SKPAR Documentation

Release 0.2.0

Stanislav Markov

Mar 29, 2023

Contents

1 News **3**

2 Contents **5**

2.1 About . . . 5
2.2 Install . . . 7
2.3 Commands . . . 8
2.4 Tutorials . . . 9
2.5 Input File Reference . . . 16
2.6 Subpackage/module Reference . . . 32
2.7 License . . . 52
2.8 Development . . . 52
2.9 Contributors . . . 53

3 Indices and tables **55**

Bibliography **57**

Python Module Index **59**

Index **61**

SKPAR is a parameter-optimisation framework for the Density Functional Tight-Binding (DFTB) theory.

CHAPTER 1

News

- SKPAR version 0.2.0 released: September 2017
  - New Configuration section in input file allows for individual directory for each model evaluation, i.e. each parameter set. Be sure to check the relevant part in *Input File Reference*.
  - Strict bounds for PSO particle space, applied per dimension
  - Minor bug fixes
- SKPAR version 0.1.0 released: February 2017.

CHAPTER 2

Contents

2.1 About

SKPAR is a software tool intended to automate the optimisation of parameters for the Density Functional Tight Binding (DFTB) theory. It allows a flexible and simultaneous use of diverse reference data, e.g. from DFT calculations or experimentally obtained physical quantities.

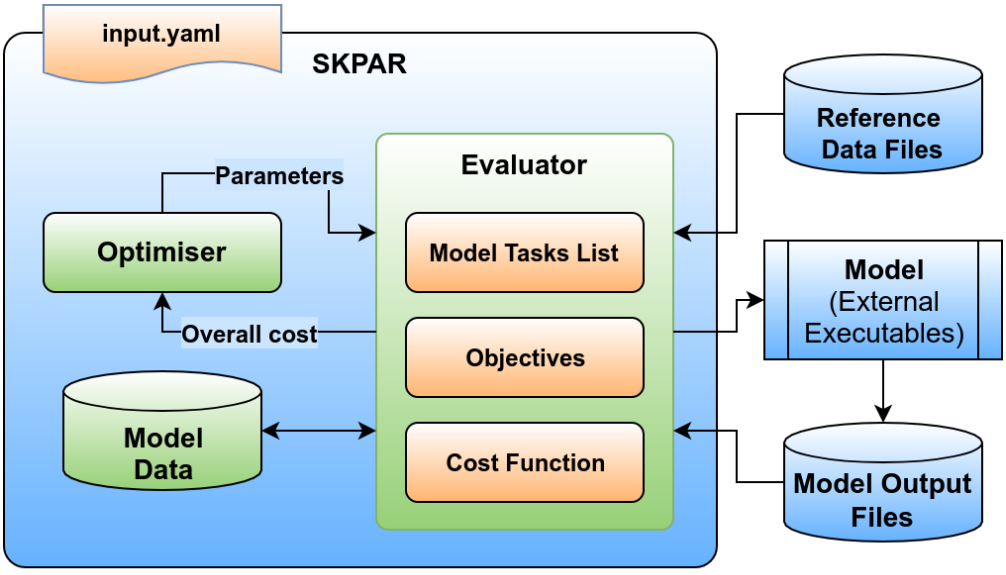

Fig. 1: **Fig. 1. Conceptual block diagram of SKPAR.**

2.1.1 Conceptual Overview

The conceptual diagram of SKPAR is shown in Fig. 1, where the relation between the following entities is suggested:

- **Model** – a collection of executables outside SKPAR that produce some data; In the context of DFTB parameterisation the model may encompass Slate-Koster table generation (driven by some parameters per chemical element), and a number of DFTB calculations that yield total energy and band-structure for one or more atomic structures. *SKPAR features a dynamic model setup via the declaration of a 'Model Task-List' in the SKPAR input file; There is no hard-coded application-specific model.*
- **Objectives** – a set of single valued functions that depend on the model parameters; Typical example is a root-mean-squared deviation between some reference data (e.g. band-structure calculated by DFT) and the model data (e.g. the band-structure calculated by DFTB). *SKPAR provides a generic facility for declaring objective function by specifying a list of Objectives in the input file; the specification includes instruction on accessing reference data and determines a query into the model and reference databases.*
- **Reference data** – a set of data items that we want the model to be able to reproduce within certain error tolerance; Reference data may come from DFT calculations or be experimentally obtained. *SKPAR admits explicit reference data in the input file, or instructions on how to obtain reference data by accessing and interpreting external files; support for database query is under development too.*
- **Cost function** – a scalar function of the individual objectives mentioned above that yields a single number representative of the quality of a given set of parameter values. *Currently SKPAR supports only weighted root mean squared deviation of the objectives from zero.*
- **Optimiser** – an algorithm for efficient exploration of the parameter space with the aim of minimising the cost function. *SKPAR features particle-swarm-optimisation (PSO) algorithm.*

The sole purpose of the *Optimiser* in Fig. 1 is to generate parameters in a way that does not depend on the specifics of the model being optimised. The *Evaluator* in Fig. 1 acts as an interface between the embodiment of the *Model* by one or more external executables, and the *Optimiser*.

The declaration of objectives and model tasks, as well as the overall functionality of SKPAR is controlled by an input file (in YAML format), where the user must define as a minimum:

1. A list of tasks that must be executed in order to obtain model data.
2. A list of objectives that must be evaluated in order to assess overall cost.
3. The optimisation strategy – algorithm, parameters, etc.
4. Aliases to complex commands involving external executables

The optimisation loop realised by SKPAR is shown in Fig. 2.

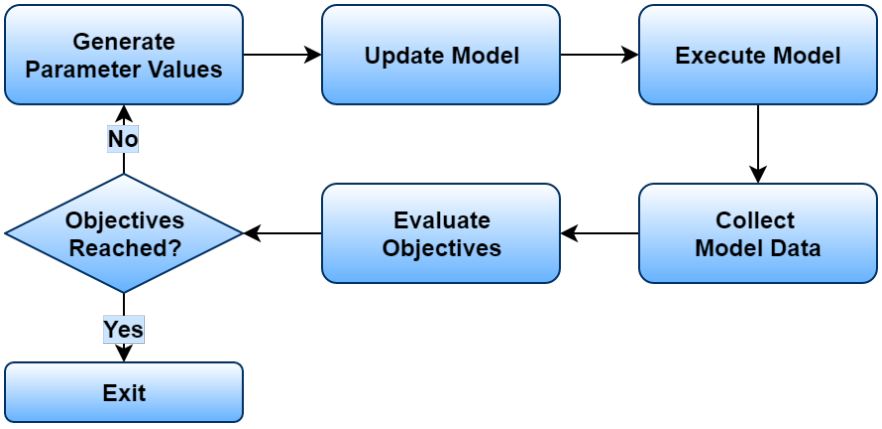

Fig. 2: **Fig. 2. Optimisation loop realised by SKPAR.**

2.1.2 Implementation Overview

SKPAR is implemented in Python and currently uses a Particle Swarm Optimisation (PSO) engine based on the DEAP library for evolutionary algorithms. Its control is done via an input file written in YAML.

Currently SKPAR provides two sub-packages: `core` and `dftbutils`.

The `core` package is of general nature, and its coupling to `dftbutils` is only via a tasks dictionary, through which SKPAR learns how to acquire data related to a DFTB model.

The `dftbutils` package concerns with all that is necessary to obtain data from a DFTB calculation. Presently, this package is limited in its support to the executables provided by BCCMS at the University of Bremen, Germany. This assumes:

- SKGEN is used for Slater-Koster File (.skf) generation (by `slateratom`, `twocnt`, and SKGEN),
- DFTB+ is used as the DFTB calculator, and
- dp_bands is used as post-processor of eigenvalue data to produce band-structure data.

However, an easy extension to alternative tool-flow is possible, and current development aims to completely decouple model execution from the core of SKPAR.

See also:

Subpackages and modules

Development

2.1.3 Extensions

The design of SKPAR features weak coupling between the core engine that deals with a general multi-objective optimisation problem, and the specifics of model execution that yields model data for a given set of parameter values. Therefore, its extension beyond DFTB parameterisation – e.g. to the closely related problems of parameter optimisation for empirical tight-bining (ETB) Hamiltonians or classical interatomic potentials for molecular dynamics, should be straightforward.

2.2 Install

The latest release of SKPAR can be found on GitHub.

User (w/o sudo or root privilege):

```
pip3 install --upgrade --user skpar
```

Please omit the *–user* option above if installing within a virtual environment.

Developer:

Clone the repository and go to the newly created directory of the repository.

Issue the following command from the root directory of the repository.

```
pip3 install --upgrade --user -e .
```

Please omit the *–user* option above if installing within a virtual environment.

To uninstall:

```
pip3 uninstall skpar
```

2.2.1 Dependencies

SKPAR's operation requires:

- YAML support, for setting up the optimisation,
- the DEAP library, for the Particle Swarm Optimisation engine,
- NumPy for data structures, and,
- Matplotlib for plotting.

2.2.2 Test

If cloning the repository, once installation of SKPAR and its dependencies is complete, it is important to ensure that the test suite runs without failures, so:

```
cd skpar_folder/test
python3 -m unittest
```

Tests runtime is under 30 sec and should result in no errors or failures.

2.3 Commands

2.3.1 `skpar`

The `skpar` command is the primary tool for setting up and running optimisation. The typical usage is:

```
skpar skpar_in.yaml
```

The few supported options could be obtained by:

```
skpar -h

usage: skpar [-h] [-v] [-n] [-e] skpar_input

Tool for optimising Slater-Koster tables for DFTB.

positional arguments:
skpar_input           YAML input file: objectives, tasks, executables,
                      optimisation options.

optional arguments:
-h, --help            show this help message and exit
-v, --verbose         Verbose console output (include full log as in
                      ./skpar.debug.log)
-n, --dry_run         Do not run; Only report the setup (tasklist,
                      objectives, optimisation).
-e, --evaluate_only   Do not optimise, but execute the task list and evaluate
                      fitness.
```

2.3.2 dftbutils

The `dftbutils` command can be seen as a wrapper around several related DFTB calculations, example being a band-structure calculation. It works via subcommands, as follows:

```
dftbutils -h

usage: dftbutils [-h] [-v] [-n] {bands} ...

Wrapper of DFTB+ for chaining several calculation in a single command

optional arguments:
-h, --help     show this help message and exit
-v, --verbose  Verbose console output
-n, --dry_run  Do not run; Only report the setup, i.e. tasklist.

Available sub-commands::
{bands}
    bands        Calculate bandstructure
```

dftbutils bands

This commands makes the calculations of a band-structure into a single execution step. It assumes that the relevant calculation on a *k*-grid, for the average density, and the following calculation along *k*-lines are setup in the `scc` and `bs` directories respectively. Currently it supports dftb+ as DFTB executable, and dp_bands from dptools as the executable that yields a band-structure array. So what it does in the end is:

```
cd workdir/scc && dftb+ & cd ../
/bin/cp scc/charges.bin bs
cd bs && dftb+
dp_bands band.out bands & cd ../../
```

Other options may be added in the future, to eliminate the implicit reliance on *dftb+* and *dp_bands*.

dftbutils set

This command should allow one to setup the relevant calculations for `dftbutils bands`. Currently not supported.

2.4 Tutorials

2.4.1 Tutorial 1 – Polynomial Fitting

This example covers the basic structure and content of the input file. The input file, e.g. `skpar_optimise.yaml`, would typically reside in the invocation directory. The models should have separate execution directories, typically within the invocation folder.

The relevant files for the example are under `skpar/test` directory:

- `test_optimise.yaml`, and the folder
- `test_optimise/`, where the model
- `test_optimise/model_poly3.py` is executed (and located)

The example can be run in the `skpar/test` directory by invoking:

- `skpar test_optimise.yaml`,

assuming that skpar is installed.

Input YAML file

In this example we try to fit a 3-rd order polynomial to a few points extracted from such a polynomial.

The setup of SKPAR consists of 4 items:

1. A list of objectives that steer the optimisation,
2. A list of tasks necessary to evaluate the model,
3. An optional dictionary of aliases (used in the task list) resolving to external executables,
4. A configuration of the optimisation engine (parameters, algorithm, cost-function).

The corresponding yaml file, test_optimise.yaml reads:

The reference polynomial, the reference points from it (see `ref: [...]` in the yaml file above, and the fitted 3-rd order polynomial may look as so:

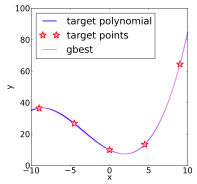

Fig. 3: **Comparison of reference and fitted (gbest) polynomials, and reference data points**

What is happening?

Objectives:

A model named `poly3` should yield data named `yval`, to be compared against explicitly provided reference data `ref: [...]`. Fitness evaluation of this specific objective should be based on root-mean-squared relative deviations, as stated after `eval:`.

Tasks (task-list):

At each iteration do:

1. **Set the environment by writing the parameters to `current.par` and** substitute values in `template.parameters.py` to `parameters.py`, both files in `./test_optimise` folder. (Note that `parameters.py` is not used by the model in this case.)
2. **Run the command `mypy` in the `./test_optimise` folder with** input file `model_poly3.py`.
3. **Get the model data from `test_optimise/model_poly3_out.dat` and** associate it with the `yval` of model `poly3` in the model database.

Optimisation

Generate four parameters (with initial range as given by a pair of min/max values) according to particle swarm optimisation algorithm, using 4-particle swarm, evolving it for 5 generations.

Executables

Whenever a run-task requires `mypy` command, use `python` instead.

2.4.2 Tutorial 2 – Optimisation of electronic parameters in DFTB

Fitting to experimental data

A more elaborate example is fitting the electronic structure of bulk Si to match a set of experimentally known *E-k* points and effective masses.

Here we set three different objectives, each of them contributing several data items.

The corresponding `skpar_in.yaml` is below, with comment annotations:

```
executables:
    skgen: ./skf/skgen-opt.sh    # script yielding an skf set
    bands: dftbutils bands       # band-structure calculation
                                 # see documentation for dftbutils sub-package

tasks:
    # Three types of tasks exist:
    # - set: [parmeter_file, working_directory, optional_template_file(s)]
    # - run: [command, working_directory]
    # - get: [what, from_sourse(dir, file or dict), to_destination(dict), optional_
↪kwargs]
    #        `what` is essentially a function name (see Get-Tasks dictionary)
    # ------------------------------------------------------------------------------
    - set: [current.par, skf, skf/skdefs.template.py]    # update ./skf/skdefs.py
    - run: [skgen, skf]                                  # generate SKF-set
    - run: [bands, Si-diam]                              # run dftb+ and dp_bands in␣
↪Si-diam
    - get: [get_dftbp_bs, Si-diam/bs, Si.bs,             # get BS data and put in Si.
↪bs model DB
            {latticeinfo: {type: 'FCC', param: 5.431}}] # must know the lattice for␣
↪what follows
    - get: [get_dftbp_meff, Si.bs,                       # get electron effective␣
↪masses
            {carriers: 'e', directions: ['Gamma-X'],     # note: destination is␣
↪ommitted,
             Erange: 0.005, usebandindex: True}]         #       hence update the␣
↪sourse
    - get: [get_dftbp_meff, Si.bs,                       # get hole effective masses
            {carriers: 'h', directions: ['Gamma-X', 'Gamma-L', 'Gamma-K'],
             nb: 3, Erange: 0.005}]
    - get: [get_dftbp_Ek  , Si.bs,                       # get eigen-values at special␣
↪points
            {sympts: ['L', 'Gamma', 'X', 'K'],
             extract: {'cb': [0,1,2,3], 'vb': [0,1,2,3]},
             align: 'Evb'}]

objectives:

    - Egap:                           # item to be queried from model database
        doc: Band-gap of Si (diamond) # doc-string for report purposes (optional)
        models: Si.bs                 # model name must match destination of a get-
↪tasks
        ref: 1.12                     # explicit reference data in for this objective
        weight: 4.0                   # relative importance of this objective
```

(continues on next page)

(continued from previous page)

```
                                        # objective weight in the overall cost function
        eval: [rms, relerr]             # objective function: RMS of relative error

    - effective_masses:                 # items to be queried here will be defined by
        doc: Effective masses, Si       # explicit keys, since the reference data␣
↪consists
        models: Si.bs                   # of key-value pairs
        ref:
            file: ./ref/meff-Si.dat     # the reference data is loaded via numpy.
↪loadtxt()
            loader_args:
                dtype:                  # NOTABENE: yaml cannot read in tuples, so we␣
↪must
                                        #           use the dictionary formulation of␣
↪dtype
                    names: ['keys', 'values']
                    formats: ['S15', 'float']
        options:
            subweights:                 # individual data items have sub-weight within␣
↪an objective
                dflt   : 0.1            # changing the default (from 1.) to 0. allows␣
↪us to consider
                me_GX_0: 1.0            # only select entries; alternatively, set␣
↪select entries
                me_Xt_0: 0.0            # to zero effectively excludes them from␣
↪consideration
        weight: 1.0                     # objective weight in the overall cost function
        eval: [rms, abserr]             # objective function: RMS of absolute error

    - special_Ek:
        doc: Eigenvalues at k-points of high symmetry
        models: Si.bs
        ref:
            file: ./ref/Ek-Si.dat
            loader_args:
                dtype:                  # NOTABENE: yaml cannot read in tuples, so we␣
↪must
                                        #           use the dictionary formulation of␣
↪dtype
                    names: ['keys', 'values']
                    formats: ['S15', 'float']
        options:
            subweights:
                dflt   : 0.1            # changing the default (from 1.) to 0. allows␣
↪us to consider
                me_GX_0: 1.0            # only select entries; alternatively, set␣
↪select entries
                mh_Xt_0: 0.0            # to zero effectively excludes them from␣
↪consideration
        weight: 1.0
        eval: [rms, relerr]


optimisation:
    algo: PSO                           # algorithm: particle swarm optimisation
    options:
        npart: 2                        # number of particles
```

(continues on next page)

(continued from previous page)

```
        ngen : 2                        # number of generations
    parameters:
        - Si_Ed  :  0.1 0.3             # parameter names must match with placeholders␣
↪in
        - Si_r_sp:  3.5 7.0             # template files given to set-tasks above
        - Si_r_d :  3.5 8.0
```

Fitting to DFT and experimental data

A yet another elaborate example is fitting the electronic structure of bulk Si using a combination of DFT-calculated band-structure *and* a set of experimentally known *E-k* points and effective masses.

This is mostly as before, but provision is made to fit against DFT calculations not only for equilibrium volume, but also for slightly strained primitive cell, e.g. within +/- 2% deviation from the equilibrium vollume.

Another important subtlety relates to the fact that the DFT-calculated band-gap is unphysically low (~0.6 eV for Si, rather than the experimentally known 1.12 eV), and the objectives aim to avoid this issue in the DFTB fit.

This is accomplished by creating a couple of separate objectives for fitting the shapes of the conduction and valence bands independently, along with the objective for reaching the experimental band-gap.

The corresponding `skpar_in.yaml` is below, with comment annotations:

```
config:
    templatedir: template
    workroot: ./_workdir
    keepworkdirs: true

executables:
    skgen: ./template/skf/skgen-opt.sh
    bands: dftbutils bands

tasks:
    - set: [skf/skdefs.template.py]
    - run: [skgen, skf]
    - run: [bands, Si-diam/100]
    - get: [get_dftbp_bs, Si-diam/100/bs, Si.diam.100,
            {latticeinfo: {type: 'FCC', param: 5.431}}]
    - get: [get_dftbp_Ek, Si.diam.100,
            {sympts: ['L', 'Gamma', 'X', 'K'],
            extract: {'cb': [0,1,2,3], 'vb': [0,1,2,3]}, align: 'Evb'}]

objectives:

    - Egap:
        # if using : inside doc string, use '' or "" to surround the string
        doc: 'Si-diam-100: band-gap'
        models: Si.diam.100
        ref: 1.12
        weight: 5.0
        eval: [rms, relerr]

    - bands:
        doc: 'Si-diam-100: valence band'
        models: Si.diam.100
        ref:
```

(continues on next page)

(continued from previous page)

```
            # This is bandstructure from VASP + vasputils, which makes it
            # in the same format as DFTB + dp_bands, i.e. bands are columns
            # in the file, with each row corresponding to a k-point, and
            # the k-points are indexed in column 1 (completely redundant)
            # The advantage of this is that the band with lowest energy
            # also has the lowest column index.
            # But for visualisation, bands span horisontally, and SKPAR
            # treats the bands-type of data as a 2D array where a band
            # is represented by a ROW in the array.
            # This is why, we must always transpose bands from dp_bands
            # or from vasputils, upon loading, and this is here accomplished
            # by the loader_args: {unpack: True} -- cf. numpy.loadtxt() for details.
            file: ~/Dropbox/projects/skf-dftb/Erep fitting/from Alfred/crystal/DFT/di-
↪Si.Markov/PS.100/band/band.dat
            loader_args: {unpack: True}
            process:
                # indexes and ranges below refer to file, not array,
                # i.e. independent of 'unpack' loader argument
                rm_columns: 1      # filter k-point enumeration
                # rm_rows: [[41,60]] # filter K-L segment; must do the same with dftb␣
↪data... but in dftb_in.hsd...
                # scale     : 1    # for unit conversion, e.g. Hartree to eV, if␣
↪needed
        options:
            # Indexes below refer to the resulting 2D array after loading,
            # transposing, and application of the rm_rows/rm_columns above.
            use_ref: [[1, 4]]                # Fortran-style index-bounds of bands to␣
↪use
            use_model: [[1, 4]]
            align_ref: [4, max]              # Fortran-style index of band-index and␣
↪k-point-index,
            align_model: [4, max]            # or a function (e.g. min, max) instead␣
↪of k-point
            subweights:
                # NOTABENE:
                # -------------------------------------------------
                # Energy values are with respect to the ALIGNEMENT above.
                # If we want to have the reference  band index as zero,
                # we would have to do tricks with the range specification
                # behind the curtain, to allow both positive and negative
                # band indexes, e.g. [-3, 0], inclusive of either boundary.
                # Currently this is *not done*, so only standard Fortran
                # range spec is supported. Therefore, band 1 is always
                # the lowest lying, and e.g. band 4 is the third above it.
                # -------------------------------------------------
                dflt: 1
                values: # [[range], subweight] for E-k points in the given range of␣
↪energy
                # notabene: the range below is with respect to the alignment value
                    - [[-0.1, 0.], 5.0]
                bands: # [[range], subweight] of bands indexes; fortran-style
                    - [[2, 3], 1.5]    # two valence bands below the top VB
                    - [4 , 2.5]        # emphasize the reference band
                # not supported yet      ipoint:
        weight: 2.5
        eval: [rms, relerr]

```

(continues on next page)

(continued from previous page)

```
    - bands:
        doc: 'Si-diam-100: conduction band'
        models: Si.diam.100
        ref:
            file: ~/Dropbox/projects/skf-dftb/Erep fitting/from Alfred/crystal/DFT/di-Si.Markov/PS.100/band/band.dat
            loader_args: {unpack: True}
            process:
                rm_columns: 1       # filter k-point enumeration
                # rm_rows: [[41,60]] # filter K-L segment
        options:
            use_ref: [5, 6]                 # fortran-style index enumeration: NOTABENE: not a range here!
            use_model: [5, 6]               # using [[5,6]] would be a range with the same effect
            align_ref: [1, 9]               # fortran-style index of band and k-point, (happens to be the minimum here)
            align_model: [1, min]           # or a function (e.g. min, max) instead of k-point
            subweights:
                values:                     # [[range], subweight] for E-k points in the given range of energy
                    - [[0.0, 2.5], 1.5]     # conduction band from fundamental minimum to the band at Gamma
                    - [[0.0, 0.1], 4.0]     # bottom of CB and 100meV above, for good meff
                bands:                      # [[range], subweight] of bands indexes; fortran-style
                    - [1, 2.5]              # the LUMO only increased in weight; note the indexing
                                            # reflects the 'use_' clauses above
        weight: 1.0
        eval: [rms, relerr]

    - special_Ek:
        doc: Si-diam-100, eigenvalues at special k-points
        models: Si.diam.100
        ref:
            file: ./ref/Ek-Si.dat
            loader_args:
                dtype:                      # NOTABENE: yaml cannot read in tuples, so we must
                                            #           use the dictionary formulation of dtype
                    names: ['keys', 'values']
                    formats: ['S15', 'float']
        options:
            subweights:
                dflt   : 0.1                # changing the default (from 1.) to 0. allows us to consider
                Ec_G_0 : 0.5                # only select entries; alternatively, set select entries
                Ec_L_0 : 0.5                # to zero effectively excludes them from consideration
                Ec_X_0 : 2.0
                Ev_L_0 : 2.0
                Ev_K_0 : 2.0
```

(continues on next page)

(continued from previous page)

```
                Ev_X_0 : 2.0
        weight: 1.0
        eval: [rms, relerr]

optimisation:
    algo: PSO   # particle swarm optimisation
    options:
        npart: 2   # number of particles
        ngen : 2   # number of generations
    parameters:
        - Si_Ed  :  0.1 0.3
        - Si_r_sp:  3.5 7.0
        - Si_r_d :  3.5 8.0
```

2.4.3 Tutorial 3 – Opitmisation of repulsive potentials in DFTB

2.5 Input File Reference

SKPAR is controlled by an input file in YAML. The filename is given as an argument to the `skpar` command (e.g. `skpar skpar_in.yaml`). Examples of input files can be seen in *Tutorials*, while here we provide the full details.

The input file must contain the following four sections, which are covered in this reference. The sections of the reference correspond to sections in the input file.

```
tasks:
    # tasks defining the model

objectives:
    # objectives steering the optimisation

optimisation:
    # optimisation strategy

# optional
executables:
    # alises of executable commands used in tasks

# optional
config:
    # settings defining work directory layout
```

2.5.1 Tasks

SKPAR features dynamic model definition by the user, which greatly enhances its flexibility. The model is defined by a sequence of tasks, which represent the steps needed to obtaining model data. The tasks are declared in the input file and are executed in the given sequence at each iteration.

There are three task categories:

- **Set** – model update – updates the model environment with the parameters generated by the optimiser at a given iteration;
- **Run** – model execution – executes external programs, scripts or commands, to perform the necessary model calculations;

- **Get** – collection of model data – acquire the relevant data from the various output files created during model execution.
- **Plot** – plotting of objectives data (model and reference) – produce visual representation of objectives at each iteration; Plot-tasks are optional.

Refer to *Fig. 2. Optimisation loop realised by SKPAR.* for the corresponding steps in the optimisation flow.

The signature of each category and a brief usage is shown below.

For more complete examples see *Tutorials*.

NOTABENE:

> Tasks should be entered as list items of the `tasks:` section in the input YAML file.

Set Tasks

```
tasks:
    - set: [parameter_file, work_directory, optional_templates]
```

Set-tasks serve to communicate the parameter values generated by the optimiser to the model.

The parameters (including iteration number and possibly parameter names) are written to `parameter_file` in `work_directory`, for which standard rules apply: '~' is expanded to user directory, and if there is no path component included then it is relative to '.'. Default parameter file is `current.par`, and default work-directory is `.` if not explicitly specified.

The most important feature of the set task is its ability to update template files – `optional_templates`, according to the dictionary of parameters used in SKPAR. The following rules apply in this respect:

- `optinonal_templates` is a filename or a list of filenames with the following structure: `template.something` or `something.template.somthingelse`;
- template files must contain named place-holders which are substituted for the corresponding parameter values;
- upon substitution, the output files bear the name of the template, except for the `template.` part being removed – for example: *skdefs.template.py* becomes *skdefs.py*;
- templates are expected to be in `work_directory`, if they have no path component; else, standard path expansion applies;
- place holders should be in the old string-formatting syntax for Python, i.e. `%(parameter_name)parameter_type`; NOTABENE: NO space after closing bracket!

Run Tasks

```
tasks:
    - run: [command, work_directory, input, output]
```

Run-tasks help to define model that must be optimised in a flexible way, depending on the specific problem *without* modifying SKPAR.

The mandatory `command` argument is a string or list of strings and may include options and arguments of an external executable, shell command, or a script.

The command is executed in `work_directory`, for which standard rules apply: '~' is expanded to user directory, and if there is no path component included then it is relative to '.'. Default work-directory is '.' if not explicitly specified.

The `input` and `output` files are optional and default to `None` and `out.log` in the working directory.

Run tasks support aliasing via the *Executables* section.

Get Tasks

```
tasks:
    - get: [get_function, source, destination, func_arguments]
```

Get-tasks serve to collect model data from *source*, optionally perform some analysis on it and put the result as a key-value item at the *destination*. The *destination* is an embodiment of the database of a model, which allows queries of the values corresponding to the available keys. (see *Fig. 1. Conceptual block diagram of SKPAR.* and *Fig. 2. Optimisation loop realised by SKPAR.*).

The signature of get-tasks shown above relies on a dictionary of known functions, which are mostly model-specific.

Available get-functions

The `get_function` must be one of those accessible to the user, as as listed in `skpar/core/taskdict.py`:

Generic	
get_model_data	Generic routine based on `numpy.loadtxt()`
Specialised: DFTB+	
get_dftbp_data	Get data resulting from a DFTB+ calculation, e.g. in *detailed.out*.
get_dftbp_bs	Get all data from DFTB+/dp_bands calculation of bandstructure.
get_dftbp_meff	Extract effective masses from DFTB+/dp_bands calculation of bandstructure.
get_dftbp_Ek	Extract named *E-k* points from DFTB+/dp_bands calculation of bandstructure.

Source and Destination

Beyond the mandatory `get-function` name, a get-task must have:

- `source` (mandatory, string) – a directory name or a dictionary in the model database;
- `destination` (optional, string) – a dictionary in the model database; `source` is tried if destination is not given, and a dictionary is automatically allocated in the database when a destination is first used in a get-task;

Optional Arguments

- `func_arguments` (optional, dict) – a dictionary of key-value pairs, being keyword arguments of the get-function invoked. The possible arguments for each get-function can be checked via the table below, with links to the implementations underlying the available get-functions.

