ForceBalance version 0.11.0

Generated by Doxygen 1.7.6.1

Contents

1 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

This can be generalized to parameters that don't correspond to atoms.

Fix the special treatment of NDDO.

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.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 gmxio

Even more stuff from forcefield.py needs to go into here.

Namespace optimizer

I might want to sample over different force fields and store past parameters

Pickle-loading is helpful mainly for non-initial parameter values, for reproducibility

Read in parameters from input file, that would be nice

2 Namespace Index

2.1 Namespace List

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

3 Class Index

3.1 Class List

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

4 Namespace Documentation

4.1 BaseReader Namespace Reference

Base class for force field line reader.

4.1.1 Detailed Description

Base class for force field line reader.

Author

Lee-Ping Wang

Date

12/2011

4.2 counterpoisematch Namespace Reference

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

4.2.1 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

4.3 custom_io Namespace Reference

Custom force field parser.

4.3.1 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

4.4 forceenergymatch_gmxx2 Namespace Reference

Force and energy matching module.

4.4.1 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

4.5 forcefield Namespace Reference

Force field module.

4.5.1 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

4.6 gmxio Namespace Reference

GROMACS input/output.

4.6.1 Detailed Description

GROMACS input/output.

Todo Even more stuff from forcefield.py needs to go into here.

Author

Lee-Ping Wang

Date

12/2011

4.7 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.

4.7.1 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? :)

4.7.2 Function Documentation

4.7.2.1 def MakeInputFile.main ( )

Print out all of the options available to ForceBalance.

Definition at line 18 of file MakeInputFile.py.

4.8 molecule Namespace Reference

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

4.8.1 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

4.9 nifty Namespace Reference

Nifty functions, intended to be imported by any module.

4.9.1 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

4.10 OptimizePotential Namespace Reference

Executable script for starting ForceBalance.

Functions

- def main

  Instantiate a ForceBalance project and call the optimizer.

4.10.1 Detailed Description

Executable script for starting ForceBalance.

4.10.2 Function Documentation

4.10.2.1 def OptimizePotential.main ( )

Instantiate a ForceBalance project and call the optimizer.

Definition at line 13 of file OptimizePotential.py.

4.11 optimizer Namespace Reference

Optimization algorithms.

4.11.1 Detailed Description

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

Todo I might want to sample over different force fields and store past parameters

Pickle-loading is helpful mainly for non-initial parameter values, for reproducibility

Read in parameters from input file, that would be nice

Author

Lee-Ping Wang

Date

12/2011

4.12 ParseInputFile Namespace Reference

Read in ForceBalance input file and print it back out.

Functions

- def main

  Input file parser for ForceBalance program.

4.12.1 Detailed Description

Read in ForceBalance input file and print it back out.

4.12.2 Function Documentation

4.12.2.1 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.

4.13 parser Namespace Reference

Input file parser for ForceBalance projects.

4.13.1 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

4.14 project Namespace Reference

ForceBalance force field optimization project.

4.14.1 Detailed Description

ForceBalance force field optimization project.

4.15 qchemio Namespace Reference

Q-Chem input file parser.

4.15.1 Detailed Description

Q-Chem input file parser.

4.16 simtab Namespace Reference

Contains the dictionary of fitting simulation classes.

4.16.1 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.

4.17 tinkerio Namespace Reference

TINKER input/output.

4.17.1 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

5 Class Documentation

5.1 forcebalance.basereader.BaseReader Class Reference

Public Member Functions

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

  Returns the parameter type (e.g.

Public Attributes

- **ln**
- **itype**
- **suffix**
- **pdict**

5.1.1 Detailed Description

Definition at line 8 of file basereader.py.

5.1.2 Member Function Documentation

5.1.2.1 def forcebalance.basereader.BaseReader.build_pid (*self, pfld*)

Returns the parameter type (e.g.

K in BONDSK) based on the current interaction type.

Both the 'pdict' dictionary (see gmxio.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.

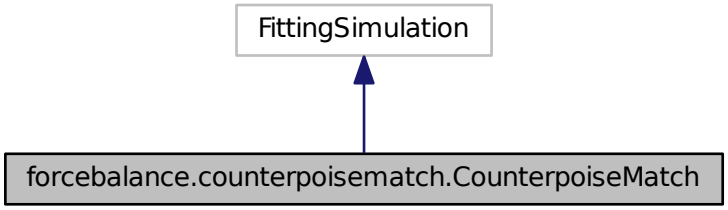
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/basereader.py

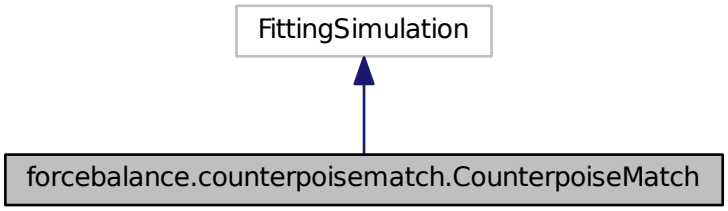
5.2 forcebalance.counterpoisematch.CounterpoiseMatch Class Reference

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.

5.2.1 Detailed Description

FittingSimulation subclass for matching the counterpoise correction.

Definition at line 31 of file counterpoisematch.py.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 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.

5.2.3 Member Function Documentation

5.2.3.1 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`	*mvals*	Mathematical parameter values
`in`	*AGrad*	Switch to turn on analytic gradient (not implemented)
`in`	*AHess*	Switch to turn on analytic Hessian (not implemented)
`in`	*tempdir*	Temporary directory for running computation

Returns

Answer Contribution to the objective function

Definition at line 123 of file counterpoisematch.py.

5.2.3.2 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.

5.2.3.3 def forcebalance.counterpoisematch.CounterpoiseMatch.loadxyz (*self, fnm*)

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

Parameters

in	*fnm*	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

5.3 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.

- tmI

  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**

5.3.1 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.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 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.

5.3.3 Member Function Documentation

5.3.3.1 def forcebalance.forcefield.FF.addff (*self, ffname*)

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.

Todo This can be generalized to parameters that don't correspond to atoms.

Fix the special treatment of NDDO.

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 `'ANGLESKCACBO'`.

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_ANGLESKCACAH /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 `ANGLESKCACAH`. 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

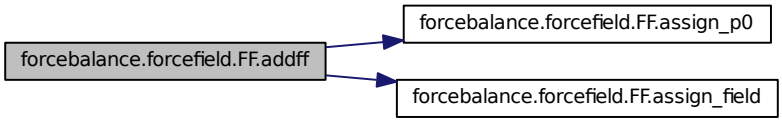
My program currently assumes that we are only using one MM program per job. If we use CHARMM and GROMACS to perform fitting simulations in the same job, we will get f-ed up. Maybe this needs to be fixed in the future, with program prefixes to parameters like C_ , G_ .. or simply unit conversions, you 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 304 of file forcefield.py.

Here is the call graph for this function:



5.3.3.2 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	*idx*	The index of the parameter.
in	*fnm*	The file name of the parameter field.
in	*ln*	The line number within the file.
in	*pfld*	The field within the line.

in	*mult*	The multiplier (this is usually 1.0)

Definition at line 611 of file forcefield.py.

5.3.3.3 def forcebalance.forcefield.FF.assign_p0 (*self, idx, val*)

Assign physical parameter values to the 'pvals0' array.

Parameters

in	*idx*	The index to which we assign the parameter value.
in	*val*	The parameter value to be inserted.

Definition at line 593 of file forcefield.py.

5.3.3.4 def forcebalance.forcefield.FF.create_mvals (*self, pvals*)

Converts physical to mathematical parameters.

We create the inverse transformation matrix using SVD.

Parameters

in	*pvals*	The physical parameters

Returns

mvals The mathematical parameters

Definition at line 397 of file forcefield.py.

5.3.3.5 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	*mvals*	The mathematical parameters

Returns

pvals The physical parameters

Definition at line 385 of file forcefield.py.

5.3.3.6 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 573 of file forcefield.py.

5.3.3.7 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.

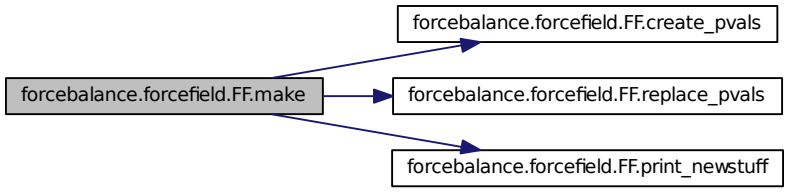
It does NOT store the mathematical parameters in the class state (since we can only hold one set of parameters).

Parameters

in	*printdir*	The directory that the force fields are printed to; as usual this is relative to the project root directory.
in	*vals*	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	*usepvals*	Switch for whether to bypass the coordinate transformation and use physical parameters directly.

Definition at line 365 of file forcefield.py.

Here is the call graph for this function:



5.3.3.8 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 504 of file forcefield.py.

5.3.3.9 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 582 of file forcefield.py.

5.3.3.10 def forcebalance.forcefield.FF.print_newstuff (*self, printdir*)

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

Parameters

in	*printdir*	The directory to which new force fields are printed.

Definition at line 428 of file forcefield.py.

5.3.3.11 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 407 of file forcefield.py.

5.3.3.12 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.

Parameters

in	*printfacs*	List for printing out the resecaling factors

Definition at line 443 of file forcefield.py.

5.3.4 Member Data Documentation

5.3.4.1 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]) constemp = 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.

5.3.4.2 forcebalance::forcefield.FF::newstuff

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

Definition at line 183 of file forcefield.py.

5.3.4.3 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.

5.3.4.4 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','BONDSK','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

5.4 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.

5.4.1 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 node: 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 FitSim 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 faciliate 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.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 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.

5.4.3 Member Function Documentation

5.4.3.1 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.

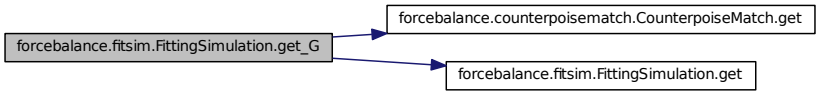
5.4.3.2 def forcebalance.fitsim.FittingSimulation.get_G (*self, mvals* = `None`)

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:



5.4.3.3 def forcebalance.fitsim.FittingSimulation.get_H (*self, mvals* = None)

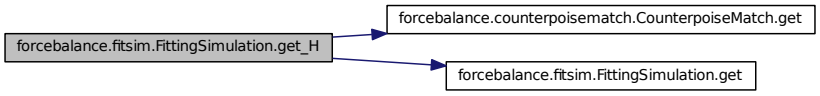
Computes the objective function contribution and its gradient / Hessian.

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:



5.4.3.4 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.

5.4.4 Member Data Documentation

5.4.4.1 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.

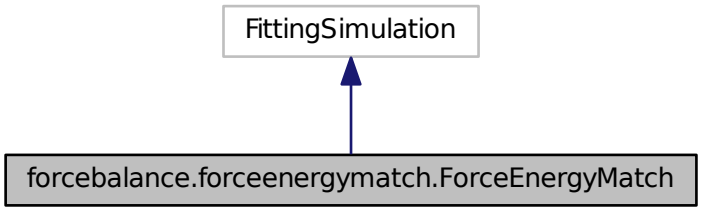
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/fitsim.py

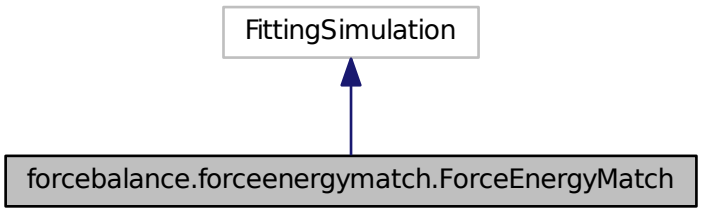
5.5 forcebalance.forceenergymatch.ForceEnergyMatch Class Reference

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.

5.5.1 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.

5.5.2 Constructor & Destructor Documentation

5.5.2.1 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.

5.5.3 Member Function Documentation

5.5.3.1 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 ...], 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

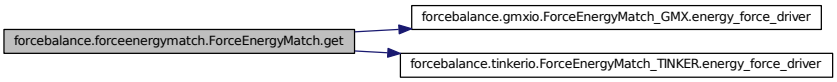
in	*mvals*	Mathematical parameter values
in	*AGrad*	Switch to turn on analytic gradient, useless here
in	*AHess*	Switch to turn on analytic Hessian, useless here
in	*tempdir*	Temporary directory for running computation

Returns

Answer Contribution to the objective function

Definition at line 307 of file forceenergymatch.py.

Here is the call graph for this function:



5.5.3.2 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 multipled 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)) 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.

5.5.4 Member Data Documentation

5.5.4.1 forcebalance::forceenergymatch.ForceEnergyMatch::efweight

The proportion of energy vs.

force.

Definition at line 58 of file forceenergymatch.py.

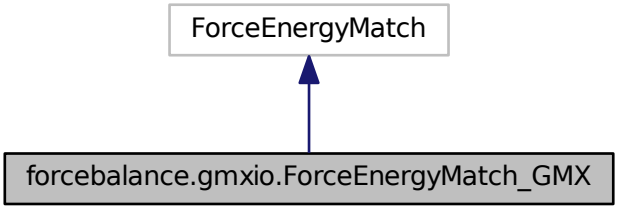
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/forceenergymatch.py

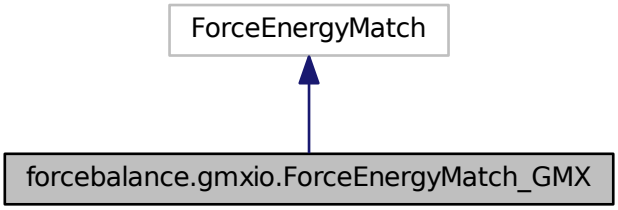
5.6 forcebalance.gmxio.ForceEnergyMatch_GMX Class Reference

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.

5.6.1 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.

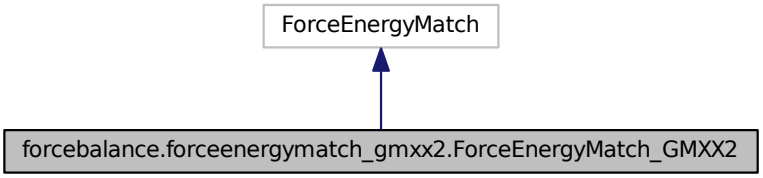
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/gmxio.py

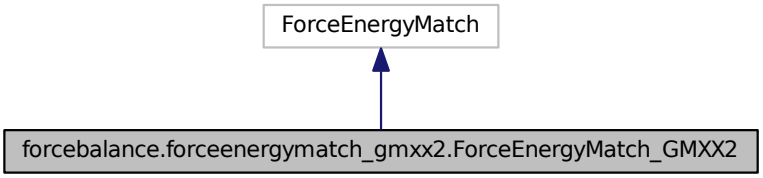
5.7 forcebalance.forceenergymatch_gmxx2.ForceEnergyMatch_GMXX2 Class Reference

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**

5.7.1 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.

5.7.2 Constructor & Destructor Documentation

5.7.2.1 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.

5.7.3 Member Function Documentation

5.7.3.1 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	*mvals*	Mathematical parameter values
in	*AGrad*	Switch to turn on analytic gradient
in	*AHess*	Switch to turn on analytic Hessian
in	*tempdir*	Temporary directory for running computation

Returns

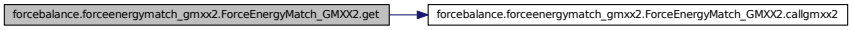
Answer Contribution to the objective function

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:



5.7.3.2 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

in	*tempdir*	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.

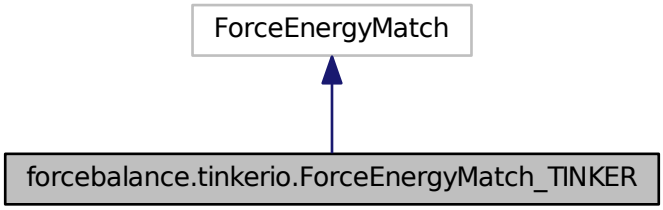
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/forceenergymatch_gmxx2.py

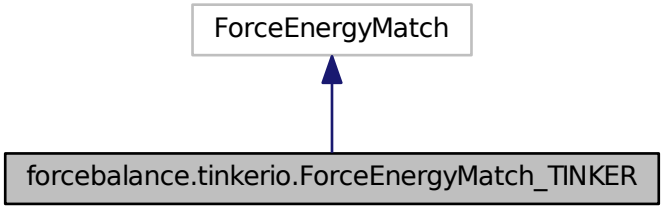
5.8 forcebalance.tinkerio.ForceEnergyMatch_TINKER Class Reference

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.

5.8.1 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.

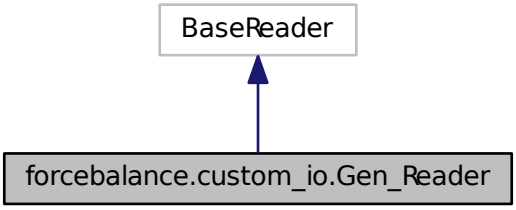
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/tinkerio.py

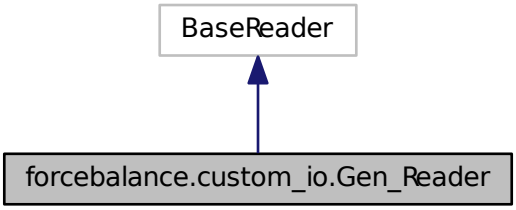
5.9 forcebalance.custom_io.Gen_Reader Class Reference

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**

5.9.1 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.

5.9.2 Member Function Documentation

5.9.2.1 def forcebalance.custom_io.Gen_Reader.feed (*self, line*)

Feed in a line.

Parameters

in	*line*	The line of data

Definition at line 57 of file custom_io.py.

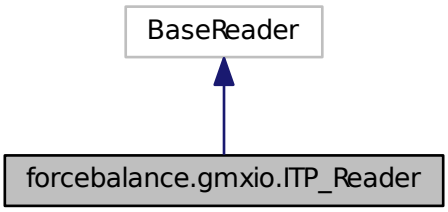
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/custom_io.py

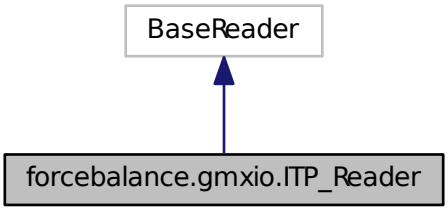
5.10 forcebalance.gmxio.ITP_Reader Class Reference

Finite state machine for parsing GROMACS force field files.

Inheritance diagram for forcebalance.gmxio.ITP_Reader:



Collaboration diagram for forcebalance.gmxio.ITP_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**

5.10.1 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.

