# CellConstructor Documentation

***Release 1.0***

**Lorenzo Monacelli**

**Jun 22, 2021**

# CONTENTS:

CHAPTER

# ONE

# INTRODUCTION

## 1.1 What is CellConstructor?

The CellConstructor is a python module originally created to manipulate atomic structures for *ab-initio* simulations.

It can be used in interactive mode, or through python scripting, and can be interfaced with other libraries like ASE (Atomic Simulation Environment) or spglib for symmetries.

The library is constituted of three main modules: 1. `Structures`: The module class that allows the manipulation of the atomic structure, including constraining symmetries, apply deformations on the cell, extracting molecular information (averaging bond lengths). 2. `Phonons`: The module that allows the manipulation of the harmonic dynamical matrix. It includes utilities like the Fourier transform, generating randomly distributed configurations, thermodynamic properties, computing frequencies and phonon modes, Raman and IR responce, as well as interpolating in bigger cells. 3. `symmetries`: The module that allows the symmetry search and constraining. It allows the symmetrization of vectors, matrices and 3 or 4 rank tensors. It has its own builtin symmetry engine, but `spglib` can be used. It can export also input files for `ISOTROPY` symmetry analysis program.

## 1.2 Requirements

The CellConstructor can be installed with `distutils`. First of all, make sure to satisfy the requirements

1. python >= 2.7 and < 3
2. ASE : Atomic Simulation Environment (suggested but not mandatory)
3. numpy
4. scipy
5. A fortran compiler
6. Lapack

The fortran compiler is required to compile the fortran libraries from Quantum ESPRESSO.

Suggested, but not required, is the installation of ASE and spglib. The presence of a valid ASE installation will enable some more features, like the possibility to load structures by any ASE supported file format, or the possibility to export the structures into a valid ASE Atoms class. This library is able to compute symmetries from the structure, and inside the symmetry module there is a convertor to let CellConstructure dealing with symmetries extracted with spglib. However, for a more carefull symmetry analisys, we suggest the use of external tools like ISOTROPY. This package can generate ISOTROPY input files for more advanced symmetry detection.

Please, note that some fortran libraries are needed to be compiled, therefore the Python header files should be localized by the compiling process. This requires the python distutils and developing tools to be properly installed. On ubuntu this can be achieved by running:

If you are using anaconda or pip, it should be automatically installed.

## 1.3 Installation

Once you make sure to have all the required packages installed on your system and working, just type on the terminal

while you are located in the same directory as the setup.py script is.

This program is also distributed upon PyPI. You can install it by typing

In this way you will not install the last developing version.

If the compilation of the modules fails and you are using an anaconda module on a 64bit machine, you have to install the conda gcc version. You can do this by typing (on Linux):

or (on MacOS):

NOTE: If you want to install the package into a system python distribution, the installation commands should be executed as a superuser. Otherwise, append the –user flag to either the setup.py or the pip installation. In this way no administrator privilages is required, but the installation will be effective only for the current user. Note that some python distribution, like anaconda, does not need the superuser, as they have an installation path inside the HOME directory.

You can install also using the intel compiler. In this case, you must edit the setup.py script so that: - remove the lapack and blas as extra library for the SCHAModules extension. - add a new flag: 'extra_link_args = ["-mkl"]' to the extension.

Remember to specify the intel compiler both to the compilation and for the running: CC="icc" LDSHARED="icc -shared" otherwise the C module will give an error when loaded reguarding some "_fast_memcpy_" linking.

### 1.3.1 Test the installation

You can run the testsuite to test your installation as

The execution of the test suite can require some time. If everything is OK, then the softwere is correctly installed and working.

CHAPTER

TWO

# GETTING STARTED

Now that we have correctly installed the CellConstructor, we can start testing the features.

The source code comes with a series of test and examples available under the `tests` directory of the git source.

As a first example, we will load a dynamical matrix in the Quantum ESPRESSO PHonon output style, compute the eigenvalues and print them into the screen.

First of all, lets copy the file `tests/TestPhononSupercell/dynmat1` inside the working directory:

```
import cellconstructor as CC
import cellconstructor.Phonons
import cellconstructor.Units

# Load the dynamical matrix
dyn = CC.Phonons.Phonons("dynmat")

# Compute the polarization vector and the phonon frequencies
w_freq, pols = dyn.DyagonalizeSupercell()

# Print the frequencies on the screen
print("\n".join(["{:16.4f} cm-1".format(x * CC.Units.RY_TO_CM) for x in w_freq]))
```

If we run this script using python, it will print for each line the frequency in cm-1 of each phonon mode at gamma.

Note that we open the dynamical matrix dynmat1 with the line:

```
dyn = CC.Phonons.Phonons("dynmat")
```

The current dynamical matrix is a (3,3,2) supercell of ice XI. Quantum ESPRESSO stores the matrix in fourier space, as it is block diagonal for the Block theorem (and occupies less space). Each blocks is a different q-point in the Brillouin zone. In total there are 3*3*2 = 18 different q-points. However, some q points are linked by symmetries. In this particular case, only 8 q-points are indepenent. This is why we have 8 files dynmat1 … dynmat8.

To load all of them, we need to specify the keyword `nqirr` to the loading line:

```
dyn = CC.Phonons.Phonons("dynmat", nqirr = 8)
# Or alternatively
# dyn = CC.Phonons.Phonons("dynmat", 8)
```

In our first example, we loaded only the first file, dynmat1, as the default value of nqirr is 1.

Given the q-points, we are able to compute the supercell. For example, try to run the code:

```
>>> dyn = CC.Phonons.Phonons("dynmat")
>>> print (dyn.GetSupercell())
[1,1,1]
```

(continues on next page)

(continued from previous page)

```
>>> dyn = CC.Phonons.Phonons("dynmat", 8)
>>> print (dyn.GetSupercell())
[3,3,2]
```

As you can see, CellConstructor correctly recognized the supercell mesh once after reading all the 8 irreducible q points. If you try to print again the frequencies when you load all the 8 irreducible q-points, you will have much more modes, that are all the possible modes of the supercell.

However, there is a way to print only the modes of a particular q point:

```
q_index = 0
w_freq, pols = dyn.DyagDinQ(q_index)
```

Here, `q_index` is the index of the q-point that you want to diagonalize. Usually `q_index = 0` corresponds to the Gamma point. To print a list of all the q-points:

```
print("\n".join("{:4d})  {:16.4f} {:16.4f} {:16.4f}".format(iq, *list(q)) for iq, q␣
↪in enumerate(dyn.q_tot)))
```

This will print on the screen the q points preceeded by the respective index. Note that the number should be always equal to the number of cells.

CHAPTER

THREE

# TUTORIALS

In this section, I will show how simple tasks can be performed using CellConstructor.

## 3.1 Extract an harmonic random ensemble

You can use CellConstructor to extract an harmonic esnemble. It can be used for measuring electron-phonon properties, as you can average dielectric function on this ensemble to get phonon mediated electronic properties.

To get a list of structures distributed according to a dynamical matrix use the following commands:

```
import cellconstructor as CC
import cellconstructor.Phonons

# We read the dynmat1 file (quantum espresso gamma dynamical matrix)
dyn = CC.Phonons.Phonons("dynmat")

structures = dyn.ExtractRandomStructures(size = 10, T = 100)
```

Then, `structures` is a list of `CC.Structures.Structures`. We have extracted 10 structures at a temperature of 100 K. You can save them in the .scf format, that is ready to be copied on the bottom of a quantum espresso pw.x input file for a calculation of the band structure for example:

```
for i, struct in enumerate(structures):
    struct.save_scf("random_struct{:05d}.scf".format(i))
```

In this way, we will create 10 files named *random_struct00000.scf*, *random_struct00001.scf*, . . .

Each of them will be ready to be used in a quantum espresso pw.x calculation with ibrav=0.

Alternatively, one can convert them into the ASE atoms object. In this way any configured calculator can be used directly inside python:

```
for i, struct in enumerate(structures):
    ase_atmoms = struct.get_ase_atoms()

    # Here the code to perform the ab-initio calculation
    # with ase
```

Note that we used a gamma dynamical matrix. To generate a random structure in a supercell you need to first generate a supercell real space dynamical matrix. This is covered in the next tutorial.

If you are using the python-sscha package, it has a class Ensemble that can automatically generate and store the ensemble using this function.

For more details on the structure generation, I remand to the specific documentation:

**class** Phonons.**Phonons**(*structure=None, nqirr=1, full_name=False, use_format=False, force_real=False*)

This class contains the phonon of a given structure. It can be used to show and display dinamical matrices, as well as for operating with them

**ExtractRandomStructures**(*size=1, T=0, isolate_atoms=[], project_on_vectors=None, lock_low_w=False*)

This method is used to extract a pool of random structures according to the current dinamical matrix.

**Parameters**

- **size** (*int*) – The number of structures to be generated
- **T** (*float*) – The temperature for the generation of the ensemble
- **isolate_atoms** (*list, optional*) – A list of the atom index. Only the specified atoms are present in the output structure and displaced. This is very usefull if you want to measure properties of a particular region of the structure. By default all the atoms are used.
- **lock_low_w** (*bool*) – If True, frequencies below __EPSILON_W__ are fixed.

**Returns** A list of Structure.Structure()

**Return type** list

## 3.2 Generate a real space supercell dynamical matrix

Many operations are possible only when the dynamical matrix is expressed in real space supercell, as the extraction of the random ensemble.

So it is crucial to be able to define a dynamical matrix in the supercell. Luckily, it is very easy:

```python
import cellconstructor as CC
import cellconstructor.Phonons

# We load the dynmat1 ... dynmat8 files
# They are located inside tests/TestPhononSupercell
dyn = CC.Phonons.Phonons("dynmat", nqirr = 8)

super_dyn = dyn.GenerateSupercellDyn(dyn.GetSupercell())

# Now we can do what we wont with the super_dyn variable.
# For example we can extract a random ensemble in the supercell

structures = super_dyn.ExtractRandomStructures(size = 10, T = 100)
```

Also in this case, please refer to the official documentation

**class** Phonons.**Phonons**(*structure=None, nqirr=1, full_name=False, use_format=False, force_real=False*)

This class contains the phonon of a given structure. It can be used to show and display dinamical matrices, as well as for operating with them

**GenerateSupercellDyn**(*supercell_size, img_thr=1e-06*)

This method returns a Phonon structure as it was computed directly in the supercell.

NOTE: For now this neglects bohr effective charges

**Parameters** **supercell_size** (*array int (size=3)*) – the dimension of the cell on which you want to generate the new Phonon

**dyn_supercell** [Phonons()] A Phonons class of the supercell

## 3.3 Force a dynamical matrix to be positive definite

This is an important task if you want to use the dynamical matrix to extract random configurations. Often, harmonic calculation leads to imaginary frequencies. This may happen if:

1. The calculation is not well converged.
2. The dynamical matrix comes from interpolation of a small grid
3. The structure is in a saddle point of the Born-Oppenheimer energy landscape.

In all these cases, to generate a good dynamical matrix for extracting randomly distributed configurations we need to define a dynamical matrix that is positive definite.

A good way to do that is to redefine our matrix as

$$\Phi'_{ab} = \sqrt{m_a m_b}\sum_\mu e_\mu^a e_\mu^b \left|\omega_\mu^2\right|$$

where $\omega_\mu^2$ and $e_\mu$ are, respectively, eigenvalues and eigenvectors of the original $\Phi$ matrix.

This can be achieved with just one line of code:

```
dyn.ForcePositiveDefinite()
```

In the following tutorial, we create a random structure, associate to it a random dynamical matrix, print the frequencies of this dynamical matrix on the screen, force it to be positive definite, and the print the frequencies again.

```
import cellconstructor as CC
import cellconstructor.Structure
import cellconstructor.Phonons
import numpy as np

# We create a structure with 6 atoms
struct = CC.Structure.Structure(6)

# By default the structure has 6 Hydrogen atoms
# struct.atoms = ["H"] * 6

# We must setup the masses (in Ha)
struct.masses = {"H" : 938}

# We extract the 3 cartesian coordinates
# Randomly for all the coordinates
struct.coords = np.random.uniform(size = (6, 3))

# We set a cubic unit cell
# eye makes the identity matrix, 3 is the size (3x3)
struct.unit_cell = np.eye(3)
struct.has_unit_cell = True # Set the periodic boundary conditions

# Now we create a Phonons class with the structure
dyn = CC.Phonons.Phonons(struct)

# We fill the gamma dynamical matrix with a random matrix
dyn.dynmats[0] = np.random.uniform(size = (3*struct.N_atoms, 3*struct.N_atoms))
```

(continues on next page)

(continued from previous page)

```
# We impose the matrix to be hermitian
dyn.dynmats[0] += dyn.dynmats[0].T

# We print all the frequencies of the dynamical matrices
freqs, pols = dyn.DyagonalizeSupercell()
print("\n".join(["{:3d}) {:10.4f} cm-1".format(i+1, w * CC.Units.RY_TO_CM) for i, w␣
↪in enumerate(freqs)]))

# Now we Impose the dynamical matrix to be positive definite
dyn.ForcePositiveDefinite()

# Now we can print again the frequencies
print("")
print("After the  force to positive definite")
freqs, pols = dyn.DyagonalizeSupercell()
print("\n".join(["{:3d}) {:10.4f} cm-1".format(i+1, w * CC.Units.RY_TO_CM) for i, w␣
↪in enumerate(freqs)]))
```

The output of this code is shown below. Note, since we randomly initialized the dynamical matrix, the absolute value of the frequencies may change.

```
  1) -5800.6476 cm-1
  2) -5690.3146 cm-1
  3) -4521.1801 cm-1
  4) -4373.4443 cm-1
  5) -4184.2736 cm-1
  6) -4028.2141 cm-1
  7) -3220.6904 cm-1
  8) -2043.1963 cm-1
  9) -1073.2832 cm-1
 10)  2216.9358 cm-1
 11)  2952.8991 cm-1
 12)  3478.0255 cm-1
 13)  3801.1767 cm-1
 14)  4805.3197 cm-1
 15)  5073.9250 cm-1
 16)  5524.8080 cm-1
 17)  6162.4137 cm-1
 18) 14400.8452 cm-1

After the  force to positive definite
  1)  1073.2832 cm-1
  2)  2043.1963 cm-1
  3)  2216.9358 cm-1
  4)  2952.8991 cm-1
  5)  3220.6904 cm-1
  6)  3478.0255 cm-1
  7)  3801.1767 cm-1
  8)  4028.2141 cm-1
  9)  4184.2736 cm-1
  10)  4373.4443 cm-1
  11)  4521.1801 cm-1
  12)  4805.3197 cm-1
  13)  5073.9250 cm-1
  14)  5524.8080 cm-1
```

(continues on next page)

(continued from previous page)

```
  15)  5690.3146 cm-1
  16)  5800.6476 cm-1
  17)  6162.4137 cm-1
  18) 14400.8452 cm-1
```

In this tutorial we saw also how to create a random dynamical matrix. Note, the negative frequencies are imaginary: they are the square root of the negative eigenvalues of the dynamical matrix.

You may notice that this dynamical matrix at gamma does not satisfy the acoustic sum rule: it should have 3 eigenvalues of 0 frequency. We will see in the next tutorials how to enforce all the symmetries and the acoustic sum rule on the dynamical matrix

## 3.4 Symmetries of the structure

It is very common to have a structure that violates slightly some symmetry. It can come from a relaxation that is affected from some noise, from experimental data.

Here, we will see how to use spglib and CellConstructor to enforce the symmetries on the structure. We will create a BCC structure, then we will add to the atoms a small random displacements so that spglib does not recognize any more the original group. Then we will enforce the symmetrization with CellConstructor.

This tutorial requires spglib and the ASE packages installed.

```
from __future__ import print_function
from __future__ import division
import cellconstructor as CC
import cellconstructor.Structure

import spglib

# We create a structure of two atoms
struct = CC.Structure.Structure(2)

# We fix an atom in the origin, the other in the center
struct.coords[0,:] = [0,0,0]
struct.coords[1,:] = [0.5, 0.5, 0.5]

# We add an unit cell equal to the identity matrix (a cubic structure with :math:`a =␣
↪1`)
struct.unit_cell = np.eye(3)
struct.has_unit_cell = True # periodic boundary conditions on

# Lets see the symmetries that prints spglib
print("Original space group: ", spglib.get_spacegroup(struct.get_ase_atoms()))

# The previous command should print
# Original space group: Im-3m (299)

# Lets store the symmetries and convert from spglib to the CellConstructor
syms = spglib.get_symmetry(struct.get_ase_atoms())
cc_syms = CC.symmetries.GetSymmetriesFromSPGLIB(syms)

# We can add a random noise on the atoms
struct.coords += np.random.normal(0, 0.01, size = (2, 3))
```

(continues on next page)

(continued from previous page)

```
# Let us print again the symmetry group
print("Space group with noise: ", spglib.get_spacegroup(struct.get_ase_atoms()))

# This time the code will print
# Space group with noise: P-1 (2)
#
# This means that the structure lost all the symmetries

# Now we enforce the symmetrization on the structure
struct.impose_symmetries(cc_syms)

# The previous command will print details on the symmetrization iterations
print("Final group: ", spglib.get_spacegroup(struct.get_ase_atoms()))
# Now the structure will be again in the Im-3m group.
```

You can pass to all spglib commands a threshold for the symmetrization. In this case you can also use a large threshold and get the symmetries of the closest larger space group. You can use them to constrain the symmetries.

Note, in some cases the symmetrization does not converge. If this happens, then the symmetries cannot be enforced on the structure. It could also be possible that spglib identifies some symmetries with a threshold, the code impose them, but then the symmetries are still not recognized by spglib with a lower threshold. This happens when the symmetries you are imposing are not satisfied by the unit cell. In that case, you have to manually edit the unit cell before imposing the symmetries.

## 3.5 Symmetries of the dynamical matrix

In this tutorial we will create a random dynamical matrix of a high symmetry structure and enforce the symmetrization. Constraining symmetries on the dynamical matrix can be achieved in two ways. Firstly, we will use the builtin symmetry engine from QuantumESPRESSO, then we will use the spglib.

