ForceBalance version 0.11.0

Generated by Doxygen 1.7.6.1

Mon Dec 19 2011 17:36:08

Contents

1 Main Page 1

1.1 Introduction . . . 1
1.2 to-do list . . . 1
1.3 Glossary . . . 1
1.4 Usage . . . 2

2 Todo List 2

3 Namespace Index 3

3.1 Namespace List . . . 3

4 Class Index 4

4.1 Class Hierarchy . . . 4

5 Class Index 5

5.1 Class List . . . 5

6 Namespace Documentation 5

6.1 counterpoisematch Namespace Reference . . . 5
6.1.1 Detailed Description . . . 5
6.2 custom_io Namespace Reference . . . 6
6.2.1 Detailed Description . . . 6
6.2.2 Function Documentation . . . 6
6.2.3 Variable Documentation . . . 7
6.3 forceenergymatch_gmx Namespace Reference . . . 7
6.3.1 Detailed Description . . . 7
6.3.2 Function Documentation . . . 8
6.4 forcefield Namespace Reference . . . 8
6.4.1 Detailed Description . . . 8
6.4.2 Function Documentation . . . 9
6.5 gmxio Namespace Reference . . . 10
6.5.1 Detailed Description . . . 10
6.5.2 Function Documentation . . . 11
6.5.3 Variable Documentation . . . 12
6.6 nifty Namespace Reference . . . 13
6.6.1 Detailed Description . . . 14
6.6.2 Function Documentation . . . 15

6.7 optimization Namespace Reference . . . 18
6.7.1 Detailed Description . . . 18
6.8 parser Namespace Reference . . . 19
6.8.1 Detailed Description . . . 19
6.8.2 Function Documentation . . . 20
6.8.3 Variable Documentation . . . 21
6.9 project Namespace Reference . . . 22
6.9.1 Detailed Description . . . 22
6.10 simtab Namespace Reference . . . 22
6.10.1 Detailed Description . . . 23
6.10.2 Variable Documentation . . . 23

7 Class Documentation 23

7.1 counterpoisematch.CounterpoiseMatch Class Reference . . . 23
7.1.1 Detailed Description . . . 24
7.1.2 Constructor & Destructor Documentation . . . 24
7.1.3 Member Function Documentation . . . 25
7.2 forcefield.FF Class Reference . . . 26
7.2.1 Detailed Description . . . 27
7.2.2 Constructor & Destructor Documentation . . . 27
7.2.3 Member Function Documentation . . . 28
7.2.4 Member Data Documentation . . . 32
7.3 fitsim.FittingSimulation Class Reference . . . 33
7.3.1 Detailed Description . . . 34
7.3.2 Constructor & Destructor Documentation . . . 35
7.3.3 Member Function Documentation . . . 35
7.3.4 Member Data Documentation . . . 36
7.4 forceenergymatch.ForceEnergyMatch Class Reference . . . 36
7.4.1 Detailed Description . . . 38
7.4.2 Constructor & Destructor Documentation . . . 38
7.4.3 Member Function Documentation . . . 38
7.4.4 Member Data Documentation . . . 39
7.5 forceenergymatch_gmx.ForceEnergyMatch_GMX Class Reference . . . 39
7.5.1 Detailed Description . . . 41
7.5.2 Constructor & Destructor Documentation . . . 42
7.5.3 Member Function Documentation . . . 42
7.6 CallGraph.node Class Reference . . . 44

7.6.1 Detailed Description . . . 44
7.6.2 Constructor & Destructor Documentation . . . 44
7.6.3 Member Data Documentation . . . 45
7.7 optimization.Optimizer Class Reference . . . 45
7.7.1 Detailed Description . . . 47
7.7.2 Constructor & Destructor Documentation . . . 47
7.7.3 Member Function Documentation . . . 47
7.7.4 Member Data Documentation . . . 52
7.8 project.Project Class Reference . . . 53
7.8.1 Detailed Description . . . 53
7.8.2 Constructor & Destructor Documentation . . . 54
7.8.3 Member Function Documentation . . . 54

1 Main Page

1.1 Introduction

Welcome to ForceBalance! :)

This is a *theoretical and computational chemistry* program primarily developed by Lee-Ping Wang during graduate school at MIT and postdoctoral work at Stanford. Contributors to the code include Jiahao Chen, Vijay Pande, Troy Van Voorhis, and Matt Welborn.

The function of this program is *automatic potential optimization*; given a force field (a.k.a. empirical potential) and a set of reference data, the program tunes the empirical parameters in the potential such that it reproduces the reference data as closely as possible.

The philosophy of this program is to present force field optimization in a unified and easily extensible framework. Since there are many different ways in theoretical chemistry to compute the potential energy of a collection of atoms, and similarly many types of reference data to fit these potentials to, we do our best to provide an infrastructure which allows a user or a contributor to fit any type of potential to any type of reference data.

1.2 to-do list

Todo Write 'Installation' section

Create an installer

Write 'How to read the documentation' section

Write 'How to create documentation' section

1.3 Glossary

It is useful to define several terms for the sake of our discussion.

- **Empirical Potential** : A formula or method that computes the potential energy of a collection of atoms that contains adjustable empirical parameters.

- **Force field** : Same as empirical potential.
- **Empirical parameter** : An adjustable number that enters into the empirical potential and has an effect on

1.4 Usage

Upon extracting the distribution you will notice three directories: 'bin', 'doc', and 'lib'.

The 'bin' directory contains all executable scripts and programs, the 'lib' directory contains all Python modules and libraries, and the 'doc' directory contains documentation.

To run this program, you may execute the scripts located in the 'bin' directory.

Figure 1: width=2cm

2 Todo List

Member CallGraph.node.__init__

document

document

document

document

Member CallGraph.node.dtype

document

Member CallGraph.node.name

document

Member CallGraph.node.parent

document

Member CallGraph.node::oid

document

Member counterpoisematch.CounterpoiseMatch.loadxyz

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

Member forceenergymatch.ForceEnergyMatch.__init__

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

Member forceenergymatch.ForceEnergyMatch.readrefdata

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)

As of now (Dec 2011) the WHAM Boltzmann weights are generated by external scripts (wanalyze.py and make-wham-data.sh) and passed in; tighter integration would be nice.

Member forceenergymatch_gmx.ForceEnergyMatch_GMX.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 forceenergymatch_gmx.ForceEnergyMatch_GMX.prepare_temp_directory

Currently we don't have a switch to turn the covariance off. Should be simple to add in.

