SSCHA School 2026 Tutorials

Hands-on sessions on the Stochastic Self-Consistent Harmonic Approximation

Ion Errea, Raffaello Bianco, Lorenzo Monacelli, Dorde Dangic, Giovanni Marini

Contents

Chapter 1

Setup guide

The hands-on sessions of the 2026 SSCHA School will be run on a virtual machine, with the SSCHA software pre-installed. It is important that all the partecipants have the VM ready and working **before** the beginning of the school, as during the limited time available for the school we will not go through the software installation and the download and extracted data is significative, so it must be performed beforehand.

In the following we detail all the step required to be ready to run the tutorials and partecipate to the hands-on session

1.1 The virtual machine

To run the tutorials, we recommend using a custom version of the Quantum Mobile virtual machine. The download links for Intel or Apple Silicon chips are the following:

1. Intel x86/AMD64
2. Apple Silicon/ARM64

(A huge thanks to Marnik Bercx, Nicola Colonna and all the people involved from Quantum Mobile for building this version for the ICTP/MARVEL College: Materials simulations in the age of AI).

To start the virtual machine, you must import it inside Oracle VirtualBox.

NOTE: VirtualBox version 7.2 at least is required. The virtual machine will not work on old VirtualBox versions to update or install the latest version of virtualbox, visit the official site.

Run and try that the virtual machine works **before** the start of the school. Note that running the VM for the first time requires a lot of free space on the disc (up to about 30 GB).

1.2 Installing required force fields

Once inside the virtual machine, open a terminal. Every tutorial will be run inside the SSCHA conda environment. To access it use

```
conda activate sscha
```

This needs to be done every time you open a new terminal. A `(sscha)` text should appear at the beggining of the prompt line. Then install the force-fields required for running the tutorials:

```
pip install quippy-ase F3ToyModel
```

You are now ready for the hands-on sessions 1-4.

1.3 EPIq installation

Deactivate the conda environment if it is active, typing `conda deactivate`. Enter the folder where you want to install epiq in your virtual machine and type on the command line

```
git clone --depth 1  --branch develop https://gitlab.com/the-epiq-team/epiq.git
```

then, enter the just created directory with `cd epiq` and install it typing the following command

```
./configure && make all CC="gcc -std=gnu89"
```

all the executables (.x files) will be installed in epiq/bin.

1.4 Clone the repository with the tutorials

All the tutorials have force-fields and files that you need to download. This can be done from the github repository. Open the terminal, enter the folder where you want to download the tutorials and type

```
git clone https://github.com/SSCHAcode/sscha_school_2026.git
```

This will create a folder named `sscha_school_2026` with all the files needed for the tutorials. Note that the force field is compressed in a tar.gz file. You need to extract it. To do so, enter the folder with `cd sscha_school_2026` and type

```
cd Materials
tar xf force_field_CsPbI3.tar.gz
```

This is important to locate the force field in the correct path.

1.5 After the school: Installing the SSCHA on your personal computer

If you want to run production runs of the SSCHA, you can install them in your personal computer. For that, follow the instructions provided in the official web page: www.sscha.eu.

Chapter 2

Hands-on Session 1 - First SSCHA Simulations: Free Energy and Structural Relaxations

In this hands-on session, we provide few simple examples of how to use the SSCHA code for its most basic calculations: free energy calculations and structural relaxations considering anharmonic effects. The example will be based on calculations performed using a GAP machine learning potential for the $Pm-3m$ phase of $CsPbI_3$ perovskite structure.

The variational minimization of the free energy within the SSCHA requires several steps:

- Define a starting crystal structure with definite positions to start the minimization of the free energy assuming ionic wave functions are centeres at these positions.
- Define starting auxiliary force constants in a supercell as starting force constants for the minimization of the free energy. Even if this is not necessary, usually harmonic force constants are a good starting point.
- Create random ionic positions in the chosen supercell based on the probability distribution function defined by the initial positions and force constants.
- Calculate Born-Oppenheimer (BO) total energies, BO atomic forces, and BO lattice stresses for all these configurations. This has to be done with an external code independent of the SSCHA. The theory level used for these calculations (DFT, Monte Carlo, Empirical Force Field, Machine Learning potential…) determined the theory level behid the SSCHA.
- Perform the minimization of the free energy, optimizing the atomic positions (centers of the SSCHA ionic wave functions) and auxiliary force constants.
- The minimization will stop if the minimum of the free energy is found within the determined threshold or if the calculation has gone out of the statistical save range.
- In the latter case, one should restart the calculation by creating a new set of configurations using as input the obtained positions and auxiliary force constants in the previous run. The process should be repeated until convergence is found.

In this tutorial we will show how to perform such calculations, starting from the most basic usage of the code to more advanced automatic type of calculations.

In the following, we assume we work inside the `Tutorials/01-Free_Energy_Structural_Relaxations/` directory. All paths are given relative to this directory.

2.1 A standard manual calculation with an external force engine

The starting ionic (centroid) positions and force constants are read by the SSCHA from Quantum Espresso dynamical matrix files. We provide some harmonic calculations for $CsPbI_3$ in the folder `../../Materials/CsPbI3_cubic_harmonic_*`. In these files the structure of the crystal is given, and each of the files corresponds to the force constants matrix obtained in reciprocal space for all the irreducible q points in a $2 \times 2 \times 2$ grid. In this case there are 4 different q points in the irreducible grid. This is enough to create the force constants in a commensurate $2 \times 2 \times 2$ supercell.

In this first example we will do all the steps of the SSCHA minimization separately, one by one. In the first step we will read the input positions and force constants from the harmonic dynamical matricesand create 50 random configurations based on probability distribution functions defined by these positions and force constants. This is done by the stript `create_configurations.py`, which is copied here:

```
# Import cellconstructor needed things
import cellconstructor as CC
import cellconstructor.Phonons

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

#------------------------------------------
# We set up the parameters for the calculation

# The temperature for the calculation in K
TEMPERATURE = 450

# We tell the system that we are goin to create the configurations
# of the first population. This is just used for labeling purposes.
POPULATION = 1

# We determine the number of configurations that we will generate
# for this population
N_RANDOM = 50

# We tell where the starting dynamical matrices in reciprocal space are
# for the q points commensurate with the supercell we want to use for
# the calculation. These dynamical matrices should be in Quantum Espresso
# format.
START_DYN = "../../Materials/CsPbI3_cubic_harmonic_"
# In case we want to create the configurations from the result of a previous
 ↪  minimimization we ca use this starting dynamcial matrices
#START_DYN = namefile='dyn_end_population'+str(POPULATION-1)+'_'
#
# We indicate how many irreducible q points are there for the q points in the grid.
NQIRR = 4

#------------------------------------------

# Step 1: load the harmonic dynamical matrices and the crystal structure

dyn = CC.Phonons.Phonons(START_DYN, NQIRR) # Load them and read the structure
dyn.ForcePositiveDefinite()               # Force positive phonons in case there are
 ↪  imaginary phonon frequencies
```

```
dyn.Symmetrize()                          # Impose symmetries and the acoustic sum rule

# Step 2: create the ensemble, the configurations for which we have to calculate the
# forces, energies and stresses

ensemble = sscha.Ensemble.Ensemble(dyn, TEMPERATURE)
ensemble.generate(N_RANDOM)

# Step 3: We save the ensemble

namefile='population'+str(POPULATION)+'_ensemble' # Name of the folder to store the
 ↪  ensemble
ensemble.save(namefile, POPULATION)
```

After running thie python script as

```
python create_configurations.py
```

we will obtain a folder with the name `population1_ensemble`, where files `u_population1_*.dat` and `scf_population1_*.dat` are stored. In the former how much atom is displaced from the starting positions is given in Bohr units and in the latter positions and lattice vectors in the supercell with the corresponding displacement ready to be used in DFT calculations are given. This run will aslo produce `dyn_start_population1_*` dynamical matrices, which are those used to create the configurations, basically the input ones with positive frequencies and ASR forced.

The next step is to calculate the BO total energies, BO atomic forces and BO stresses for all these configurations. These has to be stored in the folder `population1_ensemble` with names `energies_supercell_population1.dat`, where all energiescalculated in the supercell in Ry are concatanated; `forces_population1_*.dat`, where in each file the forces of the atoms are given in Ry/Bohr (one line per atom with Cartesian coordinates in the same line); and `pressure_population1_*.dat`, where in each file the 3×3 stress tensor is given in Ry/Bohr^3. These calculations can be done externally with any code and prepare these files a posteriori.

In this case we perform the calculations with the GAP machine learning potential given in `../../Materials/`. This is the python script `run_force_energy_engine.py` that does that:

```
# Import cellconstructor needed things
import cellconstructor as CC
import cellconstructor.Phonons

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

# Import the module to be able to run the ML potential
from quippy.potential import Potential

# Import ASE for reading data
import ase

#------------------------------------------
# We set up the parameters for the calculation

# The temperature for the calculation in K
TEMPERATURE = 450
```

```
# We tell the system that we are goin to create the configurations
# of the first population. This is just used for labeling purposes.
POPULATION = 1

# We determine the number of configurations that we will generate
# for this population
N_RANDOM = 50

# We define the dynamical matrices that generated the ensemble and the folder where
# the ensemble is stored
START_DYN = 'dyn_start_population'+str(POPULATION)+'_'
folder_with_ensemble = 'population'+str(POPULATION)+'_ensemble'
NQIRR = 4

# We dertermine the potential to be used
POTENTIAL ='../../Materials/GAP_1.xml'
calc = Potential("IP GAP", param_filename=POTENTIAL)

#-----------------------------------------

# We define numbers for the change of units between ASE and SSCHA
RyToEv = 13.605693
BohrToAngstrom = 0.529177

# We perform the calculations and write the results in the appropriate files with the
↪ appropriate units
energy_file=str(folder_with_ensemble)+'/energies_supercell_population'+str(POPULATION)+'.dat'
with open(energy_file, "w") as f_energy:

    struct = CC.Structure.Structure()
    for i in range(N_RANDOM):

↪   namefile=str(folder_with_ensemble)+'/scf_population'+str(POPULATION)+'_'+str(i+1)+'.dat'
        struct.read_scf(namefile)
        ase_struct = struct.get_ase_atoms()
        ase_struct.set_calculator(calc)
        energy = ase_struct.get_potential_energy()
        forces = ase_struct.get_forces()
        stress = ase_struct.get_stress()

        # --- Write energy in Ry ---
        f_energy.write(f"{energy/RyToEv:.10f}\n")

        # --- Write forces file in Ry/Bohr ---
        forces_file =
↪   str(folder_with_ensemble)+'/forces_population'+str(POPULATION)+'_'+str(i+1)+'.dat'
        with open(forces_file, "w") as f_forces:
            for f in forces:
                f_forces.write(f"{f[0]* (BohrToAngstrom / RyToEv):.10f} {f[1]*
↪   (BohrToAngstrom / RyToEv):.10f} {f[2]* (BohrToAngstrom / RyToEv):.10f}\n")

        # --- Write stress file in Ry/Bohr^3 ---
```

```
        stress_file =
    str(folder_with_ensemble)+'/pressure_population'+str(POPULATION)+'_'+str(i+1)+'.dat'
        with open(stress_file, "w") as f_stress:
            # ASE gives 6 components: xx yy zz yz xz xy
            f_stress.write(f"{stress[0]* (BohrToAngstrom**3 / RyToEv)}  {stress[5]*
    (BohrToAngstrom**3 / RyToEv)} {stress[4]* (BohrToAngstrom**3 / RyToEv)}\n")
            f_stress.write(f"{stress[5]* (BohrToAngstrom**3 / RyToEv)}  {stress[1]*
    (BohrToAngstrom**3 / RyToEv)} {stress[3]* (BohrToAngstrom**3 / RyToEv)}\n")
            f_stress.write(f"{stress[4]* (BohrToAngstrom**3 / RyToEv)}  {stress[3]*
    (BohrToAngstrom**3 / RyToEv)} {stress[2]* (BohrToAngstrom**3 / RyToEv)}\n")
```

After running thie python script as

```
python run_force_energy_engine.py
```

in the `population1_ensemble` folder we will obtain all the energies, forces and stresses that the SSCHA needs for the minimization. If an external code is done for this part, one should prepare these files externally.

With all these files ready, the variational minimization is ready to be done. We will do that with the script `minimize.py`:

```
import sys,os

# Import cellconstructor needed things
import cellconstructor as CC
import cellconstructor.Phonons

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

# Import the module to be able to run the ML potential
from quippy.potential import Potential

# Import ASE for reading data
import ase

# Import matplotlib and numpy
import numpy as np

# NumPy moved ComplexWarning in newer versions. This block keeps the script
# compatible with both older and newer NumPy releases.

#------------------------------------------
# We set up the parameters for the calculation

# The temperature for the calculation in K
TEMPERATURE = 450

# We tell the system that we are goin to create the configurations
# of the first population. This is just used for labeling purposes.
POPULATION = 1

# We determine the number of configurations that we will generate
# for this population
```

```
N_RANDOM = 50

# We define the dynamical matrices that generated the ensemble and the folder where
# the ensemble is stored
START_DYN = 'dyn_start_population'+str(POPULATION)+'_'
folder_with_ensemble = 'population'+str(POPULATION)+'_ensemble'
NQIRR = 4
dyn = CC.Phonons.Phonons(START_DYN, NQIRR)

# We dertermine the potential to be used
POTENTIAL ='../../Materials/GAP_1.xml'
calc = Potential("IP GAP", param_filename=POTENTIAL)

# We load the ensemble and read the results of the configurations
ensemble = sscha.Ensemble.Ensemble(dyn, TEMPERATURE)
ensemble.load(folder_with_ensemble, population = POPULATION, N = N_RANDOM)
ensemble.has_stress = True

# Define the minimization

minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)

# Define the steps for the centroids and the force constants

minimizer.min_step_dyn = 0.005         # The minimization step on the dynamical matrix
minimizer.min_step_struc = 0.05        # The minimization step on the structure
minimizer.kong_liu_ratio = 0.2         # The parameter that estimates whether the
 ↪  ensemble is still good
minimizer.meaningful_factor = 0.000001 # How much small the gradient should be before I
 ↪  stop?

# Store the gradients during the minimization
ioinfo = sscha.Utilities.IOInfo()
ioinfo.SetupSaving("minim_{}".format(POPULATION))

# Let's start the minimization
minimizer.init()
minimizer.run(custom_function_post = ioinfo.CFP_SaveAll)

# Save the results
minimizer.finalize()
namefile='dyn_end_population'+str(POPULATION)+'_'
minimizer.dyn.save_qe(namefile)
```

This script can be run as

```
python minimize.py > population1.log
```

The evolution of the minimization can be seen in the `population1.log` file, where in the end we will see why the minimization stopped, the obtained free energy in the unit cell, the gradients of the free energy, effective Kong-Liu sampling, the force on the centroids, and the stress tensor. The minimization probably stopped because the statistical sampling worsened beyond the input criteria. The obtained auxiliary dynamical matrices and positions are stored in the files `dyn_end_population1_*`.

Exercise:

Considering that the minimization stopped because it got out of the statistical range, perform a new minimization step (population = 2) starting from the output of the first minimization.

2.2 An automatic calculation for fixed lattice parameters

The manual calculations explained before can be easy if not many populations are needed to converge the result and we have no option but performing the energy-force calculations externally, for instance in a cluster. However, with force fields or Machine Learning potentials the SSCHA can make all the process above automatically, population after population, in one single script. This can be done by making use of the relax feature of the SSCHA. An example can be done with the following script `sscha_relax.py`:

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

from quippy.potential import Potential

import sys, os

import ase
import numpy as np

import warnings

# NumPy moved ComplexWarning in newer versions. This block keeps the script
# compatible with both older and newer NumPy releases.
try:
    ComplexWarning = np.exceptions.ComplexWarning
except AttributeError:
    ComplexWarning = np.ComplexWarning

warnings.filterwarnings("ignore", category=ComplexWarning)

TEMPERATURE = 450 # K
NQIRR = 4
START_DYN = "../../Materials/CsPbI3_cubic_harmonic_"
POTENTIAL = "../../Materials/GAP_1.xml"
N_CONFIGS = 50


# Load the harmonic dynamical matrix
dyn = CC.Phonons.Phonons(START_DYN, NQIRR)

# Force positive phonons
dyn.ForcePositiveDefinite()

# Impose symmetries and ASR
dyn.Symmetrize()

# Load the interatomic Potential for CsPbI3
calc = Potential("IP GAP", param_filename=POTENTIAL)
```

```
# Run the sscha
ensemble = sscha.Ensemble.Ensemble(dyn, TEMPERATURE)
minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
minimizer.meaningful_factor = 0.000001
minimizer.set_minimization_step(0.001)
minimizer.kong_liu_ratio = 0.2
relax = sscha.Relax.SSCHA(minimizer, calc, N_configs = N_CONFIGS)

# Setup to save the minimization details every good minimization step
# Allows to plot minimization results
ioinfo = sscha.Utilities.IOInfo()
ioinfo.SetupSaving("minimization_data")
relax.setup_custom_functions(custom_function_post = ioinfo.CFP_SaveAll)

relax.relax()

# Save the final ensemble
relax.minim.ensemble.save_bin("data", 1)
relax.minim.dyn.save_qe("sscha_auxiliary_dyn_")
```

This script can be run as:

```
python sscha_relax.py > sscha_relax.log
```

With this script the minimization continues until the gadients of the free energy become smaller than the input value, realtive to the error. In this case, the final dynamical matrices are stored in `sscha_auxiliary_dyn_`. The evolution of the minimization can be visualized more clearly using the following command

```
sscha-plot-data minimization_data
```

by plotting the evolution of the minimizaton stored in the files `minimization_data.freqs` and `minimization_data.dat`. We should obtain results similar to those in fig. 2.1 and fig. 2.2.

Be careful that the number of configurations used is rather low and one should check convergence with respect to the number of configurations. One way of checking that is by running a final new step with more configurations starting from the result obtained at the end of the minimization.

2.3 An automatic calculation relaxing also the lattice parameters

Even if the automatic calculation simplifies the procedure enormously, the SSCHA can further simplify the calculations by relaxing also the lattice parameters to a target pressure. This is done by replacing the relax feature by the vcrelax one as shown in the `sscha_vcrelax.py` script:

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

from quippy.potential import Potential

import sys, os

import ase
import numpy as np
```

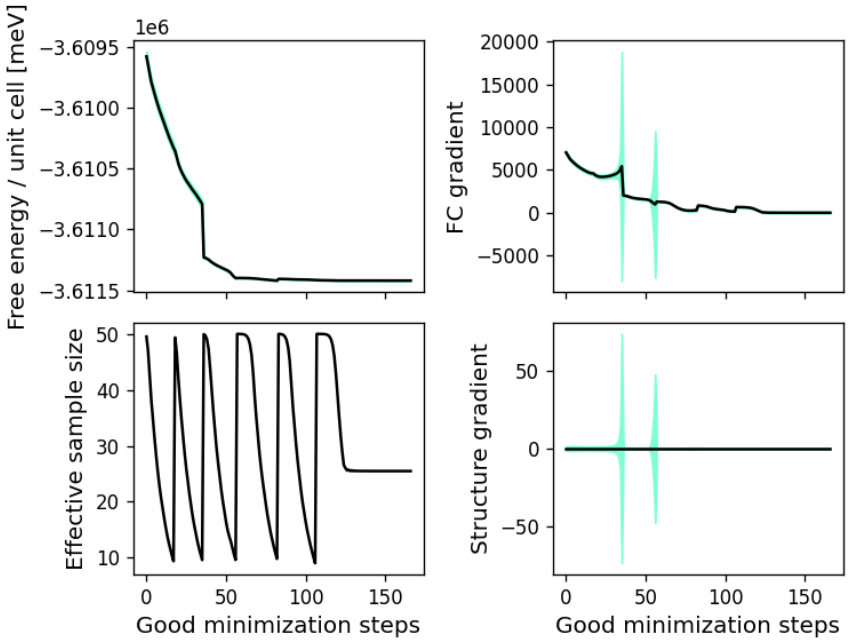

Figure 2.1: Evolution of the free energy, gradients with respect to the force constants and positions, and Kong Liu ratio along the minimization. The width of the points is related to the stochasti error.

