t) for the first time in active fitting. The default value 10 indicates that the threshold will be tried to update after 10 fittings.

thres_average_count

type
Integer

default
10

This field determines the number (n) of sampled uncertainty values after fitting (σ_i) to be averaged for updating the FP threshold. See *Configuring the first OTF MD run*.

thres_update_std_rate

type
Float

default
0.3

This field determines the maximal allowed ratio of standard deviation over average value of sampled uncertainty values (σ_i) for updating the FP threshold. See *Configuring the first OTF MD run*.

thres_init_std_rate

type
Float

default
0.3

Same as `thres_update_std_rate`, but for the first time the FP threshold will be determined.

thres_update_scale

type
Float

default
0.98

This field determines the multiplier (a) on the averaged collected uncertainty values after fitting ($\bar{\sigma}$) for updating the threshold. See *Configuring the first OTF MD run*.

> **Tip**
>
> As suggested by VASP document for its MLFF implementation (see [VaspMLFF2019], and its document), we currently use a default value which is less than 1.0, which tends to lower the threshold and call FP calculations more frequently, to improve the parameter precision. However, we sometimes find (in some testcases) this would lead to too large number of FP calculations. In this case, this field may be set to larger value (e.g., 1.2 or even 2.0). The default value may also be changed in later versions.

thres_update_without_fp

type
Boolean

default
false

> **Warning**
>
> This feature is highly experimental. The related behavior of the program may be changed in future versions.

If set to true, the threshold is updated based on the observed uncertainties of the steps where FP is *not* executed. See *Controlling FP threshold* for more information.

thres_average_count_without_fp

type
Integer

default
100

The number of steps n considered while updating threshold when `thres_update_without_fp = true`.

See *Controlling FP threshold*.

thres_update_std_rate_without_fp

type
Float

default
0.2

The maximal allowed standard deviation ratio $D_\sigma/\bar{\sigma}$ value p_σ while updating the threshold when `thres_update_without_fp = true`.

See *Controlling FP threshold*.

thres_update_scale_without_fp

type
Float

default
1.1

The scale of new threshold while updating threshold (a) when `thres_update_without_fp = true`.

See *Controlling FP threshold*.

`freeze_thres_steps_after_restart`

type
Integer

default
0

Experimentally available since Apr. 18, 2026.

Freeze the threshold value (when `thres_strategy = "Dynamic"`) after the restart of OTF fittings (including the beginning of any continuing OTF fitting) for the steps set by this field. The default value 0 means no such freezing operation taken. If the fitting is not an OTF fitting, this field has no effect.

This is typically used along with `otf.rescale_thres_after_restart`.

`check_stop_interval`

type
Integer

default
1000

This field determines the frequency (number of interval steps) for checking whether stop is required (via `stop.toml`) if FP is not called. The checking is always performed before any FP calculations. See *Termination of OTF MD fitting*.

`uncertainty_log_csv`

type
String (file path)

default
`uncertainty_log.csv`

This field determines the filename for writing the uncertainty log. Non-empty value must be given. This is a CSV output file.

`fp_log_csv`

type
String (file path)

default
`cal_log.csv`

This field determines the filename for writing the FP calculation log. Non-empty value must be given.

`active_fit_out_json`

type
String (file path)

default
`active_fit_out.json`

This field determines the filename for writing the active fitting data file, which is required for continuing simulations (see *Continuing the OTF fitting*). Use empty string to disable this output.

fit_info_log_yaml

type
String (file path)

default
`param_fit_log.yaml`

This field determines the filename for writing the parameter fitting log file, including the fitted parameter from each parameter fitting call. This file is in YAML format and may be read by other programs. Use empty string to disable this output.

fit_summary_csv

type
String (file path)

default
`param_fit_summary.csv`

This field determines the filename for writing the parameter fitting summary file, including the fitting root mean square error (RMSE) data, etc. This file is in CSV format. Non-empty string must be given.

continuing

type
Boolean

default
false

This key is the switch for continuing active fitting (see *Continuing the OTF fitting*). If this is set to true, the active fitting continuing data file (configured by `active_fit_in_json`) will be loaded, a parameter fitting will be called before further fitting.

Important

Always remember to turn on this flag before running continuing fitting.

clear_uncertainty_history

type
Boolean

default
false

If this is set to true, the uncertainty history will be cleared while loading continuing data file. By default, the uncertainty history (for determining the uncertainty threshold) is kept in continuing OTF fittings.

This configuration may be useful when the continuing simulation has a rather different structure, mixed site element concentrations, atomic distributions, or simulation conditions, that may lead to very different uncertainty values from previous OTF runs.

running_state_file

type
String (file path)

default
`active_fit_running_state.toml`

This field specifies the filename for the running state file, which is a TOML file written by this program, containing the basic information about the active fitting process. A non-empty string must be given.

This file is mainly used for checking the running status while a new active fitting process is tried to be created. If another active fitting process is run in current working directory, as indicated by this file, the new active fitting process which is tried to be created will be rejected. By default, the program would set the running status to `false` if it finishes normally, or with most of the exceptions by the program itself. However, in some special cases (e.g., the program is terminated by outside controller, for example, the task is killed), the file will not be updated. In this case, if a new active fitting process is desired in the same directory, please remove this file at first.

write_running_state_file

type
Boolean

default
true

This field controls whether the running state file will be written. See `running_state_file`.

skip_running_check

type
Boolean

default
false

If set to true, the active fitting status (see `running_state_file`) will not be checked before starting a new active fitting process.

> **Warning**
>
> We strongly discourage to skip this running state checking, because running more than one active fitting process in one working directory simultaneously will cause very severe data conflicting.

max_fp_count

type
Integer

default
0

This field sets the maximal allowed FP calculations during this active fitting process. If the number of FP calculations exceeds this count, the active fitting process will be stopped. The default value 0 indicates no such limitation.

stop_after_last_fp

type
Integer

default
0

Available as of Sep. 25, 2025.

Stop the simulation after the configured number of steps passed since last FP calculation. If set to 0, this feature is disabled. This feature prevents the active fitting from running too many steps without any FP calculations.

9.3.9 otf table

This `otf` table (short for on-the-fly) controls the details of OTF fitting, which is a type of active learning powered by MD simulation.

extract_init_modes

type
Boolean

default
false

If set to true, the initial structure (provided by *atomic* structure) will be loaded as the initial state of the MD simulation. Otherwise, the initial structure is built according to the `init` table, as in normal MD simulations. The initial structure filename is specified by `init_struct_file`, with format specified by `init_struct_format`. For the file formats, please see *ASE structure files*.

Warning

If this is set to true, the provided structure file must be consistent with the defined cell setting (in the same Cartesian coordinate system, and the same supercell matrix). Otherwise undefined behavior may occur. The program does not guarantee to detect such inconsistency.

init_struct_file

type
String (file path)

default
`POSCAR_init`

Determines the filename for initial structure when `extract_init_modes = true`.

init_struct_format

type
String (Literals)

default
None

Determines the file format for initial structure when `extract_init_modes = true`. See also *ASE structure files*.

write_sites_txt

type
String (file path)

default
`sites.txt`

If a non-empty file path is given, the initial site occupation will be written to the specified file.

occupations_history_yaml

type
String (file path)

default
`occupations_history.yaml`

If non-empty file path provided, the atomic occupation history will be written to the provided file in YAML format. The file is basically a YAML Map, with keys being the step indexes and the values being the occupation indexes of each site in the supercell.

This file may be later used by other programs like `merge_otf_fit`, and may be converted to `sites_txt` files for input of Heff simulation (including the initial structure for OTF) (see utility program `heff_gen.fit.cui.extract_occupation_history`).

restart_interval

type
Integer

default
0

This field determines the interval (in number of MD steps) to restart the OTF MD simulation. Here, the "restart" indicates the configuration (structure) is reset to the reference structure and MD simulation starts from this structure. All FP data and active learning history data (uncertainty history, threshold, etc) are kept. If set to 0, this restarting is not performed.

This restart may be useful for improving the phase space exploring efficiency which may be helpful for improving the OTF fitting efficiency. See also *Configuring the first OTF MD run*.

reset_momenta_interval

type
Integer

default
0

This value determines the interval (in number of MD steps) to reset the momenta in the OTF MD simulation. On the required steps, the momenta will be regenerated (reinitialized) according to the configured temperature. This is provided for (possibly) improving the OTF exploring efficiency. Set this value to 0 to disable this resetting.

reset_occupations_interval

type
Integer

default
0

This value determines the interval (in number of MD steps) to reset the atom occupations (for mixed sites). For models without mixed sites, this is not effective. Set this value to 0 to disable this feature.

Note that this occupation resetting is only available for *invariable* reference energy (see *Reference energy*).

check_pader_interval

type
Integer

default
100

This field is available as of Sep. 14, 2025.

Set the interval (in number of MD steps) for checking the parameter derivatives (SAT and PCG) computations to find possible errors. Set to 0 to disable this checking (not recommended). For the old versions without this key, this is equivalent to set this key to 1, meaning such check is performed every step.

reset_momenta_after_fp

type
Boolean

default
false

Experimental feature available as of Jul. 24, 2025.

If set to true, the momenta will be redistributed (according to the configured temperature) after each FP calculation.

restart_on_fp_failed

type
Boolean

default
false

If set to true, restart the OTF fitting (as configured by `restart_interval`) when FP calculation fails. Otherwise, stop the OTF fitting when FP calculation fails.

restart_on_strain_values

type
Float

default
0.5

Restart the OTF fitting when the absolute value of any component of the strain tensor exceeds the configured value. Set to 0 to disable this feature.

This is provided to prevent the OTF fitting from entering non-physical structures with very large strain (which is typically not possible for realistic materials).

update_thres_next_step

type
Boolean

default
false

If set to `true`, the FP threshold will be updated based on the uncertainty of the *next step* of the step that the parameters is fitted. Otherwise, it is updated based on the uncertainty predicted for the same structure after the parameter is fitted.

The behavior of setting this to `true` is consistent with the algorithm (for updating the threshold) described in [VaspMLFF2019].

`rescale_thres_after_restart`

type
Float

default
1.0

Experimental feature since Apr. 18, 2026.

In OTF fittings, if the threshold is updated dynamically (`active.fp_strategy = "Dynamic"`), it will be rescaled by a factor defined by this variable after each time the simulation is restarted (see also `restart_interval`). The beginning of the continuing OTF fitting is also treated as a restart here. The default value 1.0 means the threshold is not changed. If threshold is not updated since the last time that the threshold is rescaled by this feature, such rescale is not performed, to prevent the threshold being scaled repeatedly.

This option may be useful when the uncertainty changes drastically after restart operations. In such case, try to set this value to some value larger than 1.0.

Tip

`active.freeze_thres_steps_after_restart` may be used along with this key.

9.3.10 `intrusive_md` table

The so-called intrusive MD is the underlain MD algorithm for OTF MD. Here, the "intrusive" means (technically) each MD step is split into several parts and are called by other flow (with respect to the library implementing such MD algorithm; in OTF MD, by the OTF fitting process) in some proper position and order. Such external call to the parts of MD step is considered as "intrusive" for the MD step, as compared to ordinary MD algorithms.

The `intrusive_md` table controls some details for such MD process, which is not shared by ordinary MD simulations.

`dof_steps_file`

type
String (file path prefix)

default
`dof-steps`

suffix
`.h5`

This field determines the filename for the step-DoF file, which is an HDF5 file. This file includes the DoF information in *each* step of the intrusive MD simulation. Use empty string to disable this output. Typically, in common practice of OTF fitting, the `heff_dof_potential` utility program uses the data from this file.

9.3.11 `fp` table

The `fp` table controls the first-principles (FP) calculations executed by the OTF fittings (or other fitting methods that actually calls FP calculations).

`calculator`

type
String

default
empty

This field specifies the type of FP calculator. For any fitting processes using FP calculations, this field must be specified with a non-empty value.

This field should be filled with the *class name* of the calculator (inherited from ASE calculator). See *Customize the FP calculators* for more details.

`module`

type
String

default
empty

This field specifies the Python module for loading the calculator. If empty, the calculator is loaded (possible recursively) first from the `heff_gen.fit.fp_interface.calculators` package, and then (if not found) from `ase.calculators` package. If provided, it must be a full name of the module containing the calculator class (`calculator` field) that could be directly imported.

`directory`

type
String (file path)

default
`fp_dir`

This field determines the directory at which the FP calculation will be executed. If such directory does not exist, the program will create it.

`disallow_calculation`

type
Boolean

default
false

If set to `true`, any *actual* calculation will be disallowed. If it is required by the program, the fitting will be stopped.

In the OTF fitting process, the program will check the ASE checkpoint containing the previous calculation results (if any) before running actual FP calculation. The checkpoint will be created after each actual FP calculation. If the loaded checkpoint is consistent with current required FP calculation (both structure and FP configurations), the actual FP calculation will not be performed, and the result will be loaded from the checkpoint directly. With such feature, the re-running of OTF fitting at the same directory may just use (some of) the FP results previously collected.

> **Note**
>
> Re-running OTF fitting at same directory does not guarantee the MD trajectory is the same as previous run, even with the same configuration. This difference may come from the numeric errors. Thus, the FP results stored by previous OTF runnings are not guaranteed to be loaded by a re-running even with exactly the same configurations.

check_calculator_info_for_checkpoint

type
Boolean

default
true

By default, the program will check the calculator information (mainly the configurations) stored in the checkpoint files before using it as the FP result. If the configuration does not consistent with currently required FP calculation, the checkpoint will be dropped.

If this field is set to `false`, such check is not performed. We strongly *discourage* to do so.

remove_wave_function_steps

type
Integer

default
10

For some FP calculators, the wave function files may be saved and loaded by a later calculation, which may speed up the convergence if the two structures are close enough. However, if the structures are considerably different, loading the wave function file is not favorable, or even harmful, to the convergence.

For some supported calculators, the program will try to remove the wave function file (if any) before actual FP calculation, if the distance between last FP calculation and current FP calculation exceeds the configured value by this field (in number of MD steps). For these calculators, the program will also try to remove the wave function file before the first FP calculation required by current OTF fitting process.

Currently, this is only supported for `Vasp` calculator (defined in `heff_gen.fit.fp_interface.calculators.vasp` module)

checkpoint_tolerance

type
Float

default
1e-7

This field specifies the tolerance allowed for the comparison of structures to consider them as consistent structures, while loading the checkpoints from previous FP calculations. Such check is implemented using the `ase.calculators.checkpoint.atoms_almost_equal` function.

skip_convergence_check

type
Boolean

default
false

If the calculator has an attribute called `converged`, such attribute is checked after FP calculation. If the result is false, the FP calculation is considered to be failed. If otherwise the calculator does not have such attribute, a warning message is generated.

To skip the whole convergence checking process (and also disable the warning message if such attribute does not exist), set this value to false.

kwargs

type
Table

default
empty

This field is a TOML table. Any extra arguments passed to the calculator constructor could be passed via this. The configurations in this table will be passed to the calculator constructor using the `**kwargs` syntax after converting this TOML table to Python dictionary.

For ASE calculators, please refer to their document for the available parameters: https://ase-lib.org/ase/calculators/calculators.html

Example:

```
[fp]
calculator = "Vasp"

[fp.kwargs]
command = "mpirun vasp_std"
xc = "pbe"
gga = "PS"
encut = 550
ediff = 1e-6
ismear = 0
sigma = 0.05
isif = 2
prec = "Accuracy"
lreal = "Auto"
kspacing = 0.3

[fp.kwargs.setups]
Ba = "_sv"
Ti = "_sv"
O = ""
```

This sets up a calculator for VASP software. The calculator class is `heff_gen.fit.fp_interface.calculators.vasp.Vasp`, which is a slightly extended version of ASE vasp calculator class `ase.calculators.vasp.Vasp`. The configuration is equivalent to the following calculator setup in Python:

```
calc = Vasp(
    command="mpirun vasp_std",
    xc="pbe", gga="ps", encut=550, ediff=1e-6, ismear=0, sigma=0.05,
    isif=2, prec="Accuracy", lreal="Auto", kspacing=0.3,
    setups={"Ba":"_sv", "Ti":"_sv", "O": ""}
)
```

As the calculator class in `heff_gen` does not change the constructor, the available keys are provided by the ASE calculator, which is available at https://ase-lib.org/ase/calculators/vasp.html#module-ase.calculators.vasp

9.3.12 `merge_otf` table

The `merge_otf` table is provided for `merge_otf_fit` (*merge_otf_fit*).

`data_dirs`

type
Array (of Strings)

default
Empty

This field is essential for `merge_otf_fit`, a non-empty array of string must be given. Each element of the array is a path of one data directory (relative path or absolute path), which is a directory at which the OTF fitting has been finished. The program will go through each of them to pick all the FP results generated in the previous OTF fitting to fit the parameters.

`continuing_first`

type
Boolean

default
false

By default, the `merge_otf_fit` process will ignore all `active.continuing` keys, the continuing active fitting data will not be loaded for each data directory. If the user would like to load the continuing data for the *first* data directory, set this field to true.

`no_fit`

type
Boolean

default
false

If true, generate only the merge data result, without a parameter fitting.

`continuing_data_file`

type
String (file path)

default
`active_fit_in.json`

The continuing data file when `continuing_first = true`.

`active_fit_out_json`

type
String (file path)

default
`active_fit_out.json`

This key controls the active fitting continuing data file name. For some loading data algorithms, this file *in the data directory* would be optionally loaded to gain some information of the OTF fitting.

data_fit_config_file

type
String (file path)

default
`fit_config.toml`

This field controls the fitting configuration file of each data directory (relative to the data directory).

data_fp_dir

type
String (directory path)

default
`fp_dir`

This field controls the FP calculation directory of each data directory (relative to the data directory).

dump_json_file

type
String (file path)

default
`merge_otf_data.json`

This field controls the output file name containing all the data collected from this fitting.

This file has the same format as the active fitting continuing data file, but contains no uncertainty history file.

load_data_method

type
String (Literals)

options
`LoadStructFast`, `LoadStructSafe`, `RepeatOtf`

default
`LoadStructFast`

Available as of Sep. 16, 2025.

This field controls the method used for loading the FP data. Typically, this field need not to be changed, unless problems encountered by the default configuration.

The `LoadStructFast` and `LoadStructSafe` methods analyze the atomic structures directly. The atomic structures (on which FP calculations have performed) are loaded and extracted (during which each atom is assigned to one unit cell and one site inside the unit cell). The difference between the two methods is that the `LoadStructFast` method makes use of the occupations history file, and use the same Heff supercell layout when the occupation has not changed between current and last structure, while the `LoadStructSafe` method extracts the atomic structure directly for every structure. The former should be much faster than the latter.

These two methods should work well when the atomic displacement (with respect to the reference structure) of the loaded structures are not large comparing to the unit cell, and thus each atom could be assigned undoubtedly to one site in one unit cell. If in some cases, the atomic displacement is large enough in some of the structures, such structure extraction process may failed. In these cases, the `RepeatOtf` method may be used. In the `RepeatOtf` method, the

program first change the Heff supercell layout when occupation has changed, and then load the structure to fill in this layout. No structure extraction taken place. This process is simulating the structure generation process in OTF fitting.

> **Note**
>
> We have to emphasize that the Heff is built based on Taylor expansion. Thus, if the atomic displacements are large, the efficiency of the expansion may be questionable. One should consider whether the structure is reasonable, or even whether Heff is applicable to current problem.

9.3.13 `reuse_data` table

The `reuse_data` table controls the `reuse_data_fit` process (see *reuse_data_fit*).

`data_dir`

type
String

default
.

This field controls the data directory for loading the FP data. This directory must *not* be current running directory.

`fp_dir`

type
String

default
`fp_dir`

This field controls the FP calculation directory in the data directory.

`continuing_data_only`

type
Boolean

default
false

If set to true, only the continuing data is loaded, the FP data is not actually loaded.

`read_occupations_history_yaml`

type
String (file path)

default
`occupations_history.yaml`

This field determines the filename for loading the occupation history from the data directory.

`write_occupations_history_yaml`

type
String (file path)

default
`occupations_history.yaml`

This field determines the filename for writing the occupation history in current working directory.

`max_fp_count`

type
Integer

default
0

The maximal count of FP data to be loaded. 0 for unlimited.

`select_data`

type
Boolean

default
false

This feature is experimental.

If true, the loaded data will be selected by its predicted uncertainty (as in the active fitting process). Otherwise, all the loaded data will be used directly.

`check_struct_fitting_error`

type
Boolean

default
true

If set to `true`, the structure fitting error will be checked for each structure. This is strongly recommended. See also *Check selected local mode basis*.

`allow_struct_fitting_error`

type
Boolean

default
false

By default, non-zero structure fitting error (see *Check selected local mode basis*) is not allowed in reuse data fitting. The reused data makes sense only if the current local mode basis spans the same subspace as the previous basis that generated such data. If set to `true`, such fitting error will be allowed (not recommended).

`struct_fitting_error_tol`

type
Float

default
1e-6

Determines the maximal allowed structure fitting error (see also `allow_struct_fitting_error`) which mainly comes from numeric errors.

reuse_snap_file

type
String (file path)

default
`reuse_snap.csv`

The snap file (similar to the OTF MD snap file) name. Useful for `reuse_plot`.

dof_steps_file

type
String (file path)

default
`dof-steps.h5`

The filename for the DoF-step file similar to OTF MD.

write_sites_txt

type
String (file path)

default
`sites.txt`

Similar to `otf.write_sites_txt`.

9.3.14 mlcons table

The `mlcons` table is used in the MLCons process (see *Machine-learning-based model construction (MLCons)*).

pow_lambdas

type
Array (of Floats)

default
[1.3, 1.2, 1.1, 1.08, 1.06, 1.04, 1.03, 1.02, 1.015, 1.01, 1.008, 1.006, 1.005, 1.004, 1.003, 1.002, 1.0015, 1.001, 1.0005, 1.0]

see also
`min_lambda`

This field controls the regularization coefficient λ. In the MLCons process, the program use the λ values listed in this array in order. The values in the default configuration are those suggested by [MLMCH2020] with some modifications.

min_lambda

type
Float

default
1.0001

see also
`pow_lambdas`

Available as of Oct. 6, 2025.

Determines the minimal possible value of λ to be tried. In `pow_lambdas`, if the value is less than this `min_lambda` value, it will be ignored.

`use_validation_set`

type
Boolean

default
true

Whether validation set is used in the λ-searching process. See [MLMCH2020].

`test_set_prop`

type
Float

default
0.2

The proportion of test set out of the whole data set.

`validation_set_prop`

type
Float

default
0.4

The proportion of validation set out of the whole data set if `use_validation_set = true`.

`trial_range_add`

type
Float

default
1.0

Available as of Nov. 7, 2025.

The portion of parameters to be considered when trying to add a new parameter.

> **Warning**
>
> If set this value to some value less than 1.0, there will be some parameters that are *never considered* during the whole MLCons process, because the parameter adding trial order is determined at the beginning of MLCons process and is never changed.

`trial_range_delete`

type
Float

default
0.2

Available as of Nov. 7, 2025.

The portion of parameters to be considered when trying to delete a parameter during λ-iteration.

trial_range_substitution

type
Float

default
0.2

Available as of Nov. 7, 2025.

The portion of parameters to be considered when trying to substitute a parameter. Note that we always consider substitute only the first possible parameter (according to relative uncertainty) as indicated by [MLMCH2020].

trial_range_sbs_delete

type
Float

default
0.2

Available as of Nov. 7, 2025.

The portion of parameters to be considered when trying to delete a parameter in SBS iteration.

continuing_search

type
Boolean

default
false

see also
`init_selection_yaml`

Available as of Oct. 11, 2025.

If set to `true`, the search will starts from provided initial selection (`init_selection_yaml`). Otherwise, the search starts from empty model (with only the reference energy term).

init_selection_yaml

type
String (file path)

default
`init_selection.yaml`

see also
`continuing_search`

Available as of Oct. 11, 2025.

Initial parameter selection file when `continuing_search = true`. The input file has the same format as the parameter selection output file (for different models). The input selection file must be consistent with current interaction set, otherwise the behavior is undefined.

disable_bounding_only

type
Boolean

default
true

see also
`interact_topo_yaml`

Available as of Oct. 18, 2025.

If set to true, all the bounding-only SATs (see *Bounding-only interactions*) will not be considered in the MLCons procedure (i.e., not added to the model).

This requires the interaction topology file as input (`interact_topo_yaml`).

interact_topo_yaml

type
String (file path)

default
`heff_interactions_topology.yaml`

see also
`disable_bounding_only`

Available as of Oct. 18, 2025.

The interaction topology input file. This file is generated in the interaction generation stage (*Generate interactions*).

The default value was `interactions_topology.yaml` in version 0.19.

max_steps_for_lambda

type
Integer

default
500

Available as of Nov. 2, 2025.

Maximal steps to be executed for a given λ value. 0 for unlimited.

add_by_order_distance

type
Boolean

default
false

Available as of Nov. 7, 2025.

This field determines the trial order while adding new SATs (parameters) to selection.

At the beginning of the MLCons process, all the optional parameters are sorted according to some criterion (see below). When a new parameter is to be added, the program goes through all optional parameters in such order from the beginning, until a new parameter satisfying the variance requirement is found.

By default, the parameters are sorted by their relative uncertainty in the maximal model. If this key is set to true, the parameters are sorted by the orders and distances of the corresponding SATs, which may be called "order-distance sort". In such order-distance sort, the SATs are sorted first by their polynomial orders (i.e., the number of mode and strain COMPs in each PROD), and then by their alloy order (number of alloy COMPs in each PROD), and finally by the distances between the COMPs.

> **Note**
>
> Setting this key to `true` requires the interaction topology file to be available (see `interact_topo_yaml`). Besides, the order-distance sort require some metadata in the interaction json file which is only available as of Oct. 27, 2025. Interaction set files generated with the program prior to this version are not supported.

`incremental_adding`

type
Boolean

default
true

Available as of Nov. 11, 2025.

Use the incremental algorithm while searching for new parameters to add. This is typically faster than the legacy algorithm.

`incremental_deletion`

type
Boolean

default
true

Available as of Nov. 11, 2025.

Use the incremental algorithm while searching for new parameters to delete. This is typically faster than the legacy algorithm.

`keep_params_selected`

type
Array (of integers)

default
empty

Available as of Nov. 20, 2025.

Keep the given parameters (in parameter indexes) selected during the MLCons process.

`data_file`

type
String (file path)

default
`linear_data.json`

The input file containing the linear fitting data.

original_param_file

type
String (file path)

default
`heff_params.yaml`

Original parameter (template) file. Only used for generating the final output parameter files.

precalc_param_file

type
String (file path)

default
`heff_params_precalc.yaml`

pre-calculated parameter file. Only used for generating the final output parameter files.

best_predict_model_label

type
String

default
`best_predict`

The "label" for output files of the "best predict" model. Appear in the file name of the corresponding output files.

best_safe_model_label

type
String

default
`best_safe`

The "label" for output files of the "best safe" model. Appear in the file name of the corresponding output files.

search_last_model_label

type
String

default
`search_last`

Available as of Oct. 11, 2025.

The "label" for output files of the "search last" model (the last λ value used in the MLCons searching). Appear in the file name of the corresponding output files.

search_last_model_label

type
String

default
`max_model`

Available as of Nov. 4, 2025.

The "label" for output files of the "max" model including all possible parameters (except the excluded ones). Appear in the file name of the corresponding output files.

sbs_iter_best_file

type
String (file path)

default
`sbs_summary_iter_best.csv`

The output file name for the sequential back search (SBS) report file of the model found in λ searching with minimal test variance. Provides the parameter removing order.

sbs_iter_first_min_file

type
String (file path)

default
`sbs_summary_iter_first_min.csv`

The output file name for the sequential back search (SBS) report file of the model with the first local minimal test variance in λ searching. Provides the parameter removing order.

param_selection_file

type
String (file path)

default
`param_selection.yaml`

The output file name of parameter selection file. This is one of the main output files of the MLCons process.

9.3.15 boundness table

This table is available as of version 0.22.

This table configures some details in boundedness determination, and some details for initializing the open bounding and closed bounding procedures (see *Energy boundedness*).

The following keys control the input files. These files are loaded as in other procedures, using the filenames provided by these fields, with the same default values (if not specially noted). For simplicity, the detailed introductions are omitted. We just list these keys and the related keys in other tables (for other procedures). Note that the latter key (i.e., key from other tables) are *not* used in the bounding procedures.

- `original_param_file`: see `energy.files.param_yaml`.
- `precalc_param_file`: see `fit_general.params_precalc_yaml`.
- `interaction_json`: see `energy.files.interactions_json`.
- `unitcell_yaml`: see `cell.unitcell_yaml`.
- `interactions_topology_yaml`: see `mlcons.interact_topo_yaml`.