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 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 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.

Namespace gmxio

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

Member gmxio.pdict

This needs to become more flexible because the parameter isn't always in the same field. Still need to figure out how to do this.

How about making the PDIHS less ugly?

page Main Page

Write 'Installation' section

Create an installer

Write 'How to read the documentation' section

Write 'How to create documentation' section

Member nifty.floatornan

I could use suggestions for making this better.

Namespace optimization

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

Member optimization.Optimizer.Scan_Values

Maybe a multidimensional grid can be done.

Member parser.parse_inputs

The section variable type hasn't been implemented yet.

3 Namespace Index

3.1 Namespace List

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

counterpoisematch
Match an empirical potential to the counterpoise correction for basis set superposition error (BSSE) 5

custom_io
Custom force field parser 6

forceenergymatch_gmx
Force and energy matching with interface to modified GROMACS 7

forcefield
Force field module 8

gmxio
GROMACS input/output 10

nifty
Nifty functions for ForceBalance, intended to be imported by any module 13

optimization
Optimization algorithms 18

parser
Input file parser for ForceBalance projects 19

project
ForceBalance force field optimization project 22

simtab
Contains the dictionary of fitting simulation classes 22

4 Class Index

4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

forcefield.FF 26

fitsim.FittingSimulation 33

- **counterpoisematch.CounterpoiseMatch** 23
- **forceenergymatch.ForceEnergyMatch** 36
  - **forceenergymatch_gmx.ForceEnergyMatch_GMX** 39

CallGraph.node 44

optimization.Optimizer 45

project.Project 53

5 Class Index

5.1 Class List

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

counterpoisematch.CounterpoiseMatch
FittingSimulation subclass for matching the counterpoise correction 23

forcefield.FF
Force field class 26

fitsim.FittingSimulation
Base class for all fitting simulations 33

forceenergymatch.ForceEnergyMatch
Subclass of FittingSimulation for force and energy matching 36

forceenergymatch_gmx.ForceEnergyMatch_GMX
ForceBalance class for force and energy matching with the modified GROMACS 39

CallGraph.node
Data structure for holding information about python objects 44

optimization.Optimizer
Optimizer class 45

project.Project
Container for a ForceBalance force field optimization project 53

6 Namespace Documentation

6.1 counterpoisematch Namespace Reference

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

Classes

- class CounterpoiseMatch

  FittingSimulation subclass for matching the counterpoise correction.

6.1.1 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

6.2 custom_io Namespace Reference

Custom force field parser.

Functions

- def makeaftype

  Given a line, section name and list of atom names, return the interaction type and the atoms involved.

Variables

- list cptypes = [None, 'CPGAUSS', 'CPEXPG', 'CPGEXP']

  Types of counterpoise correction.
- list ndtypes = [None]

  Types of NDDO correction.
- dictionary fdict

  Section -> Interaction type dictionary.
- dictionary pdict

  Interaction type -> Parameter Dictionary.

6.2.1 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

6.2.2 Function Documentation

6.2.2.1 def custom_io.makeaftype (*line, section*)

Given a line, section name and list of atom names, return the interaction type and the atoms involved.

Parameters

in	*line*	The line of data
in	*section*	The section that we're in

Returns

atom The atoms involved in the interaction
ftype The interaction type

Definition at line 40 of file custom_io.py.

6.2.3 Variable Documentation

6.2.3.1 dictionary custom_io::fdict

Initial value:

```
1 {
2     'counterpoise'  : cptypes    }
```

Section -> Interaction type dictionary.

Definition at line 22 of file custom_io.py.

6.2.3.2 dictionary custom_io::pdict

Initial value:

```
1 {'CPGAUSS':{3:'A',4:'B',5:'C'},
2          'CPEXPG' :{3:'A1',4:'B',5:'X0',6:'A2'},
3          'CPGEXP' :{3:'A',4:'B',5:'G',6:'X'}
4          }
```

Interaction type -> Parameter Dictionary.

Definition at line 26 of file custom_io.py.

6.3 forceenergymatch_gmx Namespace Reference

Force and energy matching with interface to modified GROMACS.

Classes

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

Functions

- def set_gmx_paths
  Set the gmxrunpath, gmxtoolpath and gmxsuffix attributes of a class.

6.3.1 Detailed Description

Force and energy matching with interface to modified GROMACS. 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_GMX class) allows us to call the modified GROMACS to compute the objective function for us.

Author

Lee-Ping Wang

Date

12/2011

6.3.2 Function Documentation

6.3.2.1 def forceenergymatch_gmx.set_gmx_paths (*me, options*)

Set the gmxrunpath, gmxtoolpath and gmxsuffix attributes of a class.

Parameters

in	*me*	The class whose attributes we want to set.
in	*options*	Simulation options dictionary

Definition at line 33 of file forceenergymatch_gmx.py.

6.4 forcefield Namespace Reference

Force field module.

Classes

- class FF

  Force field class.

Functions

- def rs_override

  This function takes in a dictionary (tvgeomean) and a string (termtype).

6.4.1 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

6.4.2 Function Documentation

6.4.2.1 def forcefield.rs_override (*tvgeomean, termtype, Temperature =* 298.15)

This function takes in a dictionary (tvgeomean) and a string (termtype).

If termtype matches any of the strings below, tvgeomean[termtype] is assigned to one of the numbers below.

This is LPW's attempt to simplify the rescaling factors.

Parameters

out	*tvgeomean*	The computed rescaling factor.
in	*termtype*	The interaction type (corresponding to a physical unit)
in	*Temperature*	The temperature for computing the kT energy scale

Definition at line 591 of file forcefield.py.

6.5 gmxio Namespace Reference

GROMACS input/output.

Functions

- def parse_atomtype_line

  Parses the 'atomtype' line.
- def makeaftype

  Given a line, section name and list of atom names, return the interaction type and the atoms involved.
- def gmxprint

  Prints a vector to a file to feed it to the modified GROMACS.

Variables

- list nftypes = [None, 'VDW', 'VDW_BHAM']

  VdW interaction function types.
- list pftypes = [None, 'VPAIR', 'VPAIR_BHAM']

  Pairwise interaction function types.
- list bftypes = [None, 'BONDS', 'G96BONDS', 'MORSE']

  Bonded interaction function types.
- list aftypes

  Angle interaction function types.
- list dftypes = [None, 'PDIHS', 'IDIHS', 'RBDIHS']

  Dihedral interaction function types.
- dictionary fdict

  Section -> Interaction type dictionary.
- dictionary pdict

  Interaction type -> Parameter Dictionary.

