SSCHA

Stochastic Self-Consistent
Harmonic Approximation

SSCHA SCHOOL 2023

R. Bianco, D. Dangic, I. Errea, G. Marchese,

G. Marini, D. Martinez, L. Monacelli

June 27, 2023

CONTENTS:

CHAPTER

ONE

SOFTWARE INSTALLATION

This installation guide here is valid for Linux machines as well as MacOS with intel processors.

Users of Windows can use the virtual machine provided by Quantum Mobile available here. The instructions to install the this virtual machine are provided in that link. However, you will need to increase the RAM allocated to the virtual machine to at least 4Gb to be able to run all the tutorials on it. Once you have installed the virtual machine, follow the instructions below to install all the needed software to work on the tutorials.

If your laptop is a new Mac with the M1 or M2 processors, the installation guides below and the virtual machine provided in the previous paragraph may not work. If you are in this situation, we recommend that you follow the instructions in *Setting up a virtual machine on a Mac with M1/M2*.

1.1 Requirements

Most of the codes require a fortran or C compiler and MPI configured. Here we install all the requirements to properly setup the SSCHA code. To properly compile and install the SSCHA code, you need a fortran compiler and LAPACK/BLAS available.

On Debian-based Linux distribution, all the software required is installed with (Tested on Quantum Mobile and ubuntu 20.04):

```
sudo apt update
sudo apt install libblas-dev liblapack-dev liblapacke-dev gfortran openmpi-bin
```

Note that some of the name of the library may change slightly in different linux versions or on MacOS.

1.1.1 Python installation

SSCHA is a Python library and program. Most linux distribution come with python already installed, however, for a performance boost, it is usually better to use the python distribution provided by the anaconda environment. The Quantum Mobile virtual machine already comes with anaconda installed. In case you are running on your own machine, anaconda python can be downloaded and installed from [www.anaconda.com/download]

Once you installed the software, at the beginning of your terminal you should see a

```
(base) $
```

The (base) identify the current anaconda environment. It may be necessary to restart the terminal after the installation to have Anaconda start properly.

If you are **not** using anaconda but the default python from the linux distribution, it may be necessary to install the python header files to correctly compile the SSCHA extension

```
sudo apt install python-dev
```

1.1.2 Install python packages

Most of the code can be easily installed with the following pip command:

```
pip install ase spglib quippy-ase
```

The Atomic Simulation Environment (ASE), employed to read and write structure files. SPGLIB is used to recognize the space-group and perform symmetry anaylisys. Quippy is employed to train a machine-learning force field.

1.2 SSCHA

Once the prerequisites have been installed, python-sscha can be downloaded and installed with

```
pip install cellconstructor python-sscha
```

Alternatively, it is possible to use the most recent version from the github repository [https://github.com/SSCHAcode], under CellConstructor and python-sscha repositories.

The installation is performed in this case with

```
python setup.py install
```

1.2.1 Personalize the compiler

If you have multiple compilers installed, and want to force pip to employ a specific fortran compile, you can specify its path in the FC environment variable. Remember that the compiler employed to compile the code should match with the linker, indicated in the LDSHARED variable.

For example

```
FC=gfortran LDSHARED=gfortran pip install cellconstructor python-sscha
```

For the development version of the code, subtitute the pip call with the python setup.py install.

1.2.2 Running the testsuite

To be sure everything is working, you can run the testsuite. Make sure to install the pytest package with

```
pip install pytest
```

Then run the testsuite with

```
cellconstrutor_test.py
```

If it works without errors, then the code has been correctly installed.

1.3 TDSCHA

As for the SSCHA code, also TDSCHA is distributed on PyPi

```
pip install tdscha
```

Alternatively, the code to compute Raman and IR spectrum can be downloaded from GitHub at [https://github.com/SSCHAcode/tdscha]

To install the github code, that enables the MPI parallelization also without the JULIA speedup, you can use:

```
git clone https://github.com/SSCHAcode/tdscha.git
cd tdscha
MPICC=mpicc python setup.py install
```

where mpicc is a valid mpi c compiler (the specification of MPICC can be dropped, but parallelization will not be available aside for the julia mode discussed below).

1.3.1 JULIA speedup enhancement

The TDSCHA code exploits JULIA to speedup the calculation by a factor of 10x-15x with the same number of processors.

To have it working, download and install julia from [https://julialang.org/downloads/]. Alternatively, to install julia on linux we can employ juliaup:

```
curl -fsSL https://install.julialang.org | sh
```

Hit enter when asked to install julia.

To use julia, either open a new terminal, or hit:

```
source ~/.bashrc
```

Then, open a terminal and type `julia`. Inside the julia prompt, type `]` The prompt should change color and display the julia version ending with `pkg>`

Install the required julia libraries

```
pkg> add SparseArrays, LinearAlgebra, InteractiveUtils, PyCall
```

This should install the required libraries. press backspace to return to the standard julia prompt and exit with

```
julia> exit()
```

Then, install the python bindings for julia with

```
pip install julia
```

Now, you should be able to exploit the julia speedup in the TDSCHA calculations. It is not required to install julia before TDSCHA, it can also be done in a later moment.

1.3.2 MPI Parallelization

MPI parallelization is not necessary for the tutorial, however you may like to configure it in practical calculation to further speedup the code. For production runs, it is suggested to combine the mpi parallelization with the julia speedup.

The TDSCHA code exploits the mpi parallelization using mpi4py, This assumes that you have a MPI C compiler installed. This is done by installing the library `openmpi-bin` which we installed in the requirements.

You can now install mpi4py

```
pip install mpi4py
```

The parallelization is automatically enabled in the julia version and if mpi4py is available. However, to run the parallel code without the julia speedup, you need to recompile the code from the github repository as (not the version installed with pip)

```
MPICC=mpicc python setup.py install
```

e sure that at the end of the installation no error are displayed, and the write PARALLEL ENVIRONMENT DECTECTED SUCCESFULLY is displayed. Note that, if using the julia enhanced version, the last command is not required, and you can install only mpi4py.

1.4 Install qe-5.1.0_elph

In order to install this old version of Quantum Espresso, which is tuned to allow the combination of electron-phonon matrix elements with SSCHA dynamical matrices, follow these instructions:

```
git clone https://github.com/SSCHAcode/qe-5.1.0_elph.git
cd qe-5.1.0_elph
./configure
make all
```

It may happen that the compilation fails with a message like

```
Error: Rank mismatch between actual argument at...
```

In this case you need to edit the make.sys file with the following command

```
sed -i "s/FFLAGS         = -O3 -g/FFLAGS         = -O3 -g -fallow-argument-
↪mismatch/g" make.sys
```

and rerun

```
make all
```

again.

1.5 EPIq

The EPIq code is hosted in a git repository. The last stable version can be downloaded here.

Once the source code has been downloaded, unzip the archive and enter the epiq main folder (`cd epiq`). EPIq has very few prerequisites:

- `BLAS` and `LAPACK` libraries.
- Any MPI fortran compiler (e.g. `mpif90` for `openmpi`).

Then compile *EPIq*. Enter in the source directory and run `make` as:

```
cd epiq
make all
```

In some cases (like in quantum mobile), the compilation may fail. If it fails with error:

```
gfortran: error: unrecognized command line option ‘-fallow-argument-mismatch’;
↪ did you mean ‘-Wno-argument-mismatch’?
```

This can be fixed replacing `-fallow-argument-mismatch` with `-Wno-argument-mismatch` in the *make.sys* file. This can be done automatically with the following command:

```
sed -i 's/-fallow-argument-mismatch/-Wno-argument-mismatch/g' make.sys
```

Then run again `make all`.

If everything went smoothly, an executable file named epiq.x will be created in the `bin` folder. If the compilation was not successful, this probabily means that the `configure` could not find the necessary libraries/compiler. You should manually modify the *make.sys* file in order to correctly locate them.

1.6 fermisurfer installation

Fermisurfer is a program for the visualization of Fermi surface resolved physical quantities. First, install preresquisites with:

```
sudo apt-get install -y libwxgtk3.0-gtk3-dev
```

Download fermisurfer here and extract the tar archive:

```
tar -xsf fermisurfer-2.1.0.tar.gz
```

Finally, enter the fermisurfer directory and install with:

```
./configure
 make
 sudo make install
```

1.7 F3ToyModel installation

F3ToyModel is a force-field that can mimic the physics of ferroelectric transitions in FCC lattices. All preresquisites are met with the SSCHA installation.

The code for this force-field can be downloaded from the SSCHA github here with the command:

```
git clone https://github.com/SSCHAcode/F3ToyModel.git
```

Now enter the F3ToyModel directory and install with:

```
python setup.py install
```

1.8 Setting up a virtual machine on a Mac with M1/M2

To set up a valid virtual machine valid for M1 or M2 processors, follow the instructions in this document prepared as well by the Quanum Mobile team. Once you have installed it, you will need to increase the memory of the virtual machine within the UTM settings at least to 4 Gb.

Once you have the Virtual Machine up and running you need to install the software described above in it. You can follow the instructions in *Requirements* to set up the machine, in *SSCHA* to install the SSCHA, in *TDSCHA* to install TDSCHA, in *EPIq* to install EPIq, in *fermisurfer installation* to install fermisurfe, and in *F3ToyModel installation* to install the F3ToyModel.

One important thing in this machine is that you should use for python the command

```
python3
```

The installation of quippy-ase will probably fail. In order to install it follow these points:

```
git clone --recursive https://github.com/libAtoms/QUIP.git
cd QUIP
export QUIP_ARCH=linux_x86_64_gfortran
make config
```

This will start making lots of questions. For python selection choose /usr/bin/python3 and answer y to GAP support. The rest of the questions can be left with the default answer. Then continue with these commands:

```
make
make quippy
install-quippy
```

In order to install qe-5.1.0_elph follow these instructions:

```
git clone https://github.com/SSCHAcode/qe-5.1.0_elph.gitc
cd qe-5.1.0_elph
./configure --build=aarch64-unknown-linux-gnu
make all
```

This should be enough to set up the virtual machine on your M1/M2 Mac.

CHAPTER

TWO

HANDS-ON-SESSION 1 - FIRST SSCHA SIMULATIONS: FREE ENERGY AND STRUCTURAL RELAXATIONS

In this hands-on we provide ready to use examples to setup your first SSCHA calculation.

2.1 The free energy of gold: a simulation in the NVT ensemble

This simple tutorial explains how to setup a SSCHA calculation starting just from the structure, in this case a cif file we downloaded from the [Materials Project](https://materialsproject.org/materials/mp-81/) database that we can find in the *01_First_SSCHA_simulations* directory.

Starting from the Gold structure in the primitive cell, to run the SSCHA we need:

- Compute the harmonic phonons (dynamical matrix)
- Remove imaginary frequencies (if any)
- Run the SSCHA

At the very beginning, we simply import the sscha libraries, cellconstructor, the math libraries and the force field. This is done in python with the *import* statemets.

```
# Import the sscha code
import sscha, sscha.Ensemble, sscha.SchaMinimizer
import sscha.Relax, sscha.Utilities

# Import the cellconstructor library to manage phonons
import cellconstructor as CC, cellconstructor.Phonons
import cellconstructor.Structure, cellconstructor.calculators

# Import the force field of Gold
import ase, ase.calculators
from ase.calculators.emt import EMT

# Import numerical and general pourpouse libraries
import numpy as np, matplotlib.pyplot as plt
import sys, os
```

The first thing we do is to initialize a cellconstructor structure from the cif file downloaded from the material database (*Au.cif*). We initialize the EMT calculator from ASE, and relax the structure:

```
gold_structure = CC.Structure.Structure()
gold_structure.read_generic_file("Au.cif")

# Get the force field for gold
calculator = EMT()

# Relax the gold structure
relax = CC.calculators.Relax(gold_structure, calculator)
gold_structure_relaxed = relax.static_relax()
```

In the case of Gold the relaxation is useless, as it is a FCC structure with Fm-3m symmetry group and 1 atom per primitive cell. This means the atomic positions have no degrees of freedom, thus the relaxation will end before even start.

Next, we perform the harmonic phonon calculation using cellconstructor and a finite displacement approach:

```
gold_harmonic_dyn = CC.Phonons.compute_phonons_finite_displacements(gold_
↪structure_relaxed, calculator, supercell = (4,4,4))

# Impose the symmetries and
# save the dynamical matrix in the quantum espresso format
gold_harmonic_dyn.Symmetrize()
gold_harmonic_dyn.save_qe("gold_harmonic_dyn")
```

The method *compute_phonons_finite_displacements* is documented in the CellConstructor guide. It requires the structure (in this case *gold_structure_relaxed*), the force-field (*calculator*) and the supercell for the calculation. In this case we use a 4x4x4 (equivalent to 64 atoms). This may not be sufficient to converge all the properties, especially at very high temperature, but it is just a start.

Note that *compute_phonons_finite_displacements* works in parallel with MPI, therefore, if the script is executed with *mpirun -np 16 python myscript.py* it will split the calculations of the finite displacements across 16 processors. You need to have mpi4py installed. However, in this case, due to the high symmetries, only one calculation is reqired to get the harmonic dynamical matrix, therefore the parallelization is useless.

After computing the harmonic phonons in gold_harmonic_dyn, we impose the correct symmetrization and the acousitic sum rule with the *Symmetrize* method, and save the result in the quantum ESPRESSO format with *save_qe*.

We are ready to submit the SSCHA calculation in the NVT ensemble.

The important parameters are:

- The temperature
- The number of random configurations in the ensemble
- The maximum number of iterations

These parameters are almost self-explaining. In contrast with Molecular Dynamics (MD) or Metropolis-Monte Carlo (MC) calculations, where the equilibrium probability distribution is sampled from a dynamical evolution of a structure, the SSCHA encodes the whole probability distribution as an analytical function. Therefore, to compute properties, we can generate on the fly the configurations that sample the equilibrium distribution.

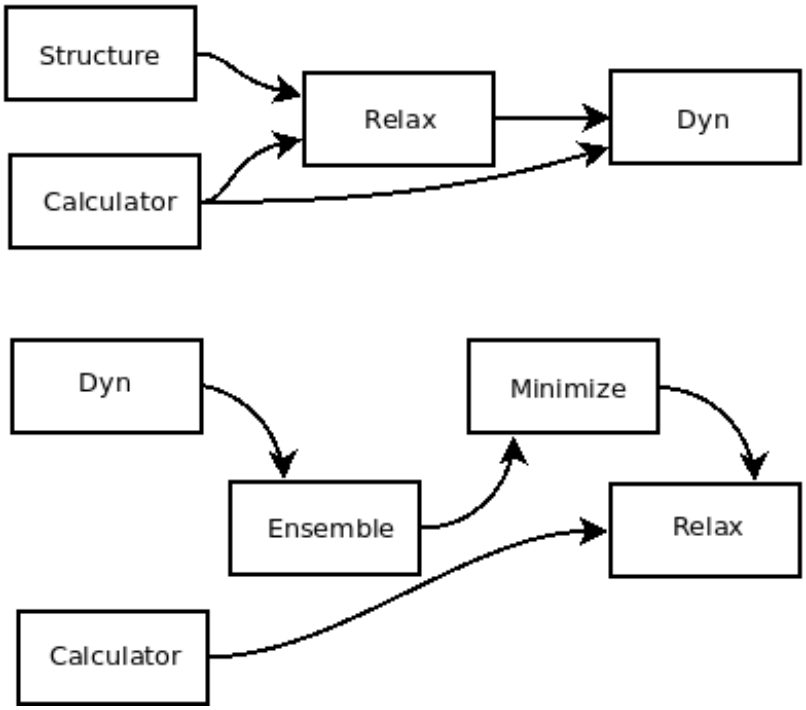


Fig. 2.1.1: Workflow of the SSCHA objects for a free energy minimization.

The code that sets up and perform the SSCHA is the following:

```
TEMPERATURE = 300
N_CONFIGS = 50
MAX_ITERATIONS = 20

# Initialize the random ionic ensemble
ensemble = sscha.Ensemble.Ensemble(gold_harmonic_dyn, TEMPERATURE)

# Initialize the free energy minimizer
minim = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
minim.set_minimization_step(0.01)

# Initialize the NVT simulation
relax = sscha.Relax.SSCHA(minim, calculator, N_configs = N_CONFIGS,
max_pop = MAX_ITERATIONS)

# Define the I/O operations
# To save info about the free energy minimization after each step
ioinfo = sscha.Utilities.IOInfo()
ioinfo.SetupSaving("minim_info")
relax.setup_custom_functions(custom_function_post = ioinfo.CFP_SaveAll)

# Run the NVT simulation
relax.relax(get_stress = True)

# Save the final dynamical matrix
relax.minim.dyn.save_qe("sscha_T300_dyn")
```

In the previous code we defined the main object to run the simulation:

- *ensemble* (*sscha.Ensemble.Ensemble*), represents the ensemble of ionic configurations. We initialize it with the dynamical matrix (which represent how much atoms fluctuate around the centroids) and the temperature.
- *minim* (*sscha.SchaMinimizer.SSCHA_Minimizer*) performs the free energy minimization. It contains all the info regarding the minimization algorithm, as the initial timestep (that here we set to 0.01). You can avoid setting the time-step, as the code will automatically guess the best value.
- *relax* (*sscha.Relax.SSCHA*) automatizes the generation of ensembles, calculation of energies and forces and the free energy minimization to perform a NVT or NPT calculation. To initialize it, we pass the *minim* (which contains the ensemble with the temperature), the force-field (*calculator*), the number of configurations *N_configs* and the maximum number of iterations.

In this example, most of the time is spent in the minimization, however, if we replace the force-field with ab-initio DFT, the time to run the minimization is negligible with respect to the time to compute energies and forces on the ensemble configurations. The total (maximum) number of energy/forces calculations is equal to the number of configurations times the number of iterations (passed through the *max_pop* argument).

The calculation is submitted with *relax.relax()*. However, before running the calculation we introduce another object, the *IOInfo*. This tells the *relax* to save information of the free energy, its gradient and the anharmonic phonon frequencies during the minimization in the files *minim_info.dat* and *minim_info.freqs*. It is not mandatory to introduce them, but it is very usefull as it allows to visualize the minimization while it is running.

The full input file is:

```
# Import the sscha code
import sscha, sscha.Ensemble, sscha.SchaMinimizer, sscha.Relax, sscha.
↪Utilities

# Import the cellconstructor library to manage phonons
import cellconstructor as CC, cellconstructor.Phonons
import cellconstructor.Structure, cellconstructor.calculators

# Import the force field of Gold
import ase, ase.calculators
from ase.calculators.emt import EMT

# Import numerical and general pourpouse libraries
import numpy as np, matplotlib.pyplot as plt
import sys, os


"""
Here we load the primitive cell of Gold from a cif file.
And we use CellConstructor to compute phonons from finite differences.
The phonons are computed on a q-mesh 4x4x4
"""

gold_structure = CC.Structure.Structure()
gold_structure.read_generic_file("Au.cif")
```

(continues on next page)

(continued from previous page)

```
# Get the force field for gold
calculator = EMT()

# Relax the gold structure (useless since for symmetries it is already␣
↪relaxed)
relax = CC.calculators.Relax(gold_structure, calculator)
gold_structure_relaxed = relax.static_relax()

# Compute the harmonic phonons
# NOTE: if the code is run with mpirun, the calculation goes in parallel
gold_harmonic_dyn = CC.Phonons.compute_phonons_finite_displacements(gold_
↪structure_relaxed, calculator, supercell = (4,4,4))

# Impose the symmetries and
# save the dynamical matrix in the quantum espresso format
gold_harmonic_dyn.Symmetrize()
gold_harmonic_dyn.save_qe("harmonic_dyn")


# If the dynamical matrix has imaginary frequencies, remove them
gold_harmonic_dyn.ForcePositiveDefinite()

"""
gold_harmonic_dyn is ready to start the SSCHA calculation.

Now let us initialize the ensemble, and the calculation at 300 K.
We will run a NVT calculation, using 100 configurations at each step
"""

TEMPERATURE = 300
N_CONFIGS = 50
MAX_ITERATIONS = 20

# Initialize the random ionic ensemble
ensemble = sscha.Ensemble.Ensemble(gold_harmonic_dyn, TEMPERATURE)

# Initialize the free energy minimizer
minim = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
minim.set_minimization_step(0.01)

# Initialize the NVT simulation
relax = sscha.Relax.SSCHA(minim, calculator, N_configs = N_CONFIGS,
max_pop = MAX_ITERATIONS)

# Define the I/O operations
# To save info about the free energy minimization after each step
ioinfo = sscha.Utilities.IOInfo()
ioinfo.SetupSaving("minim_info")
```

(continues on next page)

(continued from previous page)

```
relax.setup_custom_functions(custom_function_post = ioinfo.CFP_SaveAll)


# Run the NVT simulation (save the stress to compute the pressure)
relax.relax(get_stress = True)

# Save the final dynamical matrix
# And print in stdout the info about the minimization
relax.minim.finalize()
relax.minim.dyn.save_qe("sscha_T{}_dyn".format(TEMPERATURE))
```

Now save the file as *sscha_gold.py* and execute it with:

```
$ python sscha_gold.py > output.log
```

And that's it. The code will probably take few minutes on a standard laptop computer. **Congratulations!** You run your first SSCHA simulation!

If you open a new terminal in the same directory of the SSCHA submission, you can plot the info during the minimization. Starting from version 1.2, we provide a visualization utilities installed together with the SSCHA. Simply type

```
$ sscha-plot-data.py minim_info
```

You will see two windows.

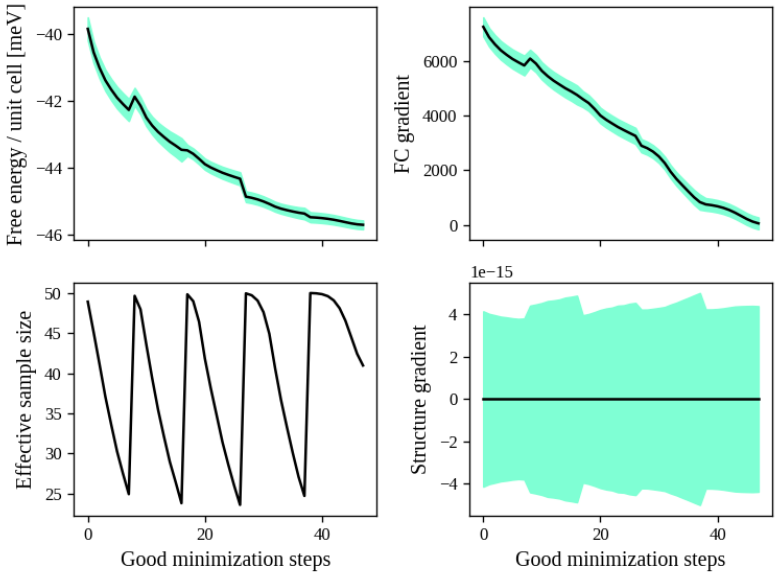


Fig. 2.1.2: Minimization data of Gold.

In Fig. 2.1.2 we have all the minimization data. On the top-left panel, we see the free energy. As expected, it decreases (since the SSCHA is minimizing it). You can see that at certain values of the steps there are discontinuities. These occurs when the code realizes that the ensemble on which it is computing is no more good and a new one is generated. The goodness of an ensemble is determined by the Kong-Liu effective sample size (bottom-left). When it reaches 0.5 of its initial value (equal to the number

of configurations), the ensemble is extracted again and a new iteration starts. You see that in the last iteration, the code stops before getting to 25 ($0.5 \cdot 50$). This means that the code converged properly: the gradient reached zero when the ensemble was still good.

On the right-side you see the free energy gradients, which must go to zero to converge. The top-right is the gradient of the SSCHA dynamical matrix, while on bottom-right there is the gradient of the average atomic positions.

Indeed, since the gold atomic positions are all fixed by symmetries, it is always zero (but it will be different from zero in more complex system).

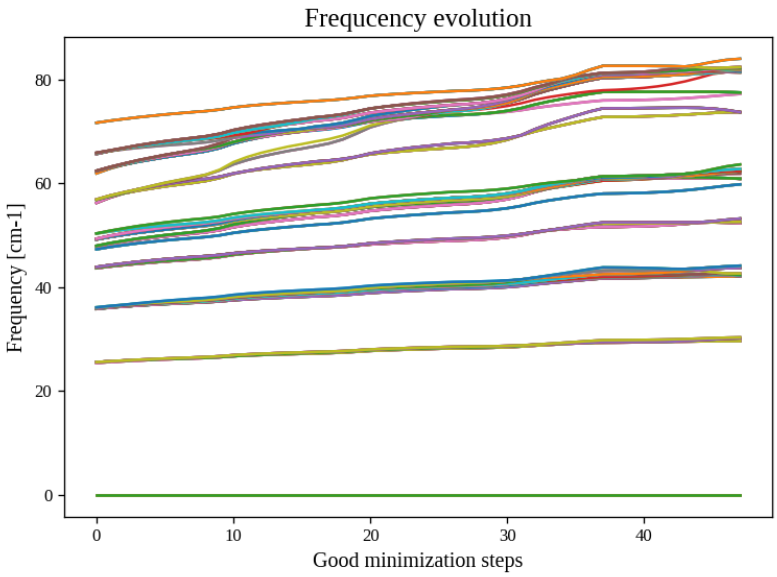


Fig. 2.1.3: All the SSCHA phonon frequencies as a function of the step in the NVT simulation.

Instead, Fig. 2.1.3 represents the evolution of the SSCHA phonon frequencies. Here, all the frequencies in the supercell (at each q point commensurate with the calculation) are shown.

NOTE

The sscha auxiliary frequencies in Fig. 2.1.3 are not the real frequencies observed in experiments, but rather are linked to the average displacements of atoms along that mode.

By looking at how they change you can have an idea on which phonon mode are more affected by anharmonicity. In this case, it is evident that Gold is strongly anharmonic and that the temperature makes all the phonon frequencies harder.

At the end of the simulation, the code writes the final dynamical matrix in the quantum espresso file format: *sscha_T300_dynX* where X goes over the number of irreducible q points.

In the next section, we analyze in details each section of the script to provide a bit more insight on the simulation, and a guide to modify it to fit your needs and submit your own system.

2.1.1 Plot the phonon dispersion

Now that the SSCHA minimization ended, we can compare the harmonic and anharmonic phonon dispersion of Gold.

To this purpouse, we can simply run a script like the following. You find a copy of this script already in Examples/ThermodynamicsOfGold/plot_dispersion.py.

You can use it even in your simulation, simply edit the value of the uppercase keyword at the beginning of the script to match your needs.

```
# Import the CellConstructor library to plot the dispersion
import cellconstructor as CC, cellconstructor.Phonons
import cellconstructor.ForceTensor

# Import the numerical libraries and those for plotting
import numpy as np
import matplotlib.pyplot as plt

import sys, os

# Let us define the PATH in the brilluin zone and the total number of points
PATH = "GXWXKGL"
N_POINTS = 1000

# Here we define the position of the special points
SPECIAL_POINTS = {"G": [0,0,0],
              "X": [0, .5, .5],
              "L": [.5, .5, .5],
              "W": [.25, .75, .5],
              "K": [3/8., 3/4., 3/8.]}

# The two dynamical matrix to be compared
HARM_DYN = 'harmonic_dyn'
SSCHA_DYN = 'sscha_T300_dyn'

# The number of irreducible q points
# i.e., the number of files in which the phonons are stored
NQIRR = 13


# ---------------------- THE SCRIPT FOLLOWS ----------------------

# Load the harmonic and sscha phonons
harmonic_dyn = CC.Phonons.Phonons(HARM_DYN, NQIRR)
sscha_dyn = CC.Phonons.Phonons(SSCHA_DYN, NQIRR)

# Get the band path
qpath, data = CC.Methods.get_bandpath(harmonic_dyn.structure.unit_cell,
              PATH,
              SPECIAL_POINTS,
              N_POINTS)
```

(continues on next page)

(continued from previous page)

```
xaxis, xticks, xlabels = data # Info to plot correclty the x axis

# Get the phonon dispersion along the path
harmonic_dispersion = CC.ForceTensor.get_phonons_in_qpath(harmonic_dyn, qpath)
sscha_dispersion = CC.ForceTensor.get_phonons_in_qpath(sscha_dyn, qpath)

nmodes = harmonic_dyn.structure.N_atoms * 3

# Plot the two dispersions
plt.figure(dpi = 150)
ax = plt.gca()

for i in range(nmodes):
    lbl=None
    lblsscha = None
    if i == 0:
        lbl = 'Harmonic'
        lblsscha = 'SSCHA'

    ax.plot(xaxis, harmonic_dispersion[:,i], color='k',
            ls='dashed', label=lbl)
    ax.plot(xaxis, sscha_dispersion[:,i],
            color='r', label=lblsscha)

# Plot vertical lines for each high symmetry points
for x in xticks:
   ax.axvline(x, 0, 1, color = "k", lw = 0.4)
ax.axhline(0, 0, 1, color = 'k', ls = ':', lw = 0.4)

# Set the x labels to the high symmetry points
ax.set_xticks(xticks)
ax.set_xticklabels(xlabels)

ax.set_xlabel("Q path")
ax.set_ylabel("Phonons [cm-1]")

ax.legend()

plt.tight_layout()
plt.savefig("dispersion.png")
plt.show()
```

If we save the script as *plot_dispersion.py* in the same directory of the calculation, we can run it with

```
$ python plot_dispersion.py
```

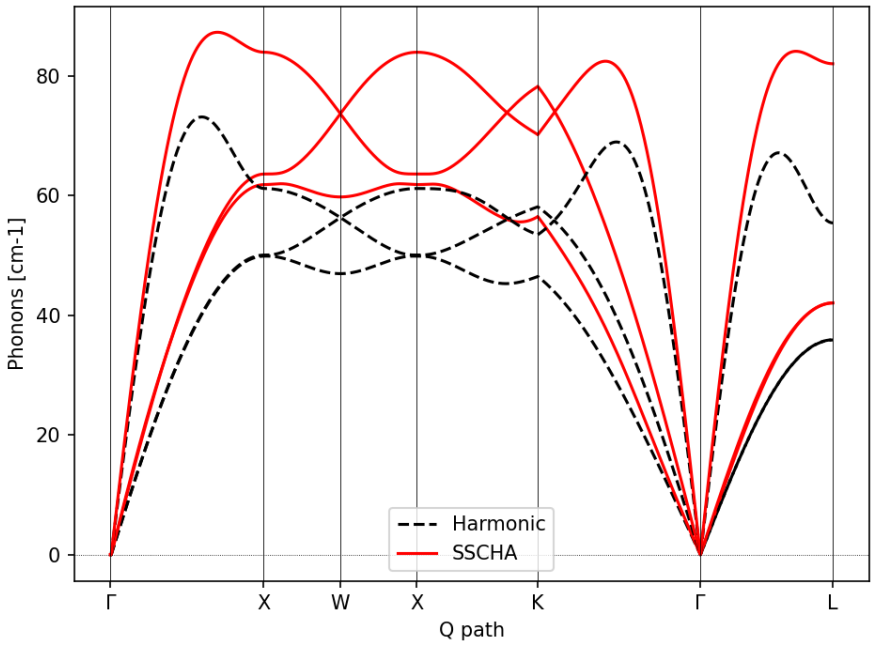


Fig. 2.1.4: Comparison between the SSCHA and the harmonic phonon dispersion of Gold.

The script will plot the figure of the phonon dispersion Fig. 2.1.4. It is quite different from the experimental one because of the poor accuracy of the force field, however, the SSCHA results is much closer to the experimental value.

Exercise

Try to perform the simulation of Gold but at a different temperature, plot then the SSCHA phonon dispersion as a function of temperature.

How does the phonon bands behave if the temperature is increased? Do they become more rigid (energy increases) or softer?

2.2 Running in the NPT ensemble: simulating thermal expansion

Now that you have some experience with the NVT simulation we are ready for the next step: NPT, or relaxing the lattice.

With python-sscha it is very easy to run NPT simulation, you simply have to replace the line of the NVT script with the target pressure for the simulation:

```
# Replace the line
# relax.relax(get_stress = True)
# with
relax.vc_relax(target_press = 0)
```

And that is all! The target pressure is expressed in GPa, in this case 0 is ambient conditions (1 atm = 0.0001 GPa)

You can also perform NVT simulation with variable lattice parameters: In this case the system will constrain the total volume to remain constant, but the lattice parameter will be optimized (if the system is not cubic and has some degrees of freedom, which is not the case for Gold).

The NVT ensemble with variable lattice parameters (cell shape) is

```
# Replace the line
#    relax.vc_relax(target_press = 0)
# with
relax.vc_relax(fix_volume = True)
```

Indeed, this is a NVT simulation, therefore there is no need to specify the target pressure.

The following script, we run the NPT ensemble at various temperatures, each time starting from the previous ensemble, to follow the volume thermal expansion of gold.