The other keys are introduced below.

min_unbound_order

type
Integer

default
0

Determines the minimal order of SATs that may be considered as unbounded. For example, if set to 3, the SATs with order 2 will never be considered as unbounded during the bounding procedures; instead, the SATs with order 3 may be considered as unbounded.

See *Closed bounding procedure* for more details.

perturbated_as_unbound

type
Boolean

default
false

If set to true, all the SATs with perturbation SATs are considered as unbounded SAT (regardless of the parameter values). This setting may be useful for alloy models, if the constrained fitting is not triggered for a long time. See *Closed bounding procedure* for more details.

9.3.16 bounding_closed table

This table is available as of version 0.22.

The `bounding_closed` table controls the details of the closed bounding procedure, as well as the embed closed bounding procedure in the open bounding procedure. See *Closed bounding procedure* for more details.

init_param_selection_file

type
String (file path)

default
`param_selection_init.yaml`

Determines the input file name for the initial parameter selection file. This is typically obtained from the MLCons process. This input file is mandatory. A non-empty string must be provided for this key.

data_file

type
String (file path)

default
`linear_data.json`

Same as `mlcons.data_file`.

use_validation_set

type
Boolean

default
true

Determine whether the validation set is used for the bounding procedure. This validation set is similar to that used in the MLCons process. Note that the bounding procedure does *not* use a test set.

> **Tip**
>
> We recommend to set this to true to prevent over fitting.

test_set_prop

type
Float

default
0.3

The proportion of the test set data, out of all available data, if `use_validation_set` is set to true.

max_control_steps

type
Integer

default
1000

The maximal allowed steps in the closed bounding procedure.

result_tag

type
String

default
`bounded`

File name tag for the output parameter file and parameter selection file with bounded model.

result_direct_fit_tag

type
String

default
`direct_fit`

File name tag for the output parameter file with the directly fitted parameters (i.e., no constrained fitting applied). The model has the same interaction set as that used for the `result_tag`.

pow_lambda_adding

type
Float

default
1.0

The λ factor for the regularized variance used in adding controlling SATs. See MLCons for more information.

faster_adjustment

type
Boolean

default
true

If set to true, an SAT selection process that may lead to a faster converged constrained-fitting is applied. See *Closed bounding procedure* for more details.

first_select_adjustment_variance_scale_range_to_largest

type
Float

default
0.5

This is used if `faster_adjustment = true`. For the case where all unbounded SATs have the `NegativeTop` type, the unbounded SAT leading to the maximal variance *increasing* by adjusting it value is selected. At the same time, the SATs leading to the increasing variance (using the same operation) with the scale *no less than* the set value of this key times the maximal scale is also considered.

constrain_lowest_order

type
Boolean

default
true

If set to true, only the controlling SATs with the lowest possible order is considered in each step.

reject_all_linearly_dependent_model

type
Boolean

default
true

If set to true, all parameter adding trials that lead to linearly dependent model will be rejected.

allow_constrained_fitting

type
Boolean

default
true

Whether constrained-fitted is allowed. Strongly recommended to set to true.

constrained_fitting_impl

type
String (Literals)

options
`scipy` `fixed_linear`

default
`scipy`

Select constrained-fitting implementation. Options are:

- `scipy`. Use the constrained optimization algorithm in `scipy` package (`scipy.optimize.minimize`).
- `fixed_linear`. Use simple linear least square fitting, with parameters for uncontrolled SATs fixed to some positive values. This can be faster, but lead to larger variance.

constrained_fitting_scipy_method

type
String (Literals)

default
SLSQP

This is valid only if `constrained_fitting_impl = "scipy"`. The constrained fitting method to be employed. See documents of scipy for more information.

constrained_fitting_scipy_options

type
Table

default
empty

This is valid only if `constrained_fitting_impl = "scipy"`. Any extra arguments passed to the `scipy.optimize.minimize` function.

constrained_fitting_scipy_from_fixed

type
Boolean

default
true

This is valid only if `constrained_fitting_impl = "scipy"`. If set to true, the `fixed_linear` fitting is done first, and the resulting parameter serves as the initial parameters, for the constrained fitting of `scipy`. This may lead to a faster convergence.

adding_max_variance_scale

type
Float

default
1.3

The maximal allowed variance scale (in both training set and validation set) for adding a new SAT.

incremental_adding

type
Boolean

default
true

Use incremental algorithm to predict the new parameters and variance while adding new SATs. Recommended to set to true because this should have better performance.

Set this to true will implicitly indicate `reject_all_linearly_dependent_model = true` (but may with a different numeric threshold for determining the linear dependency due to implementation).

9.3.17 bounding_open table

The `bounding_table` controls the open bounding procedure. Note that the embed closed bounding procedure is controlled by the `bounding_closed` table.

max_iters

type
Integer

default
1000

Maximal number of allowed open bounding iterations.

max_unused_param_proportion

type
Float

default
0.2

Maximal allowed proportion of unused parameters, out of the total number of parameters in the interaction set. This is evaluated and checked before each controlling SAT addition process. If such proportion exceeds the configured value, interaction set compressing is performed, to remove unused SATs. This can greatly reduce the computational effort required in the merge-otf process.

model_dir_prefix

type
String (directory path prefix)

default
`model_iter_`

The prefix of the directory name storing the generated (or updated) model during the open bounding procedure. The actual directory name will be such prefix followed by the open bounding iteration index.

Note

Not all open bounding iteration generate corresponding model directory. This directory is generated only if the model is updated during this iteration (SAT compressing and/or controlling SAT adding is performed).

final_model_dir

type
String (directory path)

default
`model_final`

The directory name holding the final model in the open bounding procedure.

compressed_model_dir

type
String (directory path)

default
`model_compressed`

The directory name holding the final *compressed* model in the open bounding procedure. This model is equivalent to that stored in `model_final`, but with an extra model compressing operation to reduce the number of SATs (i.e., remove unused SATs), making it more efficient. This is the recommended result of the open bounding procedure.

max_closed_steps

type
Integer

default
10

Maximal allowed number of closed bounding steps in one open bounding iteration. At the beginning of each open bounding iteration, the minimal possible order of bounding SATs is checked. If it is possible to find bounding SATs with lower order than current available ones, or if there is no available controlling SATs, the interaction set will be updated to include the desired controlling SATs. This will trigger the merge-otf fitting process, which is relatively time-consuming.

This field controls the maximal allowed steps for one open bounding iteration. The iteration is forced to finish after the configured steps, and another check of minimal possible order is performed (note that the minimal possible controlling order may be changed during the embed closed bounding procedure because of the re-fitting after adding new SATs).

iter_by_order

type
Boolean

default
true

If set to true (recommended), the open bounding iterations are performed according to the minimal possible controlling order.

max_controlling_order

type
Integer

default
10

Maximal order of the controlling SATs. If the required minimal controlling SAT order exceeds this value, the bounding procedure will be terminated.

merge_otf_log_file

type
String (file path)

default
`merge_otf_log.txt`

The filename for the redirected output of merge-otf fitting.

Available as of Jan. 20, 2026. In this version, all the logs of the merge-otf fitting executed during the open bounding procedures are redirected to the given file, in order to clarify the stdout information. Before this version, all the logs are directly sent to stdout.

CHAPTER

TEN

OUTPUT REFERENCE

10.1 DoF output files

The degrees of freedom (DoFs) output files contain the full DoF (local mode and strain) information at a certain instant of the simulation or optimization. These files are provided in HDF5 format, and could be read by some commonly used packages, like `h5py` package of Python. They can also be loaded by some programs provided by this package for some processing or continuing simulation.

There are two different structures of DoF files provided by this package: the single-frame structure and the multi-frame structure, and they are used in different scenes. Here, a "frame" means a full set of DoFs at a instant of the simulation. A single-frame structure file contains only one frame, and could be used as the input (initial) structure of simulation and optimization. A multi-frame structure file contains (possibly) multiple frames. They can be loaded using the `heff::cell::DofH5Reader` class provided in this package as an API, or its Python binding `heffpy.cell.DofH5Reader`. Besides, we also provide a utility tool to pick one frame from the multi-frame file and write into single-frame structure file and/or text files, see `heff_dof_split` in *Utility programs*.

10.1.1 Single-frame DoF file

The single-frame DoF files are written at the end of each simulation run, and optimization, etc. The writing process is implemented by the function `heff::cell::HeffCell::dump_dof_h5` in source code. Briefly, the resulting HDF5 file contains three main parts: (1) the `meta` dataset containing the metadata related to the model; (2) the `loc` dataset containing the local mode variables; and (3) the `eta` dataset containing the strain variables.

The single-frame DoF file could be used as the input DoF file in the simulations, see `init.load_dof_h5` in *Input reference*.

10.1.2 Multi-frame DoF file

The multi-frame DoF file are written by several different scenes, including the DoF file at the end of each simulation run (`dof-runs.h5`), or at instants with specified distances (`dof-tape.h5`), or at each step of OTF MD simulation (`dof-steps.h5`), etc. The writing process is implemented by the class `heff::cell::DofH5Writer`.

In each multi-frame DoF file, the data are organized by two levels: (1) simulation runs, and (2) frames. Each file contains one or more simulation runs, and each with one or more frames. For the simulation tasks (`heff_run_sim`), each simulation run in this file corresponds to one original simulation run (with certain `sim_id` in the thermodynamics output file) and contains all the frames generated by such simulation run. For `dof-runs.h5` file, only the final frame of each simulation run is written, thus each simulation run in this file contain only one frame. For `dof-tape.h5`, each simulation run may contain multiple frames. For `dof-steps.h5` in OTF MD simulations, there is only one simulation run by construction.

In each multi-frame DoF file, the metadata about the model is provided in the `meta` dataset. The number of simulation runs is provided by the `simulation_count` attribute of the file. The data of each simulation run is given in one HDF5 group called `run_<sim_id>` where `<sim_id>` is the simulation index (1-based). Such group contains two datasets: the `sites` dataset containing the local mode variables, and `strains` dataset containing the strain variables. Both datasets

are two-dimensional, with each row containing the data of one frame. For local mode variables, the data is expanded in a flat form, with data of each unit cell in adjacent positions.

The multi-frame DoF files cannot be used directly as the initial structure of simulations. But it is possible to pick frames from them to form single-frame DoF files to do so (see `heff_dof_split` in *Utility programs*). Besides, it is also possible to load it using the API `heff::cell::DofH5Reader` or its Python binding `heffpy.cell.DofH5Reader` (see source codes and comments there in for usages). One can also process the HDF5 file directly using other tools, like `h5py` package in Python, or directly use the HDF5 library, etc.

10.2 Site text file

The site text file contains the information of the unit cell positions and mixed site occupations. The default output filename is `out_sites.txt` while running simulations with single process. Example of this file:

```
num     a       b       c       sigs
0       0       0       0       1       2       -1      -1
1       0       0       1       2       2       -1      -1
2       0       0       2       2       1       -1      -1
...
```

The site text file contains a header line, followed by several data lines. The number of data lines is equal to the number of unit cells in the supercell. In each data line, the first column is the index (zero-based) of the unit cell in the supercell, and the following three columns represent the position of the unit cell (origin point) in the supercell, in the unit of unit cell fractional coordinate. The values in these three columns are always integers by definition. The following columns are the occupation numbers of the sites in the unit cell. The occupation number is 0 if the site is a vacuum site, or -1 if the site is a occupied site which is not mixed, and other positive number if the site is a mixed site. In this example, each unit cell contains 4 sites, with the first two sites being mixed sites. In the first unit cell, these two mixed sites are occupied by element indexed 1 and 2, respectively.

10.3 Local mode and strain text files

Besides the DoF H5 files (*DoF output files*), the local mode and strain text files provide another version of DoF data, provided in plain text format, which may be easier to be read, parse, and more importantly, generated. The local mode and strain files contain two separate text files, usually output at the same time, providing the information of configuration in combination. In simulations, the final DoF configuration is written into the files named `final-modes.txt` (local mode text file) and `final-strain.txt` (strain text file), containing basically the same information as the DoF H5 file (`dof-last.h5`).

The local mode text file contains a header line followed by several data lines. The number of data lines is equal to the number of unit cells in the supercell, and in exactly the same order as that in the site text file (*Site text file*). In each data line, the first column is the unit cell index, and the remaining columns are the local mode variable values in this unit cell, in the order of LWFs and their axes, as specified in the unit cell input file. The corresponding LWF names and axis indexes are given in the header line, where, for example, `v_1` denotes the first axis (one-based index) of the LWF named `v`.

The strain text file simply contains six floating-point numbers in a single line, representing the strain values, in Voigt index.

CHAPTER

ELEVEN

FREQUENTLY ASKED QUESTIONS

11.1 Installation

11.1.1 Why is the installation so "complex"?

We emphasize our package uses *standard* workflow for building a package from source using `CMake`. This building and installation workflow is also applied to many famous and widely-used packages. If you have difficulties while installing or finding the dependent packages, please refer to the `CMake` document https://cmake.org/cmake/help/latest/index.html , the document of the dependent packages, or use search engine or AI tools.

11.1.2 Why do we need such dependencies?

As we introduced in *Installation*, we need several dependency packages to be installed before this package, or be built along with this package. These dependent packages provide some basic supports for this package. For more details:

- The `FFTW` package provides the support for fast Fourier transformation (FFT). The MKL package could also be an alternative for the FFT implementation.
- The `Eigen` package provides essential supports for the matrix representation and computation, and linear algebra algorithms, which is widely used in this package.
- The `spdlog` package provides the logging and formatting support.
- The `HDF5` package provides the basic support for reading and writing HDF5 files. The `HighFive` package provides modern C++ interfaces for `HDF5` package.
- The `toml11` package is required for parsing and writing the TOML files, which are used for configuration files in this package.
- The `yaml-cpp` package is required for reading and writing YAML files.
- The `nlohmann-json` package provides the support for reading and writing JSON files with modern C++ interfaces.
- The `Boost` packages are used for several different purposes:
  - `Boost.describe` provides the reflection support for reading and writing configuration files and Python binding.
  - `Boost.containers` provides some containers which is not provided by C++ standard library.
  - `Boost.math` provides some special functions (added in C++17) if they are not implemented by the used standard C++ library.
  - `Boost.spirit` is used for parsing the DSL for defining the physical quantities.
  - (Optional) `Boost.unit_test_framework` is required by the testing codes.

- (Optional) `Boost.program_options` is required for parsing the command-line arguments in some utility programs.

11.1.3 Why cannot I use GCC 4.8.5 for building this package?

As we emphasized before, the GCC 4.8.5, which is the default compiler for some widely-used Linux distributions, *cannot* be used for building this package. This is because our package requires at least C++17 standard, which is not supported by GCC 4.8.5. The oldest GCC compiler that tested for building this package is GCC 8. For more information, please refer to *Installation*.