Plot Tasks

```
tasks:
    - plot: [plot_function, plot_name, list_of_objectives,
                optional_abscissa_key,
                optional_queries_list,
                kwargs]
```

Plot tasks produce `.pdf` plots with specified `plot_name`, visualising the reference and model data associated with an objective at each iteration. The filenames are tagged by an iteration number.

The data of the `list_of_objectives` is used as ordinate values. An objective is selected by a pair-list of `[query_name, model_name]`, e.g. `bands: Si`. The abscissa values may be obtained via the `optional_abscissa_key`, or alternatively, the index numbers of individual data items are used. The `optional_queries_list` allows the plotting routine to obtain extra data produced by the model at each evaluation, by declaring a query within the plot-task. Obviously, both the abscissa key and the extra query keys must be present in the model database, and this must be guaranteed by the use of appropriate get-tasks.

The plot is realised by the `plot_function`, which should be selected from the table below (follow hyperlinks for details):

`plot_objvs`	generic plotting of 1D or 2D data
`plot_bs`	specialised routine to plot bandstructures

Plot-Task Examples:

1) Generic plot of a 1D array data:

```
tasks:
    ...
    # get both yval and xval from the model and put in poly3 database
    - get: [get_model_data, test_optimise/model_poly3_out.dat,  poly3, yval]
    - get: [get_model_data, test_optimise/model_poly3_xval.dat, poly3, xval]

    # plot yval of poly3 using xval of poly3 as abscissa key
    - plot: [plot_objvs, 'test_optimise/pdf/polyfit1', [[yval, poly3]], xval]

objectives:
    - yval:
        doc: 3-rd order polynomial values for some values of the argument
        models: poly3
        ref: [ 36.55, 26.81875, 10., 13.43125,  64.45 ]
        eval: [rms, relerr]
```

2) Generic plot of a 2D array data.

This example plots a bandstructure (a fake one). Two problem with the resulting plot below is its integer x-axis, i.e. the k-line lengths are (generally) not correct, since k-point index is used as an abscissa; no labels either.

```
tasks:
    ...
    # load bands in fakemodel database; transpose input array after removing column 1
    - get: [get_model_data, reference_data/fakebands-2.dat, fakemodel, bands,
                    {loader_args: {unpack: True}, process: {rm_columns: [1]}}]
    # plot the bands using y-value index (along axis 1) as an x-value
    - plot: [plot_objvs, 'test_optimise/pdf/fakebandsplot', [[bands, fakemodel]]]

- bands:
    models: fakemodel
    ref:
        file: reference_data/fakebands.dat
        loader_args: {unpack: True}
        process:
            rm_columns: [1, 2, [8, 9]]
```

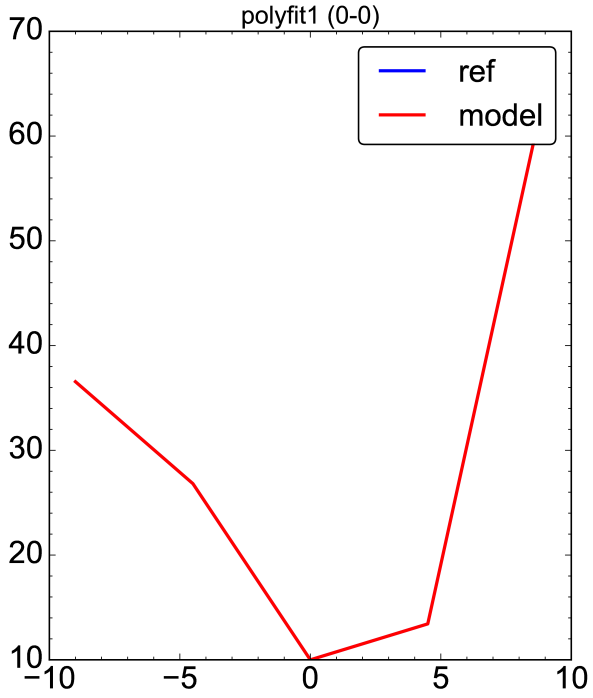

Fig. 4: **Fig. 4. 1-D array plotted with the generic ‘‘plot_objvs‘‘ function**

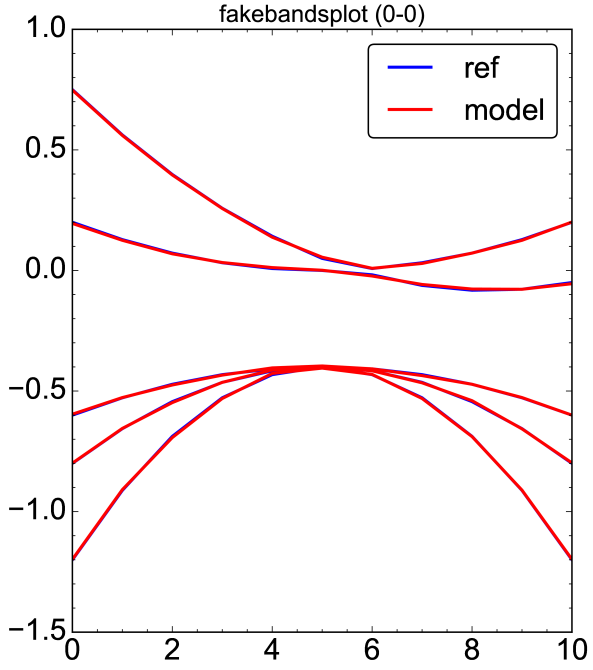

Fig. 5: **Fig. 5. 2-D array data plotted by the generic plot_objvs function**

3) Specialised plot of a bandstructure.

This example plots a bandstructure properly. For this, the x-values are constructed and passed as an abscissa value. Moreover, it shows how to handle the case where the bandstructure information is split over three different objectives – we have one objective for the band-gap, another for the valence bands and yet another for the conduction band, and both VB and CB are 0-aligned. The magic here relies on:

- strict definition of objectives with `bands` query: first VB, then CB
- strict enumeration of objectives: fisrt `Egap`, then `bands`

```
tasks:
    ...
    # Get all data from DFTB+/dp_bands. This includes all needed for BS plot,
    # including 'Egap', 'bands', 'kticklabels'
    - get: [get_dftbp_bs, Si-diam/100/bs, Si.diam.100,
            {latticeinfo: {type: 'FCC', param: 5.431}}]

    # The plot_bs does magic when it sees the first objective being 'Egap'.shape==(1,)
    # it shifts the CB by the band-gap, so the band-structure is properly shown.
    # For this to happen, objectives declaration must be such that VB precedes CB!!!
    # The plot_bs will also show k-ticks and labels if requested, as below via
    # 'kticklabels'
    - plot: [plot_bs, Si-diam/100/bs/bs_2, [[Egap, Si.diam.100], [bands, Si.diam.
↪100]],
                kvector, queries: [kticklabels]]

objectives:
    - Egap:
        doc: 'Si-diam-100: band-gap'
        models: Si.diam.100
        ref: 1.12
        weight: 5.0
        eval: [rms, relerr]

    - bands:
        doc: 'Si-diam-100: VALENCE band'
        models: Si.diam.100
        ref:
            file: ~/Dropbox/projects/skf-dftb/Erep fitting/from Alfred/crystal/DFT/di-
↪Si.Markov/PS.100/band/band.dat
            loader_args: {unpack: True}
            process:
                rm_columns: 1       # filter k-point enumeration
        options:
            use_ref: [[1, 4]]                # Fortran-style index-bounds of bands to␣
↪use
            use_model: [[1, 4]]
            align_ref: [4, max]              # Fortran-style index of band-index and␣
↪k-point-index,
            align_model: [4, max]            # or a function (e.g. min, max) instead␣
↪of k-point
        eval: [rms, relerr]

    - bands:
        doc: 'Si-diam-100: CONDUCTION band'
        models: Si.diam.100
        ref:
            file: ~/Dropbox/projects/skf-dftb/Erep fitting/from Alfred/crystal/DFT/di-
↪Si.Markov/PS.100/band/band.dat
```

(continues on next page)

(continued from previous page)

```
            loader_args: {unpack: True}
            process:
                rm_columns: 1       # filter k-point enumeration
        options:
            use_ref: [5, 6]                 # fortran-style index enumeration:␣
↪NOTABENE: not a range here!
            use_model: [5, 6]               # using [[5,6]] would be a range with the␣
↪same effect
            align_ref: [1, 9]               # fortran-style index of band and k-point,␣
↪(happens to be the minimum here)
            align_model: [1, min]           # or a function (e.g. min, max) instead of␣
↪k-point
        eval: [rms, relerr]
```

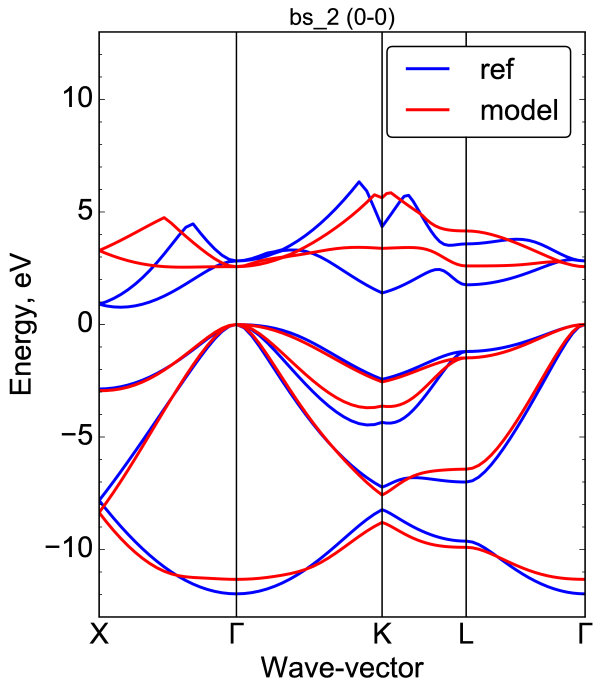

Fig. 6: **Fig. 6. Band-structure plotted by the ‘‘plot_bs‘‘ function**

2.5.2 Objectives

Central to optimisation are objectives, which in an abstract sense define the direction in which to steer the process of parameter refinement. The optimisation problem is defined as a weighted multi-objective optimisation, where each objective typically is associated with multiple data items itself. Each objective is scalarized, meaning that it is evaluated to a single scalar that represents its own cost or fitness. Each objective is assigned a weight, corresponding to its relative significance; weights are automatically normalised. Marler and Arora provide a good review on multi-objective optimisation [MOO-review].

The declaration of an objective establishes a way for a direct comparison between some reference data and some model data. With each pair of items from the reference and model data there is associated a weight, referred to as sub-weight that corresponds to the significance of this item relative to the rest of the items associated with an objective. These sub-weights are used in the scalarisation of the objective, and are also normalised.

Overview of Objectives Declaration

The declaration of an objective in the input file of SKPAR consists of the following elements:

```
objectives:
    - query: # a name selected by the end-user
        doc: "Doc-string of the objective (optional)"
        models:
            # Name of models having query_item in their database (mandatory)
        ref:
            # Reference data or instruction on obtaining it (mandatory)
        options:
            # Options for interpretation of reference/model data (optional)
        weight:
            # Weight of the objective (dflt: one)
        eval:
            # How to evaluate the objective (dflt: [rms, abserr]
```

An example of the simplest objective declaration – the band-gap of bulk Si in equilibrium diamond lattice – may look like that:

```
objectives:
    - Egap:
        doc: 'Si-diam-100: band-gap'
        models: Si.diam.100
        ref: 1.12
        weight: 5.0
        eval: [rms, relerr]
```

See also:

- *Tutorials*

Details of Objective Declaration

Query Label (`query`)

`query` is just a label given by the user. SKPAR does not interpret these labels but uses them to query the model database in order to obtain model data. Therefore, the only condition that must be met when selecting a label is that the label must be available in the database(s) of the model(s) that are listed after `models`.

It is the responsibility of the Get-Tasks to satisfy this condition. Recall that a get-task yields certain items (key-value pairs) in the dictionary that embodies the model database accessed as a destination of the task.

Certain get-tasks allow the user to define the key of the item, and this key can be used as a query-label when declaring an objective. Example of that is shown in *Tutorial 1*, where the simple `get_model_data` task is used, and the query label is `yval`.

Other tasks yield a fixed set of items – examples are the get-tasks provided by the `dftbutils` package. Please, consult their documentation to know which items are available as query-labels: *Available get-functions*.

There is one case however, in which the above significance of `query` is disregarded, and the specified label becomes irrelevant. This is the case where the reference data of an objective is itself a dictionary of key-value pairs (or results in such upon acquisition from a file). This case is automatically recognised by SKPAR and the user need not do anything special. The query-label in this case can be something generic. Example of such an objective can be found in *Tutorial 2*, with queries labeled as `effective_masses` or `special_Ek`.

Doc-string (doc)

This is an optional description – preferably very brief, which would be used in reporting the individual fitness of the objective, and also as a unique identifier of the objective (complementary to its index in the list of objectives). If not specified, SKPAR will assign the following doc-string automatically: `doc: "model_name: query_item"`.

Model Name(s) (models)

This is a single name, or a list of names given by the user, and is a mandatory field. A model name given here must be available in the model database. For this to happen, the model must appear as a *destination* of a Get-Task declaration (see *Get Tasks*).

Beyond a single model name and a list of model names, SKPAR supports also a list of pairs – [model-name, model-factor]. In such a definition, the data of each model is scaled by the model-factor, and a summation over all models is done, prior to comparison with reference data.

Therefore, the three (nonequivalent) ways in which models can be specified are:

```
objectives:
    - query:
        # other fields
        models: name   # or [name, ]
        # or
        models: [name1, name2, name3..., nameN]
        # or
        models:
            - [name1, factor1]
            - [name2, factor2]
            # ...
            - [nameN, factorN]
```

Reference Data (ref)

Reference data could be either explicitly provided, e.g.: `ref: [1.5, 23.4]`, or obtained from a file. The latter gives flexibility, but is correspondingly more complicated.

Loading data from file is invoked by:

```
objectives:
    - query
        # other fields in the declaration
        ref:
            file: filename
            # optional
            loader_args: {key:value-pairs}
            # optional
            process:
                # processing options
```

SKPAR loads data via Numpy loadtxt() function, and the optional arguments of this function could be specified by the user via `loader_args`

Typical loader-arguments are:

- `unpack: True` – transposes the input data; mandatory when loading band-structure produced from `dp_bands` or `vasputils`

- `dtype: {names: ['keys', 'values'], formats: ['S15', 'float']}` – loads string-float pairs; mandatory when the reference data file consists of key-value pairs per line.

The `process` options are interpreted only for 2D array data (ignored otherwise), and are as follows:

- `rm_columns: index, list_of_indices, or, range_specification`
- `rm_rows: index, list_of_indices, or, range_specification`
- `scale: scale_factor`

NOTABENE: The indexes apply to the rows and columns of the file, and are therefore independent of the loader arguments (i.e. prior to potential transpose of the data). The indexes and index ranges are Fortran-style – counting from 1, and inclusive of boundaries.

Example:

```
objectives:
    - query:
      ...
      ref:
        file: filename
        process:
            rm_columns: 1                # filter k-point enumeration, and bands, potentially
            rm_rows   : [[18,36], [1,4]] # filter k-points if needed for some reason
            scale     : 27.21            # for unit conversion, e.g. Hartree to eV, if needed
      ...
```

Objective Weight (`weight`)

This is a scalar, corresponding to the relative significance of the objective compared to the other objectives. Objective weights are automatically normalised so that there sum is one.