The builtin Quantum ESPRESSO module performs the symmetrization in q space, it is much faster than spglib in large supercells. Moreover, it is installed together with CellConstructor and no additional package is required. However, spglib is much better in finding symmetries; it is a good tool to be used when the structure is already in a supercell. The CellConstructor module interfaces with spglib symmetry identification and performs the symmetrization of the dynamical matrix.

```
from __future__ import print_function

# Import numpy
import numpy as np

# Import cellconstructor
import cellconstructor as CC
import cellconstructor.Structure
import cellconstructor.Phonons
import cellconstructor.symmetries

# Define a rocksalt structure
bcc = CC.Structure.Structure(2)
bcc.coords[1,:] = 5.6402 * np.array([.5, .5, .5]) # Shift the second atom in the␣
↪center
bcc.atoms = ["Na", "Cl"]
bcc.unit_cell = np.eye(3) * 5.6402 # A cubic cell of 5.64 A edge
bcc.has_unit_cell = True # Setup periodic boundary conditions
```

(continues on next page)

(continued from previous page)

```
# Setup the mass on the two atoms (Ry units)
bcc.masses = {"Na": 20953.89349715178,
"Cl": 302313.43272048925}



# Lets generate the random dynamical matrix
dynamical_matrix = CC.Phonons.Phonons(bcc)
dynamical_matrix.dynmats[0] = np.random.uniform(size = (3 * bcc.N_atoms,
3* bcc.N_atoms))

# Force the random matrix to be hermitian (so we can diagonalize it)
dynamical_matrix.dynmats[0] += dynamical_matrix.dynmats[0].T

# Lets compute the phonon frequencies without symmetries
w, pols = dynamical_matrix.DiagonalizeSupercell()

# Print on the screen the random frequencies
print("Non symmetric frequencies:")
print("\n".join(["{:d}) {:.4f} cm-1".format(i, w * CC.Units.RY_TO_CM)for i,w in␣
↪enumerate(w)]))

# Symmetrize the dynamical matrix
dynamical_matrix.Symmetrize() # Use QE to symmetrize

# Recompute the frequencies and print them in output
w, pols = dynamical_matrix.DiagonalizeSupercell()
print()
print("frequencies after the symmetrization:")
print("\n".join(["{:d}) {:.4f} cm-1".format(i, w * CC.Units.RY_TO_CM) for i,w in␣
↪enumerate(w)]))
```

In this tutorial we first build the NaCl structure, then we generate a random force constant matrix. After the symmetrization, the system will have 3 frequencies to zero (the acoustic modes at gamma) and 3 identical frequencies (negative or positive). If you want to use this dynamical matrix, it is recommanded to force it to be positive definite with the command dynamical_matrix.ForcePositiveDefinite().

The only command actually required to symmetrize is:

```
dynamical_matrix.Symmetrize() # Use QE to symmetrize
```

This command will always use the espresso subroutines. You can, however, setup the symmetries from SPGLIB and force the symmetrization in the supercell. This procedure is a bit more involved, as you need to create and initialize manually the symmetry class.

The code to perform the whole symmetrization in spglib is

```
# Initialize the symmetry class
syms = CC.symmetries.QE_Symmetry(bcc)
syms.SetupFromSPGLIB() # Setup the espresso symmetries on spglib

# Generate the real space dynamical matrix
superdyn = dynamical_matrix.GenerateSupercellDyn(dynamical_matrix.GetSupercell())

# Apply the symmetries to the real space matrix
CC.symmetries.CustomASR(superdyn.dynmats[0])
```

(continues on next page)

(continued from previous page)

```
syms.ApplySymmetriesToV2(superdyn.dynmats[0])

# Get back the dyanmical matrix in q space
dynq = CC.Phonons.GetDynQFromFCSupercell(superdyn.dynmats[0], np.array(dynamical_
↪matrix.q_tot), dynamical_matrix.structure, superdyn.structure)

# Copy each q point of the symmetrized dynamical matrix into
# the original one
for i in range(len(dynamical_matrix.q_tot)):
       dynamical_matrix.dynmats[i] = dynq[i,:,:]
```

The symmetrization here occurs in real space, therefore it is necessary in principle to transform the matrix in real space. In this case it is not strictly necessary, as we have only one q-point at Gamma, therefore, the dynamical_matrix.dynmats[0] can be directly passed to the symmetrization subroutines, however, this is not true when more q points are present.

In this part we also showed how to explicitly perform a fourier transformation between real spaces dynamical matrices and q space quantities using the subroutine GetDynQFromFCSupercell.

## 3.6 Load and Save structures

Cellconstructor supports many standard structure files. For the structure, the basic format is the ‘scf’. It is a format where both the unit cell and the cartesian position of all the atoms are explicitly given. It is compatible with quantum espresso, therefore it can be appended on a espresso input file to perform a calculation on the given structure.

However, Cellconstructor can be interfaced with other libraries like ASE (Atomic Simulation Environment). ASE provide I/O facilities for all most common DFT and MD programs, as well as the ability to read and write in many format, including ‘cif’, ‘pdb’, ‘xyz’ and so on.

To read and write in the native ‘scf’ format, use the following code:

```
struct = CC.Structure.Structure()
struct.read_scf("myfile.scf")

# --- some operation ----
struct.save_scf("new_file.scf")
```

You can read all the file format supported by ASE using the read_generic_file function, however, you must have ASE installed

```
struct = CC.Structure.Structure()
struct.read_generic_file("myfile.cif")
```

You can also directly convert an ASE Atoms into the CellConstructor Structure, and vice-versa

```
import ase
import cellconstructor as CC
import cellconstructor.Structure

# Generate the N2 molecule using ASE
N2_mol = ase.Atoms("2N", [(0,0,0), (0,0, 1.1)])

struct = CC.Structure.Structure()
struct.generate_from_ase_atoms(N2_mol)
struct.save_scf("N2.scf")
```

Vice-versa, you can generate an ASE Atoms structure using the function get_ase_atoms() of the Structure class

## 3.7 Load and save the dynamical matrix

CellConstructor is primarly ment for phonon calculations. The main interface of the Phonon object is with quantum espresso.

We can read a quantum espresso dynamical matrix easily.

```
import cellconstructor as CC
import cellconstructor.Phonons

# Load the dynamical matrix in the quantum espresso format
# ph.x of quantum espresso splits the output of the
# dynamical matrices dividing per irreducible q points of the Brilluin zone.
# This command will load a dynamical matrix with 3 irreducible q points.
# The files are dyn1, dyn2 and dyn3
dyn = CC.Phonons.Phonons("dyn", 3)

# -- do something --

# Now we save the dynamical matrix in the espresso format
dyn.save_qe("new_dyn")
```

When loading from quantum espresso, CellConstructor will also import raman_tensor, dielectric tensor and effective charges. The structure is read from the first dynamical matrix (usually the gamma point). An experimental interface to phonopy is under developement.

## 3.8 Generate a video of phonon vibrations

It could be very usefull to generate a video of a phonon mode, for post-processing reasons.

In this tutorial I will introduce the Manipulate module, that can be used for post-processing analysis.

In the following simple example, we will read the dynamical matrix ice_dyn1. The methods that makes the video is GenerateXYZVideoOfVibrations. It is quite self-explaining: it needs the dynamical matrix, the name of the file in witch to save the video, the index of the vibrational mode.

You also need info on the video, the vibrational amplitude (in angstrom) the actual time step (in femtoseconds) and the number of time steps to be included.

Take in consideration that a vibration of 800 cm-1 has a period of about 40 fs. In this case we are seeing a vibration of 3200 cm-1, whose period is about 10 fs. Therefore we pick a dt = 0.5 fs to correctly sample the vibration and a total time of 50 fs (100 steps). In this way we will have about 5 full oscillations.

```
from __future__ import print_function

import cellconstructor as CC
import cellconstructor.Phonons
import cellconstructor.Manipulate


# Load a dynamical matrix that represent an ice structure (at gamma)
dyn = CC.Phonons.Phonons("ice_dyn")

```

(continues on next page)

(continued from previous page)

```
# We dyagonalize the dynamical matrix
w, p = dyn.DiagonalizeSupercell()

# We pick the hardest mode
mode_id = len(w) - 1

# We must specify the amplitude of the vibrations (in A)
amplitude = 0.8 #A

# The time steps between two frams (in Femtoseconds)
dt = 0.5

# The total number of time steps
N_t = 100


# Save the video of the trajectory in a xyz file.
CC.Manipulate.GenerateXYZVideoOfVibrations(dyn, "vibration.xyz",  mode_id, amplitude,␣
↪dt, N_t)
```

This code will save the video as 'vibration.xyz'. You can load this file in your favorite viewer. If you have ASE installed, you can view the video just typing in the console

```
ase gui vibration.xyz
```

Here you will find more details about this API.

This module contains the methods that requre to call the classes defined into this module.

`Manipulate.`**`GenerateXYZVideoOfVibrations`**(*dynmat*, *filename*, *mode_id*, *amplitude*, *dt*, *N_t*, *supercell=(1, 1, 1)*)

This function save in the filename the XYZ video of the vibration along the chosen mode.

NOTE: this functionality is supported only at gamma.

**Parameters**

- **filename** (*str*) – Path of the filename in which you want to save the video. It is written in the xyz format, so it is recommanded to use the .xyz extension.
- **mode_id** (*int*) – The number of the mode. Modes are numbered by their frequencies increasing, starting from imaginary (unstable) ones (if any).
- **amplitude** (*float*) – The amplitude in Angstrom of the vibration per atom.
- **dt** (*float*) – The time step between two different steps in femtoseconds.
- **N_t** (*int*) – The total number of frames.
- **supercell** (*list of 3 ints*) – The dimension of the supercell to be shown

## 3.9 Radial pair distribution function

A very usefull quantities, directly related to the static structure factor, is the pair radial distribution function, defined as

$$g_{AB}(r) = \frac{\rho_{AB}^{(2)}(r)}{\rho_A(r)\rho_B(r)}$$

This quantity probes the how many couples of the AB atoms are inside a shell of distance $r$ with respect to what we would expect in a non interacting system.

It is a standard quantity for liquid simulation.

In this tutorial we will take an harmonic dynamical matrix, generate the harmonic ensemble, and compute the g(r) on it.

```
from __future__ import print_function
import cellconstructor as CC
import cellconstructor.Phonons
import cellconstructor.Methods
import matplotlib.pyplot as plt


# Some info on the g(r)
ENSEMBLE_SIZE=10000
T = 0 # temperature in K

# The maximum distances for computing the g(r)
R_MIN = 0.9
R_MAX_OH = 2
R_MAX_HH = 3

# The number of bins to divide the interval
N_bins = 1000

# The limits for the final plot
R_LIM_OH= (0.75, 2)
R_LIM_HH=(1, 3)


# Load the ice XI dynamical matrix
iceXI_dyn = CC.Phonons.Phonons("h2o.dyn", full_name = True)
iceXI_dyn.Symmetrize() # Impose the sum rule

# Use the dynamical matrix to generate the displacements
print ("Generating displacements...")
structures = iceXI_dyn.ExtractRandomStructures(ENSEMBLE_SIZE, T)

# Get the g(r) between O and H atoms
print ("Computing OH g(r)...")
grOH = CC.Methods.get_gr(structures, "O", "H", R_MIN, R_MAX_OH, N_bins)
print ("Computing HH g(r)...")
grHH = CC.Methods.get_gr(structures, "H", "H", R_MIN, R_MAX_HH, N_bins)

# Plot the result
plt.plot(grOH[:,0], grOH[:,1])
plt.xlabel("r [$\\AA$]")
plt.ylabel("$g_{OH}(r)$")
```

(continues on next page)

(continued from previous page)

```
plt.title("O-H radial distribution function")
plt.xlim(R_LIM_OH)
plt.tight_layout()

plt.figure()

plt.plot(grHH[:,0], grHH[:,1])
plt.xlabel("r [$\\AA$]")
plt.ylabel("$g_{HH}(r)$")
plt.title("H-H radial distribution function")
plt.xlim(R_LIM_HH)
plt.tight_layout()
plt.show()
```

In this example I use a ice XI harmonic dynamical matrix at $\Gamma$ computed with quantum espresso and the PBE exchange correlation functional. You can find this file inside the tutorials/RadialDistributionFunction, or you can replace it with your own dynamical matrix, to test it for your case. You can decide your temperature, to test temperture effects.

The example is quite self-explaining, it will produce two figures one for HH distance and one for the OH.

Indeed, if you are using molecular dynamics, you can load your array of structures and pass it through the get_gr functions of the Methods module.

A detailed documentation of this method is available here:

Created on Wed Jun 6 10:45:50 2018

@author: pione

`Methods.`**`get_gr`**(*structures*, *type1*, *type2*, *r_min*, *r_max*, *N_r*)

Computes the radial distribution function for the system. The $g_{AB}(r)$ is defined as

$$g_{AB}(r) = \frac{\rho^{(2)}_{AB}(r)}{\rho_A(r)\rho_B(r)}$$

where $A$ and $B$ are two different types

**Parameters**

- **`structures`** (–) – A list of atomic structures on which compute the $g_{AB}(r)$
- **`type1`** (–) – The character specifying the $A$ atomic type.
- **`type2`** (–) – The character specifying the $B$ atomic type.
- **`r_min`** (–) – The minimum and maximum cutoff value of $r$
- **`r_max`** (*`float`*) – The minimum and maximum cutoff value of $r$
- **`N_r`** (–) – The number of bins for the distributions

- **g_r** [ndarray.shape() = (r/dr + 1, 2)] The $g(r)$ distribution, in the first column the $r$ value in the second column the corresponding value of g(r)

CHAPTER

FOUR

# DEVELOPER'S GUIDE

In this chapter, I will introduce you to the code development.

## 4.1 The code management

CellConstructor is distributed through the GIT control version system.

### 4.1.1 Get the last development code

To download the last development distribution, use the command:

```
$ git clone https://github.com/mesonepigreco/CellConstructor.git
```

In this way a new directory CellConstructor will be created. After you created the directory, you can upload your local repository with the last changes using the pull command.

```
$ git pull
```

The pull command must be executed inside the CellConstructor directory.

### 4.1.2 What should I do BEFORE modify the code?

Note, the master branch of CellConstructor is holy: you should push your changes inside this branch only after you are absolutely sure that they will not brake any function. Moreover, unless you are an official developer of CellConstructor, you will not have the permission to push your commits inside the official repository.

Then, how can you make your own changes available to other developers? The answer is **forking**.

Fork the CellConstructor repository, in this way you will create a new repository, synced with the original one, where you are the owner.

You can do all the changes, commit and push them inside this repository. When you think that everything is ok and ready to be merged inside the main repository, you can ask for a **pull request**.

CellConstructor uses GitHub (this may change in future). A detailed guide on how to manage forks and pull requests on the github web site, please see:

https://help.github.com/en/github/getting-started-with-github/fork-a-repo

### 4.1.3 How do I report a bug?

If you spot a bug, please open an issue on our GitHub page

https://github.com/mesonepigreco/CellConstructor

Before opening a new post, please, look if someone already spotted your same error (and maybe managed to find a workaround). If not, open a new issue.

Describe in detail the problem and the calculation you are trying to run. Please, **include an executable script with data** that triggers the bug.

We developers are not working 24h to answer issues, however, we will do our best to give you an answer as soon as we can.

## 4.2 Coding guidelines

The CellConstructor module is written in mixed python, C and Fortran. Python is a glue between the Fortran and C parts. If you want to add a new utility to the code, consider in writing it directly in python, as interfacing between Fortran or C code could be very difficult.

In particular, when coding in python, keep in mind the following rules.

1. Keep the code compatible between Python 2 and Python 3. Always begin the source files with

   ```
   from __future__ import print_function
   from __future__ import division
   ```

   and keep in mind to use a syntax that does not complain if executed by both python2 and python3 interprets.

2. Write a docstring for each function and classes implemented. Remember to use the numpy syntax for the docstring. In this way, the function can be automatically documented and included in the API reference guide. An example of numpy syntax docstrign for the my_new_function(x)

   ```
   def my_new_function(x):
       """
       My New Function
       ===============

       This function takes a x parameter and returns its square.

       .. math ::

           y = x^2

       The previous equation will become LaTeX code in the API.

       Parameters
       ----------
           x : float
               The input variable
       Return
       ------
           y : float
               The output (x*x)
       """

       return x*x
   ```

This is an example of well documented code. Doing this each new function will simplify the life of the other developers, and also yourself when you will return to use this function some week after you wrote it. Please, remember to specify the types and dimensions of input arrays, as well as output. It is very upsetting having to look to the source to know what to pass to the function.

3. Name classes with capital letters for new words, for example:

```
class MyNewClass(x):
    # Very good
```

   Name functions without capital letters, using underscores, for example:

```
def my_function(x):
    pass
```

   NOTE: The code not always follows this convention, however, this is not a good excuse to avoid doing that yourself ;-)

4. Always use self-explaining names for the input variables. Avoid naming variables like `pluto` or `goofy`
5. Comment each step of the algorithm. Always state what the algorithm does with comments, do not expect other people to understand what is in your mind when coding.
6. Avoid copy&paste. If you have to do an action twice, use a for loop, or define a function, do not copy&paste the code. If your code needs to be edited, it can be a pain to track all the positions of your copy&paste. Moreover, the smaller the code, the better.
7. Use numpy for math operation. Numpy is the numerical scientific library, it includes all the math libraries, linear algebra, fft, and so on. It can be compiled with many frontends, like Lapack, MKL. It is much faster than the native python math library.
8. Avoid unnecessary loop. Python is particularly slow when dealing with for loops. However, in math, most loop can be replaced by summation, products, and so on. Use the numpy functions sum, einsum, prod, dot, and so on to perform mathematical loop. Remember, a numpy sum call can be 1000 times faster than an explicit for loop to do a summation of an array.
9. Use exception handling and assertion. When you write a new function, do not expect the user to provide exactly the right data. If you need an input array of 3x4 elements, check it ussing the assert command:

```
def MyFunction(x_array):

    # Check that x_array is a 2 rank tensor
    assert len(x_array.shape) == 2

    # Check the shape of the x_array
    assert x_array.shape[0] == 3
    assert x_array.shape[1] == 4
```

   Raise exceptions when the input is wrong:

```
def sqrt(x):

    if x < 0:
        raise ValueError("Error, x is lower than 0")
```

10. Always write a test inside the unittest suite to reproduce a known result. In this way, if bugs are introduced in future, they will be spotted immediately. To this purpose, see the next section.

