BOPcat Documentation

Release 1.0

Alvin Noe Ladines

Sep 26, 2018

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 About

The **BOPcat** package is a software written in Python to generate new tight-binding or bond-order potentials or optimize existing models using the **BOPfox** code. The parameters of the models are determined by reproducing various target properties including energies, forces, stresses, defect formation energies, elastic constants, etc. The structures and their properties are taken from a DFT database but can also include experiments or other data sources. It employs the optimization libraries of **Scipy** but external optimization modules can also be used. The structures and their properties are handled using the **ASE** Atoms object. These are read from a text file which are generated from an external database.

1.2 Installation

Prior to using the code, BOPfox should be compiled and the path to the executable should be included in `$PATH`. As some BOPfox source files are read to generate some variables, it is recommended to keep the executable in the `bopfox/src` folder. BOPcat utilizes a number of python modules such as numpy (http://www.numpy.org/), ASE (https://wiki.fysik.dtu.dk/ase/), Scipy (http://www.scipy.org/), matplotlib (http://www.matplotlib.org/) and pyspglib (https://atztogo.github.io/spglib/). BOPcat depends on the the BOPfox-ASE interface to calculate the properties of the structure which is not yet included in the original ASE module. The relevant files are included in the BOPcat source files. The files `bopio.py` and `bopcal.py` should be copied to the `ase/io` and `ase/calculators` folders respectively. Both files should be renamed `bopfox.py`. In order to know the path to your ase libraries, simpy execute the following in python:

```
import ase
ase.__file__
```

In case you do not have permission to make changes to the ase folder, it is necessary to install a local version of ASE (https://wiki.fysik.dtu.dk/ase/). A more straightforward way is to install **Anaconda** (https://www.continuum.io/downloads). The latter is recommended as this will also update and consolidate all your python modules in a local directory.

Note: It may be necessary to restart your computer.

To make sure that you have set the correct path to `ase`, execute the following in python:

```
from ase.calculators import bopfox as bopcal
from ase.io import bopfox as bopio
```

To install `BOPcat`, run the installation script

```
python setup.py install
```

Alternatively, one simply specify the path to the BOPcat source files, i.e. you should append the following to your `.bashrc`:

```
export PYTHONPATH=<path to bopcat>:/bopcat:$PYTHONPATH          export
PYTHONPATH=<path to bopcat>:$PYTHONPATH
```

These make it possible to execute:

```
from bopcat import variables
import variables
```

To test if the required paths are set correctly, execute

```
python test_install.py path
```

1.3 Examples

The examples to test the basic functionalities of the code are found in `examples/`. To get started, run the examples in the following order:

1. **`ASE`** The usage of the BOPfox ASE interface is illustrated. See example1.py and example2.py
2. **`strucscan`** BOPcat reference data are constructed from strucscan. See example.py
3. **`optimize_Fe-Madsen-2011`** An existing Fe model is optimized. See input2.py and main2.py
4. **`optimize_Re-Cak-2014`** An existing W model is optimized for Re. See input3.py and main3.py
5. **`construct_Fe`** A new Fe model is constructed. See input4.py and main4.py
6. **`construct_FeNb`** A new FeNb model is constructed. See input5.py and main5.py
7. **`test_Fe_Madsen-2011`** BOPcat utilities are illustrated. See input6.py and main6.py

BOPcat is a collection of tools for the optimizing models. It is necessary for the user to write a script to specify the procedure. In addition, the input controls should also be provided which are handled by the `cat_controls` object. Assuming that the main script is `main.py` and the input file is `input.py`, the scripts should be executed as

```
python main.py input.py
```

The basic form of the script is as follows:

1. Execute and initialize the input controls. These are then attributes of the `cat_controls` object:

```
execfile(sys.argv[-1])
cat_contols.initialize()
```

2. Generate reference data. The `cat_data` object essentially reads the text file of structures and their properties (see *Reference structures database*):

```
cat_data = CATData(controls=cat_controls)
```

3. Generate or read initial model. The `cat_param` object can also be used to store the resulting models at each level of optimization:

```
cat_param = CATParam(controls=cat_controls,data=cat_data)
```

4. Set up the calculator. To set up the calculator, one provides the input controls and a model, in the following, we use the last model (`models[-1]`) saved in `cat_param`

```
cat_calc = CATCalc(controls=cat_controls,model=cat_param.models[-1])
```

5. To proceed with the optimization, one needs to determine which structures are included in the target set. These are then assigned to the calculator:

```
ref_atoms = cat_data.get_ref_atoms(structures=['Fe/229/0/1/*'],quantities=['energy'])
cat_calc.set_atoms(ref_atoms)
ref_data = cat_data.get_ref_data()
```

