pychemia Documentation

Release 0.1.2

Guillermo Avendano-Franco

Feb 20, 2018

Contents

Contents:

CHAPTER 1

Introduction

PyChemia is a python library for automatize atomistic simulations. PyChemia is build around a core module with two classes and a set of some other modules that offers a variety of operations in order to perform more complex operations.

The core is made of two classes 'Structure' and 'Composition'. Other modules are:

- **analysis:** Uses pure structure information for changing structures, matching atoms between two structures, create slabs, surfaces and some other geometrical operations on structures.
- **code:** This module deals with creating inputs and reading outputs from several atomistic simulation codes. Right now, we support ABINIT, DFTB+, Fireball, an internal calculator for LennardJones Clusters, Octopus and VASP.
- **core:** The core of PyChemia are two classes that are imported at the root level of the library: Structure and Composition. For PyChemia, Structure is a set of sites with one or more atoms located on each site with a probability associated to them. The Structure could be finite or periodic in one or more directions. In the case of a crystal the Structure will have also a lattice. Composition is a set o atoms of a define species. For crystals is the set of atoms on each unit cell. No geometry is store on a Composition object, and the order of atoms is irrelevant.
- **crystal:** For the case of periodic structures in three directions, a set of modules is created for three basic properties of a crystal. The class 'KPoints' store the description a a k-point mesh, path, or direct list of points on the reciprocal lattice. The class 'Lattice' store and manipulate the cell vectors of a crystal. The third class is 'CrystalSymmetry' for computing the Space Group, getting structures on the Bravais cell and finding the primitive cell.
- **db:** PyChemia uses MongoDB as Database for storing Structures and the properties computed by atomistic simulation codes. There are two classes defined, PyChemiaDB to store structures and properties and PyChemiaQueue to store a massive set of calculations and the status of those calculations.
- **dm:** This is a module in development, it contains classes for DataMining PyChemia Databases and global searches. Right now it contains a class for Network analysis, but in the future will provide interfaces with some other libraries for machine learning.
- **evaluator:** There are two circumstances where a atomistic simulation can be perform. If you have a machine without a queue system PyChemia will provide a very simplified queue for computing concurrent calcu-

lations under the constrains of your number of cores. If you use a queue system, the evaluator will setup the batch scripts, submit the jobs and monitor the status of those jobs, finally when the job is done, it will collect the final data and update the the databases.

- **external:** There are some other packages for which PyChemia worth interact, ASE (Atomistic Simulation Environment) is a python package supporting a remarkable number of calculators. Findsym is a close software to compute spacegroups and computing the CIF file of a give structure. 'pymatgen' is python code behind 'Materials Project' and a outstanding piece of very well written code. Originally implemented for VASP only now includes support for ABINIT.
- **io:** There are a pletora of formats for describing atomistic structures. We support three basic file-formats, ascii, CIF and XYZ. This module provides the classes for reading and writing them.
- **md:** Molecular Dynamics is an important operation for atomistic simulations. This module provide a 'in-house' calculator for MD simulations, using the forces computed from static calculations from an external atomistic code.
- **population:** A population is basically a collection of candidates collected somehow by a global searcher or from a DB query. PyChemia includes classes for populations of Lennard-Jones clusters, populations of vectors on a N-dimensional space.
- **runner:** Controls the executions in cluster by creating batch scripts and checking the status of the queue. Right now only supports Torque
- **searcher:** Global search operations, the prototypical case is structural search but any pychemia population could be use for searching. Several metaheuristic algorithms have been implemented
- **utils:** Several small routines, like a periodic table, mathematical operations, conversions and serializers.
- **visual:** PyChemia includes a set of classes and routines for interfacing with some other libraries and external software for data visualization, 3D plotting and graphic representation. pyprocar plots band-structures, LatticePlot and StructurePlot uses mayavi for visualizing atomic configurations and lattices. We have also interfaces for Povray and a developing interface to D3.js.
- **web:** On development, creation of a web interface for looking into the pychemia databases and controling executions. The web interface is build on top of Python-Bottle and CherryPy to access the database.

PyChemia is a open-source Python library for High-throughput first-principles materials discovery. The focus of this library is on structural search and data analysis. The ultimate purpose of the code is to optimize the search of new materials using a variety of methods such as Minima hopping method (MHM), soft-computing-based methods and statistical methods.

The main objectives of the code are:

1. Provide flexible classes for atomic structures such as molecules, clusters, thin films and crystals.
2. Manipulate both input and output for DFT and Tight-binding codes such as VASP. ABINIT, Fireball and DFTB+
3. Offer a robust architecture for storing a large collection of structures. Structural search methods generate many structures that are stored in repositories. Calculations done on those structures are also store in repositories.
4. Similarity analysis based on fingerprints, pair correlation and comparators.
5. Stability analysis for crystals. Including thermal analysis (Enthalpy) and dynamic stability (Phonons)
6. Tools for producing comprehensive reports, convex hulls, band structures, density of states, etc
7. Datamining tools to extract knowledge from the structures found, identify patterns in the data and identify suitable candidates for technological applications
8. A web interface

This is a new project and many classes and methods are refactored frequently. This is and will be a work in progress. We hope to stabilize the most critical classes for the release 1.

This code is open-source. We also welcome extra hands to improve this library with your own contributions. At present only one developer has being in charge of the project. More hands and eyes are very welcomed.

CHAPTER 2

Installation

There are basically two ways of installing Pychemia, using pip or by cloning the Github repository.

2.1 Installing PyChemia with pip

This is probably the easiest way, pip will download the code, check and eventually install dependencies and installing the package on a system-wide place or the home directory. All you have to do is execute this command:

```
sudo pip install pychemia
```

or for python 3.x:

```
sudo pip3 install pychemia
```

If you are on a machine where you do not have superuser privileges you can install pychemia on your home directory by adding the command '–user':

```
pip install pychemia --user
```

or for python 3.x:

```
pip3 install pychemia --user
```

This is a simple and usually will work for you. The only situation where it could give you problems is when pip tries to compile and install dependencies. In particular, pychemia requires the following packages and versions:

```
numpy  >= 1.11.0
scipy  >= 0.17.0
future >= 0.15.2
spglib >= 1.9.4
```

Now, most Linux distributions will probably come with older versions of numpy and scipy. Even more, Linux clusters usually have a very conservative approach related to packages and old versions of `numpy` and `scipy` are probably installed. `numpy` and `scipy` are python libraries that use the old and known BLAS and LAPACK libraries for all the usual linear algebra operations. If you encounter problems related with numpy and scipy, check if you have installed blas and lapack on your system.

The python library `future` is a small package that helps keeping compatibility with python 2 and python 3 on the same source code. Finally `spglib` is a C-library with python wrappers for computing space groups and related functionality.

We try to keep the dependencies of PyChemia to a very minimum. Some other libraries provides extra functionality that could be necessary for some tasks. Consider install pymongo, nose and matplotlib. You can do that using pip with the command:

```
sudo pip install pymongo nose matplotlib
```

Remember that you can use `--user` if you want to install on your home directory without special privileges. The package 'pymongo' offers connectivity with a Mongo database, a must if you plan to use 'pychemia.population' subpackage or doing global searches with the 'pychemia.searcher' subpackage. Matplotlib is the 'standard de-facto' for 2D plots on python. Many of the functionalities on 'pychemia.visual' subpackage depends on it. Nose is a python package for executing automatize tests for PyChemia. If you want to use it more information is below.

2.2 Installing PyChemia on a virtual environment

PyChemia requieres a relative recent version of Python (Python 2.7 appeared on July 3, 2010). However, it is common for some HPC infraestructures to use Linux installations such as RHEL 6 that only provides python 2.6 dating back to 2008. You can contour this situation by using Software Collections:

```
https://www.softwarecollections.org/en/
```

Install a more recent version of python and virtualenv. After that you can install PyChemia and its dependencies contained on a virtual environment following this commands:

1. Create a new virtual environment, lets call it "ve27"

virtualenv ve27

2. Activate the environment by sourcing the activate script

source ve27/bin/activate

3. You will notive that the prompt changes and you get your path pointing to a containerized python environment where you can install PyChemia and its dependencies

pip install pychemia

4. After a couple of minutes, you will get all the dependencies installed and the software ready for use without touching the libraries and modules installed by your Linux distribution. You can leave the virtual environment with the command:

deactivate

To re-enable the environment all you have to do is execute step 2 and you get the environment ready to work.

2.3 Installing PyChemia from github

The current stable repository for PyChemia is on Github, you can download the master branch with the command:

```
git clone https://github.com/MaterialsDiscovery/PyChemia.git
```

If you get a message such as:

```
$ git clone https://github.com/MaterialsDiscovery/PyChemia.git
-bash: git: command not found
```

You need to install `git` first. On machines from the Debian 'lineage' (Ubuntu, Mint, and many others) you can use the command:

```
sudo apt-get install git
```

On systems with `yum` you can use:

```
sudo yum install git
```

Now that you have 'cloned' the PyChemia repository you have two options. Install the package by using the set of commands:

```
cd PyChemia
python setup build
python setup install --user
```

Use `python3` for the commands above, if you want to use python 3.x instead.

Another alternative is add path where you downloaded the repository to the variable $PYTHONPATH. You can do that by editing your `.bashrc` file. Supposing that you execute the `git clone` command directly on your home directory you can add the path for PyChemia adding this line to your .bashrc:

```
export PYTHONPATH=$HOME/PyChemia:$PYTHONPATH
```

If you want the changes on .bashrc take inmediate effect execute:

```
source $HOME/.bashrc
```

2.4 Importing the library

No matter how you installed PyChemia, you should be able to load the library. You can use the traditional python terminal, for example:

```
$ python3.5
Python 3.5.1 (default, Mar  2 2016, 03:38:02)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import pychemia
>>> pychemia.info()
PyChemia
--------

Version: 0.1.2
Path:    /Users/guilleaf/PyChemia/pychemia
Date:    May 13, 2016

Python version=3.5.1 (default, Mar  2 2016, 03:38:02)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)]
```

```
pymongo Not Found
     numpy     1.11.0   /opt/local/Library/Frameworks/Python.framework/Versions/3.5/
↪lib/python3.5/site-packages/numpy
     scipy     0.17.1   /opt/local/Library/Frameworks/Python.framework/Versions/3.5/
↪lib/python3.5/site-packages/scipy
    mayavi            Not Found
Scientific            Not Found
matplotlib      1.5.1   /opt/local/Library/Frameworks/Python.framework/Versions/3.5/
↪lib/python3.5/site-packages/matplotlib
    future     0.15.2   /opt/local/Library/Frameworks/Python.framework/Versions/3.5/
↪lib/python3.5/site-packages/future
      nose      1.3.7   /opt/local/Library/Frameworks/Python.framework/Versions/3.5/
↪lib/python3.5/site-packages/nose
  coverage      4.0.3   /opt/local/Library/Frameworks/Python.framework/Versions/3.5/
↪lib/python3.5/site-packages/coverage
    spglib      1.9.4   /Users/guilleaf/Library/Python/3.5/lib/python/site-packages/
↪spglib
    pyhull            Not Found
  pymatgen            Not Found
      qmpy            Not Found
       ase            Not Found
```

The method `pcyhemia.info()` will inform about the several libraries that PyChemia uses, both mandatory and optional, their versions and path. That could be informative in case of something not working as expected.

2.5 Testing with nose

It is always important to test a library, not only from the developer point of view, but also for an user. Nose is a python package that offers a simple command to execute predefined test for a python package and report any errors or inconsistencies from the expected resuts.

Direct the terminal to the place where PyChemia is located. Lets suppose that you have pychemia on `/Users/guilleaf/PyChemia`, test PyChemia using the command:

```
cd /Users/guilleaf/PyChemia
nosetests -v
```

The name of the command could have small variations according to your distribution. On a MacOS using macports the name could be for example `nosetests-2.7` or `nosetests-3.5` for python 2 and 3 respectively.

`nosetests` will search for tests on the entire package and subpackages. If everything is fine (and you use `-v` for verbose output) you will get something like:

```
...
Example of a simple calc                                     : ... ok
Example of a multiple calc                                   : ... ok

----------------------------------------------------------------------
Ran 38 tests in 5.469s

OK
```

That is an indication that all tests were successful and eventually you are ready to use the library.

CHAPTER 3

First Steps

This is short and direct introduction to some of the basic functionalities from pychemia. To keep things simple, we will be using only the mandatory libraries, and we will not call any external simulation packages.

You can use the official python terminal for executing this examples, however, you can gain some extra advantage by using the far more powerfull IPython terminal. On Linux machines with Debian, Ubuntu or Linux Mint, you can use the following command to install the IPython terminal:

```
sudo apt-get install python-ipython
```

or for python 3.x:

```
sudo apt-get install python3-python
```

On MacOS using macports,

> sudo port install py27-ipython

or for python 3.x

> sudo port install py35-ipython

3.1 FCC crystal of gold

Most of the case of use for pychemia start with a Structure. A PyChemia Structure stores atomic positions and cell parameters for periodic structures. Lets start with a very simple structure, the FCC structure of gold. One way of creating and structure is by directly specifying atomic positions and cell parameters, for example:

```
$ ipython3-3.5
Python 3.5.1 (default, Mar  2 2016, 03:38:02)
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
```

```
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import pychemia

In [2]: a=4.05

In [3]: b=a/2

In [4]: fcc = pychemia.Structure(symbols=['Au'], cell=[[0, b, b], [b, 0, b], [b, b,␣
↪0]], periodicity=True)

In [5]: fcc
Out[5]: Structure(symbols=['Au'], cell=[[0.0, 2.025, 2.025], [2.025, 0.0, 2.025], [2.
↪025, 2.025, 0.0]],
reduced=[[0.0, 0.0, 0.0]], periodicity=True)
```

You have created your first structure on PyChemia. The structure contains one atom of Gold and the cell is the primitive cell of a FCC structure with one atom. You can 'see' the structure using the 'print' function:

```
In [6]: print(fcc)
1

 Symb  (             Positions            ) [     Cell-reduced coordinates     ]
   Au  (     0.0000     0.0000     0.0000 ) [     0.0000     0.0000     0.0000 ]

Periodicity:  X Y Z

Lattice vectors:
     0.0000     2.0250     2.0250
     2.0250     0.0000     2.0250
     2.0250     2.0250     0.0000
```

We can now be interested in see the spacegroup of the structure just created. A pychemia.crystal.CrystalSymmetry object allow us to get access to symmetry calculations provided by the `spglib` library:

```
In [7]: cs=pychemia.crystal.CrystalSymmetry(fcc)

In [8]: cs.number()
Out[8]: 225

In [9]: cs.symbol()
Out[9]: 'Fm-3m'
```

Both the space group number and the International symbol are accessible from the CrystalSymmetry object. Now we can use the CrystalSymmetry object to recreate the convenctional FCC cell:

```
In [10]: fcc_conv=cs.refine_cell()

In [11]: fcc
fcc        fcc_conv

In [12]: fcc_conv
Out[12]: Structure(symbols=['Au', 'Au', 'Au', 'Au'], cell=4.05, reduced=[[0.0, 0.0, 0.
↪0], [0.0, 0.5, 0.5],
[0.5, 0.0, 0.5], [0.5, 0.5, 0.0]], periodicity=True)

In [11]: print(fcc_conv)
```

```
4

 Symb  (             Positions            ) [     Cell-reduced coordinates     ]
   Au  (     0.0000     0.0000     0.0000 ) [     0.0000     0.0000     0.0000 ]
   Au  (     0.0000     2.0250     2.0250 ) [     0.0000     0.5000     0.5000 ]
   Au  (     2.0250     0.0000     2.0250 ) [     0.5000     0.0000     0.5000 ]
   Au  (     2.0250     2.0250     0.0000 ) [     0.5000     0.5000     0.0000 ]

Periodicity:  X Y Z

Lattice vectors:
     4.0500     0.0000     0.0000
     0.0000     4.0500     0.0000
     0.0000     0.0000     4.0500
```

The lattice vectors are now making a cube, and the structure is made of 4 gold atoms. Now we can want to create a supercell, 2x2x1 using the supercell method:

```
In [13]: fcc221=fcc_conv.supercell(size=(2,2,1))

In [14]: print(fcc221)
16

 Symb  (             Positions            ) [     Cell-reduced coordinates     ]
   Au  (     0.0000     0.0000     0.0000 ) [     0.0000     0.0000     0.0000 ]
   Au  (     0.0000     2.0250     2.0250 ) [     0.0000     0.2500     0.5000 ]
   Au  (     2.0250     0.0000     2.0250 ) [     0.2500     0.0000     0.5000 ]
   Au  (     2.0250     2.0250     0.0000 ) [     0.2500     0.2500     0.0000 ]
   Au  (     0.0000     4.0500     0.0000 ) [     0.0000     0.5000     0.0000 ]
   Au  (     0.0000     6.0750     2.0250 ) [     0.0000     0.7500     0.5000 ]
   Au  (     2.0250     4.0500     2.0250 ) [     0.2500     0.5000     0.5000 ]
   Au  (     2.0250     6.0750     0.0000 ) [     0.2500     0.7500     0.0000 ]
   Au  (     4.0500     0.0000     0.0000 ) [     0.5000     0.0000     0.0000 ]
   Au  (     4.0500     2.0250     2.0250 ) [     0.5000     0.2500     0.5000 ]
   Au  (     6.0750     0.0000     2.0250 ) [     0.7500     0.0000     0.5000 ]
   Au  (     6.0750     2.0250     0.0000 ) [     0.7500     0.2500     0.0000 ]
   Au  (     4.0500     4.0500     0.0000 ) [     0.5000     0.5000     0.0000 ]
   Au  (     4.0500     6.0750     2.0250 ) [     0.5000     0.7500     0.5000 ]
   Au  (     6.0750     4.0500     2.0250 ) [     0.7500     0.5000     0.5000 ]
   Au  (     6.0750     6.0750     0.0000 ) [     0.7500     0.7500     0.0000 ]

Periodicity:  X Y Z

Lattice vectors:
     8.1000     0.0000     0.0000
     0.0000     8.1000     0.0000
     0.0000     0.0000     4.0500
```

CHAPTER 4

Tutorials

This is a set of tutorials intended to follow complete tasks that interconnect several modules and classes to achieve a given objective.

4.1 PyChemia CookBook

This is a set of small recipes for getting the 'job done' quickly using the methods and classes implemented on PyChemia

4.1.1 Reading a formula and extracting the composition

Consider that you have a formula such as 'YBa2Cu3O7' and you would like to get the composition and the number of atoms of each species. The recipe is very simple:

```
>>> import pychemia
>>> formula = 'YBa2Cu3O7'
>>> comp = pychemia.Composition(formula)
>>> comp['Ba']
2
```

The Composition object acts like a python dictionary, with the particularity of returning 0 when we ask for species non present on a given composition:

```
>>> comp['Au']
0
```

4.1.2 Converting an ascii file into a POSCAR

Consider the following ascii file stored on `test.ascii`:

```
None
11.47012476778924 0.79937702290141 9.51246277071292
-2.99939838492446 -0.12947182393907 7.79142604544631
   0.6793241103939325 -0.0865078526005124  5.2059167421975845  Mg
   7.2961243488047218  7.8694796435395364  5.3644840894866279  Mg
   2.9186811537636199  0.3979635592494910  1.4510323645408516  Ca
  -0.9269026176064651  2.5195450955593404  2.9685499969876323  Ca
  -0.8684364924472798  6.8223765997007311  5.7332541305138323  Ca
   7.7311137749974712  6.1831255116012693  0.0000000000000000  Ca
```

This is how pychemia can convert it into a POSCAR file:

```
>>> import pychemia
>>> st = pychemia.io.ascii.load('test.ascii')
>>> pychemia.code.vasp.write_poscar(st, 'POSCAR.test')
```

The final archive called `POSCAR.test` will looks like this:

```
 Mg Ca
1.0
  11.4701247677892404   0.0000000000000000   0.0000000000000000
   0.7993770229014100   9.5124627707129203   0.0000000000000000
  -2.9993983849244601  -0.1294718239390700   7.7914260454463102
 Mg Ca
 2 4
Direct
   0.2339469912681612   1.0000000000000000   0.6681596811459407
   0.7578333625215485   0.8366521516169854   0.6885111991304717
   0.3000667985403045   0.0443708102021889   0.1862345039376849
   0.0000000000000000   0.2700535286675975   0.3810021400026761
   0.0660256684506364   0.7272193856947440   0.7358414360955946
   0.6287217253130448   0.6500025977119140   0.0000000000000000
```

4.1.3 Compute the hardness of a given structure

4.1.4 Compute the ideal strength

4.1.5 Read and Write structures between several atomistic codes

4.1.6 Search for structures from the PyChemia database

4.1.7 Generate and manipulate VASP input files

4.1.8 Convergence studies for VASP

4.1.9 Read and write the input from an atomistic code

4.1.10 Read the output from an atomistic code

4.1.11 Build an manipulate the lattice

4.1.12 Rotate the cell along a Miller index

Very often for creating surfaces we need to rotate the structure along specific Miller indices. We can achieve this with a routine for that, see this example:

```
import pychemia
pychemia_path = pychemia.__path__[0]
st=pychemia.code.vasp.read_poscar(pychemia_path + '/test/data/vasp_06/POSCAR')
st2=pychemia.analysis.surface.rotate_along_indices(st, 1,1,1, 2)
```

Both structures represent the same crystal, the second structure has the 'c' axis parallel to the 111 Miller direction. Additionally we ask for having 2 layers of the cristal along the 'c' axis. We can check that the symmetry of the crystal remains the same:

```
sym=pychemia.crystal.CrystalSymmetry(st)
sym.number()
139

sym=pychemia.crystal.CrystalSymmetry(st2)
sym.number()
139
```

Here you see the result of the two structures, the original and the rotated along the 111 axis.

images/conventional.jpg

images/rotated.jpg

4.1.13 Get the spatial group

Lets start reading a POSCAR for a Carbon diamond structure:

```
import pychemia
pychemia_path = pychemia.__path__[0]
st=pychemia.code.vasp.read_poscar(pychemia_path + '/test/data/vasp_08/POSCAR_old')
print(st)
```

You should get:

```
2

 Symb  (             Positions            ) [     Cell-reduced coordinates     ]
    C  (     0.0000      0.0000      0.0000 ) [     0.0000      0.0000      0.0000 ]
    C  (     0.9250     0.9250     0.9250 ) [     0.2500     0.2500     0.2500 ]

Periodicity:  X Y Z

Lattice vectors:
     1.8500     1.8500     0.0000
     0.0000     1.8500     1.8500
     1.8500     0.0000     1.8500
```

PyChemia uses spglib to get the space-group and use some other routines to get the primitive and convectional cells and to reposition the atoms to precise positions for a given symmetry. All this functionality is provided by creating `CrystalSymmetry` object:

```
sym=pychemia.crystal.CrystalSymmetry(st)
```

We can get space groups using the number:

```
sym.number()
227
```

Or as internation symbol:

```
sym.symbol()
'Fd-3m'
```

In both cases, there is a tolerance that can be adjusted, the default value is 1e-5 that could be too strong for structures produced by DFT calculations with crude relaxations.

To exemplify this situation consider this structure (Zn2V2O7) whit positions truncated to 4 decimals:

```
st = pychemia.code.vasp.read_poscar(pychemia_path + '/test/data/vasp_07/POSCAR_trunc')
sym = pychemia.crystal.CrystalSymmetry(st)
sym.number()
```

You will get space group equal to 9, however adjusting the tolerance you will get the value from the precise structure:

```
sym.number(1E-2)
15
```

And the same works for the symbol:

```
sym.symbol(1E-2)
'C2/c'
```

4.1.14 Get the primitive cell

The primitive cell is obtained from the `CrystalSymmetry` object again using the functionality of the spglib library:

```
import pychemia
pychemia_path = pychemia.__path__[0]
st = pychemia.code.vasp.read_poscar(pychemia_path + '/test/data/vasp_06/POSCAR')
sym = pychemia.crystal.CrystalSymmetry(st)
print(st)
```

The structure looks like this:

```
    4

 Symb  (             Positions            ) [     Cell-reduced coordinates     ]
    P  (     0.0000     0.0000     3.7190 ) [     0.0000     0.0000     0.7078 ]
    P  (     0.0000     0.0000     1.5350 ) [     0.0000     0.0000     0.2922 ]
    P  (     2.0308     2.0308     1.0920 ) [     0.5000     0.5000     0.2078 ]
    P  (     2.0308     2.0308     4.1620 ) [     0.5000     0.5000     0.7922 ]

Periodicity:  X Y Z

Lattice vectors:
     4.0616     0.0000     0.0000
     0.0000     4.0616     0.0000
     0.0000     0.0000     5.2540
```

We can get the primitive like this (eventually using a tolerance as an argument):

```
stp = sym.find_primitive()
print(stp)
```

And the structure is reduced to a cell with just 2 atoms:

```
2

 Symb  (             Positions            ) [     Cell-reduced coordinates     ]
    P  (     0.0000     0.0000     1.5350 ) [     0.2922     0.2922     0.0000 ]
    P  (     0.0000     0.0000     3.7190 ) [     0.7078     0.7078     0.0000 ]

Periodicity:  X Y Z

Lattice vectors:
    -2.0308     2.0308     2.6270
     2.0308    -2.0308     2.6270
     2.0308     2.0308    -2.6270
```

4.1.15 Get the conventional cell

The conventional cell is obtained as a result of the refinement of the cell, ie adjusting the positions precisely to satisfy a given tolerance. From the previous section, lets reconstruct a convectional cell from the primitive:

```
sym = pychemia.crystal.CrystalSymmetry(stp)
stc = sym.refine_cell()
print(stc)
```

You should get:

```
4

 Symb  (             Positions            ) [     Cell-reduced coordinates     ]
    P  (     0.0000     0.0000     1.5350 ) [     0.0000     0.0000     0.2922 ]
    P  (     0.0000     0.0000     3.7190 ) [     0.0000     0.0000     0.7078 ]
    P  (     2.0308     2.0308     4.1620 ) [     0.5000     0.5000     0.7922 ]
    P  (     2.0308     2.0308     1.0920 ) [     0.5000     0.5000     0.2078 ]

Periodicity:  X Y Z
```

```
Lattice vectors:
     4.0616      0.0000      0.0000
     0.0000      4.0616      0.0000
     0.0000      0.0000      5.2540
```

Now lets refine the structure with positions truncated and reconstruct a cell with positions precisely in place to the symmetry found:

```
st = pychemia.code.vasp.read_poscar(pychemia_path + '/test/data/vasp_07/POSCAR_trunc')
sym = pychemia.crystal.CrystalSymmetry(st)
st2 = sym.refine_cell(1E-2)
sym = pychemia.crystal.CrystalSymmetry(st2)
```

Here we took the structure with positions truncated, and refined the cell using a tolerance that return the an space group 15, after that we create a new `CrystalSymmetry` object from the new structure and we can verify that the space group is preserved up to very strict tolerances:

```
sym.number()
15

sym.number(1E-14)
15
```

4.2 Global Minimimization of Lennard-Jones Clusters

This tutorial will guide to how search for global minima using the methods implemented on PyChemia.

4.2.1 Quick version

The shortest version of a global search using the FireFly method will look like this

```
>>> from pychemia.searcher import FireFly
>>> from pychemia.population import LJCluster
>>> popu = LJCluster('LJ13', composition='Xe13', refine=True, direct_evaluation=True)
>>> searcher = FireFly(popu, generation_size=16, stabilization_limit=10)
>>> searcher.run()
```

For this case, you should have a mongo server running on you local machine, no SSL encryption and no authorization with username and password. The population will be created with Lennard-Jones clusters with 13 particles each. Each new candidate is locally relaxed when created. The searcher will use 16 candidates on each generation and will stop when the best candidate survives for 10 generations.

4.3 PyChemia Software Framework

PyChemia is framework for materials discovery, more than just compute DFT calculations for structures already present in databases such as ICSD, PyChemia searches for new structures that could never have been synthesized or reported before.

To achieve its goal, PyChemia relies on a number of methods of structural search such as minima hoping method, genetic algorithms and other population based algorithms.

As a software framework, PyChemia is structurated around five axis. They are:

1. Structural Search Methods
2. Storage and Databases
3. Data Mining or Knowledge Discovery in Data
4. Execution and Analysis
5. Reporting and Visualization

We will develop those five axis and the relations between them

4.3.1 Structural Search Methods

One of the distinctive characteristics of PyChemia is its ability to search for new structures. Ab-initio calculations of structures present of databases such as ICSD becomes limited to the extension of the original databases. Even if such effort is indeed a big challenge with large databases, those databases represent a small portion of the structures that could be created. The ability to predict new structures and target those findings to specific applications is a task of technological relevance.

PyChemia was created with an strong focus on structural prediction. PyChemia implements several methods, from minima hoping method to metaheuristics

4.3.1.1 Minima Hoping Method

4.3.1.2 Metaheuristics

4.3.2 Storage and Databases

If PyChemia were a software to compute ab-initio calculations from structures taken from another database or from very predictive set of prototypes only one database will be enough. However, as we mention before PyChemia is focused on structural search.

The kind of algorithms that we describe above typically explore thousands of different structures before selecting a reduced subset of thermodynamically stable or metastable ones.

Also, the search for new structures must be guided by specific applications of interest, batteries, thermoelectrics, superconductors, etc. Different applications are associated to different properties in the electronic structure.

Those two elements, the dynamic nature of structural search methods and the flexibility in the data that we would like to store is the reason why we are not using a single database and why we departed from traditional SQL schemas.

PyChemia was designed to create new databases for each structural search. We still keep one large database with a curated selection of structures but the ability to create small and flexible databases is central for the success of PyChemia.

PyChemia relies on MongoDB. MongoDB is an open-source document database, and the leading NoSQL database. We take advantage of dynamic schemas that offer simplicity and power and are in harmony with the own principles of scientific research. Different structures are intended for different applications and different applications requieres the calculation of different physical properties.

4.3.3 Data Mining or Knowledge Discovery in Data

Data Mining or more correctly Knowledge Discovery in Data is the computational process of discovering patterns in large data sets involving methods at the intersection of artificial intelligence, machine learning, statistics, and database systems.

Structural Search algorithms has the ability to gererate in the long term more structures that those that could be synthesized in reasonable time. As the database becomes large traditional techniques based on estimate good candidates based on just a few properties will be replaced by automatize algorithms that search patterns in structures and predict were new stochiometries deserve exploration.

4.3.3.1 Bayesian networks and Gaussian Processes

4.3.4 Reporting and Visualization

Doing electronic structure calculations for thousands of materials is challenging but there is not point in doing that without the ability to communicate to experimentalist structures that worth to be synthesized. PyChemia uses Django as a web-frontend to display and explore the computations not only on the global database but also on the specific database product of a structural search run.

4.3.5 Execution and Analysis

There are two levels of execution in structural search runs. Ab-initio calculations and structural analysis. PyChemia provides concurrent execution and basic job management for running multiple ab-initio calculations on clusters or non-queue systems

PyChemia relies on state-of-the-art ab-initio software packages to compute electronic structure and their properties. In particular PyChemia has support for VASP, ABINIT, Octopus, Fireball and DFTB+

Different levels of theory allows for a better compromise between accuracy and computational cost.

Structural analysis is defined as all kinds of procedures that relies only on structural data, for example structural fingerprints, topological analysis, hardness calculations, comparators and symmetry calculations. PyChemia provides multi-threaded execution of structural analysis routines

4.4 Euler Angles in PyChemia

4.4.1 Short Version

You can use the following routines to obtain the $k(k-1)/2$ Generalized Euler angles from a orhogonal matrix of dimension k and for building the orthogonal matrix from a set of angles:

```
>>> angles_list = pychemia.utils.mathematics.gea_all_angles(ortho_matrix)


>>> ortho_matrix = gea_orthogonal_from_angles(angles_list)
```

Remember that the list of angles matters. The SO(k) group is non-abelian.

The algorithm is based on the paper:

```
Generalization of Euler Angles to N-Dimensional Orthogonal Matrices
David K. Hoffman, Richard C. Raffenetti, and Klaus Ruedenberg
Journal of Mathematical Physics 13, 528 (1972)
doi: 10.1063/1.1666011
```

4.4.2 Long Version

Rotation on a two-dimensional plane can be described with one angle. For a three dimensional space 3 angles are needed. You can define angles around each axis and general rotation matrices as a product of three single axis rotation matrices.

In general, for k dimensions, the number of angles is $k(k-1)/2$. One angle for each plane that you can get from any pair among the k vectors defining the space.

Any orthogonal matrix (with determinant equal to +1) represent a rotation matrix for the space that its dimension. Sometimes could be necessary to obtain the set of independent angles that the orthogonal matrix represents and having a way of regaining the original orthogonal matrix from a given set of angles.

One the reasons for moving between one orthogonal matrix and its angles is reducing an orthogonal matrix to its minimal independent parameters. The paper intitled: "Generalization of Euler Angles to N-Dimensional Orthogonal Matrices" provides an effective way to get the so called Euler angles for matrices of arbitrary dimension and reconstitute the matrix from a given set of angles.

This algorithm is implemented on PyChemia. As an example, lets build first a 7-dimensional orthogonal matrix by a QR decomposition:

```
In [1]: import pychemia

In [2]: import numpy as np

In [3]: np.set_printoptions(linewidth=200, suppress=True, precision=5)

In [4]: ortho = pychemia.utils.mathematics.gram_smith_qr(7)

In [5]:ortho
Out[5]:

array([[-0.23503793, -0.6039233 ,  0.31346146,  0.56383925,  0.18441349, -0.35292927,␣
↪ 0.07275729],
       [-0.31380685, -0.21108679, -0.25878856, -0.40710624, -0.23657101, -0.54173578,␣
↪-0.52423003],
       [-0.48237691,  0.1056854 , -0.2376831 ,  0.11195934,  0.63817833,  0.3621268 ,␣
↪-0.38562619],
       [-0.44643339, -0.33091574,  0.37743259, -0.27151849, -0.39852265,  0.56178535,␣
↪ 0.02431587],
       [-0.46408875,  0.67693995,  0.31398053,  0.30715275, -0.28175741, -0.23147005,␣
↪-0.02194835],
       [-0.43149085,  0.00696596, -0.2664925 , -0.31313612,  0.19768295, -0.20337001,␣
↪ 0.75117024],
       [-0.11282486, -0.10838891, -0.68280287,  0.48754035, -0.47483487,  0.20071602,␣
↪ 0.07649819]])
```

The variable 'ortho' is an orthogonal matrix as you can easily show by computing its determinant:

```
In [6]: np.linalg.det(ortho)
Out[6]: 1.0
```

Now, for a 7-dimension space we should expect 21 generalized Euler angles. We can get them by calling the function:

```
In [8]: angles_list = pychemia.utils.mathematics.gea_all_angles(ortho)

In [9]: np.array(angles_list)
Out[9]:
```

```
array([ -0.1392 , -0.03975, -0.37052, -0.48339,  0.13503,  0.39547, -0.22866, -0.
↪31214, -0.57967,  0.84238,
        -2.66098,  0.05294,  0.3535 , -0.83619, -0.06953, -0.51353, -1.01991,  2.
↪28529, -0.25373, -0.33108,
        -2.27736])
```

In fact we got 21 angles, all the angles in the range $[-\pi, \pi]$ We can rebuild the original orthogonal matrix from those angles with:

```
In [10]: matrix=pychemia.utils.mathematics.gea_orthogonal_from_angles(angles)
```

In fact, we can verify that we recover the original matrix:

```
In [11]: np.max(matrix-ortho)
Out[11]: 3.3306690738754696e-16
```

4.5 Optimization of Magnetic Moments (VASP)

This tutorial shows how to perform global search for optimal magnetic moments on VASP.

In VASP there are two variables that control the magnetic moments imposed on atoms and for contrain magnetic moments along predefined directions.

Lets start with one INCAR file that looks like this:

```
$SYSTEM    =  CaMnO3 Pnma
PREC       =  Accurate
NELMIN     =  6
NELM       =  100
EDIFF      =  1E-09
EDIFFG     = -5E-6
IBRION     =  1
ISIF       =  2
LREAL      = .FALSE.
ADDGRID    = .TRUE.
NSW        =  0
ISMEAR     = -5
ENCUT      =  500
ISPIN      =  2
LORBIT     =  11
LMAXMIX    =  4
ISYM       =  0
LSORBIT    = .TRUE.
RWIGS      = 2.00 1.87 1.78 1.24 # d-d/2
SAXIS      = 0  0  1
LPLANE     = .TRUE.
NPAR       = 2
LSCALU     = .FALSE.
NSIM       = 4
LWAVE      = .FALSE.
AMIX       = 0.8
BMIX       = 0.9
AMIX_MAG   = 0.4
BMIX_MAG   = 0.9

MAGMOM     = 12*0
```

```
            0.0000000 0.0416472 2.3859677
            0.0000000 0.0416472 -2.3859677
            0.0000000 0.0416472 -2.3859677
            0.0000000 0.0416472 2.3859677
            60*0
M_CONSTR  = 12*0
            0.0000000 0.0416472 2.3859677
            0.0000000 0.0416472 -2.3859677
            0.0000000 0.0416472 -2.3859677
            0.0000000 0.0416472 2.3859677
            60*0
I_CONSTRAINED_M = 1
LAMBDA    = 10

#LDAU      =  .TRUE.
#LDAUTYPE  =  1
#LDAUL     =  -1  2 -1
#LDAUU     =  0.0 4.0 0.0
#LDAUJ     =  0.0 0.0 0.0
#LDAUPRINT =  2
```

We have commented the variables related with LDA+U but the procedure works enabling those variables too. The variables MAGMOM and M_CONSTR controls the initial direction of Magnetic Moments and the constrained direction using LAMBDA as a parameter to control the intensity of the contrain.

Now, for different values of MAGMOM you can get variations on the total energy and the optimal magnetization can only be obtained by covering all possible directions.

The population 'NonCollinearMagMoms' defined the procedures to create a pool of candidates with random directions for Magnetic Moments and modify their directions in several ways suitable for being used by the global search algorithms implemented on PyChemia.

On this tutorial we will explore step by step how the methods on 'NonCollinearMagMoms' where implemented and how use them for efficiently

4.6 Global optimization of correlation matrices for DFT+U (Abinit)

4.6.1 The variable dmatpawu

The objective of this global search is finding the optimal values for the correlation matrices in DFT+U. For this tutorial consider the abinit input file stored on 'pychemia/test/data/abinit_dmatpaw/abinit.in'. First, we can read the abinit input and access the contents of the variable 'dmatpawu':

```python
import pychemia
import numpy as np
pychemia_path = pychemia.__path__[0]
abiinput = pychemia.code.abinit.InputVariables(pychemia_path + '/test/data/abinit_
↪dmatpawu/abinit.in')
dmatpawu = np.array(abiinput['dmatpawu']).reshape(-1,5,5)
```

The variable 'dmatpawu' stores the contents of 4 5x5 matrices, the correlation matrices for the corresponding 4 Co atoms in the crystal. The matrices can be converted into a numpy array with shape (4, 5, 5) and they look like this:

```
array([[[ 0.06256,  0.     ,  0.01218,  0.     ,  0.     ],
        [ 0.     ,  0.481  ,  0.     ,  0.45877,  0.     ],
        [ 0.01218,  0.      ,  0.07148,  0.      ,  0.      ],
        [ 0.      ,  0.45877,  0.      ,  0.481   ,  0.      ],
        [ 0.      ,  0.      ,  0.      ,  0.      ,  0.94038]],

       [[ 0.97852,  0.      ,  0.00305,  0.      ,  0.      ],
        [ 0.      ,  0.95493,  0.      , -0.00449,  0.      ],
        [ 0.00305,  0.      ,  0.98115,  0.      ,  0.      ],
        [ 0.      , -0.00449,  0.      ,  0.95493,  0.      ],
        [ 0.      ,  0.      ,  0.      ,  0.      ,  0.95168]],

       [[ 0.97852,  0.      , -0.00305,  0.      ,  0.      ],
        [ 0.      ,  0.95493,  0.      ,  0.00449,  0.      ],
        [-0.00305,  0.      ,  0.98115,  0.      ,  0.      ],
        [ 0.      ,  0.00449,  0.      ,  0.95493,  0.      ],
        [ 0.      ,  0.      ,  0.      ,  0.      ,  0.95168]],

       [[ 0.06256,  0.      , -0.01218,  0.      ,  0.      ],
        [ 0.      ,  0.481   ,  0.      , -0.45877,  0.      ],
        [-0.01218,  0.      ,  0.07148,  0.      ,  0.      ],
        [ 0.      , -0.45877,  0.      ,  0.481   ,  0.      ],
        [ 0.      ,  0.      ,  0.      ,  0.      ,  0.94038]]])
```

The objective is to find the set of correlation matrices that minimize the energy. Those are density matrices so even if we have 100 numbers, any set of numbers is a valid set of correlation matrices. We will now convert this set of matrices into a reduced set of variables that can be treated independently.

A correlation matrix can be express as the following product:

```
R*O*R^{-1}
```

Where R is a rotation matrix, O is a diagonal matrix with a trace that is the total number of electrons correlated. We need to find a set of independent variables to recreate any correlation matrix. We know that not any arbitrary set of 25 numbers is a good rotation matrix. However, a 5x5 rotation matrix can be effectively decomposed into 10 independent numbers, the so called "Generalized Euler angles", this set of angles reduces the 25 values from a 5x5 rotation matrix into 10 independent variables. We should also be aware that the occupations on the diagonal of the matrix 'O' are not exactly integers, we will account for the small differences into a separate set of values. With those premises a 5x5 correlation matrix is converted into a set with 10 euler angles, 5 occupations and 5 deltas. This is done by the routines 'dmatpawu2params' and 'params2dmatpawu' that allow us to go back and forward from the set of correlation matrices into the set of 'euler_angles', intger 'occupations' and the small 'deltas':

```
params = pychemia.population.orbitaldftu.dmatpawu2params(dmatpawu, 5)
```

The variable 'params' is a dictionary with values for 'occupations', 'deltas' and 'euler_angles':

```
{'deltas': array([[ 0.02223 ,  0.054049,  0.079991,  0.06023 ,  0.05962 ],
         [ 0.04956 ,  0.04832 ,  0.04058 ,  0.023486,  0.016844],
         [ 0.04956 ,  0.04832 ,  0.04058 ,  0.023486,  0.016844],
         [ 0.02223 ,  0.054049,  0.079991,  0.06023 ,  0.05962 ]]),
 'euler_angles': array([[ 0.      ,  0.      ,  0.      ,  0.      ,  0.785398,  0.   ␣
↪    ,
          0.      , -0.609893,  0.      , -1.570796],
         [-0.581865,  0.      , -1.570796,  0.      ,  0.785398,  0.      ,
          1.570796,  1.570796,  1.570796,  3.141593],
         [-0.581865,  0.      ,  1.570796,  0.      ,  0.785398,  0.      ,
          1.570796,  1.570796,  1.570796, -0.      ],
         [ 0.      ,  0.      ,  0.      ,  0.      ,  0.785398,  0.      ,
          0.      , -0.609893,  0.      ,  1.570796]]),
 'ndim': 5,
 'num_matrices': 4,
 'occupations': array([[0, 0, 0, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [0, 0, 0, 1, 1]])}
```

This is in fact the set of independent variables that we can use to optimize the correlation using a global searcher. We can also go back and recover the correlation matrices using the inverse procedure, ie, from the dictionary params recover the correlation matrices:

```
dmatpawu_new = pychemia.population.orbitaldftu.params2dmatpawu(params)
```

The dmatpawu is recovered from the values stored in 'params':

```
array([[[ 0.06256, -0.     , -0.01218, -0.     ,  0.     ],
        [-0.     ,  0.481  , -0.     ,  0.45877,  0.     ],
        [-0.01218, -0.     ,  0.07148, -0.     ,  0.     ],
        [-0.     ,  0.45877, -0.     ,  0.481  ,  0.     ],
        [ 0.     ,  0.     ,  0.     ,  0.     ,  0.94038]],

       [[ 0.97852, -0.     ,  0.00305, -0.     , -0.     ],
        [-0.     ,  0.95493, -0.     , -0.00449, -0.     ],
        [ 0.00305, -0.     ,  0.98115, -0.     , -0.     ],
        [-0.     , -0.00449, -0.     ,  0.95493,  0.     ],
        [-0.     , -0.     , -0.     ,  0.     ,  0.95168]],

       [[ 0.97852, -0.     , -0.00305, -0.     , -0.     ],
        [-0.     ,  0.95493,  0.     ,  0.00449,  0.     ],
        [-0.00305,  0.     ,  0.98115, -0.     ,  0.     ],
        [-0.     ,  0.00449, -0.     ,  0.95493,  0.     ],
        [-0.     ,  0.     ,  0.     ,  0.     ,  0.95168]],

       [[ 0.06256,  0.     , -0.01218, -0.     ,  0.     ],
        [ 0.     ,  0.481  ,  0.     , -0.45877,  0.     ],
        [-0.01218,  0.     ,  0.07148, -0.     ,  0.     ],
        [-0.     , -0.45877, -0.     ,  0.481  ,  0.     ],
        [ 0.     ,  0.     ,  0.     ,  0.     ,  0.94038]]])
```

Each correlation matrix contains 25 values, using the procedure above, we reduce this number to 20: 10 euler angles, 5 integer occupations and 5 deltas. The values of deltas can be ignored for the purpose of the global searcher and the occupations are contrained by the condition that their sum must be the equal to the number of electrons in the correlated orbital. We have now the ingredients to move into the next step, create a population of correlation matrices.

4.6.2 The population

The most simple way of creating the population requires just the name of the mongo database to be created and one abinit input file. The relevant information to setup the search will be infered from the contents of the abinit input file:

```
popu=pychemia.population.orbitaldftu.OrbitalDFTU('test', abinit_input=pychemia_path +
                                                     '/test/data/abinit_dmatpawu/
↪abinit.in')

Orbital population:
Species [znucl]: [19, 27, 9]
Orbitals corrected:
 19 : False
 27 : True (l=2)
  9 : False
Number of atoms where DFT+U is applied: 4
Correlation of 'd' orbitals
Variables controling the total number of matrices
nsppol : 1
nspinor: 1
nspden : 2
Total number of matrices expected on dmatpawu: 4
Number of electrons for each correlation matrix: [2 5 5 2]
Number of independent matrices: 4
```

Create random correlation matrices can be done with:

```
popu.add_random()
```

The return is the Indentifier of the new entry on the database. Also a set of new random correlation matrices can be created with:

```
popu.random_population(16)
```

We have the basic ingredients for creating the first population for the global searcher. How the correlation matrices are evaluated is out of scope of the population and depends on the particularities of the machines where Abinit is used to evaluate them. We will move our focus to the methods needed to produced new correlation matrices based on the results of a given set of correlation matrices.

CHAPTER 5

API Documentation

5.1 pychemia package

5.1.1 Subpackages

5.1.1.1 pychemia.analysis package

Submodules

pychemia.analysis.analysis module

pychemia.analysis.changer module

pychemia.analysis.cluster module

pychemia.analysis.matching module

pychemia.analysis.splitting module

pychemia.analysis.surface module

5.1.1.2 pychemia.code package

Subpackages

pychemia.code.abinit package

Subpackages

pychemia.code.abinit.task package

Submodules

pychemia.code.abinit.task.relax module

CHAPTER 6

Contributors

1. Prof. Aldo H. Romero [West Virginia University] (Project Director)
2. Guillermo Avendaño-Franco [West Virginia University] (Basic Infraestructure)
3. Adam Payne [West Virginia University] (Bug Fixes)
4. Irais Valencia Jaime [West Virginia University] (Simulation and testing)
5. Sobhit Singh [West Virginia University] (Data-mining)
6. Francisco Muñoz (Universidad de Chile) (pyprocar)

CHAPTER 7

Indices and tables

- genindex
- modindex
- search