Evaluation function (`eval`)

Each objective is scalarised by a cost function that can be optionally modified here. Currently only Root-Mean-Squared Deviation is supported, but one may choose whether absolute or relative deviations are used. The field is optional and defaults to RMS of absolute deviations.

```
objectives:
    ...
    - query:
        ...
        eval: [rms, abserr] # default, absolute deviations used
        # or
        eval: [rms, relerr] # relative devations
```

Options (`options`)

Options depend on the type of objective. One common option is `subweights`, which allows the user to specify the relative importance of each data-item in the reference data. These sub-weights are used in the cost-function representing the individual objective.

For details, see the sub-weights associated with different *Reference Data And Objective Types* below.

Reference Data And Objective Types

The format of reference data could be:

- single item: e.g. a scalar representing the band-gap of a semiconductor, or a reaction energy;
- 1-D array: e.g. the energy values of an energy-volume relation of a solid;
- 2-D array: e.g. the band-structure of a solid, i.e. the set of eigenstates at different *k*-number;
- key-value pairs: e.g. named physical quantities, like effective masses, specific E-k points within the first Brilloin zone, etc.

Declaring an objective for a single model is straight forward, and in this case a signle item reference data may be thought of as a special case of 1-D array. However, the distinction between the two makes sense if we want to construct an objective based on more than one models, as shown further below.

There are five types of objectives. The type is deduced from the combination of *format of the reference data* and *number of model names*. Therefore, SKPAR automatically distinguishes between the following five objectives types:

1) Single model, single item/1-D array reference data

This is the simplest objective type that associates one or more (1-D array) items with a query of one model.

In the case of an array reference data, one option is admitted: `subweights`. The number of sub-weights must match the length of the reference data array. Sub-weights are normalised automatically.

Example:

```
objectives:
    # single model, single scalar reference data
    - band_gap:
        ref: 1.12
        models: Si/bs

    # single model, 1-D array reference data
    - levels:
        ref: [-13.6, -5, -3.0]
        options:
            subweights: [1, 1, 2]
        models: molecule
```

2) Single model, 2-D array reference data

This objective type allows for greater flexibility in defining the association between individual reference and model data items, which may not be the trivial one-to-one correspondence between the entire arrays yielded by the query item.

The 2-D arrays (of reference and model data) are viewed as composed of *bands* – each row is referred to as a *band*, each column is referred to as a *point*. A visual representation of the concept is shown in *Fig. 3*.

Correspondence between different bands of the model and reference data can be established via the following options:

- `use_ref: [...]`
- `use_model: [...]`
- `align_ref: [...]`
- `align_model: [...]`

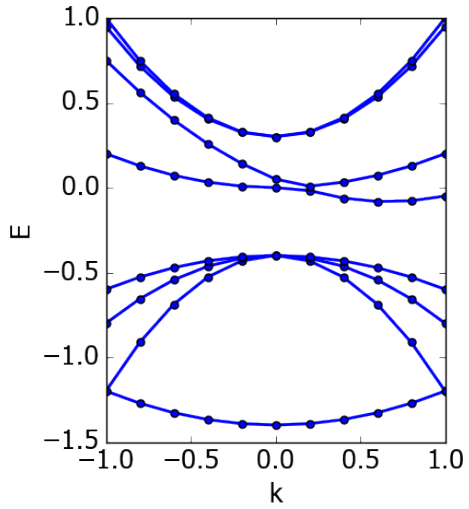

Fig. 7: **Fig. 3. Visual representation of a 2-D array in terms of bands. Each data item in the array is a circle on the plot. The lines represent the association of a row of data in the array with a band.**

The `use_` options admit a list of indexes, ranges, or a mix of indexes and ranges – e.g. `[[1, 4], 7, 9]`, and instruct SKPAR to retain only the enumerated bands for the comparison of the resulting 2-D sub-arrays of model and reference data. Fortran-style indexing must be used, i.e. counting starts from 1, and ranges are inclusive of both boundaries.

NOTABENE: In any case, the final comparison (model *vs* reference) is over arrays of identical shape, and the resulting arrays after the `use_` clause should be of the same shape.

The `align_` options instruct SKPAR to subtract the value of a specific data item in the array from all values in the array, i.e. change the reference. This option admits a pair of band-index and point-index, or a pair of band-index and a function name (`min` or `max`) to operate on the indexed band. In either case, the value of the indexed data item or the value returned from the function is subtracted from the model or reference data prior to objective evaluation.

This objective type also admits `subweights` option, but in this case the correspondence between sub-weights and data items needs more flexible specification. This is to avoid the necessity of providing a full 2-D array of sub-weight coefficients for each data item. The following sub-options facilitate that:

```
objectives:
    ...
    - bands:
        ...
        options:
            ...
            subweights:
                dflt: 1.0
                values: [[[value range], subweight], ...]
                bands: [[[band-index range], subweight], ...]
```

Note: Correspondence between sub-weights and data, per data item, is established **after** the application of `use_` and `align_` options have taken effect.

Example:

```
objectives:

    - bands:
        doc: Valence Band, Si
        models: Si/bs
        ref:
            file: ./reference_data/fakebands.dat #
            loader_args: {unpack: True}
            process:       # eliminate unused columns, like k-pt enumeration
                # indexes and ranges below refer to file, not array,
                # i.e. independent of 'unpack' loader argument
                rm_columns: 1                # filter k-point enumeration, and bands,␣
↪potentially
                # rm_rows   : [[18,36], [1,4]] # filter k-points if needed for some␣
↪reason
                # scale     : 1                # for unit conversion, e.g. Hartree to␣
↪eV, if needed
        options:
            use_ref: [[1, 4]]                # fortran-style index-bounds of bands to␣
↪use
            use_model: [[1, 4]]
            align_ref: [4, max]              # fortran-style index of band and k-
↪point,
            align_model: [4, max]            # or a function (e.g. min, max) instead␣
↪of k-point
            subweights:
                # NOTABENE:
                # ------------------------------------------------
                # Energy values are with respect to the ALIGNEMENT.
                # If we want to have the reference  band index as zero,
                # we would have to do tricks with the range specification
                # behind the curtain, to allow both positive and negative
                # band indexes, e.g. [-3, 0], INCLUSIVE of either boundary.
                # Currently this is not done, so only standard Fortran
                # range spec is supported. Therefore, band 1 is always
                # the lowest lying, and e.g. band 4 is the third above it.
                # ------------------------------------------------
                dflt: 1
                values: # [[range], subweight] for E-k points in the given range of␣
↪energy
                # notabene: the range below is with respect to the alignment value
                    - [[-0.3, 0.], 3.0]
                bands: # [[range], subweight] of bands indexes; fortran-style
                    - [[2, 3], 1.5]    # two valence bands below the top VB
                    - [4 , 3.5]        # emphasize the reference band
        weight: 3.0
```

3) Single model, key-value pairs reference data

For this objective, a number of queries are made over a single model. The reference data is a dictionary of key-value pairs. Note that the name of the objective (*meff* below) has a generic meaning, and is *not* defining the query items. Instead, The queries are based on the keys from the reference data.

One options is admitted – `subweights`, and its value must be a dictionary associating a weighting coefficient with a key. One of the subweight-keys is `dflt`, allowing to specify a weight simultaneously over all keys. The subweights are normalised automatically. A key is excluded from the queries if its sub-weight is 0.

Example:

```
objectives:

    - meff:
        doc: Effective masses, Si
        models: Si/bs
        ref:
            file: ./reference_data/meff-Si.dat
            loader_args:
                dtype:
                # NOTABENE: yaml cannot read in tuples, so we must
                #           use the dictionary formulation of dtype
                    names: ['keys', 'values']
                    formats: ['S15', 'float']
        options:
            subweights:
                # consider only a couple of entries from available data
                dflt: 0.
                me_GX_0: 2.
                mh_GX_0: 1.
        weight: 1.5
```

4) Multiple models, single item reference data

All of the models are queried individually for the same query-item, and the result is scaled by the non-normalised model-weights or model factors, prior to performing summation over the data, to produce a single scalar. This scalar is seen as a new model data item. Accordingly, reference data is a single value too. This type of objective is convenient for expressing reaction energies as targets.

Example:

```
objectives:
    - Etot:
        doc: "heat of formation, SiO2"
        models:
            - [SiO2-quartz/scc, 1.]
            - [Si/scc, -0.5]
            - [O2/scc, -1]
        ref: 1.8
```

5) Multiple models, 1-D array reference data

A single query per model is performed, over several models. The dimension of the 1-D array of reference data must match the number of models.

The admitted option is `subweights` – a list of floats, being normalised weighting coefficients in the cost function of the objective. This type of objective is convenient for expressing energy-volume relation as target, where the different models correspond to different volume.

Example:

```
objectives:
    - Etot:
        models: [Si/scc-1, Si/scc, Si/scc+1,]
```

(continues on next page)

(continued from previous page)

```
        ref: [23., 10, 15.]
        options:
            subweights: [1., 3., 1.,]
```

REFERENCES

2.5.3 Optimisation

Optimisation is driven towards cost minimisation. The cost is evaluated at each iteration and new parameters are generated according to a prescribed algorithm. The user could select an algorithm and set the options specific to the algorithm in the `optimisation:` section of the input file. Declaration of parameters is done in the same section too.

Example:

```
optimisation:
    algo: PSO                       # algorithm: particle swarm optimisation

    options:                        # algorithm specific
        npart: 8                    # number of particles
        ngen : 100                  # number of generations

    parameters:
        - Si_Ed  :  0.1 0.3         # parameter names must match with placeholders
        - Si_r_sp:  3.5 7.0         # in template files given to set-tasks above
        - Si_r_d :  3.5 8.0
```

Cost function

The overall cost function is:

$$G(\{\lambda_p\}) = \sqrt{\left(\sum_j^N \omega_j F_j(\{\lambda_p\})^2\right)}$$

where λ_p are the parameters, $F_j()$ are the N individual objective functions (called objectives, for brevity), and ω_j are the weights associated with each objective.

The scalarisation of individual objectives allows one to declare objectives related to different types of physical quantities and magnitudes, and adjust separately their contribution towards to overall cost via the objective weights.

Weights represent the relative significance of the different objectives towards the overall cost, and are automatically normalised:

$$\omega_j = \omega_j / \sum_j \omega_j$$

Each objective yields a scalar according to its own cost function:

$$F_j(\{\lambda_p\}) = \sqrt{\sum_i^M \omega_i \Delta_i(\{\lambda_p\})^2}$$

where Δ_i are the deviations between model and reference data for each of the M data item associated with an objective (see *Objectives* for details of the declaration), and ω_i are the sub-weights associated with each data-item. Sub-weights are also normalised internally.

These deviations may be either absolute or relative, i.e.: $\Delta_i = m_i - r_i$ or $\Delta_i = (m_i - r_i)/r_i$, with special treatment applied in the latter case where denominator vanishes. Specifically, if both m_i and r_i vanish, $\Delta_i = 0$, while for a finite m_i, $\Delta_i = (m_i - r_i)/m_i$.

Optimisation Algorithm

Currently SKPAR supports only Particle Swarm Optimisation algorithm. The implementation follows Eq.(3) in [PSO-1] by J. Kennedy; See also the equivalent and more detailed Eqs(3-4) in [PSO-2].

This algorithm accepts only two options at present:

- `npart` – number of particles in the swarm
- `ngen` – number of generations through which the swarm must evolve

Each of the parameters to be optimised represents a degree of freedom for each particle. Since parameters may have different physical units and magnitudes, the parameters are internally normalised within the PSO optimiser. Upon generation of parameter values by the PSO, these are automatically re-normalised to yield their physical significance upon passing to the evaluator.

See the module reference for implementation details (*PSO (skpar.core.pso)*).

Note that the PSO is a stochastic algorithm, and it reports basic statistics of the cost associated with each iteration.

An iteration is tagged by the pair `(generation, particle)` throughout the report and log messages of the optimiser.

Parameter declaration

From the viewpoint of an optimiser, the minimal required information related to parameters is their number. However, SKPAR permits a more extensive declaration of parameters:

```
optimisation:
    ...
    parameters:
        - name: initial_value min_value max_value optional_type
        # or
        - name: initial_value optional_type
        # or
        - name: min_value max_value optional_type
        # or
        - name: optional type
```

The names of the parameters are important for using template files in Set-Tasks (see *Set Tasks*), and for reporting/logging purposes.

The default `type` of all parameters is float (`f`), but integer `i` may be supported in the future by different algorighms.

For the PSO algorithm, the initial value is ignored, so specifying the minimal and maximal value is sufficient.

References

2.5.4 Executables

Executables are simple aliases to more complex commands invoking external executables. The alias may contain command-line arguments and options, a path to the actual command, etc.

Examples:

```
# define aliases to run-task commands
executables:
    # alias an executable found along $PATH
    atom: gridatom
    # alias a shell script in ./skf/ directory
    skgen: skf/skgen.sh
    # alias a command including input arguments
    dftb: mpirun -n 4 dftb+
    # alias a command including input arguments
    bands: ~/sw/dp_tools/dp_bands band.out bands

# use the aliases
tasks:
    - run: [skgen, skf, skdefs.py]
    - run: [dftb,  Si/bs, out.dftb]
    - run: [bands, Si/bs, out.bands]
```

2.5.5 Configuration

Configuration of the working directory allows for a choice whether to execute each evaluation step in a separate subdirectory (labeled by iteration number). Thus it permits to save each model evaluation and is the first step towards parallelisation.

Examples:

```
config:
    # Template files and directories are copied to the individual
    # iteration directory under work-root; default is ``.``.
    templatedir: template

    # Workroot is the directory where each iteration dir will go
    # Default is ``.``.
    workroot: _workdir

    # All results are kept if true
    # NOTABENE: if true, then workroot may become very large!!!
    # NOTABENE: if false (DEFAULT), then plots should be written outside
    #           workroot; else will be destroyed.
    keepworkdirs: true
```

The complete example can be found in the examples/C.dia directory, while the directory tree layout after the run is recorded in examples/C.dia/workdir.tree.

2.6 Subpackage/module Reference

SKPAR currently includes the following sub-packages:

core	the core modules realising the optimisation framework
dftbutils	the modules related to a DFTB model

2.6.1 core sub-package modules

Main Program (skpar.core.skpar)

Main environment of SKPAR

class skpar.core.skpar.**SKPAR**(*infile='skpar_in.yaml'*, *verbose=True*)
Bases: object

The main executable object.

Input handler (skpar.core.input)

Routines to handle the input file of skpar

skpar.core.input.**get_config**(*userinp*, *report=True*)
Parse the arguments of 'config' key in user input

skpar.core.input.**get_input**(*filename*)
Read input; Exception for non-existent file.

skpar.core.input.**parse_input**(*filename*, *verbose=True*)
Parse input filename and return the setup

Tasks (skpar.core.tasks)

Tasks module, defining relevant classes and functions

class skpar.core.tasks.**Task**(*name*, *func*, *fargs*)
Bases: object

Generic wrapper over functions or executables.

skpar.core.tasks.**check_taskdict**(*tasklist*, *taskdict*)
Check task names are in the task dictionary; quit otherwise.

skpar.core.tasks.**get_tasklist**(*userinp*)
Return a list of tuples of task names and task arguments.

skpar.core.tasks.**initialise_tasks**(*tasklist*, *taskdict*, *report=False*)
Transform a tasklist into a list of callables as per taskdict.