This script assume you already performed the NVT calculation, so that we can start from that results, and avoid the harmonic calculation (It is always a good practice to start with NVT simulation and then run NPT from the final result).

```
# Import the sscha code
import sscha, sscha.Ensemble, sscha.SchaMinimizer, sscha.Relax
import sscha.Utilities

# Import the cellconstructor library to manage phonons
import cellconstructor as CC, cellconstructor.Phonons
import cellconstructor.Structure, cellconstructor.calculators

# Import the force field of Gold
import ase, ase.calculators
from ase.calculators.emt import EMT

# Import numerical and general pourpouse libraries
import numpy as np, matplotlib.pyplot as plt
import sys, os


# Define the temperature range (in K)
T_START = 300
T_END = 1000
DT = 50

N_CONFIGS = 50
MAX_ITERATIONS = 10

# Import the gold force field
calculator = EMT()

# Import the starting dynamical matrix (final result of get_gold_free_energy.
↪py)
dyn = CC.Phonons.Phonons("sscha_T300_dyn", nqirr = 13)
```

(continues on next page)

(continued from previous page)

```
# Create the directory on which to store the output
DIRECTORY = "thermal_expansion"
if not os.path.exists(DIRECTORY):
    os.makedirs("thermal_expansion")

# We cycle over several temperatures
t = T_START


volumes = []
temperatures = []
while t <= T_END:
    # Change the temperature
    ensemble = sscha.Ensemble.Ensemble(dyn, t)
    minim = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
    minim.set_minimization_step(0.1)

    relax = sscha.Relax.SSCHA(minim, calculator, N_configs = N_CONFIGS,
                              max_pop = MAX_ITERATIONS)

    # Setup the I/O
    ioinfo = sscha.Utilities.IOInfo()
    ioinfo.SetupSaving( os.path.join(DIRECTORY, "minim_t{}".format(t)))
    relax.setup_custom_functions( custom_function_post = ioinfo.CFP_SaveAll)


    # Run the NPT simulation
    relax.vc_relax(target_press = 0)

    # Save the volume and temperature
    volumes.append(relax.minim.dyn.structure.get_volume())
    temperatures.append(t)

    # Start the next simulation from the converged value at this temperature
    relax.minim.dyn.save_qe(os.path.join(DIRECTORY,
                            "sscha_T{}_dyn".format(t)))
    dyn = relax.minim.dyn

    # Print in standard output
    relax.minim.finalize()

    # Update the temperature
    t += DT

    # Save the thermal expansion
    np.savetxt(os.path.join(DIRECTORY, "thermal_expansion.dat"),
               np.transpose([temperatures, volumes]),
               header = "Temperature [K]; Volume [A^3]")
```

You can run the script as always with:

```
$ python thermal_expansion.py
```

And … done!

This calculation is going to require a bit more time, as we run multiple SSCHA at several temperatures. After it finishes, you can plot the results written in the file thermal_expansion/thermal_expansion.dat.

A simple script to plot the thermal expansion (and fit the volumetric thermal expansion value) is the following

```
import numpy as np
import matplotlib.pyplot as plt

import scipy, scipy.optimize


# Load all the dynamical matrices and compute volume
DIRECTORY = "thermal_expansion"
FILE = os.path.join(DIRECTORY, "thermal_expansion.dat")

# Load the data from the final data file
temperatures, volumes = np.loadtxt(FILE, unpack = True)


# Prepare the figure and plot the V(T) from the sscha data
plt.figure(dpi = 150)
plt.scatter(temperatures, volumes, label = "SSCHA data")

# Fit the data to estimate the volumetric thermal expansion coefficient
def parabola(x, a, b, c):
    return a + b*x + c*x**2
def diff_parab(x, a, b, c):
    return b + 2*c*x

popt, pcov = scipy.optimize.curve_fit(parabola, temperatures, volumes,
                                      p0 = [0,0,0])

# Evaluate the volume thermal expansion
vol_thermal_expansion = diff_parab(300, *popt) / parabola(300, *popt)
plt.text(0.6, 0.2, r"$\alpha_v = "+"{:.1f}".format(vol_thermal_
↪expansion*1e6)+r"\times 10^6 $ K$^{-1}$",
         transform = plt.gca().transAxes)


# Plot the fit
_t_ = np.linspace(np.min(temperatures), np.max(temperatures), 1000)
plt.plot(_t_, parabola(_t_, *popt), label = "Fit")

# Adjust the plot adding labels, legend, and saving in eps
plt.xlabel("Temperature [K]")
plt.ylabel(r"Volume [$\AA^3$]")
```

(continues on next page)

(continued from previous page)

```
plt.legend()
plt.tight_layout()
plt.savefig("thermal_expansion.png")
plt.show()
```

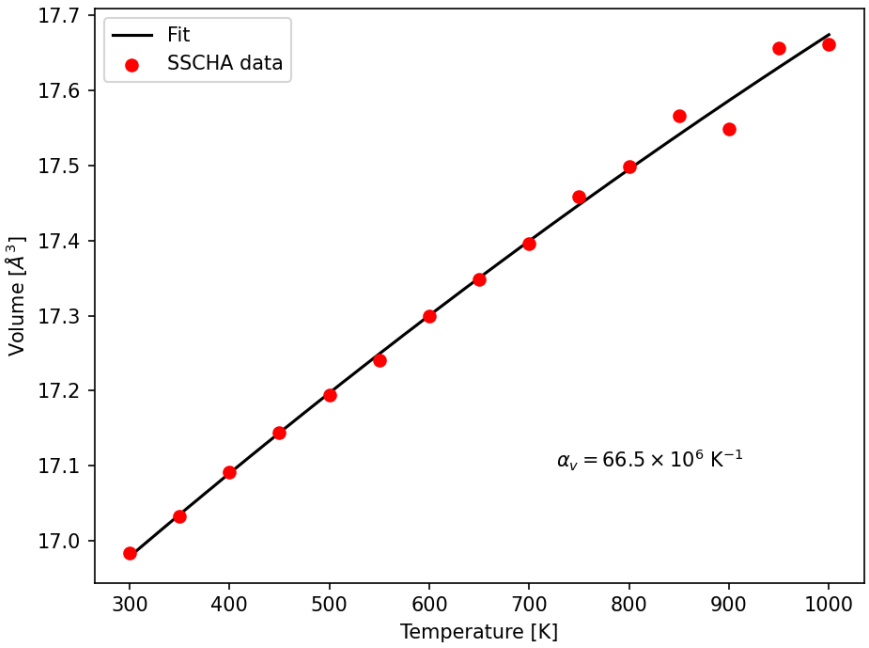


Fig. 2.2.1: Thermal expansion of Gold. From the fit of the data we can compute the volumetric thermal expansion coefficient (at 300 K).

We report the final thermal expansion in Fig. 2.2.1. The volumetric expansion coefficient α_v is obtained from the fit thanks to the thermodynamic relation:

$$\alpha_v = \frac{1}{V}\left(\frac{dV}{dT}\right)_P$$

Also in this case, the result is quite off with experiments, due to the not completely realistic force-field employed. To get a more realistic approach, you should use *ab-initio* calculations or a more refined force-field.

2.3 Ab initio calculation with the SSCHA code

The SSCHA code is compatible with the Atomic Simulation Environment (ASE), which we employed in the previous tutorial to get a fast force-field for Gold.

However, ASE already provides an interface with most codes to run ab initio simulations. The simplest way of interfacing the SSCHA to an other ab initio code is to directly use ASE.

The only difference is in the definition of the calculator, in the first example of this chapter, the Gold force field was defined as:

```
import ase
from ase.calculators.emt import EMT
calculator = EMT()
```

We simply need to replace these lines to our favourite DFT code. In this example we are going to use quantum espresso, but the procedure for VASP, CASTEP, CRYSTAL, ABINIT, SIESTA, or your favourite one are exatly the same (Refer to the official documentatio of ASE to the instruction on how to initialize these calculators).

In the case of DFT, unfortunately, we cannot simply create the calculator in one line, like we did for EMT force-field, as we need also to provide a lot of parameters, as pseudopotentials, the choice of exchange correlation, the cutoff of the basis set, and the k mesh grid for Brilluin zone sampling.

In the following example, we initialize the quantum espresso calculator for Gold.

```
import cellconstructor.calculators
from ase.calculators.espresso import Espresso

# Initialize the DFT (Quantum Espresso) calculator for gold
# The input data is a dictionary that encodes the pw.x input file namelist
input_data = {
    'control' : {
        # Avoid writing wavefunctions on the disk
        'disk_io' : 'None',
        # Where to find the pseudopotential
        'pseudo_dir' : '.'
    },
    'system' : {
        # Specify the basis set cutoffs
        'ecutwfc' : 45,   # Cutoff for wavefunction
        'ecutrho' : 45*4, # Cutoff for the density
        # Information about smearing (it is a metal)
        'occupations' : 'smearing',
        'smearing' : 'mv',
        'degauss' : 0.03
    },
    'electrons' : {
        'conv_thr' : 1e-8
    }
}

# the pseudopotential for each chemical element
# In this case just Gold
pseudopotentials = {'Au' : 'Au_ONCV_PBE-1.0.oncvpsp.upf'}

# the kpoints mesh and the offset
kpts = (1,1,1)
koffset = (1,1,1)


# Prepare the quantum espresso calculator
```

(continues on next page)

(continued from previous page)

```
#calculator = CC.calculators.Espresso(input_data,
             #                        pseudopotentials,
             #                        kpts = kpts,
             #                        koffset = koffset)
calculator = Espresso(input_data = input_data, pseudopotentials =␣
↪pseudopotentials,
                                                    kpts = kpts, koffset =␣
↪koffset)
```

If you are familiar with the quantum espresso input files, you should recognize all the options inside the input_data dictionary. For more options and more information, refer to the quantum ESPRESSO pw.x input guide.

Remember, the parameters setted here are just for fun, remember to run appropriate convergence check of the kmesh, smearing and basis set cutoffs before running the SSCHA code. Keep also in mind that this input file refers to the supercell, and the kpts variable can be properly rescaled if the supercell is increased.

All the rest of the code remains the same (but here we do not compute harmonic phonons, which can be done more efficiently within the Quantum ESPRESSO). Instead, we take the result obtained with EMT in the previous sections, and try to relax the free energy with a fully ab-initio approach.

The complete code is inside Examples/sscha_and_dft/nvt_local.py

```
# Import the sscha code
import sscha, sscha.Ensemble, sscha.SchaMinimizer, sscha.Relax, sscha.
↪Utilities

# Import the cellconstructor library to manage phonons
import cellconstructor as CC, cellconstructor.Phonons
import cellconstructor.Structure, cellconstructor.calculators

# Import the DFT calculator
import cellconstructor.calculators
from ase.calculators.espresso import Espresso

# Import numerical and general pourpouse libraries
import numpy as np, matplotlib.pyplot as plt
import sys, os


# Initialize the DFT (Quantum Espresso) calculator for gold
# The input data is a dictionary that encodes the pw.x input file namelist
input_data = {
    'control' : {
        # Avoid writing wavefunctions on the disk
        'disk_io' : 'None',
        # Where to find the pseudopotential
        'pseudo_dir' : '.'
    },
    'system' : {
```

(continues on next page)

(continued from previous page)

```
        # Specify the basis set cutoffs
        'ecutwfc' : 45,   # Cutoff for wavefunction
        'ecutrho' : 45*4, # Cutoff for the density
        # Information about smearing (it is a metal)
        'occupations' : 'smearing',
        'smearing' : 'mv',
        'degauss' : 0.03
    },
    'electrons' : {
        'conv_thr' : 1e-8
    }
}

# the pseudopotential for each chemical element
# In this case just Gold
pseudopotentials = {'Au' : 'Au_ONCV_PBE-1.0.oncvpsp.upf'}

# the kpoints mesh and the offset
kpts = (1,1,1)
koffset = (1,1,1)

# Specify the command to call quantum espresso
command = 'pw.x -i PREFIX.pwi > PREFIX.pwo'


# Prepare the quantum espresso calculator
#calculator = CC.calculators.Espresso(input_data,
             #                       pseudopotentials,
             #                       command = command,
             #                       kpts = kpts,
             #                       koffset = koffset)
 calculator = Espresso(input_data =  imput_data,
                                     pseudopotentials = pseudopotentials,
                                     command = command,
                                     kpts = kpts,
                                     koffset = koffset)



TEMPERATURE = 300
N_CONFIGS = 50
MAX_ITERATIONS = 20
START_DYN = 'harmonic_dyn'
NQIRR = 13

# Let us load the starting dynamical matrix
gold_dyn = CC.Phonons.Phonons(START_DYN, NQIRR)

# Initialize the random ionic ensemble
```

(continues on next page)

(continued from previous page)

```
ensemble = sscha.Ensemble.Ensemble(gold_dyn, TEMPERATURE)

# Initialize the free energy minimizer
minim = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
minim.set_minimization_step(0.01)

# Initialize the NVT simulation
relax = sscha.Relax.SSCHA(minim, calculator, N_configs = N_CONFIGS,
                          max_pop = MAX_ITERATIONS)

# Define the I/O operations
# To save info about the free energy minimization after each step
ioinfo = sscha.Utilities.IOInfo()
ioinfo.SetupSaving("minim_info")
relax.setup_custom_functions(custom_function_post = ioinfo.CFP_SaveAll)


# Run the NVT simulation (save the stress to compute the pressure)
relax.relax(get_stress = True)

# If instead you want to run a NPT simulation, use
# The target pressure is given in GPa.
#relax.vc_relax(target_press = 0)

# You can also run a mixed simulation (NVT) but with variable lattice␣
↪parameters
#relax.vc_relax(fix_volume = True)

# Now we can save the final dynamical matrix
# And print in stdout the info about the minimization
relax.minim.finalize()
relax.minim.dyn.save_qe("sscha_T{}_dyn".format(TEMPERATURE))
```

Now you can run the SSCHA with an ab-initio code! However, your calculation will probably take forever. To speedup things, lets discuss parallelization and how to exploit modern HPC infrastructures.

2.3.1 Parallelization

If you actually tried to run the code of the previous section on a laptop, it will take forever. The reason is that DFT calculations are much more expensive than the SSCHA minimization. While SSCHA minimizes the number of ab initio calculations (especially when compared with MD or PIMD), still they are the bottleneck of the computational time.

For this reason, we need an opportune parallelization strategy to reduce the total time to run a SSCHA.

The simplest way is to call the previous python script with MPI:

```
$ mpirun -np 50 python nvt_local.py > output.log
```

The code will split the configurations in each ensemble on a different MPI process. In this case we have 50 configurations per ensemble, by splitting them into 50 processors, we run the full ensemble in parallel.

However, still the single DFT calculation on 1 processor is going to take hours, and in some cases it may even take days. Luckily, also quantum ESPRESSO (and many other software) have an internal parallelization to work with. For example, we can tell quantum espresso to run itself in parallel on 8 processors. To this purpouse, we simply need to modify the command used to run quantum espresso in the previous script.

```
# Lets replace
# command = 'pw.x -i PREFIX.pwi > PREFIX.pwo'
# with
command = 'mpirun -np 8 pw.x -npool 1 -i PREFIX.pwi > PREFIX.pwo'

# The command string is passed to the espresso calculator
calculator = CC.calculators.Espresso(input_data,
                                     pseudopotentials,
                                     command = command,
                                     kpts = kpts,
                                     koffset = koffset)
```

In this way, our calculations will run on 400 processors (50 processors splits the ensemble times 8 processors per each calculation). This is achieved by nesting mpi calls. However, only the cellconstructor calculators can nest mpi calls without raising errors. This is the reason why we imported the Espresso class from cellconstructor and not from ASE. If you want to use ASE for your calculator, you can only use the inner parallelization of the calculator modifying the command, as ASE itself implements a MPI parallelization on I/O operations that conflicts with the python-sscha parallelization. This limitation only applies to FileIOCalculators from ASE (thus the EMT force-field is not affected and can be safely employed with python-sscha parallelization).

With this setup, the full code is parallelized over 400 processors. However the SSCHA minimization algorithm is a serial one, and all the time spent in the actual SSCHA minimization is wasting the great number of resources allocated. Moreover, the SSCHA code needs to be configured and correctly installed on the cluster, which may be a difficult operation due to the hybrid Fortran/pyhton structure.

CHAPTER

THREE

HANDS-ON-SESSION 2 - ADVANCED FREE ENERGY MINIMIZATION

This tutorial will cover more advanced code features, like the SSCHA code's interoperability with a high-performance computer (HPC). The tutorial is divided into two sections. In the first section, we will perform a free energy minimization manually; then we will learn how to automatize the interaction with a cluster to run *ab initio* calculations automatically.

3.1 Manual submission

The SSCHA calculation comprises three main steps iterated until convergence:

1. The generation of a random ensemble of ionic configurations
2. Calculations of energies and forces on the ensemble
3. The SSCHA free energy minimization

In the first hands-on session, you configured the code to do these iterations automatically. Thanks to the ASE EMT force field, the code can automatically compute energies, forces, and stress tensors without user interaction.

However, if you need to compute energies and forces from an *ab initio* calculation like DFT, you may want to run the DFT code on a different machine, like a cluster.

You can use the manual submission if you want more control over the procedure.

We will compute the sulfur hydride (superconductor with $T_c = 203$ K), using a DFT code like quantum Espresso to calculate energy and forces.

The harmonic phonons (computed using quantum Espresso) is provided in the directory **02_manual_submission**, where you can find the input and output files of the quantum espresso calculation to calculate the harmonic phonons, and the dynamical matrices, named dyn_h3s_harmonic_1, dyn_h3s_harmonic_2 and dyn_h3s_harmonic_3.

They respect the naming convention so that each file contains a different q point: since we are using a 2x2x2 mesh to sample the Brillouin zone of phonons, the different q points are ordered in three separate files, each one grouping the *star* of q (the q points related by symmetry operations).

We start by plotting the dispersion of the harmonic dynamical matrix. Please write in a file the following script and run it.

```
import cellconstructor as CC, cellconstructor.Phonons
import cellconstructor.ForceTensor
```

(continues on next page)

(continued from previous page)

```
import ase, ase.dft

import matplotlib.pyplot as plt
import numpy as np

dyn = CC.Phonons.Phonons("dyn_h3s_harmonic_", 3) # Load 3 files

PATH = "GHNPGN"
N_POINTS = 1000

# Use ASE to get the q points from the path
band_path = ase.dft.kpoints.bandpath(PATH,
    dyn.structure.unit_cell,
    N_POINTS)

# Get the q points in cartesian coordinates
q_path = band_path.cartesian_kpts()

# Get the values of x axis and labels for plotting the band path
x_axis, xticks, xlabels = band_path.get_linear_kpoint_axis()

# Perform the interpolation of the dynamical matrix along the q_path
frequencies = CC.ForceTensor.get_phonons_in_qpath(dyn, q_path)

# Plot the dispersion
fig = plt.figure()
ax = plt.gca()
ax.set_title("Harmonic H3S Phonon dispersion")
for i in range(frequencies.shape[-1]):
   ax.plot(x_axis, frequencies[:, i], color = 'r')


for x in xticks:
   ax.axvline(x, 0, 1, color='k', lw=0.4) # Plot vertical lines for each high-
↪symmetry point

# Set the labels of the axis as the Brilluin zone letters
ax.set_xticks(xticks)
ax.set_xticklabels(xlabels)

ax.set_ylabel("Frequency [cm-1]")
ax.set_xlabel("q-path")
plt.tight_layout()
plt.savefig("harmonic_h3s_dispersion.png")
plt.show()
```

You should see the figure *Dispersion of the harmonic phonons of H3S*.

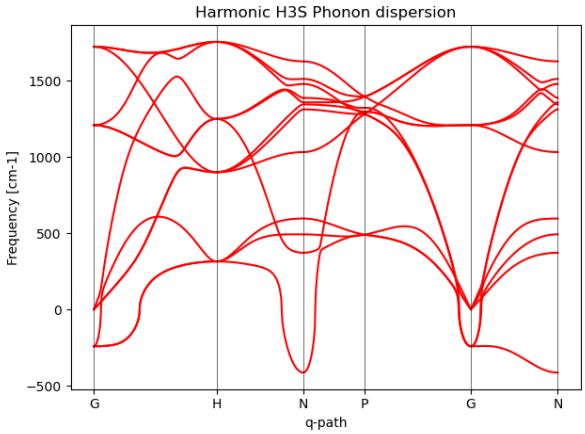


Fig. 3.1.1: Dispersion of the harmonic phonons of H3S

The dispersion presents imaginary phonons throughout most of the Brillouin zone. To start the SSCHA, we need a **positive** definite dynamical matrix. Since the starting point for the SSCHA does not matter, we may flip the phonons to be positive:

$$\Phi_{ab} = \sum_{\mu} \sqrt{m_a m_b} \, |\omega_\mu|^2 \, e_\mu^a e_\mu^b$$

where m_a is the mass of the a-th atom, ω_μ is the frequency of the dynamical matrix, and e_μ is the corresponding eigenvector. This operation can be performed with the command

```
dyn.ForcePositiveDefinite()
```

and save the results into `start_sscha1`, `start_sscha2`, and `start_sscha3` with

```
dyn.save_qe("start_sscha")
```

Exercise

Plot the phonon dispersion of the positive definite dynamical matrix obtained in this way. Save the resulting dynamical matrix as 'start_sscha' to continue with the following section.

3.1.1 Ensemble generation

Now that we have a good starting point for the dynamical matrix, we are ready to generate the first ensemble to start the free energy optimization. Here is a script to generate the ensemble.

The following script supposes that you saved the dynamical matrix after enforcing them to be positive definite as "start_sscha". However, you can edit the script to read the harmonic dynamical matrices and impose the positiveness within the same script.

```
import cellconstructor as CC, cellconstructor.Phonons
import sscha, sscha.Ensemble
import numpy as np

```

(continues on next page)

(continued from previous page)

```
# Fix the seed so that we all generate the same ensemble
np.random.seed(0)

# Load the dynamical matrix
dyn = CC.Phonons.Phonons("start_sscha", nqirr=3)

#[ apply here the needed changes to dyn ]

# Prepare the ensemble
temperature = 300 # 300 K
ensemble = sscha.Ensemble.Ensemble(dyn, temperature)

# Generate the ensemble
number_of_configurations = 10
ensemble.generate(number_of_configurations)

# Save the ensemble into a directory
save_directory = "data"
population_id = 1
ensemble.save(save_directory, population_id)
```

If you try to run the code, you can face an error telling you that the dynamical matrix does not satisfy the acoustic sum rule (ASR). This occurs because quantum Espresso does not impose the ASR by default. However, we can enforce the acoustic sum rule with the following:

```
dyn.Symmetrize()
```

Besides the ASR, this function will also impose all the symmetries on the dynamical matrix, ensuring it is correct.

Exercise

Impose the acoustic sum rule and the symmetries and generate the ensemble. Either add this after loading the dynamical matrix or do it once overriding the 'start_sscha' files.

3.1.2 Calculation of energies and forces

Very good; if you imposed the sum rule correctly, the ensemble should have been correctly generated. The script should have created the *data* directory and two sets of dynamical matrices:

1. dyn_start_population1_x
2. dyn_end_population1_x

where x goes from 1 to 3. These are the same dynamical matrix as the original one. In particular, dyn_start is the dynamical matrix used to generate the ensemble, and dyn_end is the final dynamical matrix after the free energy optimization. Since we did not run the sscha, they are the same.

If we look inside the *data* directory, we find:

1. energies_supercell_population1.dat

2. scf_population1_x.dat
3. u_population1_x.dat

where x counts from 1 to the total number of configurations, the energies_supercell file contains any structure's total DFT energy (in Ry). Since we have not yet performed DFT calculations, it is full of 0s.

u_population1_x.dat files contain the cartesian displacements of each atom in the supercell with respect to the average position. We will not touch this file, but the sscha uses it to load the ensemble much faster when we have many configurations and big systems.

The last files are the *scf_population1_x.dat*, containing the ionic positions, including the atomic type, in Cartesian coordinates.

This file contains the structure in the supercell; it is already in the standard quantum espresso format, so you can attach this text to the header file of the quantum espresso input to have a complete input file for this structure. However, you can easily manipulate this file to adapt it to your favorite programs, like VASP, ABINIT, SIESTA, CP2K, CASTEP, or any other.

You can visualize a structure using ASE and Cellconstructor:

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

struct = CC.Structure.Structure()
struct.read_scf("data/scf_population1_1.dat")
ase_struct = struct.get_ase_atoms()
ase.visualize.view(ase_struct)
```

Indeed, using the same trick, you can export the structure in any file format that ASE support, including input files for different programs mentioned above.

Here, we will use quantum Espresso. The header file for the quantum espresso calculation is in *espresso_header.pwi*. Remember that the configurations are in the supercell, so the number of atoms (here 32 instead of 4) and any extensive parameter like the k-point mesh should be rescaled accordingly. Here we employ an 8x8x8 k-mesh for the electronic calculation, while to compute the harmonic phonons with a unit cell calculation, we use a 16x16x16 k-mesh since the sscha configurations are 2x2x2 bigger than the original one, and thus the Brillouin zone is a factor 0.5x0.5x0.5 smaller.

You can append each scf file to this header to get the espresso input.

We have only ten configurations; in production runs, using at least hundreds of configurations per ensemble is appropriate. Therefore, it is impractical to create the input file for each of them manually.

```
#!/bin/bash

HEADER_FILE=espresso_header.pwi
DATA_DIR=data
POPULATION=1

# Define a directory in which to save all the input files
TARGET_DIRECTORY=$DATA_DIR/input_files_population$POPULATION

mkdir -p $TARGET_DIRECTORY
```

(continues on next page)

(continued from previous page)

```
for file in `ls $DATA_DIR/scf_population${POPULATION}*.dat`
do
    # Extract the configuration index
    # (the grep command returns only the expression
    #  that matches the regular expression from the file name)
    index=`echo $file | grep -oP '(?<=population1_).*(?=\.dat)'`

    target_input_file=$TARGET_DIRECTORY/structure_${index}.pwi

    # Copy the template header file
    cp $HEADER_FILE $target_input_file

    # Attach after the header the structure
    cat $file >> $target_input_file
done
```

Executing this script, you have created a directory inside the data dir called *input_files_population1* which contains all the input files for quantum Espresso.

You can run these with your own laptop if you have a good computer. However, the calculation is computationally demanding: each configuration contains plenty of atoms and no symmetries at all, as they are snapshots of the quantum/thermal motion of the nuclei. The alternative is to copy these files on a cluster and submit a calculation there.

The espresso files are run with the command

```
mpirun -np NPROC pw.x -i input_file.pwi > output_file.pwo
```

where NPROC is the number of processors in which we want to run. **Remember to copy the pseudopotential in the same directory where you run the pw.x executable.**

However, we skip this part now (try it yourself later!)

We provide the output files in the folder *output_espresso*

Once we have the output files from Espresso, we need to save the energies, forces, and stress tensors in the ensemble directory.

```
#!/bin/bash

N_CONFIGS=10
POPULATION=1
PATH_TO_DIR="data/output_espresso"
N_ATOMS=32


ENERGY_FILE="data/energies_supercell_population${POPULATION}.dat"

# Clear the energy file
rm -rf $ENERGY_FILE

for i in `seq 1 10`
```

(continues on next page)

(continued from previous page)

```
do
    filename=${PATH_TO_DIR}/structure_$i.pwo
    force_file=data/forces_population${POPULATION}_$i.dat
    stress_file=data/pressures_population${POPULATION}_$i.dat

    # Get the total energy
    grep ! $filename | awk '{print $5}' >> $ENERGY_FILE
    grep force $filename | grep atom | awk '{print $7, $8, $9}' > $force_file
    grep "total   stress" $filename -A3 | tail -n +2 | awk '{print $1, $2, $3}
↪' > $stress_file
done
```

This script works specifically for quantum Espresso. It extracts energy, forces, and the stress tensor and fills the files *data/energies_supercell_population1.dat*, *forces_population1_X.dat*, and *pressures_population1_X.dat* with the results obtained from the output file of quantum Espresso.

The units of measurement are

1. Ry for the energy (in the supercell)
2. Ry/Bohr for the forces
3. Ry/Bohr^3 for the stress tensor

Here, we do not need conversion, as these are the default units quantum Espresso gives. However, remember to convert correctly to these units if you use a different program, like VASP.

3.1.3 Free energy minimization

We have the ensemble ready to be loaded back into the Python script and start a minimization. This is done with the following scripts

```
import sscha, sscha.Ensemble, sscha.SchaMinimizer
import sscha.Utilities
import cellconstructor as CC, cellconstructor.Phonons

POPULATION = 1

dyn = CC.Phonons.Phonons("start_sscha", 3)
ensemble = sscha.Ensemble.Ensemble(dyn, 0)
ensemble.load("data", population = POPULATION, N = 10)

minim = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
minim.init()

# Save the minimization details
ioinfo = sscha.Utilities.IOInfo()
ioinfo.SetupSaving("minim_{}".format(POPULATION))


minim.run(custom_function_post = ioinfo.CFP_SaveAll)
```

(continues on next page)

(continued from previous page)

```
minim.finalize()
minim.dyn.save_qe("final_sscha_dyn_population{}_".format(POPULATION))
```

You can plot the results of the minimization with

```
sscha-plot-data.py minim_1
```

Congratulations! You run your first completely manual SSCHA run.

The output file informs us that minimization ended because the ensemble is *out of the stochastic criteria*. This means that the dynamical matrix changed a sufficient amount that the original ensemble was not good enough anymore to describe the free energy of the new dynamical matrix; therefore, a new ensemble should be extracted.

In the early days of the SSCHA, this procedure should have been iterated repeatedly until convergence. Nowadays, we have a fully automatic procedure that can automatize all these steps configuring the ssh connection to a cluster.

3.2 Automatic submission with a cluster

In the previous section, you made all the steps to run a sscha calculation manually. This consists of iterating through the following steps:

1. generating the input files for Espresso,
2. transferring them to a cluster,
3. submitting the calculations,
4. retrieving the outputs,
5. reload the ensemble
6. run the free energy minimization

In this section, we learn how to automatize these passages. We must set up the interaction between the SSCHA library and the HPC cluster running the DFT calculations. As of June 2023, this automatic interaction is only supported for quantum Espresso and SLURM-based clusters. However, writing plugins to support different DFT codes and cluster schedulers should be easy.

The configuration of the DFT parameter has been introduced in the previous hands-on session; thus, we skip and provide a file called *espresso_calculator.py*, which defines a function *get_h3s_calculator* returning the calculator object for quantum Espresso with the input parameters for H3S.

We focus instead on the configuration of the cluster. Create a new file called *cluster.py*. The following script provides an example to connect to a cluster with username `sschauser` and login node `my.beautiful.cluster.eu`:

```
import cellconstructor as CC, cellconstructor.Phonons
import sscha
import sscha.Cluster

import sys, os

```

(continues on next page)

(continued from previous page)

```
def configure_cluster(cluster_workdir = "H3S"):
    cluster = sscha.Cluster.Cluster(hostname = "sschauser@my.beautiful.
↪cluster.eu")

    cluster.use_memory = True
    cluster.ram = 180000
    cluster.use_partition = True
    cluster.partition_name = "workstations"
    cluster.account_name = "my_allocation_resources"
    cluster.n_nodes = 1
    cluster.use_cpu = False
    cluster.custom_params["get-user-env"] = None
    cluster.custom_params["cpus-per-task"] = 2
    cluster.custom_params["ntasks-per-node"] = 48
    cluster.time = "12:00:00"
    cluster.n_cpu = 48
    cluster.n_pool = 48
    cluster.job_number = 12
    cluster.batch_size = 2

    home_workdir=os.path.join("$HOME", cluster_workdir)
    scratch_workdir = os.path.join("/scratch/$USER/", cluster_workdir)
    cluster.workdir = home_workdir
    cluster.add_set_minus_x = True  # Avoid the set -x
    cluster.load_modules = f"""

module purge
module load intel
module load intel-mpi
module load intel-mkl
module load quantum-espresso/6.8.0-mpi

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

mkdir -p {scratch_workdir}
cp $HOME/espresso/pseudo/* {scratch_workdir}/
"""

    def cp_files(lbls):
        extrain = f"cd {scratch_workdir}\n"
        extraout = "sleep 1\n"
        for lbl in lbls:
            extrain += f"cp {home_workdir}/{lbl}.pwi {scratch_workdir}/\n"
            extraout += f"mv {scratch_workdir}/{lbl}.pwo {home_workdir}/\n"

        return extrain, extraout

    # Add the possibility to copy the input files
```

(continues on next page)

(continued from previous page)

```
    cluster.additional_script_parameters = cp_files

    # Force to open a shell when executing ssh commands
    # (Otherwise the cluster will not load the module environment)
    cluster.use_active_shell = True
    cluster.setup_workdir()

    # Check the communication
    if not cluster.CheckCommunication():
        raise ValueError("Impossible to connect to the cluster.")

    return cluster
```

This file contains all the information to connect with the cluster that you can customize to adapt to your HPC center.

Let us dive a bit into the options.

The first thing to know how to configure is the ssh host connection. For example, if I connect to a cluster using the command

```
ssh sschauser@my.beautiful.cluster.eu
```

You have to specify the entire string *sschauser@my.beautiful.cluster.eu* inside the `hostname` key at the first definition of the cluster. If you have an ssh config file enabled, you can substitute the hostname with the name in the configuration file corresponding to a HostName inside `.ssh/config` located in your home directory.

The best procedure is to enable a public-private key **without** encryption. You can activate the encryption if you have a wallet system in your PC that keeps the password saved, but in this case, the user must log in with the screen unlocked to work.

If the HPC does not allow you to configure a pair of ssh keys for the connection and requires the standard username/password connection, you can add the `pwd` keyword in the definition of the cluster. This is not encouraged, as you will store your password in clear text inside the script (so if you are in a shared workstation, remember to limit the read access to your scripts to other users, and *do not* send the script accidentally to other people with your password).

For example:

```
cluster = sscha.Cluster.Cluster(hostname="sschauser@my.beautiful.cluster.eu",␣
↪pwd="mybeautifulpassword")
```

The other options are all standard SLURM configurations, as the amount of ram, name of partition, and account for the submission, number of nodes, total time, and custom parameters specific for each cluster These parameters are transformed into the submission script for slurm as

```
#SLURM --time=12:00:00
#SLURM --get-user-env
#SLURM --cpus-per-task=2
# [...]
```

Most variables have the *use_xxx* attribute; if set to False, the corresponding option is not printed. In the

last version of SSCHA, if you manually edit a variable, it should automatically set the corresponding *use_xxx* to true.