11.2 usage

11.2.1 Why are there so many different file formats?

As you have noticed, there are several different file formats used in the input and/or output files of this package, including TOML, YAML, HDF5, etc. We choose these general formats so that these files could be parsed by not only our package, but also some commonly used packages, for example the `h5py` package for parsing HDF5 format in Python, the `yaml` package for parsing YAML format in Python, etc. In addition, different formats are chosen for different files according to the characteristics, design purposes and expected usage of these files:

- For simple input files that are expected to be edited by the user (mostly manually created), we prefer the TOML format, which is human-readable and simply editable. Note that our implementation contains some slight extension to TOML, so that some files accepted by our C++ program may not be accepted by other implementations. Meanwhile, the "echoed" file for these configuration files are also given in TOML format.
- For input or output files with (possibly) large amount of data and not expected to be read or edited by the user, we prefer the HDF5 format. Examples are the local DoFs dumped during the simulation process. The HDF5 files (`*.h5` or `*.hdf5`) are binary files and cannot be read or edited directly. The user typically needs not to read or edit these files. However, they could still be parsed, viewed or edited using some other programs or packages, for example the `h5dump` program or `h5py` package in Python.
- For input or output files that are typically not created by the user but may be read and slightly modified by the user, we prefer the YAML format. Examples are the parameter file (`heff_params.yaml`), and the unit cell file for simulation (`heff_unitcell.yaml`).
- For files containing possibly large amount of data and complex hierarchy, we prefer the `JSON` (Javascript object notation) format. These files typically need not to be read or edited directly by the user. However it is still easy to read or edit them, ether manually or using external packages, for example the `json` standard library in Python. Examples are the interaction data file `heff_interactions.json` and the LWF data files.

11.2.2 What is "undefined behavior"?

In this documentation sometimes says the behavior of the program is undefined within some input conditions. The term "undefined behavior" has basically the same meaning to that in the C++ standard (see also C++ standard, section `defns.undefined`). Briefly, if the behavior is undefined upon some input conditions, the program could do anything, including but not limited to: (1) crashes immediately or later, with error messages related or not related to the input files, or without any error message at all; (2) continues but with wrong results, or results that are meaningless physically; (3) causes undefined behavior defined by C++ language; (4) shows different results in different runs, etc.

A correct set of input should never trigger undefined behaviors. Any input settings causing undefined behaviors are considered to be incorrect input. And it is the users' responsibility to avoid the conditions causing undefined behaviors.

Undefined behaviors will be triggered if any requirements specified explicitly in this document or the input hints of the program are violated, or if physical unreasonable inputs are made for the program.

CHAPTER

TWELVE

CHANGE LOG SUMMARIES

Here we note some important changes in reversed time order. Note that not all changes are introduced here.

12.1 Jun. 13, 2026

Heff internal version: 0.30.0; revision: 20260604.0

New features:

1. Add multi-simulation-OTF iteration (`multi_sim_otf_iter`) process, which detect the boundedness with respect to given simulation condition and supercell matrix by doing a test MD simulation and do continuing OTF fitting if such simulation is considered to be unbounded, and repeat such iteration for a set of pre-given supercell matrices. See the section entitled "Multi-simulation-OTF iteration" in the document.
2. Add the approximately periodic sub-cell (APS-cell) program, see the document for the utility program `heff_approx_periodic_sub_cell`, and the section entitled "Approximately periodic sub-cell".

Other updates:

1. Add the `merge_otf_count_fps` utility program for counting the number of FP calculations in OTF fittings.
2. Add the automatic data-split output in MLCons and bounding procedure and the support for reading from file.
3. Add the output file `mpi_error<id>.yaml` for MPI parallel simulations if errors encountered.
4. Add the support for write (read) momenta to (from) files.
5. Other details and document updating.

12.2 May 13, 2026

Heff internal version: 0.28.2; revision: 20260513.0

Important:

This version contains important fixes for LWF generation part. The fixed problems may cause the generated LWF fail to pass the transformation check or to have poor centralization. Recommend everyone who will generate LWFs to update to this version immediately.

Other changes:

1. Add the support of so-called approximated periodic sub-cell (APS-cell) generation (experimental).
2. Add physical quantities `DipoleMomentAmp` and `PolarInsLocAmp`.
3. Other minor changes.

CHAPTER

THIRTEEN

DEVELOPMENT NOTES

This document mainly provides some notes for developers. Users who just want to *use* this package could ignore this document.

13.1 Checklist for adding new physical quantities

To add a physical quantity into the statistic system, complete the following checklist:

1. Add entry in `enum class VarType` (`var_type.h`).
2. Add `case` in `get_predefined_type_dim(VarType type)` and `type_require_potential(VarType type)` functions if applicable (`var_type.h`).
3. Add implementation of header formatter in `phys_var.cpp`; add dispatching statements in `extend_field_names` and `extend_header`.
4. Add parser implementation for the new type (`var_parse.cpp`).
5. Add overload of `PhysStatus::compute_status_impl<>` (`phys_status.h`); add dispatching statements in `compute_status`.
6. Add conversion in `PhysVarManager::convert_to_output` if conversion is required (`phys_var_manager.cpp`).
7. For final statistical quantities, add the dependent variable implementation in `add_final_stat_var` (`dependent_var_manager.cpp`).
8. For final statistical quantities, add calculation implementation function and dispatching statement in `thermo_calculator.cpp`.
9. If there are special requirements on the conditions, update `check_conditions` in `thermo_statistic_meta.cpp`.

13.2 Checklist for adding new condition variables

To add a new condition variable, complete the following checklist:

1. (In `cond_type.h`) Add entry in `enum class CondType`.
2. Add `case` in `get_cond_type_dim`.
3. Add `case` in `cond_type_to_string`.
4. (In `condition_values.cpp`) Add dispatch statement in `write_to_h5_dataset`.
5. Add dispatch in `emit_yaml`.
6. Add dispatch in `from_yaml`.

7. (In `sim_conditions.h`) Add specialization of `struct cond_data_accessor<>`.
8. Add dispatch in `format_output`.
9. Add dispatch in `to_condition_values`.
10. (In `cond_manager.cpp`) Add dispatch in `field_names()` and `header()`.
11. (In `status_h5_reader.cpp`) Add dispatch in `load_conditions`.

CHAPTER

FOURTEEN

REFERENCES

For the HTML version, the references are listed in this page, distributed in their categories. For PDF version, please refer to the BIBLIOGRAPHY section.