Parameters

- **tasklist** (*list*) – a list of (task-name, user-arguments) pairs
- **taskdict** (*dict*) – a dictionary mapping task names to actual functions

Returns callable objects, instances of Task class

Return type tasks(list)

Task Dictionary (skpar.core.taskdict)

Dictionary with default tasks and their underlying functions.

class skpar.core.taskdict.**PlotTask**(*func*, *plotname*, *objectives*, *abscissa_key=None*, ***kwargs*)
Wrapper for skparplot; extracts data from objectives prior to plotting.

This is a callable object that plots to file the model and reference data associated with one or more objectives. The model and reference data constitute the Y-coordinates (ordinates). The X-coordinates (abscissas) are potentially

held in a separate field of the model data dictionary, and implicitly it is assumed that the X-coordinates of the reference data are the same (else the model-vs-reference comparison would make no sense). The fundamental concept is that we want to visualise our objectives. So the ordinate can be obtained by the user's stating which objectives is to be visualised. The challenge is that a definition of objective contains no info about the abscissa, so it has to be explicitly specified by the user or else the default indexing of the reference or model data items will be used as abscissa. The whole mechanism must work with the simplest possible (default) plotting routine, as well as with a more specialised plotter object. The initialisation of the PlotTask should establish what dictionary items are to be plotted as abscissas and ordinates and from which model dictionary, and how the latter are matched to the corresponding reference data. Note however, that objectives may not be visible at the time the task is initialised. So at init time, we merely record the user's directions. Later – at call time – we do the data queries and call the plot function with the latest model data.

pick_objectives(*objectives*, *database*)

Get the references corresponding to the objective tags.

This function acquired the reference data that must be plotted, by analysing the objectives referred to in the definition of the plot task. It is not called within the init of the plot task itself, since at the time the plot task is being declared, the objectives may not yet be. So a separate agency is suppoosed to call this method once both objectives and task are declared. Currently this happens within input.py – at the end of processing of the input file.

skpar.core.taskdict.**execute**(*implargs*, *database*, *cmd*, *workdir='.'*, *outfile='run.log'*, *purge_workdir=False*, ***kwargs*)

Execute external command in workdir, streaming output/error to outfile.

Parameters

- **implargs** (*dict*) – caller environment variables
- **database** (*dict-like*) – not used, but needed to maintain a task-signature
- **cmd** (*str*) – command; executed in *implargs['workroot']+workir*; if it contains *$* or *-globbing, these are shell-expanded
- **workdir** (*path-like*) – execution directory relative to workroot
- **outfile** (*str*) – output file for the stdout/stderr stream; continuously updated during execution
- **purge_workdir** (*bool*) – if true, any existing working directory is purged
- **kwargs** (*dict*) – passed directly to the underlying *subprocess.call()*

Returns None

Raises

- OSError – if *cmd* cannot be executed
- RuntimeError – if *cmd* returncode is nonzero
- SubprocessError – other possible circumstances

skpar.core.taskdict.**get_model_data**(*implargs*, *database*, *item*, *source*, *model*, *rm_columns=None*, *rm_rows=None*, *scale=1.0*, ***kwargs*)

Get data from file and put it in a database under a given key.

Use numpy.loadtxt to get the data from *source* file and write the data to *database* under *dst*.'key' field. If *dst* does not exist, it is created. All *kwargs* are directly passed to numpy.loadtxt. Additionally, some post-processing can be done (removing rows or columns and scaling).

Parameters

- **implargs** (*dict*) – dictionary of implicit arguments from caller
- **database** (*object*) – must support dictionary-like get/update()
- **source** (*str*) – file name source of data; path relative to implargs[workroot]
- **model** (*str*) – model name to be updated in *database*
- **key** (*str*) – key under which to store the data in under *dst*
- **rm_columns** – [index, index, [ilow, ihigh], otherindex, [otherrange]]
- **rm_rows** – [index, index, [ilow, ihigh], otherindex, [otherrange]]
- **scale** (*float*) – multiplier of the data

skpar.core.taskdict.**parse_cmd**(*cmd*)
Parse shell command for globbing and environment variables.

skpar.core.taskdict.**prepare_for_plotsave**(*iteration*, *filename*)
Ensure directory of filename exists and embed iteration number

skpar.core.taskdict.**substitute_parameters**(*implargs*, *database*, *templatefiles*, ***kwargs*)
Substitute parameters (within implicit arguments) in given templates.

skpar.core.taskdict.**wrapper_PlotTask**(*env*, *database*, **args*, ***kwargs*)
Wrapper around the legacy PlotTask

Objectives (skpar.core.objectives)

Classes and functions related to the:

- parsing the definition of objectives in the input file,
- setting the objectives for the optimizer, and,
- evaluation of objectives.

class skpar.core.objectives.**ObjBands**(*spec*, ***kwargs*)
Bases: *skpar.core.objectives.Objective*

get(*database*)
Return the value of the objective function.

class skpar.core.objectives.**ObjKeyValuePairs**(*spec*, ***kwargs*)
Bases: *skpar.core.objectives.Objective*

get(*database*)
Return the corresponding model data, reference data, and sub-weights. This method must be overloaded in a child-class if a more specific way to yield the model data in required.

class skpar.core.objectives.**ObjValues**(*spec*, ***kwargs*)
Bases: *skpar.core.objectives.Objective*

get(*database*)
Get the model data, align/mask it etc, and return calculated cost.

class skpar.core.objectives.**ObjWeightedSum**(*spec*, ***kwargs*)
Bases: *skpar.core.objectives.Objective*

get(*database*)

class skpar.core.objectives.**Objective**(*spec*, ***kwargs*)
Bases: object

Decouples the declaration of an objective from its evaluation.

Objectives are declared by human input data that defines:

- reference data,
- models - from which to obtain model data, and possibly model weights,
- query - the way to obtaining data
- model weights - relative contribution factor of each model,
- options, e.g. to specify sub-weights of individual reference items,
- relative weight of the objective, in the context of multi-objective optimisation.

Instances are callable, and return a triplet of model data, reference data, and sub-weights of relative importance of the items within each data.

evaluate(*database=None*)
Evaluate objective, i.e. fitness of the current model against the reference.

get()
Return the corresponding model data, reference data, and sub-weights. This method must be overloaded in a child-class if a more specific way to yield the model data in required.

summarise()

skpar.core.objectives.**get_models**(*models*)
Return the models (names) and corresponding weights if any.

Parameters **(str, list of str, list of [str** (*models*) – float] items): The string is always a model name. If [str: float] items are given, the float has the meaning of weight, associated with the model.

Returns

(model_names, model_weights). Weights are set to 1.0 if not found in *models*. Elements of the tuple are lists if *models* is a list.

Return type tuple

skpar.core.objectives.**get_objective**(*spec*, ***kwargs*)
Return an instance of an objective, as defined in the input spec.

Parameters **spec** (*dict*) – a dictionary with a single entry, being query: {dict with the spec of the objective}

Returns

an instance of the Objective sub-class, corresponding an appropriate objective type.

Return type list

skpar.core.objectives.**get_refdata**(*data*)
Parse the input data and return a corresponding array.

Parameters **data** (*array or array-like, or a dict*) – Data, being the reference data itself, or a specification of how to get the reference data. If dictionary, it should either contain key-value pairs of reference items, or contain a 'file' key, storing the reference data.

Returns

an array of reference data array, subject to all loading and post-processing of a data file, or pass *data* itself, transforming it to an array as necessary.

Return type array

skpar.core.objectives.**get_refval**(*bands*, *align*, *ff={'max': <function amax>, 'min': <function amin>}*)

Return a reference (alignment) value selected from a 2D array.

Parameters

- **bands** (*2D numpy array*) – data from which to obtain a reference value.
- **align** – specifier that could be (band-index, k-point), or (band-index, function), e.g. (3, 'min'), or ('7, 'max')
- **ff** (*dict*) – Dictionary mapping strings names to functions that can operate on an 1D array.

Returns the selected value

Return type value (float)

skpar.core.objectives.**get_refval_1d**(*array*, *align*, *ff={'max': <function amax>, 'min': <function amin>}*)

Return a reference (alignment) value selected from an array.

Parameters

- **array** (*1D numpy array*) – data from which to obtain a reference value.
- **align** – specifier that could be and index, e.g. 3, or 'min', 'max'
- **ff** (*dict*) – Dictionary mapping string names to functions that can operate on an 1D array.

Returns the selected value

Return type value (float)

skpar.core.objectives.**get_subset_ind**(*rangespec*)

Return an index array based on a spec – a list of ranges.

skpar.core.objectives.**get_type**(*n_models*, *ref*, *dflt_type='values'*)

Establish the type of objective from attributes of reference and models.

skpar.core.objectives.**parse_weights**(*spec*, *refdata=None*, *nn=1*, *shape=None*, *i0=0*, *normalised=True*, *ikeys=None*, *rikeys=None*, *rfkeys=None*)

Parse the specification defining weights corresponding to some data.

The data may or may not be specified, depending on the type of specification that is provided. Generally, the specification would enumerate either explicit indexes in the data, or a range of indexes in the data or a range of values in the data, and would associate a weight with the given range. A list of floats is also accepted, and an array view is returned, for cases where weights are explicitly enumerated, but no check for length.

To give freedom of the user (i.e. the caller), the way that ranges are specified is enumerated by the caller by optional arguments – see *ikeys*, *rikeys* and *rfkeys* below.

Parameters

- **spec** (*array-like or dict*) – values or specification of the subweights, for example:

  spec = dflt: 1.0 # default value of subweights indexes: # explicit [index, weight] for 1d-array data

  - [0, 1]
  - [4, 4]

– [2, 1]

ranges: # ranges for 1d-array

– [[1,3], 2]

– [[3,4], 5]

bands: # ranges of bands (indexes) in bands (refdata)

– [[-3, 0], 1.0] # all valence bands

– [[0, 1], 2.0] # top VB and bottom CB with higher weight

values: # ranges of energies (values) in bands (refdata)

– [[-0.1, 0.], 4.0]

– [[0.2, 0.5], 6.0]

indexes: # explicit (band, k-point) pair (indexes) for bands (refdata)

– [[3, 4], 2.5]

– [[1, 2], 3.5]

- **refdata** (*numpy.array*) – Reference data; mandatory only when range of values must be specified
- **nn** (*int*) – length of *refdata* (and corresponding weights)
- **shape** (*tuple*) – shape of *reference* data, if it is array but not given
- **i0** (*int*) – index to be assumed as a reference, i.e. 0, when enumerating indexes explicitly or by a range specification.
- **ikeys** (*list of strings*) – list of keys to be parsed for explicit index specification, e.g. [‘indexes’, ‘Ek’]
- **rikeys** (*list of strings*) – list of keys to be parsed for range of indexes specification, e.g. [‘ranges’, ‘bands’]
- **rfkeys** (*list of strings*) – list of keys to be parsed for range of values specification, e.g. [‘values’, ‘eV’]

Returns the weight to be associated with each data item.

Return type numpy.array

skpar.core.objectives.**parse_weights_keyval**(*spec*, *data*, *normalised=True*)

Parse the weights corresponding to key-value type of data.

Parameters

- **spec** (*dict*) – Specification of weights, in a key-value fashion. It is in the example format:

```
{ 'dflt': 0., 'key1': w1, 'key3': w3}
```

with w1, w3, etc. being float values.

- **data** (*structured numpy array*) – Data to be weighted.

Typical way of obtaining *data* in this format is to use:

```
loader_args = {'dtype': [('keys', 'S15'), ('values', 'float')]}
data = numpy.loadtxt(file, **loader_args)
```

Returns

weights corresponding to each key in *data*, with the same length as *data*.

Return type numpy.array

skpar.core.objectives.**set_objectives**(*spec*, *verbose=True*, ***kwargs*)
Parse user specification of Objectives, and return a list of Objectives for evaluation.

Parameters **spec** (*list*) – List of dictionaries, each dictionary being a, specification of an objective of a recognised type.

Returns

a List of instances of the Objective sub-class, each corresponding to a recognised objective type.

Return type list

Query (skpar.core.query)

Evaluator (skpar.core.evaluate)

Evaluator engine of SKPAR.

class skpar.core.evaluate.**Evaluator**(*objectives*, *tasklist*, *taskdict*, *parameternames*, *config=None*, *costf=<function cost_rms>*, *utopia=None*, *verbose=False*)
Bases: object

Evaluator

The evaluator is the only thing visible to the optimiser. It has several things to do:

1. Accept a list of parameter values (or key-value pairs), and an iteration number (or a tuple: (e.g. generation,particle_index)).
2. Update tasks (and underlying models) with new parameter values.
3. Execute the tasks to obtain model data with the new parameters.
4. Evaluate fitness for individual objectives.
5. Evaluate global fitness (cost) and return the value. It may be good to also return the max error, to be used as a stopping criterion.

evaluate(*parametervalues*, *iteration=None*)
Evaluate the global fitness of a given point in parameter space.

This is the only object accessible to the optimiser, therefore only two arguments can be passed.

Parameters

- **parametervalues** (*list*) – current point in design/parameter space
- **iteration** – (int or tupple): current iteration or current generation and individual index within generation

Returns global fitness of the current design point

Return type fitness (float)

skpar.core.evaluate.**abserr**(*ref*, *model*)
Return the per-element difference model and reference.

skpar.core.evaluate.**cost_rms**(*ref*, *model*, *weights*, *errf=<function abserr>*)
Return the weighted-RMS deviation

skpar.core.evaluate.**create_workdir**(*workdir*, *templatedir*)
Create a new and clean work directory tree from template

skpar.core.evaluate.**destroy_workdir**(*workdir*)
Remove the entire work directory tree

skpar.core.evaluate.**eval_objectives**(*objectives*, *database*)
Evaluate fitness/cost

skpar.core.evaluate.**get_workdir**(*iteration*, *workroot*)
Find what is the root of the work-tree at a given iteration

skpar.core.evaluate.**relerr**(*ref*, *model*)
Return the per-element relative difference between model and reference.

To handle cases where *ref* vanish, and possibly *model* vanish at the same time, we:

- translate directly a vanishing absolute error into vanishing relative error (where both *ref* and *model* vanish.
- take the model as a denominator, thus yielding 1, where *ref* vanishes but *model* is non zero

Optimiser (skpar.core.optimise)

Defines a wrapper around user selectable optimisation engines.

class skpar.core.optimise.**Optimiser**(*algo*, *parameters*, *evaluate*, *options=None*, *verbose=True*)
Bases: object

Wrapper for different optimization engines.

report(**args*, ***kwargs*)
Report optimiser state.

skpar.core.optimise.**get_optargs**(*userinp*)
Parse user input for optimisation related arguments.

Parameters (skpar.core.parameters)

Parameters

Module for handling parameters in the context of optimisation.