```
cluster.use_partition = True
cluster.partition_name = "workstations"
cluster.account_name = "my_allocation_resources"
```

Most clusters must run on specific partitions; in this case, activate the partition flag with the `use_partition` variable and specify the appropriate `partition_name`. Also, most of the time, the computational resources are related to specific accounts indicated with `account_name`.

Particular attention needs to be taken to the following parameters

```
cluster.n_nodes = 1
cluster.time = "12:00:00"
cluster.n_cpu = 48
cluster.n_pool = 48
cluster.job_number = 12
cluster.batch_size = 2
```

These parameters are specific for the kind of calculation.

1. n_nodes specifies the number of nodes
2. time specifies the total time
3. n_cpu specify how many processors to call quantum Espresso with
4. n_pool is the number of pools for the quantum espresso parallelization; it should be the greatest common divisor between the number of CPUs and K points.
5. batch_size how many pw.x calculations to group in the same job (executed one after the other without queue time).
6. job_number how many jobs will be submitted simultaneously (executed in parallel, but with queue time).

The total time requested must be roughly the time expected for a single calculation multiplied by the batch_size. It is convenient to overshoot the requested time, as some configurations may take a bit more time.

The workdir is the directory in which all the input files are copied inside the cluster. This cluster uses a local scratch for the submission (the job must copy all the input on a local scratch of the node and then copy back the results to the shared filesystem). If no local scratch is requried, then we can set the working directory (usually a shared scratch) with the command

```
cluster.workdir = "/scratch/myuser/"
```

However, this submission script (as ekhi) must work on a shared workdir, which is inside the home directory. Therefore, we must tell the cluster to copy the files from the workdir to the local scratch before and after each calculation. This is done by setting a custom function, executed for each calculation

```
def cp_files(lbls):
    extrain = f"cd {scratch_workdir}\n"
    extraout = "sleep 1\n"
    for lbl in lbls:
```

(continues on next page)

(continued from previous page)

```
        extrain += f"cp {home_workdir}/{lbl}.pwi {scratch_workdir}/\n"
        extraout += f"mv {scratch_workdir}/{lbl}.pwo {home_workdir}/\n"

    return extrain, extraout

# Add the possibility to copy the input files
cluster.additional_script_parameters = cp_files
```

Each cluster must load modules to run a calculation. All the modules and other commands to run before the calculations are stored in the text variable *load_modules*

```
cluster.load_modules = f"""
module purge
module load intel
module load intel-mpi
module load intel-mkl
module load quantum-espresso/6.8.0-mpi

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

mkdir -p {scratch_workdir}
cp $HOME/espresso/pseudo/* {scratch_workdir}/
"""
```

The specific of the modules to load depends on the cluster, in this case, we also create the local scratch directory and copy the pseudopotential.

To check the connection and set up the working directory (create it on the cluster if it does not exist) use the

```
cluster.setup_workdir()

if not cluster.CheckCommunication():
    raise ValueError("Cluster connection failed!")
```

Exercise

Customize the cluster.py file to connect to the ekhi server, following the instructions provided in the *ekhi guide*.

3.2.1 How to submit a calculation with a cluster automatically

Now that we have seen how to configure the cluster, it is time to start an actual calculation. We can use this option to directly evaluate the ensemble generated manually before in the following way:

```
import cellconstructor as CC, cellconstructor.Phonons
import sscha, sscha.Ensemble

# Import the two python scripts for the cluster and espresso configurations
import espresso_calculator
import cluster

# Generate an ensemble with 10 configurations
dyn = CC.Phonons.Phonons("start_sscha", 3)
ensemble = sscha.Ensemble.Ensemble(dyn, 300)
ensemble.generate(10)

# Get the espresso and cluster configurations
espresso_config = espresso_calculator.get_calculator()
cluster_config = cluster.configure_cluster()

# Compute the ensemble
ensemble.compute_ensemble(espresso_config, cluster=cluster_config)

# Save the ensemble (using population 2 to avoid overwriting the other one)
ensemble.save("data", 2)
```

As seen here, once the cluster is configured (but this needs to be done only once), it is straightforward to compute the ensemble's energy, forces, and stresses.

While the calculation is running, the temporary files copied from/to the cluster are stored in a directory that is local_workdir. This is, by default, called cluster_work. They are called ESP_x.pwi EXP_x.pwo, the input and output files, and with ESP_x.sh you have the SLURM submission script.

Indeed, as you have seen in the previous hands-on session, it is possible to use the cluster keyword also in the *SSCHA* object of the *Relax* module to automatize all the procedures.

The following script runs the complete automatic relaxation of the SSCHA.

```
import cellconstructor as CC, cellconstructor.Phonons
import sscha, sscha.Ensemble
import sscha.SchaMinimizer, sscha.Relax

# Import the two python scripts for the cluster and espresso configurations
import espresso_calculator
import cluster

# Generate an ensemble with 10 configurations
dyn = CC.Phonons.Phonons("start_sscha", 3)
ensemble = sscha.Ensemble.Ensemble(dyn, 300)

# Get the espresso and cluster configurations
```

(continues on next page)

(continued from previous page)

```
espresso_config = espresso_calculator.get_calculator()
cluster_config = cluster.configure_cluster()

# Setup the minimizer
minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)

# Setup the automatic relaxation
relax = sscha.Relax.SSCHA(minimizer, espresso_config,
        N_configs=10,
        max_pop=3,
        save_ensemble=True,
        cluster=cluster_config)

# Setup the IO to save the minimization data and the frequencies
ioinfo = sscha.Utilities.IOInfo()
ioinfo.SetupSaving("minimization_data")

# Activate the data saving in the minimization
relax.setup_custom_functions(custom_function_post=ioinfo.CFP_SaveAll)

# Perform the NVT simulation
relax.relax(get_stress=True)

# Save the data
relax.minim.finalize()
relax.minim.dyn.save_qe("final_dyn")
```

As for this NVT, you can also use *vc_relax* for the NPT simulation or the NVT with variable cell shape.

CHAPTER

FOUR

HANDS-ON-SESSION 3 - CALCULATIONS OF SECOND-ORDER PHASE TRANSITIONS WITH THE SSCHA

In this hands-on, we learn how to calculate second-order phase transitions within the SSCHA.

4.1 Structural instability: calculation of the Hessian

According to Landau's theory, a second-order phase transition occurs when the free energy curvature around the high-symmetry structure on the direction of the order parameter becomes negative:

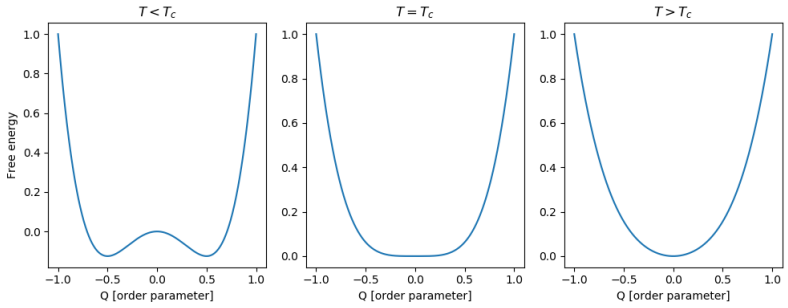


Fig. 4.1.1: Landau's theory of second-order phase transitions.

For structural *displacive* phase transitions, the order parameter is associated to phonon atomic displacements:

$$\frac{\partial^2 F}{\partial R_a \partial R_b}$$

Thus, the Free energy Hessian is the central quantity to study second-order phase transitions. The SSCHA provides an analytical equation for the free energy Hessian, derived by Raffaello Bianco in the work Bianco et. al. Phys. Rev. B 96, 014111. The free energy curvature can be written as:

$$\frac{\partial^2 F}{\partial R_a \partial R_b} = \Phi_{ab} + \sum_{cdef} \overset{(3)}{\Phi}_{acd} \left[1 - \Lambda \overset{(4)}{\Phi}\right]^{-1}_{cdef} \overset{(3)}{\Phi}_{efb}$$

Fortunately, this complex equation can be evaluated from the ensemble with a simple function call:

```
ensemble.get_free_energy_hessian()
```

Lets see a practical example, first we calculate the SSCHA dynamical matrix for the SnTe:

To speedup the calculations, we will use a force-field that can mimic the physics of ferroelectric transitions in FCC lattices.

We begin importing some libraries:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
#  SSCHA_exercise_Calculus.py
#
# Import the cellconstructor stuff
import cellconstructor as CC
import cellconstructor.Phonons
import cellconstructor.ForceTensor
import cellconstructor.Structure
import cellconstructor.Spectral

# Import the modules of the force field
import fforces as ff
import fforces.Calculator

# Import the modules to run the sscha
import sscha, sscha.Ensemble, sscha.SchaMinimizer
import sscha.Relax, sscha.Utilities

import spglib
from ase.visualize import view

# Import Matplotlib to plot
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import timeit
```

Next we set some variables for the calculation:

```
#Setting the variables:
#Setting the temperature in Kelvin:
Temperature = 0
#Setting the number of configurations:
configurations = 50
#Setting the names and location of the files:
Files_dyn_SnTe = "ffield_dynq"
#Set the number of irreducible q (reated to the supercell size):
nqirr = 3
#Setting the frequencies output file:
File_frequencies = "frequencies.dat"
#Setting the dynamical matrix output filename:
File_final_dyn = "final_sscha_T{}_".format(int(Temperature))
```

Now we need to calculate the SSCHA dynamical matrix. For that we follow some steps:

1. First we prepare the Toy model force field that substitutes the usual *ab-initio* for this tutorial. This

force field needs the harmonic dynamical matrix to be initialized, and the higher order parameters. Finally, the dynamical matrix for the minimization is loaded and readied. Since we are studying a system that has a spontaneous symmetry breaking at low temperature, the harmonic dynamical matrices will have imaginary phonons. We must enforce phonons to be positive definite to start a SSCHA minimization.

```
# Load the dynamical matrix for the force field
ff_dyn = CC.Phonons.Phonons("ffield_dynq", 3)

# Setup the forcefield with the correct parameters
ff_calculator = ff.Calculator.ToyModelCalculator(ff_dyn)
ff_calculator.type_cal = "pbtex"
ff_calculator.p3 = 0.036475
ff_calculator.p4 = -0.022
ff_calculator.p4x = -0.014

# Initialization of the SSCHA matrix
dyn_sscha = CC.Phonons.Phonons(Files_dyn_SnTe, nqirr)
# Flip the imaginary frequencies into real ones
dyn_sscha.ForcePositiveDefinite()
# Apply the ASR and the symmetry group
dyn_sscha.Symmetrize()
```

2. The next step is to create the ensembles for the specified temperature. As an extra, we also look for the space group of the structure.

```
ensemble = sscha.Ensemble.Ensemble(dyn_sscha,
        T0 = Temperature, supercell = dyn_sscha.GetSupercell())
# Detect space group
symm=spglib.get_spacegroup(dyn_sscha.structure.get_ase_atoms(),
        0.005)
print('Initial SG = ', symm)
```

3. Next comes the minimization step. Here we can set the fourth root minimization, in which, instead of optimizing the auxiliary dynamical matrices themselves, we will optimize their fourth root.

$$\Phi = \left(\sqrt[4]{\Phi}\right)^4$$

This constrains the dynamical matrix to be positive definite during the minimization. Next the automatic relaxation is set with the option here to use the Sobol sequence for the ensemble generation.

We also set a custom function to save the frequencies at each iteration, to see how they evolves. This is very useful to understand if the algorithm is converged or not.

Then the dynamical matrix of the converged minimization is saved in a file, and finally we take a look at the space group and the structure.

```
minim = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)

# Now we setup the minimization parameters
# Since we are quite far from the correct solution,
```

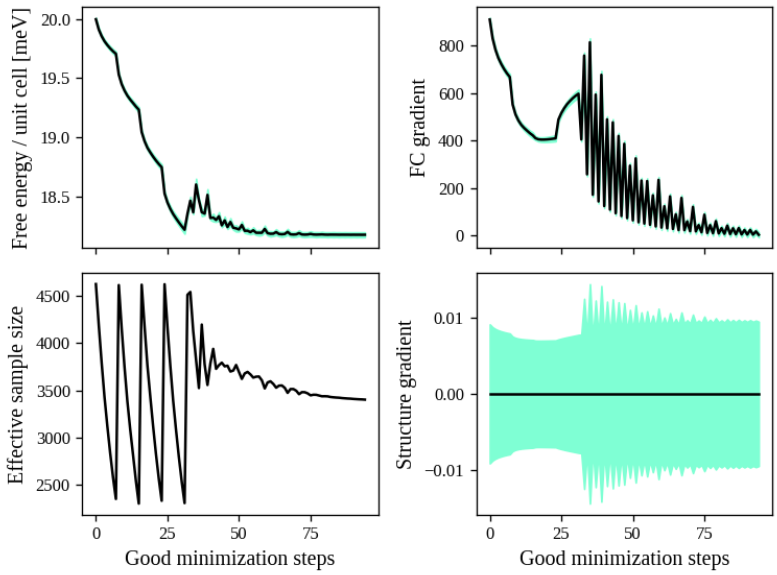
(continues on next page)

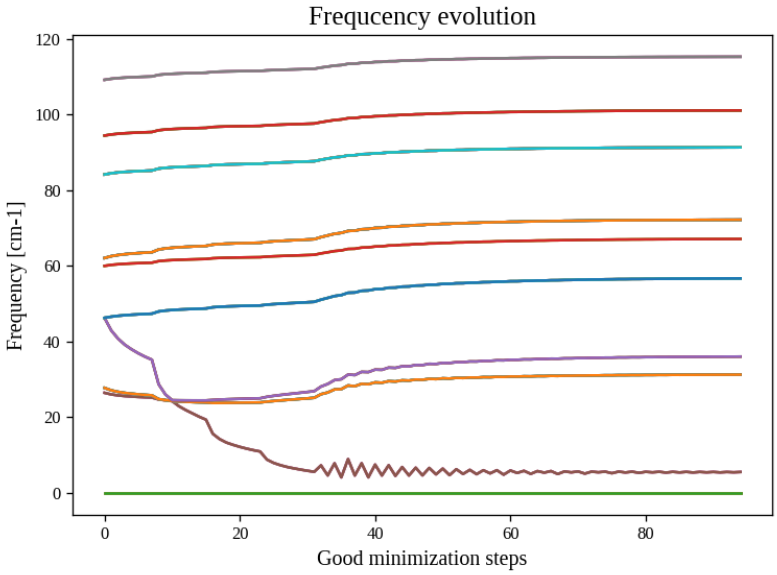
(continued from previous page)

```
# we will use a small optimization step
minim.set_minimization_step(0.25)

# Reduce the threshold for the gradient convergence
minim.meaningful_factor = 0.01

# If the minimization ends with few steps (less than 10),
# decrease it, if it takes too much, increase it

# We decrease the Kong-Liu effective sample size below
# which the population is stopped
minim.kong_liu_ratio = 0.5 # Default 0.5
# We relax the structure
relax = sscha.Relax.SSCHA(minim,
                  ase_calculator = ff_calculator,
                  N_configs = configurations,
                  max_pop = 50)

# Setup the custom function to print the frequencies
# at each step of the minimization
io_func = sscha.Utilities.IOInfo()
io_func.SetupSaving(File_frequencies)
# The file that will contain the frequencies is frequencies.dat

# Now tell relax to call the function to save the frequencies
# after each iteration
# CFP stands for Custom Function Post (Post = after the minimization step)
#relax.setup_custom_functions(custom_function_post = io_func.CFP_
↪SaveFrequencies)
relax.setup_custom_functions(custom_function_post = io_func.CFP_SaveAll)
# Finally we do all the free energy calculations.
relax.relax()
#relax.vc_relax(static_bulk_modulus=40, fix_volume = False)

# Save the final dynamical matrix
relax.minim.dyn.save_qe(File_final_dyn)
# Detect space group
symm=spglib.get_spacegroup(relax.minim.dyn.structure.get_ase_atoms(),
            0.005)
print('New SG = ', symm)
view(relax.minim.dyn.structure.get_ase_atoms())
```

This code will calculate the SSCHA minimization with the *ff_calculator*. We cat use **sscha-plot-data.py** to take a look at the minimization.

```
python sscha-plot-data.py frequencies.dat
```





Note: this force field model is not able to compute stress, as it is defined only at fixed volume, so we cannot use it for a variable cell relaxation.

Now we can search for instabilities.

If we have a very small mode in the SSCHA frequencies, it means that associated to that mode we have huge fluctuations. This can indicate an instability. However, to test this we need to compute the free energy curvature along this mode. This can be obtained in one shot thanks to the theory developed in Bianco et. al. Phys. Rev. B 96, 014111.

For that we create another program to do the job.

As before, we begin importing some libraries and setting variables:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

(continues on next page)

(continued from previous page)

```
#
#  SSCHA_exercise_Unstable.py
#
# Import the cellconstructor stuff
import cellconstructor as CC
import cellconstructor.Phonons
import cellconstructor.ForceTensor
import cellconstructor.Structure
import cellconstructor.Spectral

# Import the modules of the force field
import fforces as ff
import fforces.Calculator

# Import the modules to run the sscha
import sscha, sscha.Ensemble, sscha.SchaMinimizer
import sscha.Relax, sscha.Utilities

import spglib
from ase.visualize import view

# Import Matplotlib to plot
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import timeit

#Setting the variables:
#Setting the temperature in Kelvin:
Temperature = 0
#Setting the number of configurations:
configurations = 50
#Setting the names and location of the files:
Files_dyn_SnTe = "ffield_dynq"
#Set the number of irreducible q (reated to the supercell size):
nqirr = 3
#Setting the frequencies output file:
File_frequencies = "frequencies.dat"
#Setting the dynamical matrix output filename:
File_final_dyn = "final_sscha_T{}_".format(int(Temperature))
```

Now we look for that instability:

1. The *ff_calculator* toy potential is defined as we have seen in the previous program.

```
# Load the dynamical matrix for the force field
ff_dyn = CC.Phonons.Phonons("ffield_dynq", 3)

# Setup the forcefield with the correct parameters
ff_calculator = ff.Calculator.ToyModelCalculator(ff_dyn)
```

(continues on next page)

(continued from previous page)

```
ff_calculator.type_cal = "pbtex"
ff_calculator.p3 = 0.036475
ff_calculator.p4 = -0.022
ff_calculator.p4x = -0.014

# Initialization of the SSCHA matrix
dyn_sscha = CC.Phonons.Phonons(Files_dyn_SnTe, nqirr)
dyn_sscha.ForcePositiveDefinite()

# Apply also the ASR and the symmetry group
dyn_sscha.Symmetrize()
```

2. Next, we will load the dynamical matrix calculated previously with the *ff_calculator* toy potential, so there is no need to calculate it again.

```
# The SSCHA dynamical matrix is needed (the one after convergence)
# We reload the final result (no need to rerun the sscha minimization)
dyn_sscha_final = CC.Phonons.Phonons(File_final_dyn, nqirr)
```

3. Then, as the Hessian calculation is more sensible, we generate a new ensemble with more configurations. To compute the hessian we will use an ensemble of 10000 configurations. Note here that we can use less if we use Sobol sequence or we can load a previously generated ensemble.

```
# We reset the ensemble
ensemble = sscha.Ensemble.Ensemble(dyn_sscha_final, T0 = Temperature,
                    supercell = dyn_sscha_final.GetSupercell())

# We need a bigger ensemble to properly compute the hessian
# Here we will use 10000 configurations
ensemble.generate(5000, sobol = True)
#ensemble.generate(10000, sobol = False)
#We could also load the ensemble with
# ensemble.load("data_ensemble_final", N = 100, population = 5)
```

4. We now compute forces and energies using the force field calculator.

```
# We now compute forces and energies using the force field calculator
ensemble.get_energy_forces(ff_calculator, compute_stress = False)
```

5. Finally the free energy hessian is calculated in the *hessian* function. We can choose if we neglect or not in the calculation the four phonon scattering process. Four phonon scattering processes require a huge memory allocation for big systems, that scales as (3N)^4 with N the number of atoms in the supercell. Moreover, it may require also more configurations to converge.

   In almost all the systems we studied up to now, we found this four phonon scattering at high order to be negligible. We remark, that the SSCHA minimization already includes four phonon scattering at the lowest order perturbation theory, thus neglecting this term only affects combinations of one or more four phonon scattering with two three phonon scatterings (high order diagrams). For more details, see Bianco et. al. Phys. Rev. B 96, 014111.

   We can then print the frequencies of the hessian. If an imaginary frequency is present, then the system wants to spontaneously break the high symmetry phase.

```
print("Updating the importance sampling...")
# If the sscha matrix was not the one used to compute the ensemble
# We must update the ensemble weights
# We can also use this function to simulate a different temperature.
ensemble.update_weights(dyn_sscha_final, Temperature)
# ----------- COMPUTE THE FREE ENERGY HESSIAN -----------
print("Computing the free energy hessian...")
dyn_hessian = ensemble.get_free_energy_hessian(include_v4 = False)
# We neglect high-order four phonon scattering
#dyn_hessian = ensemble.get_free_energy_hessian(include_v4 = True,
#              get_full_hessian = True,verbose = True) # Full calculus
# We can save the free energy hessian as a dynamical matrix
# in quantum espresso format
dyn_hessian.save_qe("hessian")
# ------------------------------------------------------
# We calculate the frequencies of the hessian:
w_hessian, pols_hessian = dyn_hessian.DiagonalizeSupercell()

# Print all the frequency converting them into cm-1 (They are in Ry)
print("\n".join(["{:16.4f} cm-1".format(w * CC.Units.RY_TO_CM) for w in w_
↪hessian]))
```

The frequencies in the free energy hessian are temperature dependent.

We can look at the eigenmodes of the free energy hessian to check if we have imaginary phonons. If there are negative frequencies then we found an instability. You can check what happens if you include the fourth order.

Exercise

The Sobol sequences reduces the number of configurations by doing a better mapping of the gaussian than a random distribution. By uniformity spreading the samplings with a low discrepancy sequence like Sobol it is possible to reduce the number of configurations needed. Low discrepancy sequences tend to sample space "more uniformly" than random numbers. Algorithms that use such sequences may have superior convergence. You can test this in the calculation of the hessian by changing the number of configurations and the mapping scheme in the *ensemble.generate()* function.

4.2 Second order phase transition

Up to now we studied the system at T=0K and we found that there is an instability. However, we can repeat the minimization at many temperatures, and track the phonon frequency to see which is the temperature at which the system becomes stable.

Again we load and set the variables. Now the we have several temperatures so we store them in an array:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
#  SSCHA_exercise_Unstable.py
```

(continues on next page)

(continued from previous page)

```
#
# Import the cellconstructor stuff
import cellconstructor as CC
import cellconstructor.Phonons
import cellconstructor.ForceTensor
import cellconstructor.Structure
import cellconstructor.Spectral

# Import the modules of the force field
import fforces as ff
import fforces.Calculator

# Import the modules to run the sscha
import sscha, sscha.Ensemble, sscha.SchaMinimizer
import sscha.Relax, sscha.Utilities

import spglib
from ase.visualize import view

# Import Matplotlib to plot
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import timeit

#Setting the variables:
#Setting the temperature in Kelvin:
Temperature = 0
#Setting the number of configurations:
configurations = 50
#Setting the names and location of the files:
Files_dyn_SnTe = "ffield_dynq"
#Set the number of irreducible q (reated to the supercell size):
nqirr = 3
#Setting the frequencies output file:
File_frequencies = "frequencies.dat"
#Setting the dynamical matrix output filename:
File_final_dyn = "final_sscha_T{}_".format(int(Temperature))
sobol = False
sobol_scatter = False
```

1. Like in the previous program, first we prepare the Toy model force field

```
# Load the dynamical matrix for the force field
ff_dyn = CC.Phonons.Phonons("ffield_dynq", 3)

# Setup the forcefield with the correct parameters
ff_calculator = ff.Calculator.ToyModelCalculator(ff_dyn)
ff_calculator.type_cal = "pbtex"
```

(continues on next page)

(continued from previous page)

```
ff_calculator.p3 = 0.036475
ff_calculator.p4 = -0.022
ff_calculator.p4x = -0.014
```

2. We are going to need a range of temperatures for this calculation:

```
# Define the temperatures, from 50 to 300 K, 6 temperatures
temperatures = np.linspace(50, 300, 6)

lowest_hessian_mode = []
lowest_sscha_mode = []

# Perform a simulation at each temperature
t_old = Temperature
```

3. In the next part we condense the calculation of the hessians in a loop for different temperatures. In the end, it searches for the lowest non acoustic frequency to save with the correspondent auxiliar sscha frequency.

```
for Temperature in temperatures:
    # Load the starting dynamical matrix
    dyn = CC.Phonons.Phonons(File_final_dyn.format(int(t_old)), nqirr)

    # Prepare the ensemble
    ensemble = sscha.Ensemble.Ensemble(dyn, T0 = Temperature,
                                       supercell = dyn.GetSupercell())

    # Prepare the minimizer
    minim = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
    minim.min_step_struc = 0.05
    minim.min_step_dyn = 0.002
    minim.kong_liu_ratio = 0.5
    minim.meaningful_factor = 0.000001
    #minim.root_representation = "root4"
    #minim.precond_dyn = False
    #minim.minim_struct = True
    #minim.neglect_symmetries = True
    minim.enforce_sum_rule = True  # Lorenzo's solution to the error

    # Prepare the relaxer (through many population)
    relax = sscha.Relax.SSCHA(minim, ase_calculator = ff_calculator,
                  N_configs=configurations, max_pop=20)

    # Relax
    relax.relax(sobol = sobol, sobol_scramble = sobol_scatter)
    #relax.relax()

    # Save the dynamical matrix
    relax.minim.dyn.save_qe(File_final_dyn.format(int(Temperature)))
```

(continues on next page)

(continued from previous page)

```
    # Detect space group
    symm=spglib.get_spacegroup(relax.minim.dyn.structure.get_ase_atoms(),
                                       0.005)
    print('Current SG = ', symm,' at T=',int(Temperature))

    # Recompute the ensemble for the hessian calculation
    ensemble = sscha.Ensemble.Ensemble(relax.minim.dyn, T0 = Temperature,
                            supercell = dyn.GetSupercell())
    ensemble.generate(configurations, sobol = sobol,
                                       sobol_scramble = sobol_scatter)
    ensemble.get_energy_forces(ff_calculator, compute_stress = False)
    #gets the energies and forces from ff_calculator

    #update weights!!!
    ensemble.update_weights(relax.minim.dyn, Temperature)
    # Get the free energy hessian
    dyn_hessian = ensemble.get_free_energy_hessian(include_v4 = False)
    #free energy hessian as in Bianco paper 2017
    dyn_hessian.save_qe("hessian_T{}_".format(int(Temperature)))

    # Get the lowest frequencies for the sscha and the free energy hessian
    w_sscha, pols_sscha = relax.minim.dyn.DiagonalizeSupercell() #dynamical␣
↪matrix
    # Get the structure in the supercell
    superstructure = relax.minim.dyn.structure.generate_supercell(relax.minim.
↪dyn.GetSupercell())

    # Discard the acoustic modes
    acoustic_modes = CC.Methods.get_translations(pols_sscha,
                                   superstructure.get_masses_array())
    w_sscha = w_sscha[~acoustic_modes]

    lowest_sscha_mode.append(np.min(w_sscha) * CC.Units.RY_TO_CM) # Convert␣
↪from Ry to cm-1

    w_hessian, pols_hessian = dyn_hessian.DiagonalizeSupercell() #recomputed␣
↪dyn for hessian
    # Discard the acoustic modes
    acoustic_modes = CC.Methods.get_translations(pols_hessian,
                                     superstructure.get_masses_array())
    w_hessian = w_hessian[~acoustic_modes]
    lowest_hessian_mode.append(np.min(w_hessian) * CC.Units.RY_TO_CM) #␣
↪Convert from Ry to cm-1
    #print ("\n".join(["{:.4f} cm-1".format(w * CC.Units.RY_TO_CM) for w in␣
↪pols_hessian]))
    #exit()

    t_old = Temperature
# We prepare now the file to save the results
```

(continues on next page)

(continued from previous page)

```
freq_data = np.zeros( (len(temperatures), 3))
freq_data[:, 0] = temperatures
freq_data[:, 1] = lowest_sscha_mode
freq_data[:, 2] = lowest_hessian_mode

# Save results on file
np.savetxt("{}_hessian_vs_temperature.dat".format(configurations),
           freq_data, header = "T [K]; SSCHA mode [cm-1]; Free energy␣
↪hessian [cm-1]")
```

4. Finally we make a graphic output of the data.