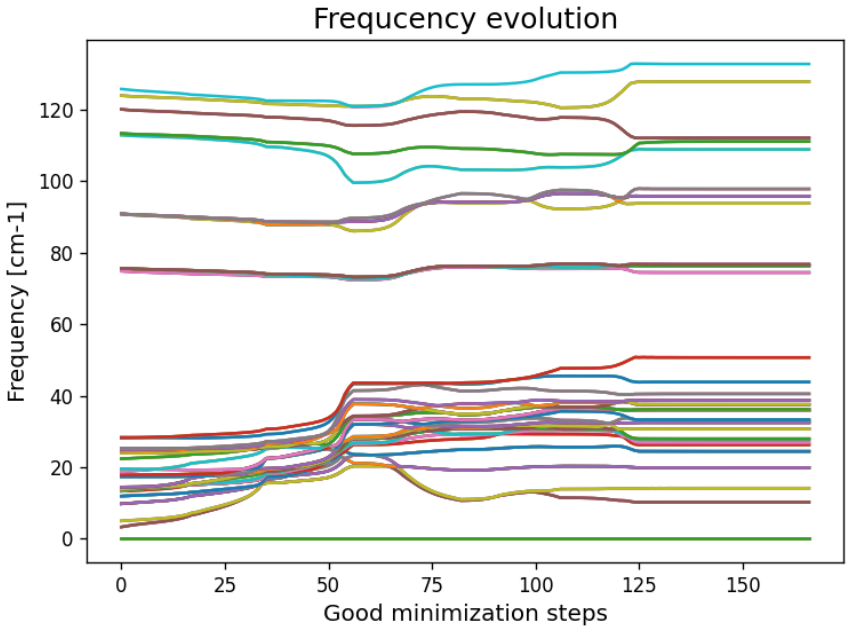

Figure 2.2: Evolution of the auxiliary frequencies along the minimization.

```
import warnings

# NumPy moved ComplexWarning in newer versions. This block keeps the script
# compatible with both older and newer NumPy releases.
try:
    ComplexWarning = np.exceptions.ComplexWarning
except AttributeError:
    ComplexWarning = np.ComplexWarning

warnings.filterwarnings("ignore", category=ComplexWarning)

TEMPERATURE = 450 # K
NQIRR = 4
START_DYN = "../../Materials/CsPbI3_cubic_harmonic_"
POTENTIAL = "../../Materials/GAP_1.xml"
N_CONFIGS = 50


# Load the harmonic dynamical matrix
dyn = CC.Phonons.Phonons(START_DYN, NQIRR)

# Force positive phonons
dyn.ForcePositiveDefinite()

# Impose symmetries and ASR
dyn.Symmetrize()

# Load the interatomic Potential for CsPbI3
calc = Potential("IP GAP", param_filename=POTENTIAL)

# Run the sscha
ensemble = sscha.Ensemble.Ensemble(dyn, TEMPERATURE)
minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
minimizer.meaningful_factor = 0.000001
minimizer.set_minimization_step(0.001)
minimizer.kong_liu_ratio = 0.2
relax = sscha.Relax.SSCHA(minimizer, calc, N_configs = N_CONFIGS)

# Setup to save the minimization details every good minimization step
# Allows to plot minimization results
ioinfo = sscha.Utilities.IOInfo()
ioinfo.SetupSaving("minimization_data")
relax.setup_custom_functions(custom_function_post = ioinfo.CFP_SaveAll)

# We perform a variable cell relaxation with the target pressure in GPaf
relax.vc_relax(target_press = 0)

# Save the final ensemble
relax.minim.ensemble.save_bin("data", 1)
relax.minim.dyn.save_qe("sscha_auxiliary_dyn_")
```

This script can be run as:

```
python sscha_vcrelax.py > sscha_vcrelax.log
```

The final auxiliary dynamical matrices in this case will correspond to a different lattice parameter as specified in the `sscha_auxiliary_dyn_` files.

Exercise:

Calculate the lattice parameter as a function of temperature.
You should obtain something similar to fig. 2.3.
As you can see the result is not very smooth although a clear positive trend is seeing. This calculation was perfomred with only 50 configurations per population. This noisy result suggests that the result can improve with more configurations.

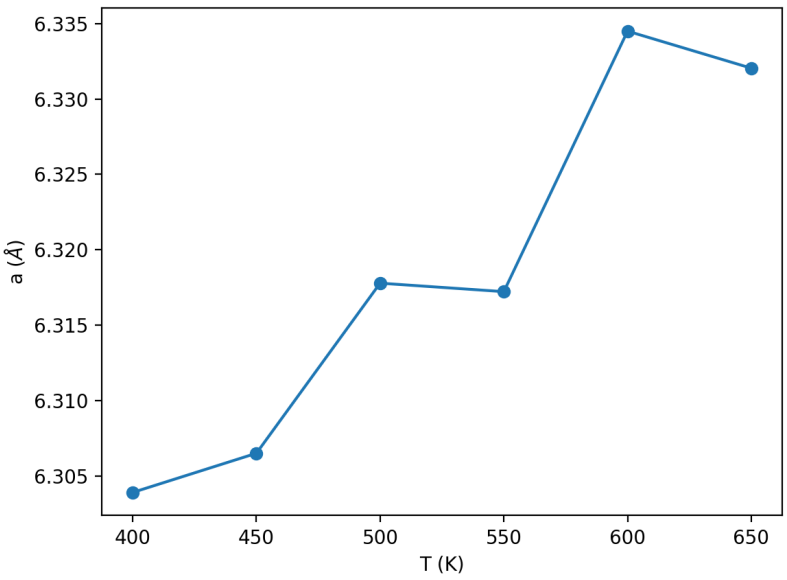

Figure 2.3: Lattice parameter obained at different temperatures.

Chapter 3

Hands-on Session 2 - Calculating Second-Order Phase Transitions with the SSCHA

In this hands-on session, we will learn how to study second-order phase transitions using the SSCHA.

The calculations will be performed using a toy force-field model for ferroelectric transitions in FCC materials, originally introduced by Ai *et al.* Phys. Rev. B **90**, 014308 (2014). This model captures the essential physics of ferroelectric instabilities in FCC compounds while keeping the computational cost low. In the present tutorial, we will focus on the specific case of SnTe. To do so, first activate the appropriate environment and install the required package:

```
conda activate sscha
pip install F3ToyModel
```

The toy model requires the harmonic dynamical matrices of the system. In this tutorial, we use dynamical matrices computed from first principles on a $2 \times 2 \times 2$ q-point grid of the Brillouin zone. These are provided in the `ffield_dynq_*` files located in the `Materials/tutorial_02` directory.

3.1 Structural Instability: Calculation of the Hessian

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

In structural *displacive* phase transitions, the order parameter is associated with the amplitude of a collective pattern of atomic displacements that lowers the symmetry of the high-symmetry phase. The transition occurs when the free-energy curvature along this distortion pattern vanishes and eventually becomes negative.

Thus the key quantity for investigating displacive second-order phase transitions is the free-energy Hessian with respect to the centroid positions,

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

where $F(R)$ is the free energy as a function of the average atomic positions (centroids).

Within the SSCHA framework, an analytical expression for the free-energy Hessian was derived by Bianco *et al.* Phys. Rev. B **96**, 014111 (2017). The free-energy Hessian (i.e. the curvature of the free-energy surface) can be expressed in compact tensor notation as

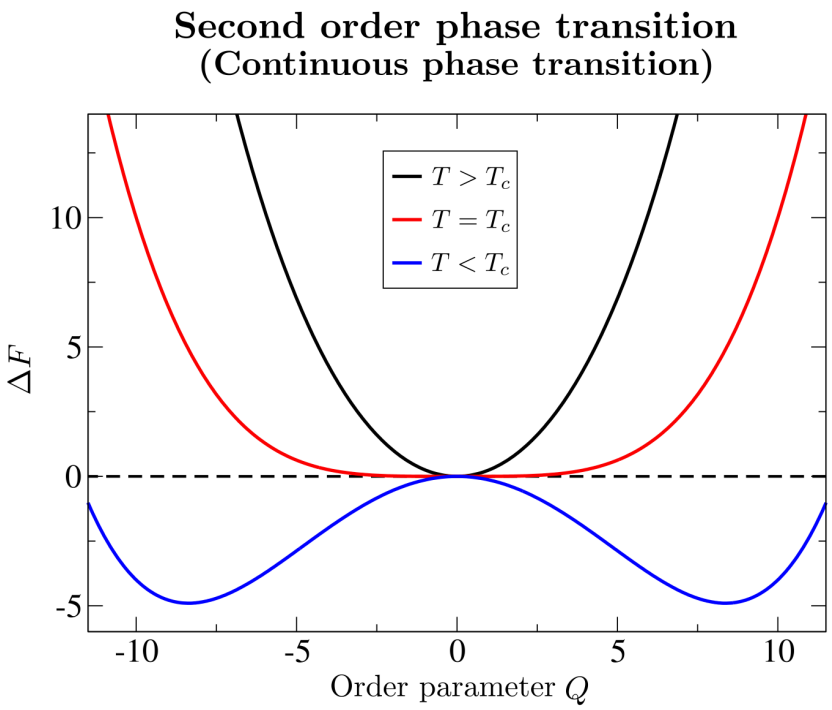

Figure 3.1: Landau's theory of second-order phase transitions.

$$\frac{\partial^2 F}{\partial R\,\partial R} = \Phi + \overset{(3)}{\Phi}\,\Lambda \left[1 - \overset{(4)}{\Phi}\,\Lambda\right]^{-1} \overset{(3)}{\Phi},$$

with Φ_{ab} the SSCHA force-constant matrix, which is also equal to the statistical average of the potential-energy Hessian over the SSCHA probability distribution ρ

$$\Phi_{ab} = \left\langle \frac{\partial^2 V}{\partial R_a \partial R_b} \right\rangle_\rho$$

and similarly

$$\overset{(3)}{\Phi}_{abc} = \left\langle \frac{\partial^3 V}{\partial R_a \partial R_b \partial R_c} \right\rangle_\rho, \qquad \overset{(4)}{\Phi}_{abcd} = \left\langle \frac{\partial^4 V}{\partial R_a \partial R_b \partial R_c \partial R_d} \right\rangle_\rho.$$

The tensor Λ_{abcd} depends on the eigenvalues and eigenvectors of the SSCHA dynamical matrix, $D_{ab} = \Phi_{ab}/\sqrt{M_a M_b}$.

The free-energy Hessian can be evaluated within the SSCHA framework using a stochastic approach through the function call:

```
ensemble.get_free_energy_hessian()
```

The dynamical matrix associated with the free-energy Hessian,

$$D^{(F)} = \frac{1}{\sqrt{M_a M_b}} \frac{\partial^2 F}{\partial R_a\,\partial R_b},$$

is the quantity that determines the stability of the crystal beyond the harmonic approximation, as it incorporates quantum, thermal, and anharmonic effects through the curvature of the free-energy surface.

By diagonalizing $D^{(F)}$ and tracking its eigenvalues as a function of temperature, we can identify when one of them becomes zero. This signals that the free-energy curvature vanishes along a specific direction in configuration space and therefore marks the onset of a second-order phase transition. The corresponding eigenvector defines the instability mode, i.e. the pattern of atomic displacements that lowers the free energy.

3.2 Practical Example: The Ferroelectric Transition in SnTe

To illustrate the concepts introduced above, we now consider a practical example: the ferroelectric phase transition in SnTe.

SnTe crystallizes at room temperature and ambient pressure in the NaCl structure (space group $Fm\bar{3}m$), known as the β-SnTe phase, in which two interpenetrating fcc sublattices of Sn and Te are present. Upon cooling below approximately 100 K, SnTe undergoes a structural phase transition to a rhombohedral phase (space group $R3m$), known as the α-SnTe phase.

The transition can be understood as a symmetry-lowering distortion of the high-symmetry cubic structure. It is commonly described as a two-step process: first, a relative polar displacement of the Sn and Te sublattices along the cubic [111] direction removes the inversion center; second, a rhombohedral strain develops along the same direction. In this tutorial, we focus exclusively on the polar distortion.

To describe this behavior, we employ a toy interatomic potential $V(u)$ expressed as a function of the atomic displacements $u_a = R_a - R_a^{\mathrm{eq}}$ from the equilibrium positions of the cubic rock-salt structure. Beyond the harmonic contribution, the model retains only third- and fourth-order anharmonic terms:

$$V(u) = \frac{1}{2}\sum_{ab}\phi_{ab}u^a u^b + \frac{1}{3!}\sum_{abc}\phi^{(3)}_{abc}u^a u^b u^c + \frac{1}{4!}\sum_{abcd}\phi^{(4)}_{abcd}u^a u^b u^c u^d.$$

For the harmonic contribution ϕ_{ab}, we use dynamical matrices computed from first principles on a $2\times2\times2$ q-point grid. The third- and fourth-order force constants are instead parametrized within the toy model. In particular, the third-order term $\phi^{(3)}_{abc}$ is controlled by a single parameter p_3, whereas the fourth-order term $\phi^{(4)}_{abcd}$ depends on two parameters, p_4 and $p_{4\chi}$. Throughout this tutorial, we use

$$p_4 = -0.022\,\mathrm{eV/\AA^4}, \qquad p_{4\chi} = -0.014\,\mathrm{eV/\AA^4}, \qquad p_3 = 0.036475\,\mathrm{eV/\AA^3}.$$

The harmonic force constants ϕ_{ab} exhibit negative eigenvalues, indicating that the high-symmetry cubic phase is dynamically unstable within the harmonic approximation. This instability manifests itself as imaginary phonon frequencies in the harmonic phonon spectrum. As a first step, let us compute and visualize the harmonic phonon dispersion of SnTe along a high-symmetry path of the Brillouin zone (BZ) using the following script:

```
import cellconstructor.Phonons
import cellconstructor.ForceTensor
import cellconstructor.Methods

import matplotlib.pyplot as plt

NQIRR = 3
PATH = "GXMGR"
N_POINTS = 1000

SPECIAL_POINTS = {
    "G": [0.0, 0.0, 0.0],
    "X": [0.5, 0.0, 0.0],
    "M": [0.5, 0.5, 0.0],
```

```
    "R": [0.5, 0.5, 0.5],
}

# Load HARM phonons
harm_dyn = cellconstructor.Phonons.Phonons("../../Materials/tutorial_02/ffield_dynq_",
    NQIRR)


# Get band path
qpath, data = cellconstructor.Methods.get_bandpath(
    harm_dyn.structure.unit_cell,
    PATH,
    SPECIAL_POINTS,
    N_POINTS)

xaxis, xticks, xlabels = data

# Fourier interpolation along q-path
harm_dispersion = cellconstructor.ForceTensor.get_phonons_in_qpath(
    harm_dyn,
    qpath)

nmodes = harm_dyn.structure.N_atoms * 3

plt.figure(dpi=150)
ax = plt.gca()

for i in range(nmodes):
    ax.plot(
        xaxis,
        harm_dispersion[:, i],
        color="r",
        lw=1)

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)

ax.set_xticks(xticks)
ax.set_xticklabels(xlabels)

ax.set_xlabel("Q path")
ax.set_ylabel("Phonons [cm$^{-1}$]")

plt.tight_layout()
plt.savefig("harm_dispersion.png", dpi=300)
plt.show()
```

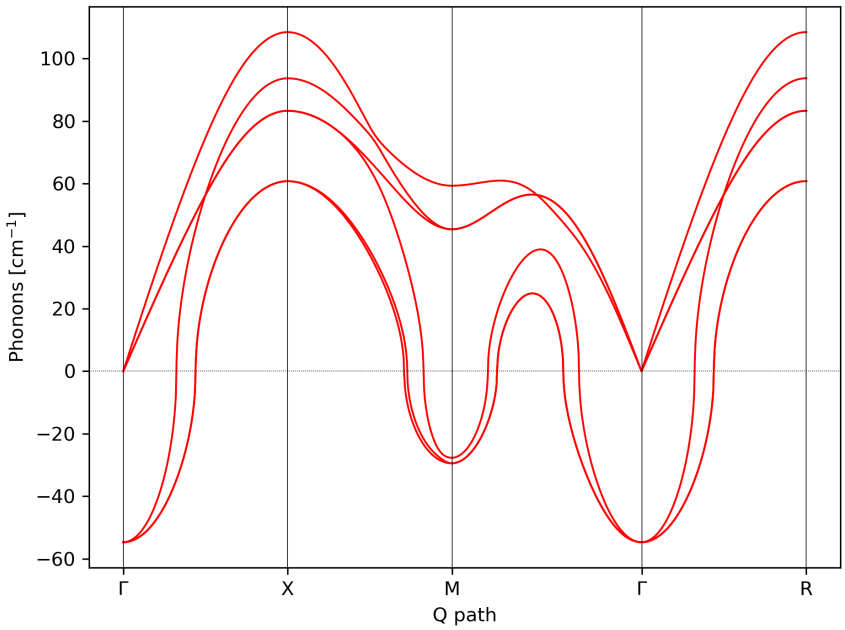

Figure 3.2: Harmonic phonon dispersion of SnTe.

Question

The harmonic phonon spectrum of the high-symmetry cubic phase contains imaginary frequencies, indicating a dynamical instability. Why, then, is the cubic phase experimentally observed above the critical temperature?

Answer

Harmonic **Free-energy Hessian**

$$\boxed{\frac{1}{\sqrt{M_a M_b}} \frac{\partial^2 V}{\partial R_a \partial R_b}} \quad \text{VS} \quad \boxed{\frac{1}{\sqrt{M_a M_b}} \frac{\partial^2 F}{\partial R_a \partial R_b}}$$

The harmonic approximation determines the stability of the high-symmetry structure from the curvature of the potential-energy surface.
However, the actual stability of a crystal is determined by the curvature of the free energy rather than by the potential energy alone, with the free energy including both quantum and thermal effects:

$$F = E - TS,$$

where the energy is determined by the full nuclear Hamiltonian

$$H = K + V.$$

Therefore, quantum fluctuations arising from the kinetic-energy operator K and thermal fluctuations associated with the entropic term TS can stabilize the high-symmetry phase even when the harmonic approximation predicts a dynamical instability. The SSCHA accounts for these effects non-perturbatively by directly evaluating the free-energy Hessian.

3.3 Calculation of the SSCHA Dynamical Matrix

As a first step, let us compute the SSCHA dynamical matrix for SnTe at $T = 0\,\mathrm{K}$. We proceed with this script:

```
import os
import numpy as np
import warnings

# CellConstructor modules
import cellconstructor as CC
import cellconstructor.Phonons

# SSCHA modules
import sscha
import sscha.Ensemble
import sscha.SchaMinimizer
import sscha.Relax

# Toy-model force field
import fforces as ff
import fforces.Calculator


# ============================================================
# Ignore NumPy ComplexWarning generated during SSCHA updates
# ============================================================

try:
    ComplexWarning = np.exceptions.ComplexWarning
except AttributeError:
    ComplexWarning = np.ComplexWarning

warnings.filterwarnings("ignore", category=ComplexWarning)


# ============================================================
# INPUT PARAMETERS
# ============================================================

# Temperature in Kelvin
TEMPERATURE = 0

# Number of irreducible q-points
NQIRR = 3

# Number of configurations used in the SSCHA ensemble
N_CONFIGS = 126

# Prefix of the starting dynamical matrices
START_DYN = "../../Materials/tutorial_02/ffield_dynq_"

# Dynamical matrices used to define the harmonic reference of the toy model
MODEL_DYN = "../../Materials/tutorial_02/ffield_dynq_"
```

```
# ============================================================
# DEFINE THE TOY FORCE FIELD
# ============================================================

# Load the harmonic dynamical matrices
ff_dyn = CC.Phonons.Phonons(MODEL_DYN, NQIRR)

# Create the toy-model calculator
ff_calculator = ff.Calculator.ToyModelCalculator(ff_dyn)

# Select the type of anharmonic toy potential
ff_calculator.type_cal = "pbtex"

# Anharmonic parameters
ff_calculator.p3 = 0.036475
ff_calculator.p4 = -0.022
ff_calculator.p4x = -0.014


# ============================================================
# CREATE OUTPUT DIRECTORY
# ============================================================

output_dir = f"RELAX/T_{TEMPERATURE}"
os.makedirs(output_dir, exist_ok=True)


# ============================================================
# LOAD STARTING DYNAMICAL MATRICES
# ============================================================

start_dyn = CC.Phonons.Phonons(START_DYN, NQIRR)

# If the starting dynamical matrices are not positive definite
# (i.e. imaginary phonon frequencies are present),
# enforce positive definiteness and reimpose crystal symmetries
# and the acoustic sum rule.

w, pols = start_dyn.DiagonalizeSupercell()

w_sorted = np.sort(w)
tol = -1e-4

if np.min(w_sorted[3:]) < tol:
    print("Non-acoustic imaginary phonon modes detected")
    start_dyn.ForcePositiveDefinite()
    start_dyn.Symmetrize()


# ============================================================
# CREATE SSCHA ENSEMBLE
# ============================================================
```

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


# ================================================================
# SETUP THE SSCHA MINIMIZER
# ================================================================

minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)

# Control minimization stability and convergence
minimizer.meaningful_factor = 0.01
minimizer.set_minimization_step(0.001)


# ================================================================
# SETUP SSCHA RELAXATION
# ================================================================

relax = sscha.Relax.SSCHA(
    minimizer,
    ase_calculator=ff_calculator,
    N_configs=N_CONFIGS
)


# ================================================================
# RUN THE SSCHA MINIMIZATION
# ================================================================

relax.relax()


# ================================================================
# SAVE FINAL SSCHA DYNAMICAL MATRICES
# ================================================================

relax.minim.dyn.save_qe(
    os.path.join(output_dir, "sscha_dyn_")
)
```

In this way, by minimizing the free energy, we obtain the SSCHA dynamical matrices. Plotting the phonon dispersion of the SSCHA dynamical matrices together with the harmonic phonon dispersion, we obtain the following result.

By definition, the SSCHA dynamical matrices are positive definite. As a consequence, structural instabilities cannot be detected directly from their eigenvalues. To assess the stability of the high-symmetry phase, we must compute the free-energy Hessian.

The computation of $\overset{(4)}{\Phi}$ is significantly more expensive than that of $\overset{(3)}{\Phi}$, both in terms of CPU time and memory consumption. For this reason, in this tutorial we adopt the so-called *bubble approximation*,

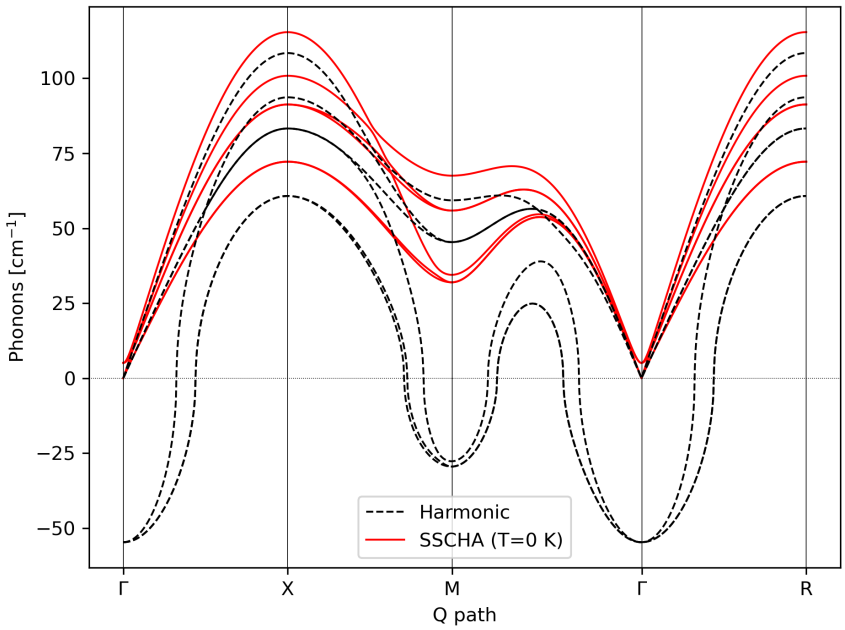

Figure 3.3: Harmonic vs SSCHA phonon dispersion at T=0 K.

$$\frac{\partial^2 F}{\partial R\,\partial R} \simeq \Phi + \overset{(3)}{\Phi}\,\Lambda\,\overset{(3)}{\Phi}\,.$$

Although the terms beyond the bubble approximation can be included in principle, they are often found to be negligible in practice. Nevertheless, their importance should always be assessed on a case-by-case basis.

3.4 Calculation of the Free-Energy Hessian

We now evaluate the free-energy Hessian at $T = 0\,\mathrm{K}$ using the following script:

```
import os
import numpy as np
import warnings

import cellconstructor as CC
import cellconstructor.Phonons

import sscha.Ensemble
import sscha.SchaMinimizer
import sscha.Relax

import fforces as ff
import fforces.Calculator


# ============================================================
# Ignore NumPy ComplexWarning generated during SSCHA updates
# ============================================================
```

```
try:
    ComplexWarning = np.exceptions.ComplexWarning
except AttributeError:
    ComplexWarning = np.ComplexWarning

warnings.filterwarnings("ignore", category=ComplexWarning)


# ============================================================
# INPUT PARAMETERS
# ============================================================

TEMPERATURE = 0
NQIRR = 3


# Number of configurations used for this first illustrative Hessian calculation.
# (A larger ensemble than that used in the SSCHA relaxation is
#  generally required to compute the free-energy Hessian accurately).
N_CONFIGS = 250
# A convergence study with respect to this parameter will be performed below.

# Already relaxed SSCHA dynamical matrices
SSCHA_DYN = f"RELAX/T_{TEMPERATURE}/sscha_dyn_"

# Dynamical matrices defining the harmonic part of the toy model
MODEL_DYN = "../../Materials/tutorial_02/ffield_dynq_"

# Output directory for the free-energy Hessian dynamical matrices
OUTPUT_DIR = f"HESSIAN/T_{TEMPERATURE}"
os.makedirs(OUTPUT_DIR, exist_ok=True)


# ============================================================
# DEFINE THE TOY FORCE FIELD
# ============================================================

ff_dyn = CC.Phonons.Phonons(MODEL_DYN, NQIRR)

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


# ============================================================
# LOAD RELAXED SSCHA DYNAMICAL MATRICES
# ============================================================

sscha_dyn = CC.Phonons.Phonons(SSCHA_DYN, NQIRR)
sscha_dyn.Symmetrize()
```

```
# ================================================================
# CREATE SSCHA ENSEMBLE
# ================================================================

ensemble = sscha.Ensemble.Ensemble(
    sscha_dyn,
    TEMPERATURE
)


# ================================================================
# SETUP THE SSCHA MINIMIZER
# ================================================================

minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
minimizer.meaningful_factor = 0.01
minimizer.set_minimization_step(0.001)


# ================================================================
# SETUP SSCHA RELAXATION
# ================================================================

relax = sscha.Relax.SSCHA(
    minimizer,
    ase_calculator=ff_calculator,
    N_configs=N_CONFIGS
)


# ================================================================
# REFINE THE SSCHA DYNAMICAL MATRIX WITH A LARGER ENSEMBLE
# ================================================================

relax.relax()

relax.minim.dyn.save_qe(
    os.path.join(OUTPUT_DIR, "refined_sscha_dyn_")
)


# ================================================================
# COMPUTE THE FREE-ENERGY HESSIAN
# ================================================================

# Reweight the ensemble using the refined dynamical matrix
ensemble.update_weights(minimizer.dyn, TEMPERATURE)

dyn_hessian = ensemble.get_free_energy_hessian(
    include_v4=False,       # bubble approximation
    get_full_hessian=True,  # full free-energy Hessian
```

```
    return_d3=False,        # do not save third-order SSCHA force constants
)


# ================================================================
# SAVE THE FREE-ENERGY HESSIAN DYNAMICAL MATRICES
# ================================================================

dyn_hessian.save_qe(
    os.path.join(OUTPUT_DIR, "hessian_dyn_")
)

print("\nDone.")
```

We can now compute the phonon dispersion corresponding to the free-energy Hessian dynamical matrix $D^{(F)}$ and compare it with the SSCHA phonon dispersion.

Exercise

Compute and plot the phonon dispersion associated with the free-energy Hessian dynamical matrix $D^{(F)}$ along a high-symmetry path of the Brillouin zone. Compare the result with the corresponding SSCHA phonon dispersion.

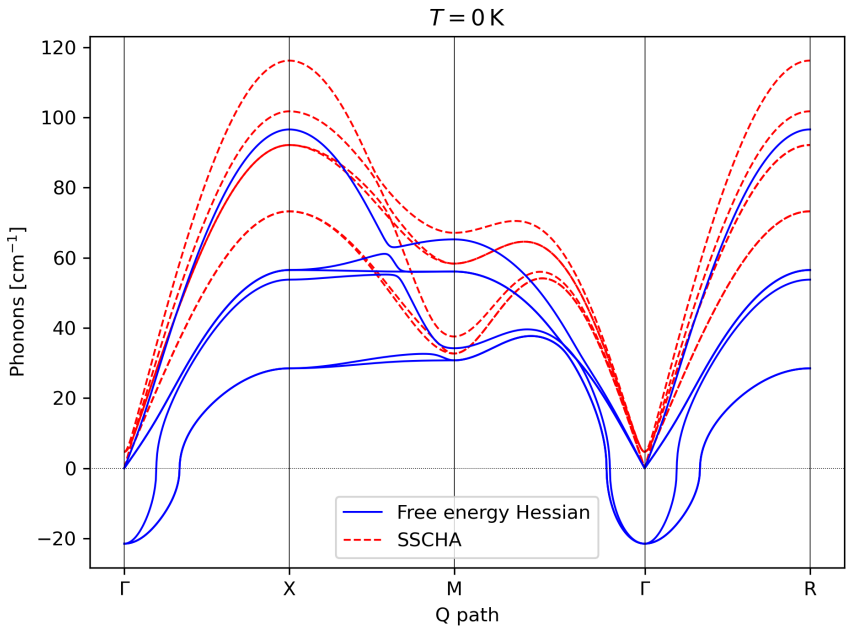

Figure 3.4: SSCHA vs Hessian phonon dispersion at T=0 K.

An instability is observed in the optical mode at the Γ point, indicating the expected ferroelectric distortion. In contrast, no instability is found at the M point.

Question

The harmonic phonon spectrum exhibits instabilities at both the Γ and M points. Why does the instability at M disappear in the free-energy Hessian spectrum?

Answer

The harmonic approximation determines the stability of the crystal from the curvature of the potential-energy surface alone. The free-energy Hessian, on the other hand, includes the effects of quantum and thermal fluctuations through the free energy.
In particular, even at zero temperature, quantum fluctuations associated with the zero-point motion modify the free-energy curvature. In SnTe, these quantum effects are strong enough to stabilize the distortion pattern associated with the M-point instability, making the corresponding free-energy Hessian eigenvalue positive. The ferroelectric instability at Γ, however, remains unstable and therefore continues to drive the phase transition.

Before drawing quantitative conclusions, however, we must verify that the free-energy Hessian is converged with respect to the number of configurations used in its stochastic evaluation. Indeed, the bubble correction depends on the third-order force-constant tensor, $\overset{(3)}{\Phi}$, whose statistical estimation generally requires a larger ensemble than that needed for the SSCHA minimization itself.

A convergence study with respect to the ensemble size is therefore essential.

Exercise

Compute the optical eigenvalue of the free-energy Hessian dynamical matrix $D^{(F)}$ as a function of the number of configurations N_{conf} used in the stochastic evaluation of the Hessian.

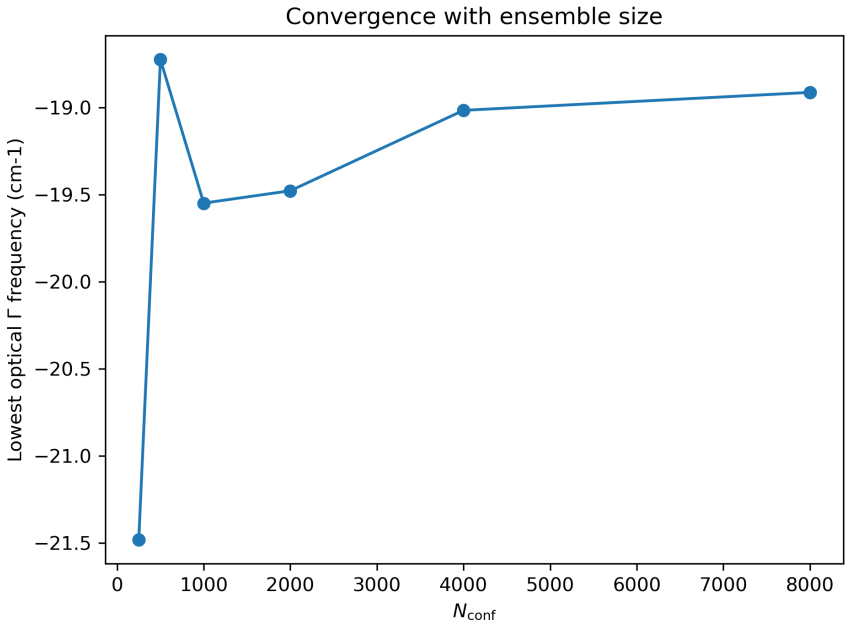

Figure 3.5: Soft-mode frequency as a function of the number of configurations.

In Monte Carlo calculations, the statistical uncertainty decreases approximately as

$$\frac{1}{\sqrt{N_{\mathrm{conf}}}}.$$

For this reason, convergence studies are often analyzed as a function of $1/\sqrt{N_{\mathrm{conf}}}$.

Exercise

Replot the soft-mode frequency as a function of $1/\sqrt{N_{\mathrm{conf}}}$.

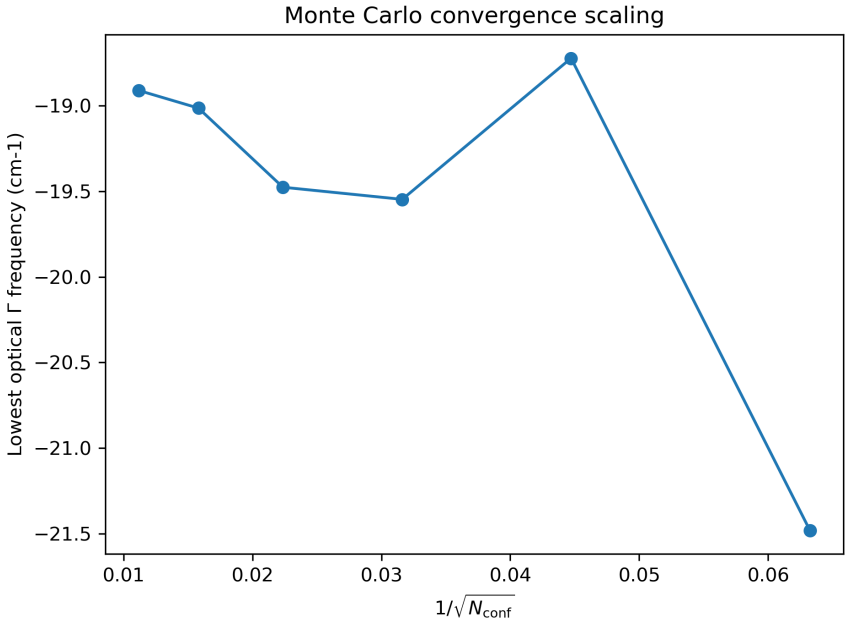

Figure 3.6: Soft-mode frequency as a function of the inverse square root of the number of configurations.

The convergence test shows that increasing the ensemble size beyond approximately 1000 configurations changes the estimated frequency by less than about $1\,\mathrm{cm}^{-1}$. For most practical purposes, further reducing the stochastic error is of limited value, as other sources of uncertainty—such as the choice of pseudopotentials, exchange-correlation functional, supercell size, or the accuracy of the underlying force field—typically dominate the overall error budget.

Nevertheless, since the toy-model calculations are computationally inexpensive, we can afford to use larger ensembles. In the following, we therefore adopt $N_{\mathrm{conf}} = 4000$, which further reduces the statistical uncertainty of the free-energy Hessian, especially in the vicinity of the phase transition.

Once the number of configurations required to obtain a converged free-energy Hessian has been determined, we can perform a systematic study as a function of temperature.

As a first step, we compute the phonon dispersion associated with the free-energy Hessian at several temperatures.

Exercise

Compute the phonon dispersion associated with the free-energy Hessian at several temperatures and estimate the temperature range in which the phase transition occurs.

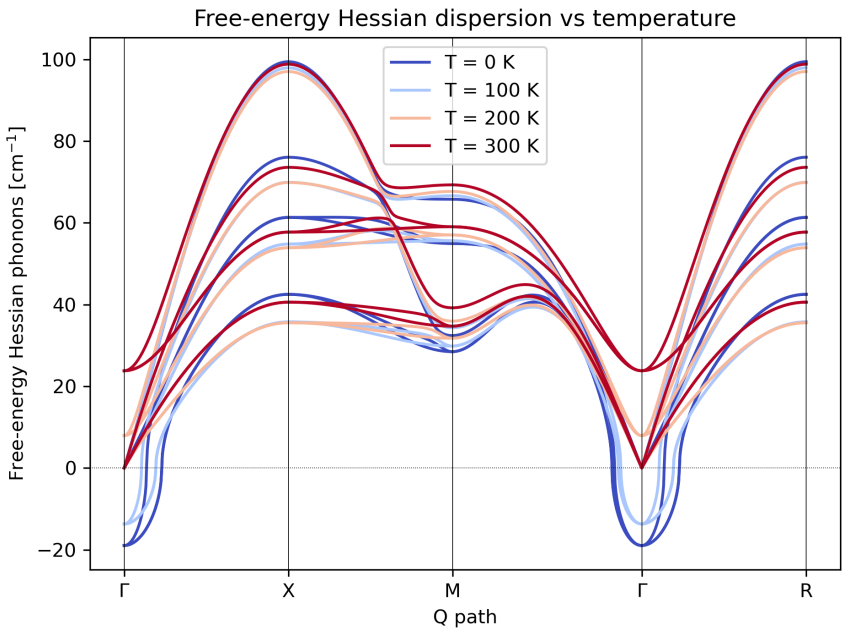

Figure 3.7: Temperature evolution of the phonon dispersion associated with the free-energy Hessian.

As the temperature decreases, the optical mode at the Γ point progressively softens and eventually becomes unstable. From these dispersions, it is already possible to obtain a rough estimate of the critical temperature, which appears to lie between 100 and 200 K.

To determine the transition temperature more accurately, it is convenient to focus on the soft optical mode at the Γ point and follow its frequency as a function of temperature.

Exercise

Compute the frequency of the soft optical mode at Γ as a function of temperature using both the SSCHA dynamical matrix and the free-energy Hessian dynamical matrix. Compare the temperature evolution of the two quantities.

The figure clearly highlights the different physical meaning of the two quantities. While the frequency obtained from the free-energy Hessian eventually becomes imaginary, signaling a structural instability, the SSCHA frequency remains real at all temperatures. This is expected, since the SSCHA auxiliary dynamical matrix is positive definite by construction and therefore cannot directly exhibit unstable modes.

As the temperature decreases, the free-energy Hessian mode softens continuously and eventually crosses zero. The corresponding temperature identifies the onset of the ferroelectric instability.

Question

What temperature dependence is expected for the soft-mode frequency close to a second-order phase transition?

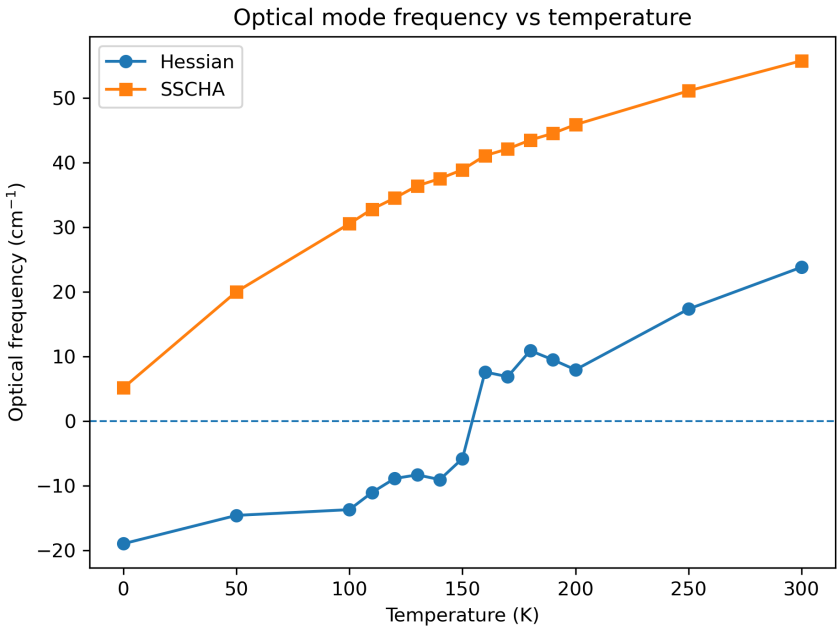

Figure 3.8: Temperature dependence of the soft optical mode at Γ.

Answer

Within a mean-field description of a second-order phase transition, the soft-mode frequency is expected to satisfy

$$\omega^2(T) \propto T - T_c.$$

Equivalently,

$$\omega(T) \propto \sqrt{T - T_c}.$$

Therefore, close to the transition, the frequency is expected to follow a square-root dependence on temperature.

Exercise

Fit the temperature dependence of the soft-mode frequency using the expected square-root behavior and estimate the critical temperature.

The square-root fit already provides a reasonable estimate of the critical temperature. However, extracting T_c from a non-linear fit can be sensitive to statistical fluctuations.

Since mean-field theory predicts a linear dependence of ω^2 on temperature, it is often more convenient to analyze the squared frequency instead.

Exercise

Plot the squared frequency of the soft mode obtained from the free-energy Hessian as a function of temperature. Perform a linear fit and estimate the critical temperature T_c.

The linear dependence predicted by mean-field theory is clearly observed. The temperature at which the

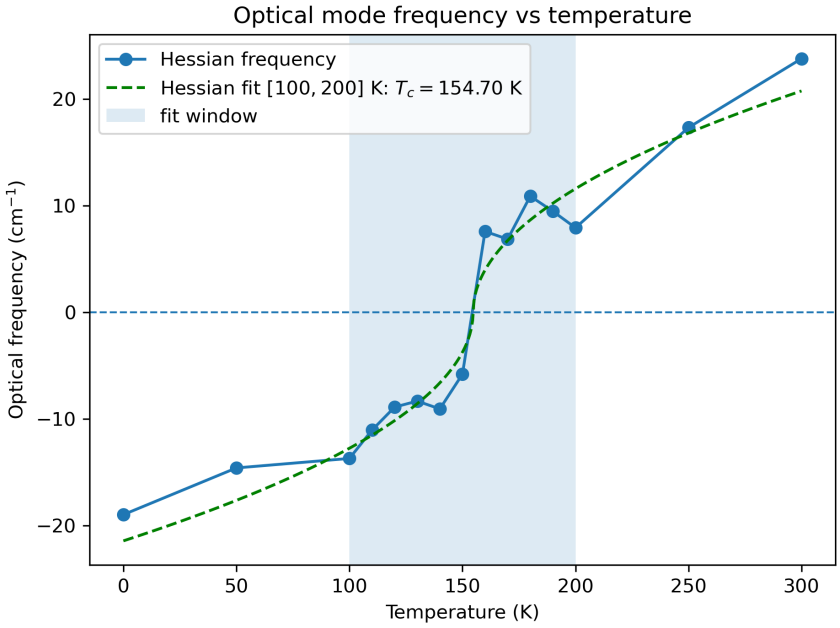

Figure 3.9: Temperature dependence of the soft-mode frequency together with a square-root fit.

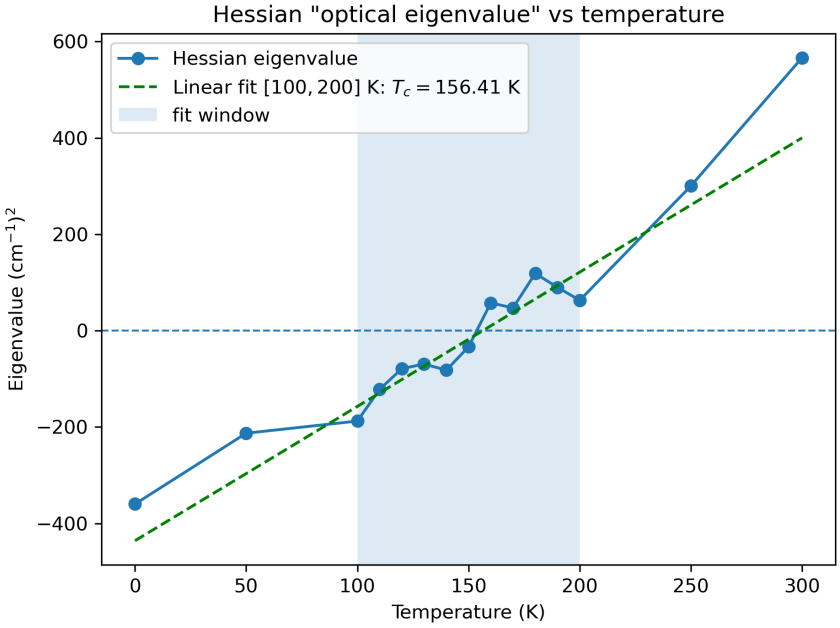

Figure 3.10: Squared soft-mode frequency as a function of temperature together with a linear fit.

fitted line crosses zero provides an estimate of the critical temperature.

The linear fit yields

$$T_c \simeq 150\,\mathrm{K}.$$

Chapter 4

Hands-on Session 3 - Spectral functions, Raman and infrared spectra with the Time-Dependent SSCHA

4.1 Setup

This tutorial is about the computation of dynamical properties of materials, focusing on Raman and IR spectroscopy. To run, it requires the `quippy-ase` force field, a converged SSCHA calculation on the system $CsPbI_3$, and the DFT calculation of Raman tensor and Born effective charges. You can find the all the materials inside the `Materials/tutorial_03` folder of the repository. Copy all these files in your working directory. Also, remember to adjust the relative path to the `GAP_1.xml` file for the force field with respect to your working directory in all the scripts.

To run the calculations, be sure to be inside the anaconda `sscha` environment. To enter in this environmnet, from the terminal run