The following assumptions are made:

- parameters have no meaning to the optimiser engine
- for the optimiser, parameters are just a list of values to be manipulated
- the user must create an OrderedDict of parameter name:value pairs; the OrderedDict eliminates automatically duplicate definitions of parameters, yielding the minimum possible number of degrees of freedom for the optimiser
- the OrderedDictionary is built by parsing an skdefs.template file that contains parameter-strings (conveying the name, initial value and range).
- skdefs.template is parsed by functions looking for a given format from which parameter-strings are extracted
- the Class Parameter is initialised from a parameter-string.

- a reduced skdefs.template is obtained, by removing the initial value and range from the input skdefs.template which is ready for creating the final skdefs file by string.format(dict(zip(ParNames,Parvalues)) substitution.

class skpar.core.parameters.**Parameter**(*string*, ***kwargs*)
Bases: object

A parameter object, that is initialised from a string.

ParameterName InitialValue MinValue Maxvalue [ParameterType] ParameterName MinValue Maxvalue [ParameterType] ParameterName InitialValue [ParameterType] ParameterName [ParameterType]

ParameterName must be alphanumeric, allowing _ too. Iinit/Min/MaxValue can be integer or float ParameterType is optional (float by default), and indicated by either 'i'(int) or 'f'(float) White space separation is mandatory.

typedict = {'f': <class 'float'>, 'i': <class 'int'>}

skpar.core.parameters.**get_parameters**(*userinp*)
Parse user input for definitions of parameters.

The definitions should be of the form ('name', 'optionally_something'). The optional part could in principle be one or more str/float/int.

skpar.core.parameters.**substitute_template**(*parameters*, *parnames*, *templatefile*, *resultfile*)
Substitute a template with actual parameter values.

Parameters

- **parameters** (*list*) – Parameter list, items being either floats or objects with .value and .name attributes.
- **parnames** (*list*) – If *parameters* is a list of floats, then parnames is the list of corresponding names.
- **templatefile** (*str*) – Name of template file with substitution patterns.
- **resultfile** (*str*) – Name of file to contain the substituted result.

skpar.core.parameters.**update_parameters**(*workroot*, *templates*, *parameters*, *parnames=None*)
Update relevant templates with new parameter values.

Parameters

- **workroot** (*str*) – Root working directory, template names are relative to this directory.
- **templates** (*list*) – List of ascii filenames containing placeholders for inserting parameter values. The place holders must follow the old string formatting of Python: %(ParameterName)ParameterType.
- **parameters** (*list*) – Either a list of floats, or a list of objects (each having .value and .name attributes)
- **parnames** (*list*) – If *parameters* is a list of floats, then parnames is the list of corresponding names.

skpar.core.parameters.**update_template**(*template*, *pardict*)
Makes variable substitution in a template.

Parameters

- **template** (*str*) – Template with old style Python format strings.
- **pardict** (*dict*) – Dictionary of parameters to substitute.

Returns String with substituted content.

Return type str

PSO (skpar.core.pso)

Particle Swarm Optimizer (PSO)

Particles

In PSO, a particle represents a set of parameters to be optimised. Each parameter is therefore a degree of freedom of the particle. Each particle is represented by its coordinate value. Additionally it needs several attributes:

- fitness – the quality of the set of parameters
- speed – how much the position of the particle changes from one generation to the next
- smin/smax – speed limits (observed only initially, in the current implementation)
- best – particles own best position (i.e. with best fitness)

Particle normalisaton/re-normalisation

Additionally to the above generic PSO-related attributes, we need to introduce position normalisation as follows. The parameters giving the particle coordinates may be with different physical meaning, magnitudes and units. However, to keep the PSO generic, it is best to impose identical scale in each dimension, so that particle range is the same in each direction. This is achieved by normalising the parameters so that the particle position in each dimension is between -1 and +1. However, when evaluating the fitness of the particle, we need the renormalized values (i.e. the true values of the parameters).

Hence we introduce three additional attributes:

- norm – a list with the scaling factors for each dimension (η),
- shift – offset of the parameter range from 0 (σ),
- renormalized – the true value of the parameters (λ) represented by the particle.

The user must supply only the true range of the particle in the form of a tuple, per dimension, e.g. $(\lambda_{min}, \lambda_{max})$.

Then $(\lambda_{max} - \lambda_{min})\eta = 2.0$, or, $\eta = 2.0/(\lambda_2 - \lambda_1)$, and $\sigma = 0.5 * (\lambda_{max} + \lambda_{min})$.

So, the true particle position (for evaluations) is $\lambda = P/\eta + \sigma$, where P is the normalised position of the particle.

Using particles

Below, we have the declaration of the particle class and a couple of methods for creation and evolution of the particle based on the PSO algorithm.

Note that the evaluation of the fitness of the particle is problem specific and the user must supply its own evaluation function and register it under the name `evaluate` with the toolbox.

Particle Swarm

The swarm is a a list of particles, with a couple of additional attributes:

- `gbest` – globally the best particle (position) ever (i.e. accross any generation so far)
- `gbestfit` – globally the best fitness (i.e. the quality value of gbest)

The swarm is declared, created and let to evolve with the help of the `PSO` class.

class skpar.core.pso.**PSO**(*parameters*, *evaluate*, *npart=10*, *ngen=100*, *objective_weights=(-1,)*, *ErrTol=0.001*, **args*, ***kwargs*)

Bases: object

Class defining Particle-Swarm Optimizer.

halloffame = <deap.tools.support.HallOfFame object>

nBestKept = 10

optimise(*ngen=None*, *ErrTol=None*)

Let the swarm evolve for ngen (or self.ngen) generations.

pAcceleration = 2.9922

pInertia = 0.7298

report()

toolbox = <deap.base.Toolbox object>

skpar.core.pso.**createParticle**(*prange*, *strict_bounds=True*)

Create particle of dimensionality len(prange), assigning initial particle coordinate in the i-th dimension within the prange[i] tuple. Note that the range is normalised and shifted, so that the result is a coordinate within -1 to 1 for each dimension, and initial speed between -.5 and +.5. To get the true (i.e. physical) coordinates of the particle, one must use part.renormalized field. :param prange – list of tuples. each tuple is a range of _initial_ coord.:

Returns

particle – an instance of the Particle class, with initialized coordinates both normalized (the instance itself) and true, physical coords (self.renormalized).

skpar.core.pso.**declareTypes**(*weights*)

Declare Particle class with fitting objectives (-1: minimization; +1 maximization). Each particle consists of a set of normalised coordinates, returned by the particle's instance itself. The true (physical) coordinates of the particle are stored in the field 'renormalized'. The field 'best' contains the best fitness-wise position that the particle has ever had (in normalized coords). Declare also Swarm class; inherit from list + PSO specific attributes gbest and gbestfit. :param weights – tupple, e.g.: (+1,+0.5) for two objective maximization, etc. :type weights – tupple, e.g.: -1,

skpar.core.pso.**evolveParticle**(*part*, *best*, *inertia=0.7298*, *acceleration=2.9922*, *degree=2*)

A method to update the position and speed of a particle (part), according to the generalized formula of Eq(3) in J.Kennedy, "Particle Swarm Optimization" in "Encyclopedia of Machine Learning" (2010), which is equivalent to Eqs(3-4) of 'Particle swarm optimization: an overview'. Swarm Intelligence. 2007; 1: 33-57. :param * part – instance of the particle class, the particle to be updated: :param * best – the best known particle ever: :type * best – the best known particle ever: within the life of the swarm :param * inertia – factor scaling the persistence of the particle: :param * acceleration – factor scaling the influence of particle connection: :param * degree – unused right now; should serve for a fully informed particle swarm: :type * degree – unused right now; should serve for a fully informed particle swarm: FIPS :param but this requires best to become a list of neighbours best;: :param also u1,u2 and v_u1, v_u2 should be transformed into a Sum over neighbours:

Returns the updated particle

skpar.core.pso.**evolveParticle_0**(*part*, *best*, *phi1=2*, *phi2=2*)

This is the implementation shown in the examples of the DEAP library. The inertial factor is 1.0 (implicit) and seems to be too big. Phi1/2 are also somewhat bigger than the Psi/Ki resulting from the optimal Psi = 2.9922 A method to update the position and speed of a particle (part), according to the original PSO algorithm of Ricardo Poli, James Kennedy and Tim Blackwell, 'Particle swarm optimization: an overview'. Swarm Intelligence. 2007; 1: 33-57. :param part – instance of the particle class, the particle to be updated: :param best – the best known particle ever: :type best – the best known particle ever: within the life of the swarm :param phi1,phi2 – connectivity coefficients:

skpar.core.pso.**pformat**(*part*)
Return a formatted string for printing all particle info.

skpar.core.pso.**pso_args**(***kwargs*)

skpar.core.pso.**report_stats**(*stats*)

Utilities (skpar.core.utils)

skpar.core.utils.**arr2s**(*aa*, *precision=3*, *suppress_small=True*, *max_line_width=75*)
Helper for compact string representation of numpy arrays.

skpar.core.utils.**configure_logger**(*name*, *filename='skpar.log'*, *verbosity=20*)
Get parent logger: logging INFO on the console and DEBUG to file.

skpar.core.utils.**f2prange**(*rng*)
Convert fortran range definition to a python one.

Parameters **rng** (*2-sequence*) – [low, high] index range boundaries, inclusive, counting starts from 1.

Returns (low-1, high)

Return type 2-tuple

skpar.core.utils.**flatten**(*dd*)
Take a dictionary or list of dictionaries/lists dd, and produce a lists of values corresponding, dropping the keys.

skpar.core.utils.**flatten_two**(*d1*, *d2*)
Take two dictionaries or lists of dictionaries/lists d1 and d2, and produce two lists of values corresponding to the keys/order in d1. d2 is optional. Assume that the keys of d1 are a subset of the keys of d2. Assume nesting, i.e. some of the items in d1 are dictionaries and some are lists, numpy arrays, or a non-sequence type. The assumption is again: nested dictionaries in d2 must have at least the keys of the corresponding nested dictionaries in d1, and the lists in d1 must be no shorter than the lists in d2. . . . some assertions may help here. . .

skpar.core.utils.**get_logger**(*name*, *filename=None*, *verbosity=20*)
Return a named logger with file and console handlers.

Get a *name*-logger. Check if it is(has) a parent logger. If parent logger is not configured, configure it, and if a child logger is needed, return the child. The check for parent logger is based on *name*: a child if it contains '.', i.e. looking for 'parent.child' form of *name*. A parent logger is configured by defining a console handler at *verbosity* level, and a file handler at DEBUG level, writing to *filename*.

skpar.core.utils.**get_ranges**(*data*)
Return list of tuples ready to use as python ranges.

Parameters **data** (*int, list of int, list of lists of int*) – A single index, a list of indexes, or a list of 2-tuple range of indexes in Fortran convention, i.e. from low to high, counting from 1, and inclusive

Returns list of lists of 2-tuple ranges, in Python convention - from 0, exclusive.

skpar.core.utils.**is_monotonic**(*x*)

skpar.core.utils.**islistoflists**(*arg*)
Return True if item is a list of lists.

skpar.core.utils.**normalise**(*a*)
Normalise the given array so that sum of its elements yields 1.

Parameters **a** (*array*) – input array

Returns The norm is the sum of all elements across all dimensions.

Return type a/norm (array)

skpar.core.utils.**normaliseWeights**(*weights*)
normalise weights so that their sum evaluates to 1

2.6.2 dftbutils sub-package

Run BS (skpar.dftbutils.runBS)

Query DFTB (skpar.dftbutils.queryDFTB)

class skpar.dftbutils.queryDFTB.**BandsOut**(**args*, ***kwargs*)
A dictionary initialised with the bands from dp_bands or similar tool.

Useage:

destination_dict = BandsOut.fromfile(file)

classmethod fromfile(*fp*, *enumeration=True*)

class skpar.dftbutils.queryDFTB.**Bandstructure**(**args*, ***kwargs*)
A dictionary initialised with the bands and some analysis of the bands.

It requires two files: detailed.out from dftb+, and bands_tot.dat from dp_bands. It reads the bands via BandsOut; obtains the number of electrons via DetailedOut. It returns a dictionary with all that is in DetailedOut plus: 'bands': energy bands (excluding k-point enumeration) 'Ecb' : LUMO 'Evb' : HOMO 'Egap' : Ecb - Evb

Useage:

destination_dict = Bandstructure.fromfiles(detailed.out_file, bands_file)

classmethod fromfiles(*fp1*, *fp2*, *enumeration=True*)
Read the output of dftb+ and dp_bands and return a dictionary with band-structure data.

class skpar.dftbutils.queryDFTB.**DetailedOut**(**args*, ***kwargs*)
A dictionary initialised from file with the detailed output of dftb+.

Useage:

destination_dict = DetailedOut.fromfile(filename)

conv_tags = [('iSCC', ('nscc', 'scc_err')), ('SCC converged', True), ('SCC is NOT conv

energy_tags = [('Fermi energy:', 'Ef'), ('Fermi level:', 'Ef'), ('Band energy:', 'E

classmethod fromfile(*fp*)

nelec_tags = [('Input/Output electrons (q):', ('nei', 'neo')), ('Input / Output elect

skpar.dftbutils.queryDFTB.**calc_masseff**(*bands*, *extrtype*, *kLineEnds*, *lattice*, *meff_tag=None*, *Erange=0.008*, *forceErange=False*, *ib0=0*, *nb=1*, *usebandindex=False*, ***kwargs*)
A complex wrapper around meff(), with higher level interface.

Calculate parabolic effective mass at the specified *extrtype* of given *bands*, calculated along two points in k-space defined by a list of two 3-tuples - *kLineEnds*. *lattice* is a lattice object, defining the metric of the kspace.

Parameters

- **bands** – an array (nb, nk) energy values in [eV], or a 1D array like
- **extrtype** – type of extremum to search for: 'min' or 'max', handled by np.min()/max()

- **kLineEnds** – two 3-tuples, defining the coordinates of the endpoints of the k-line along which *band* is obtained, in terms of k-scace unit vectors, e.g. if *band* is obtained along a number of points from Gamma to X, of the BZ of a cubic lattice, then kLineEnds should read ((0, 0, 0), (1, 0, 0))
- **lattice** – lattice object, holding mapping to kspace.
- **meff_name** – the name to be featured in the logger
- **Erange** – Energy range [eV] over which to fit the parabola [dflt=8meV], i.e. 'depth' of the assumed parabolic well.

Return meff the value of the parabolic effective mass [m_0] at the *extrtype* of the given E-kline, if the extremum is not at the boundary of the given k-line.

skpar.dftbutils.queryDFTB.**expand_meffdata**(*meff_data*)

skpar.dftbutils.queryDFTB.**get_Ek**(*bsdata*, *sympts*)

skpar.dftbutils.queryDFTB.**get_bandstructure**(*implargs*, *database*, *source*, *model*, *detailfile='detailed.out'*, *bandsfile='bands_tot.dat'*, *hsdfile='dftb_pin.hsd'*, *latticeinfo=None*, **args*, ***kwargs*)

Get bandstructure and related data from dftb+.

Assume that *source* is the execution directory where *detailed.out*, and *bands_tot.dat* can be found. Additionally, the parsed input of dftb+ – *dftb_pin.hsd* is also checked if lattice info is given, in order to analyse k-paths and provide data for subsequent plotting.

skpar.dftbutils.queryDFTB.**get_dftbp_data**(*implargs*, *database*, *source*, *model*, *datafile='detailed.out'*)

Get whatever data can be obtained from detailed.out of dftb+.