```
hessian_data = np.loadtxt("{}_hessian_vs_temperature.dat".
↪format(configurations))

plt.figure(dpi = 120)
plt.plot(hessian_data[:,0], hessian_data[:,1],
                           label = "Min SCHA freq", marker = ">")
plt.plot(hessian_data[:,0], hessian_data[:,2],
                           label = "Free energy curvature", marker = "o")
plt.axhline(0, 0, 1, color = "k", ls = "dotted") # Draw the zero
plt.xlabel("Temperature [K]")
plt.ylabel("Frequency [cm-1]")
plt.legend()
plt.tight_layout()
plt.savefig('{}_Temp_Freq.png'.format(configurations))
#plt.show()

plt.figure(dpi = 120)
plt.plot(hessian_data[:,0], np.sign(hessian_data[:,2]) * hessian_data[:,2]**2,
                     label = "Free energy curvature", marker = "o")
plt.axhline(0, 0, 1, color = "k", ls = "dotted") # Draw the zero
plt.xlabel("Temperature [K]")
plt.ylabel("$\omega^2$ [cm-2]")
plt.legend()
plt.tight_layout()
plt.savefig('{}_Temp_Omeg.png'.format(configurations))
#plt.show()
```

We will simulate the temperatures up to room temperature (300 K) with steps of 50 K. Note, this will perform all the steps above 6 times, so it may take some minutes, depending on the PC (on a i3 from 2015, with one core, it took 2 hours). If it takes too long you can reduce the number of steps by changing the temperature array in *Temperature_i = np.linspace(50, 300, 6)*.

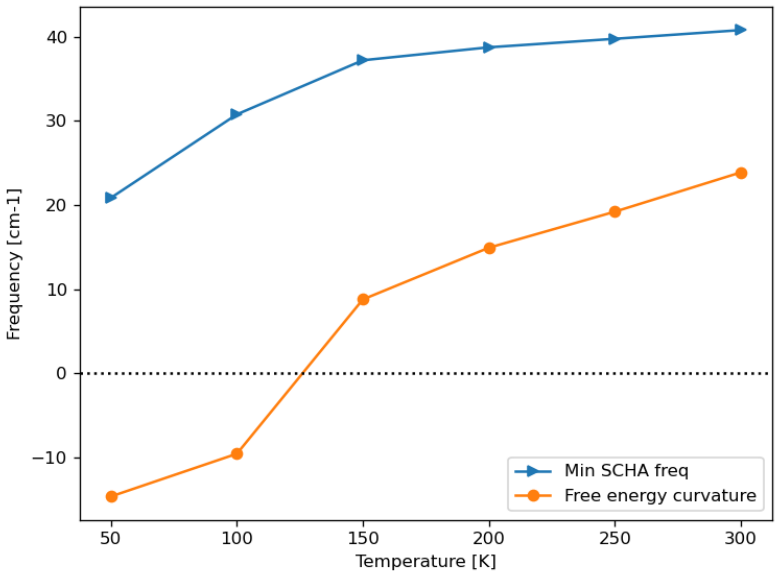


Fig. 4.2.1: Frequencies versus Temperatures

In *Frequencies versus Temperatures* we can see that the phase transition is between 100K and 150K. We see that the data points do not drawn a linear figure. We can increase the number of Temperature points to locate the exact transition temperature, but there is another better way to find it.

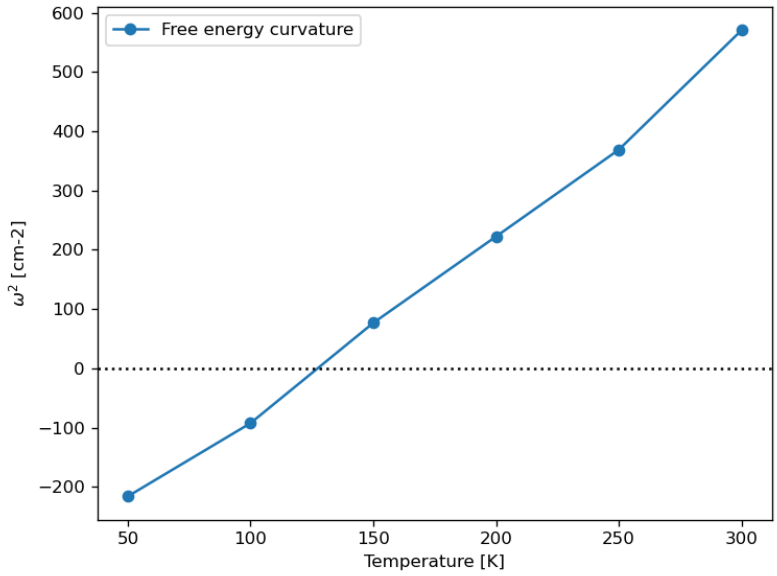


Fig. 4.2.2: squared Frequencies versus Temperatures.

For the Landau theory of phase transition, since the SSCHA is a mean-field approach, we expect that around the transition the critical exponent of the temperature goes as

$$\omega \sim \sqrt{\Phi}$$

For this reason is better to plot the temperature versus the square of the frequency as in *squared Frequencies versus Temperatures.* This makes the graph lineal and so we can easily estimate the critic temperature by linear interpolation.

We are using only 50 configurations in the ensemble. Note that this makes a fast calculation but is a low number for this calculations because the free energy calculations are more noisy than the SSCHA frequencies. This is due to the fact that the computation of the free energy requires the third order force constant tensor, and that requires more configurations to converge.

Exercise

How the calculation of the free energy changes with the number of configurations?

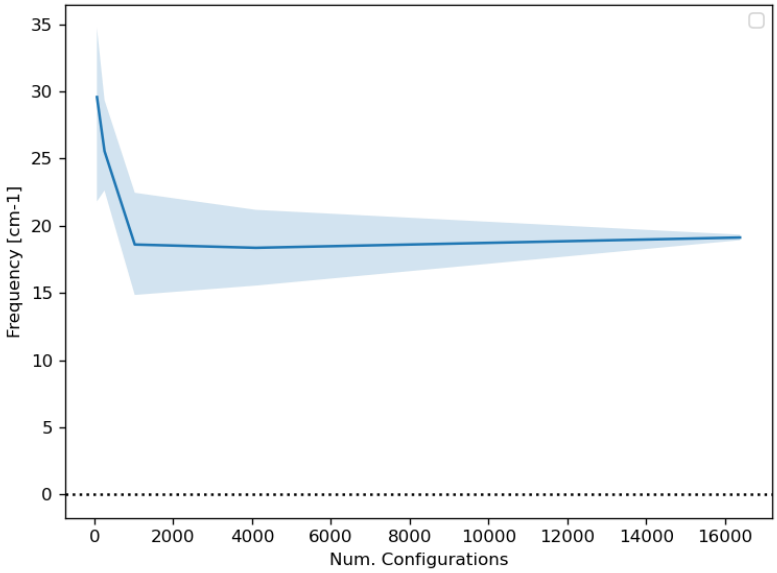
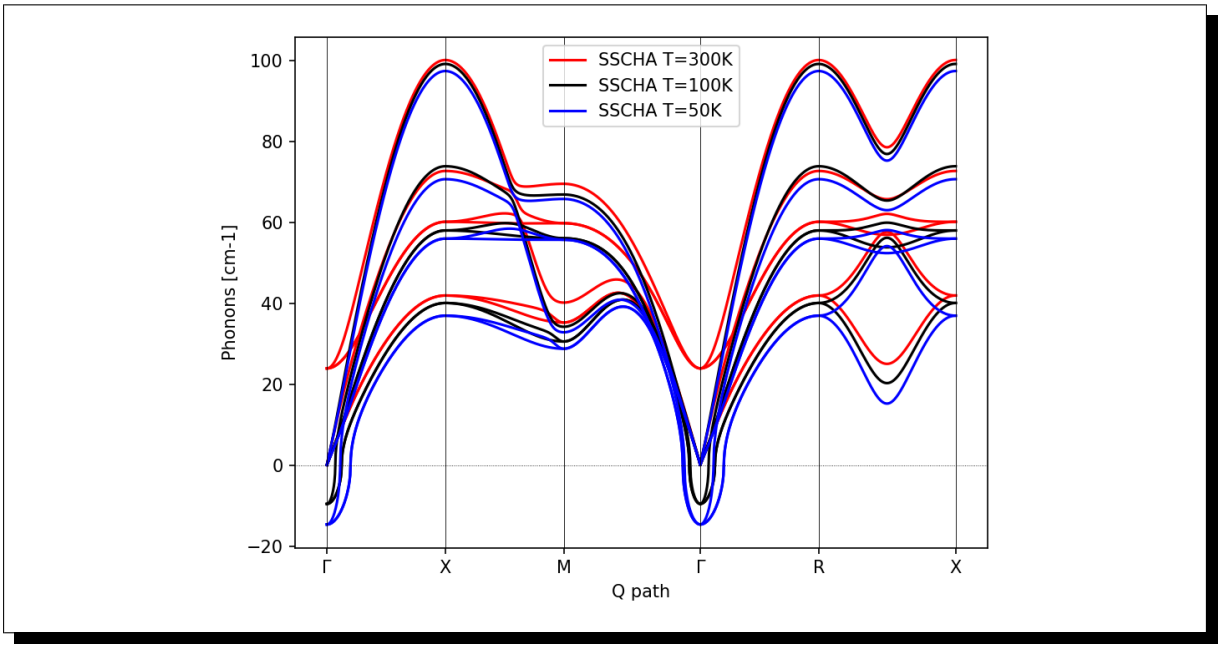


Fig. 4.2.3: Evolution of the lowest *soft* frequency in relation to the number of configurations in the ensemble with a stable configuration. The line is the media and the shade is the standard deviation.

Exercise

Plot the Hessian phonon dispersion


SSCHA T=300K
SSCHA T=100K
SSCHA T=50K
Phonons [cm-1]
100
80
60
40
20
0
−20
Γ
X
M
Γ
R
X
Q path

CHAPTER

FIVE

HANDS-ON-SESSION 4 - CALCULATION OF SPECTRAL PROPERTIES WITH THE SELF CONSISTENT HARMONIC APPROXIMATION

5.1 Theoretical introduction

The SCHA phonons are non-interacting quasiparticles that already include anharmonic effects (i.e. interaction between standard harmonic phonons) at some level. However, anharmoncity causes interaction between the SCHA phonons too. The interactions between phonons causes a change of their energy spectrum: from the overlap of simple Dirac-delta functions centered around the SSCHA phonon frequencies, to the overlap of Lorentzians with finite width (i.e. the quasiparticles have finite lifetime) and centered around shifted energies, or structures even more complex (when the anharmoncity is so strong that the quasiparticle picture has to be abandoned). For each $\boldsymbol{q}$ of the Brillouin zone, the SCHA phonons energy spectrum $\sigma(\boldsymbol{q}, \Omega)$ is given by Eq.(70)

$$\sigma(\boldsymbol{q}, \Omega) = -\frac{\Omega}{\pi} \operatorname{Im} \operatorname{Tr} \left[\boldsymbol{G}(\boldsymbol{q}, \Omega + \mathrm{i}0^+) \right] \tag{70}$$

where $\boldsymbol{G}$ is the SCHA phonons Green function, given by $\boldsymbol{G} = \boldsymbol{G}^{(0)} + \boldsymbol{G}^{(0)} \boldsymbol{\Pi}\, \boldsymbol{G}$, with $\boldsymbol{G}^{(0)}$ the Green function of the noninteracting SCHA phonons, and $\boldsymbol{\Pi}$ their selfenergy taking into account the interaction (in order to make easier the comparison with the literature, here and in the subsequent equations, the equation numbers refer to the paper here https://arxiv.org/abs/2103.03973). The SSCHA code allows to compute these quantities, even if in the current implementation the selfenergy can be computed only in the bubble approximation (Eq. (75))

$$\begin{aligned} \overset{(\mathrm{B})}{\Pi}_{\mu\nu}(\boldsymbol{q}, \Omega + i\delta_{\mathrm{se}}) = & \frac{1}{N_c} \sum_{\substack{\boldsymbol{k}_1 \boldsymbol{k}_2 \\ \rho_1 \rho_2}} \sum_{\boldsymbol{G}} \delta_{\boldsymbol{G}, \boldsymbol{q}+\boldsymbol{k}_1+\boldsymbol{k}_2} \\ & \times \mathscr{F}(\Omega + i\delta_{\mathrm{se}}, \omega_{\rho_1}(\boldsymbol{k}_1), \omega_{\rho_2}(\boldsymbol{k}_2)) \\ & \times \overset{(3)}{D}_{\mu\rho_1\rho_2}(-\boldsymbol{q}, -\boldsymbol{k}_1, -\boldsymbol{k}_2)\ \overset{(3)}{D}_{\rho_1\rho_2\nu}(\boldsymbol{k}_1, \boldsymbol{k}_2, \boldsymbol{q}) \,, \end{aligned} \tag{75}$$

i.e. the self-energy terms including the 4th order FCs are discarded. In this equation δ_{se} is an infinitely small positive number (a smearing parameter), that in actual calculations has to be chosen, toegher with the integration $\boldsymbol{k}$-grid, in order to find converged results.

The code, in addition to providing the ability to compute the spectral function through the full formula Eq. (70), allows for various approximations to be used in order to both compute the spectral function with reduced computational cost and conduct an analysis of the different contributions to the spectral function provided by each mode. The main approximation is to negeclet the off-diagonal terms in the self-energy written in the SCHA-modes basis set. In other words, we can negelct the possibility that the interaction mixes different SCHA phonons. In that case, the total spectrum is given by the the sum of the spectrum of each mode, $\sigma_\mu(\boldsymbol{q}, \Omega)$, as shown in Eq.(78),

$$\sigma(\boldsymbol{q}, \Omega) = \sum_\mu \sigma_\mu(\boldsymbol{q}, \Omega), \tag{78}$$

with $\sigma_\mu(\boldsymbol{q}, \Omega)$ having a generalized Lorentzian-like expression shown in Eq.(79)

$$\begin{aligned}\sigma_\mu(\boldsymbol{q}, \Omega) = \frac{1}{2}\Bigg[&\frac{1}{\pi}\frac{-\mathrm{Im}\ \mathcal{Z}_\mu(\boldsymbol{q}, \Omega)}{[\Omega - \mathrm{Re}\ \mathcal{Z}_\mu(\boldsymbol{q}, \Omega)]^2 + [\mathrm{Im}\ \mathcal{Z}_\mu(\boldsymbol{q}, \Omega)]^2} \\ &+ \frac{1}{\pi}\frac{\mathrm{Im}\ \mathcal{Z}_\mu(\boldsymbol{q}, \Omega)}{[\Omega + \mathrm{Re}\ \mathcal{Z}_\mu(\boldsymbol{q}, \Omega)]^2 + [\mathrm{Im}\ \mathcal{Z}_\mu(\boldsymbol{q}, \Omega)]^2}\Bigg]\end{aligned} \tag{79}$$

with $\mathcal{Z}_\mu(\boldsymbol{q}, \Omega)$ defined in Eq.(80)

$$\mathcal{Z}_\mu(\boldsymbol{q}, \Omega) = \sqrt{\omega_\mu^2(\boldsymbol{q}) + \Pi_{\mu\mu}(\boldsymbol{q}, \Omega + \mathrm{i}\delta_{\mathrm{se}})}. \tag{80}$$

where $\omega_\mu(\boldsymbol{q})$ is the frequency (energy) of the SCHA phonon $(\boldsymbol{q}, \mu)$, and $\boldsymbol{\Pi}_{\mu\mu}(\boldsymbol{q}, \Omega)$ is the corresponding diagonal element of the self-energy. This is the spectrum in the so called no mode-mixing approximation. At this level, the single-mode spectral functions resemble Lorentzian functions, but they are not true Lorentzians as they have kind of frequency-dependent center and width. As a matter of fact, in general, the spectrum of a mode can be very different from a true Lorentzian function, meaning that the quasiparticle picture for that mode is not appropriate. However, there are cases where the interaction between the SCHA phonons does not affect the quasiparticle picture but causes only a shift in the quasiparticle energy and the appearance of a finite linediwth (i.e. finite lifetime) with respect to the non-interacting case. In that case, we can write the spectral function of the mode as a true Lorentzian, Eq.(81),

$$\begin{aligned}\sigma_\mu(\boldsymbol{q}, \Omega) = \frac{1}{2}\Bigg[&\frac{1}{\pi}\frac{\Gamma_\mu(\boldsymbol{q})}{[\Omega - \Omega_\mu(\boldsymbol{q})]^2 + [\Gamma_\mu(\boldsymbol{q})]^2} \\ &+ \frac{1}{\pi}\frac{\Gamma_\mu(\boldsymbol{q})}{[\Omega + \Omega_\mu(\boldsymbol{q})]^2 + [\Gamma_\mu(\boldsymbol{q})]^2}\Bigg],\end{aligned} \tag{81}$$

i.e., the SCHA phonon $(\boldsymbol{q}, \mu)$ is a quasiparticle with definite energy $\Omega_\mu(\boldsymbol{q})$ ($\Delta_\mu(\boldsymbol{q}) = \Omega_\mu(\boldsymbol{q}) - \omega_\mu(\boldsymbol{q})$ is called the energy shift) and lifetime $\tau_\mu(\boldsymbol{q}) = 1/2\Gamma_\mu(\boldsymbol{q})$, where $\Gamma_\mu(\boldsymbol{q})$ is the Lorentzian half width at half maximum (HWHM). The quantities $\Omega_\mu(\boldsymbol{q})$ and $\Gamma_\mu(\boldsymbol{q})$ satisfy the relations given in Eqs.(82),(83)

$$\Omega_\mu(\boldsymbol{q}) = \mathrm{Re}\, \mathcal{Z}_\mu(\boldsymbol{q}, \Omega_\mu(\boldsymbol{q})) \tag{82}$$

$$\Gamma_\mu(\boldsymbol{q}) = -\mathrm{Im}\, \mathcal{Z}_\mu(\boldsymbol{q}, \Omega_\mu(\boldsymbol{q})). \tag{83}$$

Notice that the first one is a self-consistent equation. Instead of solving the self-consistent equation to evaluate $\Omega_\mu(\boldsymbol{q})$ two approximated approaches can be adpoted, both implemented in the SSCHA. One, that we call "one-shot", evaluates the r.h.s of Eq.(82) at the SCHA frequency (Eqs.(84))

$$\overset{(\mathrm{os})}{\Omega}_\mu(\boldsymbol{q}) = \mathrm{Re}\, \mathcal{Z}_\mu(\boldsymbol{q}, \omega_\mu(\boldsymbol{q})) \tag{84}$$

$$\overset{(\mathrm{os})}{\Gamma}_\mu(\boldsymbol{q}) = -\mathrm{Im}\, \mathcal{Z}_\mu(\boldsymbol{q}, \omega_\mu(\boldsymbol{q})). \tag{85}$$

This approximation is reasonable as long as the energy shift $\Delta_\mu(\boldsymbol{q}) = \Omega_\mu(\boldsymbol{q}) - \omega_\mu(\boldsymbol{q})$ is small. In particular, this is true if the SCHA self-energy is a (small) perturbation of the SCHA free propagator (not meaning that we are in a perturbative regime with respect to the harmonic approximation). In thas case, perturbation theory can be employed to evaluate the spectral function. If in Eq.(80) we keep only the first-order term in the self-energy, we get Eqs.(86),(87):

$$\overset{(\mathrm{pert})}{\Omega}_\mu(\boldsymbol{q}) = \frac{1}{2\omega_\mu(\boldsymbol{q})} \mathrm{Re}\, \Pi_{\mu\mu}(\boldsymbol{q}, \omega_\mu(\boldsymbol{q})) \tag{86}$$

$$\overset{(\mathrm{pert})}{\Gamma}_\mu(\boldsymbol{q}) = -\frac{1}{2\omega_\mu(\boldsymbol{q})} \mathrm{Im}\, \Pi_{\mu\mu}(\boldsymbol{q}, \omega_\mu(\boldsymbol{q}) + \mathrm{i}\delta_{\mathrm{se}}). \tag{87}$$

This concludes the overview on the quantities that we are going to compute for PbTe.

5.2 Calculations on PbTe

We will perform calculations on PbTe in the rock-salt structure and, in order to speed-up the calculation, we will employ the force-field model already used in previous tutorials (the force-field model can be downloaded and installed from here https://github.com/SSCHAcode/F3ToyModel). The calculations that we are going to perform are heavily underconverged and have to be consiered just as a guide to use of the SSCHA code. We will use a 2x2x2 supercell for PbTe. In order to define the force-field model on this superell we need three FCs, *PbTe.ff.2x2x2.dyn1*, *PbTe.ff.2x2x2.dyn2*, and *PbTe.ff.2x2x2.dyn3*

First, we need to do the SSCHA minimization. Create a directory *minim*, and go into it. We take the force-field FCs as starting point of the SSCHA mnimization too, with this input file *min.py*

```
import cellconstructor as CC
import cellconstructor.Phonons
import fforces as ff
import fforces.Calculator
import sscha, sscha.Ensemble, sscha.SchaMinimizer
import sscha.Relax, sscha.Utilities
```

(continues on next page)

(continued from previous page)

```
# ========================= TOY MODEL DEFINITION ============================
# Dynamical matrices that set up the harmonic part of the force-field
ff_dyn_name="../04_spectral_calculations/toy_matrices_2x2x2/PbTe.ff.2x2x2.dyn"
# Paramters that set up the anharmonic part of the force-field
p3 = -0.01408
p4 = -0.01090
p4x = 0.00254
# ===========================================================================

# ===========================================================
# dynamical matrices to be used as starting guess
dyn_sscha_name="../04_spectral_calculations/toy_matrices_2x2x2/PbTe.ff.2x2x2.
↪dyn"
# temperature
T=300
# minimization parameters
N_CONFIGS = 50
MAX_ITERATIONS = 10
# ===========================================================================
# Setup the harmonic part of the force-field
ff_dynmat = CC.Phonons.Phonons(ff_dyn_name, 3)
ff_calculator = ff.Calculator.ToyModelCalculator(ff_dynmat)
# Setup the anharmonic part of the force-field
ff_calculator.type_cal = "pbtex"
ff_calculator.p3 = p3
ff_calculator.p4 = p4
ff_calculator.p4x = p4x
# Load matrices
dyn_sscha=CC.Phonons.Phonons( dyn_sscha_name,3)
dyn_sscha.Symmetrize()
# Generate the ensemble
supercell=dyn_sscha.GetSupercell()
ens = sscha.Ensemble.Ensemble(dyn_sscha, T, supercell)
ens.generate(N_CONFIGS)
# Compute energy and forces for the ensemble elements
ens.get_energy_forces(ff_calculator , compute_stress = False)
# Set up minimizer
minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(ens)
# Ignore the structure minimization (is fixed by symmetry)
minimizer.minim_struct = False
# max number steps (negative infinite)
minimizer.max_ka=-1
# Setup the minimization parameter for the covariance matrix
minimizer.set_minimization_step(1.0)
# Setup the threshold for the ensemble wasting
minimizer.kong_liu_ratio =0.8     # Usually 0.5 is a good value
minimizer.meaningful_factor=1e-5  # meaningul factor
# Initialize the simulation
```

(continues on next page)

(continued from previous page)

```
relax = sscha.Relax.SSCHA(minimizer,
                          ff_calculator,
                          N_configs = N_CONFIGS,
                          max_pop = MAX_ITERATIONS)
# Define the I/O operations
# To save info about the free energy minimization after each step
ioinfo = sscha.Utilities.IOInfo()
ioinfo.SetupSaving("minim_info")
relax.setup_custom_functions(custom_function_post = ioinfo.CFP_SaveAll)
# Start the minimization
relax.relax()
# Print in stdout the info about the minimization
# and save the final dynamical matrix
relax.minim.finalize()
relax.minim.ensemble.save_bin("./data_pop", population_id=1)
relax.minim.dyn.save_qe("SSCHA.T{}.dyn".format(T))
```

launching

```
$ python min.py > min.out
```

after a few second the minimization has concluded. The three SSCHA FCs matrices have been saved as *SSCHA.T300.dyn#q*. Now we need to compute the third order FCs (FC3s) (and the Hessian FCs). At the end of the SSCHA minimization, we saved the last population and the dynamical matrices that generated it too. Using them, we compute the Hessian matrices and the FC3s. Exit from *minim*, create a directory *hessian*, go into it, and use this input file *hessian.py*

```
import cellconstructor as CC
import sscha.Ensemble
#
NQIRR = 3
Tg = 300
T = 300
#
pop_dyn = CC.Phonons.Phonons('../minim/data_pop/dyn_gen_pop1_', 3)
sscha_dyn = CC.Phonons.Phonons('../minim/SSCHA/SSCHA.T300.dyn', 3)
#
ens = sscha.Ensemble.Ensemble(pop_dyn, Tg)
ens.load_bin('../minim/data_pop', population_id = 1)
ens.update_weights(sscha_dyn, T)
#
hessian_dyn, d3 = ens.get_free_energy_hessian(include_v4 = False,
                                              return_d3 = True)

hessian_dyn.save_qe('Hessian.dyn')

################################ FC3 part ######################################
########

tensor3 = CC.ForceTensor.Tensor3(dyn=sscha_dyn)          # initialize 3rd␣
↪order tensor
```

(continues on next page)

(continued from previous page)

```
tensor3.SetupFromTensor(d3)                                    # assign values
tensor3.Center()                                               # center it
tensor3.Apply_ASR()                                            # apply ASR
tensor3.WriteOnFile(fname="FC3",file_format='D3Q')             # write on file
```

giving

```
$ python hessian.py > hessian.out
```

We already did an Hessian calculation in a previous tutorial. The new part is the creation of the 3rd order FCs (FC3s), which we wrote in the file *FC3*. Some comments about the input file. In `get_free_energy_hessian` we set `return_d3 = True` because we need these informations to set up the FC3s. Moreover, since we are not going to use forth order FCs (we will work within the "bubble approximation"), we set `include_v4 = False`, which in general saves a lot of computation time. Before writing the FC3s on file, we center it (a step necessary to perform Fourier interpolation), and apply the acoustic sum rule (ASR), since the centering spoils it.s

The format chosen here to write the FC3 file is the same used in the d3q.x code https://anharmonic.github.io/d3q/. To be precise: exploiting the lattice translation symmetry, the third order FCs can be written as $\Phi^{\alpha_1,\alpha_2,\alpha_3}_{a_1a_2a_3}(0,\boldsymbol{R},\boldsymbol{S})$, where $\alpha_1,\alpha_2,\alpha_3$ are cartesian indices, a_1,a_2,a_3 atomic indices in the unit cell, and $\boldsymbol{R}$, $\boldsymbol{S}$ lattice vectors. The FC3 file in D3Q format is

```
alpha_1 alpha_2 alpha_3 at_1 at_2 at_3
N_RS
R_x R_y R_z S_x S_y S_z phi(alpha_1,at_1,alpha_2,at_2,alpha_3,at_3)

...

alpha_1 alpha_2 alpha_3 at_1 at_2 at_3
N_RS
R_x R_y R_z S_x S_y S_z phi(alpha_1,at_1,alpha_2,at_2,alpha_3,at_3)

...
```

For each `alpha_1 alpha_2 alpha_3 at_1 at_2 at_3` we have a block where: the first line is `N_RS`, which is the number of $\boldsymbol{R}$, $\boldsymbol{S}$ considered. Each subsequent line refers to a couple $\boldsymbol{R}$, $\boldsymbol{S}$, with `R_x R_y R_z` and `S_x S_y S_z` the crystal coordinates of $\boldsymbol{R}$ and $\boldsymbol{S}$, respectively, and `phi(alpha_1,at_1, alpha_2,at_2,alpha_3,at_3)` the corresponding FCs value $\Phi^{\alpha_1\alpha_2\alpha_3}_{a_1a_2a_3}(0,\boldsymbol{R},\boldsymbol{S})$.

Equipped with the third order SSCHA FCs ,written in real space in the *FC3* file, and the second-order SSCHA FCs, written in reciprocal space in the *SSCHA.T300.dyn#q* files, we have all the ingredient to compute the spectral functions. As first calculation, we compute the spectral function using Eq.(70) but within the "static approximation", this meaning that we keep the selfenergy blocked with $\Omega = 0$, as shown in Eq.(66) (within the bubble approximation)

$$\overset{(\mathrm{B})}{\Pi}_{\mu\nu}(\boldsymbol{q},0) = \frac{1}{N_{\boldsymbol{k}}} \sum_{\substack{\boldsymbol{k}_1\boldsymbol{k}_2\\ \rho_1\rho_2}} \sum_{\boldsymbol{G}} \delta_{\boldsymbol{G},\boldsymbol{q}+\boldsymbol{k}_1+\boldsymbol{k}_2} \mathscr{F}(0,\omega_{\rho_1}(\boldsymbol{k}_1),\omega_{\rho_2}(\boldsymbol{k}_2)) \times \overset{(3)}{D}_{\mu\rho_1\rho_2}(-\boldsymbol{q},-\boldsymbol{k}_1,-\boldsymbol{k}_2)\, \overset{(3)}{D}_{\rho_1\rho_2\nu}(\boldsymbol{k}_1,\boldsymbol{k}_2,\boldsymbol{q}) \ . \tag{66}$$

In order to do that, exit from the current directory *hessian*, create a directory *spectral_static*, enter into it, and use this input file *spectral_static.py* to compute the spectral function in the static approximation, for the special point X

```
import cellconstructor as CC
import cellconstructor.ForceTensor

dyn = CC.Phonons.Phonons("../minim/SSCHA.T300.dyn",3)
FC3 = CC.ForceTensor.Tensor3(dyn=dyn)
FC3.SetupFromFile(fname="../hessian/FC3",file_format='D3Q')


# integration grid
k_grid=[2,2,2]

# X in 2pi/Angstrom
points=[0.0,-0.1547054, 0.0]

CC.Spectral.get_full_dynamic_correction_along_path(dyn=dyn,
                                                   tensor3=FC3,
                                                   k_grid=k_grid,
                                                   e1=100, de=0.1, e0=0,     #␣
↪energy grid
                                                   T=300,
                                                   q_path=points,
                                                   static_limit = True,
                                                   filename_sp='full_spectral_
↪func_X')
```

We used as integration $\boldsymbol{k}$-grid (i.e. the $\boldsymbol{k}$-grid of the summation in Eqs.(66), (75)) the grid commensurate with the supercell, i.e. a 2x2x2 grid. In that case, using the centering is irrelevant, as there is no Fourier interpolation. With

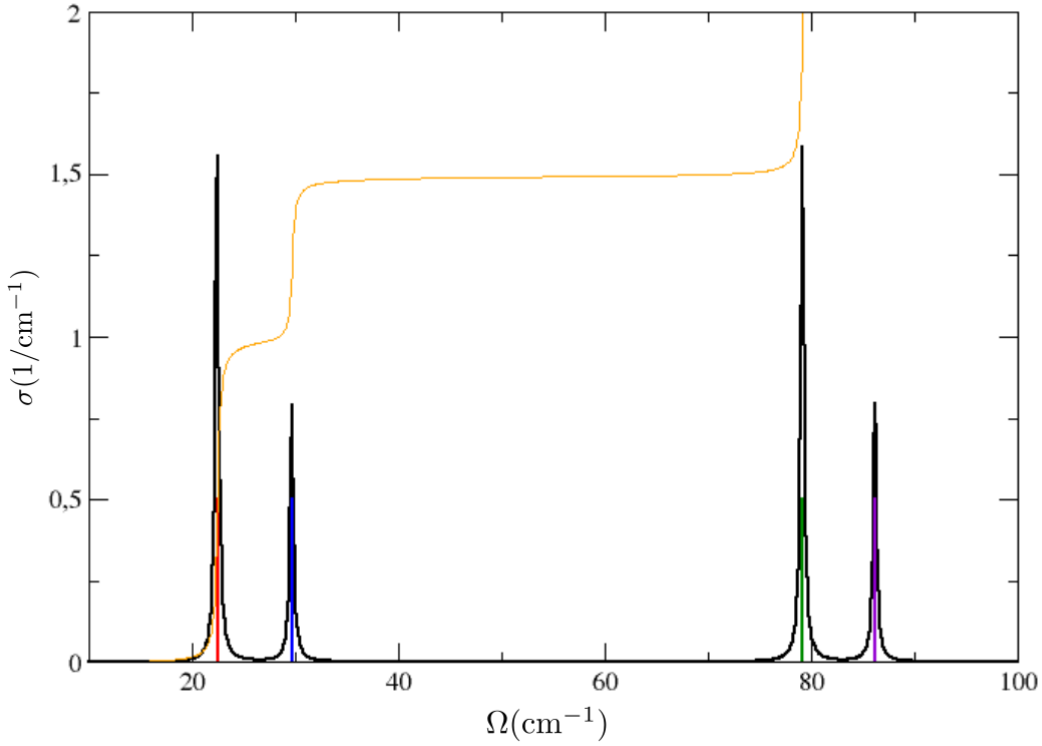
```
$ mpirun -np 4 python spectral_static.py > spectral_static.out
```

we run the code with MPI with 4 parallel processes. In output we have the file *full_spectral_func_X_static.dat* that contains the spectral function

```
# ------------------------------------------------------------
# len (2pi/Angstrom), energy (cm-1), spectral function (1/cm-1)
# ------------------------------------------------------------
 0.000000        0.0000000       0.0000000
 0.000000        0.1000000       0.0000000
 0.000000        0.2000000       0.0000000
 0.000000        0.3000000       0.0000001
 0.000000        0.4000000       0.0000002
 0.000000        0.5000000       0.0000003

                   ...
```

where the first line indicates the lenght of the path. This would be relevant in case we had not just a single $\boldsymbol{q}$ point, but a path of $\boldsymbol{q}$-points. In that case, we would have several blocks, one for each $\boldsymbol{q}$ point,

and the first column of each block would indicate the lenght of the $\boldsymbol{q}$-path. In this case, since we have just one point, we have just one block with the first column equal to zero. Plotting the 3rd vs 2nd column we obtain this result:



We have Dirac deltas (to be precise, extremely narrow Lorentzians whose width is given only by the choice of the finite size of the used energy grid) around values that coincides with the Hessian frequency values (plotted here with vertical lines), that you can find in the `Hessian.dyn3` file obtained in the previous run. Indeed, the Hessian calculation corresponds exactly to a calculation done with the static self-energy. Two observations. The height of the spikes is proportional to the degeneracy of the modes. The yellow line indicates the integral function $\int_0^{\Omega} \sigma(\Omega', \boldsymbol{q})\, d\Omega'$, which at the end returns the value: [number of modes]/2 (therefore 3 in this case). This is a general sum rule fulfilled by the spectral function (not only in the static approximation).

Now we do a full calculation (no static approximation anymore). In this case, we need to specify the smearing parameter δ_{se} to compute the dynamic selfenergy from Eq.(75). In order to be tidy, let us do this calculation in another directory *spectral* (and let us do the same for all the subsequent calculations, new calculations in new directories). Using this *spectral.py* input file