6. The optimization kernel is also necessary which takes in the calculator, the array of reference data, the constraints on variables, the weights and the input controls:

```
var  = [{"bond":["Fe","Fe"],'atom':'Fe','onsitelevels':[True]}]
optfunc = CATkernel(calc=cat_calc,ref_data=ref_data,variables=var,log='log.cat'
                    ,controls=cat_controls)
```

7. The optimization is run and the resulting model is saved in `cat_param`:

```
optfunc.optimize()
new_model = optfunc.get_optimized_model()
cat_param.models.append(new_model)
```

This can be done iteratively for different sets of target structures, constraints, starting parameters or even a completely different functional form.

CHAPTER

TWO

REFERENCE STRUCTURES DATABASE

BOPcat requires a list of structures each is an `ase.Atoms` object. These are extracted from a generic database which takes in general identifiers depending on the data type. Currently, only `dft` data are handled in but can be extended to include data from experiments or other calculation schemes. The `Atoms` also contain the calculation parameters, details of the crystal structure and various properties that are required for fitting. The data are extracted from the database once and are written in a readable text format for future use.

General DFT identifiers

key	val	description	val	description
code	0	abinit	100	crystal
	1	castep	101	fhi-aims
	2	dacapo	102	gaussian
	3	gpaw	103	octopus
	4	qbox	104	siesta
	5	quantum-espresso	105	turbomole
	6	sphinx		
	7	vasp		
	200	exciting	999	unknown
	201	fleur		
	202	wien2k		
basis_set	0	plane waves		
	1	atomic orbitals		
	2	gaussians		
	9	unknown		
xc_functional	0	hf ab-initio	100	pw lda
	1	sx ab-initio	101	pz lda
			102	vwn lda
	200	blyp gga	300	tpss meta
	201	pbe gga		
	202	pw91 gga		
	203	rpbe gga		
	204	pbesol gga		
	205	wc gga		
	206	lm gga		
	400	b3lyp hybrid	999	unknown
	401	hse hybrid		
	402	pbe0 hybrid		
	403	tpssh hybrid		
pseudopotential	0	all electron	10	martins-troullier nc

Continued on next page

Table 2.1 – continued from previous page

key	val	description	val	description
			11	bachelet-hamman-schlueter nc
			12	morrison-bylander-kleinman nc
	20	vanderbilt us	30	kresse-joubert paw
	21	rappe-rabe-kaxiras-joannopoulos us	99	unknown

For `dft` reference data, four general identifiers namely `code`, `xc_functional`, `pseudopotential` and `basis set` are passed to the database. Each of these identifiers have a corresponding integer value for the purpose of classification (see *General DFT identifiers*). The DFT codes are arranged in the following manner. The first set use mainly plawe wave basis while those from 100-199 use local atomic orbital basis while the third are all-electron DFT codes. The classification for the other identifiers are are also evident from *General DFT identifiers*. The database returns a list of `Atoms` that meet these criteria for the general identifiers. The database is expected to also supply supplementary identifiers listed in *Supplementary DFT identifiers* in `Atoms.info`. The information are used in BOPcat to filter the data. To extend the supplementary identifiers, one simply add keywords in `variables.data_keys()`

Supplementary DFT identifiers

key	val	description
hubbard	0.0	U value
lr_correction	0/1	long-range correction
relativistic	0/1	relativistic
valency	3pd7s1	valence states
nlc_correction	0/1	non-linear core correction
encut	0.0	cut-off energy
deltak	0.0	k-points(1/Angstroem)
encut_ok	0/1	converged wrt encut
deltak_ok	0/1	converged wrt deltak
author	Pedro5	author ID

The database should also return identifiers for the structure. These include the `stoichiometry`, `space group number`, `sytem type`, `calculation type` and `calculation order`. The `system_type` can refer to any of the following: `bulk`, `cluster`, `defect`, `surface` or `interface`. An integer value is also assigned to the system types, i.e.

```
0-bulk 1-cluster 2-defect 3-surface 4-interface
```

Todo

Extend system types.

The nature of `calculation_type` and `calculation_order` vary depending on `system_type`. These are listed in *Structure identifiers*.