5.10.2 Member Function Documentation

5.10.2.1 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.

5.10.3 Member Data Documentation

5.10.3.1 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

5.11 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.

5.11.1 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.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 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.

5.11.3 Member Function Documentation

5.11.3.1 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.

5.11.3.2 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.

5.11.3.3 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.

5.11.3.4 def forcebalance.molecule.Molecule.read_xyz (*self, fnm*)

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

Parameters

in	*fnm*	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)

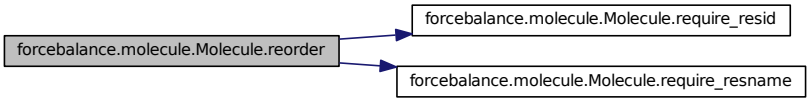
Definition at line 224 of file molecule.py.

5.11.3.5 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:



5.11.4 Member Data Documentation

5.11.4.1 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.

5.11.4.2 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

5.12 CallGraph.node Class Reference

Data structure for holding information about python objects.

Public Member Functions

- def __init__
  Constructor.

Public Attributes

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

5.12.1 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

5.13 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
  Name of the checkpoint file that we're writing out.
- wchk_step
  Whether to write the checkpoint file at every step.
- Objective
  The objective function (needs to pass in when I instantiate)
- Sims
  The fitting simulations.
- FF
  The force field itself.
- excision
  The indices to be excluded from the Hessian update.
- np
  Number of parameters.
- mvals0
  The original parameter values.
- chk
  Put data into the checkpoint file.

5.13.1 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 29 of file optimizer.py.

5.13.2 Constructor & Destructor Documentation

5.13.2.1 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 42 of file optimizer.py.

5.13.3 Member Function Documentation

5.13.3.1 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 432 of file optimizer.py.

Here is the call graph for this function:

5.13.3.2 def forcebalance.optimizer.Optimizer.BFGS (*self*)

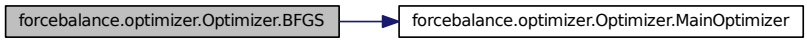
Optimize the force field parameters using the BFGS method; currently the recommended choice (.

See also

MainOptimizer)

Definition at line 357 of file optimizer.py.

Here is the call graph for this function:



5.13.3.3 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 437 of file optimizer.py.

Here is the call graph for this function:

5.13.3.4 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 532 of file optimizer.py.

5.13.3.5 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 564 of file optimizer.py.

5.13.3.6 def forcebalance.optimizer.Optimizer.Gradient (*self*)

A single-point gradient computation.

Definition at line 510 of file optimizer.py.

5.13.3.7 def forcebalance.optimizer.Optimizer.Hessian (*self*)

A single-point Hessian computation.

Definition at line 518 of file optimizer.py.

5.13.3.8 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

in	*b_BFGS*	Switch to use BFGS (True) or Newton-Raphson (False)

Definition at line 200 of file optimizer.py.

5.13.3.9 def forcebalance.optimizer.Optimizer.NewtonRaphson (*self*)

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

See also

MainOptimizer)

Definition at line 352 of file optimizer.py.

Here is the call graph for this function:

5.13.3.10 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 427 of file optimizer.py.

Here is the call graph for this function:

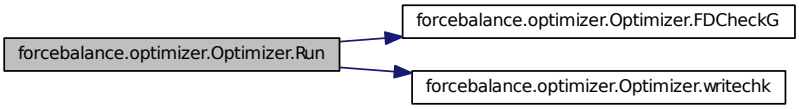
5.13.3.11 def forcebalance.optimizer.Optimizer.Run (*self*)

Call the appropriate optimizer.

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

Definition at line 141 of file optimizer.py.

Here is the call graph for this function:



5.13.3.12 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

`in`	*MathPhys*	Switch to use mathematical (True) or physical (False) parameters.

Definition at line 463 of file optimizer.py.