6.5.1 Detailed Description

GROMACS input/output.

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

Author

Lee-Ping Wang

Date

12/2011

6.5.2 Function Documentation

6.5.2.1 def gmxio.gmxprint (*fnm, vec, type*)

Prints a vector to a file to feed it to the modified GROMACS.

Ported over from the old version so it is a bit archaic for my current taste.

Parameters

in	*fnm*	The file name that we're printing the data to
in	*vec*	1-D array of data
in	*type*	Either 'int' or 'double', indicating the type of data.

Definition at line 252 of file gmxio.py.

6.5.2.2 def gmxio.makeaftype (*line, section*)

Given a line, section name and list of atom names, return the interaction type and the atoms involved.

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 return [H,O,H],'UREY_BRADLEY'

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

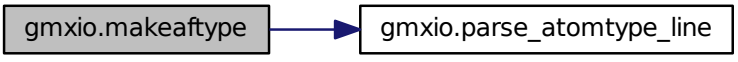
If we are in a TopolSection, it returns a list of atom names.

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 163 of file gmxio.py.

Here is the call graph for this function:



6.5.2.3 def gmxio.parse_atomtype_line (*line*)

Parses the 'atomtype' line.

Parses lines like this:

```
opls_135 CT 6 12.0107 0.0000 A 3.5000e-01 2.7614e-01
C 12.0107 0.0000 A 3.7500e-01 4.3932e-01
Na 11 22.9897 0.0000 A 6.068128070229e+03 2.662662556402e+01 0.0000e+00 ; PARM
5 6
```

Look at all the variety!

Parameters

in	*line*	Input line.

Returns

answer Dictionary containing:
atom type
bonded atom type (if any)
atomic number (if any)
atomic mass
charge
particle type
force field parameters
number of optional fields

Definition at line 106 of file gmxio.py.

6.5.3 Variable Documentation

6.5.3.1 list gmxio::aftypes

Initial value:

```
1 [None, 'ANGLES', 'G96ANGLES', 'CROSS_BOND_BOND',
2            'CROSS_BOND_ANGLE', 'UREY_BRADLEY', 'QANGLES']
```

Angle interaction function types.

Definition at line 21 of file gmxio.py.

6.5.3.2 dictionary gmxio::fdict

Initial value:

```
1 {
2     'atomtypes'     : nftypes,
3     'nonbond_params': pftypes,
4     'bonds'         : bftypes,
5     'bondtypes'     : bftypes,
6     'angles'        : aftypes,
7     'angletypes'    : aftypes,
8     'dihedrals'     : dftypes,
9     'dihedraltypes' : dftypes,
10     'virtual_sites2': ['NONE','VSITE2'],
```

```
11     'virtual_sites3': ['NONE','VSITE3','VSITE3FD','VSITE3FAD','VSITE3OUT'],
12     'virtual_sites4': ['NONE','VSITE4FD']
13     }
```

Section -> Interaction type dictionary.

Based on the section you're in and the integer given on the current line, this looks up the 'interaction type' - for example, within bonded interactions there are four interaction types: harmonic, G96, Morse, and quartic interactions.

Definition at line 32 of file gmxio.py.

6.5.3.3 dictionary gmxio::pdict

Initial value:

```
1 {'BONDS':{3:'B', 4:'K'},
2          'G96BONDS':{},
3          'MORSE':{3:'B', 4:'C', 5:'E'},
4          'ANGLES':{4:'B', 5:'K'},
5          'G96ANGLES':{},
6          'CROSS_BOND_BOND':{4:'1', 5:'2', 6:'K'},
7          'CROSS_BOND_ANGLE':{4:'1', 5:'2', 6:'3', 7:'K'},
8          'QANGLES':{4:'B', 5:'K0', 6:'K1', 7:'K2', 8:'K3', 9:'K4'},
9          'UREY_BRADLEY':{4:'T', 5:'K1', 6:'B', 7:'K2'},
10          'PDIHS1':{5:'B', 6:'K'},
11          'PDIHS2':{5:'B', 6:'K'},
12          'PDIHS3':{5:'B', 6:'K'},
13          'PDIHS4':{5:'B', 6:'K'},
14          'PDIHS5':{5:'B', 6:'K'},
15          'PDIHS6':{5:'B', 6:'K'},
16          'IDIHS':{5:'B', 6:'K'},
17          'VDW':{4:'S', 5:'T'},
18          'VPAIR':{3:'S', 4:'T'},
19          'COUL':{6:''},
20          'RBDIHS':{6:'K1', 7:'K2', 8:'K3', 9:'K4', 10:'K5'},
21          'VDW_BHAM':{4:'A', 5:'B', 6:'C'},
22          'VPAIR_BHAM':{3:'A', 4:'B', 5:'C'},
23          'QTPIE':{1:'C', 2:'H', 3:'A'},
24          'VSITE2':{4:'A'},
25          'VSITE3':{5:'A',6:'B'},
26          'VSITE3FD':{5:'A',6:'D'},
27          'VSITE3FAD':{5:'T',6:'D'},
28          'VSITE3OUT':{5:'A',6:'B',7:'C'},
29          'VSITE4FD':{6:'A',7:'B',8:'D'},
30          }
```

Interaction type -> Parameter Dictionary.

A list of supported GROMACS interaction types in force matching. The keys in this dictionary (e.g. 'BONDS','ANGLES') are values in the interaction type dictionary. As the program loops through the force field file, it first looks up the interaction types in 'fdict' and then goes here to do the parameter lookup by field.

Todo This needs to become more flexible because the parameter isn't always in the same field. Still need to figure out how to do this.

How about making the PDIHS less ugly?

Definition at line 55 of file gmxio.py.

6.6 nifty Namespace Reference

Nifty functions for ForceBalance, intended to be imported by any module.