Structure identifiers

calc_type	description	calc_order	description
bulk			
0	relaxation	N	0-unrelax, 1-relax ion, 2-relax vol, 3-relax all
1	volume	0.0	volume per atom
2	elastic	v.x	v-strain(Voigt), x-displacement
3	phonon	Iv.x	I-atom index, v-strain(Voigt), x-displacement
4	transformation	Td.x	T-transformation type, d.x-deformation factor
99	unknown	99.99	unknown

Todo

Determine identifier for other system types.

Each structure may contain an energy, forces,stresses, eigenvalues and orbital_character, vacancy energy and other properties. To extend this to include other properties, one simply needs to add the key to the `variables.data_keys()`.

CHAPTER

THREE

CLASSES AND MODULES

3.1 Input controls

class catcontrols.**CATControls**(***kwargs*)

Handles all input controls.

Parameters are categorized as either data-, opt- calculator- and model-specific

Parameters

•*elements*: list

list of chemical symbols. Bond pairs will be generated from the list.

•*calculator_settings*: dict

options to be passed to the calculator

•*calculator*: str

name of the calculator

•*calculator_nproc*: int

number of parallel processes

`None`: will not parallelize

•*data_parameters*: dict

specifications for the data, e.g.:

```
data_parameters = {'xc_functional':30, 'encut':400}
```

•*data_system_parameters*: dict

specifications for the structures, e.g.:

```
data_system_parameters = {'stoichiometry':['Fe2','Fe']}
```

•*data_filename*: str

filename or path to file containing the structures and properties

•*data_free_atom_energies*: dict

dictionary of element:free atom energies

`dimer`: will generate free_atom_energies from dimers

`None`: will read from atomic_properties in `variables`

•*opt_variables*: list

list of dictionary of contraints on the parameters:

```
variables = [{'bond':['Fe','Fe'],'rep1':[True,True]}
            ,{'bond':['Fe','Nb'],'rep2':[True,True,True]}
            ,{'atom':['Fe'],'valenceelectrons':[True]}]
```

•*opt_structures*: list

list of structures in target set:

```
structures = ['Fe*/*/0/1/*']
```

•*opt_test_structures*: list

list of structures in test set

•*opt_optimizer*: str

name of optimizer

•*opt_optimizer_options*: dict

controls to be passed to the optimizer

•*opt_objective*: str

name of the objective function

•*model*: str

name of the model

•*model_pathtomodels*: str

filename or path to file containing model

•*model_pathtobetas*: str

filename or path to file containing bond integrals

•*model_pathtoonsites*: str

filename or path to file containing onsites

•*model_functions*: dict

dictionary of keyword:bondpair:function or keyword:function, function is one of the instances in functions module. The latter implies that all bond pairs will have the same functional form:

```
import functions as funcs
functions = {'ddsigma':{'Fe-Fe':funcs.exponential()
                        ,'Nb-Nb':funcs.GSP()}
                        ,'rep1':funcs.rep_exponential()}
```

•*model_valences*: dict

dictionary of element:valency:

```
valences = {'Fe':'d','Nb':'d'}
```

•*model_valenceelectrons*: dict

dictionary of element:number of valence electrons:

```
valenceelectrons = {'Fe':7.1,'Nb':4.0}
```

`None`: will read valenceelectrons from `variables.atomic_properties()`

- *model_orthogonal*: bool

  orthogonal or non-orthogonal model

- *model_betafitstruc*: str

  structure from which the betas were derived (see format of betas files)

- *model_betatype*: str

  method used in bond integral generation

- *model_betabasis*: str

  basis used in bond integral generation

- *model_cutoff*: dict

  dictionary of cutoff keyword:bondpair:value or keyword:value.

  The latter implies that all bond pairs will have the same value:

```
cutoff = {'rcut' : {'Nb-Nb':4.5,'Fe-Fe':4.5,'Fe-Nb':4.5}
         ,'dcut' :0.5}
```

- *verbose*: int

  prints details at different levels

check_input()
Checks the consistency of input data type.

convert_params()
Convert data and system parameters into integers.

gen_functions()
Set default functions. Returns dictonary of key:functions

initialize()
Initializes input controls.

Calls *check_input()*, *make_lower()*, make_pairs(), *convert_params()*

make_lower()
Change case of all strings in calculator_settings.

3.2 Reference data

class catdata.**CATData**(***kwargs*)
Defines a set of reference data for parametrization.

It stores the structures and their properties as ASE Atoms objects.

Extended properties used in the parametrization such as eigenvalues, formation energies and elastic constants are stored in the info dictionary.

Todo

extend ASE Atoms object

The main functionality of this object is to query structures for parameterization.