Assume *source* is the directory where dftb+ was executed and that *datafile* is the detailed output file, along with *dftb_pin.hsd*, etc.

Parameters

- **implargs** (*dict*) – implicit key-word arguments passed by caller
- **database** (*obj*) – a database object that has a .update(dict) method
- **source** (*str*) – directory name where dftb+ has been executed
- **model** (*str*) – name of the model whose data is updated
- **datafile** (*str*) – base-name of the detailed output from dftb+

skpar.dftbutils.queryDFTB.**get_dftbp_evol**(*implargs*, *database*, *source*, *model*, *datafile='detailed.out'*, **args*, ***kwargs*)

Get the data from DFTB+ SCC calculation for all models.

This is a compound task that augments the source path to include individual local directories for different cell volumes, based on the assumption that these directories are named by 3 digits. Similar assumption applies for the model, where the name of the base model is augmented by the 3-digit directory number.

Parameters

- **workroot** (*string*) – base directory where model directories are found.
- **source** (*string*) – model directory
- **model** (*str*) – name of the model whose data is updated
- **datafile** (*string*) – optional filename holding the data.

skpar.dftbutils.queryDFTB.**get_effmasses**(*implargs*, *database*, *source*, *model=None*, *directions=None*, *carriers='both'*, *nb=1*, *Erange=0.04*, *usebandindex=False*, *forceErange=False*, **args*, ***kwargs*)

Get effective masses along select directions for select carrier types.

Obtain the effective masses for the given *carriers* for the first *nb* bands in the VB and/or CB, along the given *directions*, as well as the values of the extrema and their position along these *directions*. Label the effective masses by band index (starting from 0, within the band for the select carrier type), if *usebandindex* is True. Carrier types (*carriers*) could be 'e', 'h', 'both'. *Erange* is the energy range over which parabolic expansion is attempted

skpar.dftbutils.queryDFTB.**get_labels**(*ss*)

Return two labels from a string containing "-" or two words starting with a capital.

For example, the input string may be 'G-X', 'GX', 'Gamma-X', 'GammaX'. The output is always: ('G', 'X') or ('Gamma', 'X').

skpar.dftbutils.queryDFTB.**get_special_Ek**(*implargs, database, source, model=None, sympts=None, extract={'cb': [0], 'vb': [0]}, align='Ef', usebandindex=True, *args, **kwargs*)

Query bandstructure data and yield the eigenvalues at k-points of high-symmetry.

skpar.dftbutils.queryDFTB.**greek**(*label*)

Change Greek letter names to single Latin capitals, and vice versa.

Useful for some names of high-symmetry points inside the BZ, to shorten the names of Gamma, Sigma and Delta. Note that Lambda cannot be made into L, as it will make automatic L to Lambda as well, which is wrong since L is a standard point on the BZ surface.

skpar.dftbutils.queryDFTB.**is_monotonic**(*x*)

Return True if x is monotonic (either never increases or never decreases); False otherwise.

skpar.dftbutils.queryDFTB.**meff**(*band*, *kline*)

Return the effective mass, in units of *m_0*, given a band a *k*-line.

The mass is calculated as as the inverse of the curvature of *bands*, assuming parabolic dispersion within *kline*, working in atomic units: *bands* and *kline* are in Hartree and 1/Bohr, h_bar = 1, m_0 = 1

meff = (h_bar**2) / (d**2E/dk**2), [m0]

skpar.dftbutils.queryDFTB.**plot_fitmeff**(*ax*, *xx*, *x0*, *extremum*, *mass*, *dklen=None*, *ix0=None*, **args*, ***kwargs*)

Plot the second order polynomial fitted to E(k) dispersion on top of *ax* axes of a matplotlib figure object. *mass* is the fitted effective mass *extremum* is extremal energy, E0 *x0* is the relative position of the extremum along the given kline *xx*.

Assumed is that around the extremum at k0:

E"(k) = 1/mass => E(k) = E(x) = c2*x^2 + c1*x + c0.

Since E"(x) = 2*c2 => c2 = 1/(2*mass). Since E'(x) = 2*c2*x + c1, and E'(x=x0) = 0 and E(x=x0) = E0 => knowing E0 and x0, we can obtain c1 and c2:

c1 = -2*c2*x0 c0 = E0 - c2*x0^2 - c1*x0

Query k-Lines (skpar.dftbutils.querykLines)

Module for k-Lines extraction and k-Label manipulation

Recall that bands contains NO information of the k-points. So we must provide that ourselves, and reading the dftb_pin.hsd (the parsed input) seems the best way to do so. We also need to figure out what to do with equivalent points, or points where a new path starts. Finally, points internal to the Brilloin Zone are labeled with Greek letters, which should be rendered properly.

skpar.dftbutils.querykLines.**get_klines**(*lattice*, *hsdfile='dftb_pin.hsd'*, *workdir=None*, **args*, ***kwargs*)

This routine analyses the KPointsAndWeights stanza in the input file of DFTB+ (given as an input argument *hsdfile*), and returns the k-path, based on the lattice object (given as an input argument *lattice*). If the file name is not provided, the routine looks in the default dftb_pin.hsd, i.e. in the parsed file!

The routine returns a list of tuples (kLines) and a dictionary (kLinesDict) with the symmetry points and indexes of the corresponding k-point in the output band-structure.

kLines is ordered, as per the appearence of symmetry points in the hsd input, e.g.:

- [('L', 0), ('Γ', 50), ('X', 110), ('U', 130), ('K', 131), ('Γ', 181)]

therefore it may contain repetitions (e.g. for 'Γ', in this case).

kLinesDict returns a dictionary of lists, so that there's a single entry for non-repetitive k-points, and more than one entries in the list of repetitive symmetry k-points, e.g. (see for 'Γ' above):

- {'X': [110], 'K': [131], 'U': [130], 'L': [0], 'Γ': [50, 181]}

skpar.dftbutils.querykLines.**get_kvec_abscissa**(*lat*, *kLines*)

Return abscissa values for the reciprocal lengths corresponding to the k-vectors derived from kLines.

skpar.dftbutils.querykLines.**greekLabels**(*kLines*)

Check if Γ is within the kLines and set the label to its latex formulation. Note that the routine will accept either list of tupples ('label',int_index) or a list of strings, i.e. either kLines or only the kLinesLabels. Could do check for other k-points with greek lables, byond Γ (i.e. points that are inside the BZ, not at the faces) but in the future.

Lattice (skpar.dftbutils.lattice)

This module lists the component vectors of the direct and reciprocal lattices for different crystals. It is based on the following publication: Wahyu Setyawan and Stefano Curtarolo, "High-throughput electronic band structure calculations: Challenges and tools", Comp. Mat. Sci. 49 (2010), pp. 291–312.

class skpar.dftbutils.lattice.**BCC**(*param*, *setting='curtarolo'*)

This is Body Centered Cubic lattice (cF)

class skpar.dftbutils.lattice.**CUB**(*param*, *setting=None*)

This is CUBic, cP lattice

class skpar.dftbutils.lattice.**FCC**(*param*, *setting='curtarolo'*)

This is Face Centered Cubic lattice (cF)

class skpar.dftbutils.lattice.**HEX**(*param*, *setting='curtarolo'*)

This is HEXAGONAL, hP lattice

class skpar.dftbutils.lattice.**Lattice**(*info*)

Generic lattice class.

get_kcomp(*string*)

Return the k-components given a string label or string set of fraction.

Example of string:

s = 'X' s = '1/2 0 1/2' s = '1/2, 0, 1/2' s = '0.5 0 0.5'

get_kvec(*kpt*)
Return the real space vector corresponding to a k-point.

class skpar.dftbutils.lattice.**MCL**(*param*, *setting='curtarolo'*)
This is simple Monoclinic MCL_* (mP) lattice, set via a, b <= c, and alpha < 90 degrees, beta = gamma = 90 degrees as in W. Setyawan, S. Curtarolo / Computational Materials Science 49 (2010) 299-312. Setting=ITC should work for the standard setting (angle>90) of ITC-A, but is not currently implemented. Note that conventional and primitive cells are the same.

class skpar.dftbutils.lattice.**MCLC**(*param*, *setting='ITC'*)
This is base-centered Monoclinic MCLC_* mS, lattice Primitive lattice defined via: a <> b <> c, and alpha <> 90 degrees, beta = gamma = 90 degrees

class skpar.dftbutils.lattice.**ORC**(*param*, *setting='curtarolo'*)
This is ORTHOROMBIC, oP lattice

class skpar.dftbutils.lattice.**RHL**(*param*, *setting=None*)
This is Rhombohedral RHL, hR lattice Primitive lattice defined via: a = b = c, and alpha = beta = gamma <> 90 degrees Two variations exists: RHL1 (alpha < 90) and RHL2 (alpha > 90)

class skpar.dftbutils.lattice.**TET**(*param*, *setting='curtarolo'*)
This is TETRAGONAL, tP lattice

skpar.dftbutils.lattice.**getSymPtLabel**(*kvec*, *lattice*)
Return the symbol corresponding to a given k-vector, if named.

This routine returns the symbol of a symmetry point that is given in terms of reciprocal cell-vectors (*kvec* – a 3-tuple) of the *lattice* object.

skpar.dftbutils.lattice.**get_dftbp_klines**(*lattice*, *delta=None*, *path=None*)
Print out the number of points along each segment of the BZ *path* (default for the lattice chosen if None) of a given *lattice* object

skpar.dftbutils.lattice.**get_kvec**(*comp_rc*, *recipr_cell*)
comp_rc are the components of a vector expressed in terms of reciprocal cell vectors *recipr_cell*. Return the components of this vector in terms of reciprocal unit vectors.

skpar.dftbutils.lattice.**get_recipr_cell**(*A*, *scale*)
Given a set of set of three vectors *A*, assumed to be that defining the primitive cell, return the corresponding set of vectors that define the reciprocal cell, *B*, scaled by the input parameter *scale*, which defaults to 2pi. The B-vectors are computed as follows: B0 = scale * (A1 x A2)/(A0 . A1 x A2) B1 = scale * (A2 x A0)/(A0 . A1 x A2) B2 = scale * (A0 x A1)/(A0 . A1 x A2) and are returnd as a list of 1D arrays. Recall that the triple-scalar product is invariant under circular shift, and equals the (signed) volume of the primitive cell.

skpar.dftbutils.lattice.**getkLineLength**(*kpt0*, *kpt1*, *Bvec*, *scale*)
Given two k-points in terms of unit vectors of the reciprocal lattice, Bvec, return the distance between the two points, in terms of reciprocal length.

skpar.dftbutils.lattice.**len_pathsegments**(*lattice*, *scale=None*, *path=None*)
Report the lenth in terms of _scale_ (2pi/a if None) of the BZ _path_ (default for the lattice chosen if None) of a given _lattice_ object

skpar.dftbutils.lattice.**repr_lattice**(*lat*)
Report cell vectors, reciprocal vectors and standard path

Plot (skpar.dftbutils.plot)

skpar.dftbutils.plot.**magic_plot_bs**(*xval*, *yval*, *filename=None*, ***kwargs*)
A magic-wrapper around the fundamental back-end plot_bs function.

The magic is that if yval is a list of [Egap, VBand, CBand,...] the data is modified so that a band-gap, Egap, is open between cband and vband, even if they are not properly aligned, e.g. if CB bottom is 0 at the same time as VB top is 0. Note that the CB is moved, not the VB. NOTABENE: the order must be Egap, VB, CB!

We do this here, so that we don't burden the PlotTask elsewhere with knowledge of what band structure and band-gap is, and keep it independent of what it is plotting. However, somewhere, the gap need to be opened, if we've specified CB and VB as independent objectives, and certainly the band-end plot_bs is not such a place due to its intended generality (of plotting band-structures unrelated to objectives, optimisation etc.).

The magic happens only if yval contains an array shaped (1,), which is taken as a band-gap. If no such array is discovered, no shifts are applied to the bands.

Parameters

- **filename** (*str*) – valid filename to save the plot
- **xval** (*arr*) – k-points (1D array or a list of values and 1D arrays)
- **yval** (*arr*) – bands (2D-array or a list values and 2D arrays)

Kwargs: Check plot_bs for details as kwargs are passed directly to it.

Returns matplotlib figure and axes objects containing the plot

Return type fig, ax

skpar.dftbutils.plot.**plot_bs**(*xx*, *yy*, *colors=None*, *linelabels=None*, *title=None*, *figsize=(6, 7)*, *xticklabels=None*, *yticklabels=None*, *xlim=None*, *ylim=None*, *xlabel=None*, *ylabel='Energy (eV)'*, *filename=None*, *legendloc=0*, ***kwargs*)

Routine for plotting band-structure.

Accepts one or more sets of k-vector and corresponding bands, but the k-vector may be shared too. If you supply a set of ticks and labels for specific k-points, it will put them on X axis and will extend the xticks over all Y as thin lines; see xticklabels below.

Parameters

- **xx** – 1D array or a list of such; k-points.shape = nk
- **yy** – 2D array or a list of such; bands.shape = nk, nE Notabene: each band is a column in its respective array

  so that the lowest band is leftmost.
- **colors** – list of colors, one per 2D array of bands; if None, default matplotlib Vega/D3 set of colours is used.
- **linelabels** – list of strings to label each set of bands in legend
- **title** – figure title
- **figsize** – tupple for figure dimensions, in inches; defaults to (6,7)
- **ylim** (*xlim,*) – tupple of limits for X-axis, or Y-axis
- **ylabel** (*xlabel,*) – axes labels
- **yticklabels** (*xticklabels,*) – a list of explicit X- or Y-axis ticks and labels, e.g. [('label',x), ...]
- **filename** (*str*) – filename (incl directory as needed); if present the figure is saved to that file.

Kwargs: kticklabels: interpreted as xticklabels eticklabels: interpreted as yticklabels No other kwargs are interpreted, but no exception is generated if supplied.

Returns matplotlib objects holding the plot

Return type fig, ax

skpar.dftbutils.plot.**set_axes**(*ax*, *xlabel*, *ylabel*, *xticklabels=None*, *yticklabels=None*, *extend_xticks=False*, *extend_yticks=False*)

Configure axes – labels and ticks/ticklabels.

Parameters

- **ax** – matplotlib axis object
- **ylabel** (*xlabel,*) – labels for the x and y axis
- **yticklabels** (*xticklabels,*) – list of [(value, 'label'),] for each explicit position of ticks and their labels
- **extend_yticks** (*extend_xticks,*) – extend_x/yticks entire graph

skpar.dftbutils.plot.**set_mplrcpar**(***kwargs*)

Configure matplotlib rcParams.

skpar.dftbutils.plot.**set_xylimits**(*ax*, *xval*, *yval*, *xlim=None*, *ylim=None*, *issetx=False*, *issety=False*)

Set x and y axis limits to exactly fit the data if not explicit.

ax: matplotlib axis object xval, yval: array (could be record array), lists of arrays xlim, ylim: tupple of (min,max) - explicit axis limits issetx, issety (bool): used if xlim or ylim is None, in which case

these flags serve to tell us to find min and max of the xval and yval even if these are record arrays where broadcasting won't work. (e.g. yval is an array of two 1D arrays of different shape)

Unils (skpar.dftbutils.utils)

General utility functions

skpar.dftbutils.utils.**configure_logger**(*name*, *filename=None*, *verbosity=20*)

Get parent logger: logging INFO on the console and DEBUG to file.

skpar.dftbutils.utils.**execute**(*cmd*, *workdir='.'*, *outfile='run.log'*, *purge_workdir=False*, ***kwargs*)

Execute external command in workdir, streaming output/error to outfile.