### 4.2.1 Adding tests

It is very important that each part of the code can be tested automatically each time a new feature is implemented. The cellconstructor tests are based on unittest for now. There is a script inside *scripts/cellconstructor_test.py*

Here, you will find a class that contains all the test runned when the command cellconstructor_test.py is executed. Remember, after editing each part of the code, no matter how small, always check that you did not break other parts by running the testsuite (after having reinstalled the software)

```
$ cellconstructor_test.py
```

To add your own test, have a look inside that script. You just need to add a function to the class `TestStructureMethods`. Your function must start with `test_` and take only `self` as argument. To retrive some example dynamical matrix or strctures are inside `self`. For example, a ice XI structure is:

```
# ICE XI structure
self.struct_ice

# Dynamical matrix of SnSe (a supercell)
self.dynSnSe

# Dynamical matrix of TiSe (a supercell)
self.dynSky
```

You can also add your own file, by either expliciting coding it inside the `__init__(self)` method or by storing online and writing a download function. Remember that if you store them online, the file should be always be available.

Inside the testing function, you must check if the code is executed correctly by using `self.assertTrue(cond)` where `cond` is a bool condition that must be fullfilled, if not the test fails (it means a bug is present).

You can find online a more detailed guide on the `unittest` library.

You can also add custom tests inside the tests directory, naming a python script as test_my_function.py. Look to other test you find in the same directory for examples.

## 4.3 The Fortran interface

Sometimes, you have already written a code in Fortran, and you want to add it to CellConstructor.

If this is the case, and a complete python rewrite is impractical, then you can exploit the f2py utility provided by distutils to compile the fortran code into a shared library that will be read by Python.

This is done automatically by the setup.py installation script. Please, give a look to the FModules directory and the setup.py.

Insert the fortran modules inside FModules directory, then, add them to the setup.py source file list. In this way the fortran code will be automatically compiled when CellConstructor is installed.

### 4.3.1 How to include a new fortran source file

To include a new Fortran source file we must use the Extension class from distutils. Let us take a look on how the symmetrization fortran module from quantum espresso has been imported into python. The fortran source files are contained inside the directory FModules. In the setup.py we have

```
from numpy.distutils.core import setup, Extension
sources = [os.path.join("FModules", x) for x in os.listdir("FModules") if x.endswith(
↪".f90")]

symph_ext = Extension(name = "symph", sources = sources, libraries= ["lapack", "blas
↪"], extra_f90_compile_args = ["-cpp"])

setup(name = "CellConstructor", ext_modules = [symph_ext])
```

Here, I reduced only the lines we are interested in. First. I define a list that contains all the source files. In this case `sources` is a list of the paths to all the files that ends with ".f90" inside the FModules directory. Then, I create an Extension object, named `symph` (the name of the package to be imported in python), linked to all the fortran soruce files listed inside the sources list, I specify the extra libraries needed for the link (if gfortran is used as default compiler, it will add -llapack -lblas to the compiling command). I can also specify extra flags or arguments for the fortran compiler. In this case, I use the "-cpp" flag. Then, the `symph_ext` object is added to the setup of the cellconstructor as an external module.

If you want to add a new function to the `symph` module, you just have to add it into the FModules directory and to the sources list (In this example, it will be recognized automatically, but in the actual setup.py all the files are manually listed, so remember to add it to the sources list).

Let us see a very simple example of a `hello_world` fotran module.

Create a new directory with the following `hw.f90` file:

```
subroutine hello_world()
   print *, "Hello World"
end subroutine hello_world
```

Then we can create our python extension. Make a `setup.py` file:

```
from numpy.distutils.core import setup, Extension

hw_f = Extension(name = "fort_hw", sources = ["hw.f90"])
setup(name = "HW_IN_FORTRAN", ext_modules = [hw_f])
```

Now, you can try to install the module

```
$ python setup.py install --user
```

To test if the module works, let us open an interactive python shell:

```
>>> import fort_hw
>>> fort_hw.hello_world()
 Hello World
```

Congratulations! You have your first Fortran module correctly compiled, installed, and working inside python. For a more detailed guide on advanced features, refer to numpy fortran extension guide.

### 4.3.2 Fortran programming guidelines

In the previous section we managed to make a very simple fortran extension to python. However, codes are always much more complicated. **Remember: you are not writing a Fortran program, but a Fortran extension to a Python library**. Keep your fortran code as simple as possible.

1. Always specify explicitly the intent and dimension of the input arrays:

   ```
   subroutine sum(a,b,c,n)
      double precision, intent(in), dimension(n) :: a,b
      double precision, intent(out), dimension(n) :: c
      integer n

      c(:) = a(:) + b(:)
   end subroutine sum
   ```

   This code is the correct way to write a subroutine that sums two variables, avoid using `dimension(:)` in the declaration. Note that once your function is parsed in python, the fortran parser will recognized automatically that `n` is the dimension of the array. This means that **only in python** the dimension of the array can be omitted, as it will be inferred by the input variables. Note, array in fortran are numpy ndarray in python.

2. Pay attention to the typing. F2PY will automatically convert the type to match the input and output of your python functions, however, to get faster performances, it is better if you directly pass the correct type to the Fortran function. You can define a python type for the array using the dtype argument:

   ```
   import numpy as np
   a = np.zeros(10, dtype = np.double)
   ```

   This created the `a` array with 10 elements of type `double precision`. You can find a detailed list of the python dtype and the corresponding fortran typing on the internet.

3. Multidimensional arrays. To preserve the readability of the code, f2py preserves the correct indexing of multi-dimensional arrays. If you have a python array like:

   ```
   mat = np.zeros((100, 10), dtype = np.double)
   ```

   It will be converted into a fortran array as:

   ```
   double precision, dimension(100, 10) :: mat
   ```

   However, by default, fortran stores in memory the multidimensional array in a different way than python. The fast index in fortran is the first one (in python it is the last one). This means that python needs to do a copy of the array before passing to (or retriving from) fortran to exchange the two indexes. If you know that a python array will be used extensively in fortran, you tell to python to create it directly in fortran order:

   ```
   mat = np.zeros((100, 10), dtype = np.double, order = "F")
   ```

   Now the `mat` array is stored in memory directly in fortran order, so no copy is needed to pass it to fortran.

4. Do not use custom types in fortran. Always pass to a subroutine or a function all the variable needed for that computations.

5. For better readability of the fortran code, it should be auspicable that you use a different source file for any different subroutine.

6. Avoid using any external library apart from blas and lapack. Remember that python is very good for linear algebra with numpy, so try to use Fortran only to perform critical computations that would require a slow massive for loop in python.

7. You can use openmp directives, but avoid importing the openmp library and use openmp subroutines, this breaks the compatibility if openmp is unavailable on the machine.
8. Always add a test file of the new function you implemented. In this way, if bugs are introduced in future, we will spot them immediately.

Fortran is very good to program fast tasks, however, the fortran converted subroutines are not documented and the input is uncontrolled. This means that passing an array with wrong size or typing can result in a Segmentation Fault error. This is very annoying, as it can be very difficult to debug, especially if you are using a function written by someone else. **Each time you implement a fortran subroutine, write also the python parser**.

The parser is a python function that takes in input python arguments, converts them if necessary into the fortran types, verifies the size of the arrays to match exactly what the fortran function is expecting, calls the the fortran function, parses the output and return the output in python. It is very important that the user of CellConstructor must **never** call directly a fortran function. This should also apply to other developers: try to make other peaple need only to call your final python functions and not directly the fortran ones.

CHAPTER

# FIVE

# THE STRUCTURE MODULE

This module is the basis of CellConstructor. Here, the atomic structure and its method are defined.

This class is can be imported in the cellconstructor.Structure file.

**class** Structure.**Structure**(*nat=0*)

**GetBiatomicMolecules**(*atoms*, *distance*, *tollerance=0.01*, *return_indices=False*)

This function allows one to extract from a structure all the biatomic molecules that contains the two atoms specified and that are at the distance with a given tollerance. This is very usefull to compute some particular average bond length.

**Parameters**

- **atoms** (–) – The atomic symbols of the molecule
- **distance** (–) – The average distance between the two atom in the molecule
- **tollerance** (–) – The tollerance on the distance after which the two atoms are no more consider inside the same molecule.
- **return_indices** (–) – If true, per each molecule is returned also the list of the original indices inside the structure.

- **Molecules** [list] List of molecules (Structure) that matches the input. If none is found an empty list is returned

**GetTriatomicMolecules**(*atoms*, *distance1*, *distance2*, *angle*, *thr_dist=0.01*, *thr_ang=1*, *return_indices=False*)

This function allows one to extract from a structure all the triatomic molecules that contains the atoms specified and that are at the distance and angle with a given tollerance. This is very usefull to compute some particular average bond length.

The two distances are between the first-second and second-third atom, while the angle is between first-second-third atom.

Be carefull if the atoms are equal and the distance1 and distance2 are very similar the algorithm can find twice the same molecules.

**Parameters**

- **atoms** (–) – The atomic symbols of the molecule
- **distance1** (–) – The average distance between the first two atom in the molecule
- **distance2** (–) – The average distance between the last two atom in the molecule
- **angle** (–) – Angle (in degree) between the central atom and the other two.

- **thr_dist** (–) – The tollerance on the distance after which the two atoms are no more consider inside the same molecule.
- **thr_angle** (–) – Tollerance for the angle
- **return_indices** (–) – If true, per each molecule is returned also the list of the original indices inside the structure.

- **Molecules** [list] List of molecules (Structure) that matches the input. If none is found an empty list is returned

**IsolateAtoms**(*atoms_indices*)
This subroutine returns a Structure() with only the atoms indices identified by the provided list.

**Parameters** **atoms_indices** (`list of int`) – List of the atoms that you want to isolate

**Returns** **new_structure** – A structure with only the isolated atoms.

**Return type** *Structure*()

**apply_symmetry**(*sym_mat*, *delete_original=False*, *thr=1e-06*, *timer=<cellconstructor.Timer.Timer object>*)
This function apply the symmetry operation to the atoms of the current structure.

**Parameters**

- **sym_mat** (–) – The matrix of the symemtri operation, the final column is the translation
- **delete_original** (–) – If true only the atoms after the symmetry application are left (good to force symmetry)
- **thr** (–) – The threshold for two atoms to be considered the same in the reduction process (must be smaller than the minimum distance between two generic atoms in the struct, but bigger than the numerical error in the wyckoff positions of the structure).

**build_masses**()
Use the ASE database to build the masses. The masses will be in [Ry] units (the electron mass)

**change_unit_cell**(*unit_cell*)
This method change the unit cell of the structure keeping fixed the crystal coordinates.

NOTE: the unit_cell argument will be copied, so if the unit_cell variable is modified, this will not affect the unit cell of this structure.

**Parameters** **unit_cell** (`numpy ndarray (3x3)`) – The new unit cell

**check_symmetry**(*sym_mat*, *thr=1e-06*)
This method check if the provided matrix is actually a symmetry for the given system

**Parameters**

- **sym_mat** (–) – It contains the rotation matrix (the first 3x3 block) and the traslation vector (the last column) of the symmetry
- **thr** (–) – The threshold for two atoms to be considered the same.

- **check** [bool] It is true if the given matrix is a real symmetry of the system.

**copy**()
This method simply returns a copy of the current structure

- **aux** [Structure] A copy of the self structure.

**delete_copies** (*minimum_dist=1e-06*, *verbose=False*)
This method checks if double atoms are present in the structure, and delete them.

**Parameters**

- **minimum_dist** (–) – the minimum distance between two atoms of the same type allowed in the structure.
- **verbose** (–) – if True print on stdout how many atoms have been deleted (default False)

**export_unit_cell** (*filename*)
This method save the unit cell on the given file. The rows will be the direct lattice vectors.

**Parameters** **filename** (–) – The filename in which to save the unit cell

**fix_coords_in_unit_cell** (*delete_copies=True*, *debug=False*)
This method fix the coordinates of the structure inside the unit cell. It works only if the structure has predefined unit cell.

**fix_wigner_seitz** ()
Atoms will be replaced in the periodic images inside the wigner_seitz cell

**generate_espresso_input** (*flags*)
This subroutine will generate the input for a quantum espresso calculation

**generate_from_ase_atoms** (*atoms*, *get_masses=True*)
This subroutines generate the current structure from the ASE Atoms object

**Parameters**

- **atoms** (*the ASE Atoms object*) –
- **get_masses** (*bool*) – If true, also build the masses. Note that massess are saved in Ry units (electron mass)

**generate_supercell** (*dim*, *itau=None*, *QE_convention=True*, *get_itau=False*)
This method generate a supercell of specified dimension, replicating the system on the n-th neighbours unit cells.

**Parameters**

- **dim** (–) – A list that specifies the number of cells for each dimension.
- **itau** (–) – An array of integer. If it is of the correct shape and type it will be filled with the correspondance of each new vector to the corresponding one in the unit cell
- **QE_convention** (–) – If true (default) the quantum espresso set_tau subroutine is used to determine the order of how the atoms in the supercell are generated
- **get_itau** (–) – If true also the itau order is returned in output (python convention).

- **supercell** [Structure] This structure is the supercell of the system.

**get_angle** (*index1*, *index2*, *index3*, *rad=False*)
This function evaluate the angle between three atoms located in the structure at the correct indices. The unit cell is centered around the second atom to compute correctly the structure.

**Parameters**

- **indexI** (*int*) – Index of the Ith atom. (The angle is the one between 1-2-3)
- **rad** (*bool, optional*) – If true, the angle is returned in radiants (otherwise in degrees)

**Returns angle** – Value of the angle in degrees (unles rad is specified) between the index1-index2-index3 atoms of the structure.

**Return type** float

**get_ase_atoms**()

This method returns the ase atoms structure, ready for computations.

- **atoms** [ase.Atoms()] The ase.Atoms class containing the self structure.

**get_atomic_types**()

Get an array of integer, starting from 1, for each atom of the structure, so that two equal atoms share the same index.

This is how different types are stored in Quantum ESPRESSO and it is usefull for the wrapping Fortran => Python.

**ityp** [ndarray dtype=(numpy.intc)] The type array

**get_brillouin_zone**(*ISO_MESH=10*)

This function uses ase utilities to plot the Brillouin zone. TODO: Z primitive cell must be perpendicular to the others (only few reticulus)

NOT WORKING!!!!

**Parameters** **ISO_MESH** (*–*) – The number of points for the volume mesh (to the 3 power)

- **BZone** [array of 3D vectors] The points of the ISO_MESH inside the first brillouin zone

**get_classical_rotational_free_energy**(*temperature*, *unit_mass='Ry'*)

Get the classical free energy of a rigid rotor.

**Parameters**

- **temperature** (*float*) – Temperature in K
- **unit_mass** (*string*) – The unit of measurement of the masses. It can be one of:
  - ”uma” : the atomic mass unit (1/12 of the C12 nucleus)
  - ”Ry” : the rydberg mass (twice electron mass)
  - ”Ha” : the hartree mass (electron mass)

**free_energy** [float] The rotational free energy in eV

**get_displacement**(*target*, *dtype=<class 'numpy.float64'>*)

This function will return an array of displacement respect to the target of the current structure. Note that the two structures must be compatible.

**NOTE: if any the self unit_cell will be considered, otherwise the target one.** no unit cell is used only if neither the self nor the target have one.

**Parameters**

- **target** (*Structure.Structure()*) – The reference atomic positions (also this is a structure)
- **dtype** (*type*) – The type to be cast the result. By default is the double precision

**ndarray N_atoms x 3** The displacements (same shape as self.coords)

**get_equivalent_atoms**(*target_structure*, *return_distances=False*, *debug=False*)

This function returns a list of the atom index in the target structure that correspond to the current structure. NOTE: This method assumes that the two structures are equal.

**Parameters**

- **target_structure** (Structure *()*) – This is the target structure to be used to get the equivalent atoms.
- **return_distances** (*bool*) – If True it returns also the list of the distances between the atoms

**list** list of int. Each integer is the atomic index of the target_structure equivalent to the i-th element of the self structure.

**get_inertia_tensor**()

This method get the intertial tensor of the current structure. Note periodic boundary conditions will be ingored, so take care that the atoms are correctly centered.

The units will be the units given for the mass dot the position^2

**I** [ndarray ( size = (3,3), dtype = np.double)] The inertia tensor

**get_itau**(*unit_cell_structure*)

This subroutine (called by a supercell structure), returns the array of the corrispondence between its atoms and those in the unit cell.s

NOTE: The ITAU is returned in Fortran indexing, subtract by 1 if you want to use it in python

**Parameters** **unit_cell_structure** (–) – The structure of the unit cell used to generate this supercell structure.

- **itau** [ndarray (size = nat_sc, type = int)] For each atom in the supercell contains the index of the corrisponding atom in the unit_cell, starting from 1 to unit_cell_structure.N_atoms (included)

**get_ityp**()

This is for fortran compatibility. Get the ityp array for the structure. Pass it + 1 to the fortran subroutine to match also the difference between python and fortran indices

**ityp** [ndarray of int] The type of the atom in integer (starting from 0)

**get_ityp_from_species**(*species*)

Get the integer of the atomic type from the species (string)

**get_masses_array**()

Convert the masses of the current structure in a numpy array of size N_atoms

NOTE: This method will rise an exception if the masses are not initialized well

**masses** [ndarray (size self.N_atoms)] The array containing the mass for each atom of the system.

**get_min_dist**(*index_1*, *index_2*)

This method returns the minimum distance between atom index 1 and atom index 2. It uses the unit cell to correctly take into account the atoms at the edge of the unit cell.

**Parameters**

- **index_1** (–) – The index of the first atom in the structure
- **index_2** (–) – The index of the second atom in the structure

- **min_dist** [float] The minimum distance between the chosen atoms, eventually traslated by the unit cell.

