python-sscha Documentation

Release 1.0

Lorenzo Monacelli

Mar 02, 2021

CONTENTS:

CHAPTER

ONE

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!

CHAPTER

TWO

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 Requirements

To install python-sscha you need: 1. python (either 2.7 or 3.*) 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
```

2.2 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.

2.3 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).

2.4 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.

2.5 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.

CHAPTER

THREE

QUICK START

To quickly start using the code, we recommend using the jupyter notebooks with examples we provide in the Tutorials directory of the source code.

Tutorials are organized as follows:

1. Setup the first calculation: 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.

CHAPTER

FOUR

CUSTOMIZING THE SSCHA

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.

4.1 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:

```
# [...] 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.

```
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.

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

(continues on next page)

(continued from previous page)

```
# 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:

```
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.

CHAPTER

FIVE

FREQUENTLY ASKED QUESTIONS (FAQS)

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.

```
# 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
```

What are the reasonable values for the steps (lambda_a, lambda_w, min_step_dyn, and min_step_struc)?

The code minimizes using a Newton method: preconditioned gradient descend. Thanks to an analytical evaluation of the Hessian matrix, the step is rescaled so that the theoretical best step is close to 1. In other words: **one is theoretically the best (and the default) choice for the steps**. However, the SSCHA is a stochastic algorithm, therefore, if the ensemble is too small, or the gradient is very big, this step could bring you outside the region in which the ensemble is describing well the physics very soon. Since SSCHA can exploit the reweighting, and the most computationally expensive part of the algorithm is the computation of forces and energies, it is often much better using a small step (smaller than the optimal one). **Good values of the steps are usually around 0.01 and 0.1**. Rule of thumbs: the minimization should not end because it went outside the stochastic regime

before that at least 10 steps have been made. This will depend on the system, the number of configurations, and how far from the correct solution you are.

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).

If you are using the python script, the equivalent variables are the attributes of the sscha.SchaMinimizer.SSCHA_Minimizer class.

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

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

In a variable cell optimization, 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.

Usual values are between 10 GPa and 100 GPa for a system at ambient conditions. Diamond has a bulk modulus of about 500 GPa. High-pressure hydrates have a bulk modulus of around 300 GPa as well.

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).

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

This means that your step is too large. You can reduce the step of the minimization. An alternative (often more efficient) is to switch to the root representation. In this way, the square root of the dynamical matrix is minimized, and the total dynamical matrix is positive definite in the whole minimization by construction.

In the input name-list, you activate this minimization with the following keywords inside the &inputscha namelist

```
preconditioning = .false.
root_representation = "root4"
```

Or, in the python script, you set up the attributes of the sscha.SchaMinimizer.SSCHA_Minimizer class

```
minim.preconditioning = False
minim.root_representation = "root4"
```

The optimal step size for the root_representation may be different than the other one.

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 = "freqs.dat"
&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("freqs.dat")

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

The code here is providing the SSCHA code a function (IO_freq.CFP_SaveFrequencies) that is called after each minimization step. This function saves all the frequencies of the current dynamical matrix in the file specified by IO_freq.SetupSaving("freqs.dat").

To plot the results, the SSCHA offers an executable script, installed together with the code. Just run:

```
plot_frequencies.py freqs.dat
```

And the code will plot all the frequencies. You can also pass more than one file. In this case, the frequencies are concatenated.Plotting the frequencies of the dynamical matrix is a very good way to check if the algorithm is converging correctly.

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. This could be due to a step size too big, then try to reduce the value of **lambda_a** and **lambda_w** (in the input file) or **min_step_dyn** and **min_step_struc** (in the python script). It could also be due to a wasted ensemble, in this case, check the value of the Kong-Liu effective sample size, if it is below or around 0.5, then try to increase the threshold at which stop the calculation, **kong_liu_ratio** (in the python script) or **N_random_eff** (in the input file), or increase the number of configurations for the next population.

The gradients on my simulations are increasing a lot, why is this happening?

See the previous question.

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. 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.

If the code is provided with a &utils namespace, on which the code

```
&utils
   save_freq_filename = "frequencies_popX.dat"
&end
```

You can after the minimization use the plotting program to see the frequencies as they evolve during the minimizations:

```
plot_frequencies_new.pyx frequencies_pop*.dat
```

This will plot all the files *frequencies_popX.dat* in the directory. You can see all the history of the frequency minimization. If between different populations (that you will distinguish by a kink in the frequency evolutions) the frequencies will fluctuate due to the stochastic nature of the algorithm, with no general drift, then the algorithm reached its maximum accuracy with the given number of configurations. You may either stop the minimization or increase the ensemble to improve the accuracy.

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

This is a difficult question. The best way to estimate the error is to generate a new ensemble with the same number of configurations at the end of the minimization and check how the final optimized solution changes with this new ensemble. This is also a good way to test if the solution is converged to the correct solution. The magnitude of the changes in the dynamical matrix's frequencies and structure is an accurate estimation of the stochastic error.

You can always split the ensemble in two and run two minimizations with the two half of the ensemble to get a hint on the error on the structure or the dynamical matrix. To split the ensemble, refer to the *FAQ* about the error on the hessian matrix.

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.

When I relax the cell, is it necessary for the gradients to reach zero before making a step with the new cell?

In general, it is good to have a reasonable dynamical matrix before starting with a variable cell relaxation. The best strategy is to perform a fixed cell relaxation with few configurations until you are close to the final solution (the gradients are comparable with their errors). Then you can start a variable cell relaxation and submit new populations in the suggested new cell even if the previous one was not perfectly converged.

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 can improve the convergence of the variable cell algorithm by a lot.

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.

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>*

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

The target of a SSCHA minimization is to find the ionic density matrix $\rho(\Phi, \vec{\mathcal{R}})$ that minimizes the total free energy. It may happen, if we are using a too big step for the dynamical matrix Φ that it becomes not positive definite. This may be due to the stochastic noise during the minimization. For avoid this to happen, you may set **root_representation** to either **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

```
# The ensemble has been loaded as ens
minim = sscha.SchaMinimizer.SSCHA_Minimizer(ens)
minim.root_representation = "normal"
minim.precond_dyn = True
```

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.

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*.

```
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

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.

How do I understand if I have to generate a new population or the minimization converged?

In general, if the code stops because the gradient is much below the error (less then 1%), then it is converged (with a Kong-Liu threshold ratio of at least 0.5). If the code ends the minimization because it went outside the stochastic criteria, a new population is required. There are cases in which you use to few configurations to reach a small gradient before wasting the ensemble. If this is the case, print the frequencies during the minimizations (using the &utils card with `save_freq_filename` attribute). You may compare subsequent minimizations, if the frequencies are randomly moving between different minimization (and you cannot identify a trend in any of them), then you reach the limit of accuracy of the ensemble. Frequencies are a much better parameter to control for convergence than free energy, as the free energy close to the minimum is quadratic.

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.

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.

```
import sscha, sscha.Ensemble

# Load the dynamical matrix as dyn
# [...]

# ens is the Ensemble() class correctly initialized.
# We can for example load it
# Assuming it is stored in 'data_dir' with population 1 and 1000 configurations
# We assume to have loaded the original dynamical matrix dyn and to know the␣
↪generating temperature T
ens = sscha.Ensemble.Ensemble(dyn, T, dyn.GetSupercell())
ens.load("data_dir", population = 1, N = 1000)

# We create a mask that selects which configurations to take
```

(continues on next page)

(continued from previous page)

```
first_half_mask = np.zeros(ens.N, dtype = bool)
first_half_mask[: ens.N // 2] = True

# We create also the mask for the second half
# by taking the not operation on the first_half_mask
second_half_mask = ~first_half_mask

# Now we split the ensemble
ens_first_half = ens.split(first_half_mask)
ens_second_half = ens.split(second_half_mask)

# We can save the two half ensembles as population 2 and 3.
ens_first_half.save("data_dir", population = 2)
ens_second_half.save("data_dir", population = 3)
```

This simple script will generate two ensembles inside `data_dir` directory with population 2 and 3, each one containing the first and the second half of the ensemble with population 1 respectively. You can perform then your calculation of the free energy Hessian with both the ensemble to estimate the error on the frequencies and the polarization vectors.

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.

```
# 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.

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

As for version 1.0, 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:

```
import numpy as np
```

(continues on next page)

(continued from previous page)

```
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.

CHAPTER

SIX

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.

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.

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.

6.4 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

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search