```
import cellconstructor as CC
import cellconstructor.ForceTensor

dyn = CC.Phonons.Phonons("../minim/SSCHA.T300.dyn",3)
FC3 = CC.ForceTensor.Tensor3(dyn=dyn)
FC3.SetupFromFile(fname="../hessian/FC3",file_format='D3Q')
```

(continues on next page)

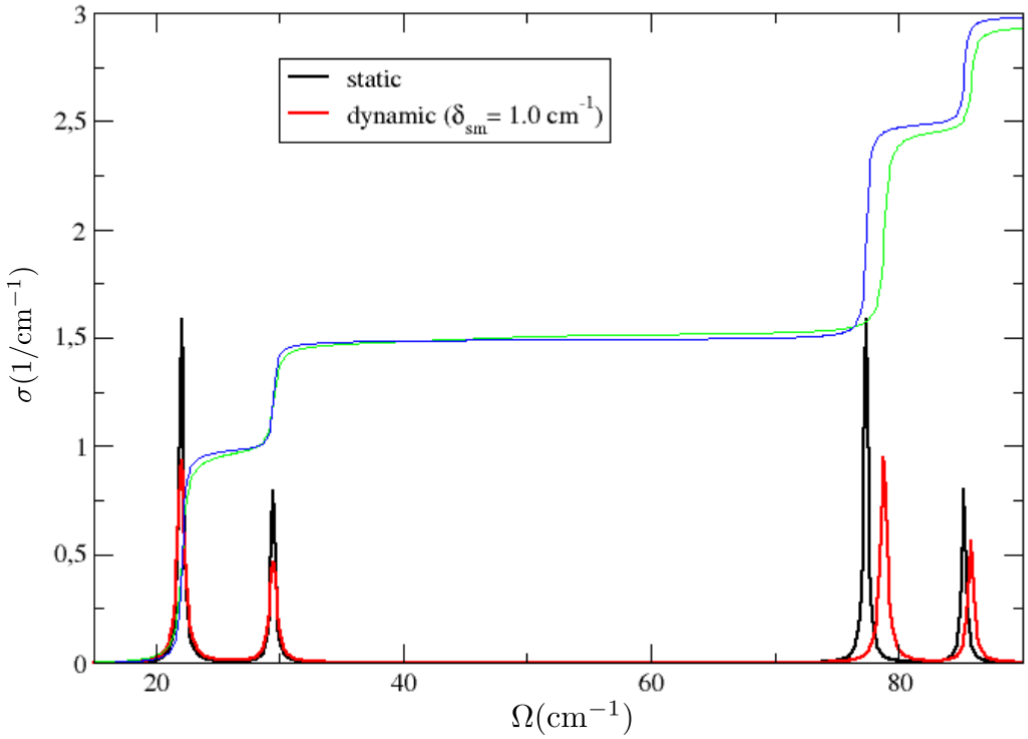
(continued from previous page)

```
# integration grid
k_grid=[20,20,20]

# X in 2pi/Angstrom
points=[0.0,-0.1547054, 0.0]

CC.Spectral.get_full_dynamic_correction_along_path(dyn=dyn,
                                                   tensor3=FC3,
                                                   k_grid=k_grid,
                                                   e1=100, de=0.1, e0=0,      # energy␣
↪grid
                                                   sm1=10.0,sm0=1.0,nsm=3,    #␣
↪smearing values
                                                   T=300,
                                                   q_path=points,
                                                   static_limit = True,
                                                   filename_sp='full_spectral_func_X')
```

where now we have specified that we want to do the calculation with 3 smearing values (equally spaced) between 1.0 and 10.0 cm-1 (thus we will have 1.0, 5.5, and 10.0 cm-1) With

```
$ mpirun -np 4 python spectral.py > spectral.out
```

in output we have three files with the spectral functions, one for each smearing value. In general, convergence must be studied with respect to the integration $\boldsymbol{k}$-grid and smearing used. Plotting the static result and the dynamic result for sm=1.0 cm-1, both computed with 20x20x20 $\boldsymbol{k}$-grid, we see this result

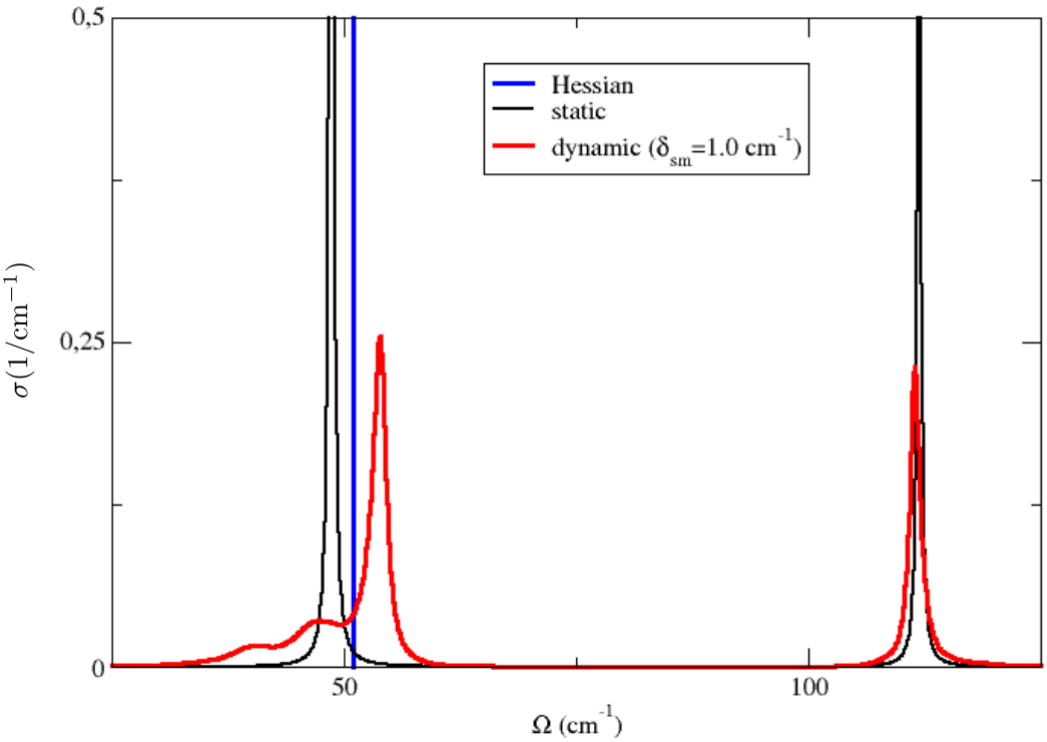


Therefore, we can conclude that in X the SSCHA phonons are barely affected by the interaction. However, the situation is different if analogous calculation is done in Γ.

Exercise

In Γ, do the same calculations previously done in X, and plot the results

This is the result you should obtain



Notice that here the triple-degenerate optical mode of the Hessian dynamical matrix is splitted into due different peaks of the static spectral function (LO and double degenerate TO). This is due to the LO-TO splitting occurring in PbTe. The frequencies in the Hessian dynamical matrix in Γ refer only to the short-range part of the FCs. However, the long-range dipole-dipole contribution coming from the Effective Charges (nonanalytic contribution), which is at the origin of the LO-TO splitting, is taken into account when the spectral function is computed. Moreover, notice that when dynamic spectral function is considered, the double-degenerate TO mode gets smeared, showing a strong non-Lorentzian character. When 4x4x4 supercell calculations are performed, it clearly appears a satellite peak.

Before continuing the spectral analysis, let us spend some time to investigate the static correction. As said, the static spectral function is nothing but a collection of Dirac-deltas centered around the Hessian eigenvalues. Therefore, in the static case the only information are the eigenvalues, there is not a complex spectrum to be analyzed. Indeed, there is a routine that we can use to compute the static correction for any q point, and for any integration grid. With this input file *static.py*

```
import cellconstructor.ForceTensor
import cellconstructor as CC
import numpy as np # will be used just to create a path

dyn = CC.Phonons.Phonons("../minim/SSCHA.T300.dyn",3) # SSCHA matrices
FC3 = CC.ForceTensor.Tensor3(dyn=dyn)
FC3.SetupFromFile(fname="../hessian/FC3",file_format='D3Q')
```

(continues on next page)

(continued from previous page)

```
# integration grid
k_grid=[4,4,4]

Xcoord=0.1547054
points=[[0.0,z,0.0] for z in np.linspace(-Xcoord,Xcoord,100)] # create the␣
↪path
                                    # you can also download the path from a␣
↪file

CC.Spectral.get_static_correction_along_path(dyn=dyn,
                                             tensor3=FC3,
                                             k_grid=k_grid,
                                             T=300,
                                             q_path=points)
```

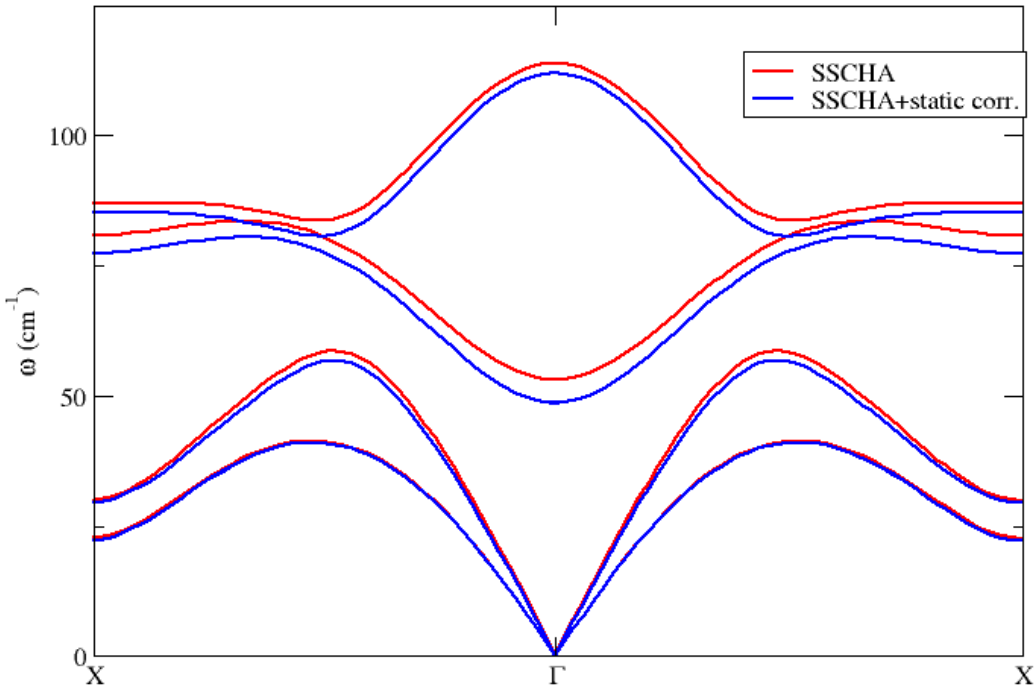
with

```
$ mpirun -np 4 python static.py > static.out
```

we obtain the file *v2+d3static_freq.dat*, done like this

```
# ------------------------------------------------------------------------
# len (2pi/Angstrom), sscha freq (cm-1), sscha + static bubble freq (cm-1)
# ------------------------------------------------------------------------
 0.000000    22.6699858  22.6699858 ...        22.1602770  22.1717260 ...
 0.003125    22.7847829  22.7847829 ...        29.9763683  80.8414572 ...

                               ...
```

where the first column is the lenght of the path in 2:math:*pi*/Angstrom, the next (6, in this case) columns are the SSCHA frequencies, and the next (6, in this case) columns are the SSCHA+static bubble self-energy-corrected frequencies. This is the plot obtained with this result



Therefore, as long as one is interested only in the static correction, e.g. because one wants to study the structural instability, the routine `get_static_correction_along_path` is the one that has to be employed. Indeed, notice that this is the proper way to detect instabilities (imaginary frequencies) in points of the Brillouin zone that do not belong to the grid used to compute the Hessian. One should not Fourier interpolate the Hessian matrices computed on a grid, in order to obtain the frequency dispersion along a path, but rather should interpolate the correction and add it to the SSCHA frequency, point by point (which is what we are doing here). Moreover, in this way we can increase the integration k-grid to reach the convergence. Notice that the LO-TO splitting has been properly taken into account.

We can now go back to the spectral calculations. In general, calculations done with Eq.(70) can be heavy, but often the off-diagonal terms of the phonon self-energy in the mode basis set can be neglected and use Eqs. (78), (79), (80). This is the case of PbTe. With this input file *nomm_spectral.py*

```
import cellconstructor as CC
import cellconstructor.ForceTensor

dyn = CC.Phonons.Phonons("../minim/SSCHA.T300.dyn",3)
FC3 = CC.ForceTensor.Tensor3(dyn=dyn)
FC3.SetupFromFile(fname="../hessian/FC3",file_format='D3Q')


# integration grid
k_grid=[20,20,20]
```

(continues on next page)

(continued from previous page)

```
G=[0.0,0.0,0.0]
points=G

CC.Spectral.get_diag_dynamic_correction_along_path(dyn,
                                                   tensor3=FC3,
                                                   k_grid=k_grid,
                                                   e1=150, de=0.1, e0=0.0,
                                                   sm1=1.0, sm0=1.0,
                                                   nsm=1,
                                                   q_path=points,
                                                   T=300.0)
```

and

```
$ mpirun -np 4 python nomm_spectral.py > nomm_spectral.out
```

we obtain several files:

- *spectral_func_1.00.dat*

  with the structure

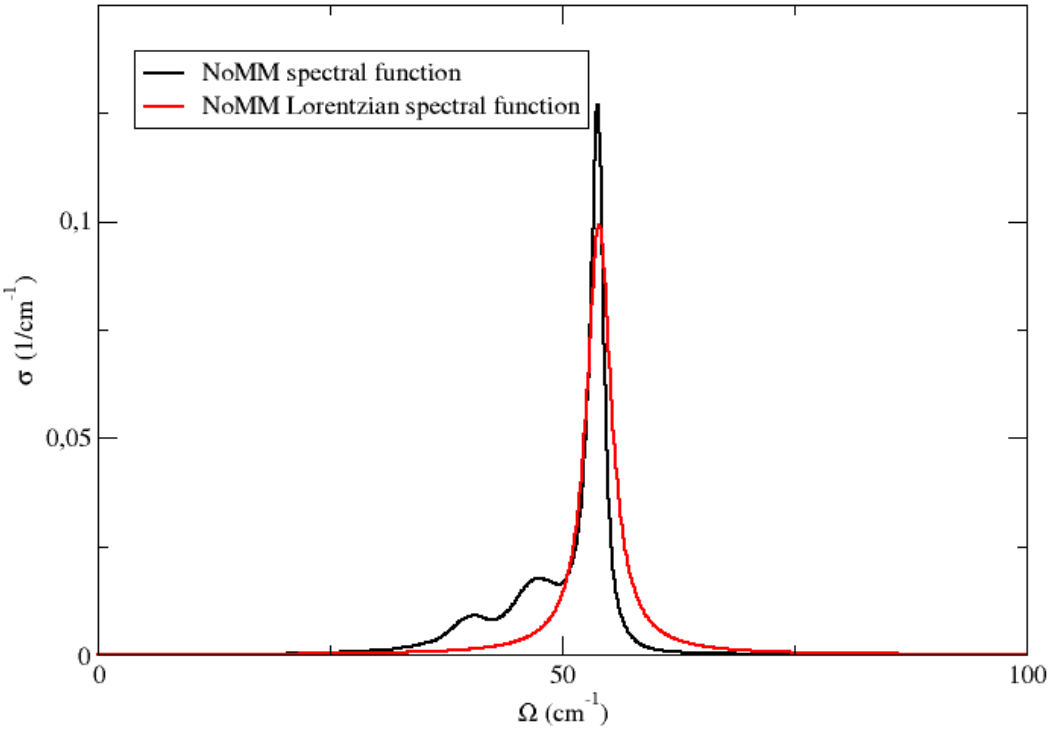
```
# ------------------------------------------------------------------
↪------------------
# len (2pi/Ang), ene. (cm-1), spec. func. (1/cm-1), spec. fun.␣
↪mode comp. (1/cm-1)
# ------------------------------------------------------------------
↪------------------
  0.000         0.000         0.000           -0.000          0.
↪000   ...
  0.000         0.100         1.527            0.509          0.
↪509   ...
  0.000         0.200         2.387            0.795          0.
↪795   ...
                           ...
```

  The file is made by several blocks, one for each q point of the path considered (now we have just one point, thus one block). The first column of the block gives the length of the path (at that point). The second and third column give the energy and total spectrum, respectively. The subsequent columns give the contribution to the spectrum given by each mode (Cfr. Eqs.(78),(79)). Plotting the 3rd vs 2nd column you can verify that in this case the spectrum is essentially identical to the one already obtained with the full formula (i.e. to the spectrum obtained considering the off-diagonal terms of the self-energy too). Notice that in this file we have the spectrum given by the three acoustic modes in Γ. We did not consider the translational modes (because trivial) in the calculation done with *get_full_dynamic_correction_along_path*, where the flag `notransl` by default was set equal to `True`.

- *spectral_func_lorentz_one_shot_1.00.dat*, *spectral_func_lorentz_perturb_1.00.dat*

  These files have the same structure of *spectral_func_1.00.dat*. However, now the spectral functions are computed in the Lorenztian approximation, Eq.(81), using the one-shot, Eqs.(84),(85) and the perturbative, Eqs.(86),(87), values of the energy

and HWHM. The codes offers also the possibility to use a still very tentative approach to solve the self-consistent relation Eqs.(82),(83), and produce the relative Lorentzian spectral functions. Plotting the spectral functions of the TO mode from *spectral_func_1.00.dat* and *spectral_func_lorentz_one_shot_1.00.dat* we obtain this



This confirms the strong non-Lorentzian character of this mode

- *v2_freq_shift_hwhm_one_shot_1.00.dat, v2_freq_shift_hwhm_perturb_1.00.dat*

  They have the structure

```
# ------------------------------------------------------------------
↪--
# len (2pi/Angstrom), SSCHA freq (cm-1), shift (cm-1) , HWHM (cm-
↪1)
# ------------------------------------------------------------------
↪--

                        ....
```

  For each $\boldsymbol{q}$ point there is a line, with the lenght along the path, the SSCHA frequencies for that point, $\omega_\mu(\boldsymbol{q})$, the energy shift $\Delta_\mu(\boldsymbol{q}) = \Omega_\mu(\boldsymbol{q}) - \omega_\mu(\boldsymbol{q})$ and the HWHM $\Gamma_\mu(\boldsymbol{q})$.

- *freq_dynamic_one_shot_1.00.dat, freq_dynamic_perturb_1.00.dat*

  They have the structure

```
# ------------------------------------------------------------
# len (2pi/Angstrom), SSCHA+shift (sorted) (cm-1), HWHM (cm-1)
# ------------------------------------------------------------

                        ....
```

For each $\boldsymbol{q}$ point there is a line, with the lenght along the path, the shifted frequencies $\Omega_\mu(\boldsymbol{q}) = \omega_\mu(\boldsymbol{q}) + \Delta_\mu(\boldsymbol{q})$ (sorted in incresing value, in principle different from the SSCHA-frequency increasing order) and the corresponding HWHMs $\Gamma_\mu(\boldsymbol{q})$. This is the file that has to be used to plot the correct phonon dispersion, together with the linewidth.

There is a dedicated routine to compute, with less computational time and more accuracy, only the energy shift (i.e. the corrected frequency) and the linewidth of the modes in the one-shot and perturbative no-mode-mixing Lorentzian approach. It is the `get_os_perturb_dynamic_correction_along_path` routine, using the *os_perturb_correction.py* input file

```
import cellconstructor as CC
import cellconstructor.ForceTensor

dyn = CC.Phonons.Phonons("../minim/SSCHA.T300.dyn",3)
FC3 = CC.ForceTensor.Tensor3(dyn=dyn)
FC3.SetupFromFile(fname="../hessian/FC3",file_format='D3Q')


# integration grid
k_grid=[10,10,10]

points=[0.0,0.0,0.0]

CC.Spectral.get_os_perturb_dynamic_correction_along_path(dyn,
                                                         tensor3=FC3,
                                                         k_grid=k_grid,
                                                         sm1=1.0, sm0=1.0,
                                                         nsm=1,
                                                         q_path=points,
                                                         T=300.0)
```

Exercise

Do this calculation to obtain the dispersion along the path $X - \Gamma - X$.

After that, using this script to extract the results

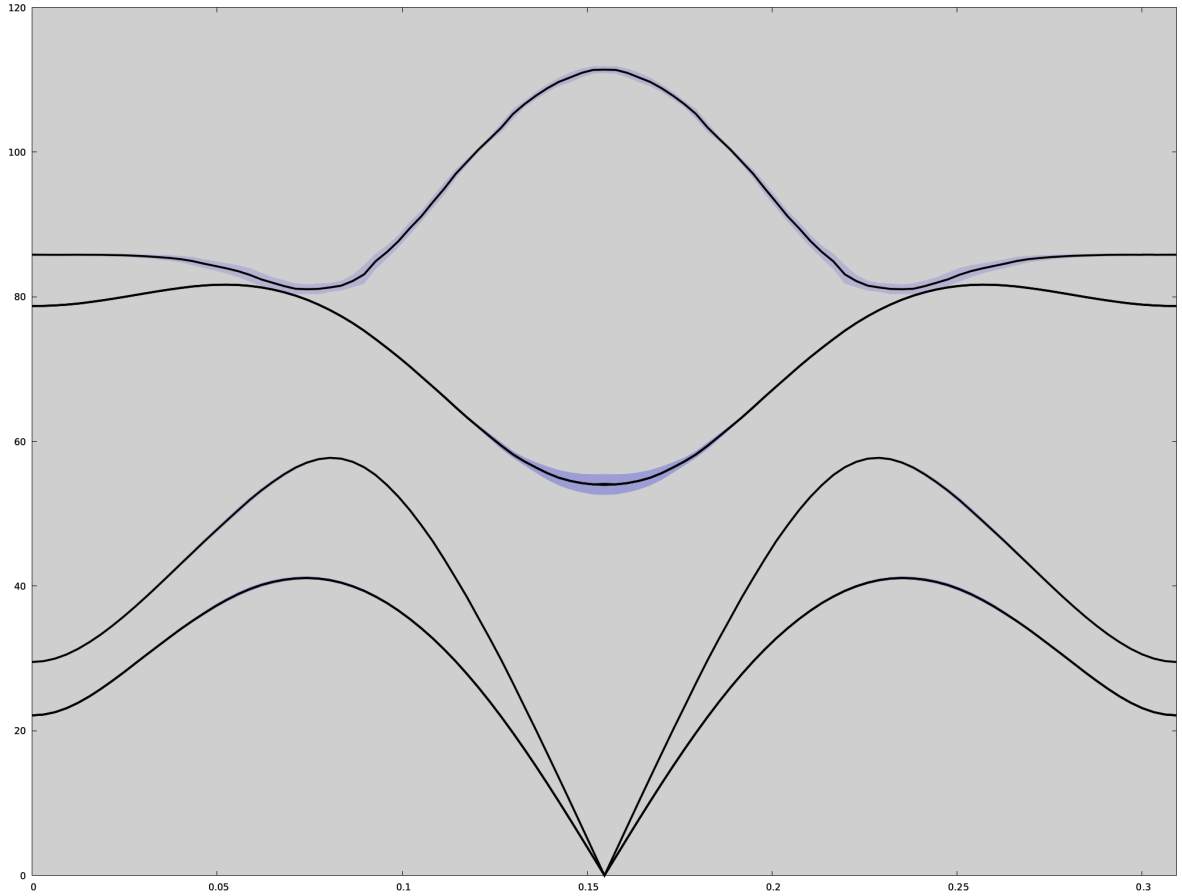
```
for i in `seq 1 6`
do
 awk -v i="${i}" '{ if ( NR  > 3 ) printf"%22.11f%22.11f%22.11f%22.11f\n", \
 $1,$(i+1),$(i+1)+$(i+7),$(i+1)-$(i+7)}' freq_dynamic_1.0.os.dat > freq_${i}
done
```

and then using this gnuplot script *plot.gp*

```
set terminal pngcairo size 2048,1536 enhanced
set output 'freq.png'

xmin=0
xmax=0.30941100000
set xrange[xmin:xmax]

set multiplot
set object rectangle from graph 0,0 to graph 1,1 \
behind fillcolor rgb 'black' fillstyle transparent solid 0.100 noborder

plot for [i=1:6] 'freq_'.i using 1:3:4 with filledcurves \
    fs transparent solid 0.125 noborder lc rgb "blue" notitle
plot for [i=1:6] 'freq_'.i using 1:2   with lines \
    lc rgb "black" lw 3.5 notitle
```

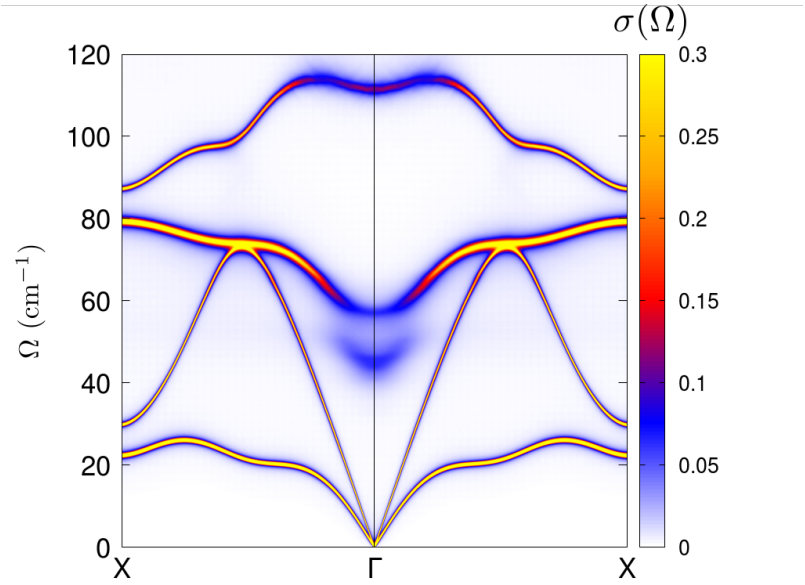
```
$ gnuplot plot.gp
```

you obtain this figure



Here you have the plot of the shifted SSCHA phonon frequencies with the linewidth. However, it must be remembered that this picture is appropriate as long as the Lorentzian picture is valid. We already

know that at least in Γ this is not really the case (as said, this is even more evident if the calculation is done with a 4x4x4 supercell). In that case, the best thing to do is a spectral calculation (full, or in the no-mode-mixing approximation), and use the three columns (lenght of the path & energy & spectral value) to do a colorplot. For example, this is the kind of result that you obtain for PbTe with the 4x4x4 supercell



Exercise

Do the no-mode-mixing calculation along the path $X - \Gamma - X$ with smearing $10.0\,\mathrm{cm}^{-1}$

We conclde this tutorial stressing that a convergence analysis in terms of integration grid and smearing has to be done in order to obtain reliable results. The `get_os_perturb_dynamic_correction_along_path` routine is the best tool to do that. With this *input.py* input file you can do the calculation for several integration grids

```
import cellconstructor as CC
import cellconstructor.ForceTensor


dyn = CC.Phonons.Phonons("../minim/SSCHA.T300.dyn",3)
FC3 = CC.ForceTensor.Tensor3(dyn=dyn)
FC3.SetupFromFile(fname="../hessian/FC3",file_format='D3Q')


for kval in [4,8,16,32]:

   print("COMPUTING {}".format(kval))

   CC.Spectral.get_os_perturb_dynamic_correction_along_path(dyn,
                                    tensor3=FC3,
                                    k_grid=[kval,kval,kval],
                                    sm1=20.0, sm0=0.1,
```

(continues on next page)

(continued from previous page)

```
                                    nsm=80,
                                    q_path=[0.0,0.0,0.0],
                                    T=300.0,
                                    filename_shift_lw  = 'shift_hwhm_{}'.
↪format(kval),
                                    filename_freq_dyn = 'freq_{}'.format(kval))
```

giving

```
$ mpirun -np 4 python input.py > output
```

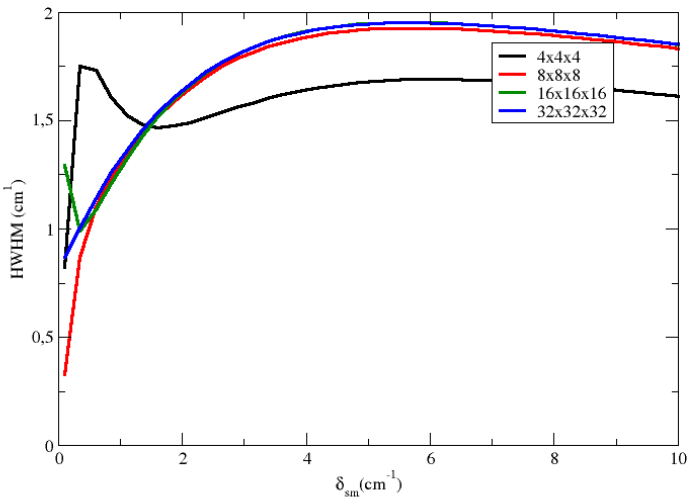
From *output* you can take the list of smearing values and write them in a file *sm.dat*. After that, you can collect the results with this *extract.sh* script

```
for grid in 4 8 16 32
do
 > ${grid}x${grid}x${grid}.dat
 while read sm
 do
  tail -1  shift_hwhm_${grid}_${sm}.os.dat \
  | awk -v sm="${sm}" '{printf"%11.7f\t%22.11f\n",sm,$(NF-1)}' \
  >> ${grid}x${grid}x${grid}.dat
 done < sm.dat
done
```

giving

```
$ bash extract.sh
```

so as to obtain the files *4x4x4.dat*, *8x8x8.dat*, *16x16x16.dat*, and *32x32x32.dat*. Plotting them you obtain



From the plot, you can see that you need at least a 8x8x8 grid to obtain a converged value, using smearing equal to 1.0 cm-1.

CHAPTER

SIX

HANDS-ON-SESSION 5 - RAMAN AND INFRARED SPECTRA WITH THE TIME-DEPENDENT SELF CONSISTENT HARMONIC APPROXIMATION

In the previous tutorial, you learned how to compute the spectral function by integrating the bubble in the Fourier space, with the dynamical ansatz formulated by Bianco et al, Physical Review B, 96 , 014111, 2017. Instead, we will employ the Lanczos algorithm within the Time-Dependent Self-Consistent Harmonic Approximation (TD-SCHA) Monacelli, Mauri, Physical Review B 103, 104305, 2021.

For this reason, we need the package `tdscha` (it is suggested to configure it with the Julia speedup to run faster, see the installation guide).

6.1 Computing the IR signal in ICE

We use an ensemble already computed of the phase XI of ice (low-temperature ice ad ambient pressure and prototype of standard cubic ice) to get the IR spectrum.

Inside the directory data, we find an already calculated ensemble of ice XI at 0K with the corresponding original dynamical matrix *start_dyn_ice1* employed to generate the ensemble and the dynamical matrix *final_dyn_ice1* after the SSCHA minimization.

6.1.1 An introduction

The infrared spectrum is related to the dipole-dipole response function:

$$\chi_{MM}(\omega) = \int_{-\infty}^{\infty} dt e^{-i\omega t} \langle M(t)M(0)\rangle$$

where the average $\langle M(t)M(0)\rangle$ is the quantum average at finite temperature.

Exploiting the TD-SCHA formalism introduced in the previous lecture, this response function can be written as:

$$\chi_{MM}(\omega) = \boldsymbol{r}(M)\boldsymbol{G}(\omega)\boldsymbol{q}(M) \tag{6.1.1}$$

where $\boldsymbol{G}(\omega)$ is the TD-SCHA green function, while the $\boldsymbol{r}$ and $\boldsymbol{q}$ are vectors that quantify the perturbation and response, respectively.

In particular, if we neglect two-phonon effects (nonlinear coupling with light), we get that

$$\chi_{MM}(\omega) = \sum_{ab} \frac{\mathcal{Z}_{\alpha a}\mathcal{Z}_{\alpha b}}{\sqrt{m_a m_b}} G_{ab}(\omega)$$

where $\mathcal{Z}_{\alpha a}$ is the Born effective charge of atom a, with polarization α, and $G_{ab}(\omega)$ is the one-phonon green function, (its imaginary part is precisely the spectral function).

Indeed, we need to compute the effective charges. This can be done directly by quantum espresso using linear response theory (ph.x).

Exercise

Use the knowledge of cellconstructor to extract a structure file from the final dynamical matrix to submit the calculation of the dielectric tensor, Effective charges, and Raman tensor in quantum espresso.

Hint. The structure is the attribute *structure* of the Phonons object. The structure in the SCF file can be saved with the *save_scf* method of the Structure object.

You can then attach the structure to the header of the espresso *ir_raman_header.pwi*.

Notice that we are using norm-conserving pseudo-potentials and LDA exchange-correlation functional, as the Raman Tensor in quantum espresso is implemented only with them. However, it is usually an excellent approximation. *ir_raman_header.pwi*.

You must run the pw.x code and the ph.x code (*ir_raman_complete.phi*), which performs the phonon calculation.

We provide the final output file in *ir_raman_complete.pho*

6.1.2 Prepare the infrared response

We need to attach the Raman Tensor and effective charges computed inside *ir_raman_complete.pho* to the final dynamical matrix, we will use this to initialize the response function calculation, as in Eq.6.1.1.

To attach the content of an espresso ph calculation (only Dielectric tensor, Raman Tensor, and Born effective charges) to a specific dynamical matrix, use

```
dyn.ReadInfoFromESPRESSO("ir_raman_complete.pho")
```

If you save the dynamical matrix in quantum espresso format, before the frequencies and the diagonalization, there will be the Dielectric tensor

```
Dielectric Tensor:

     1.890128098000           0.000000000000           0.000000000000
     0.000000000000           1.912811137000           0.000000000000
     0.000000000000           0.000000000000           1.916728724000
```

Followed by the effective charges and the Raman tensor.

6.1.3 Submitting the IR calculation

With the following script, we submit a TD-SCHA calculation for the IR.

```
import numpy as np
import cellconstructor as CC, cellconstructor.Phonons
import sscha, sscha.Ensemble
import tdscha, tdscha.DynamicalLanczos as DL

# Load the starting dynamical matrix
dyn_start = CC.Phonons.Phonons("start_dyn_ice")

# Load the ensemble
temperature = 0 # K
population = 2
n_configs = 10000

ensemble = sscha.Ensemble.Ensemble(dyn_start, temperature)
ensemble.load("data", population, n_configs)

# Load the final dynamical matrix
final_dyn = CC.Phonons.Phonons("final_dyn_ice")
final_dyn.ReadInfoFromESPRESSO("ir_raman_complete.pho")

# Update the ensemble weights for the final dynamical matrix
ensemble.update_weights(final_dyn, temperature)

# Setup the TD-SCHA calculation with the Lanczos algorithm
lanczos = DL.Lanczos(ensemble)
lanczos.ignore_v3 = True
lanczos.ignore_v4 = True

# If you have julia-enabled tdscha installed uncomment
# lanczos.mode = DL.MODE_FAST_JULIA
# for a x10-x15 speedup.

lanczos.init()


# Setup the IR response
polarization = np.array([1,0,0])  # Polarization of light
lanczos.prepare_ir(pol_vec = polarization)


# Run the algorithm
n_iterations = 1000
lanczos.run_FT(n_iterations)
lanczos.save_status("ir_xpol")
```

Congratulations! You ran your first TD-SCHA calculation. You can plot the results by using:

```
tdscha-plot.py ir_xpol.npz
```

The script `tdscha-plot.py` is automatically installed with the tdscha package.

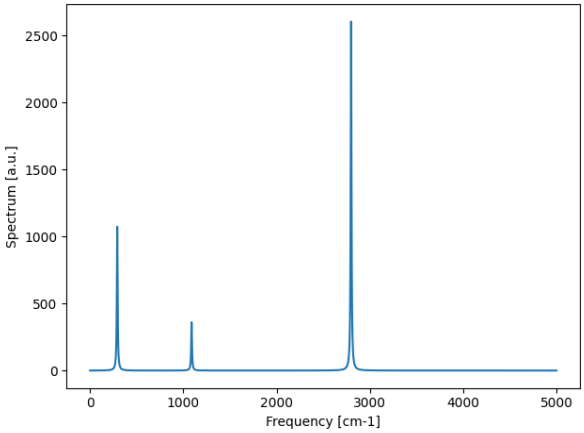


Fig. 6.1.1: IR spectrum with both *include_v3* and *include_v4* set to False.

Additionally, `tdscha-plot.py` takes three more parameters: the range of the frequencies to be displayed and the smearing.

6.1.4 Deep dive into the calculation

Let us dive a bit into the calculation. The beginning of the script should be almost self-explanatory, as we are just loading dynamical matrices, dielectric tensors, and effective charges.

The line

```
ensemble.update_weights(final_dyn, temperature)
```

deserves special attention. Here, we are changing the weights of the configurations inside the ensemble to simulate the specified dynamical matrix and temperature, even if they differ from those used to generate the ensemble. This is useful to compute the spectrum at several temperatures without extracting and calculating a new ensemble each time.

```
# Setup the TD-SCHA calculation with the Lanczos algorithm
lanczos = DL.Lanczos(ensemble)
lanczos.ignore_v3 = True
lanczos.ignore_v4 = True
lanczos.init()
```

Then we initialize the Lanczos algorithm for the tdscha, passing the ensemble.

The ignore_v3 and ignore_v4 are flags that, if set to True, the 3-phonon and 4-phonon scattering will be ignored during the calculation.

As you can see from the output, our IR signal had very sharp peaks because we ignored any phonon-phonon scattering process that may give rise to a finite lifetime.

By setting only ignore_v4 to True, we reproduce the behavior of the bubble approximation. Notably, while the four-phonon scattering is exceptionally computationally and memory demanding in free energy

hessian calculations, within the Lanczos algorithm, accounting for the four-phonon scattering is only a factor two more expensive than using just the third order, without requiring any additional memory.

```
# Setup the IR response
polarization = np.array([1,0,0])  # Polarization of light
lanczos.prepare_ir(pol_vec = polarization)
```

Here we tell the Lanczos which kind of calculation we want to do. In other words, we set the $\boldsymbol{r}$ and $\boldsymbol{q}$ vectors in Eq.6.1.1 for the Lanczos calculation. - prepare_ir - prepare_raman - prepare_mode - prepare_perturbation

The names are intuitive; besides the Raman and IR, prepare_mode allows you to study the response function of a specific phonon mode, and prepare_perturbation enables defining a custom perturbation function.

```
# Run the algorithm
n_iterations = 1000
lanczos.run_FT(n_iterations)
lanczos.save_status("ir_xpol")
```

Here we start the calculation of the response function. The number of iterations indicates how many Lanczos steps are required. Each step adds a new pole to the green function. Therefore, many steps are necessary to converge broad spectrum features, while much less if the peaks are sharp. We save the status in such a way that we can get it back later.

Last, the commented line

```
lanczos.mode = DL.MODE_FAST_JULIA
```

This line only works if Julia and PyCall are correctly set up in the PC; in that case, run the script with *python-jl* instead of python. It will exploit a massive speedup of a factor between 10x and 15x. The calculation can also be run in parallel using *mpirun* before calling the Python executable (or python-jl). In this case, to work correctly, you should have mpi4py installed and working.

Exercise

Compute the Lanczos with the bubble approximation and without any approximation, and check the differences.

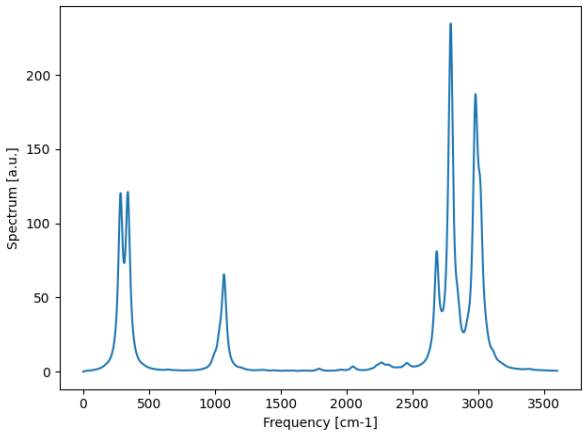


Fig. 6.1.2: IR signal accounting for the three-phonon scattering

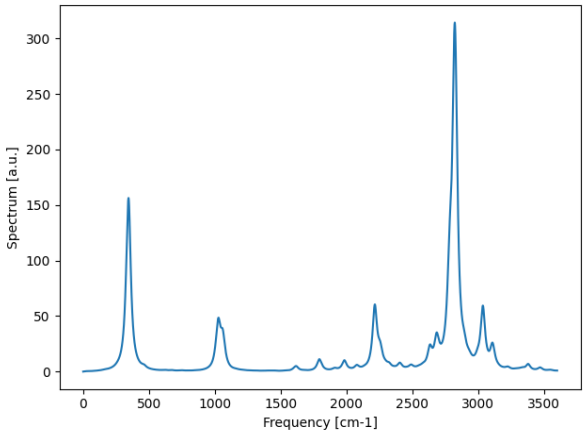


Fig. 6.1.3: IR signal accounting for all anharmonic scattering. The peaks that appear slightly below 2500 cm-1 is a combination mode known to be present in ice. See Cherubini et al, J Chem Phys 155, 184502, 2021

Exercise

Try to see how different polarization of the light affects the result.

6.1.5 Analyze the output

In the last part, we employed the script `tdscha-plot.py` to display the simulation result. This is a quick way to show the results of a calculation.

Here, we will dive deeper into the calculation output file to extract the response function and get the results.

The Lanczos algorithm provides a set of coefficients a_i, b_i, and c_i through which the green function is evaluated thanks to a continued fraction:

$$G(\omega) = \cfrac{1}{a_1 - (\omega + i\eta)^2 + \cfrac{b_1 c_1}{a_2 - (\omega+i\eta)^2 + \frac{c_2 b_2}{a_3 - \cdots}}}$$

Each iteration of the algorithm adds a new set of coefficients written in the standard output. Thanks to this expression, we only need the series of coefficients to compute the dynamical Green function at any frequency and with any smearing. The Green function can be computed with:

```
green_function = lanczos.get_green_function_continued_fraction(frequencies,␣
↪smearing=smearing)
```

Here `frequencies` is an array in Rydberg. The response function is the opposite of the imaginary part of the green function; thus, to reproduce the plot, we have:

```
import tdscha, tdscha.DynamicalLanczos
import cellconstructor as CC, cellconstructor.Units
import numpy as np
import matplotlib.pyplot as plt

# Load the result of the previous calculation
lanczos = tdscha.DynamicalLanczos.Lanczos()
lanczos.load_status("ir_xpol_v4")

# Get the green function
W_START = 0
W_END = 3700
N_W = 10000
SMEARING = 10

frequencies = np.linspace(W_START, W_END, N_W)

# Convert in RY
frequencies_ry = frequencies / CC.Units.RY_TO_CM
smearing_ry = SMEARING / CC.Units.RY_TO_CM

# Compute the green function
green_function = lanczos.get_green_function_continued_fraction(frequencies_ry,
        smearing=smearing_ry)

# Get the response function
ir_response_function = - np.imag(green_function)

# Plot the data
plt.plot(frequencies, ir_response_function)
plt.show()
```

The previous script plots the data, precisely like *plot-tdscha.py*; however, now you have full access to the response function, both its imaginary and real parts.

Exercise

Plot the IR data at various smearing and as a function of the number of steps (50, 100, 200, 300, and 1000). How does the signal change with smearing and the number of steps? When is it converged?

6.2 Raman response

The Raman response is very similar to the IR. Raman probes the fluctuations of the polarizability instead of those of the polarization, and it occurs when the samples interact with two light sources: the incoming electromagnetic radiation and the outcoming one. The outcoming radiation has a frequency that is shifted with respect to the incoming one by the energy of the scattering phonons. The signal on the red side of the pump is called Stokes, while the signal on the blue side is the Antistokes. Since the outcoming radiation has higher energy than the incoming one in the Antistokes, it is generated only by existing (thermally excited phonons) inside the sample. Therefore it has a lower intensity than the Stokes.

On the Stokes side, the intensity of the scattered light with a frequency redshift of ω is

$$I(\omega) \propto \langle \alpha_{xy}(\omega)\alpha_{xy}(0)\rangle \, (n(\omega)+1)$$

where α is the polarizability along the xy axis. We can do a linear expansion around the equilibrium position of the polarizability, and we get:

$$\alpha_{xy}(\omega) = \sum_{a=1}^{3N} \frac{\partial \alpha_{xy}}{\partial R_a(\omega)} (R_a(\omega) - \mathcal{R}_a)$$

$$\alpha_{xy}(\omega) = \sum_{a=1}^{3N} \Xi_{xya} (R_a(\omega) - \mathcal{R}_a)$$

If we insert it in the expression of the intensity, the average between the positions is the atomic green function divided by the square root of the masses, and we get

$$I(\omega) \propto \sum_{ab} \frac{\Xi_{xya}\Xi_{xyb}}{\sqrt{m_a m_b}} G_{ab}(\omega)(n(\omega)+1)$$

where $G_{ab}(\omega)$ is the atomic green function on atoms a and b, while Ξ_{xya} is the Raman tensor along the electric fields directed in x and y and on atom a.

The multiplication factor $n(\omega)+1$ comes from the observation of the Stokes nonresonant Raman (it would be just $n(\omega)$ for the antistokes).

As we did for the IR signal, we can prepare the calculation of the Raman raman scattering by computing the polarizability-polarizability.

```
# Setup the polarized Raman response
polarization_in = np.array([1,0,0])
polarization_out = np.array([1,0,0])
lanczos.prepare_raman(pol_vec_in=polarization_in,
        pol_vec_out=polarization_out)
```

Note that here we have to specify two polarization of the light, the incoming radiation, and the outcoming radiation.

Exercise

Compute and plot the Intensity of the Raman in the Stokes and antistokes configurations. Try with different polarization and even orthogonal polarization; what does it change?

The Bose-Einstein factor $n(\omega)$ can be computed with the following function:

```
# n(w) Bose-Einstein occupation number:
# w is in Ry, T is in K
n_w = tdscha.DynamicalLanczos.bose_occupation(w, T)
```

6.2.1 Unpolarized Raman and IR

In the previous section, we saw how to compute Raman and IR with specific polarization of the incoming and outcoming radiation, and on oriented crystals (single crystals). However, the most common situation is a powder sample probed with unpolarized light.

In this case, we need to look at the Raman and IR response for unpolarized samples. While this is just the average of the IR signal's x, y, and z, the Raman is more complex. In particular, unpolarized Raman signal can be computed from the so-called *invariants*, where the perturbations in the polarizations are the following:

$$
\begin{aligned}
I_A &= \frac{1}{3}(xx + yy + zz)^2/9 \\
I_{B_1} &= (xx - yy)^2/2 \\
I_{B_2} &= (xx - zz)^2/2 \\
I_{B_3} &= (yy - zz)^2/2 \\
I_{B_4} &= 3(xy)^2 \\
I_{B_5} &= 3(yz)^2 \\
I_{B_6} &= 3(xz)^2
\end{aligned}
$$

The total Intensity of unpolarized Raman is:

$$
I_{unpol}(\omega) = 45 \cdot I_a(\omega) + 7 \cdot \sum_{i=1}^{6} I_{B_i}(\omega)
$$

The tdscha code implements a way to compute each perturbation separately. For example, the Raman response related to I_A is calculated with

```
lanczos.prepare_raman(unpolarized=0)
```

While the I_{B_i} is computed using index i. For example, to compute I_{B_5}:

```
# To compute I_B5 we do
lanczos.prepare_raman(unpolarized=5)
```

To get the total spectrum, you need to add the scattering factor $n(\omega) + 1$ and sum all these perturbation with the correct prefactor (45 for I_A and 7 for the sum of all I_B).

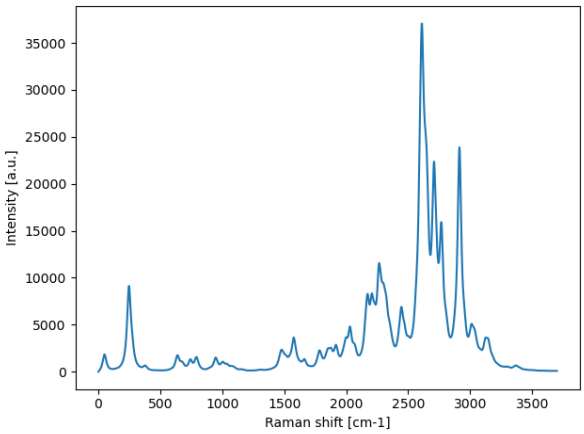
To reset a calculation and start a new one, you can use

```
lanczos.reset()
```

which may be called before preparing the perturbation.

Exercise

Compute the unpolarized Raman spectrum of ice and plot the results.



You should employ a supercell size sufficiently big to converge the simulation properly. In this case, the 1x1x1 supercell is too tiny to converge the calculation and get meaningful results.

CHAPTER

SEVEN

HANDS-ON-SESSION 6 - THE SSCHA WITH MACHINE LEARNING POTENTIALS

The minimization of the variational free energy demands a large number of single-point density functional theory (DFT) calculations. These calculations are performed on supercells, repetitions of the primitive unit cell. DFT calculations can become very costly very fast if we need to increase the size of the supercell. This can happen in case we have very slowly decaying second-order force constants, large primitive unit cells, or simply very low symmetry. In some of these cases, DFT is prohibitive due to the large number of atoms per calculation or we simply need a very large number of configurations to converge our results (for example when we need to compute free energy hessian to check the dynamical stability of the system).

In the last ten years, there has been a large amount of research put into developing machine-learned (ML) interatomic potentials. Contrary to the traditional interatomic potentials, they do not have a fixed analytical form and thus are much more flexible and transferable. They are usually trained on a very large number of DFT data and have very good accuracy. ML potentials are considerably slower than the traditional interatomic potentials, however still orders of magnitude faster than DFT, with a considerably better scaling with a number of atoms.

The synergy between SSCHA and machine learning interatomic potentials is obvious. If we can use the machine learning interatomic potentials as a calculator for forces, stresses, and energies we can go to much larger supercells and numbers of configurations. The stochastic sampling employed by SSCHA gives a very good method for obtaining training sets needed to train machine learning interatomic potentials. The force, energy and stresses errors produced by ML interatomic potentials will influence SSCHA results less due to the averaging effects (in case the errors are not biased).

There are a number of freely available implementations of ML interatomic potentials (Gaussian Approximation Potentials, NequIP, pacemaker, etc.), and at this point, they can be used without a large prior knowledge of the theory behind ML potentials.

7.1 Hands-on exercise

For this exercise we will be using Gaussian Approximation Potentials, however, the framework can be applied to any other type of ML interatomic potential. In the exercise, we will obtain the training data from the Tersoff interatomic potential, instead of the DFT.

We have provided starting dynamical matrices calculated for the structure at 0 K. Now we will calculate training and test ensemble with Tersoff potential:

```
from quippy.potential import Potential
import cellconstructor as CC
```

(continues on next page)

(continued from previous page)

```
import cellconstructor.Phonons
import sscha, sscha.Ensemble
from cellconstructor.Units import RY_PER_BOHR3_TO_EV_PER_A3

temperature = 0.0   # Temperature at which we generate SSCHA configurations
nconf_train = 1000  # Number of configurations in the training set
nconf_test = 500    # Number of configurations in the test set

# Load the Tersoff potential that we want to fit with ML GAP
pot = Potential('IP Tersoff',
        param_filename='./06_the_SSCHA_with_machine_learning_potentials/ip.
↪parms.Tersoff.xml')
# Load dynamical matrices
dyn_prefix = './06_the_SSCHA_with_machine_learning_potentials/start_dyn'
nqirr = 3
dyn = CC.Phonons.Phonons(dyn_prefix, nqirr)

# Generate training ensemble
ensemble_train = sscha.Ensemble.Ensemble(dyn, T0=temperature,
        supercell = dyn.GetSupercell())
ensemble_train.generate(N = nconf_train)
ensemble_train.compute_ensemble(pot, compute_stress = True,
        stress_numerical = False, cluster = None, verbose = True)

# This line will save ensemble in correct format
ensemble_train.save_enhanced_xyz('train.xyz', append_mode = False,
        stress_key = "stress", forces_key = "forces",
        energy_key = "energy")

# Generate test ensemble
ensemble_test = sscha.Ensemble.Ensemble(dyn, T0=temperature,
        supercell = dyn.GetSupercell())
ensemble_test.generate(N = nconf_test)
ensemble_test.compute_ensemble(pot, compute_stress = True,
        stress_numerical = False, cluster = None, verbose = True)

ensemble_test.save_enhanced_xyz('test.xyz', append_mode = False,
        stress_key = "stress", forces_key = "forces",
        energy_key = "energy")
```

The training of the ML interatomic potential can be done with a command gap_fit which should be available after installing quippy-ase. This command takes a large number of arguments so it is easier to make a bash script. We will name it train.sh:

```
#!/bin/bash

gap_fit energy_parameter_name=energy force_parameter_name=forces \
    stress_parameter_name=stress virial_parameter_name=virial \
    do_copy_at_file=F sparse_separate_file=T gp_file=GAP.xml \
```

(continues on next page)

(continued from previous page)

```
    at_file=train.xyz e0_method="average" \
    default_sigma={0.001 0.03 0.03 0} sparse_jitter=1.0e-8 \
    gap={soap cutoff=4.2 n_sparse=200 covariance_type=dot_product \
        sparse_method=cur_points delta=0.205 zeta=4 l_max=4 \
        n_max=8 atom_sigma=0.5 cutoff_transition_width=0.8 \
        add_species }
```

The meaning of each argument is not important right now, but can be easily looked up on the official website https://libatoms.github.io/GAP/gap_fit.html. We run the training command:

```
bash train.sh
```

Note, the training is memory intensive, so you may need to allocate extra memory on your virtual machine if you are employing Quantum Mobile. 4Gb of Ram are required. You may need to restart the virtual machine.

This should take a minute or so. Once it is finished, if the memory was enough and the command typed correctly, one should obtain the `GAP.xml` file in the working directory containing the GAP ML interatomic potential. We can use test.xyz file to check how well our ML potential reproduces data with this simple script:

```
import numpy as np
import ase
from ase import Atoms
from quippy.potential import Potential
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
fpaths = matplotlib.font_manager.findSystemFonts()

infile = 'test.xyz' # test datasets

# Read in .xyz files using ase method
atoms = ase.io.read(infile, ':', format='extxyz')
nconf = len(atoms)
print('Number of configurations in the dataset: ' + str(nconf))
natoms = [len(at.symbols) for at in atoms]

# Load in newly trained GAP potential
gap_file = './GAP.xml'
pot = Potential('IP GAP', param_filename=gap_file) # Read in potential

# Collect previously calculated (with Tersoff) atomic properties
dft_energies = [atom.get_potential_energy() for atom in atoms]
dft_forces = [atom.get_forces() for atom in atoms]
dft_stress = [atom.get_stress()[0:3] for atom in atoms]

# Now recalculate them with GAP
en_gap = []
forces_gap = []
```

(continues on next page)

(continued from previous page)

```
stress_gap = []
for i in range(nconf):
    if(i%100 == 0):
        print('Configuration: ', i + 1)
    # Make ase Atoms object
    atoms_gap = Atoms(symbols = atoms[i].symbols, cell = atoms[i].cell,\
    scaled_positions = atoms[i].get_scaled_positions(),\
    calculator = pot, pbc = True)
    # Calculate total energies of structures with GAP
    en_gap.append(atoms_gap.get_potential_energy())
    # Calculate forces on atoms
    forces_gap.append(atoms_gap.get_forces())
    # Calculate stress and only take diagonal elements
    stress_gap.append(atoms_gap.get_stress()[0:3])

# Calculate errors
energy_errors = np.zeros_like(en_gap)
forces_errors = np.zeros_like(forces_gap)
GPa = 1.60217733e-19*1.0e21
stress_errors = np.zeros_like(stress_gap)
for i in range(nconf):
    # Calculate energy errors
    energy_errors[i] = (atoms[i].get_potential_energy() -\
    en_gap[i])/natoms[i]
    # Calculate errors on forces
    forces_errors[i] = atoms[i].get_forces() - forces_gap[i]
    # Calculate errors on stress
    stress_errors[i] = dft_stress[i] - stress_gap[i]

# Function to plot Tersoff vs GAP results
def plot_comparison(ax, data1, data2, data3, \
xlabel = 'Original energy (eV)', ylabel = 'ML energy (eV)'):
    sizes = np.array(data3/np.amax(data3))*2.0
    ax.scatter(data1, data2, marker = 'o', s = sizes,  c = 'red')
    lims = [np.min([ax.get_xlim(), ax.get_ylim()]),\
    np.max([ax.get_xlim(), ax.get_ylim()])]
    # now plot both limits against eachother
    ax.plot(lims, lims, 'k-', alpha=0.75, zorder=0)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)

# Function to plot histogram of errors
def plot_error_histogram(ax, x, nbins, xlabel):
    import scipy.stats as st
    from scipy.stats import norm

    ax.hist(x, density=True, bins=nbins)
    mu, std = norm.fit(x)
    xmin, xmax = ax.get_xlim()
```

(continues on next page)

(continued from previous page)

```
    x1 = np.linspace(xmin, xmax, 100)
    p = norm.pdf(x1, mu, std)
    ax.plot(x1, p, 'k', linewidth=2)
    ax.set_ylabel("Probability")
    ax.set_xlabel(xlabel)

# Sometimes forces arrays can be ragged list, this will flatten them
def flatten(xs):
    res = []
    def loop(ys):
        for i in ys:
            if isinstance(i, list):
                loop(i)
            elif(isinstance(i, np.ndarray)):
                loop(i.tolist())
            else:
                res.append(i)
    loop(xs)
    return res


# Plot stuff
plt.rcParams["font.family"] = "Times New Roman"
plt.rcParams['mathtext.fontset'] = "stix"
plt.rcParams.update({'font.size': 16})

fig = plt.figure(figsize=(6.4*3.0, 4.8*2.0))
gs1 = GridSpec(2, 3)

ax00 = fig.add_subplot(gs1[0, 0])
plot_comparison(ax00, dft_energies, en_gap, energy_errors, \
xlabel = 'Original energy (eV)', ylabel = 'ML energy (eV)')
ax01 = fig.add_subplot(gs1[0, 1])
plot_comparison(ax01, dft_forces, forces_gap, forces_errors, \
xlabel = r'Original force (eV/$\AA$)', ylabel = r'ML force (eV/$\AA$)')
ax02 = fig.add_subplot(gs1[0, 2])
plot_comparison(ax02, np.array(flatten(dft_stress))*GPa, \
np.array(flatten(stress_gap))*GPa, np.array(flatten(stress_errors))*GPa, \
xlabel = 'Original stress (GPa)', ylabel = 'ML stress (GPa)')

ax10 = fig.add_subplot(gs1[1, 0])
plot_error_histogram(ax10, energy_errors, 100, 'Energy error (eV/atom)')
ax11 = fig.add_subplot(gs1[1, 1])
flattened_forces = flatten(forces_errors)
plot_error_histogram(ax11, flattened_forces, 100, 'Force error (eV/$\AA$)')
ax12 = fig.add_subplot(gs1[1, 2])
plot_error_histogram(ax12, np.array([item for sublist in stress_errors for␣
↪item in sublist])*GPa,\
100, 'Stress error (GPa)')
```

(continues on next page)

(continued from previous page)

```
fig.savefig('test.pdf')
plt.show()
```

In the upper panel figures, ideally, we would like points to lie on the diagonal. When fitting interatomic potential we aim for normal distribution of errors centered at 0 (without bias) with as small as possible standard deviation. We should have very nice results for energies and forces. However, our results are very bad for stresses, but this is probably a problem due to the use of Tersoff potential. With DFT the stresses would usually follow normal distribution as well. Now that we are happy with the potential let us use it to relax SSCHA at 2000 K:

```
from quippy.potential import Potential
import cellconstructor as CC
import cellconstructor.Phonons

# Import the SSCHA engine (we will use it later)
import sscha, sscha.Ensemble, sscha.SchaMinimizer, sscha.Relax

# Declare SSCHA variables
temperature = 2000.0
nconf = 1000
max_pop = 10000

# Load in the GAP potential
gap_file = './GAP.xml'
pot = Potential('IP GAP', param_filename=gap_file)

# Load in the SSCHA dynamical matrices
dyn_prefix = './06_the_SSCHA_with_machine_learning_potentials/start_dyn'
nqirr = 3
dyn = CC.Phonons.Phonons(dyn_prefix, nqirr)

# Relax the structure at 2000 K
ensemble = sscha.Ensemble.Ensemble(dyn, T0=temperature,
supercell = dyn.GetSupercell())
ensemble.generate(N = nconf)
minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
minimizer.min_step_dyn = 0.1
minimizer.kong_liu_ratio = 0.5
minimizer.meaningful_factor = 0.001
minimizer.max_ka = 100000
relax = sscha.Relax.SSCHA(minimizer, ase_calculator = pot,
N_configs = nconf, max_pop = max_pop, save_ensemble = True)
relax.vc_relax(ensemble_loc='Ensemble_location')
relax.minim.dyn.save_qe('final_dyn')
# We can check minimization procedure
relax.minim.plot_results(save_filename = 'sscha', plot = False)
```

We have relaxed SSCHA at 2000 K. We can check that everything went well in “sscha” file. However, we do not know whether this is correct. We need to check how our ML potential performs at 2000 K.

Exercise:

Let's create a dataset of SSCHA-generated configuration at 2000 K using GAP relaxed dynamical matrices and compute it using Tersoff potential. Next, check the performance of the GAP ML potential against this new dataset.

Excercise:

We should see GAP performing quite worse compared to the test.xyz case. How can we improve GAP potential? Let's do it.

Excercise:

How do the Tersoff phonons compare to GAP phonons?

Excercise:

Does this translate to larger supercells?

CHAPTER

EIGHT

HANDS-ON-SESSION 7: CALCULATION OF THE ELECTRON-PHONON INTERACTION AND SUPERCONDUCTING PROPERTIES WITH THE SSCHA

In this hands-on-session we will learn to calculate the electron-phonon interaction and superconductivity properties in strongly anharmonic systems combining electron-phonon matrix elements calculated within density-functional perturbation theory (DFPT), as implemented in Quantum Espresso (QE), and the anharmonic phonon frequencies and polarization vectors obtained with the SSCHA. For that purpose we will use a slightly modified version of Quantum Espresso version 5.1.0, which includes some extra features developed by us that can be used to combine the electron-phonon matrix elements with real space SSCHA force constants, calculate the $\alpha^2F(\omega)$, calculate T_c with empirical equations, and solve isotropic Migdal-Eliashberg equations.

8.1 Calculation of the electron-phonon matrix elements

As an example we will use a non-converged calculation on PdH, a strongly anharmonic superconductor, in which the Pd atoms form a fcc lattice and H atoms sit at the octahedral interstitial sites. The crystal structure is the rock-salt one, whose space group is $Fm\bar{3}m$.

We will perform first a harmonic phonon calculation and the calculation of the electron-phonon coupling constant for the irreducible q points in a 2x2x2 grid. In order to know how many irreducible q points are there for a given crystal one can use the kpoints.x program of QE (for instance, in the QE version distributed with the SSCHA, one can find it in qe-5.1.0_elph/PW/tools). In this case there are only three q points in the irreducible grid. For a particular q point, the calculation of the electron-phonon matrix elements is performed in three steps:

1. Perform a standard DFT calculation of the crystal structure
2. Perform a standard DFPT to calculate the harmonic dynamical matrix at a particular point in the grid as well as the derivative of the Kohn-Sham (KS) potential for this particular q point. The latter will be needed for the electron-phonon calculation.
3. Perform a non-self-consistent calculation of the band structure in a finer electronic k point grid, read the derivative of the KS potential, read the harmonic dynamical matrix to obtain the phonon frequencies and polarization vectors, and calculate the electron-phonon matrix elements.

The input we will use for the standard DFT calculation that we will use is the following:

```
&control
    ! Type of calculation
    calculation     =       'scf'
```

(continues on next page)

(continued from previous page)

```
    ! Show more details in the output
    verbosity       =       'high'
    ! Calculate stress tensor
    tstress         =       .true.
    ! Calculate forces
    tprnfor         =       .true.
    ! Prefix for tmp files
    prefix          =       'pdh'
    ! Location of the pseudopotentials
    pseudo_dir      =       './'
    ! Folder for the tmp files
    outdir          =       './tmp'
/
&system
    ! Type of unit cell
    ibrav           =       2
    ! Lattice parameter in a0, Bohr length
    celldm(1)       =       7.80
    ! Number of atoms
    nat             =       2
    ! Number of atom types
    ntyp            =       2
    ! Plane-wave cutoff in Ry
    ecutwfc         =       30.0
    ! Density cutoff in Ry
    ecutrho         =       300.0
    ! It is a metal so use smearing
    occupations     =       'smearing'
    ! Type of smearing
    smearing        =       'mp'
    ! Broadening of the smearing in Ry
    degauss         =       0.020
/
&electrons
    ! Parameter for the DFT scf cycle
    mixing_beta     =       0.7
    ! Energy threshold to stop the scf cycle
    conv_thr        =       1.0d-8
/
ATOMIC_SPECIES
    ! Pseudopotential for Pd
    Pd  106.42   Pd.pz-nd-rrkjus.UPF
    ! Pseudopotential for H
    H   1.00794  H.pz-rrkjus.UPF
ATOMIC_POSITIONS {crystal}
    Pd       0.0000        0.0000        0.0000
    H        0.5000        0.5000        0.5000
K_POINTS {automatic}
    10 10 10 1 1 1
```

The reader is referred to the official QE guide to check the all the details about the input files used, even if a short description is provided here. The pseudopotentials can be found in the folder 07_simple_electron_phonon/pseudos/ .

If this input file was named as pw.in, we would run QE as follows:

```
qe-5.1.0_elph/bin/pw.x < pw.in > pw.out
```

This will calculate the Kohn-Sham potential, needed for the phonon and electron-phonon calculations, apart from the usual total energy and forces of the structure. Note that this is not a converged calculation. One should carefully check the convergence with respect to the cutoffs, smearing, k-point grids, etc.

The second step is to calculate the harmonic dynamical matrix within DFPT. This is a model input file to calculate it in the first (the Γ point) q point of the 2x2x2 grid.

```
Phonon calculation on the 1st point of a 2x2x2 q point grid
&inputph
    ! Prefix for tmp files (same as for pw.x)
    prefix           = 'pdh'
    ! Folder for tmp files (same as for pw.x)
    outdir           = './tmp/'
    ! Name of dynamical matrices calculated
    fildyn           = "harmonic_dyn"
    ! Mass of 1st atom type in m.a.u
    amass(1)         = 106.42
    ! Mass of 2nd atom type in m.a.u
    amass(2)         = 1.00794
    ! File where the derivative of the KS potential will be stored
    fildvscf         = 'pdh_dv'
    ! Calculate the phonons in a grid nq1 x nq2 x nq3 grid of q points
    ldisp            = .true.
    nq1              = 2
    nq2              = 2
    nq3              = 2
    ! Threshold for the self-consistent loop
    tr2_ph           = 1.0d-16
    ! First q point to calculate
    start_q          = 1
    ! Last q point to calculate
    last_q           = 1
&end
```

The reader is referred to the official QE guide for more details on the input parameters. If this input file was named as ph.in, we would run QE as follows:

```
qe-5.1.0_elph/bin/ph.x < ph.in > ph.out
```

As an output we will obtain the dynamical matrix at the Γ point stored in the file harmonic_dyn1.

Once we have the dynamical matrix calculated and the derivative of the KS potential stored we can calculate the electron-phonon matrix elements using the modified version of QE. The input file is the following:

```
Phonon calculation on the 1st point of a 2x2x2 q point grid
&inputph
    ! Prefix for tmp files (same as for pw.x)
    prefix           = 'pdh'
    ! Folder for tmp files (same as for pw.x)
    outdir           = './tmp/'
    ! Name of dynamical matrices calculated
    fildyn           = "harmonic_dyn"
    ! Mass of 1st atom type in m.a.u
    amass(1)         = 106.42
    ! Mass of 2nd atom type in m.a.u
    amass(2)         = 1.00794
    ! File where the derivative of the KS potential will be stored
    fildvscf         = 'pdh_dv'
    ! Calculate the phonons in a grid nq1 x nq2 x nq3 grid of q points
    ldisp            = .true.
    nq1              = 2
    nq2              = 2
    nq3              = 2
    ! Threshold for the self-consistent loop
    tr2_ph           = 1.0d-16
    ! First q point to calculate
    start_q          = 1
    ! Last q point to calculate
    last_q           = 1
    ! Do not calculate dynamical matrix
    trans            = .false.
    ! Type of electron-phonon interaction
    electron_phonon  = 'simple'
    ! Minimum Gaussian broadening in Ry for the double Dirac delta
    el_ph_sigma      = 0.004
    ! The number of Gaussian broadenings that will be studied
    el_ph_nsigma     = 25
    ! nk1 x nk2 x nk3 is the grid for the non-scf calculation
    ! used in the electron-phonon calculations
    nk1              = 20
    nk2              = 20
    nk3              = 20
    ! k1, k2, k3 determine whether the grid is shifted from Gamma
    k1               = 1
    k2               = 1
    k3               = 1
&end
```

The reader is referred to the official QE guide for more details on the input parameters. If this input file was named as elph.in, we would run QE as follows:

```
qe-5.1.0_elph/bin/ph.x < elph.in > elph.out
```

As an output we will obtain several files giving information on the phonon linewidth coming from the electron-phonon interaction and so on. Most of them can be obtained with the standard version of QE,

but the modified version we are providing here prints also the 'fildyn'.elph.d.mat.'q point number' files, in this case harmonic_dyn1.elph.d.mat.1. This file is important for our purpose as it is necessary to combine the SSCHA dynamical matrices with the obtained electron-phonon matrix elements. What it contains explicitly is the following:

$$\Delta^{ab}(\mathbf{q}) = \frac{1}{N_{\mathrm{F}} N_{\mathbf{k}}} \sum_{n,n',\mathbf{k}} d^a_{n\mathbf{k},n'\mathbf{k}+\mathbf{q}} d^b_{n'\mathbf{k}+\mathbf{q},n\mathbf{k}} \delta(\epsilon_{n\mathbf{k}} - \epsilon_{\mathrm{F}}) \delta(\epsilon_{n'\mathbf{k}+\mathbf{q}} - \epsilon_{\mathrm{F}}),$$

where

$$d^a_{n\mathbf{k},n'\mathbf{k}+\mathbf{q}} = \langle n\mathbf{k} | \frac{\delta V_{KS}}{\delta u^a(\mathbf{q})} | n'\mathbf{k}+\mathbf{q} \rangle$$

are the electron-phonon matrix elements between differen KS states $|n\mathbf{k}\rangle$ (n is aband index and $\mathbf{k}$ the wave number) of the derivative of the KS potential with respect to the Fourier transformed displacement in Cartesian basis. In the above equations lower case latin indexes $(a\dots)$ denote both atoms in the unit cell as well as Cartesian indexes. Above N_F is the density of states (DOS) at the Fermi level per spin, ϵ_{F} is the Fermi energy, $N_{\mathbf{k}}$ the number of k points in the sum. The file contains first the Gaussian broadening used in the calculation (DOS and double Dirac delta in the equation), the DOS at the Fermi level calculated with that broadening, and later prints the elements of the $\Delta^{ab}(\mathbf{q})$ matrix. Then it continues with the same data for the next broadening calculated.

Exercise

- Calculate the electron-phonon matrix elements for the other q points in the 2x2x2 grid.

8.2 The SSCHA calculation

This system, even with this unconverged parameters, is extremely anharmonic and the SSCHA strongly renormalizes the phonon spectrum.

Exercise

- Perform a SSCHA calculation on a 2x2x2 supercell to obtain renormalized phonon frequencies.

As performing the SSCHA even with this unconverged parameters may take a considerable time, we provide auxiliary SSCHA dynamical matrices obtained with few configurations in 07_simple_electron_phonon/sscha/ with the name sscha_T0.0_dyn*.

8.3 Combine the SSCHA dynamical matrices with the electron-phonon matrix elements

We will now combine the SSCHA dynamical matrices with the electron-phonon matrix elements calculates previously. In order to do that (the reason will be apparent later) we will first Fourier transform the SSCHA dynamical matrices and create the real space SSCHA force constants. We can do that with the q2r.x code of QE. Let's first copy the harmonic_dyn0 obtained in the phonon calculations, file that contains the list of q points in the 2x2x2 grid, to the folder where the SSCHA dynamical matrices are and let's name it following the new notation:

```
cp {PATH_TO_HARMONIC_CALCULATION}/harmonic_dyn0 {PATH_TO_SSCHA_RESULTS}/sscha_
↪T0.0_dyn0
```

Now we are ready to perform the Fourier transform. Let's prepare the input for q2r.x:

```
&input
  ! Name of dynamical matrices
  fildyn  =  'sscha_T0.0_dyn'
  ! Type of ASR imposed
  zasr    =  'crystal'
  ! Name of obtained force constants
  flfrc   =  'sscha_T0.0.fc'
/
```

More details about the input for q2r.x can be found here. If the input file was named as q2r.in, we would run it as

```
qe-5.1.0_elph/bin/q2r.x < q2r.in > q2r.out
```

This will create the sscha_T0.0.fc file with the real space force constants.

Now we are ready to combine these SSCHA real space force constants with the electron-phonon matrix elements calculated with DFPT. For that the easiest thing to do is to copy the SSCHA force constants file (sscha_T0.0.fc) and the files where the $\Delta^{ab}(\mathbf{q})$ matrices were stored (harmonic_dyn*.elph.d.mat.*) to a new folder. We will use the elph_fc.x, which is written by us based on matdyn.x of QE and is not present in the QE distribution, to perform this calculation. The input for this code looks as follows:

```
Calculation of superconducting properties with elph_fc.x
&input
    ! ASR type imposed
    asr          =   'crystal'
    ! Mass of 1st atom in m.a.u
    amass(1)     =    106.42
    ! Mass of 2nd atom in m.a.u
    amass(2)     =    1.00794
    ! File with the SSCHA FCs
    flfrc        =   'sscha_T0.0.fc'
    ! Prefix for the files with the elph
    fildyn       =   'harmonic_dyn'
    ! Number of broadenings for the phonon Dirac Delta
    nbroad       =   1
    ! Gaussian broadening for the Dirac delta in cm-1
    minbroad     =   10
/
.004 ! Broadening for the electrons chosen
3    ! Number of q points in IBZ followed by the q list (in 2pi/a) and␣
↪multiplicity
0.000000000000000E+00   0.000000000000000E+00   0.000000000000000E+00 1
0.500000000000000E+00  -0.500000000000000E+00   0.500000000000000E+00 4
0.000000000000000E+00  -0.100000000000000E+01   0.000000000000000E+00 3
```

If the input file was named as elph_fc.in, we would run the code as follows:

```
qe-5.1.0_elph/bin/elph_fc.x < elph_fc.in > elph_fc.out
```

The code Fourier transforms the SSCHA real space force constants to the list of q points provided in order to obtain the dynamica matrices at these points, and combines them with the $\Delta^{ab}(\mathbf{q})$ matrices to calculate the Eliashberg function $\alpha^2F(\omega)$ as

$$\alpha^2F(\omega) = \frac{1}{N_\mathbf{q}} \sum_{ab\mathbf{q}\mu} \frac{e^a_\mu(\mathbf{q})\Delta^{ab}(\mathbf{q})e^b_\mu(\mathbf{q})^*}{2\omega_\mu(\mathbf{q})\sqrt{m_a m_b}} \delta(\omega - \omega_\mu(\mathbf{q})),$$

where $\omega_\mu(\mathbf{q})$ and $e^a_\mu(\mathbf{q})$ are, respectively, the phonon frequencies and polarization vectors obtained diagonalizing the Fourier interpolated SSCHA force constants at point q, and m_a the masses of the atoms. The Dirac delta on the equation is approximated with a Gaussian of 10 cm-1 broadening, following the input parameter. In the output elph_fc.out the code gives the SSCHA dynamical matrix at each q point, the phonon linewidth (HWHM), the contribution to the electron-phonon coupling of each mode, etc. Note that the code skips the Gamma point and does not consider it in the calculation. The reason is that the equation used at this point is divergent (see discussion in Appendix C in arXiv:2303.02621). The code also calculates the total electron-phonon coupling constant λ and ω_{log}. It also gives the value of the the superconducting critical temperature calculated for different values of the Coulomb pseudopotential μ^* within the semiempirical McMillan and Allen-Dynes formulas.

The code also prints the calculated Eliashberg function, phonon density of states, partial electron-phonon coupling constant, as well as the projection of both the phonon DOS and Eliashberg function on different atoms. We can do for example plots like this one with this data:

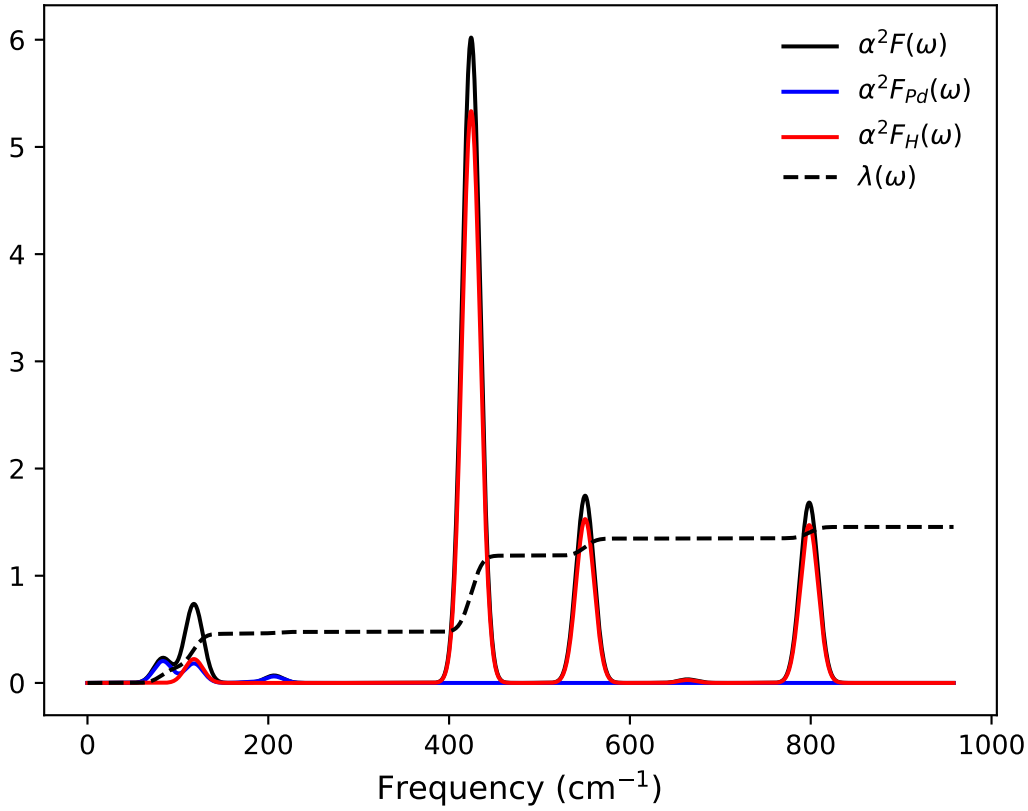


Fig. 8.3.1: Figure with the $\alpha^2F(\omega)$ Eliashberg function, its partial contributions from Pd and H atoms. The partial contribution to the electron-phonon coupling constant is also plotted, $\lambda(\omega)$.

Exercise

- Make a figure as the one above but with a different smearing for the double Dirac delta on the electronic states. Note that the electron-phonon matrix elements were calculated for smearings proportional to 0.04 Ry.

The fact that the elph_fc.x code works with real space SSCHA force constants allows us to combine the calculation of the electron-phonon matrix elements in a small supercell with electron-phonon matrix elements in a finer q point grid trivially.

Exercise

- Calculate the electron-phonon coupling constant using the electron-phonon matrix elements calculated in a 4x4x4 q point grid combining it with the SSCHA force constants on a 2x2x2 supercell. For that use the data in 07_simple_electron_phonon/elph_matrix_elements_444 .

8.4 Solution of isotropic Migdal-Eliashberg equations

Once the Eliashberg function $\alpha^2F(\omega)$ has been calculated combining the SSCHA and the electron-phonon matrix elements, one can easily solve isotropic Migdal-Eliashberg equations, which are not semiempirical. We provide a utility to perform this calculation as well, ME.x. This is the input file that needs to be prepared:

```
&inputme
     ! First temperature for the calculation
     t_first      =  1.0
     ! Last temperature for the calculation
     t_last       =  80.00
     ! Total number of temperatures
     t_number     =  80
     ! Cutoff for Matsubara frequencies in Ha
     wc_cutoff    =  0.05
     ! Initial guess for the gap in meV
     gap_guess    =  10.
     ! File with the Eliashberg function
     a2f_filename =  "a2F.10.dat"
     ! mu*
     mu_star      =  0.10
/
```

Note that the cutoff for the Matsubara frequencies should be around 10 times the highest phonon frequency, not bigger. If this input file was named as ME.in, we would run the code as follows:

```
qe-5.1.0_elph/bin/ME.x < ME.in > ME.out
```

In the output the code will calculate the superconducting gap as a function of temperature. In the provided file t_gap.dat the gap as a function of temperature is provided. This can be used to generate plots like this one to estimate the critical temperature from the temperature at which the gap closes.

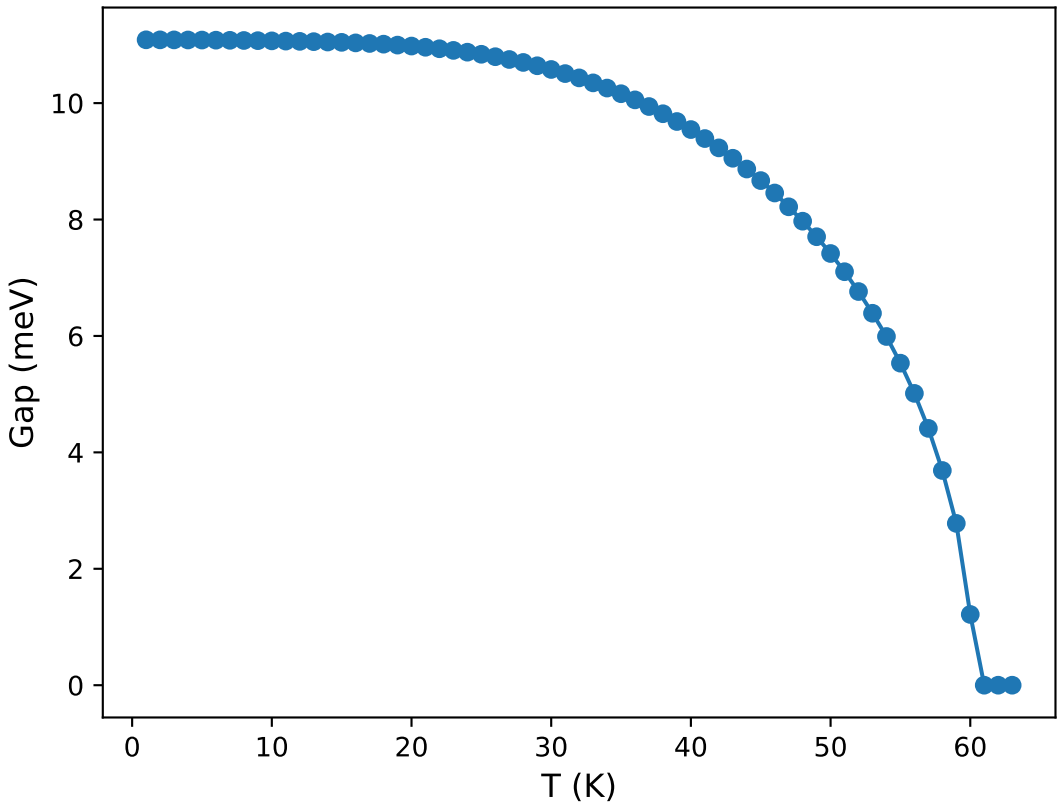


Fig. 8.4.1: Superconducting gap as a function of temperature.

8.5 Important remarks

The calculations above are not converged and are just meant to illustrate the use of the several codes. In a proper calculation one should take into account the following points:

1. The electron-phonon coupling constant needs to be converged with the number of k points for the electrons and the smearing used for the Double delta. These parameters enter into the equation of the $\Delta^{ab}(\mathbf{q})$ matrices. The typical thing is to calculate λ for different k point grids as a function of the smearing and see for which low value of the smearing λ plateaus. Note that the physical limit is the one with infinite number of k points and 0 smearing.
2. In this example we have used auxiliary SSCHA dynamical matrices to incorporate anharmonic effects into the calculation of electron-phonon properties. However, we could have also used those dynamical matrices that come from the Hessian of the SSCHA free energy. Sometimes the differences are minor, but in other cases it may be important. In this cases the best approach is to calculate the Eliashberg function with the full spectral function as described recently in arXiv:2303.07962.

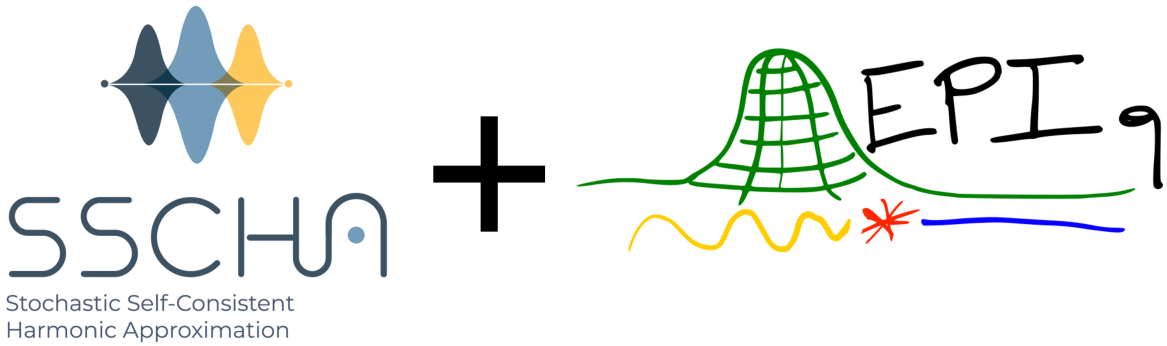
CHAPTER

NINE

HANDS-ON-SESSION 8: EPIQ - ANHARMONICITY IN ELECTRON-PHONON COUPLING RELATED PROPERTIES

9.1 Introduction

In this hands-on session we learn how to include anharmonic effects calculated within the SSCHA in the calculation of electron-phonon coupling related properties using EPIq.



In some systems the first principles calculation of electron-phonon coupling matrix elements can be demanding. EPIq (Electron-Phonon wannier Interpolation over k and q-points) is an open-source software that allows to speed up the calculation of electron–phonon coupling related properties using the Wannier interpolation technique. Details on the interpolation scheme can be found here. Within EPIq, it is possible to include anharmonic corrections to the dynamical matrices as calculated within the SSCHA.

9.2 Requirements

In the interest of time, in this hands-on session the following starting data are at your disposal:

1. Electron-phonon matrix elements $g^{\nu}_{m,n}(\mathbf{k},\mathbf{q})$ computed from first principles.
2. Wannier interpolation files which encode the trasformation to the optimally smooth subspace, U_{mn} : $|\mathbf{R}n\rangle = \frac{1}{\sqrt{N_k^w}} \sum_{\mathbf{k}=\mathbf{1}}^{N_k^w} \sum_{m=1}^{N_\mathrm{w}} e^{-i\mathbf{k}\cdot\mathbf{R}} U_{mn}(\mathbf{k}) |\psi_{\mathbf{k}m}\rangle$
3. Anharmonic dynamical matrices $D^{SCHA}_{\mu,\nu}(\mathbf{k},\mathbf{q})$.
4. Harmonic dynamical matrices (as a reference) $D^{HARM}_{\mu,\nu}(\mathbf{k},\mathbf{q})$.

Attention: A folder prepared for you with these data for the present tutorial can be downloaded 08_EPIq folder in the shared cloud. Right click on `tutorial_data` on the navigation bar and download the whole folder. It contains:

1. The electron-phonon coupling matrix elements can be found in the `mat_elem` folder. Each file corresponds to a different $\mathbf{q}$-point in the first Brillouin zone.
2. The `Wannier` folder contains the files `(.eig, .chk)`.
3. The dynamical matrices are stored in the `dyn_mat` directory. `dynq*` files are harmonic ($D^{SCHA}_{\mu,\nu}(\mathbf{k},\mathbf{q})$) while `MoS2.Hessian.dyn*` are anharmonic dynamical matrices computed with the SSCHA code ($D^{SCHA}_{\mu,\nu}(\mathbf{k},\mathbf{q})$).

Place all the downloaded material where you intend to run the tutorial. A suggested structure is for example:

```
handson_8/
    |
    + ----- epiq/
    |   |
    |   + ----- bin/
    |   |
    |   + ----- src/
    |
    |
    + ----- MoS2/
        |
        + ----- MoS2.eig
        |
        + ----- MoS2.chk
        |
        + ----- mat_elem/
        |   |
        |   + ----- MoS2_elph.mat.1_q*
        |
        + ----- dyn_mat/
            |
            + ----- dynq*
            |
            + ----- MoS2.Hessian.dyn*
```

9.3 About the usage of EPIq

9.3.1 EPIq workflow

The core steps of any calculation employing EPIq are performed using the main executable `epiq.x` and consists in two main stages.

1. A preliminary step where electron-phonon coupling matrix elements and the Hamiltonian are Fourier-transformed to real space and written to file.

2. The electron-phonon coupling matrix elements and the Hamiltonian are transformed back to reciprocal space to compute the property of interest at arbitrary k- and q- values.

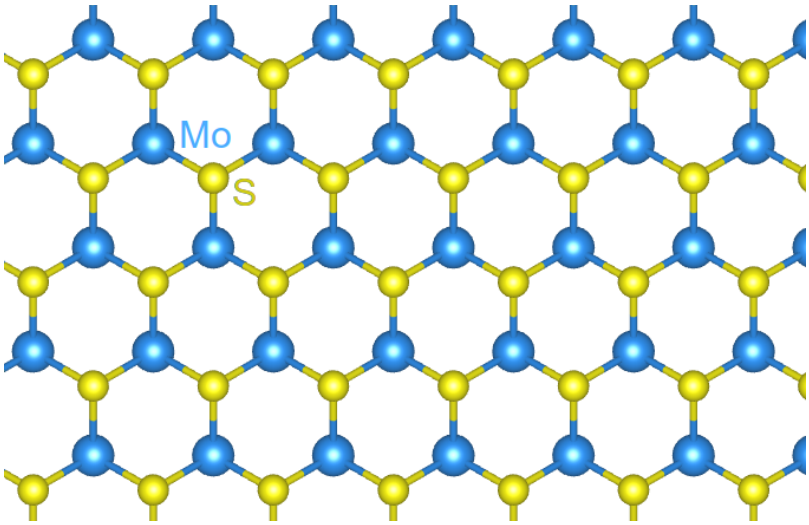
9.3.2 EPIq input file

The input file is divided in three namelists:

1. *&control*, specifying what calculation the code will perform
2. *&electrons*, specifying the electronic parameters of the property to be computed
3. *&phonons*, specifying the phonons parameters for the property to be computed

Finally, the last lines of the input indicates the electron momentum (k-) and phonon momentum (q-) meshes on which the matrix elements are interpolated to.

9.4 Let's practice: calculation of electron-phonon coupling related properties for doped monolayer MoS_2



In this tutorial we calculate electron-phonon coupling related properties for doped monolayer MoS_2, and evaluate the effect of anharmonicity on them.

9.4.1 Wannier interpolation

As explained in the previous section, any calculation within EPIq starts with a preliminary stepreferred to as `dump`. During this step, the Hamiltonian and the electron-phonon coupling matrix elements in real space are computed and written to file, to be used in any subsequent calculation. This step is performed only once. The input file is the following:

```
&control
    dump_gR=.true.,
    prefix='MoS2',
    elphmat_dir="./mat_elem/"
  /
```

(continues on next page)

(continued from previous page)

```
 &electrons
 /
 &phonons
   nq1=8,
   nq2=8,
   nq3=1,
 /
```

Notice, in particular, the following paramters:

- `dump_gR` in the namelist `control` tell the program to save the auxiliary file containing the Hamiltonian and the electron-phonon coupling matrix elements in real space. Since we are not calculating any property of the system the namelist `electron` and `phonons` are essentially empty. Only `nq1, nq2,nq3` have to be supplied in order to specifie the q-points mesh where the input electron-phonon coupling matrix elements were computed (in this case, a 8x8x1 q-grid).

Note: In order to keep everything tidy you can use keep the el-ph matrix elements in a separate folder and use the variable:

&control > elphmat_dir="<path-to-folder>"

Run this preliminary calculation using:

```
mpirun -n <NPROC> {$path_to_epiq}/bin/epiq.x -inp dump.in > dump.out
```

After the dump has ended, some output files have been produced.

- The output file is binary and named `G_and_H.bin` contains the Hamiltonian and the electron-phonon coupling matrix elements in the Wannier representation. If, however, `ascii_G_and_H=.true.` is added in the `&control` namelist then the produced output file is readable and is named `G_and_H.asc`.
- .dat files containing the real space localization of the Hamiltonian and the electron-phonon coupling matrix elements.

9.4.2 Check real space localization

If the Wannier transformation is well converged, the matrix elements are optimally localized in real space. Always check their localization.

Excercise:

Using the two-columns files:

"MoS2_gw_R_ph.mu*.dat.pe_1" $|R|$ $\sum_{m,n} |g^{\nu}_{m,n}(0, \mathbf{R})|^2$

"MoS2_gw_R_el.mu*.dat.pe_1" $|r|$ $\sum_{m,n} |g^{\nu}_{m,n}(\mathbf{r}, 0)|^2$

plot the averaged modulus of the electron-phonon matrix elements as a function of the distance in real space $|R|$. Are they localized?

9.4.3 Phonon linewidth calculation

We would now focus on one of the system properties that can be calculated with EPIq: the phonon linewidth $\gamma_{\mathbf{q},\nu}$. We will consider the Allen phonon linewidth, which is defined by the following equation:

$$\gamma_{\mathbf{q},\nu} = \frac{4\pi\omega_{\mathbf{q},\nu}}{N_k} \sum_{m,n} \sum_{\mathbf{k}} |g^{\nu}_{m,n}(\mathbf{k},\mathbf{q})|^2 \delta(\epsilon_{\mathbf{k}+\mathbf{q},m} - \epsilon_F)\delta(\epsilon_{\mathbf{k},n} - \epsilon_F)$$

Example of input file for linewidth calculation of monolayer MoS_2

We first calculate the mode-resolved γ at the M-point of the Brillouin zone. In the following example we perform the calculation for phonon of momentum $\mathbf{q} = \mathbf{M}$ for two values of the electronic smearing. The input parameters are explained in detail in the EPIq manual . Here is the input file:

```
&control
  prefix='MoS2',
  calculation='ph_linewidth',
  read_dumped_gr=.true.,
  dump_gR=.false.,
  elphmat_dir="./mat_elem/"
  out2json=.true.
 /

 &electrons
      ngauss=0,
      sigma_min=0.01,
      sigma_max=0.05
      nsigma=5,
  /

  &phonons
      use_alternative_dyn=.true.
      prefix_alt_dyn='./dyn_mat/dynq'
      Fourier_interp_dyn=.true.,
      nq1=8,nq2=8,nq3=1,
  /

  k-points
  automatic
  4 4 1 0 0 0

  q-points
  crystal
  1
  0.5 0 0 1 ! M
```

The linewidth calculation is then started by:

```
mpirun -n <NPROC> {$path_to_epiq}/bin/epiq.x -inp lw.in > lw.out
```

Note that this calculation is done within the harmonic approximation, as we are using the dynamical matrices indicated by the variable `prefix_alt_dyn='./dyn_mat/dynq'`.

Parameters

- The linewidth calculation is selected by setting the `calculation` parameter equal to `'ph_linewidth'` in the `&control` namelist.
- In the namelist `control`, `read_dumped_gr`, which is set to `.true.`, indicating that electron-phonon coupling matrix elements can be read from `G_and_H.bin`)
- In the namelist `electrons`, `sigma_min`, `sigma_max`, `ngauss` and `nsigma` specify maximum, minimum, type and number of electronic smearing values to use. `efermi` and `ef_from_input` specify the initial guess for the Fermi level (the Fermi level calculated by Quantum ESPRESSO is usually a good guess) and whether the Fermi level should be re-calculated by epiq (`ef_from_input` equal to `.false.`) or set from input (`ef_from_input` equal to `.true.`). The variable `thr_compute_k` is used to restrict the calculation only to k-points possessing at least one eigenvalue in the specified energy region near the Fermi level.
- In the namelist `phonons`, `Fourier_interp_dyn` equal to `.true.` asks to interpolate the dynamical matrices for the q-points that do not belong to the Wannier grid. Alternatively, *EPIq* gives the opportunity to read eigenvalues and eigenvectors produced by `matdyn.x` of the Quantum ESPRESSO package (`matdyn.eig` file), putting `Fourier_interp_dyn=.false.` and `read_modes=.true.` in input, or even to directly read dynamical matrices from a `matdyn.dyn` file, putting `Fourier_interp_dyn=.false.` and `read_modes=.false.` .

Output files

If the `out2json` flat is set to `.true.`, the file `MoS2_lambda.json` will be produced. It can be automatically parsed using python as in the following lines where the variable `q_pts` is a "dict" whose entries are the results of the calculation for each q-point.

```
import json
ff = './MoS2_lambda.json'
with open(ff,'r') as f:
    q_pts = json.load(f)

first_q = q_pts["1"]

print("Fraction coordinate of the q point:", first_q["xq_frac"])
print("Electronic temperaturs used:", first_q["T"])
print("Results for the first mode:", first_q["1"])
print("Frequency of the second mode in meV:", first_q["2"]["freq"])
```

Here, a simple python script to plot the linewidth esteemed with the Allen formula:

```
import matplotlib.pyplot as plt
import numpy as np
for q in list(q_pts.values())[:]:
    for mod in range(1,10):
        plt.plot( q["T"],q[f'{mod}']["gamma_allen"],label=r"$\omega_"+f"{mod}
↪$={q[f'{mod}']['freq']:.0f} (meV)",linestyle='--' )
    plt.xlabel('T  (eV)')
    plt.ylabel(r'$\gamma(T)$  (meV)')
```

(continues on next page)

(continued from previous page)

```
    plt.legend(title=f"q={ np.round(np.array(q['xq_frac']),2) }")
    plt.show()
```

The other output file, `MoS2_lambda.d`, contains all the properties calculated by EPIq. The way this file is formatted is specified in output file, `lw.out`. Notice in particular that the first column contains the electronic smearing, while the second column contains the Allen linewidth.

Exercise:

Perfom a convergence study of the linewidth at $\mathbf{M}$ as function of the smearing and the k-mesh density. What is the minimum temperature at which the linewidth of the eighth mode is to be considered at convergence with a k-mesh of 8x8x1 ? And of 12x12x1?

Exercise:

Which mode shows larger smearing dependence?

Inclusion of anharmonicity

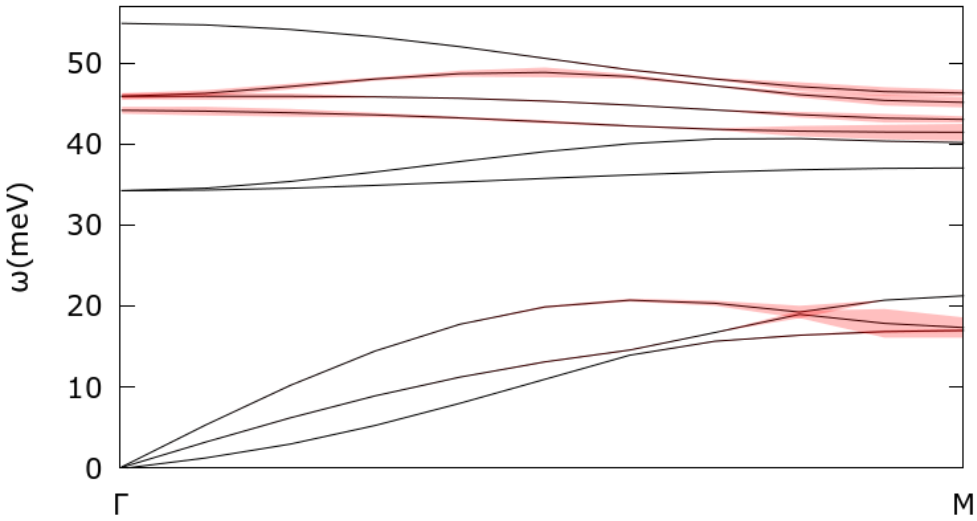
Now, we want to observe what changes with the inclusion of **anharmonic effects**. To this aim, we need to correctly specify the prefix of the Free-energy Hessian matrices calculated within the SSCHA using `prefix_alt_dyn='./dyn_mat/MoS2.Hessian.dyn'`.

Exercise:

Compute the linewidth at $\mathbf{q} = \mathbf{M}$ using anharmonic dynamical matrix computed thanks to the SSCHA code. Why it seems like the first and the second mode are exchanged with respect to the harmonic dynamical matrices? Which are the modes presenting a larger anharmonic correction?

Dispersion along a line

Now we want to perform the calculation of phonon linewidth along a certain crystalline direction and produce a plot like this one:



where the thickness of the lines is proportional to the calculated phonon linewidth. In order to to this, we repeat the phonon linewidth calculation, this time considering the whole Γ -M path:

```
&control
   dump_gR=.false.,
   read_dumped_gr=.true.,
   prefix='MoS2',
   calculation='ph_linewidth',
   elphmat_dir="./mat_elem/"
&end
&electrons
   ngauss=0,
   sigma_min=0.01,
   sigma_max=0.02
   nsigma=2,
&end
&phonons
   use_alternative_dyn=.true.
   prefix_alt_dyn='./dyn_mat/MoS2.Hessian.dyn'
   Fourier_interp_dyn=.true.,
   nq1=8,nq2=8,nq3=1,
&end
k-points
automatic
8 8 1  0  0  0
q-points
crystal
11
0.001 0 0 1
0.05 0 0 1
0.10 0 0 1
```

(continues on next page)

(continued from previous page)

```
...
...
0.50 0 0 1
```

Once the linewidth calculation has finished, we can obtain a plottable file using the `linewidth_path.x` post-processing tool. Here is an example of input file:

```
&input_lambda
  prefix='MoS2'
  lkp_sequential=.true.
  sigma_min=0.01,
  sigma_max=0.02,
  nsigma=2,
  chosen_sigma=0.01
&end
 3.159998   0.000000   0.000000
-1.579999   2.736639   0.000000
 0.000000   0.000000  19.141895
 crystal
 11
 0.001 0 0 1
 0.05 0 0 1
 0.10 0 0 1
 ...
 ...
 0.50 0 0 1
```

Notice the parameter `chosen_sigma`, which specifies what smearing will be used to produce the plottable file, and the lattice parameters at the end of the namelist. The post-processing is executed as follows:

```
{$path_to_epiq}/bin/linewidth_path.x < path.in > path.out
```

Finally, use the following gnuplot script plots the q-resolved linewidth for the acoustic modes:

```
set ylabel '{/Symbol w}(meV)'
set xlabel 'Gamma - M'
set style fill transparent solid 0.25
pl for [i=0:9] 'MoS2_lw_path.d' every 9::i u 1:2 w l lt rgb 'black'␣
↪title ''
repl  for [i=0:9] 'MoS2_lw_path.d' every 9::i u ($1):($2-$3/2):($2+
↪$3/2)\
w filledc lt rgb 'red' title ''
```

Exercise:

Try to produce two plots of the whole phonon spectrum, comparing the harmonic and the anharmonic result. Do you observe any differences?

Advanced tutorial: Migdal-Eliashberg calculation using SSCHA Hessian matrices

EPIq also allows to solve the anisotropic Eliashberg equations on the imaginary axis in order to calculate the **k**-resolved superconducting gap. We give an example for the superconducting gap of doped monolayer MoS_2 at T= 1 K.

```
&control
    dump_gR=.false.,
    read_dumped_gr=.true.,
    prefix='MoS2',
    calculation='migdal_eliashberg',
    elphmat_dir="./mat_elem/"
    &end
    &electrons
    theta=1,
    efermi=-2.0,
    ef_from_input=.false.,
&end
&phonons
    use_alternative_dyn=.false.
    prefix_alt_dyn='./dyn_mat/MoS2.Hessian.dyn'
    read_modes=.false.,
    nq1=8,
    nq2=8,
    nq3=1,
&end
&input_migdal
    initialize='step',
    gap_threshold=0.025,
    sigma_me=0.1,
    ME_Fermi_thickness=0.4,
    mustar=0.1,
    nmatsu=64,
    nitermax=100,
    alpha_mix_me=0.5,
    gap_init=4.0,
&end
nkfs
64 64 1
40
```

Run the *EPIq* calculation as follows:

```
mpirun -n 4 {$path_to_epiq}/bin/epiq.x <input_ME > out_ME
```

Note that for the `migdal_eliashberg` calculation, the number of **k**-points (40 in this case) must be a multiple of the mpi processes (4 in this case).

Notice the following parameters in the input file:

- In the `&electrons` namelist, `theta=1.0` specifies the temperature of the calculation in Kelvin.
- In the `&input_migdal` namelist, `sigma_me` specifies the broadening (in eV) to be used in the

calculation and `ME_Fermi_thickness` the energy range around the Fermi level where electron eigenvalues are considered in the calculation. `nmatsu` indicates the number of Matsubara frequencies to be employed in the sum (see here for further details:)

- The code solves the equation generating a certain number of random **k**-points, defined by the `64 64 1` grid in input and having an eigenvalue on the Fermi surface.

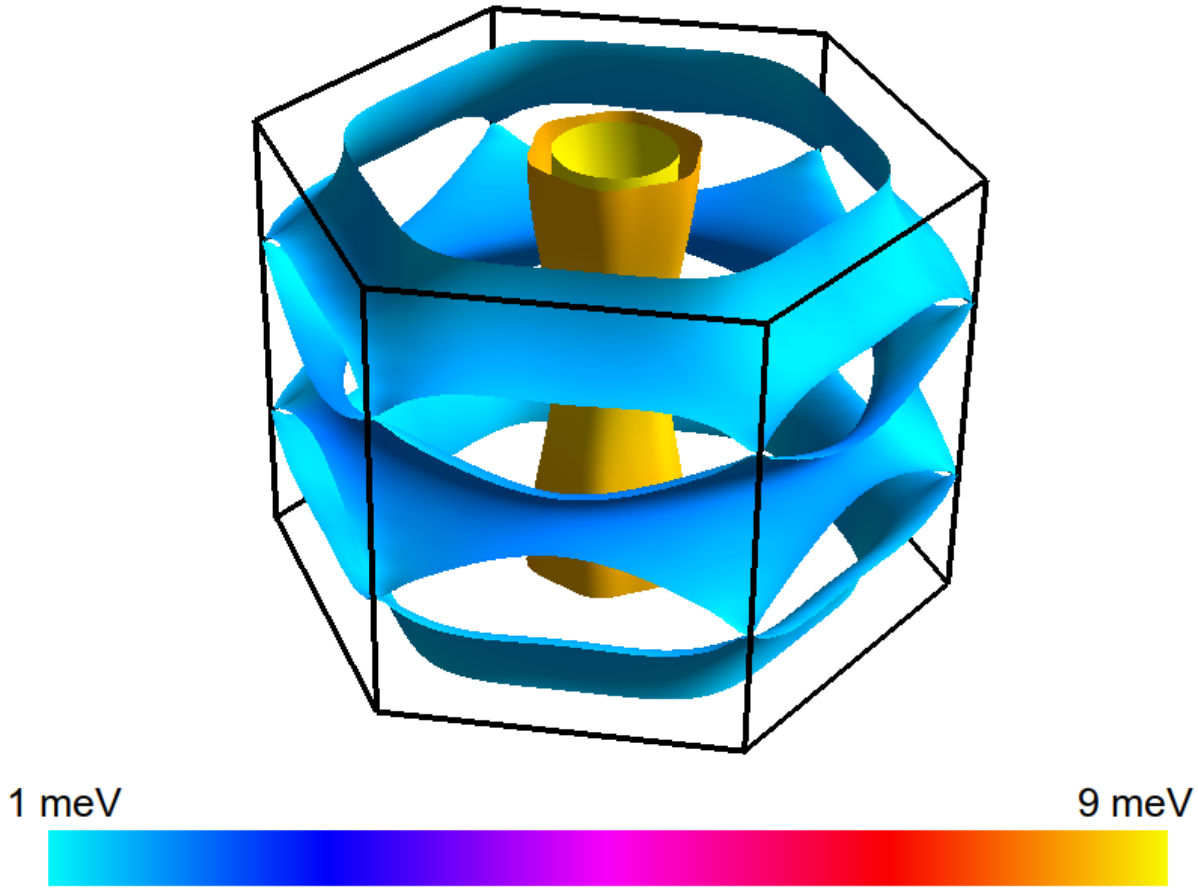
EPIq also calculates the Fermi surface using the grid specified after `nkfs` (64 64 1 here). After EPIq is done, two output files are produced:

- MgB2.bxsf contains the Fermi surface in the XCrysDen .bxsf format.
- MgB2_ME.d contains the Fermi-surface resolved Eliashberg gap.

You can produce a plot of the Fermi-surface resolved superconducting gap using the `plot_ME_fs.x` post processing. Execute it as:

```
{$path_to_epiq}/bin/plot_ME_fs.x
```

This is, for example, what you get for MgB_2, with its famous double gap:



Try to obtain the same kind of plot for MoS_2 using fermisurfer (github repository) typing:

```
fermisurfer MoS2.frmsf
```

Exercise:

Perform a convergence study of the average superconducting gap as function of the number of Mat-

subara frequencies and k-points. What is the required value of nmatsu and k-points to have a precision better than 0.1 meV?

Note: In order to perform a complete calculation of anharmonic electron phonon coupling related properties within SSCHA+EPIq requires the following steps:

1. The SSCHA code → First, compute the free energy Hessian within the stochastic self-consistent harmonic approximation (SSCHA).
2. Quantum ESPRESSO code → Then, compute electron-phonon coupling matrix elements following the instructions reported in the EPIq site https://the-epiq-team.gitlab.io/epiq-site/docs/tutorials/step1/.
3. Wannier90 code → Identify the unitary transformation connecting the Bloch and the maximally-smooth gauge (required to interpolate the electron-phonon coupling matrix elements in the Wannier basis).
4. EPIq code → Perform the electron-phonon copuling interpolation in the Wannier basis and calculate physical properties including the anharmonic correction from SSCHA.

All these open-source software can be downloaded following the instruction in each website.

If you want to know more about the full procedure, please take a look at the tutorials on the EPIq site.

CHAPTER

TEN

HANDS-ON-SESSION 9 - THERMAL CONDUCTIVITY CALCULATIONS WITH THE SSCHA

In previous lessons we saw how to calculate vibrational properties of material using SSCHA. Now we will use this acquired knowledge to calculate lattice thermal conductivity of materials. We will need dynamical matrices (**auxiliary ones, not hessians**) and the third order force constants (we already calculated them when we checked the dynamical stability of the system). With these we can calculate materials' harmonic (phonon frequencies and phonon group velocities) and anharmonic properties (phonon lifetimes and spectral functions) which is all we need to calculate lattice thermal conductivity.

10.1 Lattice thermal conductivity of silicon

As a first exercise let's calculate lattice thermal conductivity of silicon. Silicon is very harmonic material which means it's lattice thermal conductivity is very high. This also makes it a good test case to check the equivalence of Green-Kubo and Boltzmann transport equation approaches in the limit of vanishing anharmonicity. To speed up the calculation we will use Tersoff potential to obtain the second and third order force constants. We will do this with this simple script:

```
import numpy as np
from quippy.potential import Potential
from ase import Atoms
import ase.io
from ase.eos import calculate_eos
from ase.units import kJ
from ase.phonons import Phonons as AsePhonons
from ase.constraints import ExpCellFilter
from ase.optimize import BFGS, QuasiNewton
import cellconstructor as CC
import cellconstructor.Phonons
import cellconstructor.Structure
import sscha, sscha.Ensemble, sscha.SchaMinimizer, sscha.Relax


# This function will use ASE to give us a starting
# guess for dynamical matrices
def get_starting_dynamical_matrices(structure_filename,
        potential, supercell):
    atoms = ase.io.read(structure_filename)
    atoms.set_calculator(potential)
```

(continues on next page)

(continued from previous page)

```
    ecf = ExpCellFilter(atoms)
    qn = QuasiNewton(ecf)
    qn.run(fmax=0.0005)

    structure = CC.Structure.Structure()
    structure.generate_from_ase_atoms(atoms, get_masses = True)
    dyn = CC.Phonons.compute_phonons_finite_displacements(structure,
    potential, supercell = supercell)
    dyn.Symmetrize()
    dyn.ForcePositiveDefinite()

    eos = calculate_eos(atoms)
    v0, e0, B = eos.fit()
    bulk = B / kJ * 1.0e24

    return dyn, bulk


# Our input variables
temperature = 100.0
nconf = 1000
max_pop = 1000

# Load in Tersoff potential
pot = Potential('IP Tersoff', param_filename=
'../06_the_SSCHA_with_machine_learning_potentials/ip.parms.Tersoff.xml')
supercell = tuple((4*np.ones(3, dtype=int)).tolist())
dyn, bulk = get_starting_dynamical_matrices(
'../06_the_SSCHA_with_machine_learning_potentials/POSCAR', pot, supercell)


# Generate the ensemble and the minimizer objects
ensemble = sscha.Ensemble.Ensemble(dyn, T0=temperature,
supercell = dyn.GetSupercell())
ensemble.generate(N = nconf)
minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
minimizer.min_step_dyn = 0.1
minimizer.kong_liu_ratio = 0.5
minimizer.meaningful_factor = 0.001
minimizer.max_ka = 1000

# Relax structure
relax = sscha.Relax.SSCHA(minimizer, ase_calculator = pot,
N_configs = nconf, max_pop = max_pop, save_ensemble = True)
relax.vc_relax(static_bulk_modulus = bulk,
ensemble_loc = "directory_of_the_ensemble")

# Generate ensemble for third-order FC with the relaxed dynamical matrices
new_ensemble = sscha.Ensemble.Ensemble(relax.minim.dyn, T0=temperature,
```

(continues on next page)

(continued from previous page)

```
supercell = relax.minim.dyn.GetSupercell())
new_ensemble.generate(N = nconf*5)
new_ensemble.compute_ensemble(pot, compute_stress = True,
stress_numerical = False, cluster = None, verbose = True)
# We minimize the free energy with this new ensemble
new_minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(new_ensemble)
new_minimizer.minim_struct = False
new_minimizer.set_minimization_step(0.1)
new_minimizer.meaningful_factor = 0.001
new_minimizer.max_ka = 10000
new_minimizer.init()
new_minimizer.run()
new_minimizer.dyn.save_qe('final_dyn')
# Update weights with a new dynamical matrice
new_ensemble.update_weights(new_minimizer.dyn, temperature)

# Calculate Hessian and the third order tensor (return_d3 = True)
dyn_hessian, d3_tensor = new_ensemble.get_free_energy_hessian(include_v4 =␣
↪False,
    get_full_hessian = True, return_d3 = True)
np.save("d3.npy", d3_tensor)
dyn_hessian.save_qe('hessian_dyn')
```

Here we used 4x4x4 supercell. **You will need to converge results with respect to the size of the supercell**. A good check for the convergence could be the decay of the second and third order force constants with the distance. Now that we have second and third order force constants, we can calculate lattice thermal conductivity. For this we provide following script:

```
from __future__ import print_function
from __future__ import division

import numpy as np
import cellconstructor as CC
import cellconstructor.Phonons
import cellconstructor.ForceTensor
import cellconstructor.ThermalConductivity
import time

dyn_prefix = 'final_dyn'
nqirr = 8

SSCHA_TO_MS = cellconstructor.ThermalConductivity.SSCHA_TO_MS
RY_TO_THZ = cellconstructor.ThermalConductivity.SSCHA_TO_THZ
dyn = CC.Phonons.Phonons(dyn_prefix, nqirr)

supercell = dyn.GetSupercell()

fc3 = CC.ForceTensor.Tensor3(dyn.structure,
dyn.structure.generate_supercell(supercell), supercell)
```

(continues on next page)

(continued from previous page)

```
d3 = np.load("d3.npy")
fc3.SetupFromTensor(d3)
fc3 = CC.ThermalConductivity.centering_fc3(fc3)

mesh = [10,10,10]
smear = 0.03/RY_TO_THZ

tc = CC.ThermalConductivity.ThermalConductivity(dyn, fc3,
kpoint_grid = mesh, scattering_grid = mesh, smearing_scale = None,
smearing_type = 'constant', cp_mode = 'quantum', off_diag = False)

temperatures = np.linspace(200,1200,10,dtype=float)
start_time = time.time()
tc.setup_harmonic_properties(smear)
tc.write_harmonic_properties_to_file()

tc.calculate_kappa(mode = 'SRTA', temperatures = temperatures,
write_lifetimes = True, gauss_smearing = True, offdiag_mode = 'wigner',
kappa_filename = 'Thermal_conductivity_SRTA', lf_method = 'fortran-LA')

print('Calculated SSCHA kappa in: ', time.time() - start_time)

tc.calculate_kappa(mode = 'GK', write_lineshapes=False,
ne = 1000, temperatures = temperatures,
kappa_filename = 'Thermal_conductivity_GK')

print('Calculated SSCHA kappa in: ', time.time() - start_time)
# Save ThermalConductivity object for postprocessing.
tc.save_pickle()
```

Important parts of the script are:

- We define mesh on which we calculate phonon properties to be the same as the mesh we are calculating scattering processes (variable `mesh`). This does not have to be true. In most cases `scattering_grid` can be much smaller than `kpoint_grid`. **Converge your results with respect to both grids.**
- We use smearing approach to satisfy energy conservation laws. There are two ways: constant and adaptive. In the case of `smearing_type = 'constant'` we have to provide smearing value in **Ry** as the argument to `setup_harmonic_properties` function. In case we choose adaptive smearing, the smearing constant will be different for different phonon modes. We still can define global variable `smearing_scale` with which we multiply precomputed smearing constants. `smearing_scale = 1.0` works pretty well in most cases. **Converge your results with respect to smearing variables.**
- `off_diag` variable defines whether we are doing calculation with what was termed as *coherent transport*. This will be important for highly anharmonic materials with large bunching of phonon modes.
- Function `calculate_kappa` does most of the work. Here we will describe main options:

- `mode` defines which method to use to calculate lattice thermal conductivity. Options are **SRTA** which is Boltzmann transport equation solution in single relaxation time approximation and **GK** (Dangic et al.) which is Green-Kubo method that uses phonon spectral functions instead of phonon lifetimes. These two modes should give similar results in low anharmonicity materials, but different in strongly anharmonic ones.
- `gauss_smearing` defines how we treat energy conservation in the calculation of self energy. If **True** it will use Gaussian functions, if **False** it will use Lorentzian functions. In case of Gaussian smearing real part of the self energy is calculated using Kramers-Kronig transformation.
- `offdiag_mode` defines how we calculate coherent transport if `mode = 'SRTA'`. Two options: **wigner** (Simoncelli et al.) and **gk** (Isaeva et al.). If `mode` is **GK**, coherent transport is included naturally.
- `lf_method` defines how lifetimes are calculated in case `mode = 'SRTA'`. In short you want to keep *fortan-*, and then add *LA* or *P*. These should give more or less same results. Additional option is *SC* where we solve phonon lifetimes self-consistently, meaning we account for the phonon lineshifts.
- `ne` defines the number of frequency steps if we are calculating phonon lineshapes. Also important in case of `lf_method = 'SC'` because we solve self-consistent equation on a grid of frequency values linearly interpolating real and imaginary part. Larger is better. **Converge your results with respect to ne.**

This calculation should take a few minutes. The results are save in the **kappa_filename**.

Question:

If we check results we see that *SRTA* and *GK* results are different. Why? How can we improve this calculation?

10.2 Lattice thermal conductivity of GeTe

As a second example we will calculate lattice thermal conductivity of GeTe. GeTe is a highly anharmonic material with a phase transition from rhombohedral to cubic phase at around 700 K. This means its lattice thermal conductivity is very low. Additionally, it should show difference between *SRTA* and *GK* methods.

For SSCHA minimization we can calculate atomic properties using Gaussian Approximation Potential developed for this material. However, in the interest of time we provided the dynamical matrices calculated at 0 K and the third order force constantsin the folder 09_Thermal_conductivity_calculations_with_the_SSCHA.

Exercise:

Calculate lattice thermal conductivity of GeTe up to 1200 K (sample temperature from 300 K every 200 K). Is there a difference between *GK* and *SRTA* methods?

Exercise:

Check if coherent transport has an influence on thermal conductivity in this material system.

Finally, in case we want to do some postprocessing we can load in the previously saved `ThermalConductivity` object and access all previously calculated data. For example, we can calculate phonon density of states calculated with auxiliary force constants and the one calculated with phonon lineshapes:

```
from __future__ import print_function
from __future__ import division

# Import the modules to read the dynamical matrix
import numpy as np
import cellconstructor as CC
import cellconstructor.Phonons
import cellconstructor.ForceTensor
import cellconstructor.ThermalConductivity
import time
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

tc = CC.ThermalConductivity.load_thermal_conductivity()

# See at which temperatures we calculated stuff
tc.what_temperatures()

key = list(tc.lineshapes.keys()) # Get Ts for lineshapes

# DOS calculated from auxiliary force constants
harm_dos = tc.get_dos()
# Temperature dependent DOS calculated from lineshapes
# first two arrays are raw data
# second two is gaussian smoothed results \
#for the distance between energy points de
anharm_dos = tc.get_dos_from_lineshapes(float(key[-1]), de = 0.1)


# Plot results
fig = plt.figure(figsize=(6.4, 4.8))
gs1 = gridspec.GridSpec(1, 1)
ax = fig.add_subplot(gs1[0, 0])
ax.plot(harm_dos[0], harm_dos[1], 'k-',
zorder=0, label = 'Harmonic')
ax.plot(anharm_dos[0], anharm_dos[1], 'r-',
zorder=0, label = 'Anharmonic raw @ ' + key[-1] + ' K')
ax.plot(anharm_dos[2], anharm_dos[3], 'b-',
zorder=0, label = 'Anharmonic smooth @ ' + key[-1] + ' K')
ax.set_xlabel('Frequency (THz)')
ax.set_ylabel('Density of states')
```

(continues on next page)

(continued from previous page)

```
ax.legend(loc = 'upper right')
ax.set_ylim(bottom = 0.0)
fig.savefig('test.pdf')
plt.show()
```

Additionally, if we want to check a specific phonon lineshape (for example at Γ point) we can do it with a bit of hacking:

```
for iqpt in range(tc.nkpt):
    if(np.linalg.norm(tc.k_points[iqpt]) == 0.0):
        break

energies = np.arange(len(tc.lineshapes[key[-1]][0,0]),
dtype=float)*tc.delta_omega + tc.delta_omega
tc.write_lineshape('Lineshape_at_Gamma',
tc.lineshapes[key[-1]][iqpt], iqpt, energies, 'no')
```

CHAPTER

ELEVEN

APENDIX: THE EKHI CLUSTER

For this SSCHA School we will provide access to the local CFM cluster named Ekhi.

11.1 ekhi.cfm.ehu.es

Ekhi cluster, designed specifically for novel Quantum ESPRESSO calculations, is composed of 28 computing nodes with two Xeon Cascade Lake-SP 6230 processors (40 computing cores) and 96 GB of memory in each node, with an Infiniband FDR interconnection network, giving a total of 1120 cores and 2.7 TB of memory.

You can view the cluster information with *sinfo*:

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
all*         up 2-00:00:00      1    mix ekhi31
all*         up 2-00:00:00     20  alloc ekhi[1-5,7-9,11-12,15-17,20-21,24-25,
↪27-29]
all*         up 2-00:00:00     10   idle ekhi[6,10,13-14,18-19,22-23,26,30]
fat          up 14-00:00:0      1  alloc ekhi29
fat          up 14-00:00:0      1   idle ekhi30
long         up   infinite      4  alloc ekhi[1-4]
test         up      30:00      1    mix ekhi31
test         up      30:00     20  alloc ekhi[1-5,7-9,11-12,15-17,20-21,24-25,
↪27-29]
test         up      30:00     10   idle ekhi[6,10,13-14,18-19,22-23,26,30]
```

This is an example of a batch input for the cluster Ekhi:

```
#!/bin/bash
#SBATCH --job-name=Test_ESPRESSO    # Job name
#SBATCH --mail-type=NONE            # Mail events (NONE, BEGIN, END, FAIL,␣
↪ALL)
#SBATCH --mail-user=                # Where to send mail
#SBATCH -p test                     # queue (in this example the queue for␣
↪test)
#SBATCH --nodes=1                   # Run all processes on a single node
#SBATCH --ntasks=40                 # Number of processes
#SBATCH --time=00:30:00             # Time limit hrs:min:sec
#SBATCH --output=job.log            # Standard output and error log
```

(continues on next page)

(continued from previous page)

```
module load intel/2021a

node=`hostname`
echo "********************"
echo "job run at node " $node
echo "********************"
echo ""
#copies the directory from where you submited the job to lscratch

if [[ ! -e /lscratch/$USER ]]; then
    mkdir /lscratch/$USER
fi

cp -r $SLURM_SUBMIT_DIR  /lscratch/$USER/$SLURM_JOB_ID
cd /lscratch/$USER/$SLURM_JOB_ID
export NPROCS=$SLURM_NTASKS
rm slurm*.out
##############################################################################
module load QuantumESPRESSO
##############################################################################
echo "put your jobs here"
mpirun -np 40 pw.x -npool 20 -i input_espresso.pwi > output_espresso.pwo
##############################################################################


cd $SLURM_SUBMIT_DIR
echo "Making backup..."
mkdir BACKUP

for file in *
do
    if [ $file != BACKUP ] && [ $file != slurm*out ]; then
        mv $file BACKUP/$file
    fi
done

echo "Copying files from /lscratch..."
cp -r /lscratch/$USER/$SLURM_JOB_ID/* $SLURM_SUBMIT_DIR/

echo "Deleting files from /lscratch..."
rm -r /lscratch/$USER/$SLURM_JOB_ID

echo "Deleting the BACKUP..."
cd $SLURM_SUBMIT_DIR
for file in *
do
    if [ $file != BACKUP ] && [ $file != slurm*out ] && [ -e BACKUP/$file ]
    then
        rm -r BACKUP/$file
```

(continues on next page)

(continued from previous page)

```
    fi
done
rmdir BACKUP

echo "DONE"
```

This batch is run with:

```
sbatch run.sh
```

A typical usage of this cluster from a SSCHA script code includes:

```
#------------------------------------------------------------------------
username = user_name   # Put here your login name for the cluster.
pseudo = {"Sr": "Sr.pbesol-spn-kjpaw_psl.1.0.0.UPF",
          "Ti": "Ti.pbesol-spn-kjpaw_psl.1.0.0.UPF",
          "O" : "O.pbesol-n-kjpaw_psl.1.0.0.UPF"}
input_params = {"tstress" : True, # Print the stress in the output
        "tprnfor" : True, # Print the forces in the output
        "tstress" : True, #output stresses
        "ecutwfc" : 70,  #The wavefunction energy cutoff for plane-waves (Ry)
        "ecutrho" : 700, # The density energy cutoff (Ry)
        "mixing_beta" : 0.4,  # The mixing parameter in the self-consistent␣
↪calculation
        "conv_thr" : 1e-9,    # The energy convergence threshold (Ry)
        "degauss" : 0.03,  # Smearing temperature (Ry)
#                "smearing" : "mp",
        "pseudo_dir" : "./pseudo/",
        "occupations" : "fixed", #smearing or fixed (fixed for insulators␣
↪with a gap; gaussian smearing for metals; )
        "disk_io" : "none"}

k_points = (8,8,8) # The k points grid (you can alternatively specify a␣
↪kspacing)
k_offset = (1,1,1) # The offset of the grid (can increase convergence)

self.espresso_calc = Espresso(pseudopotentials = pseudo, input_data = input_
↪params,
                kpts = k_points, koffset = k_offset)
my_hpc = sscha.Cluster.Cluster(pwd = None)
# We setup the connection info
my_hpc.hostname = "{}@ekhi.cfm.ehu.es".format(username) # The command to␣
↪connect via ssh to the cluster (pippo@login.cineca.marconi.it)
my_hpc.workdir = "/scratch/{}/my_calculation".format(username) # the␣
↪directory in which the calculations are performed

# Now we need to setup the espresso
# First we must tell the cluster where to find him:
my_hpc.binary = "pw.x -npool NPOOL -i  PREFIX.pwi > PREFIX.pwo"
# Then we need to specify if some modules must be loaded in the submission␣
↪script
```

(continues on next page)

(continued from previous page)

```
my_hpc.load_modules = """
# Here this is a bash script at the beginning of the submission
# We can load modules

module load QuantumESPRESSO
export OMP_NUM_THREADS=1
"""

# All these information are independent from the calculation
# Now we need some more specific info, like the number of processors, pools␣
↪and other stuff
my_hpc.n_cpu = 40 # We will use the 40 processors
my_hpc.n_nodes = 1 #In 1 node
my_hpc.n_pool = 10 # This is an espresso specific tool, the parallel CPU are␣
↪divided in 4 pools

# We can also choose in how many batch of jobs we want to submit␣
↪simultaneously, and how many configurations for each job
my_hpc.batch_size = 10
my_hpc.job_number = 10
# In this way we submit 10 jobs, each one with 10 configurations (overall 100␣
↪configuration at time)

# We give 25 seconds of timeout
my_hpc.set_timeout(25)

# We can specify the time limit for each job,
my_hpc.time = "03:00:00" # 3 hours

# Create the working directory if none on the cluster
# And check the connection
my_hpc.setup_workdir()
#------------------------------------------------------------------------
```

Then we can use in relax with:

```
relax = sscha.Relax.SSCHA(minim, ase_calculator = espresso_calc, N_
↪configs=configurations, max_pop=20, cluster = my_hpc)
```