python-sscha Documentation

Release 1.4

Lorenzo Monacelli

Dec 18, 2023

CONTENTS:

INTRODUCTION

1.1 What is python-sscha?

python-sscha is both a python library and a stand-alone program to simulate quantum and thermal fluctuations in solid systems.

1.2 Why do I need python-sscha?

If you are simulating transport or thermal properties of materials, phase diagrams, or phonon-related properties of materials, you need python-sscha. It is a package that enables you to include the effect of both thermal and quantum phonon fluctuations into your *ab initio* simulations.

The method used by this package is the Stochastic self-consistent Harmonic Approximation (SSCHA). The SSCHA is a full-quantum method that optimizes the nuclear wave-function (or density matrix at finite temperature) to minimize the free energy. In this way, you can simulate highly anharmonic systems, like those close to a second-order phase transition (as charge density waves and thermoelectric materials). Despite the full quantum and thermal nature of the algorithm, the overall computational cost is comparable to standard classical molecular dynamics. Since the algorithm correctly exploits the symmetries of the crystal, it is also much cheaper.

python-sscha comes both as a python library that can be run inside your workflows and as stand-alone software, initialized by input scripts with the same syntax as Quantum ESPRESSO.

You can couple it with any *ab initio* engine for force and energy calculations. It can interact through the Atomic Simulation Environment (ASE) and has an implemented interface for automatic submission of jobs in a remote cluster.

Moreover, it is easy to use, with short input files highly human-readable. What are you waiting for? Download and install python-sscha, and start enjoying the Tutorials!

HOW TO INSTALL

The SSCHA code is a collection of 2 python packages: CellConstructor and python-sscha. In this guide, we refer to the installation of python-sscha.

2.1 Easy installation using Anaconda

The easy way to install python-sscha is to use the anaconda distribution of python.

```
conda create -n sscha -c conda-forge python=3.11 gfortran libblas lapack␣
↪openmpi julia openmpi-mpicc pip numpy scipy spglib
conda activate sscha
pip install ase julia mpi4py
pip install cellconstructor python-sscha tdscha
```

This will create a new environment called sscha, install all the dependencies and the packages. To use the code, you need to activate the environment:

```
conda activate sscha
```

The sscha code exploits the julia language to speed up the calculation. To install the julia dependencies, you need to run the following command:

```
python -c 'import julia; julia.install()'
```

And that's it! You can now run the sscha code.

```
sscha -h
```

SSCHA also works as a library, so you can import it in your python scripts. To check that everything is working, you can run the following script:

```
import sscha, sscha.Ensemble
print("Hello world!")
```

If it runs without problems, the SSCHA code is working correctly.

2.2 Requirements

To install python-sscha you need: 1. python<=3.11 2. numpy 3. scipy 4. matplotlib 5. Lapack 6. Blas 7. gfortran (or any fortran compiler) 8. CellConstructor

For python, we strongly recommend using the anaconda distribution, that already comes with numerical packages correctly compiled to exploit multithreading.

The numpy, scipy and matplotlib are python packages. These are usually provided with a new installation of python distributions like anaconda. Lapack and Blas are needed for compiling the FORTRAN code (together with a FORTRAN compiler like gfortran). In many Linux distributions like ubuntu they can be installed as

```
sudo apt-get install libblas-dev liblapack-dev liblapacke-dev gfortran
```

Note that this specific command may change in time.

Together with these mandatory requirements (otherwise, the code will not compile correctly or raise an exception at the startup), we strongly recommend installing also the following libraries: 1. Atomic Simulation Environment (ASE) 2. SPGLIB

If these packages are available, they will enable the automatic cluster/local calculation of forces (ASE) and the symmetry recognition in the supercell (SPGLIB).

To install all the python dependencies (and recommended) automatically, you may just run:

```
pip install -r requirements.txt
```

2.3 Installation from pip

The easiest way to install python-sscha (and CellConstructor) is through the python package manager:

```
pip install python-sscha
```

Eventually, you can append the –user option to install the package only for the user (without requiring administrator powers). Pip will check for requirements automatically and install them. This method only works if pip is already installed with python.

2.4 Installation from source

Once all the dependences of the codes are satisfied, you can unzip the source code downloaded from the website. Then run, inside the directory that contains the setup.py script, the following command:

```
python setup.py install
```

As for the pip installation, you may append the –user option to install the package only for the user (without requiring administrator powers).

2.5 Install with Intel FORTRAN compiler

The setup.py script works automatically with the GNU FORTRAN compiler. However, due to some differences in linking lapack, to use the intel compiler you need to edit a bit the setup.py script:

In this case, you need to delete the lapack linking from the setup.py and include the -mkl as linker option. Note that you must force to use the same liker compiler as the one used for the compilation.

2.6 Install with a specific compiler path

This can be achieved by specifying the environment variables on which setup.py relies:

1. CC (C compiler)
2. FC (Fortran compiler)
3. LDSHARED (linking)

If we want to use a custom compiler in /path/to/fcompiler we may run the setup as:

```
FC=/path/to/fcompiler LDSHARED=/path/to/fcompiler python setup.py install
```

A specific setup.py script is provided to install it easily in FOSS clusters.

QUICK START

In this chapter we provide ready to use examples to setup your first SSCHA calculation.

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

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

You can find there a lot of structures ready to use for your SSCHA runs.

For the purpouse of this tutorial, we are going to use the EMT force field, so that the calculation can be run in a laptop without the need of a supercomputer. We explain in a later section how to couple the SSCHA with a cluster to submit the same calculation fully ab-initio.

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

- Compute the harmonic phonons (dynamical matrix)

- Remove imaginary frequencies (if any)

- Run the SSCHA

We prepared an input file in the form of a python script (tested with python-sscha version 1.2) which makes all these passages automatically.

You find a copy of the script and the cif file of Gold inside the directory Examples/ThermodynamicsOfGold

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

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

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

# Import numerical and general pourpouse libraries
```

(continues on next page)

```python
import numpy as np, matplotlib.pyplot as plt
import sys, os


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

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

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

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

# Compute the harmonic phonons
# (as from version 1.3 of Cellconstructor, it will exploit the symmetries as␣
↪phonopy to reduce the number of displacements)
# NOTE: if the code is run with mpirun, the calculation goes in parallel
gold_harmonic_dyn = CC.Phonons.compute_phonons_finite_displacements(gold_
↪structure_relaxed, calculator, supercell = (4,4,4))

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


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

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

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

TEMPERATURE = 300
N_CONFIGS = 50
MAX_ITERATIONS = 20

# Initialize the random ionic ensemble
```

```python
ensemble = sscha.Ensemble.Ensemble(gold_harmonic_dyn, TEMPERATURE)

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

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

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



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

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

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

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

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

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

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

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

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

You will see two windows.

In Fig. 3.1 we have all the minimization data. On the top-left panel, we see the free energy. As expected, it decreases (since the SSCHA is minimizing it). You can see that at certain values of the steps there are discontinuities. These occurs when the code realizes that the ensemble on which it is computing is

---
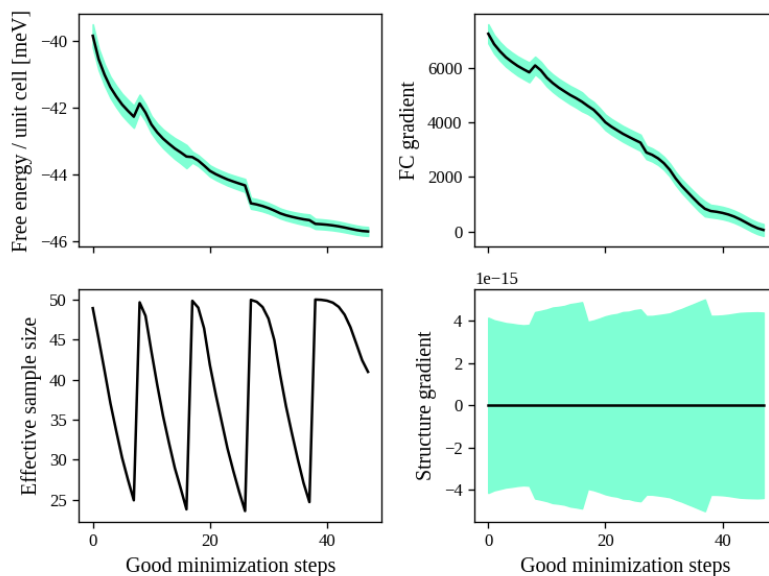
Fig. 3.1: Minimization data of Gold.

no more good and a new one is generated. The goodness of an ensemble is determined by the Kong-Liu effective sample size (bottom-left). When it reaches 0.5 of its initial value (equal to the number of configurations), the ensemble is extracted again and a new iteration starts. You see that in the last iteration, the code stops before getting to 25 ($0.5 \cdot 50$). This means that the code converged properly: the gradient reached zero when the ensemble was still good.

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

Indeed, since the gold atomic positions are all fixed by symmetries, it is always zero (but it will be different from zero in more complex system).
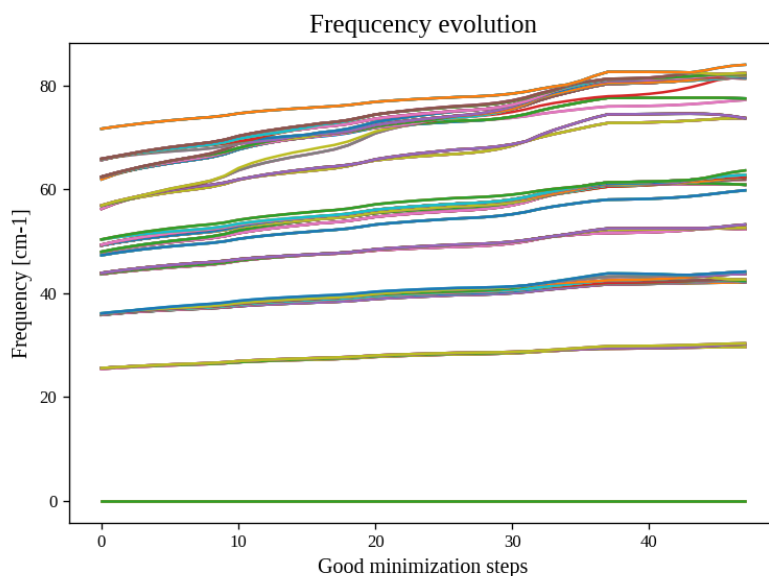
Fig. 3.2: All the SSCHA phonon frequencies as a function of the step in the NVT simulation.

Instead, Fig. 3.2 represents the evolution of the SSCHA phonon frequencies. Here, all the frequencies in the supercell (at each q point commensurate with the calculation) are shown.

By looking at how they change you can have an idea on which phonon mode are more affected by anharmonicity. In this case, it is evident that Gold is trongly anharmonci and that the finite temperature tend to harden almost all the phonon frequencies.

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

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

3.2 Plot the phonon dispersion

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

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

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

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

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

import sys, os

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

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

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

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

```python
# -------------------- THE SCRIPT FOLLOWS --------------------

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

# Get the band path
qpath, data = CC.Methods.get_bandpath(harmonic_dyn.structure.unit_cell,
            PATH,
            SPECIAL_POINTS,
            N_POINTS)
xaxis, xticks, xlabels = data # Info to plot correclty the x axis

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

nmodes = harmonic_dyn.structure.N_atoms * 3

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

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

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

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

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

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

plt.tight_layout()
```

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

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

```
$ python plot_dispersion.py
```
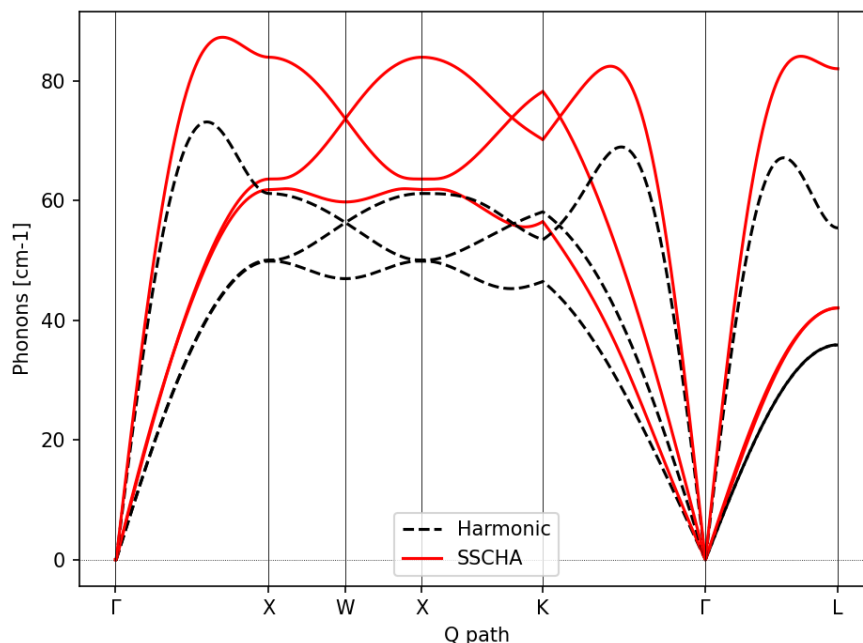

Fig. 3.3: Comparison between the SSCHA and the harmonic phonon dispersion of Gold.

