CellConstructor Documentation

Release 1.0

Lorenzo Monacelli

Mar 02, 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:

```python
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:

```python
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:

```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:

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:

```
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

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 ω_μ^2 and e_μ are, respectively, eigenvalues and eigenvectors of the original Φ 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
```

(continues on next page)

(continued from previous page)

```
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))

# 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
```

(continues on next page)

(continued from previous page)

```
 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
  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.

```python
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)
```

(continues on next page)

(continued from previous page)

```
# 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))

# 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

# 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
```

(continues on next page)

(continued from previous page)

```
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])
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:

```python
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

```python
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

```python
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.

```python
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)
```

(continues on next page)

(continued from previous page)

```
# -- 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")

# 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.

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

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

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.

```python
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_MAX_OH = 2
R_MAX_HH = 3

# The thickness of the shell
DR = 0.025

# 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...")
```

(continues on next page)

(continued from previous page)

```python
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_MAX_OH, DR)
print ("Computing HH g(r)...")
grHH = CC.Methods.get_gr(structures, "H", "H", R_MAX_HH, DR)

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

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_max`** (–) – The maximum cutoff value of r
- **`dr`** (–) – The bin value of the distribution.

- **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 the function with capital letters for new words, for example:

```
def HelloWorld(x):
   # Very good
   pass
```

   Not:

```
def hello_world(x):
   # NO!
   pass
```

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.

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")]
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
    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 into the `scripts/cellconstructor_test.py` 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.

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 α_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 α_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_max*, *dr*)

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

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

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_max`** (–) – The maximum cutoff value of r
- **`dr`** (–) – The bin value of the distribution.

- **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.

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.

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.

Besides the QE_Symmetry class, other methods are available in the symmetries module to operate directly on symmetries.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX