pieface Documentation

Release 1.0.0

James Cumby

Sep 27, 2016

pieface is a Python program to fit distortions in atomic coordination polyhedra.

pieface is designed to be easy to use, versatile and easily extendable. The *MBE* ellipsoid method used to fit polyhedra is very general, and can be applied to a wide range of coordination problems. Full details of the method, and a few interesting examples can be seen in the original research publication: James Cumby & J. Paul Attfield, *Ellipsoidal Analysis of Coordination Polyhedra*.

To get started with pieface, have a look at the *Introduction* and *Tutorials*. Further documentation can be found below.

CHAPTER

ONE

INTRODUCTION TO PIEFACE

1.1 Polyhedra Inscribing Ellipsoids For Analysis of Coordination Environments

Polyhedra **I**nscribing **E**llipsoids **F**or **A**nalysis of **C**oordination **E**nvironments (or pieface) is an open source Python project intended for the analysis of distortions of a chemical coordination polyhedron. The analysis is very general way, irrespective of polyhedron size or nature of the distortion, and could be applied to any problem where a coordination sphere with known coordinates exists, from extended inorganic solids to organic molecules. Full details of the method can be found in the original research article, Ellipsoidal Analysis of Coordination Polyhedra, including some interesting examples.

For many crystallographic polyhedra, distortion is difficult to rationalise simply in terms of deviations of bond lengths or bond angles from *ideal* values. By fitting the smallest volume *ellipsoid* around the polyhedron, distortions are defined in terms of the three principal axes of the ellipsoid and it's orientation in space. The distortion of this *ellipsoid* can then give a simple description of the distortions involved.

1.2 Citing pieface

If you use pieface, please **cite** it:

> James Cumby & J. Paul Attfield, *Ellipsoidal Analysis of Coordination Polyhedra*.

1.3 Getting Started

Once installed (see *Installation*) pieface can be accessed either through a command-line interface (`CIFellipsoid`) or a user-friendly graphical interface (`distellipsoidGUI`). Both should be available on the system command line/terminal, and also from the start menu (on Windows).

`distellipsoidGUI` Should be adequate for most users. This *GUI* provides a clickable interface to commonly used pieface functions, and allows users to import *CIF* files for analysis, and examine/save the resulting output.

`CIFellipsoid` Gives terminal-based access to a wider range of capabilities, details of which can be found by typing `CIFellipsoid --help`.

In both cases, the input required is one or more *CIF* files, and a list of atom types or labels to be used as polyhedron centres and ligands. Once calculated (which can take some time for a large number of files) the resulting ellipsoid parameters are saved as a text file (one per *CIF* file). Using `distellipsoidGUI` the resulting ellipsoids and parameters can also be visualised interactively.

More detailed examples of usage can be found in *Tutorials*.

1.4 License

pieface is distributed under the MIT license (see <../license.txt>). Any use of the software should be cited

1.5 Disclaimer

This software is provided as-is, on a best-effort basis. The authors accept no liabilities associated with the use of this software. It has been tested for accuracy of results for a number of cases, but only for uses that the authors can think of. We would be interested to hear of any suggestions for new uses, or potential additions to the software.

We will attempt to correct any bugs as they are found on a best-effort basis!

1.6 Authors

James Cumby - james.cumby “at” ed.ac.uk

CHAPTER

TWO

INSTALLATION

pieface is written in pure Python. While this makes it highly transferrable between operating systems, it does require a number of other Python packages to operate.

2.1 Requirements

- Python 2.7 (currently NOT Python 3)
- NumPy (at least version 1.9)
- matplotlib (1.4.3 or higher)
- PyCifRW (3.3 or higher)
- multiprocessing (2.6.2 or higher)
- pandas (0.17 or higher)

2.2 Installing

Detailed installation instructions specific to different operating systems can be found under *Windows*, *MAC OS X* and *Linux derivatives*.

pieface is registered on PyPI, therefore if you already have a working Python distribution, installation may be as simple as:

```
pip install pieface
```

or alternatively by manually downloading and installing:

- Download `pieface.tar.gz` to your computer
- Unpack to a local directory
- Install using:

```
python setup.py install
```

In reality, installation can sometimes be operating-system specific.

2.2.1 Windows

Due to problems with ensuring correct dependencies, the recommended method for obtaining pieface for Windows is to download the self-contained installer WinSetup_PIEFACE_1.0.0.exe and run it, following the on-screen prompts. This will also (optionally) add pieface shortcuts to the Start Menu and Windows Desktop, as well as making the two main scripts accessible from the Windows Command Line.

The installer comes packaged with a minimal Python runtime environment, therefore this installer will work without (and not interfere with an existing) Python installation.

2.2.2 MAC OS X

Unfortunately pieface is not currently available as a pre-built MAC distribution, as the author does not have access to that operating system!

Installing using the simple `pip` or `python setup.py install` routes may be possible using the default Python environment...

2.2.3 Linux derivatives

Unix-like operating systems generally come with python included. In this case,:

```
pip install pieface
```

should work as expected.

2.3 Installation from Sources

2.3.1 Stable Build

pieface can also be installed from the source distribution. The current stable build is pieface_1.0.0.tar.gz. Once downloaded, this file should be unpacked into the desired directory (`tar -xzf pieface_1.0.0.tar.gz`) before following the *setup instructions*.

2.3.2 Development Version

The latest development version of pieface can be obtained from the pieface repository using GIT:

```
GIT clone git://github.com/JCumby/pieface .
```

To update the repository at a later date, use:

```
GIT pull
```

In both cases, you should then change into the resulting directory, and follow the *setup instructions*.

2.3.3 Manual Install

Once the source code has been downloaded, it is then necessary to install it using Python from within the main pieface directory:

```
Python setup.py install
```

This may require all dependencies to already be installed.

2.4 Testing

The package contains some basic unit tests, which can be run from within the main pieface directory with the command:

```
python setup.py test
```

All tests should pass without exceptions - if not, please send me a bug report.

2.5 Run It!

Once correctly installed, the easiest way to access pieface is using either `distellipsoidGUI` or `CIFellipsoid` (see *Tutorials*).

CHAPTER

THREE

TUTORIALS

Still under construction!

CHAPTER

FOUR

USER-INTERFACE DOCUMENTATION

4.1 CIFellipsoid

This is the main command line script for using pieface to compute *MBE* from one or more *CIF* files. Details of the input parameters are given below, or by typing `CIFellipsoid --help`.

cifs
Name(s) of CIF files to import as a space-delimited list. Can also accept valid web address(es).

-m, --metal

Site label(s) (as found in the CIF file) of polyhedron centres to analyse (e.g. Fe1).

***Most* regular expressions can be used to make searching easier:**

Al* matches any site label starting `Al` (`Al1`, `Al2` ... `Al9999` etc.)

Al? matches any label beginning *Al*, but only 3 characters in length (e.g. `Al1` - `Al9`)

Al[1-9] matches any site `Al1` - `Al9`

In addition to most normal regular expressions, preceding any label by # will omit it from the search:

#Al1 will omit `Al1` from the list of acceptable centres.

By combining these terms, it should be possible to specify most desired combinations.

-o, --output
Name(s) of output files to save results to. Default is <CIF Name>.txt.

-r, --radius
Maximum distance to treat a ligand as part of a polyhedron (default 3 Angstrom).

-l, --ligandtypes
Types of atom (as specified by CIF atom_type) to be considered as valid ligands (can use regular expressions). Default is all atom types.

-n, --ligandnames
Atom labels to use as ligands (same syntax as –metal). By default, any ligand allowed by –ligandtypes is allowed. The combination of –ligandtypes and –ligandnames is taken as an `AND`-like operation, such that sites are only excluded if done so explicitly.

-t, --tolerance
Tolerance to use for fitting ellipsoid to points (default 1e-6).

--maxcycles
Maximum number of iterations to perform for fitting (default infinite).

-N
Don't save results to text files

-W, --overwriteall
If existing results files already exist, force pieface to overwrite them all.

-P, --PrintLabels
Print all valid site labels for each CIF file supplied.

-U, --Unthreaded
Turn off parallel processing of CIF files.

--procs
Number of processors to use for parallel processing (default all).

--noplot
Don't produce interactive ellipsoid images after calculation.

--writelog
Write a debugging log to `debug.log`.

CHAPTER

FIVE

API REFERENCE

5.1 Ellipsoid Module

Contains the core functionality of pieface, responsible for fitting a *MBE* to a set of points in Cartesian space. Module containing functions for calculating a minimum bounding ellipsoid from a set of points.