Parameters

- *controls*: instance of CATControls
  CATControls object to initialize parameters
- *atoms*: list of ASE Atoms instances
  None: will read from file.
- *filename*: str
  file containing the structures and their properties, can also be path to file.
- *dataparams*: dict
  dictionary of data specifications used to filter structures.

  sample keywords: *deltak*, *encut*
- *sysparams*: dict
  dictionary of system specifications used to filter structures.

  sample keywords: *calculation_type*, *spin*
- *elements*: list
  list of chemical symbols
- *free_atom_energies*: dict
  dictionary of element:free atom energies.

  `dimer`: will generate free_atom_energies from dimers.

  `None`: will read from atomic_properties in `variables`.
- *verbose*: int
  controls verbosity e.g.

  prints out structures if verbose > 1

get_atoms()
Returns list of all ASE atoms objects of all structures. Builds them from file if not initialized.

get_atoms_info(*key*)
Returns the *info[key]* values of all atoms

Parameters

- *key*: str
  key of entry in info dictionary

get_equilibrium_distance(*structure*)
Returns the smallest bond length of the lowest energy structure in the group

Parameters

- *structure*: str
  strucname, e.g. *'dimer'*

  system_ID, e.g. *'Fe/229/0/1/*'* : all bcc-Fe E-V structures

get_free_atom_energies()
Returns a ditionary of free atom energies corresponding to each of the elements.

get_ground_state(*elements=None*, *out='atoms'*, *spin=0*)
Returns the lowest energy structure containing all the elements in elements.

For alloys, energy of formation is calculated.

Parameters

- ***elements*: list**list of chemical symbols

- ***out*: str** directive for output, if 'atoms' will return the ASE Atoms object, otherwise will return info[out]
- ***spin*: int** 0 : all 1 : only non-mag 2 : only mag

get_parent(*atom*)

Returns the system_ID of the parent of atom.

If parent is not found will return system_ID of atom.

Note: Only considers bulk atoms with different volumes from parent but should be generalized for other calculation types However, for the structure map, even if two atoms are related but have different cells, e.g transformation path they will have different moments hence should be treated as separate structures.

Parameters

- *atom*: instance of *ase.Atoms*
  child of the parent atom

get_ref_atoms(*structures=None*, *quantities=None*, *sort_by=None*, *refene=None*)

Returns list of ASE atoms objects specified in structures and with property in quantities.

Parameters

- *structures*: list
  list of strings of strucname of part of strucname, e.g. 'bcc' or system_ID, wildcards are permitted, e.g. `Fe*/*/*/1/*` (see definition of system_ID)
- *quantities*: list
  list of desired properties that a structure must possess to be included. If a structure has more than one property, it will be included multiple times.

  if quantities is None will simply return what is stored in memory.

get_ref_data(*structures=None*, *quantities=None*)

Returns the properties of the reference atoms. See *get_ref_atoms()* for description of parameters.

If quantities is None, it will return what is stored in memory, otherwise, it will initialize ref_atoms if quantities is not the same as stored.

get_ref_weights(*structures=None*, *quantities=None*)

Returns the weights of the reference atoms in the optimization. See *get_ref_atoms()* for description of parameters.

If quantities is None, it will return what is stored in memory, otherwise, it will initialize ref_atoms if quantities is not the same as stored.

If any of the reference atoms has no weight then all the weights will be nullified.

get_structuremap_distance(*struc0*, *strucs*, *index='total'*)

Returns the distance of the strucs from struc0 in the structure map

Parameters

- *struc0*: str
  strucname, e.g. *'dimer'*

  system_ID, e.g. *'Fe/229/0/1/*'* : all bcc-Fe E-V structures

  ASE Atoms object, coordinates is expected to be normalized globally
- *strucs*: list
  list of structures (format similar to struc0)
- *mode*: str

average/total/maximum of the distances

- *index*: list, int, str
  list of atom indices corresponding to each struc in strucs whose distance from struc0 is returned, if int will apply same index to all strucs, if total will average the coordinates

get_structures(*name=False*)
Returns the system_ID or strucname (name=True) of all structures. The definition of system_ID varies depending on data_type of structure.

3.3 Model

class catparam.**CATParam**(***kwargs*)
Defines the model for optimization.

It includes functionalities to read or generate models.

The models are stored as list of model objects, e.g. modelsbx.

