

ForceBalance version 0.11.0

Generated by Doxygen 1.7.6.1



Contents

1	Main Page	1
1.1	Preface: How to use this document	1
1.2	Introduction	2
1.2.1	Background: Empirical Potentials	2
1.2.2	Purpose and brief description of this program	4
1.3	Credits	6
2	Installation	6
2.1	Installing ForceBalance	6
2.1.1	Prerequisites	6
2.1.2	Installing	7
2.2	Installing GROMACS-X2	8
2.2.1	Prerequisites for GROMACS-X2	8
2.3	Create documentation	9
3	Usage	9
3.1	Input file	9
3.2	Directory structure	10
4	Tutorial	10
4.1	Fitting a TIP4P potential using two fitting simulations	10
5	Glossary	11
5.1	Scientific concepts	11
5.2	Option index: General options	12
5.3	Option index: Simulation options	17
6	Todo List	21
7	Namespace Index	22
7.1	Namespace List	22
8	Class Index	23
8.1	Class List	23
9	Namespace Documentation	24
9.1	BaseReader Namespace Reference	24
9.1.1	Detailed Description	24
9.2	counterpoisematch Namespace Reference	25

9.2.1	Detailed Description	25
9.3	custom_io Namespace Reference	25
9.3.1	Detailed Description	25
9.4	forceenergymatch_gmxx2 Namespace Reference	25
9.4.1	Detailed Description	25
9.5	forcefield Namespace Reference	26
9.5.1	Detailed Description	26
9.6	gmxx Namespace Reference	27
9.6.1	Detailed Description	27
9.7	MakeInputFile Namespace Reference	28
9.7.1	Detailed Description	28
9.7.2	Function Documentation	28
9.8	molecule Namespace Reference	28
9.8.1	Detailed Description	28
9.9	nifty Namespace Reference	28
9.9.1	Detailed Description	29
9.10	OptimizePotential Namespace Reference	29
9.10.1	Detailed Description	29
9.10.2	Function Documentation	29
9.11	optimizer Namespace Reference	29
9.11.1	Detailed Description	29
9.12	ParseInputFile Namespace Reference	30
9.12.1	Detailed Description	30
9.12.2	Function Documentation	30
9.13	parser Namespace Reference	30
9.13.1	Detailed Description	30
9.14	project Namespace Reference	31
9.14.1	Detailed Description	31
9.15	qchemio Namespace Reference	31
9.15.1	Detailed Description	31
9.16	simtab Namespace Reference	31
9.16.1	Detailed Description	32
9.17	tinkerio Namespace Reference	32
9.17.1	Detailed Description	32
10	Class Documentation	32
10.1	forcebalance.basereader.BaseReader Class Reference	32

10.1.1 Detailed Description	32
10.1.2 Member Function Documentation	33
10.2 forcebalance.counterpoisematch.CounterpoiseMatch Class Reference	33
10.2.1 Detailed Description	34
10.2.2 Constructor & Destructor Documentation	34
10.2.3 Member Function Documentation	35
10.3 forcebalance.forcefield.FF Class Reference	36
10.3.1 Detailed Description	37
10.3.2 Constructor & Destructor Documentation	37
10.3.3 Member Function Documentation	37
10.3.4 Member Data Documentation	42
10.4 forcebalance.fitsim.FittingSimulation Class Reference	42
10.4.1 Detailed Description	44
10.4.2 Constructor & Destructor Documentation	44
10.4.3 Member Function Documentation	44
10.4.4 Member Data Documentation	46
10.5 forcebalance.forceenergymatch.ForceEnergyMatch Class Reference	46
10.5.1 Detailed Description	48
10.5.2 Constructor & Destructor Documentation	48
10.5.3 Member Function Documentation	48
10.5.4 Member Data Documentation	50
10.6 forcebalance.gmxio.ForceEnergyMatch_GMX Class Reference	50
10.6.1 Detailed Description	51
10.7 forcebalance.forceenergymatch_gmxx2.ForceEnergyMatch_GMXX2 Class Reference	52
10.7.1 Detailed Description	53
10.7.2 Constructor & Destructor Documentation	53
10.7.3 Member Function Documentation	53
10.8 forcebalance.tinkerio.ForceEnergyMatch_TINKER Class Reference	55
10.8.1 Detailed Description	56
10.9 forcebalance.custom_io.Gen_Reader Class Reference	56
10.9.1 Detailed Description	58
10.9.2 Member Function Documentation	58
10.10 forcebalance.gmxio.OTP_Reader Class Reference	58
10.10.1 Detailed Description	59
10.10.2 Member Function Documentation	60
10.10.3 Member Data Documentation	60
10.11 forcebalance.molecule.Molecule Class Reference	60

10.11.1 Detailed Description	62
10.11.2 Constructor & Destructor Documentation	62
10.11.3 Member Function Documentation	62
10.11.4 Member Data Documentation	63
10.12CallGraph.node Class Reference	63
10.12.1 Detailed Description	64
10.13forcebalance.optimizer.Optimizer Class Reference	64
10.13.1 Detailed Description	66
10.13.2 Constructor & Destructor Documentation	66
10.13.3 Member Function Documentation	66
10.13.4 Member Data Documentation	72
10.14forcebalance.project.Project Class Reference	72
10.14.1 Detailed Description	73
10.14.2 Constructor & Destructor Documentation	73
10.14.3 Member Function Documentation	74
10.15forcebalance.qchemio.QCIn_Reader Class Reference	74
10.15.1 Detailed Description	75
10.15.2 Member Function Documentation	75
10.16forcebalance.tinkerio.Tinker_Reader Class Reference	76
10.16.1 Detailed Description	77
10.16.2 Member Function Documentation	77

1 Main Page

1.1 Preface: How to use this document

The documentation for ForceBalance exists in two forms: a web page and a PDF manual. They contain equivalent content. The newest versions of the software and documentation, along with relevant literature, can be found on the [SimTK website](#).

Users of the program should read the *Introduction*, *Installation*, *Usage*, and *Tutorial* sections on the main page.

Developers and contributors should read the Introduction chapter, including the *Program Layout* and *Creating - Documentation* sections. The *API documentation*, which describes all of the modules, classes and functions in the program, is intended as a reference for contributors who are writing code.

ForceBalance is a work in progress; using the program is nontrivial and many features are still being actively developed. Thus, users and developers are highly encouraged to contact me through the [SimTK website](#), either by sending me email or posting to the public forum, in order to get things up and running.

Thanks!

Lee-Ping Wang

1.2 Introduction

Welcome to ForceBalance! :)

This is a *theoretical and computational chemistry* program primarily developed by Lee-Ping Wang. The full list of people who made this project possible are given in the [Credits](#).

The function of ForceBalance is *automatic potential optimization*. It addresses the problem of parameterizing empirical potential functions, colloquially called *force fields*. Here I will provide some background, which for the sake of brevity and readability will lack precision and details. In the future, this documentation will include literature citations which will guide further reading.

1.2.1 Background: Empirical Potentials

In theoretical and computational chemistry, there are many methods for computing the potential energy of a collection of atoms and molecules given their positions in space. For a system of N particles, the potential energy surface (or *potential* for short) is a function of the $3N$ variables that specify the atomic coordinates. The potential is the foundation for many types of atomistic simulations, including molecular dynamics and Monte Carlo, which are used to simulate all sorts of chemical and biochemical processes ranging from protein folding and enzyme catalysis to reactions between small molecules in interstellar clouds.

The true potential is given by the energy eigenvalue of the time-independent Schrodinger's equation, but since the exact solution is intractable for virtually all systems of interest, approximate methods are used. Some are *ab initio* methods ('from first principles') since they are derived directly from approximating Schrodinger's equation; examples include the independent electron approximation (Hartree-Fock) and perturbation theory (MP2). However, the vast majority of methods contain some tunable constants or *empirical parameters* which are carefully chosen to make the method as accurate as possible. Three examples: the widely used B3LYP approximation in density functional theory (DFT) contains three parameters, the semiempirical PM3 method has 10-20 parameters per chemical element, and classical force fields have hundreds to thousands of parameters. All such formulations require an accurate parameterization to properly describe reality.

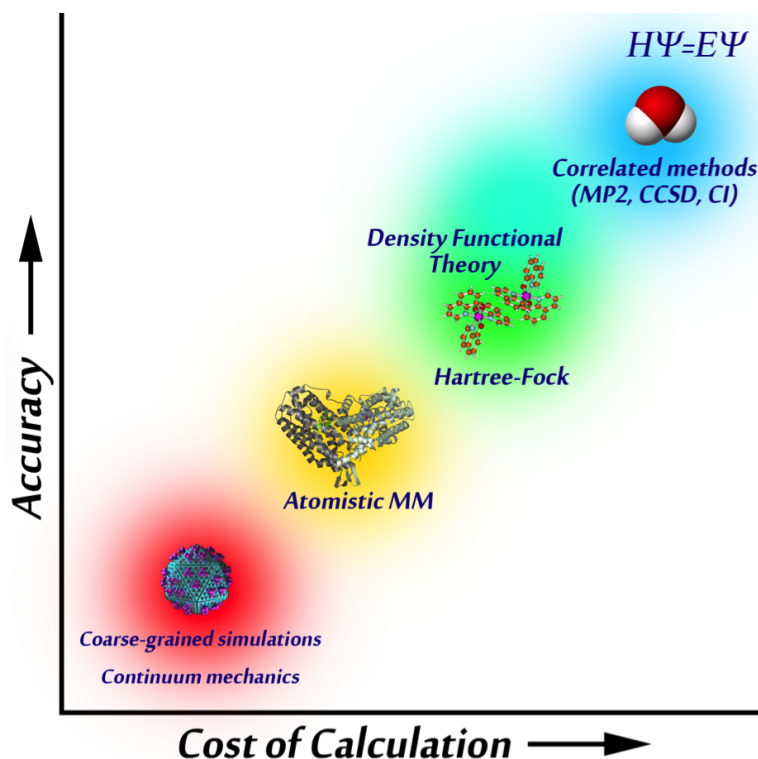


Figure 1: An arrangement of simulation methods by accuracy vs. computational cost.

A major audience of ForceBalance is the scientific community that uses and develops classical force fields. These force fields do not use the Schrodinger's equation as a starting point; instead, the potential is entirely specified using elementary mathematical functions. Thus, the rigorous physical foundation is sacrificed but the computational cost is reduced by a factor of millions, enabling atomic-resolution simulations of large biomolecules on long timescales and allowing the study of problems like protein folding.

In classical force fields, relatively few parameters may be determined directly from experiment - for instance, a chemical bond may be described using a harmonic spring with the experimental bond length and vibrational frequency. More often there is no experimentally measurable counterpart to a parameter - for example, electrostatic interactions are often described as interactions between pairs of point charges on atomic centers, but the fractional charge assigned to each atom has no rigorous experimental or theoretical definition. To complicate matters further, most molecular motions arise from a combination of interactions and are sensitive to many parameters at once - for example, the dihedral interaction term is intended to govern torsional motion about a bond, but these motions are modulated by the flexibility of the nearby bond and angle interactions as well as the nonbonded interactions on either side.

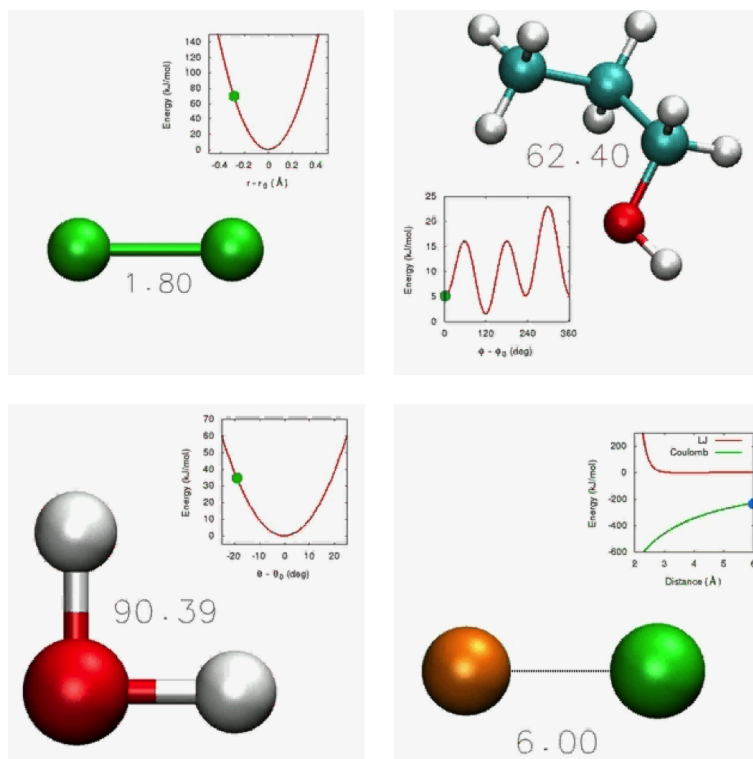


Figure 2: An illustration of some interactions typically found in classical force fields.

For all of these reasons, force field parameterization is difficult. In the current practice, parameters are often determined by fitting to results from other calculations (for example, restrained electrostatic potential fitting (RESP) for determining the partial charges) or chosen to reproduce experimental measurements which depend indirectly on the parameters (for example, adjusting the partial charges on a solvent molecule to reproduce the bulk dielectric constant.) Published force fields have been modified by hand over decades to maximize their agreement with experimental observations (for example, adjusting some parameters in order to reproduce a protein crystal structure) at the expense of reproducibility and predictive power.

1.2.2 Purpose and brief description of this program

Given this background, I can make the following statement. **ForceBalance aims to advance the methods of empirical potential development by applying a highly general and systematic process with explicitly specified input data and mathematical optimization algorithms, paving the way to higher accuracy potentials, improved reproducibility of potential development, and well-defined scopes of validity and error estimation for the parameters.**

At a high level, ForceBalance takes an empirical potential and a set of reference data as inputs, and tunes the parameters such that the reference data is reproduced as accurately as possible. Examples of reference data include energy and forces from high-level QM calculations, experimentally known molecular properties (e.g. polarizabilities and multipole moments), and experimentally measured bulk properties (e.g. density and dielectric constant).

ForceBalance presents the problem of potential optimization in a unified and easily extensible framework. Since there are many empirical potentials in theoretical chemistry and similarly many types of reference data, significant effort is taken to provide an infrastructure which allows a researcher to fit any type of potential to any type of reference data.

Conceptually, a set of reference data (usually a physical quantity of some kind), in combination with a method for computing the corresponding quantity with the empirical potential, is called a **fitting simulation**. For example:

- A force field can predict the density of a liquid by running NPT molecular dynamics, and this computed value can be compared against the experimental density.
- A force field can be used to evaluate the energies and forces at several molecular geometries, and these can be compared against energies and forces from higher-level quantum chemistry calculations using these same geometries. This is known as *force and energy matching*.
- A force field can predict the multipole moments and polarizabilities of a molecule isolated in vacuum, and these can be compared against experimental measurements.

Within the context of a fitting simulation, the accuracy of the force field can be optimized by tuning the parameters to minimize the difference between the computed and reference quantities. One or more fitting simulations can be combined to produce an aggregate **objective function** whose domain is the **parameter space**. This objective function, which typically depends on the parameters in a complex way, is minimized using nonlinear optimization algorithms. The result is a force field with high accuracy in all of the fitting simulations.

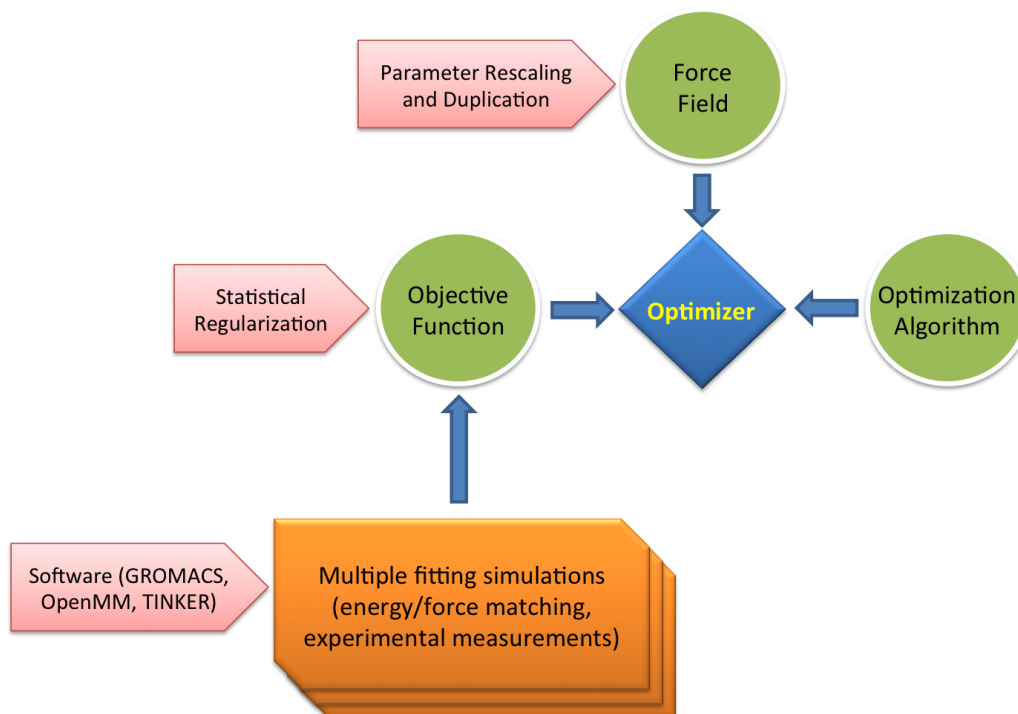


Figure 3: The division of the potential optimization problem into three parts; the force field, fitting simulations and optimization algorithm.

The problem is now split into three main components; the force field, the fitting simulations, and the optimization algorithm. ForceBalance uses this conceptual division to define three classes with minimal interdependence. Thus, if a researcher wishes to explore a new functional form, incorporate a new type of reference data or try a new optimization algorithm, he or she would only need to contribute to one branch of the program without having to restructure the entire code base.

The scientific problems and concepts that this program is based upon are further described in my Powerpoint presentations and publications, which can be found on the [SimTK website](#).

1.3 Credits

- Lee-Ping Wang is the principal developer and author.
- Troy Van Voorhis provided scientific guidance and many of the central ideas as well as financial support.
- Jiahao Chen contributed the call graph generator, the QTPIE fluctuating-charge force field (which Lee-Ping implemented into GROMACS), the interface to the MOPAC semiempirical code, and many helpful discussions.
- Matt Welborn contributed the parallelization-over-snapshots functionality in the general force matching module.
- Vijay Pande provided scientific guidance and financial support, and through the SimBios program gave this software a home on the Web at the [SimTK website](#).
- Todd Martinez provided scientific guidance and financial support.

2 Installation

This section covers how to install ForceBalance and its companion software GROMACS-X2.

Currently only Linux is supported, though installation on other Unix-based systems (e.g. Mac OS) should also be straightforward.

Importantly, note that *ForceBalance does not contain any simulation software or methods for generating the reference data*. Fitting simulations are performed by interfacing ForceBalance with simulation software like GROMACS, TINKER or OpenMM; reference data is obtained from experimental measurements (consult the literature), or from simulation / quantum chemistry software (for example, NWChem or Q-Chem).

I have provided a specialized version of GROMACS (dubbed version 4.0.7-X2) on the [SimTK website](#) which interfaces with ForceBalance through the [forceenergymatch_gmxx2](#) module. Although interfacing with unmodified simulation software is straightforward, GROMACS-X2 is optimized for our task and makes things much faster. Soon, I will also implement functions for grid-scale computation of reference energies and forces using Q-Chem (a commercial software). However, you should be prepared to write some simple code to interface with a fitting simulation or quantum chemistry software of your choice. If you choose to do so, please contact me as I would be happy to include your contribution in the main distribution.

2.1 Installing ForceBalance

ForceBalance is an ordinary Python module, so if you know how to install Python modules, you shouldn't have any trouble with this.

2.1.1 Prerequisites

The only required software for installing ForceBalance are Python and NumPy. ForceBalance also allows the usage of SciPy optimizers; they aren't as effective as the internal optimizer but still often helpful

- if you want to use these, then SciPy is needed. A few more packages are required if you want to [Create documentation](#). Here is a list of Python packages and software:

Needed for ForceBalance:

- Python version 2.7.1
- NumPy version 1.5.0
- SciPy version 0.9.0 (optional; needed for some of the non-default optimizers) Needed for making documentation:
- Doxygen version 1.7.6.1
- Doxyepy plugin for Doxygen
- LaTeX software like TeXLive

2.1.2 Installing

To install the package, first extract the tarball that you downloaded from the webpage using the command:

```
tar xvzf ForceBalance-[version].tar.gz
```

Upon extracting the distribution you will notice this directory structure:

```
<root>
+- bin
|   |- <Executable scripts>
+- forcebalance
|   |- <Python module files>
+- test
|   +- <ForceBalance example jobs>
+- doc
|   +- callgraph
|   |   |- <Stuff for making a call graph>
|   +- Images
|   |   |- <Images for the website and PDF manual>
|   |- mainpage.py (Contains most user documentation and this text)
|   |- header.tex (Customize the LaTeX documentation)
|   |- add-tabs.py (Adds more navigation tabs to the webpage)
|   |- DoxygenLayout.xml (Removes a navigation tab from the webpage)
|   |- doxygen.cfg (Main configuration file for Doxygen)
|   |- ForceBalance-Manual.pdf (PDF manual, but the one on the SimTK website is probably newer)
|- PKG-INFO (Auto-generated package information)
|- README.txt (Points to the SimTK website)
|- setup.py (Python script for installation)
```

To install the code into your default Python location, run this (you might need to be root):

```
python setup.py install
```

Alternatively, you can do a local install by running:

```
python setup.py install --prefix=/home/your_username/local_directory
```

where you would of course replace `your_username` and `local_directory` with your username and preferred install location. The executable scripts will be placed into `/home/your_username/local_directory/bin` and the module will be placed into `/home/your_username/local_directory/lib/python[version]/site-packages/forcebalance`.

Note that if you do a local installation, for Python to recognize the newly installed module you may need to append your `PYTHONPATH` environment variable using a command like the one below:

```
export PYTHONPATH=$PYTHONPATH:/home/your_username/local_directory/lib/python[version]
```

2.2 Installing GROMACS-X2

GROMACS-X2 contains major modifications from GROMACS 4.0.7. Most importantly, it enables computation of the objective function and its analytic derivatives for rapid force matching. There is also an implementation of the QTPIE fluctuating-charge polarizable force field, and the beginnings of a GROMACS/Q-Chem interface (carefully implemented but not extensively tested). Most of the changes were added in several new source files (less than ten): `qtpie.c`, `fortune.c`, `fortune_utils.c`, `fortune_vsite.c`, `fortune_nb_utils.c`, `zmatrix.c` and their corresponding header files, and `fortunerec.h` for the force matching struct. The name 'fortune' derives from back when this code was called ForTune.

The force matching functions are turned on by calling `mdrun` with the command line argument `'-fortune'`; without this option, there should be no impact on the performance of normal MD simulations.

ForceBalance interfaces with GROMACS-X2 by calling the program with special options and input files; the objective function and derivatives are computed and printed to output files. The interface is defined in `fortune.c` on the GROMACS side and `forceenergymatch_gmxx2` on the Python side. ForceBalance needs to know where the GROMACS-X2 executables are located, and this is specified using the `gmxpath` option in the input file.

2.2.1 Prerequisites for GROMACS-X2

GROMACS-X2 needs the base GROMACS requirements and several other libraries.

- FFTW version 3.3
- GLib version 2.0
- Intel MKL library

GLib is the utility library provided by the GNOME foundation (the folks who make the GNOME desktop manager and GTK+ libraries). GROMACS-X2 requires GLib for its hash table (dictionary) implementation.

GLib and FFTW can be compiled from source, but it is much easier if you're using a Linux distribution with a package manager. If you're running Ubuntu or Debian, run `sudo apt-get install libglib2.0-dev libfftw3-dev`; if you're using CentOS or some other distro with the yum package manager, run `sudo yum install glib2-devel.x86_64 fftw3-devel.x86_64` (or replace `x86_64` with `i386` if you're not on a 64-bit system).

GROMACS-X2 requires the Intel Math Kernel Library (MKL) for linear algebra. In principle this requirement can be lifted if I rewrite the source code, but it's a lot of trouble, plus MKL is faster than other implementations of BLAS and LAPACK.

The Intel MKL can be obtained from the Intel website, free of charge for noncommercial use. Currently GROMACS-X2 is built with MKL version 10.2, which ships with compiler version 11.1/072; this is not the newest version, but it can still be obtained from the Intel website after you register for a free account.

After installing these packages, extract the tarball that you downloaded from the website using the command:

```
tar xvjf gromacs-[version]-x2.tar.bz2
```

The directory structure is identical to GROMACS 4.0.7, but I added some shell scripts. `Build.sh` will run the configure script using some special options, compile the objects, create the executables and install them; you will probably need to modify it slightly for your environment. The comments in the script will help further with installation.

Don't forget to specify the install location of the GROMACS-X2 executables in the ForceBalance input file!

2.3 Create documentation

This documentation is created by Doxygen with the Doxypy plugin. To create new documentation or expand on what's here, follow the examples on the source code or visit the Doxygen home page.

To create this documentation from the source files, go to the `doc` directory in the distribution and run `doxygen doxygen.cfg` to generate the HTML documentation and LaTeX source files. Run the `add-tabs.py` script to generate the extra navigation tabs for the HTML documentation. Then go to the `latex` directory and type in `make` to build the PDF manual (You might need a LaTeX distribution for this.)

3 Usage

This page describes how to use the ForceBalance software.

A good starting point for using this software package is to run the scripts in the `bin` directory of the distribution.

`OptimizePotential.py` is the executable script that performs force field optimization. It requires an input file and a [Directory structure](#). `MakeInputFile.py` will create an example input file that contains all options, their default values, and a short description for each option. There are plans to automatically generate the correct input file from the provided directory structure, but for now the autogenerated input file only provides the hardcoded default options.

3.1 Input file

A typical input file for ForceBalance might look something like this:

```
$options
jobtype                bfgs
gmxpath                /home/leeping/opt/gromacs-4.0.7-x2/bin
forcefield              water.itp
penalty_multiplicative 0.01
convergence_objective  1e-6
convergence_step       1e-6
convergence_gradient   1e-4
$end

$simulation
simtype                forceenergymatch_gmx
name                   waterl2_gen1
weight                 1
efweight               0.5
shots                  300
fd_ptypes              VSITE
fdhessdiag             1
covariance             0
$end

$simulation
simtype                forceenergymatch_gmx
name                   waterl2_gen2
weight                 1
efweight               0.5
shots                  300
fd_ptypes              VSITE
fdhessdiag             1
covariance             0
$end
```

Global options for a ForceBalance job are given in the `$options` section while the settings for each fitting simulation are given in the `$simulation` sections. At this time, these are the only two section types.

The most important general options to note are: `jobtype` specifies the optimization algorithm to use and `forcefield` specifies the force field file name (there may be more than one of these). The most important simulation options to note are: `simtype` specifies the type of fitting simulation and `name` specifies the simulation name (must correspond to a subdirectory in `simulations/`). All options are explained in the Option Index.

3.2 Directory structure

The directory structure for our example job would look like:

```
<root>
+- forcefield
|   |- water.itp
+- simulations
|   +- water12_gen1
|   |   |- all.gro (containing 300 geometries)
|   |   |- qdata.txt
|   |   |- shot.mdp
|   |   |- topol.top
|   +- water12_gen2
|   |   |- all.gro (containing 300 geometries)
|   |   |- qdata.txt
|   |   |- shot.mdp
|   |   |- topol.top
|- input_file.in
```

The top-level directory names **forcefield** and **simulations** are fixed and cannot be changed. **forcefield** contains the force field files that you're optimizing, and **simulations** contains all of the fitting simulations and reference data. Each subdirectory in **simulations** corresponds to a single fitting simulation, and its contents depend on the specific kind of simulation and its corresponding `FittingSimulation` subclass.

The **temp** directory is the temporary workspace of the program, and the **result** directory is where the optimized force field files are deposited after the optimization job is done. These two directories are created if they're not already there.

Note the force field file, `water.itp` and the two fitting simulations `water12_gen1` and `water12_gen2` correspond to the entries in the input file. There are two energy and force matching simulations here; each directory contains the relevant geometries (in `all.gro`) and reference data (in `qdata.txt`).

4 Tutorial

This is a tutorial page, but if you haven't installed ForceBalance yet please go to the Installation page first.

It is very much in process, and there are many more examples to come.

4.1 Fitting a TIP4P potential using two fitting simulations

After everything is installed, go to the `test` directory in the distribution and run:

```
cd 001_water12_tip4p/
OptimizePotential.py 01_bfgs_from_start.in | tee my_job.out
```

If the installation was successful, you will get an output file similar to `01_bfgs_from_start.out`. `OptimizePotential.py` begins by taking the force field files from the `forcefield` directory and the fitting simulations / reference data from the `simulations` directory. Then it calls GROMACS-X2 to compute the objective function and

its derivatives, uses the internal optimizer (based on BFGS) to take a step in the parameter space, and repeats the process until convergence criteria were made.

At every step, you will see output like:

```

Step      |k|      |dk|      |grad|      --X2--      Stdev(X2)
  35      6.370e-01    1.872e-02    9.327e-02    2.48773e-01    1.149e-04

Sim: water12_sixpt   E_err(kJ/mol)=      8.8934 F_err(%)=      29.3236
Sim: water12_fourpt  E_err(kJ/mol)=     14.7967 F_err(%)=      39.2558

```

The first line reports the step number, the length of the parameter displacement vector, the gradient of the objective function, the objective function itself, and the standard deviation of the last ten *improved* steps in the objective function. There are three kinds of convergence criteria - the step size, the gradient, and the objective function itself; all of them can be specified in the input file.

The next two lines report on the two fitting simulations in this job, both of which use force/energy matching. First, note that there are two fitting simulations named `water12_sixpt` and `water12_fourpt`; the names are because one set of geometries was sampled using a six-site QTPiE force field, and the other was sampled using the TIP4P force field. However, the TIP4P force field is what we are fitting for this ForceBalance job. This shows how only one force field or parameter set is optimized for each ForceBalance job, but the method for sampling the configuration space is completely up to the user. The geometries can be seen in the `all.gro` files, and the reference data is provided in `qdata.txt`. Note that the extra virtual sites in `water12_sixpt` have been replaced with a single TIP4P site.

`E_err` and `F_err` report the RMS energy error in kJ/mol and the percentage force error; note the significant difference in the quality of agreement! This illustrates that the quality of fit depends not only on the functional form of the potential but also the configurations that are sampled. `E_err` and `F_err` are 'indicators' of our progress - that is, they are not quantities to be optimized but they give us a mental picture of how we're doing.

The other input files in the directory use the same fitting simulations, but they go through the various options of reading/writing checkpoint files, testing gradients and Hessians by finite difference, and different optimizers in SciPy. Feel free to explore some optimization jobs of your own - for example, vary the weights on the fitting simulations and see what happens. You will notice that the optimizer will try very hard to fit one simulation but not the other.

5 Glossary

This is a glossary page containing scientific concepts for the discussion of potential optimization, as well as the (automatically generated) documentation of ForceBalance keywords.

5.1 Scientific concepts

- **Empirical parameter** : Any adjustable parameter in the empirical potential that affects the potential energy, such as the partial charge on an atom, the equilibrium length of a chemical bond, or the fraction of Hartree-Fock exchange in a density functional.
- **Empirical Potential** : A formula that contains empirical parameters and computes the potential energy of a collection of atoms. Note that in ForceBalance this is used very loosely; even a DFT functional may contain many empirical parameters, and ForceBalance has the ability to optimize these as well!
- **Fitting simulation** : A simulation protocol that allows a force field to predict a physical quantity, paired with some reference data. The accuracy of the force field is given by its closeness
- **Force field** : This term is used interchangeably with empirical potential; it is more prevalent in the biomolecular simulation community.

- **Functional form** : The mathematical functions in the force field. For instance, a CHARMM-type functional form has harmonic interactions for bonds and angles, a cosine expansion for the dihedrals, Coulomb interactions between point charges and Lennard-Jones terms for van der Waals interactions.
- **Reference data** : In general, any accurately known quantity that the force field is optimized to reproduce. - Reference data can come from either theory or experiment. For instance, energies and forces from a high-level QM method can be used as reference data (for instance, a CHARMM-type force field can be fitted to reproduce forces from a DFT or MP2 calculation), or a force field can be optimized to reproduce the experimental density of a liquid, its enthalpy of vaporization or the solvation free energy of a solute.

5.2 Option index: General options

This section contains a listing of the general options available when running a ForceBalance job, which go into the \$options section. The general options are global for the ForceBalance job, in contrast to 'Simulation options' which apply to one fitting simulation within a job (described in the next section). The option index is generated by running make-option-index.py.

- **BACKUP** (Bool)
Scope : All force field optimizations (Optional)
One-line description : Write temp directories to backup before wiping them
Default Value : 1
- **CONVERGENCE_GRADIENT** (Float)
Scope : Main optimizer (Optional)
One-line description : Convergence criterion of gradient norm
Full description : The main optimizer will quit when the objective function gradient falls below this number. Since this is a newly implemented option, I can't say when this option will fail.
Default Value : 0.0001
Recommendation : Leave at the default, or set to several orders of magnitude below a typical value of the gradient (perhaps the gradient at the start of the optimization.)
- **CONVERGENCE_OBJECTIVE** (Float)
Scope : Main optimizer (Optional)
One-line description : Convergence criterion of objective function (in MainOptimizer this is the stdev of x2 over 10 steps)
Full description : The main optimizer will quit when the last ten good values of the objective function have a standard deviation that falls below this number. We use the last ten good values (instead of the latest change in the objective function), otherwise this condition would be triggered by taking tiny steps.
Default Value : 0.0001
Recommendation : Decrease this value if it's being triggered by small step sizes.
- **CONVERGENCE_STEP** (Float)
Scope : Main optimizer (Optional)
One-line description : Convergence criterion of step size (just needs to fall below this threshold)
Full description : The main optimizer will quit when the step size falls below this number. This happens if we are approaching a local minimum, or if the optimizer is constantly taking bad steps and the trust radius is reduced until it falls below this number. In the latter case, this usually means that the derivatives are wrong.

Default Value : 0.0001

Recommendation : Make sure that this value is much smaller than trust0.

- **EIG_LOWERBOUND** (Float)

Scope : Main optimizer (Optional)

One-line description : Minimum eigenvalue for applying steepest descent correction in the MainOptimizer

Full description : The main optimizer will misbehave if there are negative or very small eigenvalues in the objective function Hessian. In the former case the optimizer will travel toward a saddle point (or local maximum), and in the latter case the matrix inversion will fail because of the matrix singularity. If the smallest eigenvalue is below this value, then a multiple of the identity matrix is added to the Hessian to increase the smallest eigenvalue to at least this value.

Default Value : 0.0001

Recommendation : Shouldn't have to worry about this setting, unless the optimizer appears to be taking bad steps or inverting nearly singular matrices.

- **FFDIR** (String)

Scope : All force field optimizations (Optional)

One-line description : Directory containing force fields, relative to project directory

Default Value : forcefield

Recommendation : Unless you're using a nonstandard location for force field files, you probably shouldn't change this.

- **FINITE_DIFFERENCE_H** (Float)

Scope : fdcheck_G or fdcheck_H job types, or whenever the objective function is evaluated using finite difference (Optional)

One-line description : Step size for finite difference derivatives in many functions (get_(G/H) in fitsim, FDCheck-G)

Full description : When the objective function derivatives are checked using finite difference, or when the objective function derivative requires finite difference, this is the step size that is used (in the mathematical space). The actual parameter in the force field is changed by this amount times the rescaling factor.

Default Value : 0.01

Recommendation : 1e-2 to 1e-4; run FDCheckG to see if derivatives are accurate; if derivatives are inaccurate then adjust accordingly. If the objective function itself requires finite difference, there will still be a difference because FDCheckG(H) uses an accurate seven-point (five-point) stencil. Make sure that the derivatives agree before settling on a value to use.

- **FORCEFIELD** (List)

Scope : All force field optimizations (**Required**)

One-line description : The names of force fields, corresponding to directory forcefields/file_name.(itp|gen)

Default Value : []

- **GMXPATH** (String)

Scope : Fitting simulations that use GROMACS (GROMACS-X2 for ForceEnergyMatch_GMX) (**Required**)

One-line description : Path for GROMACS executables

Full description : Specify the path where GROMACS executables are installed, most likely ending in 'bin'. Note that executables are only installed 'bin' if the program is installed using 'make install'; this will NOT be the case if you simply ran 'make'.

Default Value : None

Recommendation : Depends on your local installation and environment.

- **GMXSUFFIX** (String)

Scope : Fitting simulations that use GROMACS (Optional)

One-line description : The suffix of GROMACS executables

Full description : Depending on how GROMACS is configured and installed, a suffix may be appended to executable names. If there is a suffix, it needs to be specified here (or else ForceBalance will not find the GROMACS executable and it will crash).

Default Value :

Recommendation : Depends on your local installation and environment.

- **JOBTYPE** (Allcap)

Scope : All force field optimizations (**Required**)

One-line description : The job type, defaults to a single-point evaluation of objective function

Full description : Here you may specify the type of ForceBalance job. This ranges from gradient-based and stochastic optimizations to simple scans over the parameter space and finite difference checking of gradients.

Default Value : sp

Recommendation : See the Optimizer class documentation for which optimizer is best suited for you.

- **MAXSTEP** (Int)

Scope : All iterative optimization jobs (Optional)

One-line description : Maximum number of steps in an optimization

Default Value : 100

Recommendation : At least 100 optimization steps are recommended.

- **PENALTY_ADDITIVE** (Float)

Scope : Objective function (Optional)

One-line description : Factor for additive penalty function in objective function

Full description : Add a penalty to the objective function (e.g. L2 or L1 norm) with this prefactor. Using an additive penalty requires an assessment of the order of magnitude of the objective function, but it is closer to the statistical concept of ridge or LASSO regression.

Default Value : 0.0

Recommendation : No recommendation; run a single-point calculation to choose a prefactor. Consider 0.01 for an objective function of order 1.

- **PENALTY_MULTIPLICATIVE** (Float)

Scope : Objective function (Optional)

One-line description : Factor for multiplicative penalty function in objective function

Full description : Multiply the objective function by (1+X) where X is this value. Using an multiplicative penalty works well for objective functions of any size but it is not equivalent to statistical regularization methods.

Default Value : 0.1

Recommendation : A value of 0.01 tends to keep the length of the parameter vector from exceeding 1.

- **PENALTY_TYPE** (String)

Scope : All force field optimizations (Optional)

One-line description : Type of the penalty, L2 or L1 in the optimizer

Full description : To prevent the optimization from changing the parameters too much, an additional penalty is applied to the objective function that depends linearly (L1) or quadratically (L2) on the norm of the parameter displacement vector. L1 corresponds to LASSO regularization while L2 is known as Tikhonov regularization or ridge regression.

Default Value : L2

Recommendation : L2; tested and known to be working. Implementation of L1 in progress.

- **READ_MVALS** (Section)

Scope : All force field optimizations (Optional)

One-line description : Paste mathematical parameters into the input file for them to be read in directly

Full description : Read in mathematical parameters before starting the optimization. There is a standardized syntax, given by:

```
read_mvals
0 [ -2.9766e-01 ] : VDWSOW
1 [  2.2283e-01 ] : VDWTOU
2 [ -1.1138e-03 ] : BONDSBHWOW
3 [ -9.0883e-02 ] : BONDSKHWOW
\read_mvals
```

Default Value : None

Recommendation : If you run the main optimizer, it will print out this block at the very end for you to use and/or modify.

- **READ_PVALS** (Section)

Scope : All force field optimizations (Optional)

One-line description : Paste physical parameters into the input file for them to be read in directly

Full description : Read in physical parameters before starting the optimization. There is a standardized syntax, given by:

```
read_pvals
0 [  2.9961e-01 ] : VDWSOW
1 [  1.2009e+00 ] : VDWTOU
2 [  9.5661e-02 ] : BONDSBHWOW
3 [  4.1721e+05 ] : BONDSKHWOW
\read_pvals
```

These are the actual numbers that go into the force field file, so note the large changes in magnitude.

Default Value : None

Recommendation : If you run the main optimizer, it will print out this block at the very end for you to use and/or modify.

- **READCHK** (String)

Scope : Main optimizer (Optional)

One-line description : Name of the restart file we read from

Full description : The main optimizer has the ability to pick up where it left off by reading / writing checkpoint files. Here you may specify the checkpoint file to read in from a previous optimization run. This is equivalent to reading in stored parameter values, except the gradient and Hessian (which contains memory from previous steps) is recorded too.

Default Value : None

- **SCAN_VALS** (String)

Scope : scan_mvals and scan_pvals job types (Optional)

One-line description : Values to scan in the parameter space for job type "scan[mp]vals", given like this: -0.1-:0.01:0.1

Full description : This specifies a range of parameter values to scan in a uniform grid. scan_mvals works in the mathematical parameter space while scan_pvals works in the physical parameter space. The syntax is lower:step:upper . Both lower and upper limits are included in the range.

Default Value : None

Recommendation : For scan_mvals, a range of values between -1 and +1 is recommended; for scan_pvals, choose values close to the physical parameter value.

- **SCANINDEX_NAME** (List)

Scope : scan_mvals and scan_pvals job types (Optional)

One-line description : Parameter name to scan over (should convert to a numerical index) in job type "scan[mp]vals"

Full description : ForceBalance assigns to each adjustable parameter a 'parameter name'. By specifying this option, this tells the parameter scanner to locate the correct parameter with the specified name and then scan over it.

Default Value : []

Recommendation : Look at the printout from a single-point job to determine the parameter names.

- **SCANINDEX_NUM** (List)

Scope : scan_mvals and scan_pvals job types (Optional)

One-line description : Numerical index of the parameter to scan over in job type "scan[mp]vals"

Full description : ForceBalance assigns to each adjustable parameter a 'parameter number' corresponding to its position in the parameter vector. This tells the parameter scanner which number to scan over.

Default Value : []

Recommendation : Look at the printout from a single-point job to decide which parameter number you wish to scan over.

- **TINKERPATH** (String)

Scope : Fitting simulations that use TINKER (**Required**)

One-line description : Path for TINKER executables

Default Value : None

Recommendation : Depends on your local installation and environment.

- **TRUST0** (Float)

Scope : Main optimizer (Optional)

One-line description : Trust radius for the MainOptimizer

Full description : The main optimizer uses a trust radius which 'adapts' (i.e. increases or decreases) based on whether the last step was a good or bad step. 'trust0' provides the starting trust radius, and the trust radius is not allowed to increase too much from trust0.

Default Value : 0.01

Recommendation : Increase from the default if the optimizer takes many good steps but takes too long; decrease if the optimizer takes many bad steps.

- **WRITECHK** (String)

Scope : Main optimizer (Optional)

One-line description : Name of the restart file we write to (can be same as readchk)

Full description : The main optimizer has the ability to pick up where it left off by reading / writing checkpoint files. Here you may specify the checkpoint file to write after the job is finished.

Default Value : None

Recommendation : Writing the checkpoint file is highly recommended.

- **WRITECHK_STEP** (Bool)

Scope : Main optimizer when 'writechk' is turned on (Optional)

One-line description : Write the checkpoint file at every optimization step

Full description : Write a checkpoint file every single step, not just after the job is finished.

Default Value : 0

Recommendation : Useful if you want to quit an optimization before it finishes and restart, but make sure you don't overwrite existing checkpoint files by accident.

5.3 Option index: Simulation options

This section contains a listing of the simulation options available when running a ForceBalance job, which go into the \$sim_opts section. There can be multiple \$sim_opts sections in a ForceBalance input file, one for each fitting simulation.

- **BATCH_FD** (Bool)

Scope : All fitting simulations (Optional)

One-line description : Whether to batch and queue up finite difference jobs, defaults to False

Full description : This is a stub for future functionality. When the flag is switched on, the jobs corresponding to finite difference derivatives are evaluated in parallel on a distributed computing platform.

Default Value : 0

- **COVARIANCE** (Bool)

Scope : Force and energy matching simulations (Optional)

One-line description : Whether to use the quantum covariance matrix (force+energy match), defaults to True

Full description : The components of the energy and force contribution to the objective function are rescaled to be on the same footing when the objective function is optimized. This can be done by dividing each component by its variance, or by multiplying the energy-force polytensor by the inverse of the quantum energy-force covariance matrix. The latter method was proposed as a way to emphasize intermolecular interactions but it is unproven.

Default Value : 1

Recommendation : No recommendation; turn the covariance off if the number of snapshots is not much larger than the number of coordinates.

- **EFWEIGHT** (Float)

Scope : Force and energy matching simulations (Optional)

One-line description : 1.0 for all energy and 0.0 for all force (force+energy match), defaults to 0.5

Full description : Energies and forces are evaluated together in a force/energy matching simulation, and this option specifies the relative weight of the energy and force contributions.

Default Value : 0.5

Recommendation : Based on experience, it should be okay to leave this option at its default value, unless you wish to emphasize only the force (then choose 0.0) or only the energy (then choose 1.0).

- **FD_PTYPES** (List)

Scope : All fitting simulations (Optional)

One-line description : The parameter types that need to be differentiated using finite difference

Full description : To compute the objective function derivatives, some components may require numerical finite difference in the derivatives. Here you may specify the parameter types that finite difference is applied to, or write 'ALL' to take finite-difference derivatives in all parameter types.

Default Value : []

Recommendation : If you aren't sure, either use 'ALL' to do finite difference in each component (this is costly), or run a fdcheckG(H) job with this option set to 'NONE' to check which analytic derivatives are missing. Usually analytic derivatives will be missing in anything but FORCEENERGYMATCH_GMXX2 jobs.

- **FDGRAD** (Bool)

Scope : All fitting simulations (Optional)

One-line description : Finite difference gradients

Full description : When this option is enabled, finite difference gradients will be enabled for selected parameter types (using the fd_ptypes option). Gradients are computed using two-point finite difference of the objective function.

Default Value : 1

Recommendation : If analytic derivatives are implemented (and correct), then they are much faster than finite difference derivatives. Run the 'fdcheckG' routine with this option set to Off to check which finite difference derivatives you need.

- **FDHESS** (Bool)

Scope : All fitting simulations (Optional)

One-line description : Finite difference Hessian diagonals (costs np times a gradient calculation)

Full description : When this option is enabled, finite difference Hessians will be enabled for selected parameter types (using the fd_ptypes option). Hessians are computed using two-point finite difference of the gradient.

Default Value : 1

Recommendation : Run the 'fdcheckH' routine with this option set to Off to check which finite difference Hessian elements you need. Note that this requires a very large number of objective function evaluations, so use sparingly.

- **FDHESSDIAG** (Bool)

Scope : All fitting simulations (Optional)

One-line description : Finite difference Hessian diagonals (cheap; costs 2np times a objective calculation)

Full description : When this option is enabled, finite difference gradients and Hessian diagonal elements will be enabled for selected parameter types (using the fd_ptypes option). This is done using a three-point finite difference of the objective function.

Default Value : 1

Recommendation : Use this as a substitute for 'fdgrad'; it doubles the cost but provides more accurate derivatives plus the Hessian diagonal values (these are very nice for quasi-Newton optimizers like BFGS).

- **FITATOMS** (Int)

Scope : Force and energy matching simulations (Optional)

One-line description : Number of fitting atoms (force+energy matching); defaults to all of them

Full description : Choose a subset of atoms from the force matching simulation to fit forces to. This is useful in situations where it is undesirable to fit the forces on part of the system (e.g. the part that is described by another

force field.) Currently, you are only allowed to choose from the atoms in the front of the trajectory; soon this will be expanded for random flexibility (see 'shots'). However, random coordinate selections are not allowed. ;)

Default Value : 0

Recommendation : Situation-dependent; this should be based on the part of the simulation that you're fitting, or leave blank if you're fitting the whole system.

- **NAME** (String)

Scope : All fitting simulations (**Required**)

One-line description : The name of the simulation, which corresponds to the directory simulations/dir_name

Default Value : None

Recommendation : Choose a descriptive name and make sure all fitting simulations have different names.

- **QMBOLTZ** (Float)

Scope : Force and energy matching simulations (Optional)

One-line description : Fraction of Quantum Boltzmann Weights (force+energy match), 1.0 for full reweighting, $0.0 < 1.0$ for hybrid

Full description : When Boltzmann sampling is used to gather snapshots for force/energy matching, there is a potential ambiguity regarding which ensemble one should sample from (either the force field's ensemble or the QM calculation's ensemble. The QM ensemble may be sampled using MM-sampled snapshots by reweighting; this tuning parameter specifies the fraction of QM Boltzmann weight to include. Note that when two ensembles are different, reweighting will decrease the statistical significance of the number of snapshots (i.e. there is less InfoContent).

Default Value : 0.0

Recommendation : If you want to reweight your snapshots entirely to the QM ensemble, choose 1.0; for hybrid weights, use 0.5. Avoid if the fitting simulation has a very large RMS energy difference between QM and MM.

- **QMBOLTZTEMP** (Float)

Scope : Force and energy matching simulations (Optional)

One-line description : Temperature for Quantum Boltzmann Weights (force+energy match), defaults to room temperature

Full description : The reweighting of an ensemble involves an exponential of $(DE)/kT$, so there is a massive degradation of sample quality if (DE) is large. This option allows you to change the temperature in the denominator, which is unphysical (but it does decrease the effect of moving toward the QM ensemble).

Default Value : 298.15

Recommendation : Irrelevant if 'qmboltz' is set to zero. Leave at the default value unless you're performing experiments.

- **SAMPCORR** (Bool)

Scope : Force and energy matching simulations that use GROMACS-X2 (Optional)

One-line description : Whether to use the (archaic) sampling correction (force+energy match), defaults to False

Full description : Every time the force field parameters are updated, the ensemble is different. In principle this applies to not only the self-consistent optimization cycles (which include re-running dynamics, QM calculations etc) but also the numerical optimization itself. When this option is turned on, the Boltzmann weights of the snapshots are updated in every step of the optimization and the derivatives are modified accordingly. My investigations reveal that this makes the force field more accurate in energy minima and less accurate for barriers, which was not very useful. I haven't touched the 'sampling corrected' code in a long time; thus this option is vestigial and may be removed in the future.

Default Value : 0

Recommendation : Off.

- **SHOTS** (Int)

Scope : Force and energy matching simulations (Optional)

One-line description : Number of snapshots (force+energy matching); defaults to all of the snapshots

Full description : This option allows you to choose a subset from the snapshots available in the force matching 'simulations' directory. The subset is simply taken from the front of the trajectory. In the future this option will be expanded to allow a random selection of snapshots, or a specific selection

Default Value : -1

Recommendation : 100-10,000 snapshots are recommended. Note that you need at least 3x (number of atoms) if the covariance matrix is turned on.

- **SIMTYPE** (Allcap)

Scope : All fitting simulations (**Required**)

One-line description : The type of fitting simulation, for instance ForceEnergyMatch_GMXX2

Full description : This is the type of fitting simulation that you are running. The current accepted values for the fitting simulation are given in the SimTab.py file: FORCEENERGYMATCH_TINKER, FORCEENERGYMATCH_GMX, COUNTERPOISEMATCH, FORCEENERGYMATCH_GMXX2.

Default Value : None

Recommendation : Choose the appropriate type, and if the fitting simulation is missing, feel free to implement your own (or ask me for help).

- **USE_PVALS** (Bool)

Scope : All fitting simulations (Optional)

One-line description : Bypass the transformation matrix and use the physical parameters directly

Full description : When this option is enabled, the coordinate transformation in parameter space will be bypassed, and parameters passed into the 'get' subroutines will be plugged directly into the force field files. This option is turned on automatically if we are running a 'scan_pvals' job. Note that the coordinate transformation is essential in multi-parameter optimizations and preserves the condition number of the Hessian, so turning it off should generally be avoided.

Default Value : 0

Recommendation : This option should almost always be off unless the user really knows what he/she is doing.

- **WEIGHT** (Float)

Scope : All fitting simulations (Optional)

One-line description : Weight of the current simulation (with respect to other simulations)

Full description : This option specifies the weight that the fitting simulation will contribute to the objective function. A larger weight for a given fitting simulation means that the optimizer will prioritize it over the others. When several fitting simulations are used, the weight should be chosen carefully such that all fitting simulations contribute a finite amount to the objective function. Note that the choice of weight determines the final outcome of the force field, although we hope not by too much.

Default Value : 1.0

Recommendation : It is important to specify something here (giving everything equal weight is unlikely to work.) Run a single-point objective function evaluation with all weights set to one to get a handle on the natural size of each fitting simulation's contribution, and then add weights accordingly.

- **WHAMBOLTZ** (Bool)

Scope : Force and energy matching simulations (Optional)

One-line description : Whether to use WHAM Boltzmann Weights (force+energy match), defaults to False

Full description : In self-consistent energy/force matching projects, the data from previous cycles can be reused by applying the Weighted Histogram Analysis Method (WHAM). However, the WHAM data is currently generated by external scripts that haven't made it into this distribution yet. In the future, generation of WHAM data will be incorporated into this program automatically.

Default Value : 0

Recommendation : Leave off unless you have an externally generated wham-master.txt and wham-weights.txt files.



Figure 4: Logo.

6 Todo List

Member [forcebalance.counterpoisematch.CounterpoiseMatch.loadxyz](#)

I should probably put this into a more general library for reading coordinates.

Member [forcebalance.fitsim.FittingSimulation.get](#)

Write documentation here later.

Member [forcebalance.forceenergymatch.ForceEnergyMatch.__init__](#)

Obtain the number of true atoms (or the particle -> atom mapping) from the force field.

Member [forcebalance.forceenergymatch.ForceEnergyMatch.get](#)

Parallelization over snapshots is not implemented yet

Member [forcebalance.forceenergymatch.ForceEnergyMatch.read_reference_data](#)

Add an option for picking any slice out of qdata.txt, helpful for cross-validation

Closer integration of reference data with program - leave behind the qdata.txt format? (For now, I like the readability of qdata.txt)

The WHAM Boltzmann weights are generated by external scripts (wanalyze.py and make-wham-data.sh) and passed in; perhaps these scripts can be added to the ForceBalance distribution or integrated more tightly.

Member [forcebalance.forceenergymatch_gmxx2.ForceEnergyMatch_GMXX2.get](#)

Some of these files don't need to be printed, they can be passed to GROMACS as arguments. Let's think about this some more.

Currently I have no way to pass out the qualitative indicators.

Member [forcebalance.forceenergymatch_gmxx2.ForceEnergyMatch_GMXX2.prepare_temp_directory](#)

Someday I'd like to use WHAM to put AIMD simulations in. :)

The fitatoms shouldn't be the first however many atoms, it should be a list.

Member [forcebalance.forcefield.FF.addff](#)

Note that I can also create the opposite virtual site position by changing the atom labeling, woo!

Member [forcebalance.forcefield.FF.mktransmat](#)

Only project out changes in total charge of a molecule, and perhaps generalize to fragments of molecules or other types of parameters.

Member [forcebalance.forcefield.FF.rsmake](#)

Pass in rsfactors through the input file

Member [forcebalance.optimizer.Optimizer.Scan_Values](#)

Maybe a multidimensional grid can be done.

Member [forcebalance.tinkerio.Tinker_Reader.feed](#)

Put the rescaling factors for TINKER parameters in here. Currently we're using the initial value to determine the rescaling factor which is not very good.

Namespace [gmxxio](#)

Even more stuff from [forcefield.py](#) needs to go into here.

7 Namespace Index

7.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

[BaseReader](#)

Base class for force field line reader

24

[counterpoisematch](#)

Match an empirical potential to the counterpoise correction for basis set superposition error (BS-SE)

25

[custom_io](#)

Custom force field parser

25

[forceenergymatch_gmxx2](#)

Force and energy matching module

25

forcefield	
Force field module	26
gmio	
GROMACS input/output	27
MakeInputFile	
Executable script for printing out an example input file with defaults and documentation	28
molecule	
Lee-Ping's convenience library for parsing molecule file formats	28
nifty	
Nifty functions, intended to be imported by any module	28
OptimizePotential	
Executable script for starting ForceBalance	29
optimizer	
Optimization algorithms	29
ParseInputFile	
Read in ForceBalance input file and print it back out	30
parser	
Input file parser for ForceBalance projects	30
project	
ForceBalance force field optimization project	31
qchemio	
Q-Chem input file parser	31
simtab	
Contains the dictionary of fitting simulation classes	31
tinkerio	
TINKER input/output	32

8 Class Index

8.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

forcebalance.basereader.BaseReader	32
forcebalance.counterpoisematch.CounterpoiseMatch	
FittingSimulation subclass for matching the counterpoise correction	33
forcebalance.forcefield.FF	
Force field class	36
forcebalance.fitsim.FittingSimulation	
Base class for all fitting simulations	42

forcebalance.forceenergymatch.ForceEnergyMatch	
Subclass of FittingSimulation for force and energy matching	46
forcebalance.gmxio.ForceEnergyMatch_GMX	
Subclass of FittingSimulation for force and energy matching using normal GROMACS	50
forcebalance.forceenergymatch_gmxx2.ForceEnergyMatch_GMXX2	
ForceBalance class for force and energy matching with the modified GROMACS	52
forcebalance.tinkerio.ForceEnergyMatch_TINKER	
Subclass of FittingSimulation for force and energy matching using TINKER	55
forcebalance.custom_io.Gen_Reader	
Finite state machine for parsing custom GROMACS force field files	56
forcebalance.gmxio.OTP_Reader	
Finite state machine for parsing GROMACS force field files	58
forcebalance.molecule.Molecule	
The Molecule class contains information about a system of molecules	60
CallGraph.node	
Data structure for holding information about python objects	63
forcebalance.optimizer.Optimizer	
Optimizer class	64
forcebalance.project.Project	
Container for a ForceBalance force field optimization project	72
forcebalance.qchemio.QCIn_Reader	
Finite state machine for parsing Q-Chem input files	74
forcebalance.tinkerio.Tinker_Reader	
Finite state machine for parsing TINKER force field files	76

9 Namespace Documentation

9.1 BaseReader Namespace Reference

Base class for force field line reader.

9.1.1 Detailed Description

Base class for force field line reader.

Author

Lee-Ping Wang

Date

12/2011

9.2 counterpoisematch Namespace Reference

Match an empirical potential to the counterpoise correction for basis set superposition error (BSSE).

9.2.1 Detailed Description

Match an empirical potential to the counterpoise correction for basis set superposition error (BSSE). Here we test two different functional forms: a three-parameter Gaussian repulsive potential and a four-parameter Gaussian which goes smoothly to an exponential. The latter can be written in two different ways - one which gives us control over the exponential, the switching distance and the Gaussian decay constant, and another which gives us control over the Gaussian and the switching distance. They are called 'CPGAUSS', 'CPEXPG', and 'CPGEXP'. I think the third option is the best although our early tests have indicated that none of the force fields perform particularly well for the water dimer.

This subclass of FittingSimulation implements the 'get' method.

Author

Lee-Ping Wang

Date

12/2011

9.3 custom_io Namespace Reference

Custom force field parser.

9.3.1 Detailed Description

Custom force field parser. We take advantage of the sections in GROMACS and the 'interaction type' concept, but these interactions are not supported in GROMACS; rather, they are computed within our program.

Author

Lee-Ping Wang

Date

12/2011

9.4 forceenergymatch_gmxx2 Namespace Reference

Force and energy matching module.

9.4.1 Detailed Description

Force and energy matching module. Force and energy matching with interface to modified GROMACS.

Author

Lee-Ping Wang

Date

12/2011

In order for us to obtain the objective function in force and energy matching, we loop through the snapshots, compute the energy and force (as well as its derivatives), and sum them up. The details of the process are complicated and I won't document them here. The contents of this package (mainly the ForceEnergyMatch_GMXX2 class) allows us to call the modified GROMACS to compute the objective function for us.

Author

Lee-Ping Wang

Date

12/2011

9.5 forcefield Namespace Reference

Force field module.

9.5.1 Detailed Description

Force field module. In ForceBalance a 'force field' is built from a set of files containing physical parameters. These files can be anything that enter into any computation - our original program was quite dependent on the GROMACS force field format, but this program is set up to allow very general input formats.

We introduce several important concepts:

1) Adjustable parameters are allocated into a vector.

To cast the force field optimization as a math problem, we treat all of the parameters on equal footing and write them as indices in a parameter vector.

2) A mapping from interaction type to parameter number.

Each element in the parameter vector corresponds to one or more interaction types. Whenever we change the parameter vector and recompute the objective function, this amounts to changing the physical parameters in the simulations, so we print out new force field files for external programs. In addition, when these programs are computing the objective function we are often in low-level subroutines that compute terms in the energy and force. If we need an analytic derivative of the objective function, then these subroutines need to know which index of the parameter vector needs to be modified.

This is done by way of a hash table: for example, when we are computing a Coulomb interaction between atom 4 and atom 5, we can build the words 'COUL4' and 'COUL5' and look it up in the parameter map; this gives us two numbers (say, 10 and 11) corresponding to the eleventh and twelfth element of the parameter vector. Then we can compute the derivatives of the energy w/r.t. these parameters (in this case, COUL5/rij and COUL4/rij) and increment these values in the objective function gradient.

In custom-implemented force fields (see [counterpoisematch.py](#)) the hash table can also be used to look up parameter values for computation of interactions. This is probably not the fastest way to do things, however.

3) Distinction between physical and mathematical parameters.

The optimization algorithm works in a space that is related to, but not exactly the same as the physical parameter space. The reasons for why we do this are:

- a) Each parameter has its own physical units. On the one hand it's not right to treat different physical units all on the same footing, so nondimensionalization is desirable. To make matters worse, the force field parameters can be small as 1e-8 or as large as 1e+6 depending on the parameter type. This means the elements of the objective function gradient / Hessian have elements that differ from each other in size by 10+ orders of magnitude, leading to mathematical instabilities in the optimizer.
- b) The parameter space can be constrained, most notably for atomic partial charges where we don't want to change the overall charge on a molecule. Thus we wish to project out certain movements in the mathematical parameters such that they don't change the physical parameters.
- c) We wish to regularize our optimization so as to avoid changing our parameters in very insensitive directions (linear dependencies). However, the sensitivity of the objective function to changes in the force field depends on the physical units!

For all of these reasons, we introduce a 'transformation matrix' which maps mathematical parameters onto physical parameters. The diagonal elements in this matrix are rescaling factors; they take the mathematical parameter and magnify it by this constant factor. The off-diagonal elements correspond to rotations and other linear transformations, and currently I just use them to project out the 'increase the net charge' direction in the physical parameter space.

Note that with regularization, these rescaling factors are equivalent to the widths of prior distributions in a maximum likelihood framework. Because there is such a correspondence between rescaling factors and choosing a prior, they need to be chosen carefully. This is work in progress. Another possibility is to sample the width of the priors from a noninformative distribution -- the hyperprior (we can choose the Jeffreys prior or something). This is work in progress.

Right now only GROMACS parameters are supported, but this class is extensible, we need more modules!

Author

Lee-Ping Wang

Date

12/2011

9.6 gmzio Namespace Reference

GROMACS input/output.

9.6.1 Detailed Description

GROMACS input/output.

Todo Even more stuff from [forcefield.py](#) needs to go into here.

Author

Lee-Ping Wang

Date

12/2011

9.7 MakeInputFile Namespace Reference

Executable script for printing out an example input file with defaults and documentation.

Functions

- def [main](#)

Print out all of the options available to ForceBalance.

9.7.1 Detailed Description

Executable script for printing out an example input file with defaults and documentation. At the current stage, this script simply prints out all of the default options, but in the future we may want to autogenerate the input file. This would make everyone's lives much easier, don't you think? :)

9.7.2 Function Documentation

9.7.2.1 def MakeInputFile.main ()

Print out all of the options available to ForceBalance.

Definition at line 18 of file MakeInputFile.py.

9.8 molecule Namespace Reference

Lee-Ping's convenience library for parsing molecule file formats.

9.8.1 Detailed Description

Lee-Ping's convenience library for parsing molecule file formats. Motivation: I'm not quite sure what OpenBabel does, don't fully trust it I need to be very careful with Q-Chem input files and use input templates I'd like to replace the grordr module because it's so old.

Currently i can read xyz, gro, com, and arc Currently i can write xyz, gro, qcin (with Q-Chem template file)

Not sure what's next on the chopping block, but please stay consistent when adding new methods.

Author

Lee-Ping Wang

Date

12/2011

9.9 nifty Namespace Reference

Nifty functions, intended to be imported by any module.

9.9.1 Detailed Description

Nifty functions, intended to be imported by any module. Named after the mighty Sniffy Handy Nifty (King Sniffy)

Author

Lee-Ping Wang

Date

12/2011

9.10 OptimizePotential Namespace Reference

Executable script for starting ForceBalance.

Functions

- `def main`
Instantiate a ForceBalance project and call the optimizer.

9.10.1 Detailed Description

Executable script for starting ForceBalance.

9.10.2 Function Documentation

9.10.2.1 `def OptimizePotential.main ()`

Instantiate a ForceBalance project and call the optimizer.

Definition at line 13 of file OptimizePotential.py.

9.11 optimizer Namespace Reference

Optimization algorithms.

9.11.1 Detailed Description

Optimization algorithms. My current implementation is to have a single optimizer class with several methods contained inside.

Author

Lee-Ping Wang

Date

12/2011

9.12 ParseInputFile Namespace Reference

Read in ForceBalance input file and print it back out.

Functions

- def [main](#)
Input file parser for ForceBalance program.

9.12.1 Detailed Description

Read in ForceBalance input file and print it back out.

9.12.2 Function Documentation

9.12.2.1 def ParseInputFile.main ()

Input file parser for ForceBalance program.

We will simply read the options and print them back out.

Definition at line 14 of file ParseInputFile.py.

9.13 parser Namespace Reference

Input file parser for ForceBalance projects.

9.13.1 Detailed Description

Input file parser for ForceBalance projects. Additionally, the location for all default options.

Although I will do my best to write good documentation, for many programs the input parser becomes the most up-to-date source for documentation. So this is a great place to write lots of comments for those who implement new functionality.

Basically, the way my program is structured there is a set of GENERAL options and also a set of SIMULATION options. Since there can be many fitting simulations within a single project (i.e. we may wish to fit water trimers and hexamers, which constitutes two fitting simulations) the input is organized into sections, like so:

\$options

gen_option_1 Big

gen_option_2 Mao

\$simulation

sim_option_1 Sniffy

sim_option_2 Schmao

\$simulation

sim_option_1 Nifty

sim_option_2 Jiffy

\$end

In this case, two sets of simulation options are generated in addition to the general option.

Each option is meant to be parsed as a certain variable type.

- String option values are read in directly; note that only the first two words in the line are processed
- Some strings are capitalized when they are read in; this is mainly for function tables like OptTab and SimTab
- List option types will pick up all of the words on the line and use them as values, plus if the option occurs more than once it will aggregate all of the values.
- Integer and float option types are read in a pretty straightforward way
- Boolean option types are always set to true, unless the second word is '0', 'no', or 'false' (not case sensitive)
- Section option types are meant to treat more elaborate inputs, such as the user pasting in output parameters from a previous job as input, or a specification of internal coordinate system. I imagine that for every section type I would have to write my own parser. Maybe a ParsTab of parsing functions would work. :)

To add a new option, simply add it to the dictionaries below and give it a default value if desired. If you add an entirely new type, make sure to implement the interpretation of that type in the parse_inputs function.

Author

Lee-Ping Wang

Date

12/2011

9.14 project Namespace Reference

ForceBalance force field optimization project.

9.14.1 Detailed Description

ForceBalance force field optimization project.

9.15 qchemio Namespace Reference

Q-Chem input file parser.

9.15.1 Detailed Description

Q-Chem input file parser.

9.16 simtab Namespace Reference

Contains the dictionary of fitting simulation classes.

9.16.1 Detailed Description

Contains the dictionary of fitting simulation classes. This is in a separate file to facilitate importing. I would happily put it somewhere else.

9.17 tinkerio Namespace Reference

TINKER input/output.

9.17.1 Detailed Description

TINKER input/output. This serves as a good template for writing future force matching I/O modules for other programs because it's so simple.

Author

Lee-Ping Wang

Date

01/2012

10 Class Documentation

10.1 forcebalance.basereader.BaseReader Class Reference

Public Member Functions

- def **__init__**
- def **feed**
- def **build_pid**

Returns the parameter type (e.g.

Public Attributes

- **In**
- **itype**
- **suffix**
- **pdict**

10.1.1 Detailed Description

Definition at line 8 of file basereader.py.

10.1.2 Member Function Documentation

10.1.2.1 `def forcebalance.basereader.BaseReader.build_pid (self, pflid)`

Returns the parameter type (e.g.

K in BONDSK) based on the current interaction type.

Both the 'pdict' dictionary (see `gmxi.pdict`) and the interaction type 'state' (here, BONDS) are needed to get the parameter type.

If, however, 'pdict' does not contain the ptype value, a suitable substitute is simply the field number.

Note that if the interaction type state is not set, then it defaults to the file name, so a generic parameter ID is 'filename.-line_num.field_num'

Definition at line 34 of file `basereader.py`.

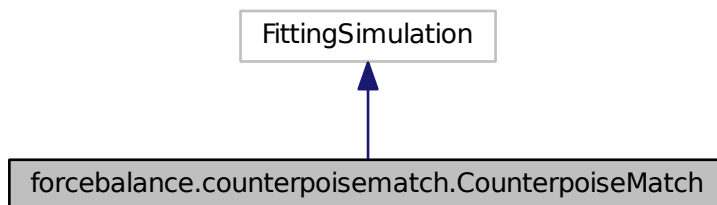
The documentation for this class was generated from the following file:

- `forcebalance/forcebalance/basereader.py`

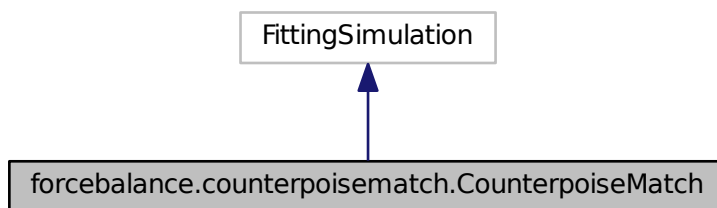
10.2 forcebalance.counterpoisematch.CounterpoiseMatch Class Reference

FittingSimulation subclass for matching the counterpoise correction.

Inheritance diagram for `forcebalance.counterpoisematch.CounterpoiseMatch`:



Collaboration diagram for forcebalance.counterpoisematch.CounterpoiseMatch:



Public Member Functions

- `def __init__`
To instantiate [CounterpoiseMatch](#), we read the coordinates and counterpoise data.
- `def loadxyz`
Parse an XYZ file which contains several xyz coordinates, and return their elements.
- `def load_cp`
Load in the counterpoise data, which is easy; the file consists of floating point numbers separated by newlines.
- `def get`
Gets the objective function for fitting the counterpoise correction.

Public Attributes

- `ns`
Number of snapshots.
- `xyzs`
XYZ elements and coordinates.
- `cpqm`
Counterpoise correction data.
- `na`
Number of atoms.

10.2.1 Detailed Description

FittingSimulation subclass for matching the counterpoise correction.

Definition at line 31 of file counterpoisematch.py.

10.2.2 Constructor & Destructor Documentation

10.2.2.1 `def forcebalance.counterpoisematch.CounterpoiseMatch.__init__(self, options, sim_opts, forcefield)`

To instantiate [CounterpoiseMatch](#), we read the coordinates and counterpoise data.

Definition at line 35 of file counterpoisematch.py.

10.2.3 Member Function Documentation

10.2.3.1 `def forcebalance.counterpoisematch.CounterpoiseMatch.get (self, mvals, AGrad=False, AHess=False, tempdir=None)`

Gets the objective function for fitting the counterpoise correction.

As opposed to ForceEnergyMatch_GMXX2, which calls an external program, this script actually computes the empirical interaction given the force field parameters.

It loops through the snapshots and atom pairs, and computes pairwise contributions to an energy term according to hard-coded functional forms.

One potential issue is that we go through all atom pairs instead of looking only at atom pairs between different fragments. This means that even for two infinitely separated fragments it will predict a finite CP correction. While it might be okay to apply such a potential in practice, there will be some issues for the fitting. Thus, we assume the last snapshot to be CP-free and subtract that value of the potential back out.

Note that forces and parametric derivatives are not implemented.

Parameters

in	<i>mvals</i>	Mathematical parameter values
in	<i>AGrad</i>	Switch to turn on analytic gradient (not implemented)
in	<i>AHess</i>	Switch to turn on analytic Hessian (not implemented)
in	<i>tempdir</i>	Temporary directory for running computation

Returns

Answer Contribution to the objective function

Definition at line 123 of file counterpoisematch.py.

10.2.3.2 `def forcebalance.counterpoisematch.CounterpoiseMatch.load_cp (self, fnm)`

Load in the counterpoise data, which is easy; the file consists of floating point numbers separated by newlines.

Definition at line 93 of file counterpoisematch.py.

10.2.3.3 `def forcebalance.counterpoisematch.CounterpoiseMatch.loadxyz (self, fnm)`

Parse an XYZ file which contains several xyz coordinates, and return their elements.

Parameters

in	<i>fnm</i>	The input XYZ file name
----	------------	-------------------------

Returns

elem A list of chemical elements in the XYZ file

xyzs A list of XYZ coordinates (number of snapshots times number of atoms)

Todo I should probably put this into a more general library for reading coordinates.

Definition at line 61 of file counterpoisematch.py.

The documentation for this class was generated from the following file:

- forcebalance/forcebalance/counterpoisematch.py

10.3 forcebalance.forcefield.FF Class Reference

Force field class.

Public Member Functions

- def `__init__`
Instantiation of force field class.
- def `addff`
Parse a force field file and add it to the class.
- def `make`
Create a new force field using provided parameter values.
- def `create_pvals`
Converts mathematical to physical parameters.
- def `create_mvals`
Converts physical to mathematical parameters.
- def `replace_pvals`
Replaces numerical fields in stored force field files with the stored physical parameter values.
- def `print_newstuff`
Prints out the new content of force fields to files in 'printdir'.
- def `rsmake`
Create the rescaling factors for the coordinate transformation in parameter space.
- def `mktransmat`
Create the transformation matrix to rescale and rotate the mathematical parameters.
- def `list_map`
Create the plist, which is like a reversed version of the parameter map.
- def `print_map`
Prints out the (physical or mathematical) parameter indices, IDs and values in a visually appealing way.
- def `assign_p0`
Assign physical parameter values to the 'pvals0' array.
- def `assign_field`
Record the locations of a parameter in a file; [[file name, line number, field number, and multiplier]].

Public Attributes

- `root`
The root directory of the project.
- `fnms`
File names of force fields.
- `ffdir`
Directory containing force fields, relative to project directory.
- `stuff`

The content of all force field files are stored in memory.

- [map](#)

The mapping of interaction type -> parameter number.

- [plist](#)

The listing of parameter number -> interaction types.

- [pfields](#)

A list where pfields[pnum] = ['file',line,field,mult], basically a new way to modify force field files; when we modify the force field file, we go to the specific line/field in a given file and change the number.

- [rs](#)

List of rescaling factors.

- [tm](#)

The transformation matrix for mathematical -> physical parameters.

- [tml](#)

The transpose of the transformation matrix.

- [excision](#)

Indices to exclude from optimization / Hessian inversion.

- [np](#)

The total number of parameters.

- [newstuff](#)

The force field content, but with parameter fields replaced with new parameters.

- [pvals0](#)

Initial value of physical parameters.

- [R](#)

10.3.1 Detailed Description

Force field class.

This class contains all methods for force field manipulation. To create an instance of this class, an input file is required containing the list of force field file names. Everything else inside this class pertaining to force field generation is self-contained.

For details on force field parsing, see the detailed documentation for addff.

Definition at line 159 of file forcefield.py.

10.3.2 Constructor & Destructor Documentation

10.3.2.1 `def forcebalance.forcefield.FF.__init__(self, options)`

Instantiation of force field class.

Many variables here are initialized to zero, but they are filled out by methods like addff, rsmake, and mktransmat.

Definition at line 167 of file forcefield.py.

10.3.3 Member Function Documentation

10.3.3.1 `def forcebalance.forcefield.FF.addff(self, fname)`

Parse a force field file and add it to the class.

First, we need to figure out the type of file file. Currently this is done using the three-letter file extension ('.itp' = gmx); that can be improved.

First we open the force field file and read all of its lines. As we loop through the force field file, we look for two types of tags: (1) section markers, in GMX indicated by [section_name], which allows us to determine the section, and (2) parameter tags, indicated by the 'PARM' or 'RPT' keywords.

As we go through the file, we figure out the atoms involved in the interaction described on each line.

When a 'PARM' keyword is indicated, it is followed by a number which is the field in the line to be modified, starting with zero. Based on the field number and the section name, we can figure out the parameter type. With the parameter type and the atoms in hand, we construct a 'parameter identifier' or pid which uniquely identifies that parameter. We also store the physical parameter value in an array called 'pvals0' and the precise location of that parameter (by filename, line number, and field number) in a list called 'pfields'.

An example: Suppose in 'my_ff.itp' I encounter the following on lines 146 and 147:

```
[ angletypes ]
CA  CB  O   1   109.47  350.00  ; PARM 4 5
```

From reading [angletypes] I know I'm in the 'angletypes' section.

On the next line, I notice two parameters on fields 4 and 5.

From the atom types, section type and field number I know the parameter IDs are 'ANGLESBCACBO' and 'ANGLE-SKCACBO'.

After building map={'ANGLESBCACBO':1,'ANGLESKCACBO':2}, I store the values in an array: pvals0=array([109.-47, 350.00]), and I put the parameter locations in pfields: pfields=[['my_ff.itp',147,4,1.0],['my_ff.itp',146,5,1.0]]. The 1.0 is a 'multiplier' and I will explain it below.

Note that in the creation of parameter IDs, we run into the issue that the atoms involved in the interaction may be labeled in reverse order (e.g. OCACB). Thus, we store both the normal and the reversed parameter ID in the map.

Parameter repetition and multiplier:

If 'RPT' is encountered in the line, it is always in the syntax: 'RPT 4 ANGLESBCACAH 5 MINUS_ANGLESKCAH /RPT'. In this case, field 4 is replaced by the stored parameter value corresponding to ANGLESBCACAH and field 5 is replaced by -1 times the stored value of ANGLESKCAH. Now I just picked this as an example, I don't think people actually want a negative angle force constant .. :) the MINUS keyword does come in handy for assigning atomic charges and virtual site positions. In order to achieve this, a multiplier of -1.0 is stored into pfields instead of 1.0.

Todo Note that I can also create the opposite virtual site position by changing the atom labeling, woo!

Warning

My program currently assumes that we are only using one MM program per job. If we use CHARMM and GROMACS to perform fitting simulations in the same job, we will get f-ed up. Maybe this needs to be fixed in the future, with program prefixes to parameters like C_, G_ .. or simply unit conversions, you get the idea.

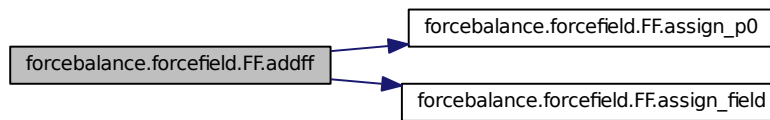
I don't think the multiplier actually works for analytic derivatives unless the interaction calculator knows the multiplier as well. I'm sure I can make this work in the future if necessary.

Parameters

in	ffname	Name of the force field file
----	--------	------------------------------

Definition at line 301 of file forcefield.py.

Here is the call graph for this function:



10.3.3.2 `def forcebalance.forcefield.FF.assign_field(self, idx, fnm, ln, pfld, mult)`

Record the locations of a parameter in a file; [[file name, line number, field number, and multiplier]].

Note that parameters can have multiple locations because of the repetition functionality.

Parameters

in	<i>idx</i>	The index of the parameter.
in	<i>fnm</i>	The file name of the parameter field.
in	<i>ln</i>	The line number within the file.
in	<i>pfld</i>	The field within the line.
in	<i>mult</i>	The multiplier (this is usually 1.0)

Definition at line 609 of file `forcefield.py`.

10.3.3.3 `def forcebalance.forcefield.FF.assign_p0(self, idx, val)`

Assign physical parameter values to the 'pvals0' array.

Parameters

in	<i>idx</i>	The index to which we assign the parameter value.
in	<i>val</i>	The parameter value to be inserted.

Definition at line 591 of file `forcefield.py`.

10.3.3.4 `def forcebalance.forcefield.FF.create_mvals(self, pvals)`

Converts physical to mathematical parameters.

We create the inverse transformation matrix using SVD.

Parameters

in	<i>pvals</i>	The physical parameters
----	--------------	-------------------------

Returns

mvals The mathematical parameters

Definition at line 394 of file `forcefield.py`.

10.3.3.5 def forcebalance.forcefield.FF.create_pvals (self, mvals)

Converts mathematical to physical parameters.

First, mathematical parameters are rescaled and rotated by multiplying by the transformation matrix, followed by adding the original physical parameters.

Parameters

in	<i>mvals</i>	The mathematical parameters
----	--------------	-----------------------------

Returns

pvals The physical parameters

Definition at line 382 of file forcefield.py.

10.3.3.6 def forcebalance.forcefield.FF.list_map (self)

Create the plist, which is like a reversed version of the parameter map.

More convenient for printing.

Definition at line 571 of file forcefield.py.

10.3.3.7 def forcebalance.forcefield.FF.make (self, printdir, vals, usepvals)

Create a new force field using provided parameter values.

This big kahuna does a number of things: 1) Creates the physical parameters from the mathematical parameters 2) Creates force fields with physical parameters substituted in 3) Prints the force fields to the specified file.

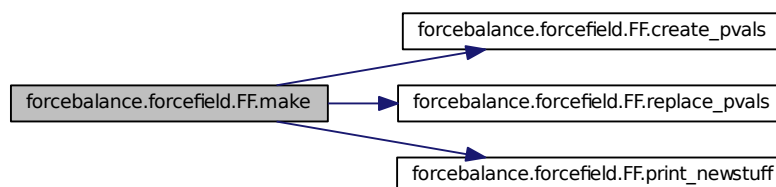
It does NOT store the mathematical parameters in the class state (since we can only hold one set of parameters).

Parameters

in	<i>printdir</i>	The directory that the force fields are printed to; as usual this is relative to the project root directory.
in	<i>vals</i>	Input parameters. I previously had an option where it uses stored values in the class state, but I don't think that's a good idea anymore.
in	<i>usepvals</i>	Switch for whether to bypass the coordinate transformation and use physical parameters directly.

Definition at line 362 of file forcefield.py.

Here is the call graph for this function:



10.3.3.8 `def forcebalance.forcefield.FF.mktransmat (self)`

Create the transformation matrix to rescale and rotate the mathematical parameters.

For point charge parameters, project out perturbations that change the total charge.

First build these:

'qmap' : Just a list of parameter indices that point to charges.

'qid' : For each parameter in the qmap, a list of the affected atoms :) A potential target for the molecule-specific thang.

Then make this:

'qtrans2' : A transformation matrix that rotates the charge parameters. The first row is all zeros (because it corresponds to increasing the charge on all atoms) The other rows correspond to changing one of the parameters and decreasing all of the others equally such that the overall charge is preserved.

'qmat2' : An identity matrix with 'qtrans2' pasted into the right place

'transmat': 'qmat2' with rows and columns scaled using self.rs

'excision': Parameter indices that need to be 'cut out' because they are irrelevant and mess with the matrix diagonalization

Todo Only project out changes in total charge of a molecule, and perhaps generalize to fragments of molecules or other types of parameters.

Definition at line 502 of file forcefield.py.

10.3.3.9 `def forcebalance.forcefield.FF.print_map (self, vals = None)`

Prints out the (physical or mathematical) parameter indices, IDs and values in a visually appealing way.

Definition at line 580 of file forcefield.py.

10.3.3.10 `def forcebalance.forcefield.FF.print_newstuff (self, printdir)`

Prints out the new content of force fields to files in 'printdir'.

Parameters

<i>in</i>	<i>printdir</i>	The directory to which new force fields are printed.
-----------	-----------------	--

Definition at line 425 of file forcefield.py.

10.3.3.11 `def forcebalance.forcefield.FF.replace_pvals (self, pvals)`

Replaces numerical fields in stored force field files with the stored physical parameter values.

Unless you really know what you're doing, you probably shouldn't be calling this directly.

Definition at line 404 of file forcefield.py.

10.3.3.12 `def forcebalance.forcefield.FF.rsmake (self, printfacs = True)`

Create the rescaling factors for the coordinate transformation in parameter space.

The proper choice of rescaling factors (read: prior widths in maximum likelihood analysis) is still a black art. This is a topic of current research.

Todo Pass in rsfactors through the input file

Parameters

<code>in</code>	<code>printfacs</code>	List for printing out the rescaling factors
-----------------	------------------------	---

Definition at line 442 of file forcefield.py.

10.3.4 Member Data Documentation

10.3.4.1 forcebalance::forcefield.FF::excision

Indices to exclude from optimization / Hessian inversion.

```
chargegrp = [[1,3],[4,5],[6,7]] for group in chargegrp: a = min(group[0]-1, group[1]) b = max(group[0]-1, group[1]) con-
stemp = zeros(tq, dtype=float) for i in range(constemp.shape[0]): if i >= a and i < b: constemp[i] += 1 else: constemp[i]
-= 1 cons0 = vstack((cons0, constemp)) print cons0 Here is where we build the qtrans2 matrix.
```

Definition at line 181 of file forcefield.py.

10.3.4.2 forcebalance::forcefield.FF::newstuff

The force field content, but with parameter fields replaced with new parameters.

Definition at line 183 of file forcefield.py.

10.3.4.3 forcebalance::forcefield.FF::pfields

A list where pfields[pnum] = ['file',line,field,mult], basically a new way to modify force field files; when we modify the force field file, we go to the specific line/field in a given file and change the number.

Definition at line 177 of file forcefield.py.

10.3.4.4 forcebalance::forcefield.FF::rs

List of rescaling factors.

Takes the dictionary 'BONDS':{3:'B', 4:'K'}, 'VDW':{4:'S', 5:'T'}, and turns it into a list of term types ['BONDSB','BOND-SK','VDWS','VDWT'].

The array of rescaling factors

Definition at line 178 of file forcefield.py.

The documentation for this class was generated from the following file:

- forcebalance/forcebalance/forcefield.py

10.4 forcebalance.fitsim.FittingSimulation Class Reference

Base class for all fitting simulations.

Public Member Functions

- def [__init__](#)
Instantiation of a fitting simulation.
- def [get_X](#)
Computes the objective function contribution without any parametric derivatives.
- def [get_G](#)

- *Computes the objective function contribution and its gradient.*
- def [get_H](#)
Computes the objective function contribution and its gradient / Hessian.
- def [refresh_temp_directory](#)
Back up the temporary directory if desired, delete it and then create a new one.
- def [get](#)

Public Attributes

- [root](#)
Root directory of the whole project.
- [name](#)
Name of the fitting simulation.
- [simtype](#)
Type of fitting simulation.
- [weight](#)
Relative weight of the fitting simulation.
- [fdgrad](#)
Switch for finite difference gradients.
- [fdhess](#)
Switch for finite difference Hessians.
- [fdhessdiag](#)
Switch for FD gradients + Hessian diagonals.
- [fd1_pids](#)
Parameter types that trigger FD gradient elements.
- [fd2_pids](#)
Parameter types that trigger FD Hessian elements.
- [h](#)
Finite difference step size.
- [usepvals](#)
Manual override: bypass the parameter transformation and use physical parameters directly.
- [simdir](#)
Relative directory of fitting simulation.
- [tempdir](#)
Temporary (working) directory.
- [FF](#)
Need the forcefield (here for now)
- [xct](#)
Counts how often the objective function was computed.
- [gct](#)
Counts how often the gradient was computed.
- [hct](#)
Counts how often the Hessian was computed.