Contains a class to hold Ellipsoid object/properties, and functions to compute the ellipsoid from a set of points

Basic usage:

- Set up an Ellipsoid object
- Assign a set of (cartesian) points to be used for fitting
- Use method findellipsoid() to compute minimum bounding ellipsoid
- After that, all other properties should be available.

class distellipsoid.ellipsoid.**Ellipsoid**(*points=None*, *tolerance=1e-06*)

An object for computing various hyperellipse properties.

centreaxes()

Return displacement along ellipsoid axes.

centredisp()

Return total displacement of centre.

ellipsvol()

Return volume of ellipsoid.

findellipsoid(*suppliedpts=None*, ***kwargs*)

Determine the number of dimensions required for hyperellipse, and call then compute it with getminvol.

getminvol(*points=None*, *maxcycles=None*)

Find the minimum bounding ellipsoid for a set of points using the Khachiyan algorithm.

This can be quite time-consuming if a small tolerance is required, and ellipsoid axes lie a long way from axis directions.

meanrad()

Return the mean radius.

numpoints()

Return the number of points.

plot(*figure=None*, *axes=None*, ***kwargs*)

Plot graph of ellipsoid

plotsummary(***kwargs*)

Plot graph of ellipsoid with text of basic parameters

points

Points to define a hyperellipse.

raderr()
Return standard deviation in radii

radvar()
Return variance of the radii.

shapeparam()
Return ellipsoid shape measure r3/r2 - r2/r1.

shapeparam_old()
Return ellipsoid shape measure r1/r2 - r2/r3.

sphererad()
Return radius of sphere of equivalent volume as ellipsoid.

strainenergy()
Return ellipsoid strain energy approximation.

uniquerad(*tolerance=None*)
Determine the number of unique radii within tolerance (defaults to self.tolerance)

5.2 Polyhedron

Represents the set of objects that define a coordination polyhedron, including transforming from a unit cell description to one of orthogonal (Cartesian) positions. Module for containing polyhedron data and functions

class distellipsoid.polyhedron.**Polyhedron**(*centre*, *ligands*, *atomdict=None*, *ligtypes=None*)
Class to hold polyhedron object

allbondlens(*mtensor*)
Return bond lengths to all ligands

alldelabc()
Return all coordinates relative to centre.

alldelxyz(*orthom*)
Return all cartesian coordinates relative to centre.

allxyz(*orthom*)
Return all atoms in cartesian coordinates.

averagebondlen(*mtensor*)
Return average centre-ligand bond length

bondlensig(*mtensor*)
Return standard deviation of bond lengths

bondlenvar(*mtensor*)
Return variance of bond lengths

cenxyz(*orthom*)
Return centre atom cartesian coordinates.

ligdelabc()
Return ligand coordinates relative to centre.

ligdelxyz(*orthom*)
Return ligand cartesian coordinates relative to centre.

ligxyz(*orthom*)
Return ligand cartesian coordinates.

makeellipsoid(*orthom*, ***kwargs*)
Set up ellipsoid object and fit minimum bounding ellipsoid

pointcolours()
Return a list of colours for points based on ligand type.

5.3 Crystal

Class to hold a number of polyhedron objects, as well as unit cell parameters and orthogonalisation matrix, etc.

class distellipsoid.readcoords.**Crystal**(*cell=None*, *atoms=None*, *atomtypes=None*)
Class to hold crystal data and resulting ellipsoids.

makepolyhedron(*centre*, *ligands*, *atomdict=None*, *ligtypes=None*)
Make polyhedron from centre and ligands.

mtensor()
Return metric tensor from cell parameters

orthomatrix()
Return orthogonalisation matrix from cell parameters.

5.4 Plot Ellipsoid

Class to generate 3D interactive images of ellipsoids.

class distellipsoid.plotellipsoid.**EllipsoidImage**(*figure=None*, *axes=None*)

clearlast(*removepoints=True*, *removeaxes=True*, *removeframe=True*)
Remove previous plot

plotell(*ellipsoid*, *plotpoints=True*, *plotaxes=True*, *cagecolor='b'*, *axcols=None*, *cagealpha=0.2*, *pointcolor='r'*, *pointmarker='o'*, *pointscale=100*, *title=None*, *equalaxes=True*)
Plot an ellipsoid

5.5 CIF calculation routines

5.5.1 calcfromcif

distellipsoid.calcellipsoid.**calcfromcif**(*CIF*, *centres*, *radius*, *allligtypes=[]*, *alllignames=[]*, ***kwargs*)
Main routine for computing ellipsoids from CIF file.

5.5.2 multiCIF

The main module for computing ellipsoids from a number of files, using multiprocessing (one core per CIF file) if required. Largely contains routines for error checking input commands and calling *calcfromcif*. Module for processing one or more CIF files using supplied options To run from the command line, call the CIFellipsoid.py script.

class distellipsoid.multiCIF.**QueueHandler**(*queue*)
This is a logging handler which sends events to a multiprocessing queue. The plan is to add it to Python 3.2, but this can be copy pasted into user code for use with earlier Python versions.

emit(*record*)
Emit a record. Writes the LogRecord to the queue.

distellipsoid.multiCIF.**check_centres**(*cifs*, *centres*)
Determine which centres to use based on CIF contents and user-supplied arguments.

distellipsoid.multiCIF.**check_labels**(*labels*, *alllabs*)
Check that all labels are present in all cif files, handling regular expressions if needed. Returns ——- list of labels to test, list of labels to omit, list of missing labels

distellipsoid.multiCIF.**check_ligands**(*cifs*, *ligtypes*, *liglbls*)
Find lists of ligand labels and types that should be used to define ellipsoids based on supplied lists.

NOTE: This function will find the appropriate combination of label/type commands to find all required ligands,
EXCEPT where a site is found in multiple CIFs with the same label, but different type. In this case, the returned lists of labels and types *should* work correctly when run through calcellipsoid.calcfromcif

distellipsoid.multiCIF.**listener_configurer**(*name=None*)
Function to configure logging output from sub processes

distellipsoid.multiCIF.**listener_empty_config**(*name=None*)
Empty listener configurer, to do nothing except pass logs directly to parent

distellipsoid.multiCIF.**listener_process**(*queue*, *configurer*)
Process waits for logging events on queue and handles then

distellipsoid.multiCIF.**main**(*cifs*, *centres*, ***kwargs*)
Process all supplied cif files using options supplied as kwargs (or defaults).

Returns

phases [Dict] Dictionary of Crystal objects (containing ellipsoid results), keyed by CIF name.

plots [dict or dicts, optional] Dictionary of summary plots (keyed by CIF name) containing Dictionary of plots (keyed by ellipsoid centre)

distellipsoid.multiCIF.**run_parallel**(*cifs*, *testcen*, *radius=3.0*, *ligtypes=[]*, *lignames=[]*, *maxcycles=None*, *tolerance=1e-06*, *procs=None*)
Run ellipsoid computation in parallel, by CIF file

distellipsoid.multiCIF.**worker_configure**(*queue*)
Initialise logging on worker

5.6 Utility Functions

A number of utility functions are provided to simplify generation of polyhedra and reading/writing of files.

5.6.1 Unit Cell Functions

distellipsoid.readcoords.**makeP1cell**(*atomcoords*, *symmops*, *symmid*)
Generate full unit cell contents from symmetry operations from Cif file

Returned atom labels (as dict keys) are appended with '_##' to denote the number of the symmetry operation that generated them from cif file (initial position label is not changed).

distellipsoid.readcoords.**findligands**(*centre*, *atomcoords*, *orthom*, *radius=2.0*, *types=[]*, *names=[]*, *atomtypes=None*)
Find all atoms within radius of centre

File Functions

distellipsoid.readcoords.**readcif**(*FILE*)
Read useful data from cif using PyCifRW.

Module to write distellipsoid ellipsoid parameters to text file.

distellipsoid.writeproperties.**writeall**(*FILE*, *phase*, *verbosity=3*, *overwrite=False*)

Write data to file

Increasing verbosity above 0 will increase the amount of data printed to file (4 is maximum output)

CHAPTER

SIX

GLOSSARY

CIF Crystallographic Information File

ellipsoid A three-dimensional object formed by rotating an ellipse.

citation James Cumby & J. Paul Attfield, *Ellipsoidal Analysis of Coordination Polyhedra*.

MBE Minimum Bounding Ellipsoid. The smallest volume *ellipsoid* that can surround a set of points.

GUI Graphical User Interface

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

d

Symbols

A

B

C

D

E