Parameters

- **cmd** (*str*) – command; executed in *workir*; if it contains *$* or *-globbing, these are shell-expanded
- **workdir** (*path-like*) – execution directory relative to workroot
- **outfile** (*str*) – output file for the stdout/stderr stream; continuously updated during execution
- **purge_workdir** (*bool*) – if true, any existing working directory is purged
- **kwargs** (*dict*) – passed directly to the underlying *subprocess.call()*

Returns None

Raises

- `OSError` – if *cmd* cannot be executed
- `RuntimeError` – if *cmd* returncode is nonzero
- `SubprocessError` – other possible circumstances

`skpar.dftbutils.utils.`**`get_logger`**(*name*, *filename=None*, *verbosity=20*)
Return a named logger with file and console handlers.

Get a *name*-logger. Check if it is(has) a parent logger. If parent logger is not configured, configure it, and if a child logger is needed, return the child. The check for parent logger is based on *name*: a child if it contains '.', i.e. looking for 'parent.child' form of *name*. A parent logger is configured by defining a console handler at *verbosity* level, and a file handler at DEBUG level, writing to *filename*.

`skpar.dftbutils.utils.`**`parse_cmd`**(*cmd*)
Parse shell command for globbing and environment variables.

2.7 License

SKPAR is distributed under The MIT License.

2.8 Development

Further development of SKPAR is along the following lines.

2.8.1 External Model and Reference Database

The aims is to completely decouple the core of the optimisation engine from the application specifics. To achieve this:

- the internal model database dictionary must be taken out of the core sub-package and queries must be made to address an external database;
- the task handling has to be done by an external task manager, which can be application specific, and its setup may be still driven by the input YAML file of SKPAR;
- an external task-manager is needed to wrap the executables that yield model data and make them store that data in the external model database;
- support for reference databases should complement the current mechanism through which the declaration of objectives is realised.

A conceptual block-diagram of the intended development is shown below.

Implementation should be straightforward, e.g. by deploying TinyDB or similar.

As far as task-manager is concerned, `dftbutils bands` already represents an example in this direction, short of putting data in a database.

2.8.2 Parallelisation

The calculations done within SKPAR consume negligible time in comparison to the evaluation of the model. The executables embodying the model are computationally very intense but are typically parallelised. However, much gain may be obtained by parallelising the evaluation of individuals within the population (e.g. per particle, when PSO algorithm is used).

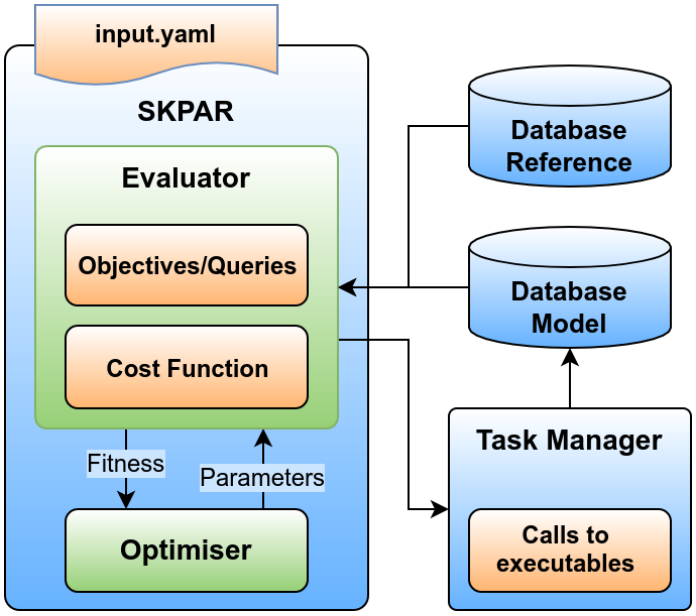

Fig. 8: **Conceptual block diagram of SKPAR with external model database**

2.9 Contributors

Stanislav Markov, Dept. Chemistry, The University of Hong Kong

Bálint Aradi, BCCMS, University of Bremen, Germany

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[MOO-review] R.T. Marler and J.S. Arora, Struct Multidisc Optim 26, 369-395 (2004), "Survey of multi-objective optimization methods for engineering"

[PSO-1] J.Kennedy, "Particle Swarm Optimization" in "Encyclopedia of Machine Learning" (2010),

[PSO-2] 'Particle swarm optimization: an overview'. Swarm Intelligence. 2007; 1: 33-57.

Python Module Index

S

skpar.core.evaluate, 39
skpar.core.input, 33
skpar.core.objectives, 35
skpar.core.optimise, 40
skpar.core.parameters, 40
skpar.core.pso, 42
skpar.core.skpar, 33
skpar.core.taskdict, 33
skpar.core.tasks, 33
skpar.core.utils, 44
skpar.dftbutils.lattice, 48
skpar.dftbutils.plot, 49
skpar.dftbutils.queryDFTB, 45
skpar.dftbutils.querykLines, 47
skpar.dftbutils.utils, 51

Index

A

about, 5
abserr() (*in module skpar.core.evaluate*), 39
arr2s() (*in module skpar.core.utils*), 44

B

BandsOut (*class in skpar.dftbutils.queryDFTB*), 45
Bandstructure (*class in skpar.dftbutils.queryDFTB*), 45
BCC (*class in skpar.dftbutils.lattice*), 48

C

calc_masseff() (*in module skpar.dftbutils.queryDFTB*), 45
check_taskdict() (*in module skpar.core.tasks*), 33
commands, 8
config, 32
configure_logger() (*in module skpar.core.utils*), 44
configure_logger() (*in module skpar.dftbutils.utils*), 51
contributors, 52
conv_tags (*skpar.dftbutils.queryDFTB.DetailedOut attribute*), 45
core, 32
cost_rms() (*in module skpar.core.evaluate*), 39
create_workdir() (*in module skpar.core.evaluate*), 40
createParticle() (*in module skpar.core.pso*), 43
CUB (*class in skpar.dftbutils.lattice*), 48

D

declareTypes() (*in module skpar.core.pso*), 43
destroy_workdir() (*in module skpar.core.evaluate*), 40
DetailedOut (*class in skpar.dftbutils.queryDFTB*), 45
develop, 52
dftbutils, 45

E

energy_tags (*skpar.dftbutils.queryDFTB.DetailedOut attribute*), 45
eval_objectives() (*in module skpar.core.evaluate*), 40
evaluate() (*skpar.core.evaluate.Evaluator method*), 39
evaluate() (*skpar.core.objectives.Objective method*), 36
Evaluator (*class in skpar.core.evaluate*), 39
evolveParticle() (*in module skpar.core.pso*), 43
evolveParticle_0() (*in module skpar.core.pso*), 43
executables, 31
execute() (*in module skpar.core.taskdict*), 34
execute() (*in module skpar.dftbutils.utils*), 51
expand_meffdata() (*in module skpar.dftbutils.queryDFTB*), 46

F

f2prange() (*in module skpar.core.utils*), 44
FCC (*class in skpar.dftbutils.lattice*), 48
flatten() (*in module skpar.core.utils*), 44
flatten_two() (*in module skpar.core.utils*), 44
fromfile() (*skpar.dftbutils.queryDFTB.BandsOut class method*), 45
fromfile() (*skpar.dftbutils.queryDFTB.DetailedOut class method*), 45
fromfiles() (*skpar.dftbutils.queryDFTB.Bandstructure class method*), 45

G

get() (*skpar.core.objectives.ObjBands method*), 35
get() (*skpar.core.objectives.Objective method*), 36
get() (*skpar.core.objectives.ObjKeyValuePairs method*), 35
get() (*skpar.core.objectives.ObjValues method*), 35
get() (*skpar.core.objectives.ObjWeightedSum method*), 35

get_bandstructure() (*in module skpar.dftbutils.queryDFTB*), 46
get_config() (*in module skpar.core.input*), 33
get_dftbp_data() (*in module skpar.dftbutils.queryDFTB*), 46
get_dftbp_evol() (*in module skpar.dftbutils.queryDFTB*), 46
get_dftbp_klines() (*in module skpar.dftbutils.lattice*), 49
get_effmasses() (*in module skpar.dftbutils.queryDFTB*), 46
get_Ek() (*in module skpar.dftbutils.queryDFTB*), 46
get_input() (*in module skpar.core.input*), 33
get_kcomp() (*skpar.dftbutils.lattice.Lattice method*), 48
get_klines() (*in module skpar.dftbutils.querykLines*), 48
get_kvec() (*in module skpar.dftbutils.lattice*), 49
get_kvec() (*skpar.dftbutils.lattice.Lattice method*), 48
get_kvec_abscissa() (*in module skpar.dftbutils.querykLines*), 48
get_labels() (*in module skpar.dftbutils.queryDFTB*), 47
get_logger() (*in module skpar.core.utils*), 44
get_logger() (*in module skpar.dftbutils.utils*), 52
get_model_data() (*in module skpar.core.taskdict*), 34
get_models() (*in module skpar.core.objectives*), 36
get_objective() (*in module skpar.core.objectives*), 36
get_optargs() (*in module skpar.core.optimise*), 40
get_parameters() (*in module skpar.core.parameters*), 41
get_ranges() (*in module skpar.core.utils*), 44
get_recipr_cell() (*in module skpar.dftbutils.lattice*), 49
get_refdata() (*in module skpar.core.objectives*), 36
get_refval() (*in module skpar.core.objectives*), 37
get_refval_1d() (*in module skpar.core.objectives*), 37
get_special_Ek() (*in module skpar.dftbutils.queryDFTB*), 47
get_subset_ind() (*in module skpar.core.objectives*), 37
get_tasklist() (*in module skpar.core.tasks*), 33
get_type() (*in module skpar.core.objectives*), 37
get_workdir() (*in module skpar.core.evaluate*), 40
getkLineLength() (*in module skpar.dftbutils.lattice*), 49
getSymPtLabel() (*in module skpar.dftbutils.lattice*), 49
greek() (*in module skpar.dftbutils.queryDFTB*), 47
greekLabels() (*in module skpar.dftbutils.querykLines*), 48

H

halloffame (*skpar.core.pso.PSO attribute*), 43
HEX (*class in skpar.dftbutils.lattice*), 48

I

initialise_tasks() (*in module skpar.core.tasks*), 33
install, 7
is_monotonic() (*in module skpar.core.utils*), 44
is_monotonic() (*in module skpar.dftbutils.queryDFTB*), 47
islistoflists() (*in module skpar.core.utils*), 44

L

Lattice (*class in skpar.dftbutils.lattice*), 48
len_pathsegments() (*in module skpar.dftbutils.lattice*), 49
license, 52

M

magic_plot_bs() (*in module skpar.dftbutils.plot*), 49
MCL (*class in skpar.dftbutils.lattice*), 49
MCLC (*class in skpar.dftbutils.lattice*), 49
meff() (*in module skpar.dftbutils.queryDFTB*), 47

N

nBestKept (*skpar.core.pso.PSO attribute*), 43
nelec_tags (*skpar.dftbutils.queryDFTB.DetailedOut attribute*), 45
normalise() (*in module skpar.core.utils*), 44
normaliseWeights() (*in module skpar.core.utils*), 45

O

ObjBands (*class in skpar.core.objectives*), 35
Objective (*class in skpar.core.objectives*), 35
objectives, 22
ObjKeyValuePairs (*class in skpar.core.objectives*), 35
ObjValues (*class in skpar.core.objectives*), 35
ObjWeightedSum (*class in skpar.core.objectives*), 35
optimisation, 30
optimise() (*skpar.core.pso.PSO method*), 43
Optimiser (*class in skpar.core.optimise*), 40
ORC (*class in skpar.dftbutils.lattice*), 49

P

pAcceleration (*skpar.core.pso.PSO attribute*), 43
Parameter (*class in skpar.core.parameters*), 41
parse_cmd() (*in module skpar.core.taskdict*), 35
parse_cmd() (*in module skpar.dftbutils.utils*), 52
parse_input() (*in module skpar.core.input*), 33

parse_weights() (*in module skpar.core.objectives*), 37
parse_weights_keyval() (*in module skpar.core.objectives*), 38
pformat() (*in module skpar.core.pso*), 44
pick_objectives() (*skpar.core.taskdict.PlotTask method*), 34
pInertia (*skpar.core.pso.PSO attribute*), 43
plot_bs() (*in module skpar.dftbutils.plot*), 50
plot_fitmeff() (*in module skpar.dftbutils.queryDFTB*), 47
PlotTask (*class in skpar.core.taskdict*), 33
prepare_for_plotsave() (*in module skpar.core.taskdict*), 35
PSO (*class in skpar.core.pso*), 42
pso_args() (*in module skpar.core.pso*), 44

R

reference, 16
relerr() (*in module skpar.core.evaluate*), 40
report() (*skpar.core.optimise.Optimiser method*), 40
report() (*skpar.core.pso.PSO method*), 43
report_stats() (*in module skpar.core.pso*), 44
repr_lattice() (*in module skpar.dftbutils.lattice*), 49
RHL (*class in skpar.dftbutils.lattice*), 49

S

set_axes() (*in module skpar.dftbutils.plot*), 51
set_mplrcpar() (*in module skpar.dftbutils.plot*), 51
set_objectives() (*in module skpar.core.objectives*), 39
set_xylimits() (*in module skpar.dftbutils.plot*), 51
SKPAR (*class in skpar.core.skpar*), 33
skpar.core.evaluate (*module*), 39
skpar.core.input (*module*), 33
skpar.core.objectives (*module*), 35
skpar.core.optimise (*module*), 40
skpar.core.parameters (*module*), 40
skpar.core.pso (*module*), 42
skpar.core.skpar (*module*), 33
skpar.core.taskdict (*module*), 33
skpar.core.tasks (*module*), 33
skpar.core.utils (*module*), 44
skpar.dftbutils.lattice (*module*), 48
skpar.dftbutils.plot (*module*), 49
skpar.dftbutils.queryDFTB (*module*), 45
skpar.dftbutils.querykLines (*module*), 47
skpar.dftbutils.utils (*module*), 51
subpackages and modules, 32
substitute_parameters() (*in module skpar.core.taskdict*), 35
substitute_template() (*in module skpar.core.parameters*), 41
summarise() (*skpar.core.objectives.Objective method*), 36

T

Task (*class in skpar.core.tasks*), 33
tasks, 16
TET (*class in skpar.dftbutils.lattice*), 49
toolbox (*skpar.core.pso.PSO attribute*), 43
tutorials, 9
typedict (*skpar.core.parameters.Parameter attribute*), 41

U

update_parameters() (*in module skpar.core.parameters*), 41
update_template() (*in module skpar.core.parameters*), 41

W

wrapper_PlotTask() (*in module skpar.core.taskdict*), 35