The script will plot the figure of the phonon dispersion Fig. 3.3. It is quite different from the experimental one because of the poor accuracy of the force field, however, the SSCHA results is much closer to the experimental value.

3.3 Analysis of the input script for the NVT simulation

While the input may seem long, it is heavily commented, but lets go through it step by step. At the very beginning, we simply import the sscha libraries, cellconstructor, the math libraries and the force field. This is done in python with the *import* statemets.

The first real part of the code is:

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

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

# Relax the gold structure (useless since for symmetries it is already␣
↪relaxed)
```

```
relax = CC.calculators.Relax(gold_structure, calculator)
gold_structure_relaxed = relax.static_relax()
```

Here we initialize a cellconstructor structure from the cif file downloaded from the material database (*Au.cif*). We initialize the EMT calculator from ASE, and relax the structure.

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

In the next part of the code, we perform the harmonic phonon calculation using cellconstructor and a finite displacement approach:

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

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



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

The method *compute_phonons_finite_displacements* is documented in the CellConstructor guide. It requires the structure (in this case *gold_structure_relaxed*), the force-field (*calculator*) and the supercell for the calculation. In this case we use a 4x4x4 (equivalent to 64 atoms). This may not be sufficient to converge all the properties, especially at very high temperature, but it is just a start. Starting from version 1.3 of Cellconstructor, this automatically exploits the symmetry of the structure to reduce the number of displacements to compute. This is extremely efficient for big supercells. Note that *compute_phonons_finite_displacements* works in parallel with MPI, therefore, if the script is executed with *mpirun -np 4 python myscript.py* it will split the calculations of the finite displacements across 4 processors. You need to have mpi4py installed.

After computing the harmonic phonons in gold_harmonic_dyn, we impose the correct symmetrization and the acousitic sum rule with the *Symmetrize* method, and save the result in the quantum ESPRESSO format with *save_qe*. This should not be the case for Gold, however, if we have a structure which has imaginary phonon frequencies, we need to get rid of them before starting the SSCHA. This is achieved with *ForcePositiveDefinite* (see CellConstructor documentation for more details on how these methods work).

Now we are ready to submit the SSCHA calculation in the NVT ensemble!. The important parameters are:

- The temperature

- The number of random configurations in the ensemble

- The maximum number of iterations

These parameters are almost self-explaining. However, we give a brief overview of how the SSCHA works to help you understand which are the best one for your case. While MD or MC calculation represent the equilibrium probability distribution over time of the system by updating a single structure, the SSCHA

encodes the whole probability distribution as an analytical function. Therefore, to compute properties, we can generate on the fly the ionic configurations that represent the equilibrium distributions. The number of random configuration is exactly how many ionic configuration we generate to compute the properties (Free energy and Stress tensors)

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

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

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

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

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

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

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

So you see many classes. *ensemble* represent the ensemble of ionic configurations. We initialize it with the dynamical matrix (which represent how much atoms fluctuate around the centroids) and the temperature. *minim* is a *SSCHA_Minimizer* object, which performs the free energy minimization. It contains all the info regarding the minimization algorithm, as the initial timestep (that here we set to 0.01). You can avoid setting the time-step, as the code will automatically guess the best value. The *relax* is a *SSCHA* object: the class that takes care about the simulation and automatizes all the steps to perform a NVT or NPT calculation. We pass the minimizer (which contains the ensemble with the temperature), the force-field (*calculator*), the number of configurations *N_configs* and the maximum number of iterations.

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

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

3.4 Exercise

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

How does the phonon bands behaves with temperature? Do they become more rigid (energy increases) or softer?

3.5 Running in the NPT ensemble: simulating thermal expansion

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

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

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

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

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

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

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

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

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

You can find the full script in Examples/ThermodynamicsOfGold/thermal_expansion.py

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

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

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

(continues on next page)

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


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



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


N_CONFIGS = 50
MAX_ITERATIONS = 10


# Import the gold force field
calculator = EMT()


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


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


# We cycle over several temperatures
t = T_START



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

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

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

3.5. Running in the NPT ensemble: simulating thermal expansion

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

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

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

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

    # Update the temperature
    t += DT

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

You can run the script as always with:

```
$ python thermal_expansion.py
```

And … done!

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

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

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

import scipy, scipy.optimize



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

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

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

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

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

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


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

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

We report the final thermal expansion in Fig. 3.4. The volumetric expansion coefficient α_v is obtained from the fit thanks to the thermodynamic relation:

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

Also in this case, the result is quite off with experiments, due to the not completely realistic force-field employed. To get a more realistic approach, you should use *ab-initio* calculations or a more refined force-field.
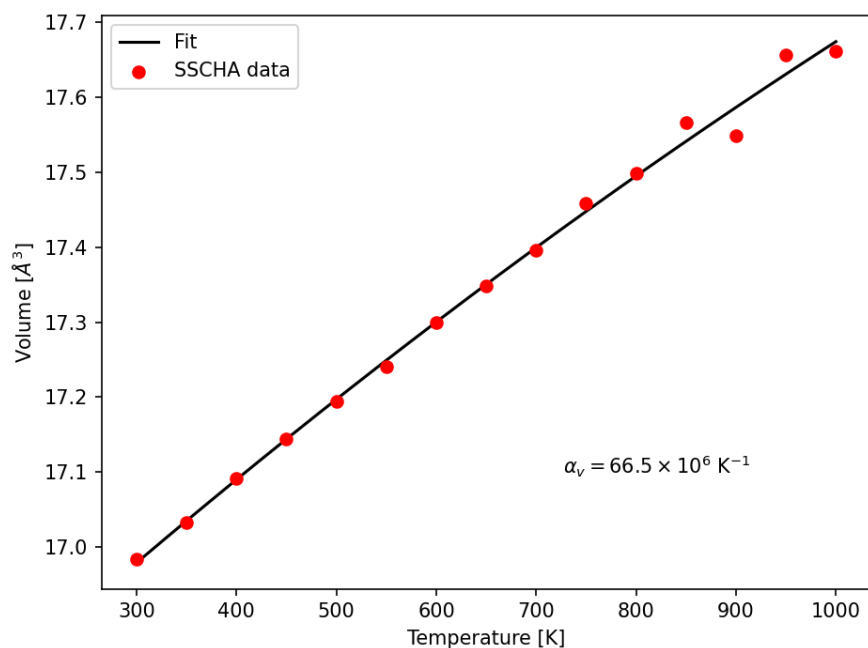
Fig. 3.4: Thermal expansion of Gold. From the fit of the data we can compute the volumetric thermal expansion coefficient (at 300 K).

3.6 Ab initio calculation with the SSCHA code

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

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

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

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

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

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

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

```python
import cellconstructor.calculators

# Initialize the DFT (Quantum Espresso) calculator for gold
```

(continues on next page)

python-sscha Documentation, Release 1.4

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


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


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



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

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

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

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

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

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

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

# Import the DFT calculator
import cellconstructor.calculators

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


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

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

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

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

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




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

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

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

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

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

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



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

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

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

# Now we can save the final dynamical matrix
```

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

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

3.7 Parallelization

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

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

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

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

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

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

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

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

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

With this setup, the full code is parallelized over 400 processors. However the SSCHA minimization

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

In the next section, we provide a workaround: Running the SSCHA code on your laptop, and configure it to automatically interact with a remote server (HPC) in which the ensemble calculation is submitted. This is the best way to use the python-sscha code, as no installation of the code in the cluster is required, and the time spent during the minimization does not occupy precious HPC resources.

3.8 Remote submission on a queue system

To configure SSCHA to work with a cluster we need to tell the code some info to ssh into it. Here is a basic configuration to connect to the Piz Daint cluster at CSCS (Switzerland), but it is very similar to any other cluster.

First of all, we need to configure a straight ssh connection. We have to add to the configuration file of ssh the information about Host and Username. In my case, the $HOME/.ssh/config file looks like:

```
Host ela
     HostName ela.cscs.ch
     User myusernameforela
Host daint
     HostName daint.cscs.ch
     ProxyJump ela
     User myusernamefordaint
```

These are two connection, one to ela (the front-end server of CSCS) and then one to daint (Piz Daint supercomputer). To connect to daint, I must before access to ela, this is the reason of ProxyJump command inside the daint block. The best way to connect is to configure your ssh private-public key so that you (and python-sscha) can login without password. An example of how to do that is Here, but you may find a lot of other on internet.

Sometimes cluster may not allow passwordless connection, in this case, you need to provide the password explicitly to python-sscha.

```
# Let us define the cluster
import sscha.Cluster
cluster = sscha.Cluster.Cluster(hostname = 'daint', pwd = None)  # Put the
↪password in pwd if needed

# Configure the submission strategy
cluster.account_name = 's1073'  # Name of the account on which to subtract
↪nodes
cluster.n_nodes = 1                 # Number of nodes requested for each job
cluster.time = '02:30:00'       # Total time requested for each job
cluster.n_pool = 1              # Number of pools for the Quantum ESPRESSO
↪calculation

# Here some custom parameters for the clusters
# These are specific for daint, but you can easily figure out those for your
↪machine
```

(continues on next page)

```python
cluster.custom_params["--constraint"] = "gpu"        # Run on the GPU partition
cluster.custom_params["--ntasks-per-node"] = '2'
cluster.custom_params["--cpus-per-task"] = '6'


# Since daint specify the partition with a custom option,
# Lets remove the specific partition option of SLURM
# Neither we want to specify the total number of cpus (automatically␣
↪determined by the node)
cluster.use_partition = False
cluster.use_cpu = False



# Now, we need to tell daint which modules to load to run quantum espresso
# Also this is cluster specific, but very simple to figure it out for you
cluster.load_modules = """
# Load the quantum espresso modules
module load daint-gpu
module load QuantumESPRESSO

# Configure the environmental variables of the job
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
export NO_STOP_MESSAGE=1
export CRAY_CUDA_MPS=1

ulimit -s unlimited
"""

# Now, what is the command to run quantum espresso on the cluster?
cluster.binary = "pw.x -npool NPOOL -i PREFIX.pwi > PREFIX.pwo"
# NOTE that NPOOL will be replaced automatically with the cluster.n_pool␣
↪variable



# Let us setup the working directory (directory in which the jobs runs)
cluster.workdir =  "$SCRATCH/Gold_NVT_300k"
cluster.setup_workdir()  # Login to the cluster and creates the working␣
↪directory if it does not exist



# Last but not least:
#  How many jobs do you want to submit simultaneously?
cluster.batch_size = 10


#  Now many DFT calculation do you want to run inside each job?
cluster.job_number = 5
```

And here we go. We configured the connection to daint. It may seem a bit complex, however, all the configuration is a simple interface to SLURM, and we are only passing informations that are going into the #SBATCH syntax.

Once you defined the cluster, you can run the calculation providing the cluster object to the ss-cha.Relax.SSCHA class:

```python
# Initialize the simulation
relax = sscha.Relax.SSCHA(minim, calculator,
            N_configs = N_CONFIGS,
            max_pop = MAX_ITERATIONS,
            cluster = cluster,
            save_ensemble = True)
```

And that's it. Instead of running espresso locally, python-sscha submit the calculations on the specified cluster. Here we also added the save_ensemble flag equal to true. Since DFT calculation are computationally expensive, we may want to save the results of each ensemble for further analysis (as computing linear response properties), or maybe to train a neural-network potential later.

By default, the ensemble is stored into a directory called 'data'. You can change it to what you want by editing the setting

```python
relax.data_dir = 'my_new_data'
```

If the directory does not exist, python-sscha creates it automatically. While the cluster configuration may seem a bit more complex, it is the best way to go.

3.9 Other tutorials

Congratulations. If you arrived here, it means that you now have a good overview of the basic functionalities of the SSCHA code, as running an NVT and NPT simulations.

We provide some other tutorial to help you to get used to some other advanced properties, like linear response (Hessian matrix) and spectral properties.

You find these tutorials as executable jupyter notebooks in the Tutorial folder, or in the static html version on the sscha website: www.sscha.eu/tutorials

Tutorials are organized as follows:

1. Setup from the structure and manual submission: PbTe tutorial. Here you learn how to set up a SSCHA calculation starting just with the structure (we provide a .cif file of the PbTe at high temperature). The tutorial will guide you step by step. You will learn how to: prepare the starting data needed for the SSCHA calculation, generate a random ensemble, save the ensemble and prepare input files for your favorite ab-initio code, read back the energies and the forces inside SSCHA, run a SSCHA minimization. You will also learn how to use ASE and the Cluster module to automatize the calculation of the ensemble and submit it to a HPC system.

2. Automatic relaxation with a force field: SnTe_ToyModel. Here, we show how to use a force-field for a SSCHA calculation, running everything on your computer. We also will explain how to calculate the free energy hessian for second-order phase transitions, and study a phase transition as a function of temperature.

3. Variable cell relaxation: LaH10 tutorial. Here you learn how to perform an automatic calculation with a variable cell. You will exploit the quantum effect to search the high-temperature superconductive phase (Fm-3m) of LaH10 below 200 GPa, starting from a distorted structure.

4. Hessian matrix calculation for second-order phase transitions: H3S tutorial. Here you reproduce the calculation of the Hessian of the free energy to assert the stability of the H3S phase.

5. Spectral properties: Spectral_Properties. In this tutorial, we explain how to use the post-processing utilities of the SSCHA to calculate the phonon spectral function, and computing phonon lifetimes, and plotting interacting phonon dispersion. We provide an ensemble for PbTe already computed ab-initio.

The jupyter notebooks are interactive, to quickly start with your simulation, pick the tutorial that resembles the kind of calculation you want to run, and simply edit it directly in the notebook.

ADVANCED FEATURES

The python-sscha code can be runned both as a stand-alone application with an input file and as a python library, writing a python script.

We will cover the python scripting as it is more general.

The SSCHA calculation is divided into 3 main steps:

- The generation of a random ensemble of ionic configurations

- Calculations of energies and forces on the ensemble

- The SSCHA free energy minimization

Then this steps are iterated until convergence is achieved.

In this chapter, we cover some advanced features of the SSCHA code, as the manual submission, the use of constrains on modes and atoms or the configurations of cluster with a DFT code different from quantum ESPRESSO.

4.1 Manual submission

The manual submission allows the user to take full controll over any steps in the simulation. It also means that the code perform just one iteration, and the user must interact with it to provide the forces and energies of the ensemble at each iterations.

It is usefull if you want to have full control on the number of configurations required to converge, or if you simply do not want to configure the automatic submission through a cluster because you have limited resources and are scared that the code could burn too much computer time without you realizing.

Indeed, it is strongly discuraged in variable cell simulations, as the code exploits the results from previous iterations to optimize the cell in a clever way.

The manual submission means that the user manually computes the energies and forces of all the configurations in the ensemble. The code stops after generating the random ensemble, and the user is requested to provide data files that contain the forces and total energies for each configuration.

Thus, the code works in two steps. In the first step we generate the ensemble. Here is the code

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

# Load the harmonic dynamical matrix
dyn = CC.Phonons.Phonons('dyn', nqirr = 4)
```

```python
# If the dynamical matrix contains imaginary frequencies
# we get rid of them with
dyn.ForcePositiveDefinite()


# Now we initialize the ensemble at the target temperature
TEMPERATURE = 300 # Kelvin
ensemble = sscha.Ensemble.Ensemble(dyn, TEMPERATURE)


# We generate the random ensemble with N_CONFIGS configurations
N_CONFIGS = 1000
ensemble.generate(N_CONFIGS)


# We save the ensemble on the disk (inside directory data)
# We specify an integer 'population' which distinguish several ensembles
# inside the same directory
ensemble.save('data', population = 1)
```

To start, we need an initial guess of the dynamical matrix (the dyn file). The default format is the one of Quantum ESPRESSO, but also phonopy and ASE formats are supported (refer to the CellConstructor documentation to load these formats). Here we assume that the dynamical matrices are 4 (4 irreducible q points) called 'dyn1', 'dyn2', 'dyn3' and 'dyn4', as the standard quantum espresso format.

The dynamical matrix contain both the information about the atomic structure and the ionic fluctuations. These can be obtained with a linear response calculation from DFT.

The previous code generates the ensemble which is stored in the disk. Inside the data directory you will find a lot of files

The files named 'scf_population1_X.dat' with X going over all the configurations contain the atomic structure in cartesian coordinates. It uses the standard espresso formalism.

You need to compute total energies and forces of each configuration, with your favourite code. The total energies are written in column inside the file 'total_energies_population1.dat', in Rydberg atomic units and ordered with the index of the configurations. The forces for each configuration should be inside 'forces_population1_X.dat' in Ry/Borh (Rydberg atomic units).

When you compute energies and forces, you can load them and run the SSCHA minimization:

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

# Load the harmonic dynamical matrix
dyn = CC.Phonons.Phonons('dyn', nqirr = 4)

# If the dynamical matrix contains imaginary frequencies
# we get rid of them with
dyn.ForcePositiveDefinite()


# Now we initialize the ensemble at the target temperature
TEMPERATURE = 300 # Kelvin
ensemble = sscha.Ensemble.Ensemble(dyn, TEMPERATURE)
```

```python
# We load the ensemble
N_CONFIGS = 1000
ensemble.load('data', population = 1, N = N_CONFIGS)


# Now we can run the sscha minimization
minim = sscha.SchaMinimizer.SSCHA_Minimizer(ensemble)
minim.init()
minim.run()


# Print on stdout the final results
minim.finalize()


# Save the output dynamical matrix
minim.dyn.save_qe('final_dyn')
```

And that's it. You run your first manual calculation.

4.2 Keep track of free energy, gradients and frequencies during minimization

It is convenient to store on the file the information during the minimization, as the Free Energy, its gradient values and the frequencies.

To do this, we need to tell the code to save them into a file.

Let us replace the 'minim.run()' line in the previous example with the following code:

```python
import sscha.Utilities
IO = sscha.Utilities.IOinfo()
IO.SetupSaving('minim_data')

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

If you run it again, the code produces (starting from verison 1.2) two data files: minim_data.dat and minim_data.freqs. You can plot all the minimization path (frequencies, free energy, gradients) calling the program:

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

The sscha-plot-data.py script is automatically installed within the SSCHA code.

4.3 Load from the output files

It is possible to load an ensemble directly from a list of output files from a specific program, like quantum ESPRESSO. This is usefull when a calculation ended with an error on the cluster, and the ensemble has not been saved on the disk (but you have the output files of the configurations that have been already computed succesfully).

In this case, you can restart the minimization using an ensemble with the following code:

```python
# Load the dynamical matrix
dyn = CC.Phonons.Phonons("dyn", 4)

# Load the ensemble
ens = sscha.Ensemble.Ensemble(dyn, 300)

# Load the ensemble from the output of the calculator
# In this case, the pwo files are output of the quantum espresso program.
# Any output file that ASE is able to read can be used to load the ensemble.
ens.load_from_calculator_output(directory="data", out_ext=".pwo")


# Run the minimization
minim = sscha.SchaMinimizer.SSCHA_Minimizer(ens)
minim.init()
minim.set_minimization_step(0.01)
minim.run()
minim.finalize()
```

An example of this procedure is provided in the `tests/test_load_ensemble_from_calculator_output` directory.

Alternatively, you can also use the ensemble to restart a full relax procedure. In this case, you need to provide the key `restart_from_ens = True` to the `relax` or `vc_relax` methods of the SSCHA class in the `Relax` module.

```python
# Initialize the minimizer minim (see example above)
relax = sscha.Relax.SSCHA(minim, ase_calculator=calculator,
                     N_configs = 100, max_pop = 20)
relax.relax(restart_from_ens = True)

# Or, if you want to perform a variable cell relaxation
relax.vc_relax(restart_from_ens = True)
```

4.4 Cluster configuration with a code different from Quantum ESPRESSO

TODO

4.5 Employ a custom function

An interesting feature provided by the SSCHA code is the customization of the algorithm. The user has access to all the variables at each iteration of the minimization. In this way, the user can print on files additional info or introduce constraints on the structure or on the dynamical matrix. The interaction between the user and the SSCHA minimization occurs through three functions, that are defined by the user and passed to the **run** method of the **SSCHA_Minimizer** class (in the **SchaMinimizer** module):

- custom_function_pre

- custom_function_gradient

- custom_function_post

These functions are called by the code before, during, and after each iteration.

The **Utilities** module already provides some basic functions, that can be used for standard purpouses. For example, the following code employs *custom_function_post* to print on a file the auxiliary dynamical matrix's frequencies at each step.

```
IO = sscha.Utilities.IOinfo()
IO.SetupSaving("freqs.dat")
# .... initialize minim as SSCHA_Minimizer class
minim.run( custom_function_post = IO.CFP_SaveAll)
```

In this case *IO.CFP_SaveAll* is the *custom_function_post*. It is a standard python method, that takes one argument (the SSCHA_Minimizer). *IO.CFP_SaveAll* prints the frequencies of the current dynamical matrix (stored in minim.dyn) in the filename defined by *IO.SetupSaving("freqs.dat")*.

The following example, we define a *custom_function_post* not provided by the Utilities module. The following code generate a file with the full dynamical matrix for each iteration of the minimization algorithm.

```
def print_dyn(current_minim):
        # Get the current step id checking the lenght of the __fe__ variable␣
↪(the free energy)
        step_id = len(current_minim.__fe__)

        # Save the dynamical matrix
        minim.dyn.save_qe("dyn_at_step_{}_".format(step_id))
```

Here, *print_dyn* is the *custom_function_post*. We must pass it to the *run* method of the *SSCHA_Minimizer* class (minim in the following case).

```
minim.run(custom_function_post = print_dyn)
```

In this way, you can interact with the code, getting access to all the variables of the minimization after each step. This could be exploited, for example, to print atomic positions, bond lenght distances or

angles during the minimization, or to setup a live self-updating plot of the free energy and its gradient, that automatically refreshes at each step.

4.6 Constraints

Another important case in which you want to interact with the code is to constrain the minimization. A standard constraint is the locking of modes, in which you only optimize a subset of phonon branches defined from the beginning. Let us have a look at the code to constrain the modes:

```python
# [...] Load the initial dynamical matrix as dyn
ModeLock = sscha.Utilities.ModeProjection(dyn)

# Setup the constrain on phonon branches from 4 to 8 (ascending energy)
ModeLock.SetupFreeModes(4, 8)

# [...] Define the SSCHA_Minimizer as minim
minim.run(custom_function_gradient = ModeLock.CFG_ProjectOnModes)
```

The function *ModeLock.CFG_ProjectOnModes* is the *custom_function_gradient*. It takes two numpy array as input: the gradient of the dynamical matrix and the gradient on the structure. Since numpy array are pointers to memory allocations, the content of the array can be modified by the function. The *SSCHA_Minimizer* calls *custom_function_gradient* immediately before emplying the gradient to generate the dyanmical matrix and the structure for the next iteration. Therefore, *custom_function_gradient* is employed to apply costraints, projecting the gradients in the desidered subspace.

In particular, *CFG_ProjectOnModes* projects the gradient of the dynamical matrix into the subspace defined only by the mode branches selected with *ModeLock.SetupFreeModes*. As done for *custom_function_post*, also here we can define a custom function instead of using the predefined one provided by the *Utilities* module.

The following code limit the projection on the subspace of modes only on the fourth q-point of the dynamical matrix.