**`get_reciprocal_vectors`**()

Get the vectors of the reciprocal lattice. The self structure must have the unit cell initialized (A NoUnitCell exception will be reised otherwise).

- **reciprocal_vectors** [float ndarray 3x3] A matrix whose rows are the vectors of the reciprocal lattice

**`get_strct_conventional_cell`**()

This methods, starting from the primitive cell, returns the same structure in the conventional cell. It picks the angle that mostly differs from 90 deg, and transfrom the axis of the cell accordingly to obtain a bigger cell, but similar to an orthorombic one. The atoms are then replicated and correctly placed inside the new cell.

If the structure does not have a unit cell, the method will raise an error.

NOTE: The new structure will be returned, but this will not be modified

**Returns** The structure with the conventional cell

**Return type** *Structure.Structure*()

**`get_sublattice_vectors`**(*unit_cell_structure*)

Get the lattice vectors that connects the atom of this supercell structure to those of the unit_cell structure.

**`get_volume`**()

Returns the volume of the unit cell

**`get_xcoords`**()

Returns the crystalline coordinates

**`impose_symmetries`**(*symmetries*, *threshold=1e-06*, *verbose=True*)

This methods impose the list of symmetries found in the given filename. It solves a self-consistente equation: Sx = x. If this equation is not satisfied at precision of the initial_threshold the method will raise an exception.

**Parameters**

- **`symmetries`** (*–*) – The simmetries to be imposed as a list of 3x4 ndarray matrices. The last column is the fractional translations
- **`threshold`** (*–*) – The threshold for the self consistent equation. The algorithm stops when Sx = x is satisfied up to the given threshold value for all the symmetries.
- **`verbose`** (*–*) – If true the system will print on stdout info about the self-consistent threshold

**`load_symmetries`**(*filename*, *progress_bar=False*, *verbose=False*)

This function loads the symmetries operation from a specific file and applies them to the system. The file must init with the total number of symmetries, and followed by N 3 rows x 4 columns matrices that represent the symmetry application.

**Parameters**

- **`filename`** (*`string`*) – The path in which the symmetries are stored, a text file.
- **`progress_bar`** (*`bool`*) – If true a progress bar on stderr is shown, usefull if the system is very large and this function can take a while.

**`read_generic_file`**(*filename*)

This reader use ASE to parse the input and build the appropriate structure. Any ASE accepted file is welcome. This very simple reader uses the ase environment.

**read_scf** (*filename*, *alat=1*)
Read the given filename in the quantum espresso format. Note: The file must contain only the part reguarding ATOMIC POSITIONS.

**Parameters**

- **filename** (–) – The filename containing the atomic positions
- **alat** (–) – If present the system will convert both the cell and the atoms position by this factor. If it is also specified in the CELL_PARAMETERS line, the one specified in the file will be used.

**read_xyz** (*filename*, *alat=False*, *epsilon=1e-08*, *frame_id=0*)
This function reads the atomic position from a xyz file format. if the passed file contains an animation, only the frame specified by frame_id will be processed.

**Parameters**

- **filename** (–) – The path of the xyz file, read access is required
- **alat** (–) – If true the coordinates will be rescaled with the loaded unit cell, otherwise cartesian coordinates are supposed (default False).
- **frame_id** (–) – The id of the frame to be processed in an animation.
- **epsilon** (–) – Each value below this is considered to be zero (defalut 1e-8)

**reorder_atoms_supercell** (*reference_structure*)
This subroutines order the atoms to match the same order as in the generate_supercell method. The self structure is supposed to be a structure that belongs to a supercell of the given unit_cell, then it is reordered so that each atom in any different supercell are consequent and the order of the supercell matches the one created by generate supercell. The code will work even if the structures do not match exactly the supercell generation. In this case, the closest unit cell atom of the correct type is used as reference.

TODO: THIS DOES NOT WORK!!!!

**Parameters** **reference_structure** (–) – The cell and coordinates that must be used as a reference to reorder the atoms

- **itau** [ndarray of int] The shuffling array to order any array of this list

**save_bcs** (*filename*, *symmetry_file=''*)
Save the current structure in the Bilbao Crystallographic Server file format This is very usefull since the BCS website provide a conversor between BCS with most of widely used crystallographic file format.

NOTE: remember to specify the correct ITA group symmetry in the structure. You can find more about ITA on BCS website. Otherwise you must specify a file with symmetries

**Parameters**

- **filename** (–) – The path of the bcs file in which you want to save the structure
- **symmetry_file** (–) – The path to a file containing the symmetries operations of the group space This is not needed if a ITA grup has been specified.

**save_scf** (*filename*, *alat=1*, *avoid_header=False*)
This methods export the phase in the quantum espresso readable format. Of course, only the data reguarding the unit cell and the atomic position will be written. The rest of the file must be edited by the user to start a calculation.

**Parameters**

- **filename** (`string`) – The name of the file that you want to save.

- **alat** (*float, optional*) – If different from 1, both the cell and the coordinates are saved in alat units. It must be in Angstrom.
- **avoid_header** (*bool, optional*) – If true nor the cell neither the ATOMIC_POSITION header is printed. Usefull for the sscha.x code.

**save_xyz** (*filename*, *comment='Generated with BUC'*, *overwrite=True*)
This function write the structure on the given filename in the xyz file format

**Parameters**

- **filename** (*string*) – The path of the file in which to save the structure. The user must have write access
- **comment** (*string, optional*) – This line is written in the comment line of the xyz file. NOTE: this string is followed by the unit cell info is present
- **overwrite** (*bool, optional*) – If true any precedent file will be erased, otherwise the structure is appended on the bottom of the previous one. In this way it is possible to save videos.

**set_from_xcoords** (*xcoords*)
Set the cartesian coordinates from crystalline

**set_ita_group** (*group*)
This function setup the ita group of the cell, the unit cell must be initialized and the ITA group must be inside the supported one. All the symmetries of the specified group are applied.

**Parameters** **group** (–) – The ITA identifier of the symmetry group.

**set_masses** (*masses*)
This method set up the masses of the system. It requires a dictionary containing the symobl and the value of the masses in a.u. (mass of the electron)

**Parameters** **masses** (–) – A dictionary containing the label and the corresponding mass in a.u. Ex. masses = {'H' : 918.68, 'O' : 14582.56}

**set_unit_cell** (*filename*, *delete_copies=False*, *rescale_coords=False*)
Read the unit cell from the filename. The rows of the filename are the unit cell vectors!

**Parameters**

- **filename** (–) – The path of the file that contains the unit cell, in the numpy datafile format (text)
- **delete_copies** (–) – If true the delete_copies subroutine is lounched after the creation of the unit cell (default False)
- **rescale_coords** (–) – If true ths system will be multiplied rows by column by the unit cell (default False)

**sort_molecules** (*distance=1.3*)
This method sorts the atom lists to have the atoms in the same molecule written subsequentially.

**Parameters** **distance** (–) – The distance below wich two atoms are considered to be bounded. The unit is in Argstrom.

CHAPTER

SIX

# THE METHODS

This library contains a set of tools to perform some general computations. For example to pass between crystal and Cartesian units, read espresso input namelists, compute reciprocal lattice vectors.

Created on Wed Jun 6 10:45:50 2018

@author: pione

`Methods.`**`DistanceBetweenStructures`**(*strc1*, *strc2*, *ApplyTrans=True*, *ApplyRot=False*, *Ordered=True*)

This method computes the distance between two structures. It is usefull to check the similarity between two structure.

Note: Ordered = False is not yet implemented

**Parameters**

- **`strc1`** (–) – The first structure. It commutes with the strc2.
- **`strc2`** (–) – The second structure.
- **`ApplyTrans`** (–) – If true both the structures are shifted in a common origin (The first atom). This works only if the atoms are ordered to match properly.
- **`ApplyRot`** (–) – If true the structure are rotated to reduce the rotational freedom.
- **`Ordered`** (–) – If true the order in which the atoms appears is supposed to match in the two structures.

- **Similarities: float** Similarity between the two provided structures

`Methods.`**`cell2abc_alphabetagamma`**(*unit_cell*)

This methods return a list of 6 elements. The first three are the three lengths a,b,c of the cell, while the other three are the angles alpha (between b and c), beta (between a and c) and gamma(between a and b).

**Parameters** **`unit_cell`** (–) – The unit cell in which the lattice vectors are the rows.

- **cell** [6 length ndarray (size = 6, dtype = type(unit_cell))] The array containing the a,b,c length followed by alpha,beta and gamma (in degrees)

`Methods.`**`covariant_coordinates`**(*basis*, *vectors*)

This method returns the covariant coordinates of the given vector in the chosen basis. Covariant coordinates are the coordinates expressed as:

$$\vec{v} = \sum_i \alpha_i \vec{e}_i$$

where $\vec{e}_i$ are the basis vectors. Note: the $\alpha_i$ are not the projection of the vector $\vec{v}$ on $\vec{e}_i$ if the basis is not orthogonal.

**Parameters**

- **basis** (–) – The basis. each $\vec{e}_i$ is a row.
- **vector** (–) – The vectors expressed in cartesian coordinates. It coould be just one ndarray(size=N)

- **cov_vector** [Nx float] The $\alpha_i$ values.

`Methods.`**`from_dynmat_to_spectrum`**(*dynmat*, *struct*)

This method takes as input the dynamical matrix and the atomic structure of the system and returns the spectrum.

**Parameters**

- **dynmat** (–) – Numpy array that contains the real-space dynamical matrix (Hartree).
- **struct** (–) – The structure of the system. The masses must be initialized.

- **Frequencies** [float, 3*N_atoms] Numpy array containing the frequencies in cm-1

`Methods.`**`get_gr`**(*structures*, *type1*, *type2*, *r_min*, *r_max*, *N_r*)

Computes the radial distribution function for the system. The $g_{AB}(r)$ is defined as

$$g_{AB}(r) = \frac{\rho^{(2)}_{AB}(r)}{\rho_A(r)\rho_B(r)}$$

where $A$ and $B$ are two different types

**Parameters**

- **structures** (–) – A list of atomic structures on which compute the $g_{AB}(r)$
- **type1** (–) – The character specifying the $A$ atomic type.
- **type2** (–) – The character specifying the $B$ atomic type.
- **r_min** (–) – The minimum and maximum cutoff value of $r$
- **r_max** (*float*) – The minimum and maximum cutoff value of $r$
- **N_r** (–) – The number of bins for the distributions

- **g_r** [ndarray.shape() = (r/dr + 1, 2)] The $g(r)$ distribution, in the first column the $r$ value in the second column the corresponding value of g(r)

`Methods.`**`get_minimal_orthorombic_cell`**(*euclidean_cell*, *ita=36*)

This function, given an euclidean cell with 90 90 90 angles, returns the minimal cell. The minimal cell will not have 90 90 90 angles.

**Parameters**

- **euclidean_cell** (–) – The rows of this matrix are the unit cell vectors in euclidean cell.
- **ita** (–) – The group class in ITA standard (36 = Cmc21)

- **minimal_cell** [matrix 3x3, double precision] The rows of this matrix are the new minimal unit cell vectors.

`Methods.`**`put_into_cell`**(*cell*, *vector*)

This function take the given vector and gives as output the corresponding one inside the specified cell.

**Parameters**

- **`cell`** (–) – The unit cell, a 3x3 matrix whose rows specifies the cell vectors
- **`vector`** (–) – The vector to be shifted into the unit cell.

- **new_vector** [double, 3 elements ndarray] The corresponding vector into the unit cell

CHAPTER

SEVEN

# THE PHONON MODULE

The Phonon module of cellconstructor deals with the dynamical matrix. All the harmonic properties of a crystal can be computed within this module.

Besides the Phonons class, some method is directly callabile in the Phonons module.

Created on Wed Jun 6 10:29:32 2018 @author: pione

`Phonons.`**`GetDynQFromFCSupercell`**(*fc_supercell*, *q_tot*, *unit_cell_structure*, *supercell_structure*, *itau=None*)

This subroutine uses the fourier transformation to get the dynamical matrices, starting from the real space force constant.

$$\tilde C_{k\alpha k'\beta}(q) = \sum_b C_{k\alpha,k'\beta}(0,b)e^{i\vec q\cdot \vec R_b}$$

Here $k$ is the atom index in the unit cell, $a$ is the supercell index, $\alpha$ is the cartesian indices.

**Parameters**

- **`fc_supercell`** (*`ndarray 3nat_sc x 3nat_sc`*) – The dynamical matrix at each q point. Note nq must be complete, not only the irreducible.
- **`q_tot`** (*`ndarray ( nq, 3)`*) – The q vectors in Angstrom^-1
- **`unit_cell_structure`** (`Structure` *`()`*) – The structure of the unit cell
- **`supercell_structure`** (`Structure` *`()`*) – The structure of the supercell

**Returns** **dynmat** – The force constant matrix in the supercell.

**Return type** ndarray (nq, 3nat, 3nat, dtype = np.complex128)

`Phonons.`**`GetSupercellFCFromDyn`**(*dynmat*, *q_tot*, *unit_cell_structure*, *supercell_structure*, *itau=None*, *img_thr=1e-06*)

This subroutine uses the fourier transformation to get the real space force constant, starting from the fourer space matrix.

$$C_{k\alpha,k'\beta}(0,b) = \frac{1}{N_q}\sum_q \tilde C_{k\alpha k'\beta}(q)e^{i\vec q\cdot \vec R_b}$$

Then the translationa property is applied.

$$C_{k\alpha,k'\beta}(a,b) = C_{k\alpha,k'\beta}(0,b-a)$$

Here $k$ is the atom index in the unit cell, $a$ is the supercell index, $\alpha$ is the cartesian indices.

**Parameters**

- **`dynmat`** (*`ndarray (nq, 3nat, 3nat, dtype = np.complex128)`*) – The dynamical matrix at each q point. Note nq must be complete, not only the irreducible.

- **q_tot** (*ndarray ( nq, 3)*) – The q vectors in Angstrom^-1
- **unit_cell_structure** (*Structure()*) – The reference structure of the unit cell.
- **supercell_structure** (*Structure()*) – The reference structure of the supercell. It is used to keep the same indices of the atomic positions. Note, it is required that consecutive atoms are placed sequently
- **itau** (*Ndarray(nat_sc) , optional*) – the correspondance between the supercell atoms and the unit cell one. If None is recomputed

**Returns** **fc_supercell** – The force constant matrix in the supercell.

**Return type** ndarray 3nat_sc x 3nat_sc

Phonons.**ImposeSCTranslations**(*fc_supercell*, *unit_cell_structure*, *supercell_structure*, *itau=None*)

This subroutine imposes the unit cell translations of the supercell force constant matrix. Note that it is very different from the acustic sum rule.

$$C_{k\alpha,k'\beta}(a,b) = C_{k\alpha,k'\beta}(0,b-a)$$

**Parameters**

- **fc_supercell** (*ndarray (3nat_sc x 3nat_sc)*) – The input-output force constant matrix in real space.
- **unit_cell_structure** (*Structure()*) – The structure in the unit cell
- **supercell_structure** (*Structure()*) – The structure of the supercell
- **itau** (*optional, ndarray (int)*) – The equivalence between unit_cell and supercell atoms. If None it is extracted by the given structures. Note it must be in fortran language

Phonons.**InterpolateDynFC**(*starting_fc*, *coarse_grid*, *unit_cell_structure*, *super_cell_structure*, *q_point*)

Interpolate the real space force constant matrix in a bigger supercell. This can be used to obtain a dynamical matrix in many other q points. This function uses the quantum espresso matdyn.x subroutines.

**Parameters**

- **starting_fc** (*ndarray(size=(3*natsc , 3*natsc), dtype = float64)*) – Array of the force constant matrix in real space.
- **coarse_grid** (*ndarray(size=3, dtype=int)*) – The dimension of the supercell that defines the starting_fc.
- **unit_cell_structure** (*Structure()*) – The structure in the unit cell
- **super_cell_structure** (*Structure()*) – The structure in the super cell
- **q_point** (*ndarray(size=3, dtype=float64)*) – The q point in which you want to interpolate the dynamical matrix.

**dyn_mat** [ndarray(size=(3*nat, 3*nat), dtype = complex128)] The interpolated dynamical matrix in the provided q point.

**class** Phonons.**Phonons**(*structure=None*, *nqirr=1*, *full_name=False*, *use_format=False*, *force_real=False*)

This class contains the phonon of a given structure. It can be used to show and display dinamical matrices, as well as for operating with them

**AdjustQStar**(*use_spglib=False*)

This function uses the quantum espresso symmetry finder to divide the q points into the proper q stars, reordering the current dynamical matrix.

**Parameters** **use_spglib** (*bool*) – If true, the SPGLIB is used to perform the symmetrization. Otherwise the quantum espresso default symmetry route is used.

**AdjustToNewCell** (*new_cell*, *symmetrize=True*)

This method is used, if you want to change the unit cell, to adjust the dynamical matrix, as the q points, in the new cell.

The method forces also the symmetrization after the strain

**Parameters** **new_cell** (*ndarray(size=(3,3), dtype=np.float64)*) – The new unit cell

**ApplySumRule** (*kind='custom'*)

The acustic sum rule is a way to impose translational symmetries on the dynamical matrix. It affects also the effective charges if any (the total effective charge must be zero). For the dynamical matrix it allows to have the self interaction terms: .. math:

```
\Phi_{n_a, n_a}^{x,y} = - \sum_{n_b \neq n_a} \Phi_{n_a,n_b}^{x,y}
```

**Parameters** **kind** (*string*) –

- **"custom"** [The polarization vectors asigned to the translation are removed from the] gamma dynamical matrix.
- "normal" : The equation written in this doc_string is applied.

A NotImplementedError is raised if kind differs from these types.

**ApplySymmetry** (*symmat*, *irt=None*)

This function apply a symmetry to the force constant matrix The matrix must be a 3 rows x 4 columns array containing the rotation and the subsequent translation of the vectors.

The symmetry check is performed by comparing the two force constant matrix within the given threshold.

$$\Phi_{s(a)s(b)}^{ij} = \sum_{h,k=1}^{3} S_{ik} S_{jh} \Phi_{ab}^{kh}$$

$$\Phi = S\Phi S^\dagger$$

where $s(a)$ is the atom in which the $a$ atom is mapped by the symmetry.