```
conda activate sscha
```

This command needs to be run every time a new terminal is open. If you have not yet done so, once inside the `sscha` environment, install quippy-ase with the command

```
pip install quippy-ase
```

4.2 Theoretical background

The key quantity which is measured by experiments is the *dynamical response function* $\chi(\omega)$. The response function probes how the material responds to a time-dependent external perturbation. We can model any experiment as follows: the material is in equilibrium for $t < t_0$, then a perturbation is turned on at t_0, and we measure a property A at a later time t. The measured response is the convolution of the perturbation over all intervening times, weighted by the response function $\chi(t - t')$, which describes how perturbations propagate:

$$A(t) = A_0 + \int_{t_0}^{t} dt' \chi(t - t') F(t')$$

where $F(t')$ is the external time-dependent perturbation, and A_0 is the value of the property A in equilibrium. Alternatively, in Fourier space, the convolution becomes a simple product:

$$A(\omega) = A_0 + \chi(\omega) F(\omega)$$

This is very general, and it applies to any experiment. In this tutorial, we focus on IR and Raman spectroscopy. For the case of IR, the perturbation is the electric field of the light, which interacts with the material by generating an oscillating dipole. The response of the material is the *emitted* (or absorbed) electric field, which is related to how the dipole moment of the material oscillates in time. So the observable $A(t)$ is the dipole moment of the material, and the perturbation $F(t)$ is the amplitude of the electric field of the light.

The Kubo equation for the response function is

$$\chi(t) = \frac{i}{\hbar}\theta(t)\,\langle M(t)M(0)\rangle$$

where $\hat{M}(t)$ is the dipole moment operator in the Heisenberg picture, and $\langle\cdot\rangle$ is the quantum average at finite temperature. The Time-Dependent SCHA (TD-SCHA) extends the SSCHA framework to compute dynamical response functions. Here we will use the Lanczos algorithm to compute the response function in Fourier space, which is more efficient than computing it in time and then Fourier transforming it.

The first step is to convert the dipole operator in phonons creation and annihilation operators. To this aim, we can approximate the dipole moment as a linear function of the atomic displacements $\hat{u}$:

$$\hat{M}(t) = \left.\frac{\partial M}{\partial u}\right|_{u=0} \hat{u}(t) + O(u^2)$$

The derivative of the dipole moment with respect to the atomic displacements is precisely the Born effective charge:

$$Z^i_{\alpha\beta} = \frac{\partial M_\alpha}{\partial u^i_\beta} = \frac{\partial^2 \mathcal{E}}{\partial u^i_\beta \partial E_\alpha}$$

where E_α is the electric field in the α Cartesian direction, u^i_β is the displacement of atom i in the β Cartesian direction, and $\mathcal{E}$ is the total energy of the system (Born-Oppenheimer). Born effective charges $Z^i_{\alpha\beta}$ are quantities that can be computed via density functional perturbation theory (DFPT). We will provide two example scripts on how to perform such a calculation using quantum-espresso without going into the details.

Using this in the expression of the susceptibility, we obtain:

$$\chi_{\mathrm{IR}\alpha}(t) = \sum_{ij}\sum_{\beta\gamma} \frac{Z^i_{\alpha\beta} Z^j_{\alpha\gamma}}{\sqrt{m_i m_j}} \sqrt{m_i m_j}\left\langle \hat{u}^i_\beta(t)\hat{u}^j_\gamma(0)\right\rangle$$

The last term is the so-called phonon Green's function, which represents the propagation of a phonon created at time $t = 0$ and destroyed at time t.

$$G^{ij}_{\alpha\beta}(t) = \sqrt{m_i m_j}\left\langle \hat{u}^i_\alpha(t)\hat{u}^j_\beta(0)\right\rangle$$

Analogously, the Raman spectrum is related to the response of the material to two electric fields, which interact with the material by generating an oscillating polarizability. The observable $A(t)$ is the polarizability of the material, and the perturbation $F(t)$ is the amplitude of the two overlapping electric fields which interfere within the material. The Raman susceptibility can be written as:

$$\chi_{\mathrm{Raman}}(t) = \left\langle \hat{\alpha}_{\alpha\beta}(t)\hat{\alpha}_{\alpha\beta}(0)\right\rangle$$

where $\hat{\alpha}_{\alpha\beta}$ is the polarizability operator, which can be approximated as a linear function of the atomic displacements:

$$\hat{\alpha}_{\alpha\beta}(t) = \left.\frac{\partial \alpha_{\alpha\beta}}{\partial u^i_\gamma}\right|_{u=0} \hat{u}^i_\gamma(t) + O(u^2)$$

$$\Xi^i_{\alpha\beta\gamma} = \frac{\partial \alpha_{\alpha\beta}}{\partial u^i_\gamma} = \frac{\partial^3 \mathcal{E}}{\partial u^i_\gamma \partial E_\alpha \partial E_\beta}$$

where $\Xi^i_{\alpha\beta\gamma}$ is the Raman tensor, which is the third derivative of the total energy (Born-Oppenheimer) with respect to the atomic displacements and the two electric fields (incoming-outgoing). Also the Raman tensor

can be computed from DFPT, and the tutorials provides scripts to perform this calculation via quantum espresso.

Notably, the Raman requires two electric fields because it is a scattering, where incoming and outcoming radiation are different. The IR instead is an absorption/emission, where incoming and outcoming radiation are the same.

The Raman susceptibility is then:

$$\chi_{\text{Raman}}(t) = \left\langle \hat{\alpha}_{\alpha\beta}(t)\hat{\alpha}_{\alpha\beta}(0)\right\rangle = \sum_{ij}\sum_{\gamma\delta} \frac{\Xi^i_{\alpha\beta\gamma}\Xi^j_{\alpha\beta\delta}}{\sqrt{m_i m_j}} \sqrt{m_i m_j}\left\langle \hat{u}^i_\gamma(t)\hat{u}^j_\delta(0)\right\rangle$$

Also here, the last term is the phonon Green's function, which represents the propagation of a phonon created at time $t = 0$ and destroyed at time t. Therefore, to compute the IR and Raman spectra, we need to compute the phonon Green's function. Going in Fourier space, we get

$$\chi_{\text{IR}\,\alpha}(\omega) = \sum_{ij}\sum_{\beta\gamma} \frac{Z^i_{\alpha\beta}Z^j_{\alpha\gamma}}{\sqrt{m_i m_j}} \sqrt{m_i m_j} G^{ij}_{\beta\gamma}(\omega)$$

$$\chi_{\text{Raman}\,\alpha\beta}(\omega) = \sum_{ij}\sum_{\gamma\delta} \frac{\Xi^i_{\alpha\beta\gamma}\Xi^j_{\alpha\beta\delta}}{\sqrt{m_i m_j}} \sqrt{m_i m_j} G^{ij}_{\gamma\delta}(\omega)$$

4.2.1 The phonon Green's function

The phonon Green's function can be computed within the TD-SCHA. In general, we can write the Green's function as a non-interacting Green's function plus a self-energy:

$$G^{-1}(\omega) = G_0^{-1}(\omega) - \Pi(\omega)$$

where $G_0(\omega)$ is a non-interacting Green's function for *harmonic*-like phonons, while $\Pi(\omega)$ is the phonon self-energy, which depends on the frequency. In the TD-SCHA theory, the non-interacting Green's function is defined as the Green's function of the auxiliary SSCHA harmonic Hamiltonian.

$$G_0^{-1}(\omega) = I\omega^2 - D_{\text{SSCHA}}$$

where I is the identity matrix and D_{SSCHA} is the dynamical matrix of the SSCHA Hamiltonian.

The self-energy involves the third- and fourth-order interatomic force constants:

$$\Pi_{ab}(\omega) = -\frac{1}{2}\sum_{cd\mu\nu} \overset{(3)}{\Phi}_{acd}\Lambda^{cd}_{\mu\nu}(\omega)\left[1 + \frac{1}{2}\overset{(4)}{\Phi}\Lambda(\omega)\right]^{-1}\overset{(3)}{\Phi}_{\nu b}$$

This corresponds to the same expression derived for the Free energy Hessian; see Bianco et al., Physical Review B, 96, 014111, 2017. where $\Lambda^{cd}_{\mu\nu}(\omega)$ is the Fourier transform of the two-phonon propagator, which can be computed from the equilibrium non interacting Green's function $G_0(\omega)$ as:

$$\Lambda^{ijlm}_{\alpha\beta\gamma\delta}(t) = \sqrt{m_i m_j m_l m_m}\left\langle \hat{u}^i_\alpha(t)\hat{u}^j_\beta(t)\hat{u}^l_\gamma(0)\hat{u}^m_\delta(0)\right\rangle_0$$

Computing the full inversion of the self-energy for every value of the frequency is computationally extremely expensive. It is possible, with an efficient algorithm (Lanczos), to show that the Green's function can be obtained by inverting a special matrix, see Monacelli, Mauri, Physical Review B 103, 104305, 2021.

In this tutorial, we will use this Lanczos algorithm to compute the phonon Green's function and, consequently, the IR and Raman spectrum of cubic and tetragonal CsPbI3.

To run these calculations, we need the `tdscha` package (already installed in the virtual machine).

4.3 Tutorial

We need to first relax a complete SSCHA calculation, exactly as for the free energy hessian. The familiar `sscha_relax.py` script performs an automatic (fixed volume) sscha relaxation from the Harmonic dynamical matrix using 256 configurations. Since this calculation has been argument of the first tutorial, we will assume to already have the results.

Inside the `Materials` directory (subdirectory `tutorial_03`), we provide the final result of the SSCHA relaxation as `sscha_auxiliary_dyn_` files. The first step for a dynamical linear response calculation is to have a very well converged auxiliary dynamical matrix. For this purpose, it is useful to run an additional minimization with a higher number of configurations once the initial sscha calculation is done.

This is performed by the script `last_sscha_minim.py` which we will analyze in the next section.

```
import sys, os
import cellconstructor as CC, cellconstructor.Phonons
import sscha, sscha.Ensemble, sscha.SchaMinimizer

from quippy.potential import Potential


TEMPERATURE = 450 # K
NQIRR = 4
START_DYN = "sscha_auxiliary_dyn_"
POTENTIAL = "../../Materials/GAP_1.xml" # Replace with your relative path
N_CONFIGS = 1024 # You can also reduce to 512 if too slow
POP_ID = 100


def last_sscha_relax(temperature = TEMPERATURE):
    # Load the sscha dynamical matrix
    dyn = CC.Phonons.Phonons(START_DYN, NQIRR)

    # Load the interatomic Potential for CsPbI3
    calc = Potential("IP GAP", param_filename=POTENTIAL)

    # Generate the last esemble with N_CONFIGS random atomic samples
    ensemble = sscha.Ensemble.Ensemble(dyn, temperature)
    ensemble.generate(N_CONFIGS)

    # Compute the energies and forces of the atomic samples
    ensemble.compute_ensemble(calc)

    # Minimize the free eenrgy on the new bigger ensemble
    minim = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
    minim.set_minimization_step(0.02)
    minim.meaningful_factor = 0.01
    minim.run()

    # Save the final dynamical matrix and ensemble for further calculations
    # The POP_ID is a unique identifier for the ensemble
    minim.ensemble.save_bin("data", POP_ID)
    minim.dyn.save_qe("sscha_converged_dyn_")


last_sscha_relax()
```

This script performs a full SSCHA minimization starting from the auxiliary dynamical matrix obtained from the previous SSCHA relaxation, but using a bigger ensemble of 1024 configurations. The final ensemble is saved in binary format inside the directory `data`, with a unique ID of `100` (any other ensemble with that ID will be overwritten). The final SSCHA auxiliary dynamical matrix is saved in the file `sscha_converged_dyn_`. These are useful if you want to perform multiple linear response calculations, as they do not need to be recomputed.

Once we have the final converged auxiliary dynamical matrix, we can compute the phonon Green's function and the IR spectrum with the script `tdscha_run_ir.py`.

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

import tdscha, tdscha.DynamicalLanczos

import sys, os

TEMPERATURE = 450 # K
NQIRR = 4 # Irreducible q points of the dynamical matrix

# Info about the dynamical matrix and the ensemble
ORIGINAL_DYN = "sscha_auxiliary_dyn_"
FINAL_DYN = "sscha_converged_dyn_"
POP_ID = 100

def compute_ir():
    # Load the original ensemble
    dyn_original = CC.Phonons.Phonons(ORIGINAL_DYN, NQIRR)
    ensemble = sscha.Ensemble.Ensemble(dyn_original, TEMPERATURE)
    ensemble.load_bin("data", POP_ID)

    # Load the final dynamical matrix
    # After the highly converged free energy minimization
    final_dyn = CC.Phonons.Phonons(FINAL_DYN, NQIRR)

    # To prepare the IR or Raman, we need
    # IR : dielectric tensor and Born effective charges
    # Raman : Raman tensor
    # Load them from quantum espresso ph.x output
    final_dyn.ReadInfoFromESPRESSO("dielectric_calc.pho")

    # Update the ensemble weights on the converged dynamical matrix
    ensemble.update_weights(final_dyn, TEMPERATURE)

    # Initialize the TD-SCHA Lanczos algorithm
    lanczos = tdscha.DynamicalLanczos.Lanczos(ensemble, lo_to_split=None)
    lanczos.init()

    # Define which level of anharmonicity we want
    lanczos.ignore_v3 = False # Add bubble contribution if false
    lanczos.ignore_v4 = True # Add RPA resummation if false (a factor 2 slower in speed -
    ↪ no extra memory)
```

```
    # If both v3 and v4 are ignored (both true), we get the 'harmonic' spectrum
    # on the sscha auxiliary frequencies

    # Define the reponse function to observe
    # In this case IR with polarization along the x axis
    polarization = np.array([1.0, 0.0, 0.0])
    lanczos.prepare_ir(pol_vec = polarization)

    # Run the lanczos algorithm for 40 steps
    lanczos.run_FT(40)

    # Save the final result in binary
    lanczos.save_status("IR_x.npz")


compute_ir()
```

4.3.1 Deep analysis of the script

Loading the ensemble

Let's analyze the script `tdscha_run_ir.py` step by step inside the function `compute_ir()`: The first step is loading the final sscha ensemble and dynamical matrix, which is achieved by the following lines:

```
    dyn_original = CC.Phonons.Phonons(ORIGINAL_DYN, NQIRR)
    ensemble = sscha.Ensemble.Ensemble(dyn_original, TEMPERATURE)
    ensemble.load_bin("data", POP_ID)

    # Lets load the final converged dynamical matrix
    final_dyn = CC.Phonons.Phonons(FINAL_DYN, NQIRR)
```

Born effective charges and Raman tensors

Notably, we want to compute the IR response. For this reason, we need to tell the `tdscha` the relation between atomic displacements and the polarization. This is achieved by loading the Born effective charges and dielectric tensor from a Quantum ESPRESSO phonon calculation, which is done by the line:

```
    final_dyn.ReadInfoFromESPRESSO("dielectric_calc.pho")
```

The file `dielectric_calc.pho` is the output of a Quantum ESPRESSO DFPT calculation containing the Born effective charges and dielectric tensor. The input files used to generate this output are provided as `dielectric_calc.pwi` and `dielectric_calc.phi`. To write the `dielectric_calc.pwi` file, we first need to extract the structure from the final dynamical matrix, since the Born effective charges must be computed at the final converged centroid positions. This is performed by the file `extract_structure.py`, in particular by the lines:

```
    # Load the final converged dynamical matrix
    dyn = CC.Phonons.Phonons("sscha_converged_dyn_")

    # Save the structure in the espresso format for pw.x
    dyn.structure.save_scf("sscha_structure.scf")
```

The final structure looks like the following

```
CELL_PARAMETERS angstrom
```

```
6.2340501132548267  0.0000000000000000  0.0000000000000000
0.0000000000000000  6.2340501132548267  0.0000000000000000
0.0000000000000000  0.0000000000000000  6.2340501132548267

ATOMIC_POSITIONS angstrom
Cs    3.1170250566274138  3.1170250566274138  3.1170250566274138
Pb    0.0000000000000002  0.0000000000000002  0.0000000000000002
I    3.1170250566274138  0.0000000000000002  0.0000000000000002
I    0.0000000000000002  0.0000000000000002  3.1170250566274138
I    0.0000000000000002  3.1170250566274138  0.0000000000000002
```

This file contains the cell parameters (rows of the cell matrix) in Angstrom and the cartesian coordinates of all the atoms in the primitive cell (also in Angstrom). In this format, it can be pasted on the bottom of a Quantum ESPRESSO input file.

The two espresso inputs can be run with the following commands (**No need to do it now, it may take time, we already provide the final output files**):

```
pw.x -i dielectric_calc.pwi > dielectric_calc.pwo
ph.x -i dielectric_calc.phi > dielectric_calc.pho
```

NOTE: Quantum-Espresso may restrict which pseudopotential or exchange-correlation potential could be used for the calculation of Born effective charges or Raman tensor. In particular, the latter can be computed at current stage only with norm-conserving pseudo potentials and LDA exchange correlation.

Once the effective charges, Raman Tensor and Dielectric Tensor have been computed, they can be loaded in the final dynamical matrix with the method `ReadInfoFromESPRESSO`, which is used in the line:

```
    final_dyn.ReadInfoFromESPRESSO("dielectric_calc.pho")
```

This is important if we want to compute IR or Raman response, as the response functions are computed with the effective charges and Raman tensor. We can skip this if, instead, we want just the displacement-displacement Green's function (e.g. to compute the free energy Hessian).

Update the ensemble

Next, we update the ensemble so that it reflects the final dynamical matrix, which is done by the line:

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

This is the reweighting, and it changes the weight of each stochastic configuration inside the ensemble so that, when we compute the average, we are computing the average with respect to the `final_dyn` rather than the dynamical matrix used to generate the ensemble. The new weights are computed as:

$$\rho_i = \frac{\rho_{\mathcal{R}_1,\Phi_1}(\mathbf{R}_i)}{\rho_{\mathcal{R}^{(0)},\Phi^{(0)}}}$$

Initialize the Lanczos algorithm for dynamical linear response

Next, we need to prepare the grounds for the dynamical linear-response calculation. This is performed using the Lanczos algorithm. The algorithm is initialized as follows:

```
    # Initialize the TD-SCHA Lanczos algorithm
    lanczos = tdscha.DynamicalLanczos.Lanczos(ensemble, lo_to_split=None)
    lanczos.init()

    # Let us define which level of anharmonicity we want
```

```
lanczos.ignore_v3 = False # Add bubble contribution if false
lanczos.ignore_v4 = True # Add RPA resummation if false
```

The flag `lo_to_split` can be used with a direction for a q vector in which the q-mesh will be offsetted. In this way, phonons at Γ will have energies and polarization vectors determined by the q vector exchanged between light and phonons. For now, we ignore `lo_to_split` setting it to `None`.

The `ignore_v3` and `ignore_v4` flags determine which approximation level we use to compute the phonon spectra. By setting both to False, we compute the full TD-SCHA response function without further approximations. Setting `ignore_v4` to True is equivalent to setting `include_v4` to False in the Free energy Hessian. However, thanks to how the Lanczos algorithm is formulated, including the full RPA resummation with the 4-phonon scattering vertices does not increase the computational cost of the algorithm. In particular, the cost only increases by a factor of 2, without any consequences on the memory. Here we ignore it for speed, but you can try setting it to False as well. This is also the reason why we can use the Lanczos algorithm from the TD-SCHA to compute the free energy Hessian accounting for the 4-phonon scatterings even in relatively large supercells without any memory issue.

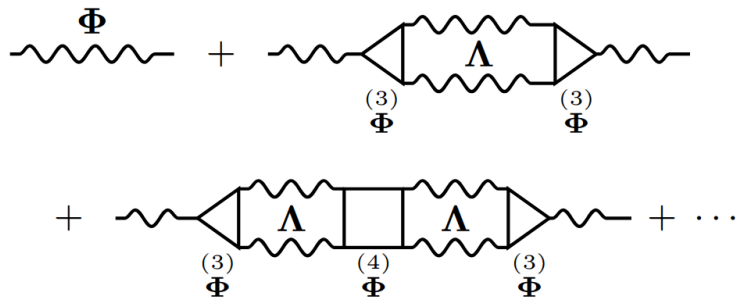

Figure 4.1: Diagrammatic series of perturbation present in the self-energy. The first term is the bare auxiliary harmonic propagation, the 3-phonon scattering vertex $\overset{(3)}{\Phi}$ enable the bubble diagram (the second term) while the 4-phonon scattering $\overset{(4)}{\Phi}$ enables all the chained diagrams obtained from the RPA like Dyson equation.

Define the perturbation

Then, we need to specify which response function we want to compute. In this case IR response, with polarization along the x cartesian axis:

```
# Define the reponse function to observe
# In this case IR with polarization along the x axis
polarization = np.array([1.0, 0.0, 0.0])
lanczos.prepare_ir(pol_vec = polarization)
```

This call needs to know which atoms are displaced and by how much when we perturb the system with an electric field along the x axis. This information is contained in the effective charges, which are read from the `dielectric_calc.pho` file, therefore this call will crash if no effective charge is defined on the dynamical matrix. If we want to run the Raman, we would use the `prepare_raman` function instead, which needs the Raman tensor instead of the effective charges. Indeed, for the Raman, we need to pass both the input and output polarization vector for the electric field (scattering). If we wanted the displacement-displacement Green's function, there is no need to preload the effective charges or the Raman tensor. In this case, we just need to call `prepare_mode` passing the index of the phonon mode we want to perturb. The index of the phonon mode is ordered from 0 (the lowest frequency mode in the supercell) to 3N-3 (the highest frequency mode in the supercell), excluding the 3 acoustic modes at Gamma.

Run the algorithm

Finally, we can run the Lanczos algorithm.

```
# Run the lanczos algorithm for 40 steps
lanczos.run_FT(40)
```

The Lanczos algorithm is iterative. We get to decide how many iterations. Feel free to reduce this to 20-30 iterations if it takes too much time. Each iteration computes the next Lanczos vector and the next element of the tridiagonal matrix, which is used to compute the response function. The more iterations we do, the better the resolution of the spectra. The run command takes multiples optional arguments. Let us see the API documentation for the `run_FT` method to see what we can do:

```
run_FT(n_iter, save_dir = None, save_each = 5, verbose = True, n_rep_orth = 0,
↪ n_ortho = 10, flush_output = True, debug = False, prefix = "LANCZOS", run_simm =
↪ None, optimized = False)
```

Among these options, worth mentioning are `save_dir`, `save_each` and `prefix`. The `save_dir`, if set to something other than `None`, is the directory in which the intermediate status of the Lanczos algorithm is saved. This can be used to resume a previous calculation, or to check the convergence of the spectra with the number of iterations before we reached the maximum number of iterations. `save_each` is the number of iterations after which the status is saved, and `prefix` is the prefix of the filename in which the status is saved. By default, the status is not saved, you can try to set `save_dir` to something like `'.'` to see that every 5 steps it will save a file with the name `LANCZOS_step_XX.npz`, where `XX` is the number of iterations.

We can also manually save at the end of the run with the method `save_status` of the Lanczos object, which takes as argument the filename in which the status is saved.

```
# Save the final result in binary
lanczos.save_status("IR_x.npz")
```

If we run the algorithm, we will get the final spectra in the file `IR_x.npz`.

4.3.2 Extract the green's function and plot

The TD-SCHA provides automatically a very simple way to plot the spectra with the command-line utility `tdscha-plot-data`

```
tdscha-plot-data IR_x.npz 0 200 0.5
```

The first argument is the file containing the Lanczos results. The next two arguments set the frequency range (in cm^{-1}), here from 0 to 200. The last argument is the smearing (in cm^{-1}) used to broaden the spectra. This command-line provides a quick way to plot the spectra. However, if you want more control, you can compute the spectrum directly in Python.

The example script that plots the spectrum is `plot_spectrum.py`.

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

DATA = "IR_x.npz"
W_START = 0 # cm-1
W_END = 200 # cm-1
TERMINATOR = True
SMEARING = 0.5 # cm-1
```

```
def plot_spectrum():
    # Load the final lanczos results
    lanczos = tdscha.DynamicalLanczos.Lanczos()
    lanczos.load_status(DATA)

    # Extract the Green function
    w_array = np.linspace(W_START, W_END, 2000)
    w_ry = w_array / CC.Units.RY_TO_CM
    smearing_ry = SMEARING / CC.Units.RY_TO_CM
    green_function = lanczos.get_green_function_continued_fraction(w_ry,
        smearing=smearing_ry,
        use_terminator = TERMINATOR)

    # The IR spectrum is proportional to - Im (G(w))
    ir_spectrum = -np.imag(green_function)

    # Plot the IR spectrum
    plt.plot(w_array, ir_spectrum)
    plt.xlabel("Frequency [cm-1]")
    plt.ylabel("- Im (G)")
    plt.title("IR spectrum - polarization x")
    plt.tight_layout()
    plt.savefig("ir_spectrum.png")
    plt.show()


plot_spectrum()
```

The final result is plotted in fig. 4.2.

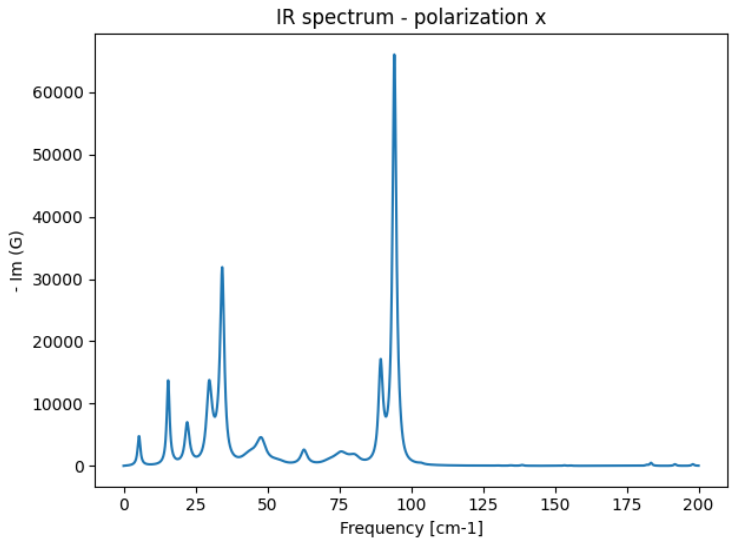

Figure 4.2: IR Spectrum obtained with the `plot_spectrum.py`.

Analysis of the plot script - Extracting the Green's Function

In particular, we first load the Lanczos algorithm status:

```
# Load the final lanczos results
lanczos = tdscha.DynamicalLanczos.Lanczos()
lanczos.load_status("IR_x.npz")
```

We then need to compute the Green's function from the Lanczos coefficient. The Lanczos algorithm finds an orthonormal basis in which the inverse-response function is a tridiagonal matrix $\mathcal{T}$

$$\mathcal{T} = \begin{pmatrix} a_1 & b_1 & 0 & 0 & ... \\ b_1 & a_2 & b_2 & 0 & ... \\ 0 & b_2 & a_3 & b_3 & ... \\ 0 & 0 & b_3 & a_4 & ... \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

where the first element of the basis is the perturbation vector. Therefore, the green function is actually the (1,1) element of the inverse of the tridiagonal matrix:

$$G(\omega + i\eta) = \left[\mathcal{T} - \mathcal{I}(\omega + i\eta)^2\right]^{-1}_{1,1}$$

where $\mathcal{I}$ is the identity matrix. Thanks to the many zeros in the tridiagonal matrix, the inverse of the first element is very easy to compute, and correspond to the following continued fraction:

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

The function of the python script `get_green_function_continued_fraction` uses the `a_i`, `b_i` saved inside the `lanczos` object (computed at each iterations) to compute the green function in the frequencies provided in input:

```
green_function = lanczos.get_green_function_continued_fraction(w_ry,
smearing=smearing_ry,
                                                               use_terminator =
TERMINATOR)
```

This function takes as input the frequency `\omega` (in Ry units), the smearing `\eta`, and whether to use a *terminator*. The terminator is a trick to reach the `N\to\infty` limit (where `N` is the number of iterations). Empirically, we see that after a certain number of iterations, the coefficients `a_i` and `b_i` oscillate around a specific value. In this case, we run 40 iterations of the Lanczos algorithm, so we will have $a_1, \cdots, a_{40}$ and $b_1, \cdots, b_{40}$. To make the plot smoother, we can extrapolate all the values of a_i and b_i for `i>N` with the average value of the coefficient. The infinite continued fraction can be solved analytically:

$$G_\infty(z) = \frac{1}{a_\infty - z^2 - b_\infty G_\infty(z)}$$
$$G_\infty(z)\left(a_\infty - z^2 - b_\infty G_\infty(z)\right) = 1$$

By solving this equation, we can replace the last fraction with the `G_\infty(z)`, simulating infinite iterations. Setting `use_terminator` to True truncates the continued fraction by replacing the tail with the asymptotic limit $G_\infty(z)$.

Exercise:

Compute the IR spectrum using the three different approximations: harmonic (SSCHA auxiliary frequencies), the bubble approximation (3-phonon only), and the full TD-SCHA (including 4-phonon scattering). Compare the results.

Question:

Is this calculation enough, or do we also need the response function for other perturbations, like y and z? Will something change in the spectrum if we do? Why?

Exercise:

Plot the IR data at various smearings and as a function of the number of steps (10, 20, 30, 40). How does the signal change with smearing and the number of steps? Try plotting with and without the terminator and see the differences.

Exercise:

Compute the free energy Hessian using the bubble approximation and the green function using a perturbation along that mode. Compare the two results.

4.4 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 sample interacts with two light sources: the incoming electromagnetic radiation and the outgoing one. The outgoing 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 Anti-Stokes. Since the outgoing radiation has higher energy than the incoming one in the Anti-Stokes, it is generated only by existing (thermally excited) phonons inside the sample. As a result, the Anti-Stokes signal has a lower intensity than the Stokes.

Using the relation between polarizability and atomic displacements, the Raman intensity becomes:

$$I(\omega) \propto \sum_{ab} \frac{\Xi_{xya}\Xi_{xyb}}{\sqrt{m_a m_b}} \left[-\mathrm{Im}G_{ab}(\omega)\right](n(\omega)+1)$$

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

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

```
# 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 polarizations of the light: the incoming radiation and the outgoing radiation.

Indeed, we need the Raman tensor Ξ_{xya} defined within the dynamical matrix.

For the cubic phase, Ξ_{abc} is zero by symmetry, therefore there is no Raman active mode. To compute the Raman response, we need a lower symmetry phase.

