EZFF Documentation

Release 0.9 Beta

Aravind Krishnamoorthy

Apr 04, 2019

Contents:

EZFF is a Python-based library for quick and easy parameterization of forcefields and interatomic potentials for molecular dynamics simulations. EZFF provides interfaces to popular atomistic simulation software, GULP, VASP and QChem and uses Platypus for solving multi-objective optimization problems. Use the links below to get started.

CHAPTER 1

Basic Usage

You will need five pieces of information to get started with focefield optimization using EZFF.

1. Ground truths - Physical properties to parameterize the forcefield against
2. Serial GULP executable
3. Forcefield template
4. Maximum and minimum values of decision variables
5. A master python script (*run.py* in examples) that defines different errors, handles GULP jobs and optimization parameters
6. (**Optional**) A working installation of MPI and mpi4py for parallel optimization

Ground truths Ground truth values (for lattice constant, elastic constants, energies, phonon dispersion curves etc) can either be provided by hand in the master script, or can be calculated using methods provided in the different ezff.interfaces modules

Serial GULP executable Serial GULP executable built from source code available from http://gulp.curtin.edu.au/gulp/ . Parallel optimization jobs simply launch multiple copies of the serial executable.

Forcefield template A forcefield template file is used to designate which variables must be considered for optimization. The forcefield template is constructed from a functioning GULP-readable forcefield by replacing the parameters that need to be optimized with variable names enclosed within dual angle-brackets. For example, the following Lennard-Jones forcefield for solid Neon (from examples lj-serial and lj-parallel)

```
lennard epsilon 12 6  # Tell GULP that the next line will contain LJ␣
↪parameters
Ne Ne 1.0 1.5  # Format: atom1 atom2 epsilon sigma
```

can be converted to a template by replacing epsilon and sigma by optimizable decision variables:

```
lennard epsilon 12 6  # Tell GULP that the next line will contain LJ␣
↪parameters
Ne Ne <<eps>> <<sgma>>  # Format: atom1 atom2 epsilon sigma
```

This fill will instruct EZFF to optimize the **eps** and **sgma** variables. Decision variables are assumed to be real-valued by default. Any decision variable beginning with an underscore (_) is assumed to be integer-valued.

The template forcefield must be paired with an appropriate file specifying the permissible ranges of these decision variables.

Decision variable ranges This file lists the permissible ranges of decision variables (i.e. minimum and maximum value that the variable can take) during forcefield optimization. This text file is written in the following format:

```
Decision_variable_1(without the angle brackets)    Minimum_value ␣
↪Maximum_value
Decision_variable_2(without the angle brackets)    Minimum_value ␣
↪Maximum_value
Decision_variable_3(without the angle brackets)    Minimum_value ␣
↪Maximum_value
.
.
Decision_variable_n(without the angle brackets)    Minimum_value ␣
↪Maximum_value
```

A valid variable range file for the Lennard-Jones template file above is given below:

```
eps 0.5 2.5
sgma 0.1 0.4
```

Warning: Please ensure that the template and variable_ranges file are compatible. Specifically, 1. Every variable defined in the forcefield template must have one (and only one) corresonding entry in the variable ranges file 2. The variable ranges file should not refer to variables not present in the template file

Python script This python script should include, at the very least, your custom function to calculate the error (i.e. deviation of the forcefield from ground-truths), an ezff.OptProblem, an ezff.Algorithm and a call to ezff.optimize, and for the case of parallel execution, an ezff.Pool.

The custom error function should be written to accept one input – a dictionary of decision_variable-value pairs (e.g. {'eps': 1.273, 'sgma': 0.12} for example above) and should return a list of all computed objectives. The length of this returned list should equal the number of errors provided during OptProblem initialization.