14.1 Effective Hamiltonian related

14.2 Symmetry, LWF building, and related

14.3 Simulations and related

14.4 Fitting algorithms and related

14.5 Optimization algorithms and related

14.6 Tools

14.7 Other resources

BIBLIOGRAPHY

[PTMC1996] KoJi Hokushima et al, Exchange Monte Carlo Method and Application to Spin Glass Simulations, Journal of the Physical Society of Japan, **65**, 1604 (1996). DOI: 10.1143/JPSJ.65.1604

[PTMC2005] David J. Earl et al, Parallel tempering: Theory, applications, and new perspectives, Physical Chemistry Chemical Physics **7**, 3910 (2005). DOI: 10.1039/b509983h

[ALFE-H-2025] Xingyue Ma et al, Active learning of effective Hamiltonian for super-large-scale atomic structures, npj Computational Materials **11**, 70 (2025). DOI: 10.1038/s41524-025-01563-z

[Zhong1995] W. Zhong et al, First-principles theory of ferroelectric phase transitions for perovskites: The case of $BaTiO_3$, Physical Review B **52**, 6301 (1995). DOI: 10.1103/PhysRevB.52.6301

[VCA2000] L. Bellaiche et al, Virtual crystal approximation revisited: Application to dielectric and piezoelectric properties of perovskites, Physical Review B **61**, 7877 (2000). DOI: 10.1103/PhysRevB.61.7877

[EwaldSum2019] D. Wang et al, Ewald summation for ferroelectric perovskites with charges and dipoles, Computational Materials Science **162**, 314 (2019). DOI: 10.1016/j.commatsci.2019.03.006

[LowDimHeff] I. Ponomareva et al, Atomic treatment of depolarization energy and field in ferroelectric nanostructures. Physical Review B **72**, 140102(R) (2005). DOI: 10.1103/PhysRevB.72.140102

[AnistroDip] Vladimir Ivchenko, Coulomb's and Ohm's laws in an anisotropic medium, European Journal of Physics **42**, 065201 (2021). DOI: 10.1088/1361-6404/ac1abc

[LaserPulse2022] Peng Chen et al, Deterministic control of ferroelectric polarization by ultrafast laser pulses, Nature Communications **13**, 2256 (2022). DOI: 10.1038/s41467-022-30324-5

[HeffMD2008] I. Ponomareva et al, Terahertz dielectric response of cubic BaTiO3, Physical Review B **77**, 012102. DOI: 10.1103/PhysRevB.77.012102

[PZT_Heff_2000] L. Bellaiche et al, Finite-Temperature Properties of Pb(Zr,Ti)O3 Alloys from First Principles, Physical Review Letters 84, 5427 (2000). DOI: 10.1103/PhysRevLett.84.5427

[PST_Heff_2022] Xingyue Ma et al, Large electrocaloric response via percolation of polar nanoregions, Physical Review B 105, 054104 (2022). DOI: 10.1103/PhysRevB.105.054104

[LWF1995] K. M. Rabe et al, Localized basis for effective lattice Hamiltonians: Lattice Wannier functions, Physical Review B **52**, 13236 (1995). DOI: 10.1103/PhysRevB.52.13236

[LWF2000] Jorge Íñiguez et al, Optimized local modes for lattice-dynamical applications, Physical Review B **61**, 3127 (2000). DOI: 10.1103/PhysRevB.61.3127

[LWF_opt_2005] Eric Cockayne et al, Generation of lattice Wannier functions via maximum localization, Physical Review B **71**, 094302 (2005). DOI: 10.1103/PhysRevB.71.094302

[phonon_band_rep_1995] M. B. Walker et al, Site Symmetry and Band Representation for Phonons, Physical Review Letters **74**, 3824 (1995). DOI: 10.1103/PhysRevLett.74.3824

[band_rep_1988] H. Bacry et al, Symmetry and analyticity of energy bands in solids, Physical Review Letters **61**, 1005 (1988). DOI: 10.110//PhysRevLett.61.1005

[band_rep_2018] Jennifer Cano et al, Building blocks of topological quantum chemistry: Elementary band representations, Physical Review B **97**, 035139 (2018). DOI: 10.1103/PhysRevB.97.035139

[MLWF-2001] Ivo Souza et al, Maximally localized Wannier functions for entangled energy bands, Physical Review B **65**, 035109 (2001). DOI: 10.1103/PhysRevB.65.035109

[SA-MLWF-2013] R. Sakuma et al, Symmetry-adapted Wannier functions in the maximal localization procedure, Physical Review B **87**, 235109 (2013). DOI: 10.1103/PhysRevB.87.235109

[PhononSupercell] Bartomeu Monserrat et al, Lattice dynamics and electron-phonon coupling calculations using nondiagonal supercells, Physical Review B **92**, 184301 (2015). DOI: 10.1103/PhysRevB.92.184301

[Omelyan2002] I. P. Omelyan et al, Optimized Verlet-like algorithms for molecular dynamics simulations, Physical Review E **65**, 056706 (2002). DOI: 10.1103/PhysRevE.65.056706

[HMC2018] Sergei Prokhorenko et al, Large scale hybrid Monte Carlo simulations for structure and property prediction, npj Computational Materials **4**, 80 (2018). DOI: 10.1038/s41524-018-0137-0

[Dielectric2017] 26. G. Fthenakis et al, Dynamics of antiferroelectric phase transition in PbZrO3, Physical Review B **96**, 184110 (2017). DOI: 10.1103/PhysRevB.96.184110

[DispField2009] Massimiliano Stengel et al, Electric displacement as the fundamental variable in electronic-structure calculations, Nature Physics **5**, 304 (2009). DOI: 10.1038/nphys1185

[SecondPrinciples2017] Jacek C. Wojdeł et al, Efficient systematic scheme to construct second-principles lattice dynamical models, Physical Review B **95**, 094115 (2017). DOI: 10.1103/PhysRevB.95.094115

[VaspMLFF2019] Ryosuke Jinnouchi et al, On-the-fly machine learning force field generation: Application to melting points, Physical Review B **100**, 014105 (2019). DOI: 10.1103/PhysRevB.100.014105

[MLMCH2020] Xue-Yang Li et al, Constructing realistic effective spin Hamiltonians with machine learning approaches, New Journal of Physics 22, 053036 (2020). DOI: 10.1088/1367-2630/ab85df

[NEB2000] Graeme Henkelman et al, Improved tangent estimate in the nudged elastic band method for finding minimum energy paths and saddle points, Journal of Chemical Physics 113, 9978 (2000). DOI: 10.1063/1.1323224

[CI-NEB-2000] Graeme Henkelman et al, A climbing image nudged elastic band method for finding saddle points and minimum energy paths, Journal of Chemical Physics 113, 9901 (2000). DOI: 10.1063/1.1329672

[G-SSNEB-2012] Daniel Sheppard et al, A generalized solid-state nudged elastic band method, The Journal of Chemical Physics 136, 074103 (2012). DOI: 10.1063/1.3684549

[NEB-real-spring-2016] Esben L. Kolsbjerg et al, An automated nudged elastic band method, The Journal of Chemical Physics 145, 094107 (2016). DOI: 10.1063/1.4961868

[Adam2017] Diederik P. Kingma et al, Adam: A Method for Stochastic Optimization, arXiv:1412.6980 (2017)

[FIRE2006] Erik Bitzek et al, Structural Relaxation Made Simple, Physical Review Letters 97, 170201 (2006). DOI: 10.1103/PhysRevLett.97.170201

[FIRE2_2020] Julien Guénolé et al, Assessment and optimization of the fast inertial relaxation engine (fire) for energy minimization in atomistic simulations and its implementation in lammps, Computational Materials Science 175, 109584 (2020). DOI: 10.1016/j.commatsci.2020.109584

[NEB_opt_2008] Daniel Sheppard et al, Optimization methods for finding minimum energy paths, The Journal of Chemical Physics 128, 134106 (2008). DOI: 10.1063/1.2841941

[Examples] The examples will be distributed on …

[HeffCXXTestData] The test data used for `ctest` in `Heff_general` C++ package building is located in the `tests/testdata` directory in the `Heff_general` package source root. Some of test data could serve as examples for using this package. Note that they are only provided for *technical* usages, and no *physical* meaning is guaranteed!