10.4.1 Detailed Description

Base class for all fitting simulations.

In ForceBalance a 'fitting simulation' is defined as a simulation which computes a quantity that we can compare to a reference. The force field parameters are tuned to reproduce the reference value as closely as possible.

The 'computable quantities' may include energies and forces where the reference values come from QM calculations (energy and force matching), energies from an EDA analysis (Maybe in the future, FDA?), molecular properties (like polarizability, refractive indices, multipole moments or vibrational frequencies), relative entropies, and bulk properties. Single-molecule or bulk properties can even come from the experiment!

The central idea in ForceBalance is that each quantity makes a contribution to the overall objective function. So we can build force fields that fit several quantities at once, rather than putting all of our chips behind energy and force matching. In the future ForceBalance may even include multiobjective optimization into the optimizer.

The optimization is done by way of minimizing an 'objective function', which is comprised of squared differences between the computed and reference values. These differences are not computed in this file, but rather in subclasses that use [FittingSimulation](#) as a base class. Thus, the contents of [FittingSimulation](#) itself are meant to be as general as possible, because the pertinent variables apply to all types of fitting simulations.

An important note: [FittingSimulation](#) requires that all subclasses have a method `get(self,mvals,AGrad=False,AHess=False,tempdir=None)` that does the following:

Inputs: `mvals` = The parameter vector, which modifies the force field (Note to self: We include `mvals` with each Fit-Sim because we can create copies of the force field and do finite difference derivatives) `AGrad`, `AHess` = Boolean switches for computing analytic gradients and Hessians `tempdir` = Temporary directory; we can create multiple of these for parallelization of our jobs (a future consideration)

Outputs: `Answer` = {'X': Number, 'G': `numpy.array(np)`, 'H': `numpy.array((np,np))`} 'X' = The objective function itself 'G' = The gradient, elements not computed analytically are zero 'H' = The Hessian, elements not computed analytically are zero

This is the only global requirement of a [FittingSimulation](#). Obviously 'get' itself is not defined here, because its calculation will depend entirely on specifically which simulation we wish to run. However, this should give us a unified framework which will facilitate rapid implementation of FittingSimulations.

Future work: Robert suggested that I could enable automatic detection of which parameters need to be computed by finite difference. Not a bad idea. :)

Definition at line 71 of file `fitsim.py`.

10.4.2 Constructor & Destructor Documentation

10.4.2.1 `def forcebalance.fitsim.FittingSimulation.__init__(self, options, sim_opts, forcefield)`

Instantiation of a fitting simulation.

All options here are intended to be usable by every conceivable type of fitting simulation (in other words, only add content here if it's widely applicable.)

If we want to add attributes that are more specific (i.e. a set of reference forces for force matching), they are added in the subclass `ForceEnergyMatch` that subclasses [FittingSimulation](#).

Definition at line 90 of file `fitsim.py`.

10.4.3 Member Function Documentation

10.4.3.1 `def forcebalance.fitsim.FittingSimulation.get (self, mvals, AGrad=False, AHess=False, tempdir=None)`

Todo Write documentation here later.

Definition at line 227 of file fitsim.py.

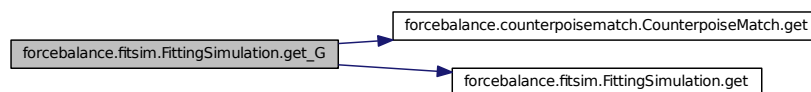
10.4.3.2 `def forcebalance.fitsim.FittingSimulation.get_G (self, mvals=None)`

Computes the objective function contribution and its gradient.

First the low-level 'get' method is called with the analytic gradient switch turned on. Then we loop through the fd1_pids and compute the corresponding elements of the gradient by finite difference, if the 'fdgrad' switch is turned on. Alternately we can compute the gradient elements and diagonal Hessian elements at the same time using central difference if 'fdhessdiag' is turned on.

Definition at line 161 of file fitsim.py.

Here is the call graph for this function:



10.4.3.3 `def forcebalance.fitsim.FittingSimulation.get_H (self, mvals=None)`

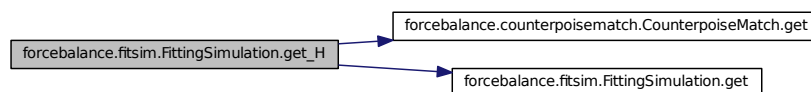
Computes the objective function contribution and its gradient / Hessian.

First the low-level 'get' method is called with the analytic gradient and Hessian both turned on. Then we loop through the fd1_pids and compute the corresponding elements of the gradient by finite difference, if the 'fdgrad' switch is turned on.

This is followed by looping through the fd2_pids and computing the corresponding Hessian elements by finite difference. Forward finite difference is used throughout for the sake of speed.

Definition at line 186 of file fitsim.py.

Here is the call graph for this function:



10.4.3.4 `def forcebalance.fitsim.FittingSimulation.refresh_temp_directory (self)`

Back up the temporary directory if desired, delete it and then create a new one.

Definition at line 208 of file fitsim.py.

10.4.4 Member Data Documentation

10.4.4.1 forcebalance::fitsim.FittingSimulation::usepvals

Manual override: bypass the parameter transformation and use physical parameters directly.

For power users only! :)

Definition at line 102 of file fitsim.py.

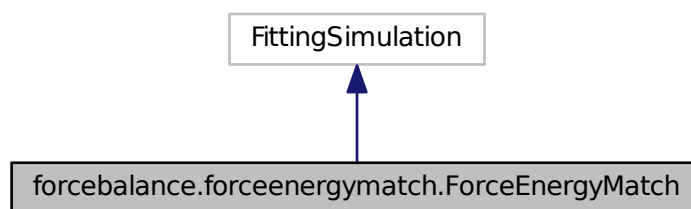
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/fitsim.py

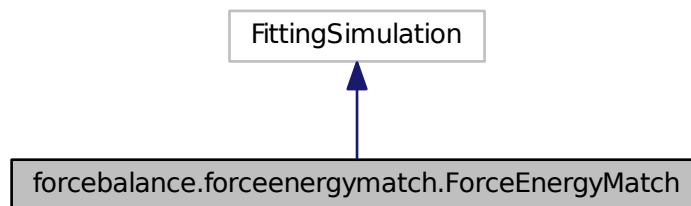
10.5 forcebalance.forceenergymatch.ForceEnergyMatch Class Reference

Subclass of FittingSimulation for force and energy matching.

Inheritance diagram for forcebalance.forceenergymatch.ForceEnergyMatch:



Collaboration diagram for forcebalance.forceenergymatch.ForceEnergyMatch:



Public Member Functions

- def [__init__](#)

Instantiation of the subclass.

- def [read_reference_data](#)
Read the reference data (for force and energy matching) from a file such as qdata.txt.
- def **indicate**
- def [get](#)
LPW 01-11-2012.

Public Attributes

- [ns](#)
Number of snapshots.
- [whamboltz](#)
Whether to use WHAM Boltzmann weights.
- [sampcorr](#)
Whether to use the Sampling Correction.
- [covariance](#)
Whether to use the Covariance Matrix.
- [qmboltz](#)
Whether to use QM Boltzmann weights.
- [qmboltztemp](#)
The temperature for QM Boltzmann weights.
- [fitatoms](#)
Number of atoms that we are fitting.
- [efweight](#)
The proportion of energy vs.
- [whamboltz_wts](#)
WHAM Boltzmann weights.
- [qmboltz_wts](#)
QM Boltzmann weights.
- [eqm](#)
Reference (QM) energies.
- [emd0](#)
Energies of the sampling simulation.
- [fqm](#)
Reference (QM) forces.
- [qfnm](#)
The qdata.txt file that contains the QM energies and forces.
- [natoms](#)
The number of true atoms.
- [e_err](#)
Qualitative Indicator: average energy error (in kJ/mol)
- [f_err](#)
Qualitative Indicator: average force error (fractional)
- [traj](#)
Read in the trajectory file.

10.5.1 Detailed Description

Subclass of FittingSimulation for force and energy matching.

In force and energy matching, we introduce the following concepts:

- The number of snapshots
- The reference energies and forces (eqm, fqm) and the file they belong in (qdata.txt)
- The sampling simulation energies (emd0)
- The WHAM Boltzmann weights (these are computed externally and passed in)
- The QM Boltzmann weights (computed internally using the difference between eqm and emd0)

There are also these little details:

- Switches for whether to turn on certain Boltzmann weights (they stack)
- Temperature for the QM Boltzmann weights
- Whether to fit a subset of atoms

This subclass contains the 'get' method for building the objective function from any simulation software (a driver to run the program and read output is still required). The 'get' method can be overridden by subclasses like ForceEnergyMatch_GMXX2.

Definition at line 37 of file forceenergymatch.py.

10.5.2 Constructor & Destructor Documentation

10.5.2.1 `def forcebalance.forceenergymatch.ForceEnergyMatch.__init__(self, options, sim_opts, forcefield)`

Instantiation of the subclass.

We begin by instantiating the superclass here and also defining a number of core concepts for energy / force matching.

Todo Obtain the number of true atoms (or the particle -> atom mapping) from the force field.

Definition at line 50 of file forceenergymatch.py.

10.5.3 Member Function Documentation

10.5.3.1 `def forcebalance.forceenergymatch.ForceEnergyMatch.get(self, mvals, AGrad=False, AHess=False, tempdir=None)`

LPW 01-11-2012.

This subroutine builds the objective function (and optionally its derivatives) from a general simulation software. This is in contrast to using GROMACS-X2, which computes the objective function and prints it out; then 'get' only needs to call GROMACS and read it in.

This subroutine interfaces with simulation software 'drivers'. The driver is only expected to give the energy and forces.

Now this subroutine may sound trivial since the objective function is simply a least-squares quantity $(M-Q)^2$ - but there are a number of nontrivial considerations. I will list them here.

0) Polytensor formulation: Because there may exist covariance between different components of the force (or covariance between the energy and the force), we build the objective function by taking outer products of vectors that have the form $[E \ F_{1x} \ F_{1y} \ F_{1z} \ F_{2x} \ F_{2y} \ \dots]$, and then we trace it with the inverse of the covariance matrix to get the objective function.

1) Boltzmann weights and normalization: Each snapshot has its own Boltzmann weight, which may or may not be normalized. This subroutine does the normalization automatically.

2) Subtracting out the mean energy gap: The zero-point energy difference between reference and fitting simulations is meaningless. This subroutine subtracts it out.

3) Hybrid ensembles: This program builds a combined objective function from both MM and QM ensembles, which is rigorously better than using a single ensemble.

Note that this subroutine does not do EVERYTHING that GROMACS-X2 can do, which includes:

1) Internal coordinate systems 2) 'Sampling correction' (deprecated, since it doesn't seem to work) 3) Analytic derivatives

In the previous code (ForTune) this subroutine used analytic first derivatives of the energy and force to build the derivatives of the objective function. Here I will take a simplified approach, because building the derivatives are cumbersome. For now we will return the objective function ONLY. A two-point central difference should give us the first and diagonal second derivative anyhow.

Todo Parallelization over snapshots is not implemented yet

Parameters

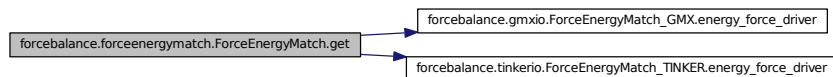
in	<i>mvals</i>	Mathematical parameter values
in	<i>AGrad</i>	Switch to turn on analytic gradient, useless here
in	<i>AHess</i>	Switch to turn on analytic Hessian, useless here
in	<i>tempdir</i>	Temporary directory for running computation

Returns

Answer Contribution to the objective function

Definition at line 307 of file forceenergymatch.py.

Here is the call graph for this function:



10.5.3.2 def forcebalance.forceenergymatch.ForceEnergyMatch.read_reference_data (self)

Read the reference data (for force and energy matching) from a file such as qdata.txt.

Todo Add an option for picking any slice out of qdata.txt, helpful for cross-validation

Todo Closer integration of reference data with program - leave behind the qdata.txt format? (For now, I like the readability of qdata.txt)

After reading in the information from qdata.txt, it is converted into the GROMACS energy units (kind of an arbitrary choice); forces (kind of a misnomer in qdata.txt) are multiplied by -1 to convert gradients to forces.

We also subtract out the mean energies of all energy arrays because energy/force matching does not account for zero-point energy differences between MM and QM (i.e. energy of electrons in core orbitals).

The configurations in force/energy matching typically come from a the thermodynamic ensemble of the MM force field at some temperature (by running MD, for example), and for many reasons it is helpful to introduce non-Boltzmann weights in front of these configurations. There are two options: WHAM Boltzmann weights (for combining the weights of several simulations together) and QM Boltzmann weights (for converting MM weights into QM weights). Note that the two sets of weights 'stack'; i.e. they can be used at the same time.

A 'hybrid' ensemble is possible where we use 50% MM and 50% QM weights. Please read more in LPW and Troy Van Voorhis, JCP Vol. 133, Pg. 231101 (2010), doi:10.1063/1.3519043.

Todo The WHAM Boltzmann weights are generated by external scripts (wanalyze.py and make-wham-data.sh) and passed in; perhaps these scripts can be added to the ForceBalance distribution or integrated more tightly.

Finally, note that using non-Boltzmann weights degrades the statistical information content of the snapshots. This problem will generally become worse if the ensemble to which we're reweighting is dramatically different from the one we're sampling from. We end up with a set of Boltzmann weights like [1e-9, 1e-9, 1.0, 1e-9, 1e-9 ...] and this is essentially just one snapshot. I believe Troy is working on something to cure this problem.

Here, we have a measure for the information content of our snapshots, which comes easily from the definition of information entropy:

$$S = -1 * \sum_i (P_i * \log(P_i)) \text{ InfoContent} = \exp(-S)$$

With uniform weights, InfoContent is equal to the number of snapshots; with horrible weights, InfoContent is closer to one.

Definition at line 169 of file forceenergymatch.py.

10.5.4 Member Data Documentation

10.5.4.1 forcebalance::forceenergymatch.ForceEnergyMatch::efweight

The proportion of energy vs.

force.

Definition at line 58 of file forceenergymatch.py.

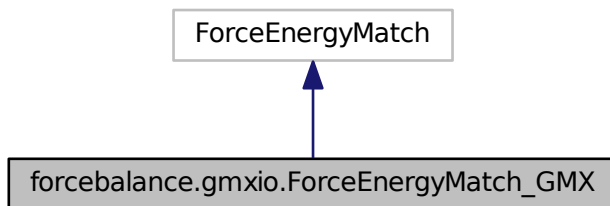
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/forceenergymatch.py

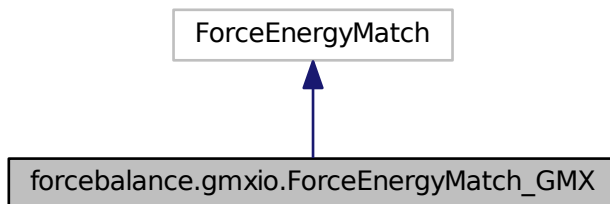
10.6 forcebalance.gmxio.ForceEnergyMatch_GMX Class Reference

Subclass of FittingSimulation for force and energy matching using normal GROMACS.

Inheritance diagram for forcebalance.gmxio.ForceEnergyMatch_GMX:



Collaboration diagram for forcebalance.gmxio.ForceEnergyMatch_GMX:



Public Member Functions

- def **__init__**
- def **prepare_temp_directory**
- def **energy_force_driver**

Public Attributes

- [trajfnm](#)

Name of the trajectory, we need this BEFORE initializing the SuperClass.

10.6.1 Detailed Description

Subclass of FittingSimulation for force and energy matching using normal GROMACS.

Implements the prepare_temp_directory and energy_force_driver methods.

Definition at line 302 of file gmxio.py.

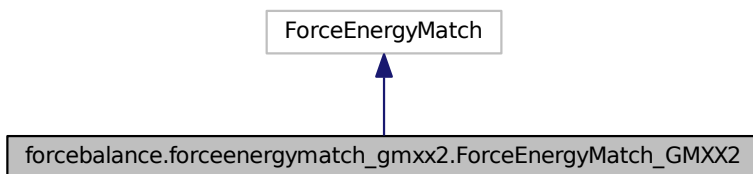
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/gmxio.py

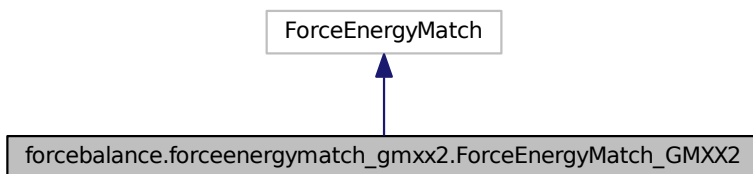
10.7 forcebalance.forceenergymatch_gmxx2.ForceEnergyMatch_GMXX2 Class Reference

ForceBalance class for force and energy matching with the modified GROMACS.

Inheritance diagram for forcebalance.forceenergymatch_gmxx2.ForceEnergyMatch_GMXX2:



Collaboration diagram for forcebalance.forceenergymatch_gmxx2.ForceEnergyMatch_GMXX2:



Public Member Functions

- def [__init__](#)
Instantiation of [ForceEnergyMatch_GMXX2](#).
- def [prepare_temp_directory](#)
Prepare the temporary directory for running the modified GROMACS.
- def [get](#)
Calls the modified GROMACS and collects the objective function contribution.
- def [callgmxx2](#)
Call the modified GROMACS!

Public Attributes

- [trajfnm](#)

Set the software to GROMACS no matter what.

- **whamboltz**
- **sampcorr**
- **covariance**
- **e_err**
- **f_err**

10.7.1 Detailed Description

ForceBalance class for force and energy matching with the modified GROMACS.

This class allows us to use a heavily modified version of GROMACS (a major component of this program) to compute the objective function contribution. The modified GROMACS does the looping through snapshots, computes the interactions as well as the derivatives, and sums them up to build the objective function. I will write that documentation elsewhere, perhaps when I port GROMACS over to version 4.5.4.

This class implements the 'get' method. When 'get' is called, the force field is printed to the temporary directory along with several files containing the information needed by the modified GROMACS (the Boltzmann weights, the parameters that need derivatives and their values, the QM energies and forces, and the energy / force weighting.)

The modified GROMACS is called with the arguments '-rerun all.gro -fortune -rerunvsite' to loop over the snapshots, turn on force matching functionality and reconstruct virtual site positions (an important consideration if we're changing the virtual site positions in the optimization). Its outputs are 'e2f2bc' which means 'energy squared, force squared, boltzmann corrected' and contains the objective function, 'a1dbc' and 'a2dbc' containing analytic first and second derivatives, 'gmxboltz' containing the Boltzmann weights used, and possibly some other stuff.

Most importantly, 'e2f2bc', 'a1dbc' and 'a2dbc' are read by 'get' after GROMACS is called and returned directly as the objective function contributions.

Other methods implemented in this class are related to the preparation of the temp directory.

Definition at line 62 of file forceenergymatch_gmxx2.py.

10.7.2 Constructor & Destructor Documentation

10.7.2.1 **def forcebalance.forceenergymatch_gmxx2.ForceEnergyMatch_GMXX2.__init__(self, options, sim_opts, forcefield)**

Instantiation of [ForceEnergyMatch_GMXX2](#).

Several important things happen here:

- We load in the coordinates from 'all.gro'.
- We prepare the temporary directory.

Definition at line 73 of file forceenergymatch_gmxx2.py.

10.7.3 Member Function Documentation

10.7.3.1 `def forcebalance.forceenergymatch_gmxx2.ForceEnergyMatch_GMXX2.get (self, mvals, AGrad = False, AHess=False, tempdir=None)`

Calls the modified GROMACS and collects the objective function contribution.

First we create the force field using the parameter values that were passed in. Note that we may pass in physical parameters directly and bypass the coordinate transformation by setting `self.usepvals` to `True`.

The physical parameters are printed to 'pvals' for GROMACS to read - of course GROMACS knows the parameters already, but this facilitates retrieval from the low level subroutines.

Several switches are printed to files, such as:

- 'FirstDerivativesOnly' to prevent computation of the Hessian
- 'NoDerivatives' to prevent computation of the Hessian AND the gradient

GROMACS is called in the `callgmxx2()` method.

The output files are then parsed for the objective function and its derivatives are read in. The answer is passed out as a dictionary: {'X': Objective Function, 'G': Gradient, 'H': Hessian}

Parameters

in	<i>mvals</i>	Mathematical parameter values
in	<i>AGrad</i>	Switch to turn on analytic gradient
in	<i>AHess</i>	Switch to turn on analytic Hessian
in	<i>tempdir</i>	Temporary directory for running computation

Returns

Answer Contribution to the objective function

Todo Some of these files don't need to be printed, they can be passed to GROMACS as arguments. Let's think about this some more.

Currently I have no way to pass out the qualitative indicators.

Definition at line 204 of file `forceenergymatch_gmxx2.py`.

Here is the call graph for this function:



10.7.3.2 `def forcebalance.forceenergymatch_gmxx2.ForceEnergyMatch_GMXX2.prepare_temp_directory (self, options, sim_opts, tempdir=None)`

Prepare the temporary directory for running the modified GROMACS.

This method creates the temporary directory, links in the necessary files for running (except for the force field), and writes the coordinate file for the snapshots we've chosen.

There are also files that specific to our *modified* GROMACS, including:

- qmboltz : The QM Boltzmann weights

- `bp` : The QM vs. MM Boltzmann weight proportionality factor
- `whamboltz` : The WHAM Boltzmann weights (i.e. MM Boltzmann weights passed from outside)
- `sampcorr` : Boolean for the 'sampling correction', i.e. updating the Boltzmann factors when the force field is updated. This required a TON of implementation into the modified Gromacs, but in the end we didn't find it to be very useful. It basically emphasizes energy minima and gets barrier heights wrong. Blah! :)
- `fitatoms` : The number of atoms that we're fitting, which may be less than the total number in the QM calculation (i.e. if we are fitting something to be compatible with a water model ...)
- `energyqm` : QM reference energies
- `forcesqm` : QM reference forces
- `ztemp` : Template for Z-matrix coordinates (for internal coordinate forces)
- `pids` : Information for building interaction name -> parameter number hashtable

Parameters

<code>in</code>	<code>tempdir</code>	The temporary directory to be prepared.
-----------------	----------------------	---

Todo Someday I'd like to use WHAM to put AIMD simulations in. :)

The fitatoms shouldn't be the first however many atoms, it should be a list.

Definition at line 112 of file `forceenergymatch_gmxx2.py`.

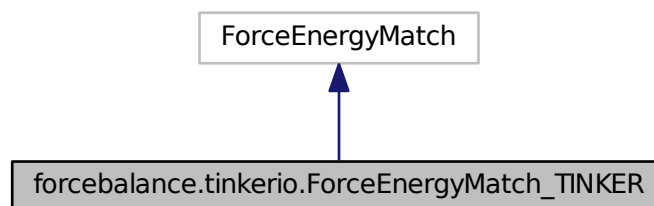
The documentation for this class was generated from the following file:

- `forcebalance/forcebalance/forceenergymatch_gmxx2.py`

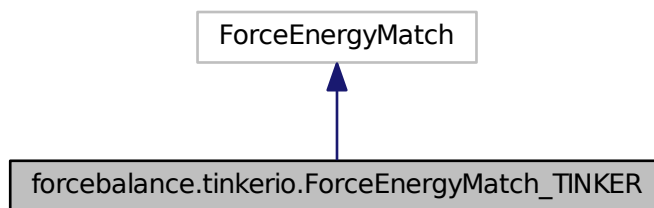
10.8 forcebalance.tinkerio.ForceEnergyMatch_TINKER Class Reference

Subclass of `FittingSimulation` for force and energy matching using TINKER.

Inheritance diagram for `forcebalance.tinkerio.ForceEnergyMatch_TINKER`:



Collaboration diagram for forcebalance.tinkerio.ForceEnergyMatch_TINKER:



Public Member Functions

- def `__init__`
- def `prepare_temp_directory`
- def `energy_force_driver`

Public Attributes

- `trajfnm`

Name of the trajectory, we need this BEFORE initializing the SuperClass.

10.8.1 Detailed Description

Subclass of FittingSimulation for force and energy matching using TINKER.

Implements the prepare and energy_force_driver methods. The get method is in the superclass.

Definition at line 133 of file tinkerio.py.

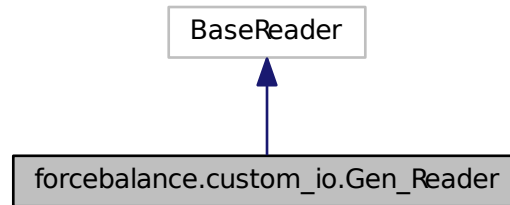
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/tinkerio.py

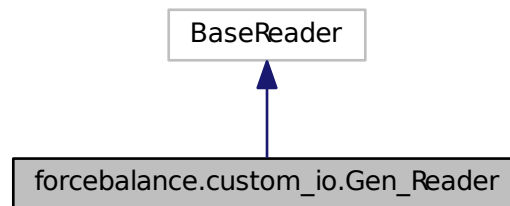
10.9 forcebalance.custom_io.Gen_Reader Class Reference

Finite state machine for parsing custom GROMACS force field files.

Inheritance diagram for forcebalance.custom_io.Gen_Reader:



Collaboration diagram for forcebalance.custom_io.Gen_Reader:



Public Member Functions

- `def __init__`
- `def feed`
Feed in a line.

Public Attributes

- `sec`
The current section that we're in.
- `pdict`
The parameter dictionary (defined in this file)
- `itype`
- `suffix`

10.9.1 Detailed Description

Finite state machine for parsing custom GROMACS force field files.

This class is instantiated when we begin to read in a file. The feed(line) method updates the state of the machine, giving it information like the residue we're currently on, the nonbonded interaction type, and the section that we're in. Using this information we can look up the interaction type and parameter type for building the parameter ID.

Definition at line 41 of file custom_io.py.

10.9.2 Member Function Documentation

10.9.2.1 def forcebalance.custom_io.Gen_Reader.feed (self, line)

Feed in a line.

Parameters

<i>in</i>	<i>line</i>	The line of data
-----------	-------------	------------------

Definition at line 57 of file custom_io.py.

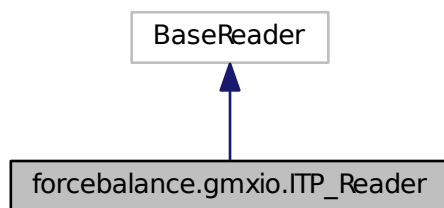
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/custom_io.py

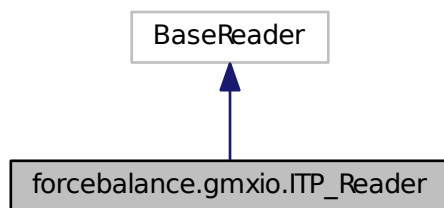
10.10 forcebalance.gmxio.ITP_Reader Class Reference

Finite state machine for parsing GROMACS force field files.

Inheritance diagram for forcebalance.gmxio.ITP_Reader:



Collaboration diagram for forcebalance.gmxio.OTP_Reader:



Public Member Functions

- `def __init__`
- `def feed`

Given a line, determine the interaction type and the atoms involved (the suffix).

Public Attributes

- `sec`
The current section that we're in.
- `nbtype`
Nonbonded type.
- `res`
The current residue (set by the moleculetype keyword)
- `adict`
The mapping of (this residue, atom number) to (atom name) for building atom-specific interactions in [bonds], [angles] etc.
- `pdict`
The parameter dictionary (defined in this file)
- `itype`
- `suffix`

10.10.1 Detailed Description

Finite state machine for parsing GROMACS force field files.

This class is instantiated when we begin to read in a file. The `feed(line)` method updates the state of the machine, giving it information like the residue we're currently on, the nonbonded interaction type, and the section that we're in. Using this information we can look up the interaction type and parameter type for building the parameter ID.

Definition at line 160 of file `gmxio.py`.

10.10.2 Member Function Documentation

10.10.2.1 def forcebalance.gmxio.ITP_Reader.feed (self, line)

Given a line, determine the interaction type and the atoms involved (the suffix).

For example, we want

```
H O H 5 1.231258497536e+02 4.269161426840e+02 -1.033397697685e-02 1.304674117410e+04
; PARM 4 5 6 7
```

to give us itype = 'UREY_BRADLEY' and suffix = 'HOH'

If we are in a TypeSection, it returns a list of atom types;

If we are in a TopolSection, it returns a list of atom names.

The section is essentially a case statement that picks out the appropriate interaction type and makes a list of the atoms involved

Note that we can call gmxdump for this as well, but I prefer to read the force field file directly.

ToDo: [atoms] section might need to be more flexible to accommodate optional fields

Definition at line 196 of file gmxio.py.

10.10.3 Member Data Documentation

10.10.3.1 forcebalance::gmxio.ITP_Reader::adict

The mapping of (this residue, atom number) to (atom name) for building atom-specific interactions in [bonds], [angles] etc.

Definition at line 166 of file gmxio.py.

The documentation for this class was generated from the following file:

- forcebalance/forcebalance/gmxio.py

10.11 forcebalance.molecule.Molecule Class Reference

The [Molecule](#) class contains information about a system of molecules.

Public Member Functions

- def [__init__](#)
To instantiate the class we simply define the table of file reading/writing functions and read in a file if it is provided.
- def [read](#)
Read in a file.
- def **na**
- def **write**
- def **oops**
- def [read_xyz](#)
Parse a .xyz file which contains several xyz coordinates, and return their elements.
- def [read_com](#)
Parse a Gaussian .com file and return a SINGLE-ELEMENT list of xyz coordinates (no multiple file support)

- def [read_arc](#)
Read a TINKER .arc file.
- def [read_ndx](#)
Read an index.ndx file and add an entry to the dictionary {'index_name': [num1, num2, num3 .
- def [reorder](#)
Reorders an xyz file using data provided from an .ndx file.
- def [read_gro](#)
Read a GROMACS .gro file.
- def [read_charmm](#)
Read a CHARMM .cor (or .crd) file.
- def [write_qcin](#)
Write a Q-Chem input file from the template.
- def **write_xyz**
- def **write_arc**
- def **write_gro**
- def **require_resid**
- def **require_resname**
- def **require_boxes**

Public Attributes

- [Read_Tab](#)
The table of file readers.
- [Write_Tab](#)
The table of file writers.
- [index](#)
If supplied with a file name, read in stuff.
- [elem](#)
File type can be determined from the file name using the extension.
- [xyzs](#)
Absolutely required; a list of lists of xyz coordinates (number of frames x number of atoms)
- [ns](#)
The number of snapshots is determined by the length of self.xyzs.
- [comms](#)
The comments that usually go somewhere into the output file.
- [charge](#)
Optional variable: the charge.
- [mult](#)
Optional variable: the multiplicity.
- [boxes](#)
Optional variable: the box vectors (there would be self.ns of these)
- [resid](#)
Optional variable: the residue number.
- [resname](#)
Optional variable: the residue name.
- [atomname](#)
Optional variable: the atom name, which defaults to the elements.

- [rawarcs](#)

Optional variable: raw .arc file in Tinker (because Tinker files are too hard to interpret!)

- [tempfnm](#)

Name of the template file.

10.11.1 Detailed Description

The [Molecule](#) class contains information about a system of molecules.

This is a centralized container of information that is useful for file conversion and processing. It is quite flexible but not perfectly flexible. For example, we can accommodate multiple sets of xyz coordinates (i.e. multiple frames in a trajectory), but we only allow a single set of atomic elements (i.e. can't have molecules changing identities across frames.)

Definition at line 113 of file molecule.py.

10.11.2 Constructor & Destructor Documentation

10.11.2.1 `def forcebalance.molecule.Molecule.__init__(self, fnm=None, ftype=None)`

To instantiate the class we simply define the table of file reading/writing functions and read in a file if it is provided.

Definition at line 117 of file molecule.py.

10.11.3 Member Function Documentation

10.11.3.1 `def forcebalance.molecule.Molecule.read(self, fnm, ftype=None)`

Read in a file.

This populates the attributes of the class. For now, I don't know what will happen if we read two files into the class. Probably unexpected behavior will occur.

Definition at line 142 of file molecule.py.

10.11.3.2 `def forcebalance.molecule.Molecule.read_com(self, fnm)`

Parse a Gaussian .com file and return a SINGLE-ELEMENT list of xyz coordinates (no multiple file support)

Definition at line 255 of file molecule.py.

10.11.3.3 `def forcebalance.molecule.Molecule.read_ndx(self, fnm)`

Read an index.ndx file and add an entry to the dictionary {'index_name': [num1, num2, num3 .
 .]}

Definition at line 355 of file molecule.py.

10.11.3.4 `def forcebalance.molecule.Molecule.read_xyz(self, fnm)`

Parse a .xyz file which contains several xyz coordinates, and return their elements.

Parameters

<code>in</code>	<code>fnm</code>	The input XYZ file name
-----------------	------------------	-------------------------

Returns

elem A list of chemical elements in the XYZ file

comms A list of comments.

xyzs A list of XYZ coordinates (number of snapshots times number of atoms)

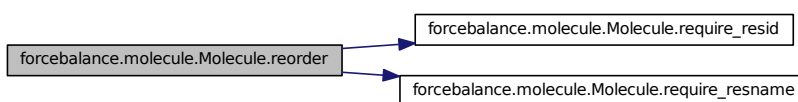
Definition at line 224 of file molecule.py.

10.11.3.5 def forcebalance.molecule.Molecule.reorder (self, idx_name = None)

Reorders an xyz file using data provided from an .ndx file.

Definition at line 369 of file molecule.py.

Here is the call graph for this function:



10.11.4 Member Data Documentation

10.11.4.1 forcebalance::molecule.Molecule::elem

File type can be determined from the file name using the extension.

Absolutely required; a list of chemical elements (number of atoms)

Definition at line 144 of file molecule.py.

10.11.4.2 forcebalance::molecule.Molecule::index

If supplied with a file name, read in stuff.

Initialize the index file here

Definition at line 121 of file molecule.py.

The documentation for this class was generated from the following file:

- forcebalance/forcebalance/molecule.py

10.12 CallGraph.node Class Reference

Data structure for holding information about python objects.

Public Member Functions

- def [__init__](#)
Constructor.

Public Attributes

- **oid**
- **name**
- **parent**
- **dtype**

10.12.1 Detailed Description

Data structure for holding information about python objects.

Definition at line 20 of file CallGraph.py.

The documentation for this class was generated from the following file:

- forcebalance/doc/callgraph/CallGraph.py

10.13 forcebalance.optimizer.Optimizer Class Reference

[Optimizer](#) class.