CHAPTER 2

Examples

The following forcefield optimization examples showcase the features of EZFF

2.1 lj-serial

Optimization of a Lennard-Jones forcefield for FCC Neon against 2 objectives – **bulk modulus** and **lattice constant**

Features demonstrated in this example

1. Basic use of forcefield templates and variable_range files
2. Reading-in elastic modulus tensor from GULP run
3. Reading-in lattice constants after a GULP run

2.2 lj-parallel

Parallel optimization of a Lennard-Jones forcefield for FCC Neon against 2 objectives – **bulk modulus** and **lattice constant**

Features demonstrated in this example

1. All features from lj-serial
2. Basic use of ezff.Pool for parallel optimization

2.3 sw-serial

Optimization of a Stillinger-Weber forcefield for the 2H-MoSe2 monolayer system against 3 objectives – **Lattice constant** (a), **Elastic modulus** (C_{11}) and **Phonon dispersion**

Features demonstrated in this example

1. Reading-in phonon dispersion from GULP and VASP data files
2. Calculating error between phonon dispersions
3. Calculating error between computed and ground-truth phonon dispersions
4. Reading-in elastic modulus tensor from GULP run

2.4 sw-parallel

Parallel optimization of a Stillinger-Weber forcefield for the 1T' monolayer system against 6 objectives – **Two lattice constants** (a and b), **One elastic modulus** (C_{11}) and **Three phonon dispersion curves** (one each for compressed, relaxed and expanded crystals)

Features demonstrated in this example

1. All features from sw-serial
2. Spawning and using MPI pools for optimization
3. Non-uniform weighting schemes for calculating phonon dispersion errors

2.5 multialgo-sw-parallel

Parallel optimization of a Stillinger-Weber forcefield for the 1T' monolayer system against 6 objectives – **Two lattice constants** (a and b), **One elastic modulus** (C_{11}) and **Three phonon dispersion curves** (one each for compressed, relaxed and expanded crystals) using a sequence of multiple multi-objective genetic algorithms. Here, the population from the last epoch of optimization with a single algorithm is used as the initial population for the next algorithm in the sequence.

Features demonstrated in this example

1. All features from sw-parallel
2. Using multiple genetic algorithms in sequence for a single problem
3. Use of different population sizes and epochs for each algorithm in the sequence

2.6 rxff-serial

Optimization of ReaxFF forcefield for a thio-ketone monomer against 1 objective – **C-S vibrational frequency**

Features demonstrated in this example

1. Using QChem interface to read-in QM energies
2. Using GULP interface to perform single-point calculations and read-in energy
3. Using utils.reaxff to construct GULP-compatible reaxff library files

2.7 rxff-parallel

Parallel optimization of ReaxFF forcefield for a thio-ketone monomer against 2 objectives – **Dissociation energy** of the C-S bond and C-S **vibrational frequency**

Features demonstrated in this example

1. All features from rxff-serial
2. Using utils.reaxff methods for generating forcefields templates and variable range files
3. Heterogeneous weighting scheme for calculating errors from potential energy surface scans

2.8 rxff-charge-serial

Optimization of charge parameters in the ReaxFF forcefield for a thio-ketone monomer against 2 objective – **atomic charges** and **structural distortion**

Features demonstrated in this example

1. All features from rxff-serial
2. Use of make_template_qeq
3. Use of ezff.error_atomic_charges and ezff.error_structure_distortion
4. Demonstration of mutation_probability keyword in the ezff.Algorithm call

CHAPTER 3

Code Documentation

The following links provide a detailed description of individual classes and methods in the EZFF package.

3.1 EZFF - Easy forcefield fitting

ezff.**OptProblem**

This module provide general functions for EZFF

ezff.**Algorithm**(*myproblem*, *algorithm_string*, *population=1024*, *mutation_probability=None*, *pool=None*)

Provide a uniform interface to initialize an algorithm class for serial and parallel execution

Parameters

- **myproblem** (*Problem*) – EZFF Problem to be optimized
- **algorithm_string** (*str*) – EZFF Algorithm to use for optimization. Allowed options are NSGAII, NSGAIII and IBEA
- **population** (*int*) – Population size for genetic algorithms
- **mutation_probability** (*float between 0.0 and 1.0*) – Probability of a decision variable to undergo mutation to a random value within defined bounds
- **pool** (*MPIPool or None*) – MPI pool for parallel execution. If this is None, serial execution is assumed

class ezff.**Pool**(*comm=None*, *debug=False*, *loadbalance=False*)

Wrapper for platypus.MPIPool

Parameters **MPIPool** (*MPIPool*) – MPI Pool

__init__(*comm=None*, *debug=False*, *loadbalance=False*)

Initialize self. See help(type(self)) for accurate signature.

This module provide general functions for EZFF

ezff.**optimize**(*problem*, *algorithm*, *iterations=100*, *write_forcefields=None*)
The optimize function provides a uniform wrapper to solve the EZFF problem using the algorithm(s) provided.

Parameters

- **problem** (*Problem*) – EZFF Problem to be optimized
- **algorithm** (*str or list (of strings)*) – EZFF Algorithm(s) to use for optimization. Allowed options are NSGAII, NSGAIII and IBEA, or a list containing any sequence of these options. The algorithms will be used in the sequence provided
- **iterations** (*int or list (of ints)*) – Number of epochs to perform the optimization for. If multiple algorithms are specified, one iteration value should be provided for each algorithm
- **write_forcefields** (*int or None*) – All non-dominated forcefields are written out every write_forcefields epochs. If this is None, the forcefields are written out for the first and last epoch

3.2 ezff.ffio - Methods to read/write forcefield files

This module provide methods to handle reading and writing forcefields

ffio.**generate_forcefield**(*template_string*, *parameters*, *FFtype=None*, *outfile=None*)
Generate a new forcefield from the template by replacing variables with numerical values

Parameters

- **template_string** (*str*) – Text of the forcefield template
- **parameters** (*dict*) – Numerical value of all decision variables in the form of variable:value pairs
- **FFtype** (*string*) – Type of forcefield being optimized. (e.g. reaxff, sw, lj, etc.)

ffio.**read_forcefield_template**(*template_filename*)
Read-in the forcefield template. The template is constructed from a functional forcefield file by replacing all optimizable numerical values with variable names enclosed within dual angled brackets << and >>.

Parameters **template_filename** (*str*) – Name of the forcefield template file to be read-in

ffio.**read_variable_bounds**(*filename*, *verbose=False*)
Read permissible lower and upper bounds for decision variables used in forcefields optimization

Parameters

- **filename** (*str*) – Name of text file listing bounds for each decision variable that must be optimized
- **verbose** (*bool*) – Print all variables read-in

3.3 ezff.errors - Error calculation modules

This module provide functions for computing errors from previously completed MD runs

ezff.errors.**error_atomic_charges**(*MD=None*, *GT=None*)
Calculate error due to difference between MD-computed atomic charges and ground-truth atomic charges

Parameters

- **MD** (*xtal.AtTraj object*) – Relaxed structure after MD run
- **GT** (*xtal.AtTraj object*) – Initial Ground-Truth structure used as input for MD calculations

ezff.errors.**error_energy**(*MD*, *GT*, *weights='uniform'*, *verbose=False*)

Calculate error between MD-computed potential energy surface and the ground-truth potential energy surface with user-defined weighting schemes

Parameters

- **md_disp** (*1D np.array*) – MD-computed potential energy surface
- **gt_disp** (*1D np.array*) – Ground-truth potential energy surface
- **weights** (str *or* list) – User-defined weighting scheme for calculating errors. Possible values are `uniform` - where errors from all points on the PES are weighted equally, `minima` - where errors from lower-energy points are assigned greater weights, `dissociation` - where errors from highest-energy points are assigned greater weights, and *list* - 1-D list of length equal to number of points on the PES scans
- **verbose** (*bool*) – Deprecated option for verbosity of error calculation routine

ezff.errors.**error_lattice_constant**(*MD=None*, *GT=None*)

Calculate error due to optimization of lattice constants in the initial structure.