Functions

- def isint

  ONLY matches integers! If you have a decimal point? None shall pass!
- def isfloat

  Matches ANY number; it can be a decimal, scientific notation, what have you CAUTION - this will also match an integer.
- def isdecimal

  Matches things with a decimal only; see isint and isfloat.
- def orthogonalize

  Given two vectors vec1 and vec2, project out the component of vec1 that is along the vec2-direction.
- def pmat2d

  Printout of a 2-D matrix.
- def printcool

  Cool-looking printout for slick formatting of output.
- def printcool_dictionary

  See documentation for printcool; this is a nice way to print out keys/values in a dictionary.
- def col

  Given any list, array, or matrix, return a 1-column matrix.
- def row

  Given any list, array, or matrix, return a 1-row matrix.
- def flat

  Given any list, array, or matrix, return a single-index array.
- def floatornan

  Returns a big number if we encounter NaN.

Variables

- float kb = 0.0083144100163

  Boltzmann constant.
- float eqcgmx = 2625.5002

  Q-Chem to GMX unit conversion for energy.
- float fqcgmx = 49621.9

  Q-Chem to GMX unit conversion for force.

6.6.1 Detailed Description

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

Author

Lee-Ping Wang

Date

12/2011

6.6.2 Function Documentation

6.6.2.1 def nifty.col (*vec*)

Given any list, array, or matrix, return a 1-column matrix.

Input: vec = The input vector that is to be made into a column

Output: A column matrix

Definition at line 138 of file nifty.py.

6.6.2.2 def nifty.flat (*vec*)

Given any list, array, or matrix, return a single-index array.

Parameters

in	*vec*	The data to be flattened

Returns

answer The flattened data

Definition at line 157 of file nifty.py.

6.6.2.3 def nifty.floatornan (*word*)

Returns a big number if we encounter NaN.

Parameters

in	*word*	The string to be converted

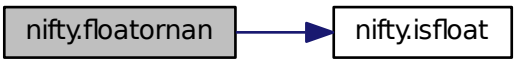
Returns

answer The string converted to a float; if not a float, return 1e10

Todo I could use suggestions for making this better.

Definition at line 167 of file nifty.py.

Here is the call graph for this function:



6.6.2.4 def nifty.isdecimal (*word*)

Matches things with a decimal only; see isint and isfloat.

Parameters

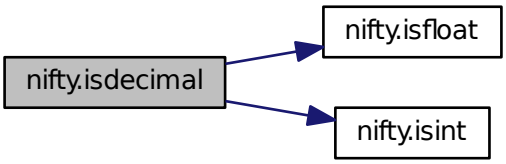
in	*word*	String (for instance, '123', '153.0', '2.', '-354')

Returns

answer Boolean which specifies whether the string is a number with a decimal point

Definition at line 49 of file nifty.py.

Here is the call graph for this function:



6.6.2.5 def nifty.isfloat (*word*)

Matches ANY number; it can be a decimal, scientific notation, what have you CAUTION - this will also match an integer.

Parameters

in	*word*	String (for instance, '123', '153.0', '2.', '-354')

Returns

answer Boolean which specifies whether the string is any number

Definition at line 39 of file nifty.py.

6.6.2.6 def nifty.isint (*word*)

ONLY matches integers! If you have a decimal point? None shall pass!

Parameters

in	*word*	String (for instance, '123', '153.0', '2.', '-354')

Returns

answer Boolean which specifies whether the string is an integer (only +/- sign followed by digits)

Definition at line 28 of file nifty.py.

6.6.2.7 def nifty.orthogonalize (*vec1, vec2*)

Given two vectors vec1 and vec2, project out the component of vec1 that is along the vec2-direction.

Parameters

in	*vec1*	The projectee (i.e. output is some modified version of vec1)
in	*vec2*	The projector (component subtracted out from vec1 is parallel to this)

Returns

answer A copy of vec1 but with the vec2-component projected out.

Definition at line 60 of file nifty.py.

6.6.2.8 def nifty.pmat2d (*mat2d*)

Printout of a 2-D matrix.

Parameters

in	*mat2d*	a 2-D matrix

Definition at line 69 of file nifty.py.

6.6.2.9 def nifty.printcool (*text, sym* = "#", *bold* = False, *color* = 2, *bottom* = '-', *minwidth* = 50)

Cool-looking printout for slick formatting of output.

Parameters

in	*text*	The string that the printout is based upon. This function will print out the string, ANSI-colored and enclosed in the symbol for example:
		`##################`
		`### I am cool ###`
		`##################`

in	*sym*	The surrounding symbol
in	*bold*	Whether to use bold print
in	*color*	The ANSI color: 1 red 2 green 3 yellow 4 blue 5 magenta 6 cyan 7 white
in	*bottom*	The symbol for the bottom bar
in	*minwidth*	The minimum width for the box, if the text is very short then we insert the appropriate number of padding spaces

Returns

bar The bottom bar is returned for the user to print later, e.g. to mark off a 'section'

Definition at line 103 of file nifty.py.

6.6.2.10 def nifty.printcool_dictionary (*dict, title* = "General options")

See documentation for printcool; this is a nice way to print out keys/values in a dictionary.

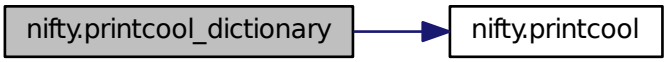
The keys in the dictionary are sorted before printing out.

Parameters

in	*dict*	The dictionary to be printed
in	*title*	The title of the printout

Definition at line 123 of file nifty.py.

Here is the call graph for this function:



6.6.2.11 def nifty.row (*vec*)

Given any list, array, or matrix, return a 1-row matrix.

Parameters

in	*vec*	The input vector that is to be made into a row

Returns

answer A row matrix

Definition at line 148 of file nifty.py.

6.7 optimization Namespace Reference

Optimization algorithms.

Classes

- class Optimizer
  Optimizer class.

6.7.1 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

6.8 parser Namespace Reference

Input file parser for ForceBalance projects.

Functions

- def parse_inputs

  Parse through the input file and read all user-supplied options.

Variables

- dictionary gen_opts_types

  Default general options.
- dictionary gen_opts_defaults = {}

  Default general options - basically a collapsed veresion of gen_opts_types.
- dictionary sim_opts_types

  Default fitting simulation options.
- dictionary sim_opts_defaults = {}

  Default simulation options - basically a collapsed version of sim_opts_types.
- list mainsections = ["SIMULATION","OPTIONS","END","NONE"]

  Listing of sections in the input file.
- dictionary subsections = {"OPTIONS":["READ_MVALS", "READ_PVALS"]}

  Listing of subsections.

6.8.1 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 haven't been implemented yet. They 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

6.8.2 Function Documentation

6.8.2.1 def parser.parse_inputs (*input_file*)

Parse through the input file and read all user-supplied options.

This is usually the first thing that happens when an executable script is called. Our parser first loads the default options, and then updates these options as it encounters keywords.

Each keyword corresponds to a variable type; each variable type (e.g. string, integer, float, boolean) is treated differently. For more elaborate inputs, there is a 'section' variable type.

There is only one set of general options, but multiple sets of fitting simulation options. Each fitting simulation has its own section delimited by the *$simulation* keyword, and we build a list of simulation options.

Parameters

in	*input_file*	The name of the input file.

Returns

options General options.
sim_opts List of fitting simulation options.

Todo The section variable type hasn't been implemented yet.

Definition at line 155 of file parser.py.

6.8.3 Variable Documentation

6.8.3.1 dictionary parser::gen_opts_defaults = {}

Default general options - basically a collapsed veresion of gen_opts_types.

Definition at line 90 of file parser.py.

6.8.3.2 dictionary parser::gen_opts_types

Default general options.

Definition at line 56 of file parser.py.

6.8.3.3 list parser::mainsections = ["SIMULATION","OPTIONS","END","NONE"]

Listing of sections in the input file.

Definition at line 130 of file parser.py.

6.8.3.4 dictionary parser::sim_opts_defaults = {}

Default simulation options - basically a collapsed version of sim_opts_types.

Definition at line 125 of file parser.py.

6.8.3.5 dictionary parser::sim_opts_types

Initial value:

```
1 {
2     'strings' : {"name"                     : None,    # The name of the
       simulation, which corresponds to the directory simulations/dir_name
3                  },
4     'allcaps' : {"simtype"                  : None     # The type of fitting
       simulation, for instance ForceEnergyMatching_GMX
5                  },
6     'lists'   : {"fd_ptypes"                : []       # The parameter types
       that need to be differentiated using finite difference
7                  },
8     'ints'    : {"shots"                    : -1,      # Number of snapshots
       (force+energy matching); defaults to all of the snapshots
9                  "fitatoms"                : 0        # Number of fitting
       atoms (force+energy matching); defaults to all of them
10                  },
11    'bools'   : {"whamboltz"                 : 0,       # Whether to use WHAM
       Boltzmann Weights (force+energy match), defaults to False
12                 "sampcorr"                : 0,       # Whether to use the
```

```
      (archaic) sampling correction (force+energy match), defaults to False
13                  "covariance"                : 1,       # Whether to use the
      quantum covariance matrix (force+energy match), defaults to True
14                  "batch_fd"                  : 0,       # Whether to batch and
      queue up finite difference jobs, defaults to False
15                  "fdgrad"                    : 1,       # Finite difference
      gradients
16                  "fdhess"                    : 1,       # Finite difference
      Hessian diagonals (costs np times a gradient calculation)
17                  "fdhessdiag"                : 1,       # Finite difference
      Hessian diagonals (cheap; costs 2np times a objective calculation)
18                  "use_pvals"                 : 0        # Bypass the
      transformation matrix and use the physical parameters directly
19                  },
20      'floats'  : {"weight"                   : 1.0,     # Weight of the current
      simulation (with respect to other simulations)
21                  "efweight"                  : 0.5,     # 1.0 for all energy
      and 0.0 for all force (force+energy match), defaults to 0.5
22                  "qmboltz"                   : 0.0,     # Fraction of Quantum
      Boltzmann Weights (force+energy match), 1.0 for full reweighting, 0.0 < 1.0 for
      hybrid
23                  "qmboltztemp"               : 298.15   # Temperature for
      Quantum Boltzmann Weights (force+energy match), defaults to room temperature
24                  },
25      'sections': {}
26      }
```

Default fitting simulation options.

Definition at line 97 of file parser.py.

6.8.3.6 dictionary parser::subsections = {"OPTIONS":["READ_MVALS", "READ_PVALS"]}

Listing of subsections.

Definition at line 132 of file parser.py.

6.9 project Namespace Reference

ForceBalance force field optimization project.

Classes

- class Project

  Container for a ForceBalance force field optimization project.

6.9.1 Detailed Description

ForceBalance force field optimization project.

6.10 simtab Namespace Reference

Contains the dictionary of fitting simulation classes.

Variables

- dictionary SimTab

The table of fitting simulations.

6.10.1 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.

6.10.2 Variable Documentation

6.10.2.1 dictionary simtab::SimTab

Initial value:

```
1 {
2     'FORCEENERGYMATCH_GMX':ForceEnergyMatch_GMX,
3     'COUNTERPOISEMATCH':CounterpoiseMatch
4     }
```

The table of fitting simulations.

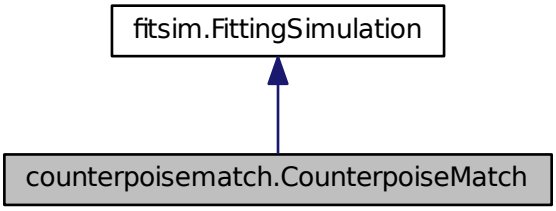
Definition at line 12 of file simtab.py.

7 Class Documentation

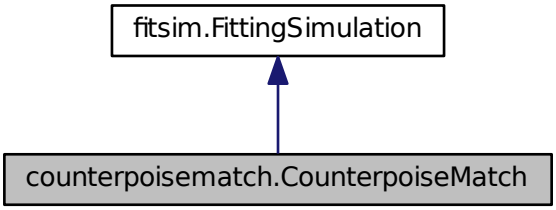
7.1 counterpoisematch.CounterpoiseMatch Class Reference

FittingSimulation subclass for matching the counterpoise correction.

Inheritance diagram for counterpoisematch.CounterpoiseMatch:



Collaboration diagram for 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.

7.1.1 Detailed Description

FittingSimulation subclass for matching the counterpoise correction.

Definition at line 31 of file counterpoisematch.py.

7.1.2 Constructor & Destructor Documentation

7.1.2.1 def counterpoisematch.CounterpoiseMatch.__init__ (*self, options, sim_opts, forcefield*)

To instantiate CounterpoiseMatch, we read the coordinates and counterpoise data.

Reimplemented from fitsim.FittingSimulation.

Definition at line 35 of file counterpoisematch.py.

7.1.3 Member Function Documentation

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

Gets the objective function for fitting the counterpoise correction.

As opposed to ForceEnergyMatch_GMX, 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.

7.1.3.2 def 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.