In this case, we take the tetragonal (β) phase of $CsPbI_3$. We provide inside the `Materials` directory an already performed SSCHA relaxation at 300 K (`sscha_auxiliary_tetra_` files)

Also in this case, we need to perform a new final relaxation. This final relaxation can be performed with the `last_sscha_minim_tetra.py`, which is very similar to the script we employed for the cubic phase

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

from quippy.potential import Potential


TEMPERATURE = 300 # K
NQIRR = 3
START_DYN = "sscha_auxiliary_tetra_"
POTENTIAL = "../../Materials/GAP_1.xml"
N_CONFIGS = 1024
POP_ID = 200


def last_sscha_relax(temperature = TEMPERATURE):
    # Load the sscha dynamical matrix
    dyn = CC.Phonons.Phonons(START_DYN, NQIRR)

    # Load the interatomic Potential for CsPbI3
    calc = Potential("IP GAP", param_filename=POTENTIAL)

    # Generate the last esemble
    ensemble = sscha.Ensemble.Ensemble(dyn, temperature)
    ensemble.generate(N_CONFIGS)

    # Compute the energies and forces with the GAP potential
    ensemble.compute_ensemble(calc)

    # Compute a full sscha minimization on the new bigger ensemble
    minim = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
    minim.set_minimization_step(0.02)
    minim.meaningful_factor = 0.01
    minim.run()

    # Save the final dynamical matrix and ensemble for further calculations
    minim.ensemble.save_bin("data", POP_ID)
    minim.dyn.save_qe("sscha_converged_tetra_dyn_")


last_sscha_relax()
```

This script is very similar to the previous one, the main differences are: 1. The number of irreducible q-points NQIRR, which is 3 instead of 4 (we have a 2x2x1 supercell, so less q points in total) 2. The temperature is set to 300 K instead of 450 K 3. The name of the dynamical matrix is obviously different 4. We save the ensemble in population 200 to avoid overwriting the cubic one in population 100.

Once this script runs, we get the new ensemble and the final highly converged dynamical matrix. At this point, we need to compute the Raman tensor, using DFPT (e.g. using quantum espresso). The Raman tensor expresses how the polarizability changes when we move each atom. CellConstructor can load it from the quantum espresso `ph.x` output, like the Born effective charges.

The script to compute the Raman is:

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

import tdscha, tdscha.DynamicalLanczos

import sys, os

TEMPERATURE = 300 # K
NQIRR = 3 # Irreducible q points of the dynamical matrix

# Info about the dynamical matrix and the ensemble
ORIGINAL_DYN = "sscha_auxiliary_tetra_"
FINAL_DYN = "sscha_converged_tetra_dyn_"
POP_ID = 200

def compute_raman():
    # Load the original ensemble
    dyn_original = CC.Phonons.Phonons(ORIGINAL_DYN, NQIRR)
    ensemble = sscha.Ensemble.Ensemble(dyn_original, TEMPERATURE)
    ensemble.load_bin("data", POP_ID)

    # Lets load the final converged dynamical matrix
    final_dyn = CC.Phonons.Phonons(FINAL_DYN, NQIRR)

    # To prepare the Raman, we need to load the Raman tensor
    # Load them from quantum espresso ph.x output
    final_dyn.ReadInfoFromESPRESSO("dielectric_calc_tetra.pho")

    # Update the ensemble weights on the converged dynamical matrix
    ensemble.update_weights(final_dyn, TEMPERATURE)

    # Initialize the TD-SCHA Lanczos algorithm
    lanczos = tdscha.DynamicalLanczos.Lanczos(ensemble, lo_to_split = None)
    lanczos.init()

    # Let us define which level of anharmonicity we want
    lanczos.ignore_v3 = False # Add bubble contribution if false
    lanczos.ignore_v4 = True # Add RPA resummation if false (a factor 2 slower in speed -
    ↪ no extra memory)

    # If both v3 and v4 are ignored (both true), we get the 'harmonic' spectrum
    # on the sscha auxiliary frequencies

    # Define the reponse function to observe
    # In this case IR with polarization along the x axis
    polarization_in = np.array([1.0, 0.0, 0.0])
    polarization_out = np.array([1.0, 0.0, 0.0])
```

```
    lanczos.prepare_raman(pol_vec_in = polarization_in,
                          pol_vec_out = polarization_out
                          )

    # Run the lanczos algorithm for 40 steps
    lanczos.run_FT(40)

    # Save the final result in binary
    lanczos.save_status("Raman_xx.npz")

if __name__ == "__main__":
    # Change the working directory to the one containing this script
    os.chdir(os.path.dirname(os.path.abspath(__file__)))
    compute_raman()
```

Also this script is very similar to the IR one, with a few main differences:

1. We load the data of the tetragonal phase.
2. The Raman initialization

Let us focus on the Raman initialization:

```
    # Define the reponse function to observe
    # In this case IR with polarization along the x axis
    polarization_in = np.array([1.0, 0.0, 0.0])
    polarization_out = np.array([1.0, 0.0, 0.0])
    lanczos.prepare_raman(pol_vec_in = polarization_in,
                          pol_vec_out = polarization_out)
```

In this case, we need an incoming field polarization and an outgoing field polarization. The simultaneous presence of incoming and outgoing radiation generates a force on the atoms, which excites phonons inside the material.

The result is shown in fig. 4.3.

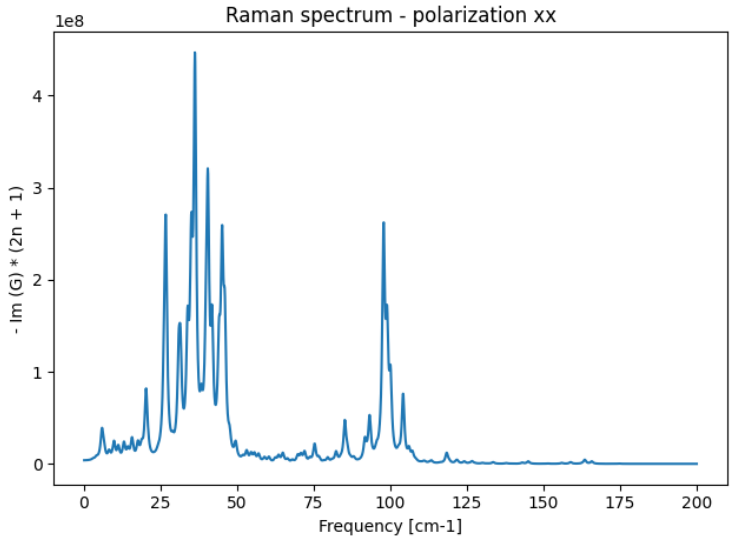

Figure 4.3: Raman spectrum with 200 iterations of the Lanczos algorithm along the x polarization for both incoming and outgoing radiation. This plot includes the Stokes scattering facctor $1 + n(\omega)$.

Exercise:

Try to run the Raman off-diagonal like x-y, or the zz. Are the photo-excited modes the same?

The solution for the unpolarized Raman is shown in fig. 4.4.

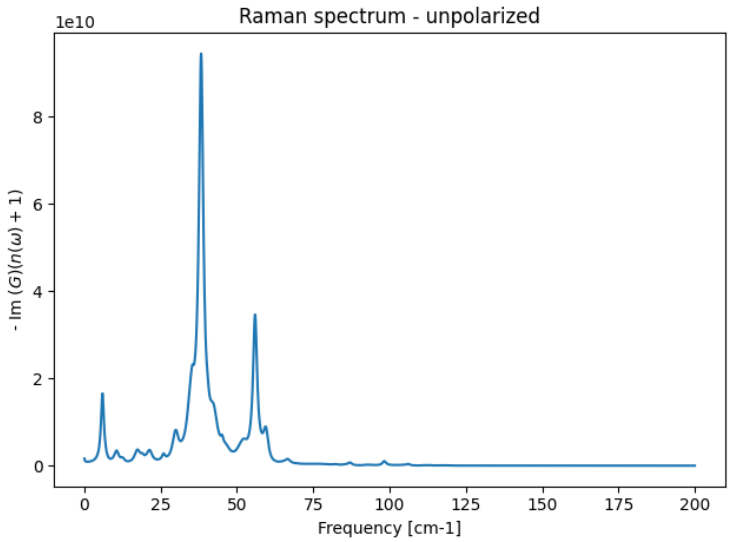

Figure 4.4: Unpolarized Raman. The scattering factor $n(\omega) + 1$ kills high frequency modes, which have a lower thermal population than low-frequency modes.

4.4.1 Unpolarized Raman

Often, experiments do not distinguish between laser polarization or sample orientation (e.g., the sample is a powder). In these cases, we need to average across all possible crystal orientation, or light polarization. This would require a Monte Carlo sampling, which could take a lot of computational time to compute the response function. An alternative approach is to directly compute the unpolarized Raman. In particular, the unpolarized Raman signal can be computed from the so-called *invariants*, where the perturbations in the polarizations are:

$$I_A = \frac{1}{9}(xx + yy + zz)^2$$

$$I_{B_1} = (xx - yy)^2/2$$

$$I_{B_2} = (xx - zz)^2/2$$

$$I_{B_3} = (yy - zz)^2/2$$

$$I_{B_4} = 3(xy)^2$$

$$I_{B_5} = 3(yz)^2$$

$$I_{B_6} = 3(xz)^2$$

The total intensity of unpolarized Raman is:

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

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} components are computed using index i. For example, to compute I_{B_5}:

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

To obtain the total spectrum, you need to add the factor $n(\omega) + 1$ and sum all these perturbations 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.

This could be run inside a loop, e.g.

```
for i in range(7):
    lanczos.prepare_raman(unpolarized=i)

    # Run the lanczos algorithm for 40 steps
    lanczos.run_FT(50)

    # Save the final result in binary
    lanczos.save_status(f"Raman_unpol{i}.npz")

    # Reset the calculation, ready for a new perturbation
    lanczos.reset()
```

Or, if the calculation is heavy, it could be run in parallel by simply running simultaneously the different perturbations.

Exercise:

Compute the unpolarized Raman spectrum of $CsPbI_3$ and plot the results.

4.5 The dielectric tensor

A real IR experiment measures the dielectric function. For normal propagation, the absorption is:

$$A_s(\omega) = -\text{Im}\sqrt{\varepsilon(\omega)}$$

While for the evanescent wave we have

$$A_e(\omega) = \text{Im}\frac{1}{\sqrt{\varepsilon(\omega)}}$$

The `tdscha` code provides the susceptibility $\chi(\omega)$ from the dipole-dipole response function, in Rydberg (Ry) atomic units. The relationship between susceptibility and dielectric constant is:

$$\varepsilon(\omega) = \varepsilon_\infty + \frac{4\pi}{V}\chi(\omega)$$

This relation holds in Hartree atomic units. If χ is expressed in Rydberg atomic units (as is the case for the `tdscha` output), a factor of 2 arises from the Ha $\rightarrow$ Ry conversion, giving:

$$\varepsilon(\omega) = \varepsilon_\infty + \frac{8\pi}{V}\chi(\omega)$$

The value ε_∞ is obtained from DFPT (via a Quantum ESPRESSO phonon calculation). It is contained in the file `dielectric_calc.pho` and is accessible from the dynamical matrix after loading it:

```
dyn.ReadInfoFromESPRESSO("dielectric_calc.pho")
print("Dielectric Tensor:")
print(dyn.dielectric_tensor)
```

Pay attention to the volume: the `tdscha` stores the volume in Å^3, which must be converted to atomic units before the division. Moreover, the response function is computed on a 2$\times$2$\times$$2 supercell, so the volume must be scaled accordingly:

```
# Get the volume in the primitive cell (in Bohr^3)
volume = dyn.structure.get_volume() * CC.Units.A_TO_BOHR**3

# Obtain the volume of the supercell
volume *= np.prod(dyn.GetSupercell())
```

Exercise:

Compute the infrared absorption in the normal and evanescent wave of the cubic $CsPbI_3$. Plot them together with the spectral function and discuss the differences and why.
(the solution is inside `exercise_solution`, but try to solve it without looking at it).
The overall result is reported in fig. 4.5 .

Question:

What is the energy difference in the peak position between the spectral function and the evaneshent wave?

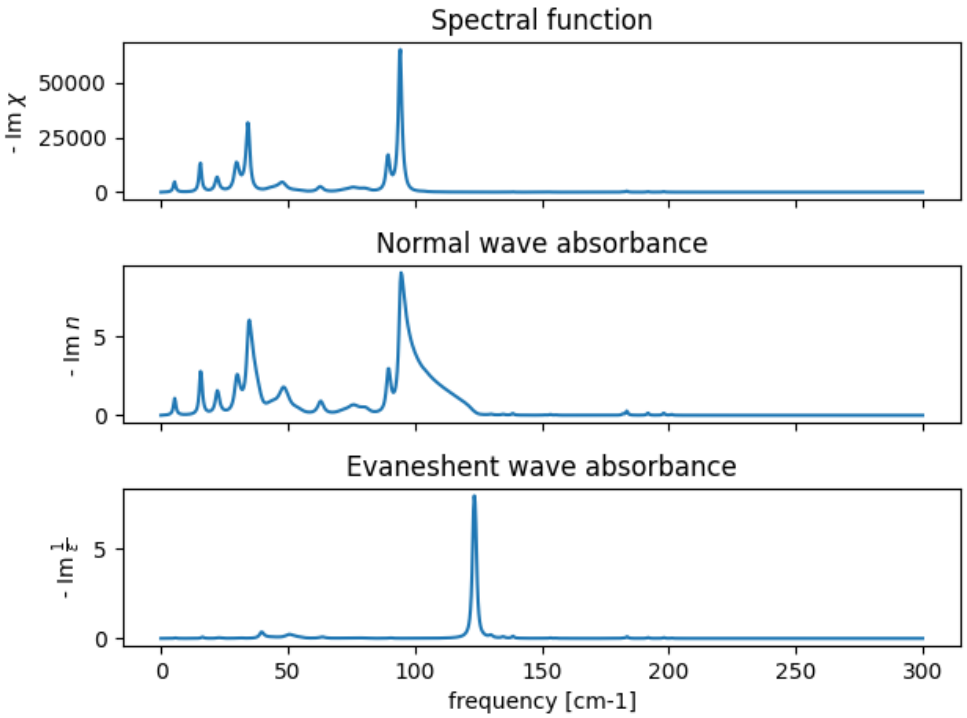

Figure 4.5: Infrared absorption in the normal and evanescent wave of the cubic $CsPbI_3$. The spectral function is also plotted for comparison.

Chapter 5

Hands-on Session 4: Thermal Conductivity Calculations with the SSCHA

In previous lessons, we saw how to calculate vibrational properties of materials using the stochastic self-consistent harmonic approximation (SSCHA). In this tutorial, we will use that knowledge to calculate the lattice thermal conductivity of materials.

To do this, we need the SSCHA dynamical matrices (**auxiliary dynamical matrices, not free-energy Hessians**) and the third-order force constants. These quantities allow us to calculate the harmonic properties, such as phonon frequencies and group velocities, and the anharmonic properties, such as phonon lifetimes and spectral functions. Together, these are the ingredients required for lattice thermal-conductivity calculations.

In this tutorial, we will:

- Use SSCHA dynamical matrices and third-order force constants to calculate lattice thermal conductivity with the Green-Kubo (GK) and single relaxation time approximation (SRTA) approaches.
- Learn how to converge lattice thermal-conductivity calculations.
- Manipulate `ThermalConductivity` objects to analyze the calculated transport properties.

The basic workflow is:

1. Relax the crystal structure and obtain SSCHA dynamical matrices in the temperature range of interest.
2. Use the relaxed SSCHA dynamical matrices to calculate the SSCHA free-energy Hessian, confirm dynamical stability, and obtain third-order force constants.
3. Calculate the lattice thermal conductivity and check its convergence with respect to:
   - the **q**-point grid used for the thermal-conductivity calculation;
   - the size of the SSCHA supercell;
   - the number of configurations used to calculate the third-order force constants;
   - the smearing parameter used in the phonon self-energy calculation, which is closely related to the size of the **q**-point grid when using an *adaptive* smearing scheme;
   - the density of frequency sampling controlled by the `ne` parameter when using the Green-Kubo method.

5.1 Lattice Thermal Conductivity of CsPbI3

We first calculate the second- and third-order force constants.

```
import warnings

import ase
import ase.io
import numpy as np
from ase import Atoms
from ase.eos import calculate_eos
from ase.filters import ExpCellFilter
from ase.optimize import QuasiNewton
from ase.units import kJ
from quippy.potential import Potential

# Import the CellConstructor and SSCHA modules used throughout this tutorial.
import cellconstructor as CC
import cellconstructor.Phonons
import cellconstructor.Structure
import sscha
import sscha.Ensemble
import sscha.Relax
import sscha.SchaMinimizer

# NumPy moved ComplexWarning in newer versions. This block keeps the script
# compatible with both older and newer NumPy releases.
try:
    ComplexWarning = np.exceptions.ComplexWarning
except AttributeError:
    ComplexWarning = np.ComplexWarning

warnings.filterwarnings("ignore", category=ComplexWarning)


def get_starting_dynamical_matrices(structure_filename, potential, supercell):
    """Relax the input structure and compute an initial dynamical matrix.

    Parameters
    ----------
    structure_filename : str
        Path to the input structure in Quantum ESPRESSO format.
    potential : quippy.potential.Potential
        Interatomic potential used to compute energies and forces.
    supercell : list[int]
        Supercell used for the finite-displacement phonon calculation.

    Returns
    -------
    dyn : cellconstructor.Phonons.Phonons
        Initial auxiliary dynamical matrix.
    bulk : float
        Static bulk modulus, used during variable-cell SSCHA relaxation.
    """
    # Read the input structure with CellConstructor.
    structure = CC.Structure.Structure()
    structure.read_scf(structure_filename)
```

```
    # Convert the CellConstructor structure to an ASE Atoms object so that
    # ASE optimizers and the QUIP/GAP potential can be used.
    atoms = Atoms(
        cell=structure.unit_cell,
        positions=structure.coords,
        symbols=structure.atoms,
        pbc=True,
    )
    atoms.set_calculator(potential)

    # Relax both atomic positions and cell vectors.
    ecf = ExpCellFilter(atoms)
    qn = QuasiNewton(ecf)
    qn.run(fmax=0.0005)

    # Convert the relaxed ASE structure back to CellConstructor format.
    structure = CC.Structure.Structure()
    structure.generate_from_ase_atoms(atoms, get_masses=True)

    # Compute an initial dynamical matrix by finite displacements.
    dyn = CC.Phonons.compute_phonons_finite_displacements(
        structure,
        potential,
        supercell=supercell,
    )
    dyn.Symmetrize()

    # Remove imaginary frequencies from the starting guess. This produces a
    # positive-definite auxiliary dynamical matrix for the SSCHA minimization.
    dyn.ForcePositiveDefinite()

    # Estimate the static bulk modulus from an equation-of-state fit. This is
    # needed by the variable-cell SSCHA relaxation below.
    eos = calculate_eos(atoms)
    v0, e0, B = eos.fit()
    bulk = B / kJ * 1.0e24

    return dyn, bulk


# Input files and calculation settings. Adjust these paths for your machine.
src = "/scratch/ddangic/sscha_school_2026/Materials/"
structure_name = "CsPbI3_cubic_phase.scf"
pot = Potential(param_filename=src + "GAP_1.xml", calc_args="local_gap_variance")
supercell = [2, 2, 2]

# Generate the initial auxiliary dynamical matrix and the bulk modulus.
dyn, bulk = get_starting_dynamical_matrices(src + structure_name, pot, supercell)

# SSCHA relaxation settings.
temperature = 300.0
nconf = 500
```

```
max_pop = 1000

# Generate the initial SSCHA ensemble from the starting dynamical matrix.
ensemble = sscha.Ensemble.Ensemble(
    dyn,
    T0=temperature,
    supercell=dyn.GetSupercell(),
)
ensemble.generate(N=nconf)

# Set up the SSCHA minimizer.
minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
minimizer.min_step_dyn = 0.001
minimizer.kong_liu_ratio = 0.5
minimizer.meaningful_factor = 0.001
minimizer.max_ka = 100000

# Perform the variable-cell SSCHA relaxation.
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",
)

# Recompute the SSCHA ensemble with an increasing number of configurations.
# This loop is used to test the convergence of the third-order force constants
# and the corresponding thermal-conductivity results.
for i in range(10):
    numconf = (i + 1) * 1000

    # Generate a new ensemble using the relaxed SSCHA dynamical matrix.
    new_ensemble = sscha.Ensemble.Ensemble(
        relax.minim.dyn,
        T0=temperature,
        supercell=relax.minim.dyn.GetSupercell(),
    )
    new_ensemble.generate(N=numconf)

    # Compute energies, forces, and stresses for the generated configurations.
    new_ensemble.compute_ensemble(
        pot,
        compute_stress=True,
        stress_numerical=False,
        cluster=None,
        verbose=True,
    )
```

```
# Minimize the free energy for the new ensemble. We keep the structure fixed
# here because the variable-cell relaxation was already performed above.
new_minimizer = sscha.SchaMinimizer.SSCHA_Minimizer(new_ensemble)
new_minimizer.minim_struct = False
new_minimizer.set_minimization_step(0.001)
new_minimizer.meaningful_factor = 0.001
new_minimizer.max_ka = 10000
new_minimizer.init()
new_minimizer.run()

# Save the converged auxiliary dynamical matrix in Quantum ESPRESSO format.
new_minimizer.dyn.save_qe("final_dyn_" + str(numconf) + "_")

# Reweight the ensemble with the updated dynamical matrix.
new_ensemble.update_weights(new_minimizer.dyn, temperature)

# Calculate the free-energy Hessian and the third-order tensor.
# Setting return_d3=True returns the third-order force constants needed for
# the thermal-conductivity calculation.
dyn_hessian, d3_tensor = new_ensemble.get_free_energy_hessian(
    include_v4=False,
    get_full_hessian=True,
    return_d3=True,
)

# Save the third-order tensor and the Hessian for later use.
np.save("d3_" + str(numconf) + ".npy", d3_tensor)
dyn_hessian.save_qe("hessian_dyn_" + str(numconf) + "_")
```

In this example, we used a $2 \times 2 \times 2$ supercell to calculate the second- and third-order force constants. The third-order force constants were calculated using an increasing number of configurations in order to test the convergence of the ensemble average.

The second- and third-order force constants must be converged with respect to both the supercell size and the number of configurations. Figure 1 shows the results for harmonic properties. The phonon band structure interpolated from the $2 \times 2 \times 2$ supercell is converged with respect to the number of configurations at around 2000 configurations. However, the results also need to be converged with respect to the supercell size, and they do not appear to be converged even for the $4 \times 4 \times 4$ supercell.

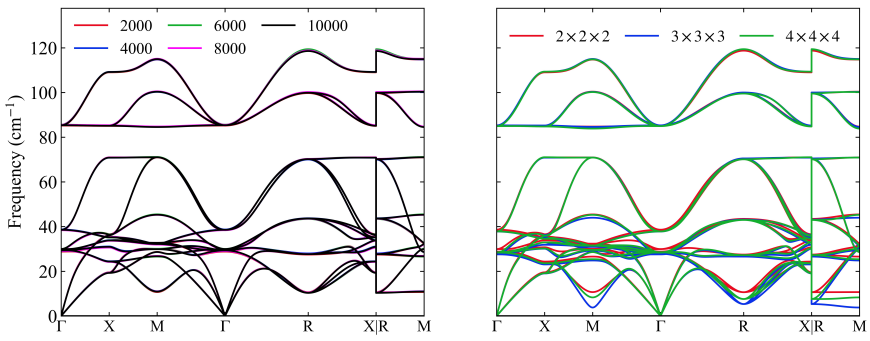

Figure 5.1: Convergence study of the phonon band structure with respect to supercell size and the number of configurations.

First, we calculate the lattice thermal conductivity using the single relaxation time approximation (**SRTA**).

In this approximation, the thermal-conductivity tensor is written as

```
\kappa^{xy}
=
\frac{1}{N V}
\sum_{\mathbf{q}, j}
v^{x}_{\mathbf{q}, j}
v^{y}_{\mathbf{q}, j}
c_{\mathbf{q}, j}
\tau_{\mathbf{q}, j}.
```

Here, N is the number of $\mathbf{q}$ points, V is the unit-cell volume, $v^{x}_{\mathbf{q},j}$ and $v^{y}_{\mathbf{q},j}$ are the phonon group-velocity components, $c_{\mathbf{q},j}$ is the mode heat capacity, and $\tau_{\mathbf{q},j}$ is the phonon lifetime for phonon mode j at wave vector $\mathbf{q}$. To calculate the thermal conductivity using force constants obtained with different numbers of configurations, we can use the following script:

```python
from __future__ import division, print_function

import time

import numpy as np

import cellconstructor as CC
import cellconstructor.ForceTensor
import cellconstructor.Phonons
import cellconstructor.ThermalConductivity

# Number of irreducible q points/files in the saved dynamical matrix.
nqirr = 4

# Useful conversion factors from CellConstructor.
SSCHA_TO_MS = cellconstructor.ThermalConductivity.SSCHA_TO_MS
RY_TO_THZ = cellconstructor.ThermalConductivity.SSCHA_TO_THZ

# Use the same grid for phonon properties and scattering processes in this
# example. In production calculations, these two grids should be converged
# separately.
meshnum = 10
mesh = [meshnum, meshnum, meshnum]

# Loop over force constants obtained from different ensemble sizes. This starts
# from 2000 configurations because range(1, 10) gives i = 1, ..., 9.
for i in range(1, 10):
    numconf = (i + 1) * 1000

    # Read the auxiliary SSCHA dynamical matrix saved in the previous step.
    dyn_prefix = "final_dyn_" + str(numconf) + "_"
    dyn = CC.Phonons.Phonons(dyn_prefix, nqirr)
    supercell = dyn.GetSupercell()

    # Create the third-order force-constant object using the same supercell
    # used to generate the SSCHA dynamical matrix.
    fc3 = CC.ForceTensor.Tensor3(
        dyn.structure,
        dyn.structure.generate_supercell(supercell),
```

```
    supercell,
)

# Load the third-order tensor and use it to initialize fc3.
d3_name = "d3_" + str(numconf) + ".npy"
d3 = np.load(d3_name)
fc3.SetupFromTensor(d3)

# Center the third-order force constants. This step is necessary
# in order to interpolate 3rd order force constants.
fc3 = CC.ThermalConductivity.centering_fc3(fc3)

# Define the ThermalConductivity object. The kpoint_grid is the grid on which
# phonon quantities are calculated on, while scattering_grid controls
# the sampling of scattering processes.
tc = CC.ThermalConductivity.ThermalConductivity(
    dyn,
    fc3,
    kpoint_grid=mesh,
    scattering_grid=mesh,
    smearing_scale=None,
    smearing_type="adaptive",
    cp_mode="quantum",
    off_diag=True,
)

# Calculate the thermal conductivity at 300 K.
temperatures = np.linspace(300, 300, 1, dtype=float)
start_time = time.time()

# Compute harmonic quantities such as phonon frequencies and group
# velocities, then write them to text files.
tc.setup_harmonic_properties()
tc.write_harmonic_properties_to_file()

# Calculate the lattice thermal conductivity in the SRTA approximation.
# Lifetimes are written to file because write_lifetimes=True.
tc.calculate_kappa(
    mode="SRTA",
    temperatures=temperatures,
    write_lifetimes=True,
    gauss_smearing=True,
    offdiag_mode="perturbative",
    kappa_filename="ConfThermal_conductivity_SRTA_" + str(numconf),
    lf_method="fortran-LA",
)
```

The most important parts of the script are:

- The mesh used to calculate phonon properties is defined to be the same as the mesh used to calculate scattering processes. This is controlled by the variable `mesh`. These grids do not have to be identical. In many cases, `scattering_grid` can be much smaller than `kpoint_grid`. **Both grids must be converged.**
- We use a smearing approach to satisfy energy-conservation laws. There are two methods: constant

and adaptive. For `smearing_type = "constant"`, the smearing value must be provided in **Ry** as an argument to `setup_harmonic_properties`. For adaptive smearing, as used here, the smearing value is different for different phonon modes and is based on the phonon density of states and **q**-point grid density. A global scaling factor can still be provided through `smearing_scale`; `smearing_scale = 1.0` often works well. **The smearing parameters must also be converged.**

- The `off_diag` variable controls whether the calculation includes what is often called *coherent transport*. This contribution can be important for strongly anharmonic materials with significant bunching of phonon modes.
- The function `calculate_kappa` performs most of the transport calculation. Its main options are:
  - `mode` defines the method used to calculate the lattice thermal conductivity. The options are **SRTA**, which corresponds to the Boltzmann transport equation in the single relaxation time approximation, and **GK** (Dangic et al.), which is the Green-Kubo method based on phonon spectral functions rather than phonon lifetimes. These two modes should give similar results in weakly anharmonic materials, but they can differ in strongly anharmonic materials.
  - `gauss_smearing` defines how energy conservation is treated in the self-energy calculation. If `True`, Gaussian functions are used. If `False`, Lorentzian functions are used. With Gaussian smearing, the real part of the self-energy is calculated using the Kramers-Kronig transformation.
  - `offdiag_mode` defines how coherent transport is calculated when `mode = "SRTA"`. The available options are **wigner** (Simoncelli et al.), **gk** (Isaeva et al.), and **perturbative** (Dangic et al.). If `mode = "GK"`, coherent transport is included naturally.
  - `lf_method` defines how phonon lifetimes are calculated when `mode = "SRTA"`. In most cases, keep the `fortran-` prefix and add either `LA` or `P`; these options should give similar results. Another option is `SC`, where phonon lifetimes are solved self-consistently and phonon lineshifts are included.
  - `ne` defines the number of frequency steps used when calculating phonon lineshapes. It is also important for `lf_method = "SC"`, because the self-consistent equation is solved on a frequency grid and the real and imaginary parts of the self-energy are linearly interpolated. Larger values are generally more accurate. **The results must be converged with respect to `ne`.**

In this example, we use the *SRTA* approximation, which is the weak-anharmonicity limit of the Green-Kubo equation. We use SSCHA auxiliary dynamical matrices because they form the basis for the dynamical extension of the SSCHA, derived from time-dependent SSCHA, which gives access to phonon lifetimes and lineshapes.

By setting `off_diag=True`, we include the coherent contribution to the lattice thermal conductivity. This contribution is important in strongly anharmonic materials, alloys, and amorphous solids. The calculation should take a few minutes, and the results are saved using the filename prefix specified by `kappa_filename`.

The thermal conductivity should also be calculated using different **q**-point grids. Figure 2 includes results obtained for a $3 \times 3 \times 3$ grid. These calculations do not appear to be converged at the $3 \times 3 \times 3$ level, so a larger supercell would be required for a production calculation.

However, third-order force constants often converge faster with respect to supercell size than second-order force constants. Therefore, one can sometimes combine dynamical matrices computed on a larger supercell with third-order force constants computed on a smaller supercell. This approach is illustrated in the script below:

```
from __future__ import print_function
from __future__ import division

import time

import numpy as np
import cellconstructor as CC
import cellconstructor.Phonons
import cellconstructor.ForceTensor
import cellconstructor.ThermalConductivity
```

```
SSCHA_TO_MS = cellconstructor.ThermalConductivity.SSCHA_TO_MS
RY_TO_THZ = cellconstructor.ThermalConductivity.SSCHA_TO_THZ

# Define the q-point grid used for both harmonic properties and scattering processes.
meshnum = 10
mesh = [meshnum, meshnum, meshnum]

# Define the temperature at which the thermal conductivity will be calculated.
temperatures = np.linspace(300, 300, 1, dtype=float)

# Read the dynamical matrix corresponding to the same supercell used to compute
# the third-order force constants.
dyn_prefix = "final_dyn_2x2x2_"
nqirr = 4
dyn_for_fc3 = CC.Phonons.Phonons(dyn_prefix, nqirr)

# Initialize and load the third-order force-constant tensor.
supercell = dyn_for_fc3.GetSupercell()
fc3 = CC.ForceTensor.Tensor3(
    dyn_for_fc3.structure,
    dyn_for_fc3.structure.generate_supercell(supercell),
    supercell,
)

d3_name = "d3_2x2x2.npy"
d3 = np.load(d3_name)
fc3.SetupFromTensor(d3)

# Center the third-order force constants.
fc3 = CC.ThermalConductivity.centering_fc3(fc3)

# Read a dynamical matrix computed using a larger supercell.
# This matrix will provide the harmonic properties used in the transport calculation.
dyn_prefix = "./final_dyn_4x4x4_"
nqirr = 10
dyn = CC.Phonons.Phonons(dyn_prefix, nqirr)

# Build the ThermalConductivity object using the large-supercell dynamical matrix
# and the small-supercell third-order force constants.
tc = CC.ThermalConductivity.ThermalConductivity(
    dyn,
    fc3,
    kpoint_grid=mesh,
    scattering_grid=mesh,
    smearing_scale=None,
    smearing_type="adaptive",
    cp_mode="quantum",
    off_diag=True,
)

start_time = time.time()
```

```
# Compute harmonic properties and write them to file.
tc.setup_harmonic_properties()
tc.write_harmonic_properties_to_file()

# Calculate the thermal conductivity in the SRTA approximation.
tc.calculate_kappa(
    mode="SRTA",
    temperatures=temperatures,
    write_lifetimes=True,
    gauss_smearing=True,
    offdiag_mode="perturbative",
    kappa_filename="Thermal_conductivity_SRTA",
    lf_method="fortran-LA",
)
```

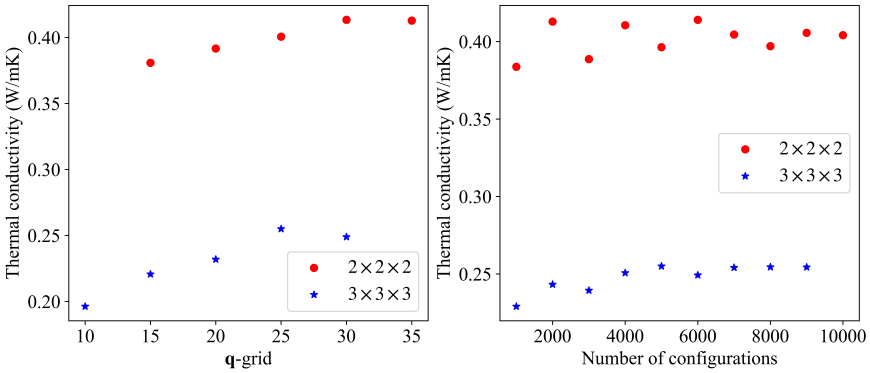

Figure 5.2: Convergence study of thermal conductivity with respect to q-point sampling, supercell size, and number of configurations.

Finally, we can calculate the lattice thermal conductivity with the Green-Kubo method:

```
        \kappa ^{xy} = \frac{2\pi\beta ^2k_{B}}{NV}\sum
↪   _{\mathbf{q},j,j'}v^{x}_{\mathbf{q},j,j'}v^{y*}_{\mathbf{q},j,j'}\omega
↪   _{\mathbf{q},j}\omega _{\mathbf{q},j'}\int_{-\infty}^{\infty}\textrm{d}\Omega
↪   \frac{\exp(\beta\Omega)}{\left(\exp(\beta\Omega) - 1\right)^2}\sigma
↪   _{\mathbf{q},j}(\Omega)\sigma_{\mathbf{q},j'}(\Omega)
```

Here the $\sigma_{\mathbf{q},j}$ is phonon spectral function. The python script that does the calculation is:

```
from __future__ import division, print_function

import time

import numpy as np

import cellconstructor as CC
import cellconstructor.ForceTensor
import cellconstructor.Phonons
import cellconstructor.ThermalConductivity

# Use the dynamical matrix and third-order force constants obtained with
# 10000 configurations.
```

```
dyn_prefix = "final_dyn_10000_"
nqirr = 4

# Useful conversion factors.
SSCHA_TO_MS = cellconstructor.ThermalConductivity.SSCHA_TO_MS
RY_TO_THZ = cellconstructor.ThermalConductivity.SSCHA_TO_THZ

# Read the auxiliary SSCHA dynamical matrix.
dyn = CC.Phonons.Phonons(dyn_prefix, nqirr)
supercell = dyn.GetSupercell()

# Build and initialize the third-order force-constant tensor.
fc3 = CC.ForceTensor.Tensor3(
    dyn.structure,
    dyn.structure.generate_supercell(supercell),
    supercell,
)
d3 = np.load("d3_10000.npy")
fc3.SetupFromTensor(d3)
fc3 = CC.ThermalConductivity.centering_fc3(fc3)

# Define the q-point grid for the Green-Kubo calculation.
meshnum = 10
mesh = [meshnum, meshnum, meshnum]

# Set up the ThermalConductivity object. With mode="GK", the method uses
# phonon spectral functions rather than phonon lifetimes.
tc = CC.ThermalConductivity.ThermalConductivity(
    dyn,
    fc3,
    kpoint_grid=mesh,
    scattering_grid=mesh,
    smearing_scale=1.0,
    smearing_type="adaptive",
    cp_mode="quantum",
    off_diag=True,
)

# Compute harmonic quantities once. They are reused for all values of ne below.
temperatures = np.linspace(300, 300, 1, dtype=float)
start_time = time.time()
tc.setup_harmonic_properties()
tc.write_harmonic_properties_to_file()

# Converge the Green-Kubo calculation with respect to the number of frequency
# points used to integrate overlaps of phonon spectral functions.
for i in range(5):
    ne = i * 2000 + 1000

    tc.calculate_kappa(
        mode="GK",
        temperatures=temperatures,
        write_lifetimes=False,
```

```
        gauss_smearing=False,
        ne=ne,
        offdiag_mode="perturbative",
        kappa_filename="Thermal_conductivity_GK_" + str(ne),
        lf_method="fortran-LA",
    )

# Save the ThermalConductivity object for post-processing.
tc.save_pickle()
```

The Green-Kubo calculation introduces an additional convergence parameter: the number of frequency points used to integrate the overlap of phonon spectral functions. The frequency spacing is approximately $2.1\omega_D/n_e$, where ω_D is the highest frequency on the **q**-point grid. For an accurate estimate of the integrals, this frequency spacing should be smaller than the smallest phonon linewidth. Therefore, materials with higher thermal conductivity will likely require larger values of `ne`.

The SRTA and GK methods usually agree well for weakly anharmonic materials. For strongly anharmonic materials, however, their results can differ significantly.

The final line of the previous script saves the `ThermalConductivity` object, which can later be used to analyze the calculated transport quantities. The snippet below reloads this object and writes the phonon spectral function at Γ:

```
from __future__ import division, print_function

import time

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

import cellconstructor as CC
import cellconstructor.ForceTensor
import cellconstructor.Phonons
import cellconstructor.ThermalConductivity

# Reload the ThermalConductivity object saved with tc.save_pickle().
tc = CC.ThermalConductivity.load_thermal_conductivity()

# Temperature-dependent quantities, such as lineshapes, are stored in
# dictionaries. The keys are strings formatted as format(T, ".1f").
keys = list(tc.lineshapes.keys())

# Find the Gamma point in the q-point grid.
for iqpt in range(tc.nkpt):
    if np.linalg.norm(tc.k_points[iqpt]) == 0.0:
        break

# Build the frequency grid associated with the saved lineshape data.
energies = (
    np.arange(len(tc.lineshapes[keys[-1]][0, 0]), dtype=float) * tc.delta_omega
    + tc.delta_omega
)

# Write the phonon spectral function at Gamma to a text file.
```

```
tc.write_lineshape(
    "Lineshape_at_Gamma",
    tc.lineshapes[keys[-1]][iqpt],
    iqpt,
    energies,
    "no",
)
```

Temperature-dependent transport properties, such as phonon lifetimes, heat capacities, and phonon lineshapes, are stored as dictionaries with keys corresponding to the temperature `T` formatted as `format(T, ".1f")`. The example above writes the phonon spectral function at Γ.

Other relevant quantities, such as phonon frequencies and group velocities, can also be accessed from the `ThermalConductivity` object. In addition, the object provides built-in functions for calculating the phonon density of states, phonon spectral conductivity, and phonon Grüneisen parameters in cubic systems

5.2 Exercise

Estimate the lattice thermal conductivity of CsPbI3 in the cubic $Pm\bar{3}m$ phase.

Before computing the thermal conductivity, first check the dynamical stability of this phase. In particular, determine the temperature range in which the cubic phase is dynamically stable and therefore the thermal-conductivity calculation is physically meaningful.

Then compare the lattice thermal conductivity obtained using:

- the single relaxation time approximation (**SRTA**), and
- the Green-Kubo (**GK**) approach.

Discuss the following points:

1. At what temperature does the cubic $Pm\bar{3}m$ phase become dynamically stable?
2. How does the thermal conductivity depend on temperature?
3. Do the SRTA and GK results agree? If not, which method gives a larger thermal conductivity and why?
4. What does the comparison between SRTA and GK tell you about the degree of anharmonicity in cubic CsPbI3?

Chapter 6

Hands-on-session 5: EPIq - Anharmonicity in electron-phonon coupling related properties

6.1 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.