Parameters

- **MD** (*xtal.AtTraj object*) – Relaxed structure after MD run
- **GT** (*xtal.AtTraj object*) – Initial Ground-Truth structure used as input for MD calculations

ezff.errors.**error_phonon_dispersion**(*MD=None*, *GT=None*, *weights='uniform'*, *verbose=False*)

Calculate error between MD-computed phonon dispersion and the ground-truth phonon dispersion with user-defined weighting schemes

Parameters

- **MD** (*2D np.array*) – MD-computed phonon dispersion curve
- **GT** (*2D np.array*) – Ground-truth phonon dispersion curve
- **weights** (str *or* list) – User-defined weighting scheme for calculating errors provided as a list of numbers, one per band. Possible values are `uniform` - where errors from all bands are equally weighted, `acoustic` - where errors from lower-frequency bands are assigned greater weights, and *list* - 1-D list of length equal to number of bands
- **verbose** (*bool*) – Deprecated option for verbosity of error calculation routine

ezff.errors.**error_structure_distortion**(*MD=None*, *GT=None*)

Calculate error due to relaxation of atoms in the initial structure. The error is the sum of root mean square displacement of atoms.

Parameters

- **MD** (*xtal.AtTraj object*) – Relaxed structure after MD run
- **GT** (*xtal.AtTraj object*) – Initial Ground-Truth structure used as input for MD calculations

3.4 ezff.interfaces.vasp - Interface to VASP

Interface to VASP, the Vienna Ab initio Simulation Package

ezff.interfaces.vasp.**read_atomic_structure**(*structure_file*)
Read-in atomic structure. Currently only VASP POSCAR/CONTCAR files are supported

Parameters **structure_file** (*str*) – Filename of the atomic structure file

Returns xtal trajectory with the structure in the first snapshot

ezff.interfaces.vasp.**read_phonon_dispersion**(*phonon_dispersion_file*)
Read-in ground-truth phonon dispersion curve from VASP+Phonopy

Parameters **phonon_dispersion_file** (*str*) – Filename for the VASP + Phonopy phonon dispersion

Returns 2D np.array of phonon dispersion values

3.5 ezff.interfaces.qchem - Interface to QChem

Interface to Q-Chem, the ab initio quantum chemistry package

ezff.interfaces.qchem.**read_atomic_charges**(*outfilename*)
Read-in a multiple partially-converged structures from a PES scan (including bond-scans, angle-scans and dihedral-scans)

Parameters **outfilename** (*str*) – Single filename for `stdout` from the QChem PES scan job or a list of filenames for `stdout` files from partial QChem PES scan jobs

Returns `xtal` trajectory object with structures and converged energies along the PES scan as individual snapshots

ezff.interfaces.qchem.**read_energy**(*outfilename*)
Read-in a multiple partially-converged structures from a PES scan (including bond-scans, angle-scans and dihedral-scans)

Parameters **outfilename** (*str*) – Single filename for `stdout` from the QChem PES scan job or a list of filenames for `stdout` files from partial QChem PES scan jobs

Returns `xtal` trajectory object with structures and converged energies along the PES scan as individual snapshots

ezff.interfaces.qchem.**read_structure**(*outfilename*)
Read-in a multiple partially-converged structures from a PES scan (including bond-scans, angle-scans and dihedral-scans)

Parameters **outfilename** (*str*) – Single filename for `stdout` from the QChem PES scan job or a list of filenames for `stdout` files from partial QChem PES scan jobs

Returns `xtal` trajectory object with structures and converged energies along the PES scan as individual snapshots

3.6 ezff.interfaces.gulp - Interface to GULP

Interface to GULP, the General Utility Lattice Program

class ezff.interfaces.gulp.**job**(*verbose=False*, *path='.'*)
Class representing a GULP calculation

cleanup()
Clean-up after the completion of a GULP job. Deletes input, output and forcefields files

read_atomic_charges()
Read atomic charge information from a completed GULP job file

Returns xtal.AtTraj object with optimized charge information

read_elastic_moduli()
Read elastic modulus matrix from a completed GULP job

Returns 6x6 Elastic modulus matrix in GPa for each input structure, as a list

read_energy()
Read energy from completed GULP job

Returns Energy of the input structure(s) in eV as a np.ndarray

read_phonon_dispersion(*units='cm-1'*)
Read phonon dispersion from a complete GULP job

Returns 2D np.array containing the phonon dispersion in THz

read_structure()
Read converged structure (cell and atomic positions) from the MD job

Returns xtal.AtTraj object with (optimized) individual structures as separate snapshots

run(*command=None*, *timeout=None*)
Execute GULP job with user-defined parameters

Parameters

- **command** (*str*) – path to GULP executable
- **timeout** (*int*) – GULP job is automatically killed after `timeout` seconds

3.7 ezff.utilities.reaxff - Utility to generate forcefield templates for ReaxFF

class ezff.utils.reaxff.**reax_forcefield**(*filename=None*, *filestring=None*, *template='ff.template.generated'*, *ranges='param_ranges'*)
ReaxFF forcefield class. Used for generating ReaxFF templates

generate_templates()
Function to write-out the current modified forcefield sections into a forcefield template file

make_template_fourbody(*e1*, *e2*, *e3*, *e4*, *bounds=0.1*, *common=False*)
Function to generate decision variables for all four-body terms for a given quartet of elements

Parameters

- **e1** (*str*) – Chemical symbol for element 1
- **e2** (*str*) – Chemical symbol for element 2
- **e3** (*str*) – Chemical symbol for element 3
- **e4** (*str*) – Chemical symbol for element 4

- **bounds** (*float*) – Maximum deviation allowed for each decision variable from its current value in the forcefield
- **common** (*bool*) – Flag for the optimization of common parameters

make_template_qeq(*e1*, *bounds=0.1*)
Function to generate decision variable for Charge Equilibration (QEq) terms

: param e1 : Chemical symbol for element 1 : type e1 : str

: param bounds: Maximum deviation allowed for each decision variable from its current value in the forcefield : type bounds: float

make_template_threebody(*e1*, *e2*, *e3*, *bounds=0.1*, *common=False*)
Function to generate decision variables for all three-body terms for a given triplet of elements

Parameters

- **e1** (*str*) – Chemical symbol for element 1
- **e2** (*str*) – Chemical symbol for element 2
- **e3** (*str*) – Chemical symbol for element 3
- **bounds** (*float*) – Maximum deviation allowed for each decision variable from its current value in the forcefield
- **common** (*bool*) – Flag for the optimization of common parameters

make_template_twobody(*e1*, *e2*, *double_bond=False*, *triple_bond=False*, *bounds=0.1*, *common=False*)
Function to generate decision variables for all two-body terms (i.e. bond-order, attractive and vdW) between two given elements

Parameters

- **e1** (*str*) – Chemical symbol for element 1
- **e2** (*str*) – Chemical symbol for element 2
- **bounds** (*float*) – Maximum deviation allowed for each decision variable from its current value in the forcefield
- **double_bond** (*bool*) – Flag for the presence of a double-bond between elements e1 and e2
- **triple_bond** (*bool*) – Flag for the presence of a triple-bond between elements e1 and e2
- **common** (*bool*) – Flag for the optimization of common parameters

read_forcefield_from_file(*filename*)
Read ReaxFF forcefield from external file

Parameters **filename** (*str*) – ReaxFF forcefield filename

read_forcefield_from_string(*filestring*)
Read ReaxFF forcefield from a given forcefield string

Parameters **filestring** (*str*) – ReaxFF forcefield string

write_formatted_forcefields(*outfilename*)
Function to write-out the current forcefield with correct ReaxFF formatting

Parameters **outfilename** (*str*) – File to which formatted forcefield to be written to

write_gulp_library(*outfilename=None*)
Function to write-out the forcefield in the GULP library format

Parameters **outfilename** (*str*) – File to which the GULP ReaxFF forcefield library

CHAPTER 4

Installing

Install from PyPI using the command

```
pip install EZFF
```

Alternatively, you can install the latest developmental version from GitHub via

```
git clone https://github.com/arvk/EZFF.git
cd EZFF
python setup.py install
```

CHAPTER 5

Contributing

1. Please make sure to submit only passing builds
2. Adhere to PEP8 where you can
3. Submit a pull request

CHAPTER 6

License

EZFF source code and documentation is released under the MIT License

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

e

f

Index

Symbols

A

C

E

F

G

J

M

O

P

R

W