7.1.3.3 def 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/lib/counterpoisematch.py

7.2 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 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 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.
- 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.
- pvals

  Current physical parameters.
- mvals0

  Initial value of mathematical parameters.
- mvals

  Current mathematical parameters.

7.2.1 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 117 of file forcefield.py.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 def 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 125 of file forcefield.py.

7.2.3 Member Function Documentation

7.2.3.1 def 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

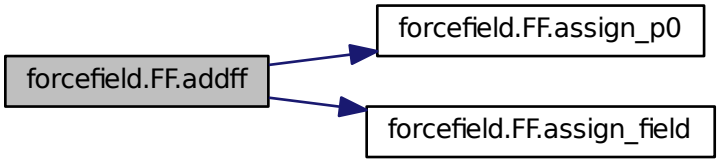
My program currently assumes that we are only using one MM program per job. If we use CHARMM and GROMACS to perform fitting simulations in the same job, we will get f-ed up. Maybe this needs to be fixed in the future, with program prefixes to parameters like C_ , G_ .. or simply unit conversions, you 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 268 of file forcefield.py.

Here is the call graph for this function:



7.2.3.2 def 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 572 of file forcefield.py.

7.2.3.3 def 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 554 of file forcefield.py.

7.2.3.4 def 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

Definition at line 362 of file forcefield.py.

7.2.3.5 def 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 534 of file forcefield.py.

7.2.3.6 def 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.

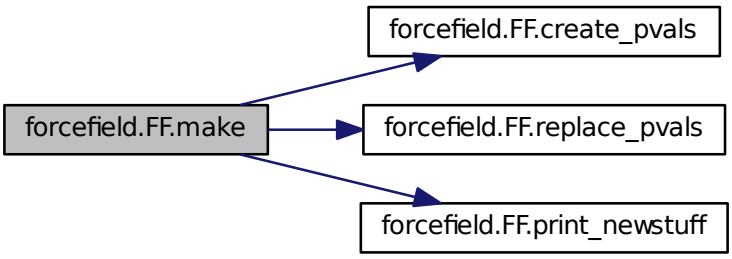
It does NOT store the mathematical parameters in the class state (since we can only hold one set of parameters).

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 344 of file forcefield.py.

Here is the call graph for this function:



7.2.3.7 def 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 468 of file forcefield.py.

7.2.3.8 def forcefield.FF.print_map (*self*)

Prints out the parameter indices, IDs and values in a visually appealing way.

Definition at line 543 of file forcefield.py.

7.2.3.9 def 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 392 of file forcefield.py.

7.2.3.10 def forcefield.FF.replace_pvals (*self*)

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 371 of file forcefield.py.

7.2.3.11 def 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 407 of file forcefield.py.

Here is the call graph for this function:

7.2.4 Member Data Documentation

7.2.4.1 forcefield.FF::newstuff

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

Definition at line 140 of file forcefield.py.

7.2.4.2 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 134 of file forcefield.py.

7.2.4.3 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 135 of file forcefield.py.

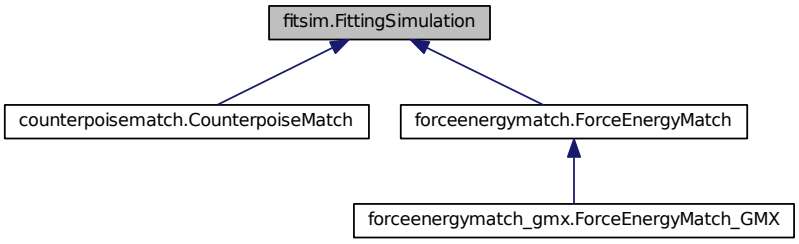
The documentation for this class was generated from the following file:

- ForceBalance/lib/forcefield.py

7.3 fitsim.FittingSimulation Class Reference

Base class for all fitting simulations.

Inheritance diagram for fitsim.FittingSimulation:



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.

Public Attributes

- root
  Root directory of the whole project.
- name
  Name of the fitting simulation.
- simtype
  Type of 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.

7.3.1 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 68 of file fitsim.py.

7.3.2 Constructor & Destructor Documentation

7.3.2.1 def 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.

Reimplemented in forceenergymatch_gmx.ForceEnergyMatch_GMX, forceenergymatch.ForceEnergyMatch, and counterpoisematch.CounterpoiseMatch.

Definition at line 85 of file fitsim.py.

7.3.3 Member Function Documentation

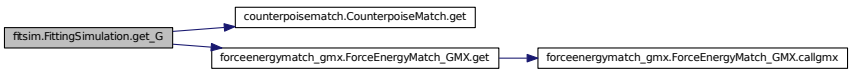
7.3.3.1 def fitsim.FittingSimulation.get_G (*self, mvals* = `None`)

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 148 of file fitsim.py.

Here is the call graph for this function:



7.3.3.2 def fitsim.FittingSimulation.get_H (*self, mvals* = `None`)

Computes the objective function contribution and its gradient / Hessian.

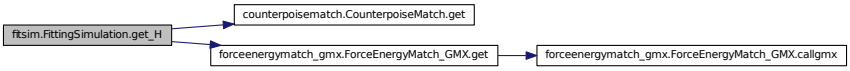
First the low-level 'get' method is called with the analytic gradient and Hessian both turned on. Then we loop through the fd1_pids and compute the corresponding elements of the gradient by finite difference, if the 'fdgrad' switch is turned

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 171 of file fitsim.py.

Here is the call graph for this function:



7.3.4 Member Data Documentation

7.3.4.1 fitsim.FittingSimulation::usepvals

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

For power users only! :)

Definition at line 96 of file fitsim.py.

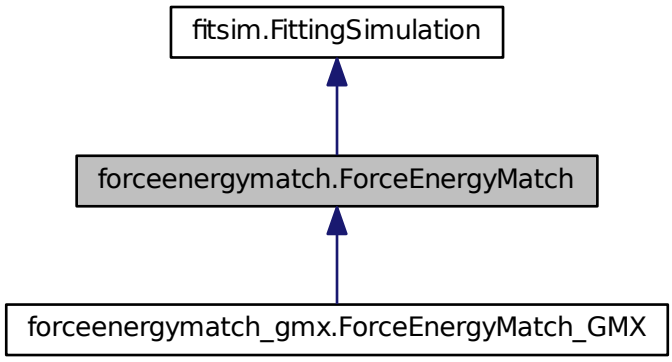
The documentation for this class was generated from the following file:

- ForceBalance/lib/fitsim.py

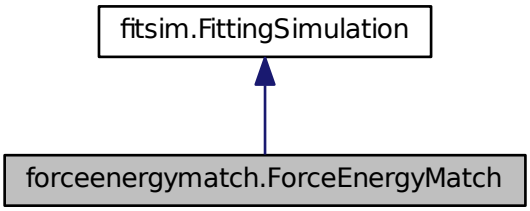
7.4 forceenergymatch.ForceEnergyMatch Class Reference

Subclass of FittingSimulation for force and energy matching.

Inheritance diagram for forceenergymatch.ForceEnergyMatch:



Collaboration diagram for forceenergymatch.ForceEnergyMatch:



Public Member Functions

- def __init__
  Instantiation of the subclass.
- def readrefdata
  Read the reference data (for force and energy matching) from a file such as qdata.txt.

Public Attributes

- ns
  Number of snapshots.
- whamboltz
  Whether to use WHAM Boltzmann weights.
- sampcorr
  Whether to use the Sampling Correction.
- 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.

7.4.1 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

Note that this subclass does not contain the 'get' method.

Definition at line 38 of file forceenergymatch.py.

7.4.2 Constructor & Destructor Documentation

7.4.2.1 def 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.

Reimplemented from fitsim.FittingSimulation.

Reimplemented in forceenergymatch_gmx.ForceEnergyMatch_GMX.

Definition at line 51 of file forceenergymatch.py.

7.4.3 Member Function Documentation

7.4.3.1 def forceenergymatch.ForceEnergyMatch.readrefdata (*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 As of now (Dec 2011) the WHAM Boltzmann weights are generated by external scripts (wanalyze.py and make-wham-data.sh) and passed in; tighter integration would be nice.

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, 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 154 of file forceenergymatch.py.

7.4.4 Member Data Documentation

7.4.4.1 forceenergymatch.ForceEnergyMatch::efweight

The proportion of energy vs.

force.

Definition at line 58 of file forceenergymatch.py.

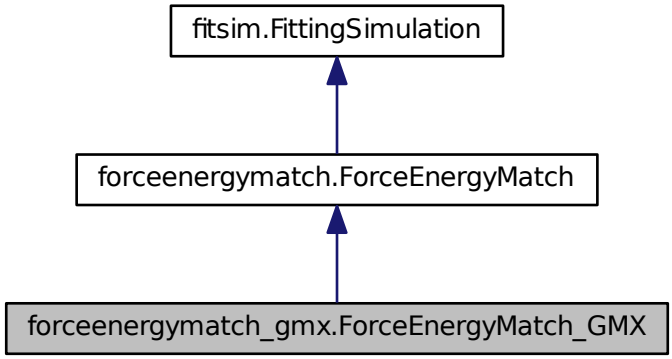
The documentation for this class was generated from the following file:

- ForceBalance/lib/forceenergymatch.py

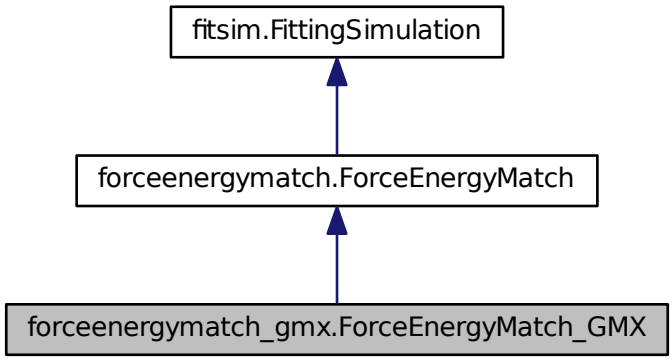
7.5 forceenergymatch_gmx.ForceEnergyMatch_GMX Class Reference

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

Inheritance diagram for forceenergymatch_gmx.ForceEnergyMatch_GMX:



Collaboration diagram for forceenergymatch_gmx.ForceEnergyMatch_GMX:



Public Member Functions

- def __init__

  Instantiation of ForceEnergyMatch_GMX.
- def load_gro_files

  Load the entire trajectory into memory.
- def backup_temp_directory

  Back up the temporary directory.
- 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 callgmx

  Call the modified GROMACS!

Public Attributes

- nparticles

  The number of particles (includes atoms, Drudes, and virtual sites)

- natoms

  The number of true atoms.

- grofnm

  Path to the all.gro file.

- allgro

  The all.gro file is loaded into memory.

- whamboltz

  Whether to use WHAM Boltzmann weights.

- sampcorr

  Whether to use the Sampling Correction.

- fitatoms

  Number of atoms that we are fitting.

7.5.1 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 77 of file forceenergymatch_gmx.py.

7.5.2 Constructor & Destructor Documentation

7.5.2.1 def forceenergymatch_gmx.ForceEnergyMatch_GMX.__init__ (*self, options, sim_opts, forcefield*)

Instantiation of ForceEnergyMatch_GMX.

Several important things happen here:

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

Reimplemented from forceenergymatch.ForceEnergyMatch.

Definition at line 88 of file forceenergymatch_gmx.py.

7.5.3 Member Function Documentation

7.5.3.1 def forceenergymatch_gmx.ForceEnergyMatch_GMX.backup_temp_directory (*self*)

Back up the temporary directory.

LPW Todo: This is a candidate for moving up to the superclass.

Definition at line 149 of file forceenergymatch_gmx.py.

7.5.3.2 def forceenergymatch_gmx.ForceEnergyMatch_GMX.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 callgmx() 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

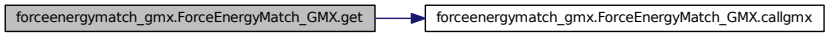
Answer Contribution to the objective function

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 290 of file forceenergymatch_gmx.py.

Here is the call graph for this function:



7.5.3.3 def forceenergymatch_gmx.ForceEnergyMatch_GMX.load_gro_files (*self, grofnm*)

Load the entire trajectory into memory.

The data is not interpreted, but the trajectory is split into multiple .gro files.

Actually this is not very useful, but I prefer to perform the coordinate file manipulations in-house rather than relying on a program like trjconv.

Parameters

in	*grofnm*	The .gro file to be loaded.

Definition at line 124 of file forceenergymatch_gmx.py.