Todo

generalize models format for other calculators

Parameters

- *controls*: instance of CATControls
  CATControls object to initialize parameters
- *elements*: list
  list of chemical symbols
- *model*: str
  model version
- *model_filename*: str
  file containing model, can also be path to file
- *valences*: dict
  dictionary of element:valency
- *valenceelectrons*: dict
  dictionary of element:number of valence electrons
  `None`: will read valenceelectrons from `variables.atomic_properties()`
- *functions*: dict
  dictionary of keyword:bondpair:function or keyword:function, function is one of the instances in functions module. The latter implies that all bond pairs will have the same functional form
- *cutoff*: dict
  dictionary of cutoff keyword:bondpair:value or keyword:value. The latter implies that all bond pairs will have the same value
- *pathtobetas*: str
  path to betas files

  `None`: will use `variables.pathtobetas()`
- *pathtoonsites*: str
  path to onsites files

  `None`: will use `variables.pathtoonsites()`
- *betafitstruc*: str

structure from which the betas were derived (see format of betas files)

`None`: will default to 'dimer'

- *betatype*: str
  method used in beta generation

  `None`: will default to 'loewdin'
- *betabasis*: str
  basis used in beta generation `None`: will default to 'tz0'
- *calculator*: str
  name of the calculator
- *calculator_settings*: dict
  dictionary of controls passed to the calculator
- *orthogonal*: bool
  orthogonal or non-orthogonal model
- *data*: instance of CATData
  CATData object to required to generate models

average_modelsbx(*iterations='all'*)

Returns a model by averaging models in iterations.

Parameters

- *iterations* : list
  indices of the models to be averaged

  all : include all models

calc_rcut(*pair*, *debug=False*)

Returns a dictionary of cut-off keyword: value.

rcut is set to the distance when the value of the most long-ranged bond integral is 5 percent its value at the equilibrium dimer distance. The hierarcy of the range of the bond integrals (m=0) for simplicity is assumed as follows:

`ppsigma, spsigma, sssigma, pdsigma, sdsigma, ddsigma`

The cut-off function starts at 15 percent. *r2cut* is *1.5*rcut* and *d2cut* is *2*dcut*

It also sets the cutoff versions to cosine.

Parameters

- *bondpair*: list
  pair of chemical symbols
- *debug*: bool
  if debug will plot the cut-off values with the reference bond integral

gen_jii(*elem*)

Returns the Jii parameter. This is read from `variables.atomic_properties()`.

Parameters

- *elem*: str
  chemical symbol

gen_model(***kwargs*)

Returns model specific to the calculator.

Generates model. Calls calculator-specific function to generate model.

Todo

Extend to other calculators.

Parameters

•*kwargs* : dict
directives for future extensions.

gen_modelsbx(***kwargs*)

Returns a modelsbx object. See *bopmodel.modelsbx*

Generates modelsbx. Calls `_gen_atomsbx()` and `_gen_bondsbx()`

Parameters

•*kwargs* : dict
directives for future extensions. For example, *part* can be set to *bond*, *rep* or *all* to specify part of modelsbx that is generated.

gen_onsite(*elem*)

Returns a dictionary of the valence:onsites read from the onsite file.

It calls the function `_read_onsites()`.

The filename must follow the format
`[structure]_[elem][elem]_[basis]_[betatype].onsites`
The structure, basis and betatype are similar to the bond integrals. The onsite is taken as the average over a range of distances.

Todo

fit distance dependence of onsites

Note: Determine if onsite should just be set to zero.

Parameters

•*elem*: str
chemical symbol

gen_stoner(*elem*, *struc='gs'*)

Returns the Stoner parameter. This is read from `variables.atomic_properties()`.

..todo:: fit to mag-non mag energy difference of struc

Parameters

•*elem*: str
chemical symbol

gen_valenceelectrons(*elem*)

Returns the number of valence electrons and orbitals. This is read from `variables.atomic_properties()`.

Parameters

•*elem*: str
chemical symbol

get_SED(*elem*, *model*, *strucs=None*)

Returns an array of the bond energy differences of the structures in strucs. The reference is the first structure not necessarily the lowest in energy in order to simplify fitting where one does not need the index of the lowest energy structure.

Calls `sedt.energy_diff()` to calculate bond energies of structures adjusted to have the same average second moment.

Parameters

•*elem* : str

chemical symbol

- *model* : calculator-specific object
  model to be used by the calculator
- *strucs* : list
  list of system_ID's corresponding to relaxed structures

  None: will include all relaxed structures

get_atoms_properties(*elem*)
Returns the mass, number of orbitals and valence electrons, Jii, onsite and Stoner parameters. This calls *gen_valenceelectrons()*, *gen_jii()*, *gen_stoner()*, *gen_onsite()*.

Parameters

- *elem*: str
  chemical symbol

get_cutoffpara(*bondpair*)
Returns a dictionary of cut-off keyword: value. Calls *calc_rcut()*

Parameters

- *bondpair*: list
  pair of chemical symbols

get_fitfunc(*bondpair*)
Returns a dictionary of keyword:functions

Parameters

- *bondpair*: list
  pair of chemical symbols

get_initial_bondparam(*bondpair*)
Returns a dictionary of keyword:functions resulting from the initial fitting of the bond/overlap integrals.

Parameters

- *bondpair*: list
  pair of chemical symbols

get_initial_repparam(*bondpair*)
Returns a dictionary of keyword:functions resulting from the initial fitting of the dimer repulsive energy

Parameters

- *bondpair*: list
  pair of chemical symbols

get_model(*iteration=0*)
Returns the model generated at N=iteration step.

Parameters

- *iteration* : int
  index of the model in list

read_model(***kwargs*)
Returns model specific to the calculator.

Reads model. Calls calculator-specific function to read model.

Todo

Extend to other calculators.

Parameters

- *kwargs* : dict
  directives for future extensions.

read_modelsbx(***kwargs*)
Returns a modelsbx object. See *bopmodel.modelsbx*

Reads modelsbx. Calls `bopmodel.read_modelsbx()`
Parameters
•*kwargs* : dict
directives for future extensions.

class `beta.`**Beta**(***kwargs*)
Defines a set of bond integrals for constructing the hamiltonian.

It reads the data points from a file stored in pathtobetas. A set of functions in fitfuncs is parametrized with respect to the data.

The betas file should be named as [struc]_[ele1][ele2]_[basis]_[betatype].betas

See the folder betas in the examples folder for reference.
Parameters
•*structure*: str
structure from which the bond integrals were derived
•*bondpair*: tuple
pair of chemical symbols
•*basis*: str
basis used for downfolding
•*betatype*: str
can be unscreened, overlap, loewdin or other beta type
•*valence*: tuple
pair of valency, e.g. `sd`
•*pathtobetas*: str
path to betas file

read_betas()
Reads in bond integrals from a betamaker output file(.betas) saved in pathtobeta. If not provided, will read from folder: /betas. The filenames must follow the format: [structure]_[elementA][elementB]_[basis]_[betatype].betas

set_fitfuncs(*fitfuncs*)
Set fitting function , default(sum_exponential)

zip_betas()
Returs a dictionary of the bond integrals

3.4 Calculator

class `catcalc.`**CATCalc**(***kwargs*)
Defines a set of calculators to determine required properties.

The calculators are stored as list of instances of the calculator.
Parameters
•*calculator*: str
name of the calculator
•*calculator_settings*: dict
dictionary of calculator-specific parameters
•*nproc*: int
number of parallel processes

`None`: will not parallelize

default: number of cpu * 2
- *controls*: instance of CATControls
  CATControls object to initialize parameters
- *parallel*: string
  serial, multiprocessing, mpi

static calc_def_ene(*atom*, *kwargs*)
Returns the Atoms object with the calculated defect energy.

The reference bulk structures are in atom.info['reference_atoms'].

Parameters
- *atom*: instance of ASE Atoms object
  structure to calculate
- *kwargs*: dict
  directives for calculator

static calc_ebs(*atom*, *kwargs*)
Returns the Atoms object with the calculated eigenvalues.

Parameters
- *atom*: instance of ASE Atoms object
  structure to calculate
- *kwargs*: dict
  directives for calculator

static calc_efs(*atom*, *kwargs*)
Returns the Atoms object with the calculated energy, forces and stresses.

Parameters
- *atom*: instance of ASE Atoms object
  structure to calculate
- *kwargs*: dict
  directives for calculator

static calculate(*atom*, *kwargs*)
Returns the Atoms object with the calculated property.

The required property is defined by atom.info['required_property']

Calls *calc_ebs()*, *calc_efs()*, *calc_def_ene()*

Parameters
- *atom*: instance of ASE Atoms object
  structure to calculate
- *kwargs*: dict
  directives for calculator

get_atoms()
Returns a list of the structures each an ASE Atoms object assigned for calculation.

get_calculator(*i*, *update*)
Returns the calculator. If None will initialize the calculator.

Parameters
- *update*: bool
  True: will update the model used by the calculator

get_calculators()
Returns a list of calculators corresponding to each structure

calls *get_calculator()*

get_property(**kwargs*)
Returns list of calculated properties for all structures.

Calls *calculate()*

Parameters

•*kwargs*: dict
directives for calculator

get_structures()
Returns a list of the system ID of all the structures

pack_atoms()
Pack all properties for the same structure so do only one calculation of all properties for one structure.

set_atoms(*atoms*)
Set the structures for calculation. The calculators and results are reset.

Parameters

•*atoms*: list
list of ASE Atoms objects

set_model(*model*)
Set the model for the calcuator. The calculators and results are reset.

Parameters

•*model*: instance of calculator-specific model
model used for the calculator

3.5 Optimization kernel

class catkernel.**CATKernel**(***kwargs*)
Defines the optimization kernel.

It generates the objective function and the optimizer.

Parameters

•*controls*: instance of CATControls
CATControls object to initialize parameters
•*calc*: instance of CATcalc
CATcalc object to define the calculator
•*ref_data*: list
list of calculated properties
•*variables*: list
list of dictionary of contraints on the parameters
•*objective*: callable object
should take parameters as argument and return the error

None: will generate default objective function

str: name of the objective function, should be implemented in *CATobjective*
•*optimizer*: callable object
should return optimized parameters

str: name of the optimizer, should be implemented in *CAToptimizer*
•*optimizer_options*: dict
controls to be passed to the optimizer
•*weights*: list
weight of each structure in the error

None: will assign weight from `variables.fit_weights()`
•*log*: str
output file for writing parameters and error at each optimization step

- *verbose*: int
  controls verbosity e.g.

  prints on screen rms at each optimization step if verbose >= 1
- *dump_min_model*: bool
  logical flag to switch on dumping of best model at each iteration

gen_objective()
" Generates the objective function. Standard setting is unknown.

get_optimized_model()
Returns the optimized model.

optimize()
Builds objective and optimizer objects if not initialized and runs optimization. The resulting model is assigned to optimized_model.

test(*param_weights=None*)
Calculates the error for current model with parameters weighted by param_weights

write_summary()
Prints to screen the starting and ending parameters.

class catkernel.**CATobjective**(***kwargs*)
Defines the objective function. Callable with the parameters as argument, will return the corresponding error.

Parameters

- *calc*: instance of CATcalc
  CATcalc object to define the calculator
- *ref_data*: list
  list of calculated properties
- *variables*: list
  list of dictionary of contraints on the parameters
- *penalty_coeffs*: list
  list of penalty coefficients for individual coefficients
- *weights*: list
  weight of each structure in the error

  None: will assign weight from `variables.fit_weights()`
- *log*: str
  output file for writing parameters and error at each optimization step
- *error_vec_log*: str
  output file for writing error for each structure at each optimization step
- *verbose*: bool
  prints on screen rms at each optimization step
- *array*: bool
  if error is vector or scalar
- *name*: str
  name of the objective function
- *dump_min_model*: bool
  logical flag to switch on dumping of best model at each iteration

default(*x0*)
Default objective function.

Returns the diffferences between the calculated properties and the reference.

difference(*x0*)
Objective function for energy differences calculations.

Returns the differences between delta(calculated properties) and delta(reference):

```
        delta_data = data[:len(data)/2] - data[len(data)/2:]
```

is_bond_required()
Checks if bond-related parameters are included in fit

write()
Dumps the parameters and errors at each step on files.

class catkernel.**CAToptimizer**(***kwargs*)
Defines the optimizer. Callable and returns the optimized parameters.

Parameters

- *objective*: callable object
  should take parameters as argument and return the error
- *x0*: list
  initial guess for parameters
- *optimizer_options*: dict
  controls to be passed to the optimizer
- *name*: str
  name of the optimizer

3.6 BOPfox-ASE interface

class bopcal.**BOPfox**(*restart=None*, *track_output=False*, *atoms=None*, *atomsbx=None*, *bondsbx=None*, *infoxbx='infox.bx'*, *modelsbx='models.bx'*, *bopfox='bopfox'*, *savelog=False*, *mem_limit=None*, *ignore_errors=True*, *debug=False*, *kpoints=None*, *root_tmp_folder='/tmp'*, *magconfig=None*, ***kwargs*)
Defines an ASE interface to BOPfox.

In order to define a calculator, the user should provide a modelsbx, or atomsbx and bondsbx. These can either be objects (see `modelsbx`, `atomsbx` and `bondsbx`), filenames or paths. In addition, bopfox input controls (infox parameters) can be optionally provided, otherwise the calculator is expecting an infox.bx file on current working directory. The calculator has other attributes on top of those of ASE such as *get_moments()*

Todo

extend ASE Atoms object to call BOPfox native functions.

Parameters

- *atomsbx*: str or instance of atomsbx
  used to create atoms.bx

  `None`: will use modelsbx
- *bondssbx*: str or instance of bondsbx
  used to create bonds.bx

  `None`: will use modelsbx
- *modelsbx*: str or instance of modelsbx
  used to create models.bx

  `None`: will use modelsbx
- *bopfox*: str
  bopfox executable
- *savelog*: bool
  logical flag to save log file for further anaylis

•*mem_limit*: float
sets memory limit for bopfox calculation
•*ignore_errors*: bool
logical flag to skip bopfox errors
•*root_tmp_folder*: str
directory where a tmp folder will be generated
•*debug*: bool
logical flag for debugging purposes, will not delete temporary folders
•*kpoints*: list
coordinates of kpoints to be used for tight-binding band structure calculation
•*kwargs*:
optional keyword arguments to be written in infox, if not provided will use infox.bx

calculate(**args*)
Sets up a temporary foler for perfoming a BOPfox calculation. Then writes the corresponding files and performs a calculation. Cleans up afterwards.

get_mode()
Returns the calculation mode (string).

get_moments(*atom_index=0*, *moment=2*)
Returns the moments.

If the atom index is 0, then average will be returned If the atom index is NOT an integer, it will return all moments

get_name()
Returns the name of the calculator (string).

inspect_modelsbx(*key*, *vtype=<type 'str'>*)
Checks the value of key in modelsbx.

write_infox(***kwargs*)
Writes infox.bx from input parameters, atomsbx and bondsbx compatible.

write_infox_new(***kwargs*)
Writes infox.bx from input parameters, models.bx compatible.

class bopmodel.**atomsbx**(***kwargs*)
Defines an atoms.bx object for BOPfox.

get_atom()
Returns the element in the atoms.bx object

get_mass()
Returns the mass of the element

get_version()
Returns the version of the atoms.bx object

rattle(*var='all'*, *factor='random'*, *maxf=0.1*)
Generate a modified atomsbx with parameters defined in var modified by factor.
Parameters

•*var*: dictionary of key, constraints
•*factor*: float

write(*filename='atoms.bx'*, *update=False*)
Writes out atomsbx object to file. The keyword update does not necessarily mean that atomsbx is updated.

It just means that succeeding entries will not have the header 'Version'.

class bopmodel.**bondsbx**(***kwargs*)
Defines a bonds.bx object for BOPfox.

get_bond()
Returns the elements in the bonds object

get_version()
Returns the version of the bonds object

rattle(*var='all'*, *factor='random'*, *maxf=0.1*)
Generate a modified bondsbx with parameters defined in var modified by factor.
Parameters

•*var*: dictionary of key, constraints
•*factor*: float

write(*filename='bonds.bx'*, *update=False*)
Writes bondsbx object to file update means that you read the current bonds.bx file and write it with bopbonds

class bopmodel.**modelsbx**(***kwargs*)
Defines a modelsbx object in BOPfox

bond_parameters_to_functions(*bondlist='all'*, *variables=None*)
Convert list of parameters in bondsbx to corresponding function (see :mod functions)

rattle(*var='all'*, *factor='random'*, *maxf=0.1*)
Generate a modified bondsbx with parameters defined in var modified by factor.
Parameters

•*var*: list of dictionary of key, constraints
e.g. [{'bond':['Fe','Fe'],'sssigma':[True,True,True]},{'atom':['Fe'], 'onsitelevels':[True]}]
•*factor*: float

3.7 Miscellaneous

class beta.**Beta**(***kwargs*)
Defines a set of bond integrals for constructing the hamiltonian.

It reads the data points from a file stored in pathtobetas. A set of functions in fitfuncs is parametrized with respect to the data.

The betas file should be named as [struc]_[ele1][ele2]_[basis]_[betatype].betas

See the folder betas in the examples folder for reference.
Parameters
•*structure*: str
structure from which the bond integrals were derived
•*bondpair*: tuple
pair of chemical symbols
•*basis*: str
basis used for downfolding
•*betatype*: str
can be unscreened, overlap, loewdin or other beta type
•*valence*: tuple
pair of valency, e.g. `sd`

- *pathtobetas*: str
  path to betas file

read_betas()
Reads in bond integrals from a betamaker output file(.betas) saved in pathtobeta. If not provided, will read from folder: /betas. The filenames must follow the format: [structure]_[elementA][elementB]_[basis]_[betatype].betas

set_fitfuncs(*fitfuncs*)
Set fitting function , default(sum_exponential)

zip_betas()
Returs a dictionary of the bond integrals

I

M

O

P

R

S

T

W

Z