```python
iq = 4
def my_constrain(dyn_gradient, structure_gradient):
        # Let us apply the standard constrain on modes
        ModeLock.CFG_ProjectOnModes(dyn_gradient, structure_gradient)

        # Now we set to zero the gradient of the dynamical matrix if it does
→not belong to the iq-th q point (ordered as they appear in the dynamical
→matrix used to initialize the minimization).

        nq, nat3, nat3_ = dyn_gradient.shape
        for i in range(nq):
                if i != iq:
                        dyn_gradient[i, :, :] = 0


# [...] define minim as the SSCHA_Minimizer
minim.run(custom_function_gradient = my_constrain)
```

The two arguments taken by custom_function_gradient are the gradient of the dynamical matrix of size (nq, 3*nat, 3*nat) and the gradient of the structure of size (nat, 3). Notice also how, inside *my_constrain*, we call *ModeLock.CFG_ProjectOnModes*. You can concatenate many different custom functions following this approach.

Remember that the gradients are numpy arrays; **you must modify their content accessing their memory using the slices** [x,y,z] as we did. In fact, if you overwrite the pointer to the memory (defining a new array), the content of the gradient will not be modified outside the function. In the following code we show an example of correct and wrong.

```python
# This puts the gradient to zero
dyn_gradient[:,:,:] = 0   # CORRECT

# This does not put to zero the gradient
dyn_gradient = np.zeros( (nq, 3*nat, 3*nat))   # WRONG
```

In particular, the second expression redefines the name *dyn_gradient* only inside the function, allocating new memory on a different position, and overwriting the name *dyn_gradient* only inside the function to point to this new memory location. It **does not** write in the memory where *dyn_gradient* is stored: the gradient outside the function is unchanged.

Indeed, you can also constrain the structure gradient. The ModeLocking class provides a function also to constrain the atomic displacement to follow the lattice vibrations identified by the selected branches at gamma. This is *ModeLock.CFG_ProjectStructure*. If you want to constrain both the dynamical matrix and the structure, you can simply concatenate them as:

```python
def my_constrain(dyn_grad, structure_grad):
        ModeLock.CFG_ProjectOnModes(dyn_grad, structure_grad)
        ModeLock.CFG_ProjectStructure(dyn_grad, structure_grad)

# [...]
minim.run(custom_function_gradient = my_constrain)
```

Resuming, *custom functions* can be used to inject your personal code inside each SSCHA iteration. Proper use of this function gives you full control over the minimization and allows you to personalize the SSCHA without editing the source code.

FREQUENTLY ASKED QUESTIONS (FAQS)

Here we answer to most common question we received.

5.1 Setup the calculation

5.1.1 How do I start a calculation if the Dynamical matrices have imaginary frequencies?

A good starting point for a sscha minimization are the dynamical matrix obtained from a harmonic calculation. However, they can have imaginary frequencies. This may be related to both instabilities (the structure is a saddle-point of the Born-Oppenheimer energy landscape) or to a not well-converged choice of the parameters for computing the harmonic frequencies. In both cases, it is very easy to get a new dynamical matrix that is positive definite and can be used as a starting point. An example is made in Turorial on H3S. Assuming your not positive definite dynamical matrix is in Quantum Espresso format "harm1" ... "harmN" (with N the number of irreducible q points), you can generate a positive definite dynamical matrix "positive1" ... "positiveN" with the following python script that uses CellConstructor.

```python
# Load the cellconstructor library
import cellconstructor as CC
import cellconstructor.Phonons

# Load the harmonic not-positive definite dynamical matrix
# We are reading 6 dynamical matrices
harm = CC.Phonons.Phonons("harm", nqirr = 6)

# Apply the acoustic sum rule and the symmetries
harm.Symmetrize()

# Force the frequencies to be positive definite
harm.ForcePositiveDefinite()

# Save the final dynamical matrix, ready to be used in a sscha run
harm.save_qe("positive")
```

The previous script (that we can save into *script.py*) will generate the positive definite matrix ready for the sscha run. It may be executed with

```
$ python script.py
```

5.1.2 What are the reasonable values for the steps?

Starting from version 1.2, the line minimization is implemented. This means that there is no need to specify the value of the minimization step as the code will automatically find it.

However, if the code takes too long to get a good timestep at the begining of a calculation (especially at the very first iteration or if few configurations are employed), you could speedup the calculation providing a smaller initial guess than the default one (1). This is done in the python script by calling the function

```
minim.set_minimization_step(0.1)
```

Where minim is the sscha.SchaMinimizer.SSCHA_Minimizer class. You can select the step also in the namespace input by setting the following variables in the inputscha namespace

lambda_w is the step in the atomic positions (stand-alone program input).

lambda_a is the step in the dynamical matrix (stand-alone program input).

5.1.3 In a NPT or NVT with variable lattice, what is a reasonable value for the bulk modulus?

The bulk modulus is just an indicative parameter used to guess the optimal step of the lattice parameters to converge as quickly as possible. It is expressed in GPa. You can find online the bulk modulus for many materials. Find a material similar to the one you are studying and look if there is in literature a bulk modulus.

The default value is good for most case (equal to 100), but it could be too low for very hard materials (like diamond, which is 500 GPa, or high-pressure stuff). If you are not sure, it is safer to choose an higher value of the bulk modulus, as the code is going to optimize it during the simulation anyway.

If you have no idea on the bulk modulus, you can easily compute them by doing two static *ab initio* calculations at very close volumes (by varying the cell size), and then computing the differences between the pressure:

$$B = -\Omega \frac{dP}{d\Omega}$$

where Ω is the unit-cell volume and P is the pressure (in GPa).

5.1.4 It is always good to run NVT before any NPT simulation?

In general, it is good to have a reasonable dynamical matrix before starting with a relaxation with variable cell (vc_relax). Therefore, to avoid mooving the volume upward and backward, always start with a NVT simulation with fixed lattice (the relax method of SSCHA class) and then run a NPT or a NVT with variable lattice (vc_relax method), starting from the static lattice solution.

5.1.5 How may I run a calculation neglecting symmetries?

You can tell the code to neglect symmetries with the `neglect_symmetries = .true.` flag. In the python script, this is done setting the attribute *neglect_symmetries* of sscha.SchaMinimizer.SSCHA_Minimizer to False.

5.1.6 In which units are the lattice vectors, the atomic positions, and the mass of the atoms in the dynamical matrix file?

The dynamical matrix follows the quantum espresso units. They are Rydberg atomic units (unit of mass is 1/2 the electron mass, energy is Ry, positions are in Bohr. However, espresso may have an ibrav not equal to zero (the third number in the header of the dynamical matrix). In this case, please, refer to the espresso ibrav guide in the *PW.x input description <https://www.quantum-espresso.org/Doc/INPUT_PW.html#idm199>*

5.1.7 What is the difference between different kinds of minimization (preconditioning and root_representation)?

You do not need to worry about these parameters, as starting from version 1.2 the code has a robust workflow that should avoid bothering you with these details. However, if you are curious and want to know a bit more on the details here it is the explanation: We provide three different advanced algorithms called in **root_representation**, that can be either **normal**, or **sqrt**, or **root4** (inside &inputscha namespace or the SSCHA_Minimizer object) In this way, instead of minimizing the Φ matrix, we minimize with respect to $\sqrt{\Phi}$ or $\sqrt[4]{\Phi}$. Therefore the new dynamical matrix is constrained in a space that is positive definite. Moreover, it has been proved that $\sqrt[4]{\Phi}$ minimization has a better condition number than the original one and thus should reach the minimum faster.

Alternatively, a similar effect to the speedup in the minimization obtained with **root4** is possible using the preconditioning (by setting **preconditioning** or **precond_dyn** to True in the input file or the python script, respectively). This way also the single minimization step runs faster, as it avoids passing in the root space of the dynamical matrix (but indeed, you can have imaginary frequencies).

Since the gradient computation is much slower (especially for a system with more than 80 atoms in the supercell) without the preconditioning, it is possible to combine the preconditioning with the root representation to have a faster gradient computation and to be guaranteed that the dynamical matrix is positive definite by construction at each step. However, in this way the good condition number obtained by the preconditioning (or the root4 representation) is spoiled. For this reason, when using the preconditioning, avoid using **root4**, and chose instead **sqrt** as root_representation.

The default values are:

```
&inputscha
   root_representation = "normal"
   preconditioning = .true.
&end
```

or in python

```python
# The ensemble has been loaded as ens
minim = sscha.SchaMinimizer.SSCHA_Minimizer(ens)
```

```
minim.root_representation = "normal"
minim.precond_dyn = True
```

5.1.8 How do I fix the random number generator seed to make a calculation reproducible?

As for version 1.2, this can be achieved only by using the python script. Since python uses NumPy for random numbers generation, you can, at the beginning of the script that generates the ensemble, use the following:

```python
import numpy as np

X = 0
np.random.seed(seed = X)
```

where X is the integer used as a seed. By default, if not specified, it is initialized with None that it is equivalent to initializing with the current local time.

5.2 On error and convergence of the free energy minimization

5.2.1 The code stops saying it has found imaginary frequencies, how do I fix it?

Update python-sscha to version 1.2 (at least)! This should be fixed.

If you do not want to update the code, set

```
minim.root_representation = 'root2'
```

This way the minimization strategy changes and it is mathematically impossible to get imaginary frequencies. The same option can be activated within the namespace input

```
&inputscha
    root_representation = 'root2'
&end
```

5.2.2 Why the gradient sometimes increases during a minimization?

Noting in principle assures that a gradient should always go down. It is possible at the beginning of the calculation when we are far from the solution that one of the gradients increases. However, when we get closer to the solution, indeed the gradient must decrease. If this does not happen it could be due to the ensemble that has fewer configurations than necessary. In this case, the good choice is to increase the number of effective sample size (the Kong-Liu ratio), to stop the minimization when the gradient starts increasing, or to increase the number of configurations in the ensemble.

In any case, what must decrease is free energy. If you see that the gradient is increasing but the free energy decreases, then the minimization is correct. However, if both the gradient and free energy are increasing, something is wrong, and you may require more configurations in each iteration. This is especially true for system with few symmetries (or big primitive cells).

5.2.3 How do I check if my calculations are well converged?

In general, if the gradient goes to zero and the Kong Liu ratio is above 0.5 probably your calculation converged very well. This means that when your calculation stops because it converged (not because it runs out of iterations), then it should be well converged.

There are some cases (especially in systems with many atoms) in which it is difficult to have an ensemble sufficiently big to reach this condition. In these cases, you can look at the history of the frequencies in the last populations (there is a drift or random fluctuations?)

5.2.4 What is the final error on the structure or the dynamical matrix of a SCHA minimization?

To test the error, you can split the ensemble in two half and repeat the last minimization. Then check at the difference between the result to have a rough estimation of the fluctuations.

To split the ensemble, refer to the FAQ *How do I split the ensemble?*.

5.2.5 How do I understand if the free energy hessian calculation is converged?

The free energy hessian requires much more configurations than the SCHA minimization. First of all, to run the free energy Hessian, the SSCHA minimization must end with a gradient that can be decreased indefinitely without decreasing the KL below 0.7 /0.8. Then you can estimate the error by repeating the hessian calculation with half of the ensemble and check how the frequencies of the hessian changes. This is also a good check for the final error on the frequencies.

You can split your ensemble in two by using the split function.

To split the ensemble, refer to the FAQ *How do I split the ensemble?*.

5.2.6 How do I split the ensemble?

After you load or compute an ensemble you can split it and select only a portion of it to run the code.

```
# Assuming you loaded or computed the ensemble inside
# ensemble

# Let us create a mask that selects only the first half of the ensemble
first_half_mask = np.zeros(ensemble.N, dtype = bool)
first_half_mask[:ensemble.N//2] = True

# Now we pass the mask to the ensemble to extract a new one
# Containing only the configurations that correspond to the True
# values of the mask
first_half_ensemble = ensemble.split(first_half_mask)
```

After this code, the varialbe first_half_ensemble is a sscha.Ensemble.Ensemble that can be used for any caluclation.

5.2.7 How can I add more configurations to an existing ensemble?

You can use the split and merge functions of the Ensemble class. First of all you generate a new ensemble, you compute the energy and force for that ensemble, then you merge it inside another one.

```python
# Load the original ensemble (first population with 1000 configurations)
ens = sscha.Ensemble.Ensemble(dynmat, T, dynmat.GetSupercell())
ens.load("data_dir", population = 1, N = 1000)

# Generate a new ensemble with other 1000 configurations
new_ensemble = sscha.Ensemble.Ensemble(dynmat, T, dynmat.GetSupercell())
new_ensemble.generate(1000)

# Compute the energy and forces for the new ensemble
# For example in this case we assume to have initialized 'calc' as an ASE
→calculator.
# But you can also save it with a different population,
# manually compute energy and forces, and then load again the ensemble.
new_ensemble.get_energy_forces(calc)

# Merge the two ensembles
ens.merge(new_ensemble)

# Now ens contains the two ensembles. You can save it or directly use it for
→a SSCHA calculation
ens.save("data_dir", population = 2)
```

Indeed, to avoid mistakes, when merging the ensemble you must be carefull that the dynamical matrix and the temperature used to generate both ensembles are the same.

5.2.8 How does the error over the gradients scale with the number of configurations?

The error scales as any stochastic method, with the inverse of the square root of the number of configurations. So to double the accuracy, the number of configurations must be multiplied by 4.

5.2.9 I cannot remove the pressure anisotropy after relaxing the cell, what is happening?

Variable cell calculation is a tricky algorithm. It could be that your bulk modulus is strongly anisotropic, so the algorithm has difficulties in optimizing well. In general, the stress tensor is also affected by the stochastic error, so it is impossible to completely remove anisotropy. However, a converged result is one in which the residual anisotropy in the stress tensor is comparable to the stochastic error on the stress tensor. If you are not able to converge, you can either increase the number of configurations, modify the bulk_modulus parameter (increase it if the stress change too much between two populations, decrease it if it does not changes enough) or fix the overall volume (by using the fix_volume flag in the &relax namespace or the vc_relax method if you are using the python script).

Fixing the volume improves the convergence of the variable cell algorithm (using the fix_volume = True argument of the vc_relax method).

5.2.10 How do I choose the appropriate value of Kong-Liu effective sample size or ratio?

The Kong-Liu (KL) effective sample size is an estimation of how good is the extracted set of configurations to describe the BO landscape around the current values of the dynamical matrix and the centroid position. After the ensemble is generated, the KL sample size matches with the actual number of configurations, however, as the minimization goes, the KL sample size is reduced. The code stops when the KL sample size is below a certain threshold.

The default value for the Kong-Liu threshold ratio is 0.5 (effective sample size = 0.5 the original number of configurations). This is a good and safe value for most situations. However, if you are very far from the minimum and the gradient is big, you can trust it even if it is very noisy. For this reason, you can lower the Kong-Liu ratio to 0.2 or 0.1. However, notice that by construction the KL effective sample size is always bigger than 2. Therefore, if you use 10 configurations, and you set a threshold ratio below 0.2, you will never reach the threshold, and your minimization will continue forever (going into a very bad regime where you are minimizing something completely random). On the other side, on some very complex systems close to the minimum, it could be safe to increase the KL ratio even at 0.6.

5.3 Post-processing the output

5.3.1 How do I plot the phonon dispersion after the calculation?

See *Plot the phonon dispersion* section.

5.3.2 How do I plot the frequencies of the dynamical matrix during the optimization?

To check if the SSCHA is converging, you should plot the dynamical matrix's frequencies during the minimization. In particular, you should look if, between different populations, the evolution of each frequency is consistent. If it seems that frequencies are evolving randomly from a population to the next one, you should increase the number of configurations, otherwise, you can keep the number fixed.

The code can print the frequencies at each step. If you run the code with an input script, you should provide in the &utils tag the filename for the frequencies:

```
&utils
    save_frequencies = "minim_info"
&utils
```

You can use the same function from the python script by calling a custom function that saves the frequencies after each optimization step. The Utilities module of the SSCHA offers this function:

```
IO_freq = sscha.Utilities.IOInfo()
IO_freq.SetupSaving("minim_info")

# Initialize the minimizer as minim [...]
minim.run(custom_function_post = IO_freq.CFP_SaveAll)
```

Then, while running you can plot all the information about the minimization with:

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

And you will see both frequencies, free energy, gradients and everything how␣
↪it evolves during the
minimization.

If you are using a version older than 1.2, the previous command should be␣
↪replaced with:
```

```
$ plot_frequencies.py minim_info
```

If you restart the calculation and save it in multiple files, you can concatenate the results with:

```
$ sscha-plot-data.py minim_info1 minim_info2 ...
```

5.4 Constrains and custom minimization

5.4.1 How do I lock modes from m to n in the minimization?

Constrains to the minimization within the mode space may be added in both the input file (for the stand-alone execution) and in the python script. In the input script, inside the namespace **&utils**, you should add:

mu_free_start = 30 and **mu_free_end = 36** : optimize only between mode 30 and 36 (for each q point).

You can also use the keywords **mu_lock_start** and **mu_lock_end** to freeze only a subset of modes.

You can also choose if you want to freeze only the dynamical matrix or also the structure relaxation along with those directions, by picking:

project_dyn = .true. and **project_structure = .false.**. In this way, I freeze only the dynamical matrix along with the specified modes, but not the structure.

Modes may be also locked within the python scripting. Look at the LockModes example in the Examples directory.

TODO: Add the same guide for the python code

5.4.2 How do I lock a special atom in the minimization?

More complex constraints may be activated in the minimization, but their use is limited within the python scripting. You can write your constraining function that will be applied to the structure gradient or the dynamical matrix gradient. This function should take as input the two gradients (dynamical matrix and structure) and operate directly on them. Then it can be passed to the minimization engine as *custom_function_gradient*.

```python
LIST_OF_ATOMS_TO_FIX = [0, 2, 3]
def fix_atoms(gradient_dyn, gradient_struct):
    # Fix the atoms in the list
    gradient_struct[LIST_OF_ATOMS_TO_FIX, :] = 0
```

(continues on next page)

```
minim.run( custom_function_gradient = fix_atoms )
```

Here, `minim` is the `SSCHA_Minimizer` class. In this case, we only fix the structure gradient. However, the overall gradient will have a translation (acoustic sum rule is violated). Be very careful when doing this kind of constrains, and check if it is really what you want.

A more detailed and working example that fixes also the degrees of freedom of the dynamical matrix is reported in the FixAtoms example.

THE API

This chapter contains the documentation for the main methods of the python-sscha code. It can be used both by advanced users, that wants to exploit python-sscha as a library, or developers, willing to add new features to the code (or adapt existing ones for their purposes).

The API is divided into Modules.

6.1 The Ensemble Module

This module deals with the ensembles of configurations. It is used to generate random configurations from the dynamical matrix, to compute observables on the ensemble used in the SSCHA optimization. These include the average force on atoms, the gradient of the SSCHA minimization, the quantum-thermal stress tensor, as well as properties of the ensemble, like reweighting.

class sscha.Ensemble.**Ensemble**(*dyn0*, *T0*, *supercell=None*, ***kwargs*)

> **compute_ensemble**(*calculator*, *compute_stress=True*, *stress_numerical=False*, *cluster=None*, *verbose=True*, *timer=None*)
>
> > This is the generic function to compute forces and stresses. It can be used both with clusters, and with simple ase calculators
> >
> > > **calculator:**
> > > > The ase calculator
> > >
> > > **compute_stress: bool**
> > > > If true compute the stress
> > >
> > > **stress_numerical**
> > > > [bool] Compute the stress tensor with finite difference, this is not possible with clusters
> > >
> > > **cluster: Cluster, optional**
> > > > The cluster in which to send the calculation. If None the calculation is performed on the same computer of the sscha code.
>
> **convert_units**(*new_units*)
>
> > This function is used to jump between several unit of measurement. You should always call this function before processing data assuming a particular kind of units.
> >
> > **Supported units are:**

- **"default" :**
    This is the default units. Here the forces are Ry/A displacements and structure are in A Dynamical matrix is in Ry/bohr^2. Mass is in Ry units

- **"hartree" :**
    Here, everything is stored in Ha units.

- **new_units**
    [string] The target units

generate(*N*, *evenodd=True*, *project_on_modes=None*, *sobol=False*, *sobol_scramble=False*, *sobol_scatter=0.0*)

This subroutine generates the ensemble from dyn0 and T0 setted when this class is created. You still need to generate the forces for the configurations.

N
[int] The number of random configurations to be extracted

evenodd
[bool, optional] If true for each configuration also the opposite is extracted

project_on_modes
[ndarray(size=(3*nat_sc, nproj)), optional] If different from None the displacements are projected on the given modes.

sobol
[bool, optional (Default = False)] Defines if the calculation uses random Gaussian generator or Sobol Gaussian generator.

sobol_scramble
[bool, optional (Default = False)] Set the optional scrambling of the generated numbers taken from the Sobol sequence.

sobol_scatter
[real (0.0 to 1) (Deafault = 0.0)] Set the scatter parameter to displace the Sobol positions randommly.

get_average_energy(*subtract_sscha=False*, *return_error=False*)

This is the average of the energy

$$\langle E \rangle = \frac{1}{N} \sum_{i=1}^{N} E_i \rho_i$$

where ρ_i is the ratio between the probability of extracting the configuration i with the current dynamical matrix and with the dynamical matrix used to extract the ensemble.

subtract_sscha
[bool, optional, default False] If true, the average difference of energy respect to the sscha one is returned. This is good, because you can compute analytically the sscha energy and sum it on an infinite ensembe. Do in this way to suppress the stochastic noise.

return_error
[bool, optional, default False] If true also the error is returned as a second value

Example where ensemble is a correctly initialized self variable

```
>>> energy = ensemble.get_average_energy()
```

The following example return also the stochastic error >>> energy, error_on_energy = ensemble.get_average_energy(return_error = True)

get_average_forces(*get_error*, *in_unit_cell=True*)

This is the average of the forces that acts on the atoms

$$\left\langle \vec{F} \right\rangle = \frac{1}{N} \sum_{i=1}^{N} \vec{F}_i \rho_i$$

where ρ_i is the ratio between the probability of extracting the configuration i with the current dynamical matrix and with the dynamical matrix used to extract the ensemble.

- **get_error**

    [bool] If true the error is also returned (as get_free_energy).

- **in_unit_cell**

    [bool, optional] If True (default True) the mean force is averaged on all the atoms in the supercell, then it returns the forces that acts on the unit cell atoms only.

get_average_stress()

This gets only the ab-initio average of the stress tensor

$$P_{\alpha\beta} = \langle P_{\alpha\beta} \rangle$$

get_covmat_from_ensemble()

This method is for testing, allows to use the ensemble to evaluate the covariance matrix stochastically. It should be equal to the matrix Upsilon^-1 that is obtained with the GetUpsilonMatrix method from the Phonons package.

$$\Upsilon_{ab}^{-1} = \langle u_a u_b \rangle$$

cov_mat

    [3nat x 3nat, ndarray] A numpy matrix of the covariance matrix.

get_effective_sample_size()

Get the Kong-Liu effective sample size with the given importance sampling.

get_energy_forces(*ase_calculator*, *compute_stress=True*, *stress_numerical=False*, *skip_computed=False*, *verbose=False*, *timer=None*)

This subroutine uses the ase calculator to compute the abinitio energies and forces of the self ensemble. This subroutine requires to have ASE installed and properly configured to interface with your favourite ab-initio software.

ase_calculator

    [ase.calculator] The ASE interface to the calculator to run the calculation. also a CellConstructor calculator is accepted

compute_stress

    [bool] If true, the stress is requested from the ASE calculator. Be shure that the calculator you provide supports stress calculation

---

stress_numerical
> [bool] If the calculator does not support stress, it can be computed numerically by doing finite differences.

skip_computed
> [bool] If true the configurations already computed will be skipped. Usefull if the calculation crashed for some reason.

get_fc_from_self_consistency(*subtract_sscha=False*, *return_error=False*)

> This function evaluate the self consistent scha equation. This can be used to evaluate the goodness of the minimization procedure, as well as an independent minimizer.

$$\Phi_{ab} = \frac{1}{2} \sum_c \Upsilon_{ac} \langle u_c f_a \rangle_\Phi$$

> The previous equation is true only if the Φ matrix is the solution of the SCHA theory. Here :math:`

ec u` are the displacements of the configurations
> and f are the forces of the real system acting on the simulation.

> **subtract_sscha**
> > [bool, optional] This is an optional parameter, if true the forces used to evaluate the new force constant matrix are subtracted by the sscha forces. This means that the result is a gradient of the new matrix with respect to the old one.

> **return_error**
> > [bool, optional] If true also the stochastic error is returned.

> **fc**
> > [ndarray (3*nat x 3*nat)] The real space force constant matrix obtained by the self-consistent equation.

get_fourier_forces(*get_error=True*)

> This method is the same as before, but it exploits the fourier transform to avoid doing the average once more.

get_fourier_gradient(*timer=None*)

> This subroutine computes the gradient in foureir space

> It employes the acceleration available thanks to the julia code.

> In brief, the calculation performs:

$$\langle u(q)\delta f(-q) \rangle$$

> To get the gradient in the q space. The method can be easily parallelized, since the julia subroutine returns also the average of the square.

> - **timer**
>   > [Timer object, optional] If present, the time spent in the julia code is added to the timer.

> - **grad**
>   > [ndarray (3*nat x 3*nat)] The gradient in the fourier space.

- **error_grad**

    [float] The error of the gradient. It does not count the symmetries

get_free_energy(*return_error=False*)

Obtain the SSCHA free energy for the system. This is done by integrating the free energy along the hamiltonians, starting from current_dyn to the real system.

The result is in Rydberg

$$\mathcal{F} = \mathcal{F}_0 + \int_0^1 \frac{d\mathcal{F}_\lambda}{d\lambda} d\lambda$$

Where λ is the parameter for the adiabatic integration of the hamiltonian.

$$H(\lambda) = H_0 + (H - H_0)\lambda$$

here H_0 is the sscha harmonic hamiltonian, while H_1 is the real hamiltonian of the system.

return_error

[bool, optional, default False] If true also the error is returned as a second value.

float

The free energy in the current dynamical matrix and at the ensemble temperature

get_free_energy_hessian(*include_v4=False*, *get_full_hessian=True*, *verbose=False*, *use_symmetries=True*, *return_d3=False*, *w_pols=None*, *timer=None*)

This subroutines computes the odd correction to the free energy hessian using the fortran subroutines, as describe in the Bianco paper …

The calculation is performed in the supercell

include_v4

[bool] If True we include the fourth order force constant matrix. This requires a lot of memory

get_full_hessian

[bool] If True the full hessian matrix is returned, if false, only the correction to the SSCHA dynamical matrix is returned.

verbose

[bool] If true, the third order force constant tensor is written in output [Ha/bohr^3 units]. This can be used to interpolate the result on a bigger mesh with cellconstructor.

use_symmetries

[bool] If true, the d3 and d4 are symmetrized in real space. It requires that spglib is installed to detect symmetries in the supercell correctly.

return_d3

[bool] If true, returns also the tensor of three phonon scattering.

w_pols

[list of ndarray] If provided, it avoids the diagonalization of the dynamical matrix to speedup the time.

phi_sc

[Phonons()] The dynamical matrix of the free energy hessian in (Ry/bohr^2)

d3

[ndarray (size = (3*nat_sc, 3*nat_sc, 3*nat_sc), Optional] Return the three-phonon-scattering tensor (in Ry atomic units). Only if return_d3 is True.

get_free_energy_interpolating(*target_supercell*, *support_dyn_coarse=None*, *support_dyn_fine=None*, *error_on_imaginary_frequency=True*, *return_error=False*)

This is a trick to interpolate the free energy in the infinite volume limit.

Note, this function report the free eenrgy in the primitive cell, while the method get_free_energy returns the energy in the supercell.

target_supercell

[list (N, N, N)] A list of three indices, where N is the dimension of the target supercell on which you want to interpolate.

support_dyn[coarse/fine]

[Phonons() Optional] The harmonic dynamical matrix in the current/target_supercell This is optional, it can be used to achieve a better interpolation. If provided only the difference between the harmonic dyn and the current dyn is interpolated.

error_on_imaginary_frequency

[bool] If Fase (default True) it will ignore imaginary frequencies arising from the interpolation. Otherwise an exception will be raised.

return_error

[bool] As the normal get_free_energy, if this flag is True, the stochastic error is returned.

free_energy

[float] The free energy in the unit_cell volume [in Ry]. Note. This free energy is rescaled on the unit cell volume, it is a different behaviour with respect to get_free_energy.

error_on free energy

[float] The stochastic error, it is returned only if requested.

get_noncomputed()

Get another ensemble with only the non computed configurations. This may be used to re-submit only the non computed values

get_odd_realspace()

This is a testing function to compute the odd3 correction using the real space v3 (similar to the raffaello first implementation)

get_preconditioned_gradient(*subtract_sscha=True*, *return_error=False*, *use_ups_supercell=True*, *preconditioned=1*, *fast_grad=False*, *verbose=True*, *timer=None*)

This function evaluate the self consistent scha equation. This can be used to evaluate the

goodness of the minimization procedure, as well as an independent minimizer. This is the same as get_fc_from_self_consistency, but works also with supercell

$$\Phi_{ab} = \sum_c \Upsilon_{ac} \langle u_c f_a \rangle_\Phi$$

The previous equation is true only if the Φ matrix is the solution of the SCHA theory. Here $\vec{u}$ are the displacements of the configurations and f are the forces of the real system acting on the simulation.

NOTE: It does not takes into account for the symmetrization.

subtract_sscha
[bool, optional] This is an optional parameter, if true the forces used to evaluate the new force constant matrix are subtracted by the sscha forces. This means that the result is a gradient of the new matrix with respect to the old one.

return_error
[bool, optional] If true also the stochastic error is returned.

use_ups_supercell
[bool, optional] If true the gradient is computed enterely in real space, and then transformed with fourier in q space. This is computationally heavier, but can be used to test if everything is working correctly. For now this flag is ignored and always True.

preconitioned
[int, optional] If 1 (default) the gradient is returned multiplied by the preconditioned, otherwise it is returned as it should be.

fc
[ndarray (nq x 3*nat x 3*nat)] The real space force constant matrix obtained by the self-consistent equation.

get_preconditioned_gradient_parallel(*args*, *timer=None*, ***kwargs*)
Compute the gradient using multiprocessing. For documentation, see get_preconditioned_gradient

get_stress_tensor(*offset_stress=None*, *use_spglib=False*)
The following subroutine computes the anharmonic stress tensor calling the fortran code get_stress_tensor. Note that the stress tensor is symmetrized to satisfy the cell constraint.

NOTE: unit of measure is Ry/bohr^3 to match the quantum espresso one

offset_stress
[3x3 matrix, optional] An offset stress to be subtracted to the real stress tensor. Usefull if you want to compute just the anharmonic contribution.

use_spglib
[bool] If true use the spglib library to perform the symmetrization

stress_tensor
[3x3 matrix] The anharmonic stress tensor obtained by averaging both the ab-initio stresses and correcting with the sscha non-linearity.

err_stress
[3x3 matrix] The matrix of the error on the stress tensor.

init()

> Call this function after the ensemble is computed. It performs the fourier transform of the forces and displacements.
>
> It also computed the displacements and all other important quantities that needs to be iterated to actually run a sscha minimization
>
> And initializes the parameters to perform the fourier transform faster at each minimization steps

init_from_structures(*structures*)

> Initialize the ensemble from the given list of structures
>
> > **structures**
> >
> > > [list of structures] The list of structures used to initialize the ensemble

init_q_opposite()

> Setup the inverse q points.
>
> This subroutine identifies the inverse of each q point from the dynamical matrix

load(*data_dir*, *population*, *N*, *verbose=False*, *load_displacements=True*, *raise_error_on_not_found=False*, *load_noncomputed_ensemble=False*, *timer=None*)

> This function load the ensemble from a standard calculation.
>
> The files need to be organized as follows
>
> data_dir / scf_populationX_Y.dat data_dir / energies_supercell_populationX.dat data_dir / forces_populationX_Y.dat data_dir / pressures_populationX_Y.dat data_dir / u_populationX_Y.dat
>
> X = population Y = the configuration id (starting from 1 to N included, fortran convention)
>
> The files scf_population_X_Y.dat must contain the scf file of the structure. It should be in alat units, matching the same alat defined in the starting dynamical matrix.
>
> The energies_supercell.dat file must contain the total energy in Ry for each configuration.
>
> The forces_populationX_Y contains the
>
> > **data_dir**
> >
> > > [str] The path to the directory containing the ensemble. If you used the fortran sscha.x code it should match the data_dir option of the input file.
> >
> > **population**
> >
> > > [int] The info to distinguish between several ensembles generated in the same data_dir. This also should match the correspective property of the fortran sscha.x input file.
> >
> > **N**
> >
> > > [int] The dimension of the ensemble. This should match the n_random variable from the fortran sscha.x input file.
> >
> > **verbose**
> >
> > > [bool, optional] If true (default false) prints the real timing of the different part during the loading.
> >
> > **load_displacement: bool**
> >
> > > If true the structures are loaded from the u_populationX_Y.dat files, otherwise they are loaded from the scf_populationX_Y.dat files.

> **raise_error_on_not_found**
>> [bool] If true, raises an error if one force file is missing
>
> **load_noncomputed_ensemble: bool**
>> If True, it allows for loading an ensemble where some of the configurations forces and stresses are missing. Note that it must be compleated before running a SCHA minimization

load_bin(*data_dir*, *population_id=1*, *avoid_loading_dyn=False*, *timer=None*)

> This function loads the ensemble saved with save_bin(. . .)
>
>> **data_dir**
>>> [string] The directory containing the ensemble
>>
>> **population_id**
>>> [int] The esnemble population identifier.
>>
>> **avoid_loading_dyn**
>>> [bool] If true, the dynamical matrix is not loaded.

load_from_calculator_output(*directory*, *out_ext='.pwo'*, *timer=None*)

> This subroutine allows to directly load the ensemble from the output files of a calculation. This works and has been tested for quantum espresso, however in principle any output file from an ase supported format should be readed.
>
> NOTE: This subroutine requires ASE to be correctly installed.
>
>> **directory**
>>> [string] Path to the directory that contains the output of the calculations
>>
>> **out_ext**
>>> [string] The extension of the files that will be readed.

merge(*other*)

> This function will merge two ensembles together.
>
>> **other**
>>> [Ensemble()] Another ensemble to be merge with. It must be generated by the same dynamical matrix as this one, otherwise wired things will happen.

remove_noncomputed()

> Removed all the incomplete calculation from the ensemble. It may be used to run a minimization even if the ensemble was not completely calculated.

save(*data_dir*, *population*, *use_alat=False*)

> This function saves the ensemble in a way the original fortran SSCHA code can read it. Look at the load function documentation to see clearly how it is saved.
>
> NOTE: This method do not save the dynamical matrix used to generate the ensemble (i.e. self.dyn_0) remember to save it separately to really save all the info about the ensemble.
>
>> **data_dir**
>>> [string] Path to the directory in which the data will be saved. If it does not exists, it will be created
>>
>> **population**
>>> [int] The id of the population, usefull if you want to save more ensemble in the same data_dir without overwriting the data.

---

use_alat

> [bool] If true the scf_populationX_Y.dat files will be saved in alat units, as specified by the dynamical matrix. Also the unit cell will be omitted. This is done to preserve retrocompatibility with ensembles generated by older versions of the sscha code

save_bin(*data_dir*, *population_id=1*)

> This function is a fast way of saving the ensemble. It is faster and make use of less disk space than the save. The drawback is that can only be opened with numpy

data_dir

> [string] path to the folder in which the ensemble is saved

population_id

> [int] The id of the population. This can be used to save several ensembles in the same data_dir

save_enhanced_xyz(*filename*, *append_mode=True*, *stress_key='stress'*, *forces_key='forces'*, *energy_key='energy'*)

> Save the ensemble as an enhanced xyz.
>
> This is the default format for training the GAP potentials with quippy.

filename

> [string] Path to the xyz file in which to save.

append_mode

> [bool] If true, does not overwrite the previous existing file, but append the ensemble on the bottom. This is the way to concatenate easily more ensembles.

save_extxyz(*filename*, *append_mode=True*)

> ASE extxyz format is used for build the training set for the nequip and allegro neural network potentials.
>
> Note, this is the same as save_enhanced_xyz, but it uses ASE builtin function. If ase is not found, it will use the save_enhanced_xyz function.

filename

> [str] The path to the .extxyz file containing the ensemble

append_mode: bool

> If true the ensemble is appended

save_raw(*root_directory*, *type_dict=None*)

> Save the ensemble as a set of raw files.
>
> This is the default format for training with deepmd

filename

> [string] The directory on which to save the ensemble. If it does not exist, it is create. NOTE: this will overwrite any other ensemble saved in raw format in that directory

type_dict

> [dict] The dictionary between integers and atomic types. If not provided, it is generated on the spot and returned.

> **type_dict**
>> [dict] The dictionary of the parameters

split(*split_mask*)

> This method will return an ensemble with only the configurations matched by the split_mask array. NOTE: The original ensemble will remain untouched.
>
>> **split_mask**
>>> [ndarray(size = self.N, dtype = bool)] A mask array. It must be of the same size of the number of configurations, and contain a True or False if you want that the corresponding configuration to be included in the splitted ensemble
>>
>> **splitted_ensemble**
>>> [Ensemble()] An ensemble tath will contain only the configurations in the split mask.

update_displacements(*new_structure*)

> Update the displacement vector in fourier space using the new structure.

update_weights(*new_dynamical_matrix*, *newT*, *update_q=False*, *timer=None*)

> This function updates the importance sampling for the given dynamical matrix. The result is written in the self.rho variable
>
>> **new_dynamical_matrix**
>>> [CC.Phonons.Phonons()] The new dynamical matrix on which you want to compute the averages.
>>
>> **new_T**
>>> [float] The new temperature.
>>
>> **update_q**
>>> [bool] If false the q_vectors are not updated. This is required for some methods and application, but not for standard minimization. Since it is the most time consuming part, it can be safely avoided.

update_weights_fourier(*new_dynamical_matrix*, *newT*, *timer=None*)

> This function updates the importance sampling for the given dynamical matrix. The result is written in the self.rho variable
>
>> **new_dynamical_matrix**
>>> [CC.Phonons.Phonons()] The new dynamical matrix on which you want to compute the averages.
>>
>> **new_T**
>>> [float] The new temperature.

6.2 The SchaMinimizer Module

This module is the main SSCHA minimizer. It allows us to set up a single (one population) minimization. In this module, the minimization algorithm is introduced, as well as stopping conditions and all the parameters usually located in the &inputscha name list are read.

class sscha.SchaMinimizer.**SSCHA_Minimizer**(*ensemble=None*, *root_representation='normal'*, *kong_liu_ratio=0.5*, *meaningful_factor=0.2*, *minimization_algorithm='sdes'*, *lambda_a=1*, *timer=None*, ***kwargs*)

> **check_imaginary_frequencies**(*timer=None*)
>
> > The following subroutine check if the current matrix has imaginary frequency. In this case the minimization is stopped.
>
> **check_stop**(*timer=None*)
>
> > Check the stopping criteria and returns True if the stopping condition is satisfied
> >
> > > **bool :**
> > >
> > > > True if the minimization must be stopped, False otherwise
>
> **finalize**(*verbose=1*, *timer=None*)
>
> > This method finalizes the minimization, and prints on stdout the results of the current minimization.
> >
> > > **verbose**
> > >
> > > > [int, optional] The verbosity level. If 0 only the final free energy and gradient is printed. If 1 the stress tensor is also printed. If 2 also the final structure and frequencies are printed.
>
> **get_free_energy**(*return_error=False*)
>
> > Obtain the SSCHA free energy per unit cell for the system. This is done through thermodynamic integration. Note that for the SSCHA this integration is performed analytically, so evaluating this function is almost immediate.
> >
> > The result is in Rydberg.
> >
> > $$\mathcal{F} = \mathcal{F}_0 + \int_0^1 \frac{d\mathcal{F}_\lambda}{d\lambda} d\lambda$$
> >
> > Where λ is the parameter for the adiabatic integration of the hamiltonian.
> >
> > $$H(\lambda) = H_0 + (H - H_0)\lambda$$
> >
> > here H_0 is the sscha harmonic hamiltonian, while H_1 is the real hamiltonian of the system.
> >
> > > **float**
> > >
> > > > The free energy in the current dynamical matrix and at the ensemble temperature
>
> **get_stress_tensor**()
>
> > For a full documentation, please refer to the same function of the Ensemble class. This subroutine just link to that one. A stress offset is added if defined in the input variable of the current class.
> >
> > NOTE: if the ensemble has not the stress tensors, an exception will be raised

init(*verbosity=False*, *delete_previous_data=True*, *init_timer=True*)

This subroutine initialize the variables needed by the minimization. Call this before the first time you invoke the run function.

> **verbosity**
>> [bool] If true prints some debugging information
>
> **delete_previous_data**
>> [bool] If true, it will clean previous minimizations from the free energies, gradients...
>
> **init_timer**
>> [bool] If true, it will initialize the timer for timing the function. This can slightly slowdown the minimization, however enables the use of the timer to check the time spent in each step.

is_converged()

Simple method to check if the simulation is converged or requires a new population to be runned.

> **bool :**
>> True if the simulation ended for converging.

minimization_step(*custom_function_gradient=None*, *timer=None*)

Perform the single minimization step. This modify the self.dyn matrix and updates the ensemble

> **custom_function_gradient**
>> [pointer to function (ndarray(nq x 3nat x 3nat), ndarray(nat, 3))] A function that can be used both to print particular component of the gradient or to impose some constraints on the minimization (like lock the position of some atoms). It takes as input the two gradient (the dynamical matrix one and the structure one), and modifies them (or does some I/O on it).

plot_results(*save_filename=None*, *plot=True*)

This usefull methods uses matplotlib to generate a plot of the minimization.

> **save_filename**
>> [optional, string] If present the plotted data will be saved in a text file specified by input.
>
> **plot**
>> [optiona, bool] If false no plot is performed. This allows only to save result even if you do not have any access in a X server.

print_info()

This subroutine is for debugging purposes, it will print the settings about the minimizer on the standard output.

run(*verbose=1*, *custom_function_pre=None*, *custom_function_post=None*, *custom_function_gradient=None*, *timer=None*)

This function uses all the setted up parameters to run the minimization

The minimization is stopped only when one of the stopping criteria are met.

The verbose level can be chosen.

---

verbose
   [int]

   The verbosity level.

   - 0 : Noting is printed

   - **1**
      [For each step only the free energy, the modulus of the gradient and] the Kong-Liu effective sample size is printed.

   - 2 : The dynamical matrix at each step is saved on output with a progressive integer

custom_function_pre
   [pointer to function (self)] It is a custom function that takes as an input the current structure. At each step this function is invoked. This allows to print particular analysis during the minimization that the user want to define to better control what is it happening to the system. This function is called before the minimization step has been performed. The info on the system saved in the self minimization reguards the previous step.

custom_function_post
   [pointer to function(self)] The same as the previous argument, but this function is invoked after the minimization step has been perfomed. The data about free energy, gradient and effective sample size have been updated.

custom_function_gradient
   [pointer to function (ndarray(NQ, 3*nat, 3*nat), ndarray(nat, 3))] A pointer to a function that takes as an input the two gradients, and modifies them. It is called after the two gradients have been computed, and it is used to impose some constraint on the minimization.

set_ensemble(*ensemble*)
   Provide an ensemble to the minimizer object

set_minimization_step(*step*)
   Set an uniform minimization step for both the dynamical matrix and the structure minimization.

   Try to always use this function unless you specifically want two difference speed between the structure and the dynamical matrix minimization.

setup_from_namelist(*input_file*)
   This function setups all the parameters of the minimization using a namelist. It is compatible with the old sscha code, and very usefull to save the input parameters in a simple input filename.

   line_list
      [list of string] List of strings obtained from the method readlines. The content must match the Quantum ESPRESSO file format

update(*timer=None*)
   This methods makes the self.dyn coincide with self.ensemble.current_dyn, and overwrites the stochastic weights of the current_dyn.

   Call this method each time you modify the dynamical matrix of the minimization to avoid errors.

> NOTE: it is equivalent to call self.ensemble.update_weights(self.dyn, self.ensemble.current_T)

6.3 The Relax Module

This module deals with relaxations that are iterated over more populations. It includes the variable cell optimization algorithm. Here the parameters read in the &relax name list are read and setup.

class `sscha.Relax.SSCHA`(*minimizer=None*, *ase_calculator=None*, *aiida_inputs:* *dict | None =* *None*, *N_configs=1*, *max_pop=20*, *save_ensemble=False*, *cluster=None*, ***kwargs*)

> **relax**(*restart_from_ens=False*, *get_stress=False*, *ensemble_loc=None*, *start_pop=None*, *sobol=False*, *sobol_scramble=False*, *sobol_scatter=0.0*)
>
> > This function performs the costant volume SCHA relaxation, by submitting several populations until the minimization converges (or the maximum number of population is reached)
> >
> > > **restart_from_ens**
> > >
> > > > [bool, optional] If True the ensemble is used to start the first population, without recomputing energies and forces. If False (default) the first ensemble is overwritten with a new one, and the minimization starts.
> > >
> > > **get_stress**
> > >
> > > > [bool, optional] If true the stress tensor is calculated. This may increase the computational cost, as it will be computed for each ab-initio configuration (it may be not available with some ase calculator)
> > >
> > > **ensemble_loc**
> > >
> > > > [string] Where the ensemble of each population is saved on the disk. If none, it will use the content of self.data_dir. It is just a way to override the variable self.data_dir
> > >
> > > **start_pop**
> > >
> > > > [int, optional] The starting index for the population, used only for saving the ensemble and the dynamical matrix. If None, the content of self.start_pop will be used.
> > >
> > > **sobol**
> > >
> > > > [bool, optional (Default = False)] Defines if the calculation uses random Gaussian generator or Sobol Gaussian generator.
> > >
> > > **sobol_scramble**
> > >
> > > > [bool, optional (Default = False)] Set the optional scrambling of the generated numbers taken from the Sobol sequence.
> > >
> > > **sobol_scatter**
> > >
> > > > [real (0.0 to 1) (Deafault = 0.0)] Set the scatter parameter to displace the Sobol positions randommly.
> > >
> > > **status**
> > >
> > > > [bool] True if the minimization converged, False if the maximum number of populations has been reached.

setup_custom_functions(*custom_function_pre=None*, *custom_function_gradient=None*, *custom_function_post=None*)

This subroutine setup which custom functions should be called during the minimization. Look for the SCHA_Minimizer.run() method for other details.

setup_from_namelist(*namelist*)

Setup the SSCHA relaxer from the given namelist.

Note the calculation will be also started by this method.

namelist
[dict] A dictionary that contains the namespaces

vc_relax(*target_press=0*, *static_bulk_modulus=100*, *restart_from_ens=False*, *ensemble_loc=None*, *start_pop=None*, *stress_numerical=False*, *cell_relax_algorithm='sd'*, *fix_volume=False*, *sobol=False*, *sobol_scramble=False*, *sobol_scatter=0.0*)

This function performs a variable cell SCHA relaxation at constant pressure, It is similar to the relax calculation, but the unit cell is updated according to the anharmonic stress tensor at each new population.

By default, all the degrees of freedom compatible with the symmetry group are relaxed in the cell. You can constrain the cell to keep the same shape by setting fix_cell_shape = True.

NOTE:
remember to setup the stress_offset variable of the SCHA_Minimizer, because in abinitio calculation the stress tensor converges porly with the cutoff, but stress tensor differences converges much quicker. Therefore, setup the stress tensor difference between a single very high-cutoff calculation and a single low-cutoff (the one you use), this difference will be added at the final stress tensor to get a better estimation of the true stress.

target_press
[float, optional] The target pressure of the minimization (in GPa). The minimization is stopped if the target pressure is the stress tensor is the identity matrix multiplied by the target pressure, with a tollerance equal to the stochastic noise. By default it is 0 (ambient pressure)

static_bulk_modulus
[float (default 100), or (9x9) matrix or string, optional] The static bulk modulus, expressed in GPa. It is used to initialize the hessian matrix on the BFGS cell relaxation, to guess the volume deformation caused by the anharmonic stress tensor in the first steps. By default is 100 GPa (higher value are safer, since they means a lower change in the cell shape). It can be also the whole non isotropic matrix. If you specify a string, it can be both:

- **"recalc"**
  [the static bulk modulus is recomputed with finite differences after] each step

- "bfgs" : the bfgs algorithm is used to infer the Hessian from previous calculations.

restart_from_ens
[bool, optional] If True the ensemble is used to start the first population,

without recomputing energies and forces. If False (default) the first ensemble is overwritten with a new one, and the minimization starts.

ensemble_loc

[string] Where the ensemble of each population is saved on the disk. You can specify None if you do not want to save the ensemble (useful to avoid disk I/O for force fields)

start_pop

[int, optional] The starting index for the population, used only for saving the ensemble and the dynamical matrix.

stress_numerical

[bool] If True the stress is computed by finite difference (usefull for calculators that does not support it by default)

cell_relax_algorithm

[string] This identifies the stress algorithm. It can be both sd (steepest-descent), cg (conjugate-gradient) or bfgs (Quasi-newton). The most robust one is SD. Do not change if you are not sure what you are doing.

fix_volume

[bool, optional] If true (default False) the volume is fixed, therefore only the cell shape is relaxed.

sobol

[bool, optional (Default = False)] Defines if the calculation uses random Gaussian generator or Sobol Gaussian generator.

sobol_scramble

[bool, optional (Default = False)] Set the optional scrambling of the generated numbers taken from the Sobol sequence.

sobol_scatter

[real (0.0 to 1) (Deafault = 0.0)] Set the scatter parameter to displace the Sobol positions randommly.

status

[bool] True if the minimization converged, False if the maximum number of populations has been reached.

6.4 The Utilities Module

This module both provides the constrains and the IOinput

6.4.1 IOInfo class

Use this class to make the python-sscha print information during the minimization

class sscha.Utilities.**IOInfo**

> **CFP_SaveAll**(*minim*)
>
> > This method saves everithing stored in this class.
> >
> > It can be passed as custom_function_post to the run method of the SchaMinimizer.
>
> **CFP_SaveFrequencies**(*minim*)
>
> > This custom method stores the total frequencies updating an exeternal file
>
> **Reset**()
>
> > Reset the data to empty.
>
> **Save**(*fname=None*)
>
> > Save the data on a file
> >
> > > **fname**
> > > [string, optional] If given, the file will be saved in the specified location. Otherwise the default one is used (must be initialized by SetupSaving)
>
> **SetupAtomicPositions**(*fname*, *save_each_step=True*)
>
> > Setup the saving of the data on atomic position along with the minimization
>
> **SetupSaving**(*fname*, *save_each_step=True*)
>
> > Setup the system to save the data each time the function is called.
> >
> > By default, the system will save frequencies and minimization info (like free energy, gradients and kong liu) The files are .freqs for the frequencies .dat for the minimization info
> >
> > > **fname**
> > > [string] path to the file to save the files
> > >
> > > **save_each_step**
> > > [bool] If true the file is saved (and updated) each time step.
>
> **SetupWeights**(*fname*, *save_each_step=True*)
>
> > Setup the weights saving
> >
> > > **fname**
> > > [string] Path to the file to which save the frequencies.

6.4.2 Constraints

The constrains are a member of the Utilities module. To implement constrains on phonon modes, use the ModeProjection class

class sscha.Utilities.**ModeProjection**(*dyn*)

> **CFG_ProjectDyn**(*dyn_grad*, *struct_grad*)
>
>> This subroutine constrains only the dynamical matrix, leaving the structure to minimize on all the possible degrees of freedom.
>
> **CFG_ProjectOnModes**(*dyn_grad*, *struct_grad*)
>
>> Function to be passed to the minimizer as 'custom_function_gradient'. It project the gradients in the polarization vector subspace. As any custom_function_gradient, it takes as input the two gradients.
>
> **CFG_ProjectStructure**(*dyn_grad*, *struct_grad*)
>
>> This subroutine constraints only the structure gradient, leaving the dynamical matrix to minimize on all the possible degrees of freedom.
>
> **SetupFreeModes**(*index_mode_start*, *index_mode_end*, *select_q_points=None*, *constrain=False*)
>
>> Constrain the minimization only in the modes between index_modes_start and index_mdoe_end in all the q points. Also the opposite is possible, by setting the constrain flag to True.
>>
>> For each q points only modes between these indices will be minimized. The select_q_points options allows to select only some q points to be constrained.
>>
>>> **index_mode_start**
>>>> [int] The index of the first mode
>>>
>>> **index_mode_end**
>>>> [int] The index of the last mode
>>>
>>> **select_q_points**
>>>> [list of int] The index of the q points affected. If None (default) all q points are affected.
>>>
>>> **constrain**
>>>> [bool] If True, the specified range of modes is constrained while all the other are free.

6.5 The Cluster Module

The Cluster module provides the interface between python-sscha and remote servers to which you submit the energy and forces calculations. The input in &cluster namespace is interpreted in this module

class sscha.Cluster.**Cluster**(*hostname=None*, *pwd=None*, *extra_options=''*, *workdir=''*, *account_name=''*, *partition_name=''*, *qos_name=''*, *binary='pw.x -npool NPOOL -i PREFIX.pwi > PREFIX.pwo'*, *mpi_cmd='srun --mpi=pmi2 -n NPROC'*)

CheckCommunication()
> This function return true if the server respond correctly, false otherwise.

ExecuteCMD(*cmd*, *raise_error=False*, *return_output=False*, *on_cluster=False*)
> This subroutine execute the cmd in the cluster, with the specified number of attempts.

> > **cmd: string**
> > > The whole command, including the ssh/scp.

> > **raise_error**
> > > [bool, optional] If True (default) raises an error upon failure.

> > **return_output**
> > > [bool, optional] If True (default False) the output of the command is returned as second value.

> > **on_cluster**
> > > [bool] If true, the command is executed directly on the cluster through ssh

> > **success**
> > > [bool] If True, the command has been executed with success, False otherwise

> > **output**
> > > [string] Returned only if return_output is True

batch_submission(*list_of_structures*, *calc*, *indices*, *in_extension*, *out_extension*, *label='ESP'*, *n_togheder=1*)
> This is a different kind of submission, it exploits xargs to perform a parallel submission of serveral structures in one single job. This is very good to avoid overloading the queue system manager, or when a limited number of jobs per user are allowed.

> NOTE: the number of structure in the list must be a divisor of the total number of processors.

> > **list_of_structures**
> > > [list] List of the structures to be computed.

> > **calc**
> > > [ase FileIOCalculator] The FileIOCalculator to perform the minimization

> > **indices**
> > > [list(int)] The indices of the configurations, this avoids interferring with other jobs when multiple jobs are lunched togheder.

> > **in_extension**
> > > [string] Extension of the input filename

> > **out_extension**
> > > [string] Extension of the output filename.

> > **label**
> > > [string, optional] The root of the input file.

> > **n_togheder**
> > > [int, optional (DO NOT USE)] If present, the job will lunch a new job immediately after the other is ended. This is usefull to further reduce the number of submitted jobs.

> **list_of_results.**
> > Returns a list of results dicts, one for each structure.

check_job_finished(*job_id*, *verbose=True*)

> Check if the job identified by the job_id is finished

> > **job_id**
> > > [string] The string that identifies uniquely the job

clean_localworkdir()

collect_results(*calc*, *submitted*, *indices*, *label*)

> Collect back all the results from the submitted data. This operation needs to be performed in thread safe mode (all threads must be locked between this operation and when the results are actually assigned to the ensemble to avoid conflicts)

compute_ensemble(*ensemble*, *ase_calc*, *get_stress=True*, *timeout=None*)

> > **ensemble :**
> > > The ensemble to be runned.

compute_ensemble_batch(*ensemble*, *cellconstructor_calc*, *get_stress=True*, *timeout=None*)

copy_files(*list_of_input*, *list_of_output*, *to_server*)

> This function copies the input files toward the HPC if to_server is True or retrieve the output files from the HPC if to_server is False

> > **list_of_input**
> > > [list] List of the path to the input files to be copied into the server.

> > **list_of_output**
> > > [list] List of the output files to be copied from the HPC server. It is needed also when submitting the input file from the server, as files that are named in the same way will be cleaned. (No risk to mistake them with the actual result of the calculation)

> > **to_server**
> > > [bool] If true, we copy the input files into the server from the local machine. If false, we copy the output files from the server to the local machine.

create_submission_script(*labels*)

> This is a function that is general and does not depend on the specific calculator. It is usefull to create the header of the submission script.

> > **labels**
> > > [list] It is a list of the labels of the calculations to be done.

> > **submission_header**
> > > [string] The text of the submission header.

get_execution_command(*label*)

> Return the command used in the submission script to actually execute the calculation.

> > **label**
> > > [string] The label of the calculation

commnad
[string] The command to be appended to the submission script

get_job_id_from_submission_output(*output*)
GET THE JOB ID

Retreive the job id from the output of the submission. This depends on the software employed. It works for slurm.

Returns None if the output contains an error

get_output_path(*label*)
Given the label of the submission, retrive the path of all the output files of that calculation

parse_string(*string*)
Parse the given string on the cluster. It can be used to resolve environmental variables defined in the cluster.

It will execute on the cluster the command:
echo "string"

and return the result of the cluster.

string :
String to be parsed in the cluster.

string :
The same as input, but with the cluster environmental variables correctly parsed.

prepare_input_file(*structures*, *calc*, *labels*)
This is specific for quantum espresso and must be inherit and replaced for other calculators.

This crates the input files in the local working directory self.local_workdir and it returns the list of all the files generated.

structures
[List of cellconstructor.Structure.Structure] The atomic structures.

calc
[the ASE or CellConstructor calculator.] In this case, it works with quantum espresso

labels
[List of strings] The unique name of this calculation

List_of_input
[list] List of strings containing all the input files

List_of_output
[list] List of strings containing the output files expected for the calculation

read_results(*calc*, *label*)
Return a dictionary of the computed property for the given calculation label

run_atoms(*ase_calc*, *ase_atoms*, *label='ESP'*, *in_extension='.pwi'*, *out_extension='.pwo'*, *n_nodes=1*, *n_cpu=1*, *npool=1*)

This function runs the given atoms in the cluster, using the ase_calculator. Note: the ase_calc must be a FileIOCalculator. For now it works with quantum espresso.

set_timeout(*timeout*)

Set a timeout time for each single calculation. This is very usefull as sometimes the calculations gets stucked after some times on clusters.

> **timeout: int**
> > The timeout in seconds after which a single calculation is killed.

setup_from_namelist(*namelist*)

This method setup the cluster using a custom input file. The inputfile must have the same shape of QuantumESPRESSO ones. The information about the cluster must be located in a namespace called as __CLUSTER_NAMELIST

> **namelist:**
> > The parsed namelist dictionary.

setup_workdir(*verbose=True*)

Parse the line contained in self.workdir in the claster to get working directory. It needs that the communication with the cluster has been correctly setted up.

It will parse correctly environmental variables of the cluster.

submit(*script_location*)

Submit the calculation. Compose the command into a cmd variable, then submit it through:

```
return self.ExecuteCMD(cmd, True, return_output=True)
```

> **script_localtion**
> > [string] Path to the submission script inside the cluster.

> **success**
> > [bool] Result of the execution of the submission command. It is what returned from self.ExecuteCMD(cmd, False)

INDICES AND TABLES

- genindex
- modindex
- search