7.5.3.4 def forceenergymatch_gmx.ForceEnergyMatch_GMX.prepare_temp_directory (*self, 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 Currently we don't have a switch to turn the covariance off. Should be simple to add in.

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 189 of file forceenergymatch_gmx.py.

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

- ForceBalance/lib/forceenergymatch_gmx.py

7.6 CallGraph.node Class Reference

Data structure for holding information about python objects.

Public Member Functions

- def __init__
  Constructor.

Public Attributes

- oid
- name
- parent
- dtype

7.6.1 Detailed Description

Data structure for holding information about python objects.

Definition at line 20 of file CallGraph.py.

7.6.2 Constructor & Destructor Documentation

7.6.2.1 def CallGraph.node.__init__ (*self*, *nodeid* = 0, *name* = '', *parent* = None, *dtype* = None)

Constructor.

Parameters

nodeid\xrefitem	todo 19.
name\xrefitem	todo 20.
parent\xrefitem	todo 21.
dtype\xrefitem	todo 22.

Definition at line 29 of file CallGraph.py.

7.6.3 Member Data Documentation

7.6.3.1 CallGraph.node::dtype

Todo document

Definition at line 33 of file CallGraph.py.

7.6.3.2 CallGraph.node::name

Todo document

Definition at line 31 of file CallGraph.py.

7.6.3.3 CallGraph.node::oid

Todo document

Definition at line 30 of file CallGraph.py.

7.6.3.4 CallGraph.node::parent

Todo document

Definition at line 32 of file CallGraph.py.

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

- ForceBalance/doc/callgraph/CallGraph.py

7.7 optimization.Optimizer Class Reference

Optimizer class.

Public Member Functions

- def __init__

  Instantiation of the 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 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.

Public Attributes

- 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.
- 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.
- Objective

  The objective function (needs to pass in when I instantiate)
- Sims

  The fitting simulations.
- FF

  The force field itself.

- OptTab

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

  The indices to be excluded from the Hessian update.
- mvals0

  The original parameter values.
- np

  Number of parameters.

7.7.1 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 31 of file optimization.py.

7.7.2 Constructor & Destructor Documentation

7.7.2.1 def optimization.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 44 of file optimization.py.

7.7.3 Member Function Documentation

7.7.3.1 def optimization.Optimizer.Anneal (*self*)

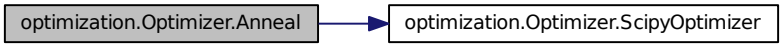
Use SciPy's built-in simulated annealing algorithm to optimize the parameters.

See also

Optimizer::ScipyOptimizer

Definition at line 315 of file optimization.py.

Here is the call graph for this function:



7.7.3.2 def optimization.Optimizer.BFGS (*self*)

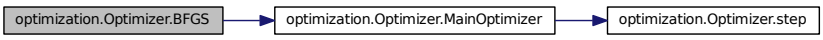
Optimize the force field parameters using the BFGS method; currently the recommended choice (.

See also

MainOptimizer)

Definition at line 265 of file optimization.py.

Here is the call graph for this function:



7.7.3.3 def optimization.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 411 of file optimization.py.

7.7.3.4 def optimization.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 443 of file optimization.py.

7.7.3.5 def optimization.Optimizer.Gradient (*self*)

A single-point gradient computation.

Definition at line 389 of file optimization.py.

7.7.3.6 def optimization.Optimizer.Hessian (*self*)

A single-point Hessian computation.

Definition at line 397 of file optimization.py.

7.7.3.7 def optimization.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.

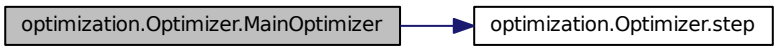
The optimization is terminated after either a function value or step size tolerance is reached.

Parameters

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

Definition at line 131 of file optimization.py.

Here is the call graph for this function:



7.7.3.8 def optimization.Optimizer.NewtonRaphson (*self*)

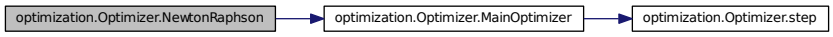
Optimize the force field parameters using the Newton-Raphson method (.

See also

MainOptimizer)

Definition at line 260 of file optimization.py.

Here is the call graph for this function:



7.7.3.9 def optimization.Optimizer.Powell (*self*)

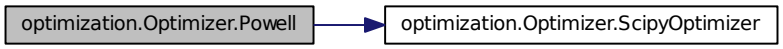
Use SciPy's built-in Powell direction-set algorithm to optimize the parameters.

See also

Optimizer::ScipyOptimizer

Definition at line 310 of file optimization.py.

Here is the call graph for this function:



7.7.3.10 def optimization.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 341 of file optimization.py.

7.7.3.11 def optimization.Optimizer.ScanMVals (*self*)

Scan through the mathematical parameter space.

See also

Optimizer::ScanValues

Definition at line 373 of file optimization.py.

Here is the call graph for this function:

7.7.3.12 def optimization.Optimizer.ScanPVals (*self*)

Scan through the physical parameter space.

See also

Optimizer::ScanValues

Definition at line 378 of file optimization.py.

Here is the call graph for this function:

7.7.3.13 def optimization.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 278 of file optimization.py.

7.7.3.14 def optimization.Optimizer.Simplex (*self*)

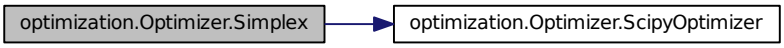
Use SciPy's built-in simplex algorithm to optimize the parameters.

See also

Optimizer::ScipyOptimizer

Definition at line 305 of file optimization.py.

Here is the call graph for this function:



7.7.3.15 def optimization.Optimizer.SinglePoint (*self*)

A single-point objective function computation.

Definition at line 383 of file optimization.py.

7.7.3.16 def optimization.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 237 of file optimization.py.

7.7.4 Member Data Documentation

7.7.4.1 optimization.Optimizer::OptTab

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

Definition at line 57 of file optimization.py.

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

- ForceBalance/lib/optimization.py

7.8 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?
- def Run

  Call the appropriate optimizer.

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.

7.8.1 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.

7.8.2 Constructor & Destructor Documentation

7.8.2.1 def 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.

7.8.3 Member Function Documentation

7.8.3.1 def project.Project.Objective (self, mvals, Order = 0, usepvals = 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 71 of file project.py.

7.8.3.2 def project.Project.Run (self)

Call the appropriate optimizer.

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

Definition at line 93 of file project.py.

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

- ForceBalance/lib/project.py