5.13.3.13 def forcebalance.optimizer.Optimizer.ScanMVals (*self*)

Scan through the mathematical parameter space.

See also

Optimizer::ScanValues

Definition at line 494 of file optimizer.py.

Here is the call graph for this function:

5.13.3.14 def forcebalance.optimizer.Optimizer.ScanPVals (*self*)

Scan through the physical parameter space.

See also

Optimizer::ScanValues

Definition at line 499 of file optimizer.py.

Here is the call graph for this function:

5.13.3.15 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 algorith mitself.

Parameters

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

Definition at line 370 of file optimizer.py.

5.13.3.16 def forcebalance.optimizer.Optimizer.Simplex (*self*)

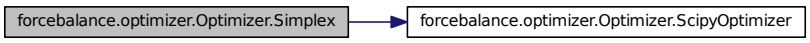
Use SciPy's built-in simplex algorithm to optimize the parameters.

See also

Optimizer::ScipyOptimizer

Definition at line 422 of file optimizer.py.

Here is the call graph for this function:



5.13.3.17 def forcebalance.optimizer.Optimizer.SinglePoint (*self*)

A single-point objective function computation.

Definition at line 504 of file optimizer.py.

5.13.3.18 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 331 of file optimizer.py.

5.13.4 Member Data Documentation

5.13.4.1 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 64 of file optimizer.py.

5.13.4.2 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 43 of file optimizer.py.

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

- forcebalance/forcebalance/optimizer.py

5.14 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.

5.14.1 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.

5.14.2 Constructor & Destructor Documentation

5.14.2.1 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.

5.14.3 Member Function Documentation

5.14.3.1 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

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

Definition at line 70 of file project.py.

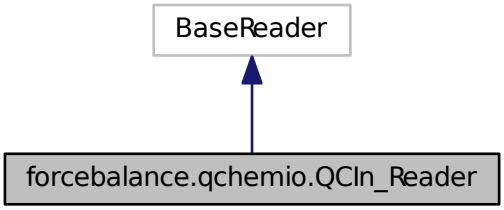
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/project.py

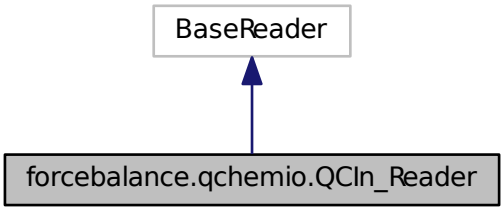
5.15 forcebalance.qchemio.QCIn_Reader Class Reference

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**

5.15.1 Detailed Description

Finite state machine for parsing Q-Chem input files.

Definition at line 26 of file qchemio.py.

5.15.2 Member Function Documentation

5.15.2.1 def forcebalance.qchemio.QCIn_Reader.feed (*self, line*)

Feed in a line.

Parameters

in	*line*	The line of data

Definition at line 43 of file qchemio.py.

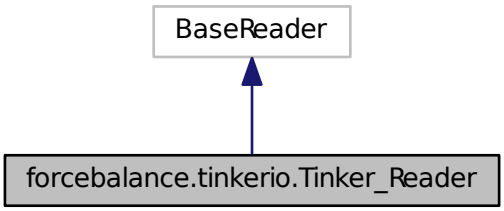
The documentation for this class was generated from the following file:

- forcebalance/forcebalance/qchemio.py

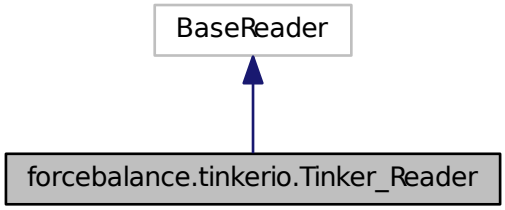
5.16 forcebalance.tinkerio.Tinker_Reader Class Reference

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**

5.16.1 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.

5.16.2 Member Function Documentation

5.16.2.1 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