Public Member Functions

- def [__init__](#)
Instantiation of the optimizer.
- def [Run](#)
Call the appropriate optimizer.
- def [MainOptimizer](#)
The main ForceBalance trust-radius optimizer.
- def [step](#)
Computes the Newton-Raphson or BFGS step.
- def [NewtonRaphson](#)
Optimize the force field parameters using the Newton-Raphson method (.
- def [BFGS](#)
Optimize the force field parameters using the BFGS method; currently the recommended choice (.
- def [ScipyOptimizer](#)
Driver for SciPy optimizations.
- def [Simplex](#)
Use SciPy's built-in simplex algorithm to optimize the parameters.
- def [Powell](#)
Use SciPy's built-in Powell direction-set algorithm to optimize the parameters.
- def [Anneal](#)
Use SciPy's built-in simulated annealing algorithm to optimize the parameters.
- def [ConjugateGradient](#)
Use SciPy's built-in simulated annealing algorithm to optimize the parameters.
- def [Scan_Values](#)
Scan through parameter values.
- def [ScanMVals](#)

- *Scan through the mathematical parameter space.*
- def [ScanPVals](#)
Scan through the physical parameter space.
- def [SinglePoint](#)
A single-point objective function computation.
- def [Gradient](#)
A single-point gradient computation.
- def [Hessian](#)
A single-point Hessian computation.
- def [FDCheckG](#)
Finite-difference checker for the objective function gradient.
- def [FDCheckH](#)
Finite-difference checker for the objective function Hessian.
- def **readchk**
- def **writechk**

Public Attributes

- [OptTab](#)
A list of all the things we can ask the optimizer to do.
- [root](#)
The root directory.
- [jobtype](#)
The job type.
- [trust0](#)
Initial step size trust radius.
- [eps](#)
Lower bound on Hessian eigenvalue (below this, we add in steepest descent)
- [h](#)
Step size for numerical finite difference.
- [conv_obj](#)
Function value convergence threshold.
- [conv_stp](#)
Step size convergence threshold.
- [conv_grd](#)
Gradient convergence threshold.
- [maxstep](#)
Maximum number of optimization steps.
- [idxnum](#)
For scan[mp]vals: The parameter index to scan over.
- [idxname](#)
For scan[mp]vals: The parameter name to scan over, it just looks up an index.
- [scan_vals](#)
For scan[mp]vals: The values that are fed into the scanner.
- [rchk_fnm](#)
Name of the checkpoint file that we're reading in.
- [wchk_fnm](#)

- `wchk_step`
Name of the checkpoint file that we're writing out.
- `Objective`
Whether to write the checkpoint file at every step.
- `Sims`
The objective function (needs to pass in when I instantiate)
- `FF`
The fitting simulations.
- `excision`
The force field itself.
- `np`
The indices to be excluded from the Hessian update.
- `mvals0`
Number of parameters.
- `chk`
The original parameter values.
- `chk`
Put data into the checkpoint file.

10.13.1 Detailed Description

`Optimizer` class.

Contains several methods for numerical optimization.

For various reasons, the optimizer depends on the force field and fitting simulations (i.e. we cannot treat it as a fully independent numerical optimizer). The dependency is rather weak which suggests that I can remove it someday.

Definition at line 25 of file optimizer.py.

10.13.2 Constructor & Destructor Documentation

10.13.2.1 `def forcebalance.optimizer.Optimizer.__init__(self, options, Objective, FF, Simulations)`

Instantiation of the optimizer.

The optimizer depends on both the FF and the fitting simulations so there is a chain of dependencies: FF --> FitSim --> `Optimizer`, and FF --> `Optimizer`

Here's what we do:

- Take options from the parser
- Pass in the objective function, force field, all fitting simulations

Definition at line 38 of file optimizer.py.

10.13.3 Member Function Documentation

10.13.3.1 `def forcebalance.optimizer.Optimizer.Anneal(self)`

Use SciPy's built-in simulated annealing algorithm to optimize the parameters.

See also

[Optimizer::ScipyOptimizer](#)

Definition at line 428 of file optimizer.py.

Here is the call graph for this function:



10.13.3.2 def forcebalance.optimizer.Optimizer.BFGS (self)

Optimize the force field parameters using the BFGS method; currently the recommended choice (.

See also

[MainOptimizer](#))

Definition at line 353 of file optimizer.py.

Here is the call graph for this function:



10.13.3.3 def forcebalance.optimizer.Optimizer.ConjugateGradient (self)

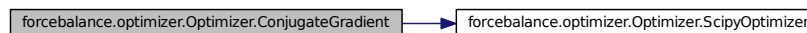
Use SciPy's built-in simulated annealing algorithm to optimize the parameters.

See also

[Optimizer::ScipyOptimizer](#)

Definition at line 433 of file optimizer.py.

Here is the call graph for this function:



10.13.3.4 def forcebalance.optimizer.Optimizer.FDCheckG (self)

Finite-difference checker for the objective function gradient.

For each element in the gradient, use a five-point finite difference stencil to compute a finite-difference derivative, and compare it to the analytic result.

Definition at line 528 of file optimizer.py.

10.13.3.5 def forcebalance.optimizer.Optimizer.FDCheckH (self)

Finite-difference checker for the objective function Hessian.

For each element in the Hessian, use a five-point stencil in both parameter indices to compute a finite-difference derivative, and compare it to the analytic result.

This is meant to be a foolproof checker, so it is pretty slow. We could write a faster checker if we assumed we had accurate first derivatives, but it's better to not make that assumption.

The second derivative is computed by double-wrapping the objective function via the 'wrap2' function.

Definition at line 560 of file optimizer.py.

10.13.3.6 def forcebalance.optimizer.Optimizer.Gradient (self)

A single-point gradient computation.

Definition at line 506 of file optimizer.py.

10.13.3.7 def forcebalance.optimizer.Optimizer.Hessian (self)

A single-point Hessian computation.

Definition at line 514 of file optimizer.py.

10.13.3.8 def forcebalance.optimizer.Optimizer.MainOptimizer (self, b_BFGS = 0)

The main ForceBalance trust-radius optimizer.

Usually this function is called with the BFGS or NewtonRaphson method. I've found the BFGS method to be most efficient, especially when we don't have access to the expensive analytic second derivatives of the objective function. If we are computing derivatives by finite difference (which we often do), then the diagonal elements of the second derivative can also be obtained by taking a central difference.

BFGS is a pseudo-Newton method in the sense that it builds an approximate Hessian matrix from the gradient information in previous steps; true Newton-Raphson needs all of the second derivatives. However, the algorithms are similar in that they both compute the step by inverting the Hessian and multiplying by the gradient.

As this method iterates toward convergence, it computes BFGS updates of the Hessian matrix and adjusts the step size. If the step is good (i.e. the objective function goes down), then the step size is increased; if the step is bad, then it steps back to the original point and tries again with a smaller step size.

The optimization is terminated after either a function value or step size tolerance is reached.

Parameters

<code>in</code>	<code>b_BFGS</code>	Switch to use BFGS (True) or Newton-Raphson (False)
-----------------	---------------------	---

Definition at line 196 of file optimizer.py.

10.13.3.9 def forcebalance.optimizer.Optimizer.NewtonRaphson (self)

Optimize the force field parameters using the Newton-Raphson method (.

See also

[MainOptimizer](#))

Definition at line 348 of file optimizer.py.

Here is the call graph for this function:



10.13.3.10 def forcebalance.optimizer.Optimizer.Powell (self)

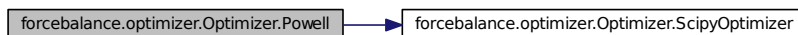
Use SciPy's built-in Powell direction-set algorithm to optimize the parameters.

See also

[Optimizer::ScipyOptimizer](#)

Definition at line 423 of file optimizer.py.

Here is the call graph for this function:



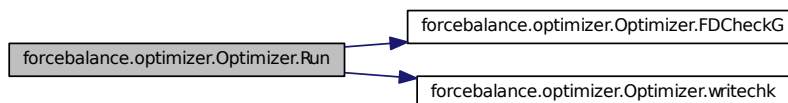
10.13.3.11 def forcebalance.optimizer.Optimizer.Run (self)

Call the appropriate optimizer.

This is the method we might want to call from an executable.

Definition at line 137 of file optimizer.py.

Here is the call graph for this function:



10.13.3.12 `def forcebalance.optimizer.Optimizer.Scan_Values (self, MathPhys = 1)`

Scan through parameter values.

This option is activated using the inputs:

```
scan[mp]vals
scan_vals low:hi:nsteps
scan_idxnum (number) -or-
scan_idxname (name)
```

This method goes to the specified parameter indices and scans through the supplied values, evaluating the objective function at every step.

I hope this method will be useful for people who just want to look at changing one or two parameters and seeing how it affects the force field performance.

Todo Maybe a multidimensional grid can be done.

Parameters

<code>in</code>	<code>MathPhys</code>	Switch to use mathematical (True) or physical (False) parameters.
-----------------	-----------------------	---

Definition at line 459 of file optimizer.py.

10.13.3.13 `def forcebalance.optimizer.Optimizer.ScanMVals (self)`

Scan through the mathematical parameter space.

See also

`Optimizer::ScanValues`

Definition at line 490 of file optimizer.py.

Here is the call graph for this function:

10.13.3.14 `def forcebalance.optimizer.Optimizer.ScanPVals (self)`

Scan through the physical parameter space.

See also

`Optimizer::ScanValues`

Definition at line 495 of file optimizer.py.

Here is the call graph for this function:



10.13.3.15 def forcebalance.optimizer.Optimizer.ScipyOptimizer (self, Algorithm = "None")

Driver for SciPy optimizations.

Using any of the SciPy optimizers requires that SciPy is installed. This method first defines several wrappers around the objective function that the SciPy optimizers can use. Then it calls the algorithm itself.

Parameters

in	Algorithm	The optimization algorithm to use, for example 'powell', 'simplex' or 'anneal'
----	-----------	--

Definition at line 366 of file optimizer.py.

10.13.3.16 def forcebalance.optimizer.Optimizer.Simplex (self)

Use SciPy's built-in simplex algorithm to optimize the parameters.

See also

[Optimizer::ScipyOptimizer](#)

Definition at line 418 of file optimizer.py.

Here is the call graph for this function:



10.13.3.17 def forcebalance.optimizer.Optimizer.SinglePoint (self)

A single-point objective function computation.

Definition at line 500 of file optimizer.py.

10.13.3.18 def forcebalance.optimizer.Optimizer.step (self, G, H, trust)

Computes the Newton-Raphson or BFGS step.

The step is given by the inverse of the Hessian times the gradient. There are some extra considerations here:

First, certain eigenvalues of the Hessian may be negative. Then the NR optimization will take us to a saddle point and not a true minimum. In these instances, we mix in some steepest descent by adding a multiple of the identity matrix to the Hessian.

Second, certain eigenvalues may be very small. If the Hessian is close to being singular, then we also add in some steepest descent.

Third, certain components of the gradient / Hessian are strictly zero, or they are excluded from our optimization. These components are explicitly deleted when we do the Hessian inversion, and reinserted as a zero in the step.

Fourth, we rescale the step size back to the trust radius.

Parameters

in	G	The gradient
in	H	The Hessian
in	$trust$	The trust radius

Definition at line 327 of file optimizer.py.

10.13.4 Member Data Documentation

10.13.4.1 forcebalance::optimizer.Optimizer::mvals0

The original parameter values.

Sometimes the optimizer doesn't return anything (i.e.

in the case of a single point calculation) In these situations, don't do anything Check derivatives by finite difference after the optimization is over (for good measure)

Definition at line 60 of file optimizer.py.

10.13.4.2 forcebalance::optimizer.Optimizer::OptTab

A list of all the things we can ask the optimizer to do.

Print the optimizer options.

Load the checkpoint file. A list of all the things we can ask the optimizer to do.

Definition at line 39 of file optimizer.py.

The documentation for this class was generated from the following file:

- forcebalance/forcebalance/optimizer.py

10.14 forcebalance.project.Project Class Reference

Container for a ForceBalance force field optimization project.

Public Member Functions

- def [__init__](#)
Instantiation of a ForceBalance force field optimization project.
- def [Objective](#)
Objective function defined within [Project](#); can you think of a better place?

Public Attributes

- [sim_opts](#)
The general options and simulation options that come from parsing the input file.
- [FF](#)
The force field component of the project.
- [Simulations](#)
The list of fitting simulations.
- [Optimizer](#)
The optimizer component of the project.

10.14.1 Detailed Description

Container for a ForceBalance force field optimization project.

The triumvirate or trinity of components are:

- The force field
- The objective function
- The optimizer

The force field is a class defined in [forcefield.py](#). The objective function is built here as a combination of fitting simulation classes. The optimizer is a class defined in this file.

Definition at line 26 of file project.py.

10.14.2 Constructor & Destructor Documentation

10.14.2.1 `def forcebalance.project.Project.__init__(self, input_file)`

Instantiation of a ForceBalance force field optimization project.

Here's what we do:

- Parse the input file
- Create an instance of the force field
- Create a list of fitting simulation instances
- Create an optimizer instance
- Print out the general options

Definition at line 40 of file project.py.

10.14.3 Member Function Documentation

10.14.3.1 `def forcebalance.project.Project.Objective (self, mvals, Order=0, usepvals=False, verbose=False)`

Objective function defined within [Project](#); can you think of a better place?

The objective function is a combination of contributions from the different fitting simulations. Basically, it loops through the fitting simulations, gets their contributions to the objective function and then sums all of them (although more elaborate schemes are conceivable). The return value is the same data type as calling the fitting simulation itself: a dictionary containing the objective function, the gradient and the Hessian.

The penalty function is also computed here; it keeps the parameters from straying too far from their initial values.

Parameters

<code>in</code>	<code>mvals</code>	The mathematical parameters that enter into computing the objective function
<code>in</code>	<code>Order</code>	The requested order of differentiation
<code>in</code>	<code>usepvals</code>	Switch that determines whether to use physical parameter values

Definition at line 70 of file `project.py`.

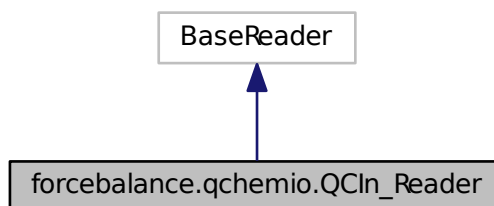
The documentation for this class was generated from the following file:

- `forcebalance/forcebalance/project.py`

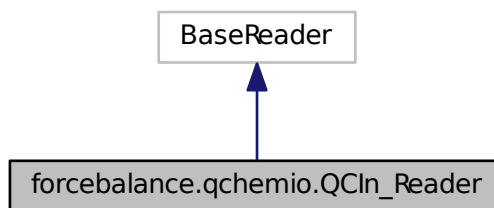
10.15 forcebalance.qchemio.QCIn_Reader Class Reference

Finite state machine for parsing Q-Chem input files.

Inheritance diagram for `forcebalance.qchemio.QCIn_Reader`:



Collaboration diagram for forcebalance.qchemio.QCIn_Reader:



Public Member Functions

- def `__init__`
- def `feed`
Feed in a line.

Public Attributes

- `atom`
- `snum`
- `cnum`
- `shell`
- `pdict`
- `sec`
- `itype`
- `suffix`

10.15.1 Detailed Description

Finite state machine for parsing Q-Chem input files.

Definition at line 26 of file `qchemio.py`.

10.15.2 Member Function Documentation

10.15.2.1 def forcebalance.qchemio.QCIn_Reader.feed (self, line)

Feed in a line.

Parameters

<code>in</code>	<code>line</code>	The line of data
-----------------	-------------------	------------------

Definition at line 43 of file `qchemio.py`.

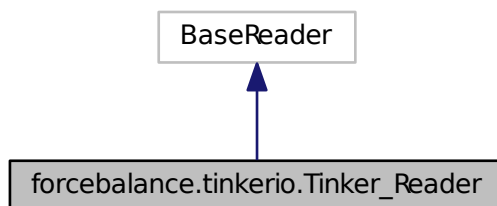
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/qchemio.py

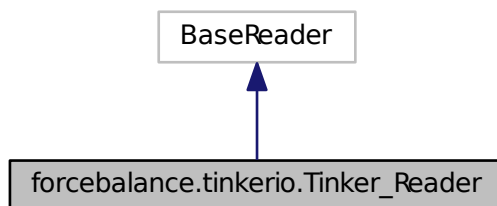
10.16 forcebalance.tinkerio.Tinker_Reader Class Reference

Finite state machine for parsing TINKER force field files.

Inheritance diagram for forcebalance.tinkerio.Tinker_Reader:



Collaboration diagram for forcebalance.tinkerio.Tinker_Reader:



Public Member Functions

- def `__init__`
- def `feed`

Given a line, determine the interaction type and the atoms involved (the suffix).

Public Attributes

- `pdict`

The parameter dictionary (defined in this file)

- [atom](#)

The atom numbers in the interaction (stored in the TINKER parser)

- [itype](#)
- [suffix](#)

10.16.1 Detailed Description

Finite state machine for parsing TINKER force field files.

This class is instantiated when we begin to read in a file. The `feed(line)` method updates the state of the machine, informing it of the current interaction type. Using this information we can look up the interaction type and parameter type for building the parameter ID.

Definition at line 53 of file `tinkerio.py`.

10.16.2 Member Function Documentation

10.16.2.1 `def forcebalance.tinkerio.Tinker_Reader.feed (self, line)`

Given a line, determine the interaction type and the atoms involved (the suffix).

TINKER generally has stuff like this:

```

bond-cubic          -2.55
bond-quartic        3.793125

vdw      1          3.4050    0.1100
vdw      2          2.6550    0.0135    0.910 # PARM 4

multipole    2      1      2          0.25983
              -0.03859    0.00000    -0.05818
              -0.03673
              0.00000    -0.10739
              -0.00203    0.00000    0.14412

```

The '#PARM 4' has no effect on TINKER but it indicates that we are tuning the fourth field on the line (the 0.910 value).

Todo Put the rescaling factors for TINKER parameters in here. Currently we're using the initial value to determine the rescaling factor which is not very good.

Every parameter line is prefaced by the interaction type except for 'multipole' which is on multiple lines. Because the lines that come after 'multipole' are predictable, we just determine the current line using the previous line.

Random note: Unit of force is kcal / mole / angstrom squared.

Definition at line 97 of file `tinkerio.py`.

The documentation for this class was generated from the following file:

- `forcebalance/forcebalance/tinkerio.py`