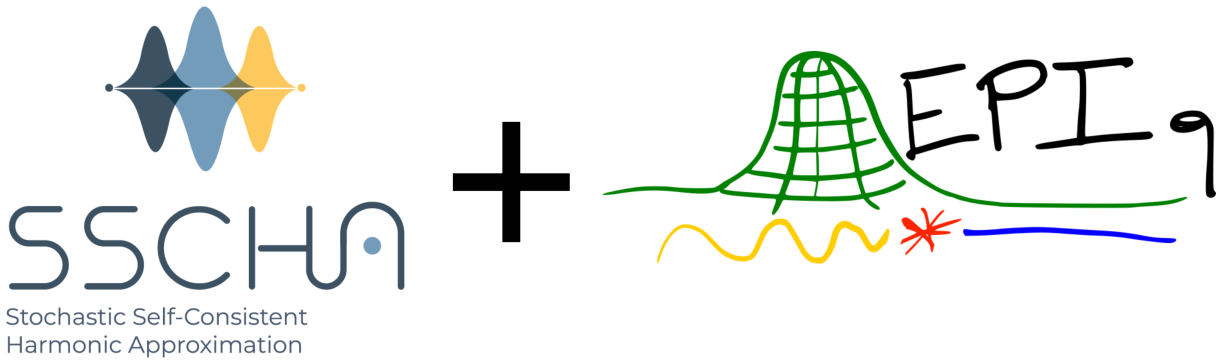

Figure 6.1: logo_epiq

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.

6.2 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 in the `Materials/tutorial_05` folder in the sscha school 2026 github repository. The folder contains

1. The electron-phonon coupling matrix elements can be found in the `mat_elem` folder. Each file corresponds to a different **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 ($D^{HARM}_{\mu,\nu}(\mathbf{k},\mathbf{q})$) 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.

6.2.1 Download and install EPIq

Deactivate the conda environment if it is active, typing `conda deactivate`. Enter the folder where you want to install epiq in your virtual machine and type on the command line

```
git clone --depth 1  --branch develop https://gitlab.com/the-epiq-team/epiq.git
```

then, enter the just created folder with `cd epiq` and install it typing the following command

```
./configure && make all CC="gcc -std=gnu89"
```

all the executables (.x files) will be installed in epiq/bin.

6.3 About EPIq

Epiq site: https://the-epiq-team.gitlab.io/epiq-site/

Epiq paper: https://www.sciencedirect.com/science/article/pii/S0010465523002953

The [Electron-Phonon Interpolation over q and k points] package exploits Wannier interpolation to obtain many proprieties in solids. Bloch theorem allows to describe an infinite system in real space with a continuum of Hamiltonian for different k-points in the Brillouin zone.

The properties of a material refers to a certain observable averaged over the sample. Thanks to the Bloch theorem it is often convenient to perform the average in the reciprocal k-space. In other words as a sum over the whole Brillouin zone of the quantities defined at each k-point.

$\langle O \rangle = \frac{1}{N_k} \sum_{\mathbf{k}} F_O(\mathbf{k})$

The quality of the averaging depends on the finesse of the sampling N_k and of the smoothness of the integrand function $F_O(\mathbf{k})$ (a constant function is totally described with just one k-point).

Wannier interpolation is an efficient way to refined the Brillouin zone sampling.

6.3.1 Calculations available in EPIq

1. Adiabatic (static) and non-adiabatic (dynamic) force constant matrices.
2. **Electron-phonon contribution to the phonon linewidth** and related quantities.
3. Isotropic and **anisotropic Eliashberg equations**.

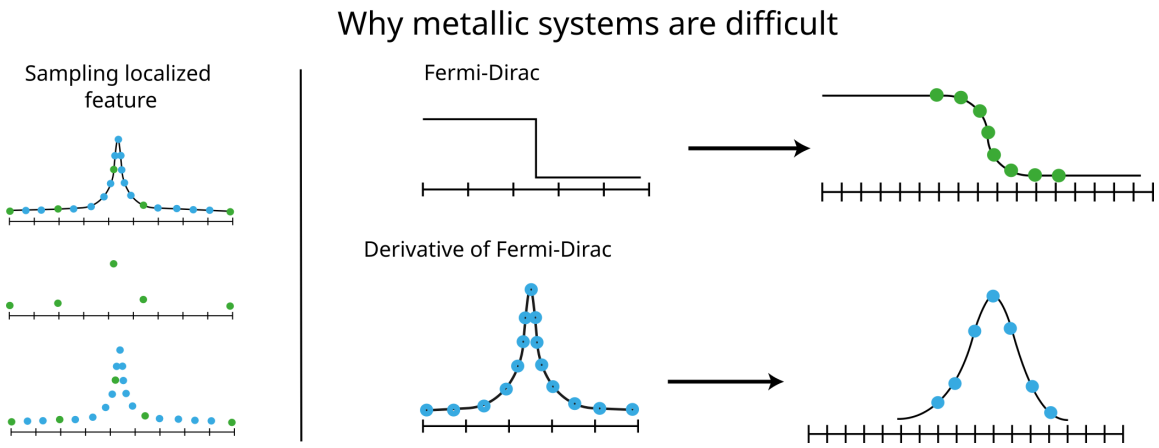

Figure 6.2: sampling

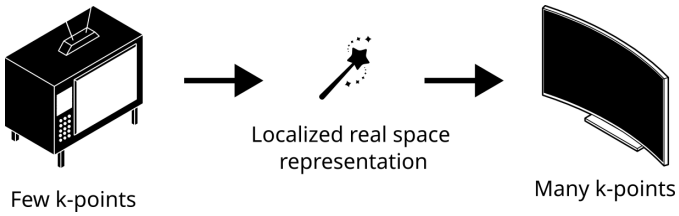

Figure 6.3: wannier_interpolation

4. Double Resonant Raman scattering.
5. Electron lifetime and relaxation time.

6.3.2 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.

6.3.3 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.

6.4 Setup phase: how to setup a calculation with EPIq

In this tutorial we calculate electron-phonon coupling related properties for doped monolayer MoS_2, and evaluate the effect of anharmonicity on them.

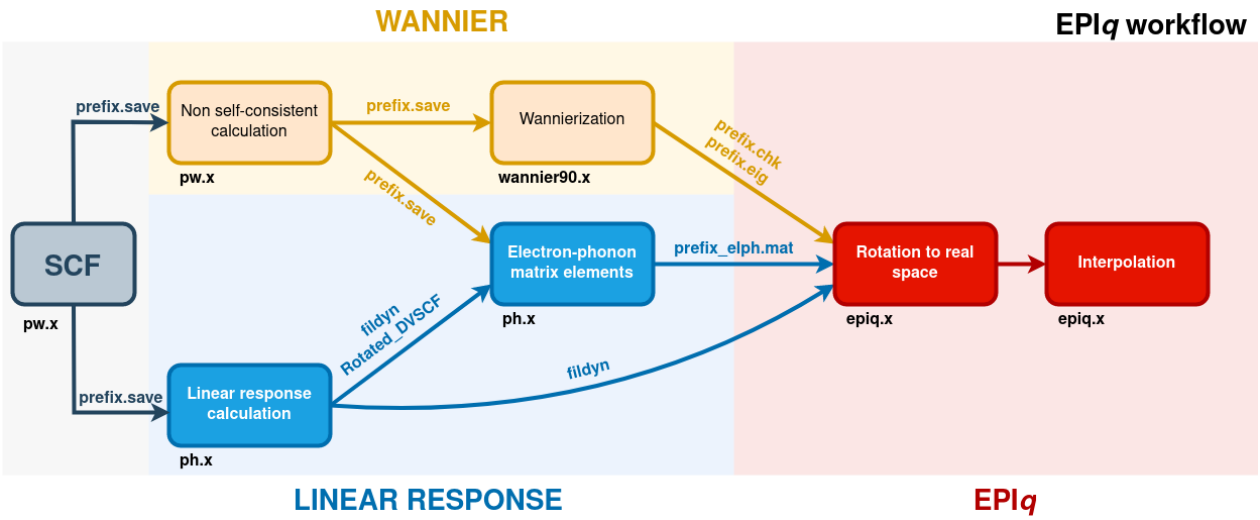

Figure 6.4: wannier

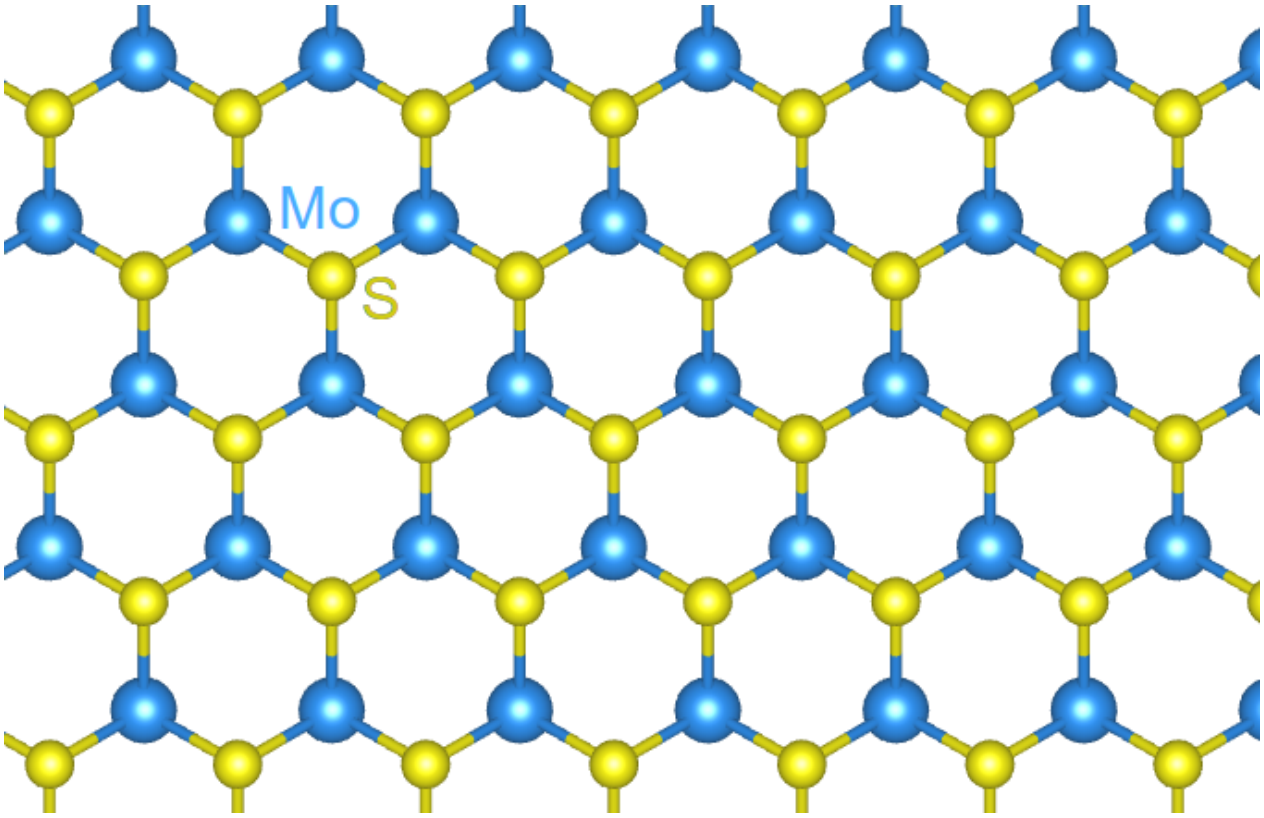

Figure 6.5: wannier

6.4.1 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/"
  /
  &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 `elphmat_dir` in the control namelist:

&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.

6.4.2 Check real space localization

If the Wannier transformation is well converged, the matrix elements are optimally localized in real space. Always check their localization.

Test:

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?

6.5 Exercise phase: calculation of electron-phonon coupling related properties for doped monolayer MoS_2

The phonon-electron linewidth $\gamma_{\mathbf{q},\nu}$ is a central quantity in conventional superconductivity theory. In the double delta (Allen) form is defined by the following equation:

$$\gamma_{\mathbf{q},\nu} = \frac{4\pi\omega_{\mathbf{q},\nu}}{N_k} \sum_{m,n} \sum_{\mathbf{k}} |g^{\nu}_{m,n}(\mathbf{k},\mathbf{q})|^2 \delta(\epsilon_{\mathbf{k}+\mathbf{q},m} - \epsilon_F)\delta(\epsilon_{\mathbf{k},n} - \epsilon_F)$$

where the electron-phonon coupling is defined from the deformation potential as:

$$g^{\nu}_{m,n}(\mathbf{k},\mathbf{q}) = \sum_s \mathbf{e}^s_{\mathbf{q},\nu} \cdot \mathbf{d}^s_{m,n}(\mathbf{k},\mathbf{q}) / \sqrt{2M_s\omega_{\mathbf{q},\nu}}$$

Consider the follwoing input file for the phonon-electron linewidth calculation γ at the M-point of the Brillouin zone. 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'`.

Exercise 1:

Perfom a convergence study of the linewidth at **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?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 2:

Compute the linewidth at $\mathbf{q} = \mathbf{M}$ using anharmonic dynamical matrix computed using SSCHA. Why does it seem like the first and the second mode are exchanged with respect to the harmonic dynamical matrices? What phonon mode(s) shows largest anharmonic correction?

More than one q-point can be specified in the phonon linewidth input file. For example:

```
crystal
 3
 0.001 0 0 1
 0.05 0 0 1
 0.10 0 0 1
```

A plottable file can then be obtained after the phonon linewidth calculation using the `linewidth_path.x` post-processing tool. Here is an example 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
```

You can then 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 ''
```

Exercise3:

Try to produce two plots of the phonon dispersion along the $\Gamma - \mathbf{M}$ high-symmetry line in the BZ, one employing the harmonic matrices and one with the anharmonic ones. Do you observe any differences?

6.6 Explanation and details: Phonon linewidth calculation

Let 's start again from the example file we considered for the calculation for phonon of momentum $\mathbf{q} = \mathbf{M}$ for two values of the electronic smearing.

```
&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
```

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 recalculated 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. To plot the results, first activate the conda environment with `conda activate sscha`.

```
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)')
    plt.legend(title=f"q={ np.round(np.array(q['xq_frac']),2) }")
    plt.show()
```

Remember to deactivate the environment when running epiq. 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.

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:

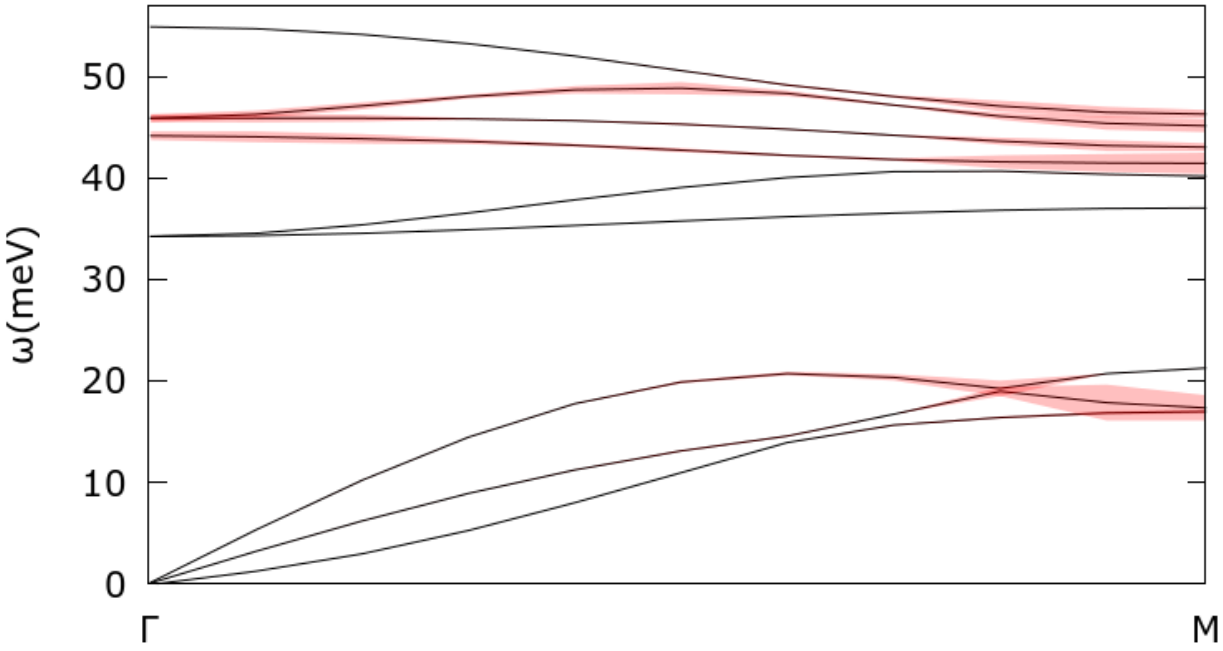

Figure 6.6: wannier

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
...
...
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
```

6.7 Advanced calculation: superconducting properties of doped MoS_2 using SSCHA Hessian matrices

Having access to phonon linewidths and phonon frequencies allow us to estimate the supercondcuting critical temperature thorugh the Allen-Dynes formula for T_c:

$$T_c = \frac{f_1 f_2 \omega_{ln}}{1.20} \exp\left[\frac{-1.04(1+\lambda)}{\lambda - \mu^*(1+0.62\lambda)}\right]$$

where the electron-phonon coupling parameter λ is defined as:

$$\lambda = 2 \int \frac{\alpha^2 F(\omega)}{\omega} d\omega,$$

and the Eliashberg function is defined as

$$\alpha^2 F(\omega) = \frac{1}{N(E_F)} \sum_{\mathbf{k}qv} \sum_{nm} \left| g^{v,mn}_{\mathbf{k}+\mathbf{q},\mathbf{k}} \right|^2 \times \delta(\omega - \omega_{\mathbf{q}v}) \delta(\epsilon^m_{\mathbf{k}+\mathbf{q}} - \epsilon_F) \delta(\epsilon^n_{\mathbf{k}} - \epsilon_F)$$

$$= \frac{1}{4\pi N(\varepsilon_F)} \sum_{\mathbf{q}v} \frac{\gamma_{\mathbf{q}v}}{\omega_{\mathbf{q}v}} \delta(\omega - \omega_{\mathbf{q}v}),$$

while the logharitmic average frequency is: $\omega_{\ln} = \exp\left[\frac{2}{\lambda}\int_0^\infty \frac{\alpha^2 F(\omega)}{\omega} \ln(\omega) d\omega\right]$

To calculate the q-sum appearing in the Eliashberg function, we would like to calculate the linewidths on a grid. In order to do so, we only have to modify the q-points part of the linewidth input file as follows:

```
&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/dynq'
   Fourier_interp_dyn=.true.,
   nq1=8,nq2=8,nq3=1,
&end
k-points
automatic
12 12 1  0  0  0
q-points
crystal
automatic
12 12 1 0 0 0
```

Finally, the superconducting T_c can be calculate with the `average_lambda.x` post processing with the following input file:

```
&input_lambda
prefix='MoS2'
n_mustar=15
lkp_sequential=.true.
mu_min=0.01
mu_max=0.15
sigma_min=0.01,
```

```
sigma_max=0.02,
nsigma=2,
&end
AUTOMATIC
12 12 1 0 0 0
```

besides the automatic grid in input (same as the q-points grid in the linewidth calculation), this input only contains σ (smearing employed in the double delta q summation) and μ^* ranges to be employed in the T_c calculation. Then simply run

```
{$path_to_epiq}/bin/average_lambda.x <input_average_lambda.in
```

This will produce an output named `MoS2_average_lambda.d` , containing relevant information for superconductivity as a function of the smearing parameter σ and μ^*.

Exercise:

Evaluate the superconducting critical temperature for doped MoS_2, first employing harmonic dynamical matrices and then the one obtained from the SSCHA minimization. Does the predicted superconducting critical temperature change?

Employ a 12x12x1 automatic k-grid and q-grid in the phonon linewidth calculation as a test. You can then study the convergence as a function of the k- and q- grid dimension in the linewidth calculation. You should see that the predicted critical temperature is lower when the anharmonic Hessian matrices are employed.

General remarks:

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.