Note: this works only in supercells at gamma point

**Parameters**

- **symmat** (*ndarray 3x4*) – The symmetry matrix to be checked. the last column contains the translations. Trans
- **irt** (*ndarray (size = N_atoms)*) – The atoms the symmetry is mapping to.

**ndarray 3Nat x 3Nat** The new force constant matrix after the application of the symmetries

**CheckCompatibility** (*other*)

This function checks the compatibility between two dynamical matrices. The check includes the number of atoms and the atomic type. :param - other: The other dynamical matrix to check the compatibility. :type - other: Phonons.Phonons()

**Returns**

**Return type** bool

**Copy**()
Return an exact copy of itself. This will implies copying all the dynamical matricies and structures inside. So take care if the structure is big, because it will overload the memory.

**DiagonalizeSupercell**(*verbose=False*)
This method dyagonalizes the dynamical matrix using the supercell approach.

In this way we simply generate the polarization vector in the supercell using those in the unit cell.

This is performed using the following equation:

$$e_\mu^0(R_0) = \frac{\sqrt{|\tilde{e}_{q\nu}^a|^2}}{N_q}$$

$$e_\mu^a(R_a) = \frac{\cos(\vec{q}\cdot\Delta R_{a0})\Re\left[\tilde{e}_{q\nu}^a \tilde{e_{q\nu}^b}^\dagger\right] - \sin(\vec{q}\cdot\Delta R_{a0})\Im\left[\tilde{e}_{q\nu}^a \tilde{e_{q\nu}^b}^\dagger\right]}{e_\mu^0(R_0)N_q}$$

Here the $\tilde{e}_{q\nu}$ are the complex polarization vectors in the q point so that $\omega_{q\nu} = \omega_\mu$.

**w_mu** [ndarray( size = (n_modes), dtype = np.double)] Frequencies in the supercell

**e_mu** [ndarray( size = (3*Nat_sc, n_modes), dtype = np.double, order = "F")] Polarization vectors in the supercell

**DyagDinQ**(*iq*, *force_real_at_gamma=True*)
Dyagonalize the dynamical matrix in the given q point index. This methods returns both frequencies and polarization vectors. The frequencies and polarization are ordered. Negative frequencies are to be interpreted as instabilities and imaginary frequency, as for QE.

They are returned.

NOTE: The normalization is forced, as it is problematic for degenerate modes NOTE: if the q point is gamma, then the matrix is forced to be real

**Parameters**

- **iq** (–) – Tbe index of the q point of the matrix to be dyagonalized.
- **force_real_at_gamma** (–) – If True (default) the matrix is forced to be real during the dyagonalization (if q = 0). This assures to have real eigenvectors. This is usefull for supercells.

- **frequencies** [ndarray (float)] The frequencies (square root of the eigenvalues divided by the masses). These are in Ry units.
- **pol_vectors** [ndarray (N_modes x 3)^2] The polarization vectors for the dynamical matrix. They are returned in a Fortran fashon order: pol_vectors[:, i] is the i-th polarization vector.

**ExtractRandomStructures**(*size=1*, *T=0*, *isolate_atoms=[]*, *project_on_vectors=None*, *lock_low_w=False*)
This method is used to extract a pool of random structures according to the current dinamical matrix.

**Parameters**

- **size** (*int*) – The number of structures to be generated
- **T** (*float*) – The temperature for the generation of the ensemble
- **isolate_atoms** (*list, optional*) – A list of the atom index. Only the specified atoms are present in the output structure and displaced. This is very usefull if you want to measure properties of a particular region of the structure. By default all the atoms are used.

- **lock_low_w** (*bool*) – If True, frequencies below __EPSILON_W__ are fixed.

**Returns** A list of Structure.Structure()

**Return type** list

**ForcePositiveDefinite**()

This method force the matrix to be positive defined. Usefull if you want to start with a matrix for a SCHA calculation.

It will take the Dynamical matrix and rebuild it as

$$\Phi'_{ab} = \sqrt{M_a M_b}\sum_\mu |\omega_\mu^2| e_\mu^a e_\mu^b$$

In this way the dynamical matrix will be always positive definite.

**ForcePositiveDefinite_2**()

This method force the matrix to be positive defined. Usefull if you want to start with a matrix for a SCHA calculation.

It will take the Dynamical matrix and rebuild it as

$$\Phi'_{ab} = \sqrt{M_a M_b}\sum_\mu |\omega_\mu^2| e_\mu^a e_\mu^b$$

In this way the dynamical matrix will be always positive definite.

**ForceSymmetries**(*symmetries*, *irt=None*, *apply_sum_rule=True*)

This method forces the phonon dynamical matrix to respect the given symmetries.

It uses the method ApplySymmetry to manipulate the force constant matrix.

Note: This method only affect the force constant matrix, the structure is supposed to respect the symmetries.

Note: This works only with gamma matrices (i.e. supercells)

**Parameters**

- **symmetries** (*list of ndarray 3x4*) – List of the symmetries matrices. The last column is the fractional translation.
- **irt** (*ndarray(size = (N_sym, N_atoms_sc), dtype = np.intc)*) – For each symmetry s, the atom i is mapped into the atom irt[s, i] If None, irt is recomputed with the symmetries module.
- **apply_sum_rule** (*bool*) – If true the default sum rule is applied.

**GenerateSupercellDyn**(*supercell_size*, *img_thr=1e-06*)

This method returns a Phonon structure as it was computed directly in the supercell.

NOTE: For now this neglects bohr effective charges

**Parameters** **supercell_size** (*array int (size=3)*) – the dimension of the cell on which you want to generate the new Phonon

**dyn_supercell** [Phonons()] A Phonons class of the supercell

**GetHarmonicFreeEnergy**(*T*, *allow_imaginary_freq=False*)

The dynamical matrix can be used to obtain the vibrational contribution to the Free energy.

..math:

```
F(\Phi) = \sum_\mu \left[\frac{\hbar \omega_\mu}{2} + kT \ln\left(1 + e^{-\
↪beta \hbar\omega_\mu}\right)\right]
```

Acustic modes at Gamma are discarded from the summation. An exception is raised if there are imaginary frequencies.

**T** [float] Temperature (in K) of the system.

**Returns** **fe** – Free energy (in Ry) at the given temperature.

**Return type** float

**GetIRActive**(*use_spglib=False*)

This subroutine uses group theory to get if a mode is IR active.

**Parameters** **use_spglib** (*bool*) – If True, spglib is used for group theory. Good if you are in a supercell.

**is_ir_active** [ndarray (size = 3*nat)] Returns a bool array with True for each mode at gamma that is IR active.

**GetIRActivityVector**()

This vector returns the activity of the infrared mode. It is the matrix element to compute the responce function of the IR experiment.

**v_ir** [ndarray(size = (3, 3*natoms), dtype = np.double)] The ir activity amplitude for each polarization mode, for each polarizations of the incoming field

**GetIRIntensities**()

This function uses the effective charges to compute the infrared responce.

A list of value is returned, at each index the IR intensity of the relative mode.

**GetMatrixCFFT**()

Generate the dynamical matrix ready for the Fast Fourier Transform. This is an alternative way to go in real space. NOTE: Use only for debug purpouses

**GetProbability**(*displacement*, *T*, *upsilon_matrix=None*, *normalize=True*, *return_braket_vals=False*)

This function, given a particular displacement, returns the probability density of finding the system around that displacement. This in practical computes density matrix of the system in this way

$$\rho(\vec{u}) = \sqrt{\det(\Upsilon/2\pi)} \times \exp\left[-\frac{1}{2}\sum_{ab} u_a \Upsilon_{ab} u_b\right]$$

Where $\vec{u}$ is the displacement, $\Upsilon$ is the inverse of the covariant matrix computed through the method self.GetUpsilonMatrix().

NOTE: I think there is an error in the implementation, in fact the Upsilon matrix is in bohr^-2 while displacements are in Angstrom.

**Parameters**

- **displacement** (*ndarray(3xN) or ndarray(N, 3)*) – The displacement on which you want to compute the probability. It can be both an array of dimension 3 x self.structure.N_atoms or a bidimensional array of structure (N_atoms, 3).
- **T** (*float*) – Temperature (Kelvin) for the calculation. It will be discarded if a costum upsilon_matrix is provided.

- **upsilon_matrix** (*ndarray (3xN)^2, optional*) – If you have to compute many times this probability it can be convenient to compute only once the upsilon matrix, and recycle it. If it is None (as default) the upsilon matrix will be recomputed each time.
- **normalize** (*bool, optional*) – If false (default true) the probability distribution will not be normalized. Useful to check if the exponential weight is the same after some manipulation
- **return_braket_vals** (*bool, optional*) – If true the value returned is only the <u | Upsilon |u> braket followed by the eigenvalues of the Upsilon matrix.

**Returns** The probability density of finding the system in the given displacement.

**Return type** float

**GetRamanActive** (*use_spglib=False*)

This simple subroutines tries to guess by symmetry analisys which mode is active or not. If a raman tensor is present, it will be used to test the activity, otherwise, a random one will be generated

**Parameters** **use_spblib** (*bool*) – If True the spglib library is used to initialize symmetries. Usefull if the phonon matrix is in a super cell.

**raman_activity_mask** [ndarray(size = (3*nat), dtype = bool)] A mask that is False or True if a mode in the unit cell is Raman-active or not.

**GetRamanResponce** (*pol_in, pol_out, T=0*)

Evaluate the raman response using the Mauri-Lazzeri equation. This subroutine needs the Raman tensor to be defined, and computes the intensity for each mode. It returns a list of intensity associated to each mode.

$$I_\nu = \left|\sum_{xy} \epsilon_x^{(1)} A_{xy}^\nu \epsilon_y^{(2)}\right|^2 \frac{n_\nu + 1}{\omega_\nu}$$

Where $\epsilon$ are the polarization vectors of the incoming/outcoming light, $n_\nu$ is the bosonic occupation number associated to the $\nu$ mode, and $A_{xy}^\nu$ is the Raman tensor in the mode rapresentation

**Parameters**

- **pol_in** (*ndarray 3*) – The polarization versor of the incominc electric field
- **pol_out** (*ndarray 3*) – The polarization versor of the outcoming electric field
- **T** (*float*) – The tempearture of the calculation

**ndarray (nmodes)** Intensity for each mode of the current dynamical matrix.

**GetRamanVector** (*pol_in, pol_out*)

Get the Raman vector. It is the vector obtained from the Raman Tensor:

$$v_\nu = \sum_{xy} \epsilon_x^{(1)} \epsilon_y^{(2)} A_{xy}^\nu$$

This is defined in real space.

**Parameters**

- **pol_in** (*ndarray (size = 3)*) – Incoming polarization
- **pol_out** (*ndarray (size = 3)*) – Outcoming polarization

**vnu** [ndarray(size = 3*nat)] The raman intensity vector along each atomic displacement.

**GetRatioProbability**(*structure*, *T*, *dyn0*, *T0*)

This method compute the ration of the probability of extracting a given structure at temperature T generated with dyn0 at T0 if the extraction is made with the self dynamical matrix.

It is very usefull to perform importance sampling tests.

$$w(\vec{u}) = \frac{\rho_{D_1}(\vec{u}, T)}{\rho_{D_0}(\vec{u}, T_0)}$$

Where $D_1$ is the current dynamical matrix, while $D_0$ is the dynamical matrix that has been actually used to generate dyn0

TODO: It seems to return wrong results NOTE: This subroutine seems to return fake results, please be carefull.

**Parameters**

- **structure** (`Structure.Structure()`) – The atomic structure generated according to dyn0 and T0 to evaluate the statistical significance ratio.
- **T** (*float*) – The target temperature
- **dyn0** (`Phonons.Phonons()`) – The dynamical matrix used to generate the given structure.
- **T0** (*float*) – The temperature used in the generation of the structure

**float** The ratio $w(\vec{u})$ between the probabilities.

**GetRealSpaceFC**(*supercell_array=(1, 1, 1)*, *super_structure=None*, *img_thr=1e-06*)

This subroutine uses the fourier transformation to get the real space force constant, starting from the fourer space matrix.

$$C_{k\alpha,k'\beta}(0,b) = \frac{1}{N_q}\sum_q \tilde{C}_{k\alpha k'\beta}(q)e^{i\vec{q}\cdot\vec{R}_b}$$

Then the translationa property is applied.

$$C_{k\alpha,k'\beta}(a,b) = C_{k\alpha,k'\beta}(0,b-a)$$

Here $k$ is the atom index in the unit cell, $a$ is the supercell index, $\alpha$ is the cartesian indices.

NOTE: This method just call the GetSupercellFCFromDyn, look at its documentation for further info.

**Returns**

- **fc_supercell** (*ndarray 3nat_sc x 3nat_sc*) – The force constant matrix in the supercell. If it is a supercell structure, it is use that structure to determine the supercell array
- **super_structure** (*Structure()*) – If given, it is used to generate the supercell. Note that in this case the supercell_array argument is ignored

**GetStrainMatrix**(*new_cell*, *T*, *threshold=1e-05*, *x_start=0.01*)

This function strains the dynamical matrix to fit into the new cell. It will modify both the polarization vectors and the frequencies.

The strain is performed on the covariance matrix.

$${\Upsilon_{axby}^{-1}}' = \sum_{\alpha,\beta=x,y,z} \varepsilon_{x\alpha}\varepsilon_{y\beta}\Upsilon_{a\alpha b\beta}^{-1}$$

Then the new $\Upsilon^{-1}$ matrix is diagonalized, eigenvalues and eigenvector are built, and from them the new dynamical matrix is computed.

**NOTE: This works only at Gamma** I think there is a bug if T != 0 in the solver. BE CAREFULL!

**Parameters**

- **new_cell** (*ndarray 3x3*) – The new unit cell after the strain.
- **T** (*float*) – The temperature of the strain (default 0)
- **threshold** (*float*) – The threshold for the convergence of the newton algorithm to find the frequencies given the eigenvalues of the upsilon matrix.
- **x_start** (*float*) – The initial guess for the newton algorithm.

**dyn** [Phonons.Phonons()] A new dynamical matrix strained. Note, the current dynamical matrix will not be modified.

**GetSupercell**()

Return the supercell along which this matrix has been generated.

**supercell** [list of 3 int] The supercell in each direction.

**GetUpsilonMatrix**(*T*, *min_w_threshold=1e-08*, *debug=False*, *verbose=False*)

This subroutine returns the inverse of the correlation matrix. It is computed as following

$$\Upsilon_{ab} = \sqrt{M_a M_b}\sum_\mu \frac{2\omega_\mu}{(1+2n_\mu)\hbar} e_\mu^a e_\mu^b$$

It is used to compute the probability of a given atomic displacement. The resulting matrix is a 3N x 3N one ordered as the dynamical matrix here. The result is in bohr^-2, please be carefull.

**Parameters**

- **T** (*float*) – Temperature of the calculation (Kelvin)
- **min_w_threshold** (*float*) – The threshold for frequency under which the modes are considered fixed and neglected (as Gamma acoustic modes).

**Returns** The inverse of the correlation matrix in the supercell. N is the number of atoms in the supercell

**Return type** ndarray(3N x3N), dtype = np.float64

**Interpolate**(*coarse_grid*, *fine_grid*, *support_dyn_coarse=None*, *support_dyn_fine=None*, *symmetrize=False*)

This method interpolates the dynamical matrix in a finer mesh. It is possible to use a different dynamical matrix as a support, then only the difference of the current dynamical matrix with the support is interpolated. In this way you can easier achieve convergence.

NOTE: This method ignores effective charges. If you want to account for effective charges you should use the ForceTensor.Tensor2 class to interpolate.

**Parameters**

- **coarse_grid** (*ndarray(size=3, dtype = int)*) – The current q point mesh size
- **fine_grid** (*ndarray(size=3, dtype = int)*) – The final q point mesh size
- **support_dyn_coarse** (*Phonons(), optional*) – A dynamical matrix used as a support in the same q grid as this one. Note that the q points must coincide with the one of this matrix.

- **support_dyn_fine** (Phonons(), optional) – The support dynamical matrix in the finer cell. If given, the fine_grid is read by the q points of this matrix, and must be compatible with the fine_grid.
- **symmetrize** (bool, optional) – If true activate the symmetrization for the new matrix

**interpolated_dyn** [Phonons()] The dynamical matrix interpolated.

**LoadFromQE** (*fildyn_prefix*, *nqirr=1*, *full_name=False*, *use_format=False*)

This Function loads the phonons information from the quantum espresso dynamical matrix. the fildyn prefix is the prefix of the QE dynamical matrix, that must be followed by numbers from 1 to nqirr. All the dynamical matrices are loaded.

**Parameters**

- **fildyn_prefix** (–) – Quantum ESPRESSO dynmat prefix (the files are followed by the q irreducible index)
- **nqirr** (–) – Number of irreducible q points in the space group (supercell phonons). If 0 or negative an exception is raised.
- **full_name** (–) – If it is True, then the dynamical matrix is loaded without appending the q index. This is compatible only with gamma point matrices.
- **use_format** (–) – If true, the IQ index of the dynamical matrix is replaced in the specified format, i.e. a standard matrix with prefix dyn (dyn1, dyn2, ...) will be dyn{} with the format notation. This allows the user to insert the IQ index in many formats and any position of the file name.

**ReadInfoFromESPRESSO** (*filename*, *read_dielectric_tensor=True*, *read_eff_charges=True*, *read_raman_tensor=True*)

This method reads the effective charges, the dielectric tensor as well as the Raman tensor from an espresso phonon output file. It is usefull if you want to run the electric field perturbation without computing all the phonon spectrum, deriving only with respect to the electric field.

**Parameters**

- **filename** (string) – Path to the standard output of the ph.x calculation.
- **read_dielectric_constant** (bool) – If False, the dielectric tensor will be ignored
- **read_effective_charge** (bool) – If False, the effective charges will be ignored
- **read_raman_tensor** (bool) – If False, the Raman tensor will be ignored.

**SwapQPoints** (*other_dyn*)

Adjust the order of the q points of this dynamical matrix (self) to match the one of the passed dynamical matrix. This is usefull if you want to compare the two dynamical matrices.

The method also checks if the q points are in different brilluin zones.

NOTE: this method will match the q points, this means that the q star could be destroyed. You need to call AdjustQStar to correctly generate the star after this method.

**Symmetrize** (*verbose=False*, *asr='custom'*, *use_spglib=False*)

This subroutine uses the QE symmetrization procedure to obtain a full symmetrized dynamical matrix.

**Parameters**

- **verbose** (bool) – If true a lot of info regarding the symmetrization are printed.
- **asr** (string) – The kind of the acustic sum rule. Allowed are 'crystal', 'simple' or 'custom'. for crystal and simple refer to the quantum-espresso guide.

- **use_spglib** (*bool*) – If True, the simmetrization is performed with SPGLIB in the supercell

**SymmetrizeSupercell** (*supercell_size=None*)
Testing function, it applies symmetries in the supercell.

**get_energy_forces** (*structure*, *vector1d=False*, *real_space_fc=None*, *super_structure=None*, *supercell=None*, *displacement=None*, *use_unit_cell=True*)

This subroutine computes the harmonic energy and the forces for the given dynamical matrix at harmonic level.

$$E =$$

rac 12 sum_{alphabeta} left(r_{alpha} - r^0_{alpha}right)Phi_{alphabeta} left(**r_**beta - r^0_beta ight)

**F_**alpha = sum_{beta} Phi_{alphabeta} left(**r_**beta - r^0_beta

ight)

The energy is given in Rydberg, while the force is given in Ry/Angstrom

NOTE: In this very moment it has been tested only at Gamma (unit cell)

**structure** [Structure.Structure()] A unit cell structure in which energy and forces on atoms are computed

**vector1d** [bool, optional] If true the forces are returned in a reshaped 1d vector.

**real_space_fc** [ndarray 3nat_sc x 3nat_sc, optional (default None)] If provided the real space force constant matrix is not recomputed each time the method is called. Usefull if you have to repeat this calculation many times. You can get the real_space_fc using the method GetRealSpaceFC.

**super_structure** [Structure.Structure()] Optional, not required. If given is the super-structure used to compute the distance from the target one. You can pass it to avoid regenerating it each time this subroutine is called. If you do not pass it, you must provide the supercell size (if different than the unit cell)

**super_cell** [list of 3 items] This is the supercell on which compute the energy and force. If none it is inferred by the dynamical matrix.

**displacement:** The displacements from the self average position to be used. It is not necessary since they can be recomputed, however if provided, the calculation is faster. It must be in Angstrom. To speedup the calculations, many displacements can be provided, in the form: displacement.shape = (N_config, 3*nat_sc) where N_config are the number of configurations, nat_sc the atoms in the supercell

**use_unit_cell** [bool] If ture, do not compute the real space force constant matrix on the super cell. This is the fastest option. Put it to false only for debugging purpouses.

**energy** [float (or ndarray.shape(N_config))] The harmonic energy (in Ry) of the structure

**force** [ndarray N_atoms x 3 or N_config, nat_sc, 3)] The harmonic forces that acts on each atoms (in Ry / A)

**get_phonon_dos** (*w_array*, *smearing*, *exclude_acoustic=True*, *use_cm=False*)
This function plots the phonon dos.

**Parameters**

- **w_array** (*ndarray*) – The frequencies at which you want to compute the phonon dos [in Ry] (or cm-1, see use_cm).
- **smearing** (*float*) – The smearing [in Ry] (or cm-1, see use_cm).
- **exclude_acoustic** (*bool*) – If true, the acoustic modes at gamma are excluded.
- **use_cm** (*bool*) – If true, the frequency array and the smearing is supposed to be given in cm-1 instead of Ry.

**dos** [ndarray(size = (w_array), dtype = np.float64)] The phonon density of state.

**get_phonon_propagator**(*w_array*, *smearing=1e-05*, *only_gamma=False*)

This method computes the single phonon harmonic propagator. It is computed in the supercell

$$G_{ab}(\omega) = \sum_\mu \frac{e_\mu^a e_\mu^b}{(\omega - i\eta)^2 - \omega_\mu^2}$$

This is in real space

**Parameters**

- **w_array** (–) – The frequencies at which you want to compute the propagator. In [Ry]
- **smearing** (–) – The $\eta$ value.
- **only_gamma** (–) – If True, only the phonons at gamma will be used

- **G_abw** [ndarray(size = (3nat, 3nat, len(w)))] The real space green function

**get_two_phonon_dos**(*w_array*, *smearing*, *temperature*, *q_index=0*, *exclude_acoustic=True*)

This subroutine compute the two phonon DOS of the given dynamical matrix. It analyzes all possible phonon-phonon scattering and decayment to build the two body density of states. This can be used to get an idea how much each phonon can interact with the other in presence of anharmonicity just considering energy conservation law and Bose-Einstein statistic.

The DOS equation is

$$\rho^{(2)}(q,\omega) = \int d^3k_1 d^3k_2 \sum_{\mu\nu} \Big[(n_\mu + n_\nu + 1)\delta(\omega - \omega_\mu(k_1) - \omega_\nu(k_2))\delta^3(\vec{k}_1 + \vec{k}_2 - \vec{q})$$
$$+2(n_\mu - n_\nu)\delta(\omega - \omega_\mu(k_1) + \omega_\nu(k_2))\delta^3(\vec{q} + \vec{k}_1 - \vec{k}_2)\Big]$$

Where the Delta function are replaced by the Lorenzian shape to consider a smearing.

**Parameters**

- **w_array** (*ndarray*) – The frequency of the dos
- **smearing** (*float*) – The smearing used to compute the DOS. To converge the smearing you need to study the limit $\lim_{\sigma\to 0}\lim_{N_q\to\infty} DOS$
- **q_index** (*int*) – The q point in which to compute the phonon DOS. You must pass the index that matches the q_tot list.
- **exclude_acustic** (*bool, default = False*) – If True the acoustic modes at gamma are neglected in the DOS. NOTE: if you have few q points, you will not see the frequencies of the real mode in the DOS!

**dos** [ndarray] The array of the density of state returned. Same shape as w_array

**get_two_phonon_propagator**(*w*, *T*, *smearing=1e-05*)

This subroutine computes the two phonons propagator defined as

$$\chi_{\mu\nu}(z,q) = \frac{1}{\beta}\sum_l G_\mu(i\Omega_l, \vec k)G_\nu(z - i\Omega_l, \vec q - \vec k)$$

$$\chi_{\mu\nu}(z,q) = \frac{\hbar}{2\omega_\mu\omega_\nu}\left[\frac{(\omega_\nu+\omega_\mu)[1+n_\nu+n_\mu]}{(\omega_\nu+\omega_\mu)^2 - z^2} - \frac{(\omega_\nu-\omega_\mu)[n_\nu-n_\mu]}{(\omega_\nu-\omega_\mu)^2 - z^2}\right]$$

This is the phonon dynamical bubble. This is computed in the polarization basis. The translational modes are discarted.

**Parameters**

- **w** (*ndarray*) – The values of the dynamical frequency to compute the phonon propagator.
- **T** (*float*) – The temperature to compute the bosonic occupation numbers $n_\mu$.
- **semaring** (*float, default = 1e-5*) – The smearing [Ry] to achieve a faster convergence with the k-mesh sampling.

**chi** [ndarray(size=(3*nat, 3*nat), dtype = np.complex128)] The bubble phonon propagator

**save_phononpy**(*supercell_size=(1, 1, 1)*)

This tool export the dynamical matrix into the PHONONPY plain text format. We save them in Ry/bohr^2, as the quantum espresso format. Please, remember this when using Phononpy for the conversion factors.

It will create a file called FORCE_CONSTANTS, one called unitcell.in with the info on the structure

**Parameters** **supercell_size** (*list of 3*) – The supercell that defines the dynamical matrix, note phononpy works in the supercell.

**save_qe**(*filename*, *full_name=False*)

This subroutine saves the dynamical matrix in the quantum espresso file format. The dynmat is the force constant matrix in Ry units.

$$\Phi_{ab} = \sum_\mu \omega_\mu^2 e_\mu^a e_\mu^b \sqrt{M_a M_b}$$

Where $\Phi_{ab}$ is the force constant matrix between the a-b atoms (also cartesian indices), $\omega_\mu$ is the phonon frequency and $e_\mu$ is the polarization vector.

**Parameters**

- **filename** (*string*) – The path in which the quantum espresso dynamical matrix will be written.
- **full_name** (*bool*) – If true only the gamma matrix will be saved, and the irreducible q point index will not be appended. Otherwise all the file filenameIQ where IQ is an integer between 0 and self.nqirr will be generated. filename0 will contain all the information about the Q points and the supercell.

Phonons.**get_dyn_from_ase_phonons**(*ase_ph*)

This function converts an ASE phonons object into the cellconstructor Phonons.

**Parameters** **ase_ph** (*ase.phonons.Phonons()*) – The ASE Phonons. It must be already computed

**dyn** [CC.Phonons.Phonons()] The dynamical matrix

CHAPTER

EIGHT

# THE MANIPULATE MODULE

This is an extra tool used to manipulate the Phonons and the structures to perform some analysis. It can be used to get video of vibrations.

This module contains the methods that requre to call the classes defined into this module.

`Manipulate.`**`AlignStructures`**(*source*, *target*, *verbose=False*)

This method finds the best alignment between the source and the target. The source structure coordinates are then translated and the periodical images are chosen so to match as closely as possible the target structure

NOTE: This subroutine will rise an exception if the structures are not compatible.

**Parameters**

- **`source`** (`Structure()`) – The structure to be aligned
- **`target`** (`Structure()`) – The target structure of the alignment
- **`verbose`** (*`bool`*) – If true prints info during the minimization. Good for debugging.

**align_cost** [float] The sum of all the residual distances between atoms (after alignment)

`Manipulate.`**`BondPolarizabilityModel`**(*structure*, *bonded_atoms*, *distance*, *threshold*, *alpha_l*, *alpha_p*, *alpha1_l*, *alpha1_p*)

This method uses an empirical model to obtain the raman tensor of a structure. It is based on the assumption that all the polarizability of the system is encoded in a molecular bond.

NOTE: UNTESTED

**Parameters**

- **`structure`** (`Structure.Structure()`) – The molecular structure on which you want to build the polarizability
- **`bonded_atoms`** (*`list of two strings`*) – The atomic species that form a bond
- **`distance`** (*`float`*) – The average distance of a bond
- **`threshold`** (*`float`*) – The threshold on the atomic distance to identify two atoms that are bounded
- **`alpha_l`** (*`float`*) – The longitudinal polarization of the bond
- **`alpha_p`** (*`float`*) – The perpendicular polarization of the bond
- **`alpha1_l`** (*`float`*) – The derivative of the longitudinal polarization with respect to the bond length.
- **`alpha1_p`** (*`float`*) – The derivative of the perpendicular polarization with respect to the bond length.

**raman_tensor** [ndarray(size = (3,3, 3*nat), dtype = np.float64)] The raman tensor for the structure as computed by the bond polarizability model

Manipulate.**ChooseParamForTransformStructure**(*dyn1*, *dyn2*, *small_thr=0.8*, *n_ref=100*, *n_max=10000*)

This subroutine is ment for automatically check the best values for mode_exchange and mode_sign to pass to the TransfromStructure. This is needed if you want to maximize correlation between the structures, because the transformation between two dynamical matrix is not uniquily defined. The algorithm checks exchange the mode and sign to algin at best the polarization vectors.

At each step a random moovement is aptented. The first vector to moove is piked with a probability equal to

$$P_0(\mu) \propto 1 - \left|\left< e_\mu^{(0)} | e_\mu^{(1)}\right>\right|$$

Then the second mode is chosen as

$$P(\nu|\mu) \propto P_0(\nu)\cdot \frac{1}{|\Delta\omega| + \eta}$$

Where $\eta$ is a smearing factor to avoid explosion in degenerate modes. The exchange between the two modes is accepted if the total score is increased. The total score is measured as

$$S = \sum_\mu \left|\left< e_\mu^{(0)} | e_\mu^{(1)}\right>\right|$$

NOTE: Works only at Gamma

**Parameters**

- **dyn1** (`Phonons.Phonons`) – The starting dynamical matrix
- **dyn2** (`Phonons.Phonons`) – The final dynamical matrix
- **small_thr** (*float, optional*) – If the scalar product for each mode is bigger than this value, then the optimization is considered to be converged
- **n_ref** (*int, optional*) – A positive integer. Even if small_thr is not converged. After n_ref refused mooves, the system is considered to be converged.
- **n_max** (*int, optional*) – The total number of move, after which the method is stopped even if convergence has not been achieved.

**mode_exchange** [list] The mode_exchange list to be feeded into TransformStructure

**mode_sign** [list] The mode_sign list to be feeded into TransformStructure

Manipulate.**GenerateXYZVideoOfVibrations**(*dynmat*, *filename*, *mode_id*, *amplitude*, *dt*, *N_t*, *supercell=(1, 1, 1)*)

This function save in the filename the XYZ video of the vibration along the chosen mode.

NOTE: this functionality is supported only at gamma.

**Parameters**

- **filename** (*str*) – Path of the filename in which you want to save the video. It is written in the xyz format, so it is recommanded to use the .xyz extension.
- **mode_id** (*int*) – The number of the mode. Modes are numbered by their frequencies increasing, starting from imaginary (unstable) ones (if any).
- **amplitude** (*float*) – The amplitude in Angstrom of the vibration per atom.
- **dt** (*float*) – The time step between two different steps in femtoseconds.

- **N_t** (*int*) – The total number of frames.
- **supercell** (*list of 3 ints*) – The dimension of the supercell to be shown

`Manipulate.`**`GetIRSpectrum`**(*dyn, w_array, smearing*)

This method get a ready to plot harmonic IR spectrum. The effective charge must be included in the dynamical matrix

**Parameters**

- **dyn** (–) – The dynamical matrix on which to compute the IR. It must contain valid effective charges.
- **w_array** (–) – The values of the frequencies at which to compute the IR. in [Ry]
- **smearing** (–) – The value of the smearing for the plot (in Ry).

- **ir_spectrum: ndarray( size = (len(w)))** The ir spectrum at each frequency.

`Manipulate.`**`GetQ_vectors`**(*structures, dynmat, u_disps=None*)

The q vector is a vector of the displacement respect to the polarization basis. It is computed as

$$q_\mu = \sum_\alpha \sqrt{M_\alpha} u_\alpha e_\mu^\alpha$$

where $M_\alpha$ is the atomic mass, $u_\alpha$ the displacement of the structure respect to the equilibrium and $e_\mu^\alpha$ is the $\mu$-th polarization vector in the supercell.

**Parameters**

- **structures** (*list*) – The list of structures to compute q
- **dynmat** (`Phonons()`) – The dynamical matrix from which the polarization vectors and the reference structure are computed. This must be in the supercell
- **u_disps** (*ndarray(size=(N_structures, 3*nat_sc), dtype = np.float64)*) – The displacements with respect to the average positions. This is not necessary (if None it is computed), but can speedup a lot the calculation if provided.

**q_vectors** [ndarray (M x 3N)] A fortran array of the q_vectors for each of the M structures dtype = numpy.float64

`Manipulate.`**`GetScalarProductPolVects`**(*dyn1, dyn2, mode_exchange=None, mode_sign=None*)

This subroutines computes the scalar product between all the polarization vectors of the dynamical matrix. It is very usefull to check if the associated mode is almost the same. This subroutine checks also if the dynamical matrices are compatible

Also this subroutine works only at GAMMA

**Parameters**

- **dyn1** (`Phonons()`) – The dynamical matrix 1
- **dyn2** (`Phonons()`) – The dynamical matrix 2
- **mode_exchange** (*ndarray (int)*) – The array that shuffle the modes of the first matrix according to its data. This transformation is applied before mode_sign
- **mode_sign** (*ndarray(int)*) – An array made only by int equal to 1 and -1. It changes the sign of the corresponding polarization vector of the first dynamical matrix. This transformation is applied after mode_exchange

**ndarray ( 3*n_atoms )** The scalar products between the polarization vectors.

Manipulate.**GetSecondOrderDipoleMoment**(*original_dyn*, *structures*, *effective_charges*, *T*, *symmetrize=True*)

This method computes the second order dipole moment. It is the average second derivative of the dipole moment with respect to the atomic displacements.

$$\frac{\partial M_x}{\partial R_a \partial R_b}$$

It can be used to compute the two phonon IR response.

Note: both the structures and the effective charges must be defined on the same supercell as the original_dyn

The derivative of the dipole moment is computed using the average ensemble rule:

$$\frac{\partial^2 M_x}{\partial R_a \partial R_b} = -\sum_q \Upsilon_{aq} \left\langle u_q \frac{\partial M_x}{\partial R_b} \right\rangle$$

If the original dynamical matrix has an effective charge, then it is removed from any effective charge, to better sample the linear dependence.

**Parameters**

- **original_dyn** (–) – The dynamical matrix of the equilibrium system.
- **structures** (–) – A list of CC.Structure.Structure() of the displaced structure with respect to original_dyn on which the effective charges are computed. Alternatively, you can pass a list of strings that must point to the .scf files of the structures.
- **effective_charges** (–) – A list of the effective charge tensor. Alternatively you may provide a list of strings that must point to the output of the ph.x package from Quantum ESPRESSO, where the effective charges are printed.
- **T** (–) – The temperature (in K)
- **symmetrize** (–) – If True the tensor is simmetrized. This requires the symmetrization in real space, therefore spglib must be available.

- **dM_dRdR** [ndarray (size = (3*nat_sc, 3*nat_sc, 3))] The second derivative of the dipole moment. nat_sc are the atoms in the supercell. The first two components are the cartesian indices of the displacements, while the last is the dipole vector

Manipulate.**GetTwoPhononIR**(*original_dyn*, *structures*, *effective_charges*, *T*, *w_array*, *smearing*, **args*, ***kwargs*)

This method computes the two phonon IR response function for an harmonic dynamical matrix. The two phonon IR is due to quadratic terms in the dipole moment (linear part of the effective charge)

The IR intensity due to the two phonon structures is

$$I(\omega) = \frac{1}{4}\sum_{x=1}^{3}\sum_{abcd} \frac{\partial^2 M_x}{\partial R_a \partial R_b}\frac{\partial^2 M_x}{\partial R_c \partial R_d}\frac{-\Im G_{abcd}(\omega)}{\sqrt{m_a m_b m_c m_d}}$$

where the $G_{abcd}(omega)$ is the two phonon propagator

$$G_{abcd}(z) = \sum_{\mu\nu} \frac{e_\mu^a e_\nu^b e_\mu^c e_\nu^d}{2\omega_\mu \omega_\nu}\left[\frac{(\omega_\mu + \omega_\nu)(n_\mu + n_\nu + 1)}{(\omega_\mu + \omega_\nu)^2 - z^2} - \frac{(\omega_\mu - \omega_\nu)(n_\mu - n_\nu)}{(\omega_\mu - \omega_\nu)^2 - z^2}\right]$$

NOTE: This function calls the GetTwoPhononIRFromSecondOrderDypole

So refer to that one for documentation in extra parameters

**Parameters**

    - **original_dyn** (–) – The dynamical matrix of the equilibrium system.
    - **structures** (–) – A list of CC.Structure.Structure() of the displaced structure with respect to original_dyn on which the effective charges are computed. Alternatively, you can pass a list of strings that must point to the .scf files of the structures.
    - **effective_charges** (–) – A list of the effective charge tensor. Alternatively you may provide a list of strings that must point to the output of the ph.x package from Quantum ESPRESSO, where the effective charges are printed.
    - **T** (–) – The temperature (in K)
    - **w_array** (–) – A real valued array of the frequencies for the IR signal. The energy must be in [Ry]
    - **smearing** (–) – The value of the 0^+ in the phonon propagator. This allows the response to have a non vanishing imaginary part

- **ir_2_ph** [ndarray] The 2-phonons IR intensity at each w_array frequency.

Manipulate.**GetTwoPhononIRFromSecondOrderDypole**(*original_dyn*, *dM_dRdR*, *T*, *w_array*, *smearing*, *use_fortran=True*, *verbose=False*)

This method computes the two phonon IR response function for an harmonic dynamical matrix. The two phonon IR is due to quadratic terms in the dipole moment (linear part of the effective charge)

The IR intensity due to the two phonon structures is

$$I(\omega) = \frac{1}{4}\sum_{x=1}^{3}\sum_{abcd}\frac{\partial^2 M_x}{\partial R_a\partial R_b}\frac{\partial^2 M_x}{\partial R_c\partial R_d}\frac{-\Im G_{abcd}(\omega)}{\sqrt{m_a m_b m_c m_d}}$$

where the $G_{abcd}(omega)$ is the two phonon propagator

$$G_{abcd}(z) = \sum_{\mu\nu}\frac{e_\mu^a e_\nu^b e_\mu^c e_\nu^d}{2\omega_\mu\omega_\nu}\left[\frac{(\omega_\mu+\omega_\nu)(n_\mu+n_\nu+1)}{(\omega_\mu+\omega_\nu)^2-z^2}-\frac{(\omega_\mu-\omega_\nu)(n_\mu-n_\nu)}{(\omega_\mu-\omega_\nu)^2-z^2}\right]$$

**Parameters**

- **original_dyn** (–) – The dynamical matrix of the equilibrium system.
- **structures** (–) – A list of CC.Structure.Structure() of the displaced structure with respect to original_dyn on which the effective charges are computed. Alternatively, you can pass a list of strings that must point to the .scf files of the structures.
- **effective_charges** (–) – A list of the effective charge tensor. Alternatively you may provide a list of strings that must point to the output of the ph.x package from Quantum ESPRESSO, where the effective charges are printed.
- **T** (–) – The temperature (in K)
- **w_array** (–) – A real valued array of the frequencies for the IR signal. The energy must be in [Ry]
- **smearing** (–) – The value of the 0^+ in the phonon propagator. This allows the response to have a non vanishing imaginary part
- **use_fortran** (–) – If true the fortran library is used to perform the calculation This is very convenient, as it speed up the calculation and avoids storing massive amount of memory. Use it to false only for testing purpouses or for very small system with many freqiencies (in that case the python implemetation could be faster)

- **verbose** (*–*) – If true prints the timing on output.

- **ir_2_ph** [ndarray] The 2-phonons IR intensity at each w_array frequency.

`Manipulate.`**`LoadXYZTrajectory`**(*fname*, *max_frames=- 1*, *unit_cell=None*)

This function returns a list of structures containing the frames of the animation in the xyz file from fname.

- **fname** [string] Path to the xyz animation file
- **max_frames** [int, default = -1] Final number of the frame to be loaded. If negative read all the frames
- **unit_cell** [ndarray 3x3, default None] Unit cell of the structure, if None the structure will not have a cell

- **animation** [list(Structure)] A python list of the structures representing the frames

`Manipulate.`**`MeasureProtonTransfer`**(*structures*, *list_mol*, *verbose=False*)

This subroutine measures the proton transfer distribution from a list of structures. It requires the user to indentify the indices for the giving and the accemtping molecule.

The proton transfer coordinate is defined as

$$\nu = d(XY) - d(YZ)$$

Then the proton transfer ratio is defined as:

$$\int_0^\infty P(\nu) d\nu$$

That is the overall probability of finding the Proton that belongs to the molecule X closer to the molecule Z. This function returns a list of the $\nu$ coordinates, then the user can plot an histogram or measure the proton transfer ratio as the simple ration of $\nu > 0$ over the total.

NOTE: This subroutine will exploit MPI if available. NOTE: UNTESTED

**Parameters**

- **structures** (*list of Structure()*) – The ensemble used to measure the proton transfer
- **list_mols** (*list of (X,Y, Z)*) – A list of truples. Here X is the index of the donor ion, Y is the proton involved in the proton transfer and Z is the acceptor
- **verbose** (*bool*) – If true (default is False) it prints some debugging stuff (like the number of processors thorugh which the process is spanned)

**proton_transfers** [list of floats] list of the $\nu$ proton transfer coordinates.

`Manipulate.`**`PlotRamanSpectra`**(*w_axis*, *T*, *sigma*, *dyn*, *pol1=None*, *pol2=None*)

This function computes the Raman spectrum of the dynamical matrix in the harmonic approximation. Note: it requires that the dynamical matrix to have a Raman spectrum.

If Pol1 and pol2 are none the light is chose unpolarized

**Parameters**

- **w_axis** (*ndarray*) – The axis of the Raman shift (cm-1)
- **T** (*float*) – Temperature
- **sigma** (*float*) – The beam frequency standard deviation
- **dyn** (*Phonons.Phonons()*) – The dynamical matrix

- **pol1** (*ndarray 3*) – The incident polarization vector
- **pol2** (*ndarray 3*) – The scattered polarization vector

**I(w)** [ndarray] The intensity for each value of the w_axis in input

`Manipulate.`**`QHA_FreeEnergy`**(*ph1*, *ph2*, *T*, *N_points=2*, *return_interpolated_dyn=False*)

This function computes the quasi harmonic approximation interpolating the dynamical matrices between two different given. It will return as an output a matrix of free energies, at several N_points - 1 in between the two phonon matrices given and at each temperature specified. The free energy is computed then diagonalizing all the dynamical matrices in between by using the equation:

$$F(T) = \sum_\mu \frac{\hbar\omega_\mu}{2} + k_bT\ln\left(1 - e^{-\frac{\hbar\omega}{k_bT}}\right)$$

The interpolation is done on the dynamical matrix itself, not on the frequencies, as in this way it is possible to avoid the problems that wuold have occurred if two parameters where exchanged.

**Parameters**

- **ph1** (*–*) – This is the first dynamical matrix through which the interpolation is performed.
- **ph2** (*–*) – This is the second dynamical matrix. Must share the same number of atoms and q points with ph1, otherwise a ValueError exception is raised.
- **T** (*–*) – Numpy array of the temperatures on which you wan to compute the Free energy. (Kelvin)
- **N_points** (*–*) – The number of points for the interpolation. If 2 only the original ph1 and ph2 will used. Note that any number lower then 2 will rise a ValueError exception.
- **return_interpolated_dyn** (*–*) – If true the interpolated dynamical matrix will be returned as well

- **free_energy** [ndarray ( (N_points)x len(T) )] The matrix containing the free energy on all the points in between the interpolation at the temperature given as the input array. It is in Ry and in the unit cell
- **interp_dyns** [list (Phonons()), len(N_points) [Only if asked]] Only if return_interpolated_dyn is True, also the interpolated dyns are returned

`Manipulate.`**`RotationTranslationStretching`**(*structure*, *molecules*, *indices*, *vector*)

Project the given atomic vector into the molecular rotation, translations and stretching. This is an usefull subroutine to analyze the vibrational modes of molecular crystals.

**Parameters**

- **structure** (`Structure`) – The atomic structure.
- **molecules** (*list of Structure*) – A list of structures that identifies the molecules.
- **indices** (*list of truples of int*) – A list containing the indices of the atoms in the structure that belong to the respective molecule.
- **vector** (*ndarray(size = (N_atoms * 3), dtype = float64)*) – The polarization vector that you want to analyze.

**rotations** [float] The fraction of the polarization projected on the molecular rotations

**translations** [float] The fraction of the polarization projected on the molecular translations

**stretching** [float] The fraction of the polarization projected on the molecular stretching

`Manipulate.`**`SaveXYZTrajectory`**(*fname*, *atoms*, *comments=[]*)

This function save on a file a list of structures containing the frames of the animation in the xyz file from fname.

- **fname** [string] Path to the xyz animation file
- **atoms** [list] A list of the Structure() object to be written
- **comments: list, optional** A list of comments for each frame.

`Manipulate.`**`TransformStructure`**(*dyn1*, *dyn2*, *T*, *structures*, *mode_exchange=None*, *mode_sign=None*)

This function transforms a set of randomly generated structures from the matrix dyn1 to one generated accordingly to the dynamical matrix dyn2, at the temperature T.

The transformation take place using the matrix:

$$T_{ba} = \sqrt{\frac{M_a}{M_b}}\sum_\mu \sqrt{\left(\frac{1+2n_\mu^{(1)}}{1+2n_\mu^{(0)}}\right)\frac{\omega_\mu^{(0)}}{\omega_\mu^{(1)}}} e_\mu^{b\,(1)} e_\mu^{a\,(0)}$$

Where $n_\mu$ are the bosonic occupation number of the $\mu$ mode, (0) and (1) refers respectively to the starting and ending dynamical matrices and $e_\mu^a$ is the a-th cartesian coordinate of the polarization vector associated to the $\mu$ mode.

The mode_exchange parameters can be used to associate the modes of the dyn1 matrix to the corresponding mdoe of the dyn2. They are by default ordered with increasing phonon frequencies. The translational modes are neglected by the summation.

NOTE: for now only Gamma space dynamical matrices are supported.

**Parameters**

- **`dyn1`** (`Phonons.Phonons`) – The dynamical matrix used to generate the structure
- **`dyn2`** (`Phonons.Phonons`) – The dynamical matrix of the target structure
- **`T`** (*`float`*) – The temperature of the transformation
- **`structure`** (*`list of ndarray or Structure.Structur()`*) – The original structures to be transformed. If the type is the numpy ndarray, then they are interpreted as displacement respect to the average positions defined by the structure of the dynamical matrices. In this case an ndarray (or a list of ndarray) is given in output.
- **`mode_exchange`** (*`list (3*N_atoms, int), optional`*) – If present the modes order of dyn1 is exchanged when transforming the structure to dyn2. This is usefull to optimize the transformation along many possible degenerate systems. This is applied before the mode_sign, if any
- **`mode_sign`** (*`list, optional`*) – If different from none, is a list of length equal 3*N_atoms, containing only -1 and 1. The polarization vector of the first dynmat will be changed of sign if requested. This is applied after the mode_exchange

**Returns** The transformed structure, or a list of structures, depending on the input.

**Return type** *Structure.Structure*

`Manipulate.`**`apply_symmetry_on_fc`**(*structure*, *fc_matrix*, *symmetry*)

This functio apply the given symmetry on the force constant matrix. The original structure must satisfy the symmetry, and it is used to get the atoms transformed one into the other.

The application of the symmetries follow the following rule.

The symmetry check is performed by comparing the two force constant matrix within the given threshold.

$$\Phi_{s(a)s(b)}^{ij} = \sum_{h,k=1}^{3} S_{ik} S_{jh} \Phi_{ab}^{kh}$$

$$\Phi = S\Phi S^\dagger$$

where $s(a)$ is the atom in which the $a$ atom is mapped by the symmetry.

Note: this work only in the gamma point, no phase for q != 0 are supported. Look at the documentation of the ApplySymmetry in the Phonon package

**Parameters**

- **`structure`** (*`CC.Structure()`*) – The structure satisfying the symmetry, it is used to get the atoms -> S(atoms) mapping
- **`fc_matrix`** (*`ndarray (3N x 3N)`*) – The force constant matrix to be applyed the symmetry on
- **`symmetry`** (*`ndarray (3x4)`*) – The symmetry operation as a 3x3 rotational. The last column is the fractional translation if any.

**new_fc** [ndarray (3N x 3N)] The result of the application of the symmetry on the original dynamical matrix.

CHAPTER

NINE

# THE SYMMETRY CLASS

This is the symmetry class, used to perform symmetry operation. It can be used to enforce symmetries on dynamical matrix, vectors, 3 or 4 rank tensors. It can apply symmetries both in real space (slow) and in q space.

To recognize symmetries it has a builtin Fortran code taken from QuantumESPRESSO software suite. If symmetries are used in real-space it is possible to use spglib instead.

**class** `symmetries.`**`QE_Symmetry`**(*structure*, *threshold=1e-05*)

**`ApplyQStar`**(*fcq*, *q_point_group*)

Given the fc matrix at each q in the star, it applies the symmetries in between them.

**Parameters**

- **`fcq`** (–) – The dynamical matrices for each q point in the star
- **`q_point_group`** (–) – The q vectors that belongs to the same star

**`ApplySymmetriesToV2`**(*v2*, *apply_translations=True*)

This subroutines applies the symmetries to a 2-rank tensor. Usefull to work with supercells.

**Parameters**

- **`v2`** (*`ndarray (size = (3*nat, 3*nat), dtype = np.double)`*) – The 2-rank tensor to be symmetrized. It is directly modified
- **`apply_translation`** (*`bool`*) – If false pure translations are neglected.

**`ApplySymmetryToEffCharge`**(*eff_charges*)

This subroutine applies the symmetries to the effective charges.

As always, the eff_charges will be modified by this subroutine.

**Parameters** **`eff_charges`** (–) – The effective charges tensor. The first dimension is the index of the atom in the primitive cell the second index is the electric field. The third index is the cartesian axis.

**`ApplySymmetryToMatrix`**(*matrix*, *err=None*)

Apply the symmetries to the 3x3 matrix. It can be a stress tensor, a dielectric tensor and so on.

**Parameters** **`matrix`** (*`a 3x3 matrix`*) – The matrix to which you want to apply the symmetrization. The matrix is overwritten with the output.

**`ApplySymmetryToRamanTensor`**(*raman_tensor*)

This subroutine applies the symmetries to the raman tensor

As always, the raman_tensor will be modified by this subroutine.

**Parameters** **`raman_tensor`** (–) – The raman tensor. The first two indices indicate the polarization of the incoming/outcoming field, while the last one is the atomic/cartesian coordinate

**ApplySymmetryToSecondOrderEffCharge**(*dM_drdr*, *apply_asr=True*)
This subroutine applies simmetries to the two phonon effective charges.

Note, to symmetrize this tensor, symmetries must be imposed on the supercell.

**Parameters**

- **dM_drdr** (*ndarray (size = (3 nat_sc, 3nat_sc, 3))*) – The derivative of effective charges.
- **apply_asr** (*bool*) – If True the sum rule is applied. The sum rule is the 'custom' one where translations are projected out from the space for each polarization components.

**ApplySymmetryToTensor3**(*v3*, *initialize_symmetries=True*)
This subroutines uses the current symmetries to symmetrize a rank-3 tensor. This tensor must be in the supercell space.

The v3 argument will be overwritten.

NOTE: The symmetries must be initialized in the supercell using spglib

**Parameters**

- **v3** (*ndarray( size=(3*nat, 3*nat, 3*nat), dtype = np.double, order = "F")*) – The 3-rank tensor to be symmetrized. It will be overwritten with the new symmetric one. It is suggested to specify the order of the array to "F", as this will prevent the parser to copy the matrix when doing the symmetrization in Fortran.
- **initialize_symmetries** (*bool*) – If True the symmetries will be initialized using spglib. Otherwise the already present symmetries will be use. Use it False at your own risk! (It can crash with seg fault if symmetries are not properly initialized)

**ApplySymmetryToTensor4**(*v4*, *initialize_symmetries=True*)
This subroutines uses the current symmetries to symmetrize a rank-4 tensor. This tensor must be in the supercell space.

The v4 argument will be overwritten.

NOTE: The symmetries must be initialized in the supercell using spglib

**Parameters**

- **v4** (*ndarray( size=(3*nat, 3*nat, 3*nat, 3*nat), dtype = np.double, order = "F")*) – The 4-rank tensor to be symmetrized. It will be overwritten with the new symmetric one. It is suggested to specify the order of the array to "F", as this will prevent the parser to copy the matrix when doing the symmetrization in Fortran.
- **initialize_symmetries** (*bool*) – If True the symmetries will be initialized using spglib. Otherwise the already present symmetries will be use. Use it False at your own risk! (It can crash with seg fault if symmetries are not properly initialized)

**ApplyTranslationsToVector**(*vector*)
This subroutine applies the translations to the given vector. To be used only if the structure is a supercell structure and the symmetries have been initialized with SPGLIB

**Parameters** **vector** (*size (nat, 3)*) – A vector that must be symmetrized. It will be overwritten.

**ChangeThreshold**(*threshold*)
Change the symmetry threshold sensibility

**ForceSymmetry**(*structure*)
Force the symmetries found at a given threshold to be satisfied also in a lower threshold.

This use the irt trick

**GetQIrr**(*supercell*)

This method returns a list of irreducible q points given the supercell size.

**Parameters** **supercell** (*(X, Y, Z) where XYZ are int*) – The supercell size along each unit cell vector.

**Returns** **q_irr_list** – The list of irreducible q points in the brilluin zone.

**Return type** list of q vectors

**GetQStar**(*q_vector*)

Given a vector in q space, get the whole star. We use the quantum espresso subrouitine.

**Parameters** **q_vector** (*ndarray(size= 3, dtype = np.float64)*) – The q vector

**q_star** [ndarray(size = (nq_star, 3), dtype = np.float64)] The complete q star

**GetSymmetries**(*get_irt=False*)

This method returns the symmetries in the CellConstructor format from the ones elaborated here.

**Parameters** **get_irt** (*bool*) – If true (default false) also the irt are returned. They are the corrispondance between atoms for each symmetry operation.

**list :** List of 3x4 ndarray representing all the symmetry operations

**irt** [ndarray(size=(nsym, nat), dtype = int), optional] Returned only if get_irt = True. It is the corrispondance between atoms after the symmetry operation is applied. irt[x, y] is the atom mapped into y by the x symmetry.

**GetUniqueRotations**()

This subroutine returns an alternative symmetries that contains only unique rotations (without fractional translations). This is usefull if the peculiar cell is a supercell and the symmetrization was performed with SPGLIB

**Returns**

- **QE_s** (*ndarray(size = (3,3,48), dtype = np.intc)*) – The symmetries
- **QE_invs** (*ndarray(size = 48, dtype = np.intc)*) – The index of the inverse symmetry
- **QE_nsym** (*int*) – The number of symmetries

**ImposeSumRule**(*force_constant*, *asr='simple'*, *axis=1*, *zeu=None*)

This subroutine imposes on the given force constant matrix the acustic sum rule

**Parameters**

- **force_constnat** (*3xnat , 3xnat*) – The force constant matrix, it is overwritten with the new one after the sum rule has been applied.
- **asr** (*string, optional, default = 'custom'*) – One of 'custom', 'simple', 'crystal', 'one-dim' or 'zero-dim'. For a detailed explanation look at the Quantum ESPRESSO documentation. The custom one, default, is implemented in python as CustomASR. No ASR is imposed on the effective charges in this case.
- **axis** (*int, optional*) – If asr = 'one-dim' you must set the rotational axis: 1 for x, 2 for y and 3 for z. Ohterwise it is unused.
- **zeu** (*ndarray (N_atoms, 3, 3), optional*) – If different from None, it is the effective charge array. As the force_constant, it is updated.

**InitFromSymmetries**(*symmetries*, *q_point=array([0, 0, 0])*)
This function initialize the QE symmetries from the symmetries expressed in the Cellconstructor format, i.e. a list of numpy array 3x4 where the last column is the fractional translation.

**TODO: add the q_point preparation by limitng the symmetries only to** those that satisfies the specified q_point

**PrintSymmetries**()
This method just prints the symmetries on stdout.

**SelectIrreducibleQ**(*q_vectors*)
This methods selects only the irreducible q points given a list of total q points for the structure.

**Parameters** **q_vectors** (*list of q points*) – The list of q points to be polished fromt he irreducible

**q_irr** [list of q points] The q_vectors without the copies by symmetry of the dynamical matrix.

**SetupFromSPGLIB**()
This function uses spglib to find symmetries, recognize the supercell and setup all the variables to perform the symmetrization inside the supercell.

NOTE: If spglib cannot be imported, an ImportError will be raised

**SetupQPoint**(*q_point=array([0., 0., 0.])*, *verbose=False*)
Get symmetries of the small group of q

Setup the symmetries in the small group of Q.

**Parameters**

- **q_point** (*ndarray*) – The q vector in reciprocal space (NOT in crystal axes)
- **verbose** (*bool*) – If true the number of symmetries found for the bravais lattice, the crystal and the small group of q are written in stdout

**SetupQStar**(*q_tot*, *supergroup=False*)
This method divides the given q point list into the star. Remember, you need to pass the whole number of q points

**Parameters**

- **q_tot** (*list*) – List of q vectors to be divided into stars
- **supergroup** (*bool*) – If true then assume we have initialized a supercell bigger

**q_stars** [list of lists] The list of q_star (list of q point in the same star).

**sort_mask** [ndarray(size=len(q_tot), dtype = int)] a mask to sort the q points in order to match the same order than the q_star

**SymmetrizeDynQ**(*dyn_matrix*, *q_point*)
Use the Quantum ESPRESSO fortran code to symmetrize the dynamical matrix at the given q point.

NOTE: the symmetries must be already initialized.

**Parameters**

- **dyn_matrix** (*ndarray (3nat x 3nat)*) – The dynamical matrix associated to the specific q point (cartesian coordinates)
- **q_point** (*ndarray 3*) – The q point related to the dyn_matrix.

The input dynamical matrix will be modified by the current code.

**SymmetrizeFCQ** (*fcq*, *q_stars*, *verbose=False*, *asr='simple'*)

Use the current structure to impose symmetries on a complete dynamical matrix in q space. Also the simple sum rule at Gamma is imposed

**Parameters**

- **fcq** (*–*) – The q space force constant matrix to be symmetrized (it will be overwritten)
- **q_stars** (*–*) – The list of q points divided by stars, the fcq must follow the order of the q points in the q_stars array

**SymmetrizeVector** (*vector*)

This is the easier symmetrization of a generic vector. Note, fractional translation and generic translations are not imposed. This is because this simmetrization acts on displacements and forces.

**Parameters** **vector** (*ndarray(natoms, 3)*) – This is the vector to be symmetrized, it will be overwritten with the symmetrized version

Besides the QE_Symmetry class, other methods are available in the symmetries module to operate directly on symmetries.

Created on Fri Sep 29 11:10:21 2017

@author: darth-vader

symmetries.**ApplySymmetriesToVector** (*symmetries*, *vector*, *unit_cell*, *irts*)

Apply the symmetry to the given vector of displacements. Translations are neglected.

$$\vec{v'}[irt] = S\vec{v}$$

**Parameters**

- **symmetries** (*list of ndarray(size = (3,4))*) – The symmetries operation (crystalline coordinates)
- **vector** (*ndarray(size = (nat, 3))*) – The vector to which apply the symmetry. In cartesian coordinates
- **unit_cell** (*ndarray( size = (3,3))*) – The unit cell in which the structure is defined
- **irts** (*list of ndarray(nat, dtype = int)*) – The index of how the symmetry exchanges the atom.

symmetries.**ApplySymmetryToVector** (*symmetry*, *vector*, *unit_cell*, *irt*)

Apply the symmetry to the given vector of displacements. Translations are neglected.

$$\vec{v'}[irt] = S\vec{v}$$

**Parameters**

- **symmetry** (*ndarray(size = (3,4))*) – The symmetry operation (crystalline coordinates)
- **vector** (*ndarray(size = (nat, 3))*) – The vector to which apply the symmetry. In cartesian coordinates
- **unit_cell** (*ndarray( size = (3,3))*) – The unit cell in which the structure is defined
- **irt** (*ndarray(nat, dtype = int)*) – The index of how the symmetry exchanges the atom.

`symmetries.`**`ApplyTranslationsToSupercell`**(*fc_matrix*, *super_cell_structure*, *supercell*)

Impose the translational symmetry directly on the supercell matrix.

**Parameters**

- **`fc_matrix`** (–) – The matrix in the supercell. In output will be modified
- **`super_cell_structure`** (–) – The structure of the super cell
- **`supercell`** (–) – The dimension of the supercell.

`symmetries.`**`CheckSupercellQ`**(*unit_cell*, *supercell_size*, *q_list*)

This subroutine checks that the given q points of a dynamical matrix matches the desidered supercell. It is usefull to spot bugs like the wrong definitions of alat units, or error not spotted just by the number of q points (confusion between 1,2,2 or 2,1,2 supercell).

**Parameters**

- **`unit_cell`** (*`ndarray(size=(3,3), dtype = np.float64)`*) – The unit cell, rows are the vectors
- **`supercell_size`** (*`ndarray(size=3, dtype = int)`*) – The dimension of the supercell along each unit cell vector.
- **`q_list`** (*`list of vectors`*) – The total q point list.

**Returns** **is_ok** – True => No error False => Error

**Return type** bool

`symmetries.`**`CustomASR`**(*fc_matrix*)

This function applies a particular sum rule. It projects out the translations exactly.

**Parameters** **`fc_matrix`** (*`ndarray(3nat x 3nat)`*) – The force constant matrix. The sum rule is applied on that.

`symmetries.`**`ExcludeRotations`**(*fc_matrix*, *structure*)

We exclude the rotations from the force constant matrix.

**Parameters**

- **`fc_matrix`** (*`ndarray(3*nat, 3*nat)`*) – The force constant matrix
- **`structure`** (*`Structure()`*) – The structure that is identified by the force constant matrix

`symmetries.`**`GetIRT`**(*structure*, *symmetry*, *timer=<cellconstructor.Timer.Timer object>*, *debug=False*)

Get the irt array. It is the array of the atom index that the symmetry operation swaps.

the y-th element of the array (irt[y]) is the index of the original structure, while y is the index of the equivalent atom after the symmetry is applied.

**Parameters**

- **`structure`** (*`Structure.Structure()`*) – The unit cell structure
- **`symmetry`** (*`list of 3x4 matrices`*) – symmetries with frac translations
- **`timer`** (*`Timer class`*) – The functions will be timed using the timer object.

`symmetries.`**`GetISOTROPYFindSymInput`**(*structure*, *title='Prepared with Cellconstructor'*, *latticeTolerance=1e-05*, *atomicPositionTolerance=0.001*)

As the method PrepareISOTROPYFindSymInput, but the input is returned as a list of string (lines).

`symmetries.`**`GetNewQFromUnitCell`**(*old_cell*, *new_cell*, *old_qs*)

This method returns the new q points after the unit cell is changed. Remember, when changing the cell to mantain the same kind (cubic, orthorombic, hexagonal…) otherwise the star identification will fail.

The q point are passed (and returned) in cartesian coordinates.

**Parameters**

- **structure** (`Structure.Structure()`) – The structure to be changed (with the old unit celll)
- **new_cell** (*ndarray(size=(3,3), dtype = np.float64)*) – The new unit cell.
- **old_qs** (*list of ndarray(size=3, dtype = np.float64)*) – The list of q points to be converted

**Returns** **new_qs** – The list of the new q points adapted in the new cell.

**Return type** list of ndarray(size=3, dtype = np.float64)

symmetries.**GetQForEachMode**(*pols_sc*, *unit_cell_structure*, *supercell_structure*, *supercell_size*, *crystal=True*)

For each polarization mode in the supercell computes the corresponding q vector.

Indeed the polarization vector will be a have components both at q and at -q.

If a polarization vector mixes two q an error will be raised.

NOTE: use DiagonalizeSupercell of Phonons to avoid mixing q.

**Parameters**

- **pols_sc** (*ndarray ( size = (3*nat_sc, n_modes), dtype = np.float64)*) – The polarization vector of the supercell (real)
- **unit_cell_structure** (`Structure()`) – The structure in the unit cell
- **supercell_structure** (`Structure()`) – The structure in the super cell
- **supercell_size** (*list of 3 int*) – The supercell
- **crystal** (*bool*) – If True, q points are returned in cristal coordinates.

**q_list** [ndarray(size = (n_modes, 3), dtype = np.float, order = "C")] The list of q points associated with each polarization mode. If crystal is true, they will be in crystal coordinates.

symmetries.**GetQGrid**(*unit_cell*, *supercell_size*, *enforce_gamma_first=True*)

This method gives back a list of q points given the reciprocal lattice vectors and the supercell size.

**Parameters**

- **unit_cell** (*ndarray(size=(3,3), dtype = np.float64)*) – The unit cell, rows are the vectors
- **supercell_size** (*ndarray(size=3, dtype = int)*) – The dimension of the supercell along each unit cell vector.
- **enforce_gamma_first** (*bool*) – If true, the Gamma point is the first one of the list.

**Returns** **q_list** – The list of q points, of type ndarray(size = 3, dtype = np.float64)

**Return type** list

symmetries.**GetQGrid_old**(*unit_cell*, *supercell_size*)

This method gives back a list of q points given the reciprocal lattice vectors and the supercell size.

**Parameters**

- **unit_cell** (*ndarray(size=(3,3), dtype = np.float64)*) – The unit cell, rows are the vectors

- **supercell_size** (*ndarray(size=3, dtype = int)*) – The dimension of the supercell along each unit cell vector.

**Returns** **q_list** – The list of q points, of type ndarray(size = 3, dtype = np.float64)

**Return type** list

`symmetries.`**`GetSupercellFromQlist`**(*q_list*, *unit_cell*)

This method returns the supercell size from the list of q points and the unit cell of the structure.

**Parameters**

- **q_list** (*list*) – List of the q points in cartesian coordinates
- **unit_cell** (*ndarray(3,3)*) – Unit cell of the structure (rows are the unit cell vectors)

**supercell_size** [list of 3 integers] The supercell dimension along each unit cell vector.

`symmetries.`**`GetSymmetriesFromSPGLIB`**(*spglib_sym*, *regolarize=False*)

This module comvert the symmetry fynction from the spglib format.

**Parameters**

- **spglib_sym** (*dict*) – Result of spglib.get_symmetry( … ) function
- **regolarize** (*bool, optional*) – If True it rewrites the translation to be exact. Usefull if you want to constrain the symmetry exactly

**Returns** **symmetries** – A list of 4x3 matrices containing the symmetry operation

**Return type** list

`symmetries.`**`GetSymmetriesOnModes`**(*symmetries*, *structure*, *pol_vects*, *timer=None*, *debug=False*)

This methods returns a set of symmetry matrices that explains how polarization vectors interacts between them through any symmetry operation.

**Parameters**

- **symmetries** (*list*) – The list of 3x4 matrices representing the symmetries.
- **structure** (*Structure.Structure()*) – The structure (supercell) to allow the symmetry to correctly identify the atoms that transforms one in each other.
- **pol_vects** (*ndarray(size = (n_dim, n_modes))*) – The array of the polarization vectors (must be real)

**pol_symmetries** [ndarray( size=(n_sym, n_modes, n_modes))] The symmetry operation between the modes. This allow to identify which mode will be degenerate, and which will not interact.

`symmetries.`**`GetSymmetryMatrix`**(*sym*, *structure*, *crystal=False*)

This subroutine converts the 3x4 symmetry matrix to a 3N x 3N matrix. It also transform the symmetry to be used directly in cartesian space. However, take care, it could be a very big matrix, so it is preverred to work with the small matrix, and maybe use a fortran wrapper if you want speed.

NOTE: The passe structure must already satisfy the symmetry

**Parameters**

- **sym** (*ndarray(size = (3, 4))*) – The symmetry and translations
- **structure** (*CC.Structure.Structure()*) – The structure on which the symmetry is applied (The structure must satisfy the symmetry already)
- **crystal** (*bool*) – If true, the symmetry is returned in crystal coordinate (default false)

sym_mat : ndarray(size = (3*structure.N_atoms, 3*structure.N_atoms))

`symmetries.`**`PrepareISOTROPYFindSymInput`**(*structure*, *path_to_file='findsym.in'*, *title='Prepared with Cellconstructor'*, *latticeTolerance=1e-05*, *atomicPositionTolerance=0.001*)

This method can be used to prepare a suitable input file for the ISOTROPY findsym program.

**Parameters**

- **path_to_file** (*string*) – A valid path to write the findsym input.
- **title** (*string, optional*) – The title of the job

`symmetries.`**`get_degeneracies`**(*w*)

From the given frequencies, for each mode returns a list of the indices of the modes of degeneracies.

**Parameters** **w** (*ndarray (n_modes)*) – Frequencies

**deg_list** [list of lists] A list that contains, for each mode, the list of the modes (indices) that are degenerate with the latter one

`symmetries.`**`get_diagonal_symmetry_polarization_vectors`**(*pol_sc*, *w*, *pol_symmetries*)

This function is very usefull to have a complex basis in which the application of symmetries is trivial.

In this basis, each symmetry is diagonal. Indeed this forces the polarization vectors to be complex in the most general case.

NOTE: To be tested, do not use for production run It seems to be impossible to correctly decompose simmetries when we have multiple rotations.

If the symmetries are not unitary, an exception will be raised.

**Parameters**

- **pol_sc** (*ndarray (3*nat, n_modes)*) – The polarizaiton vectors in the supercell (obtained by DiagonalizeSupercell of the Phonon class)
- **w** (*ndarray (n_modes)*) – The frequency for each polarization vectors
- **pol_symmetries** (*ndarray (N_sym, n_modes, n_modes)*) – The Symmetry operator that acts on the polarization vector

**pol_vects** [ndarray(3*nat, n_modes)] The new (complex) polarization vectors that diagonalizes all the symmetries.

**syms_values** [ndarray(n_modes, n_sym)] The (complex) unitary eigenvalues of each symmetry operation along the given mode.

`symmetries.`**`get_invs`**(*QE_s*, *QE_nsym*)

For each symmetry operation, get an index that its inverse Note, the array must be in Fortran indexing (starts from 1)

**Parameters**

- **QE_s** (*ndarray (size = (3,3,48), dtype = np.intc)*) – The symmetries
- **QE_nsym** (*int*) – The number of symmetries

**QE_invs** [ndarray(size = 48, dtype = np.intc)] The index of the inverse symmetry. In fortran indexing (1 => index 0) `symmetries.`**`get_symmetries_from_ita`**`(`*ita*, *red=False*`)`

This function returns a matrix containing the symmetries from the given ITA code of the Group. The corresponding ITA/group label can be found on the Bilbao Crystallographic Server.

**Parameters**

- **`ita`** (–) – The ITA code that identifies the group symmetry.
- **`red`** (–) – If red is True then load the symmetries only in the smallest unit cell (orthorombic)

- **symmetries** [list] A list of 3 rows x 4 columns matrices (ndarray), containing the symmetry operations of the chosen group.

CHAPTER

# TEN

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX