python-sscha Documentation

Release 1.0

Lorenzo Monacelli

Aug 25, 2020

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 What is python-sscha?

python-sscha is a 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, than you need python-sscha. This 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). This 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 simulatie 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 the comparable to standard classical molecular dynamics. Since the algorithm correctly exploits the symmetries of the crystal, in highly symmetric cases it is also much cheaper.

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

It can be coupled 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 quite easy to use, a standard input is highly human readable and less than 10 lines! So, what are you waiting? Download and install python-sscha, and start enjoing the Tutorials!

CHAPTER

TWO

HOW TO INSTALL

2.1 Requirements

The requirements of the python-sscha package are: 1. python >= 2.7 and < 3 2. numpy 3. matplotlib 3. Lapack 4. Blas 5. gfortran (or any fortran compiler) 6. CellConstructor

Recommended packages: 1. Atomic Simulation Environment (ASE) 2. SPGLIB

These packages are fundamental. In particular CellConstructor is the shoulder on which python-sscha is builded on. You can find the last development version on GitHub

The ASE (Atomic Simulation Environment) is another dependency, since CellConstructor relies on it.

To install all the dependences, simply run

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

2.2 Installation

To install the package it is recommended the last version of anaconda-python2, that cames with all the updated numpy and matplotlib, already compiled to work in parallel. Moreover the last version of matplotlib will allow the user to modify the plots after they are produced.

It can be simply installed from command line:

```
`bash python setup.py install `
```

This command must be executed in the same directory as the setup.py script. Note, if you installed python in a system directory, administration rights may be requested (add a sudo before the command). Please, consider adopting anaconda-python to install the software on clusters where you do not have administration rights.

Installation on clusters:

It is suggested to install the package with the anaconda distribution.

Please, remember that if you use the intel compiler, you need to delete the lapack linking from the setup.py and include the -mkl (as done for cellconstructor). Note that you must force to use the same liker compiler as the one used for the compilation.

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

CHAPTER

THREE

QUICK START

In this section we will guide you step by step on the first use of the python-sscha code.

CHAPTER

FOUR

FREQUENTLY ASKED QUESTIONS (FAQS)

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

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 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 computational 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 whent 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 in order 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 letterature a bulk modulus.

Usual values are between 10 GPa and 100 GPa for system at ambient conditions. Diamond has a bulk modulus about 500 GPa. High pressure hydrates have a bulk modulus around 500 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 you 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 namelist input you activate this minimization with the following keywords inside the &inputscha namelist

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

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

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

It is possible that the optimal step size for the root_representation is different than the other one.

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), in order to stop the minimization when the gradient start increasing, or to increase the number of configurations in the ensemble.

In any case, what must decrease is the 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 the free energy are increasing, something is wrong. This could be due to a too big step size, 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 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 configuration 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 actually converged to the correct solution. The magnitude of the changes in the dynamical matrix's frequencies and structure is an accurate estimation on the stochastic error.

You can always split the ensemble in two and run two minimization with the two half of the ensembe to get a hint on the error on the structure or on 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 stronlgy anisotropic, so the algorithm has difficulties in optimizing well. In general the stress tensor is also affected by 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 in 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 the different kind 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 are constrained in a space that is positive definite. Moreover, it has been proved that $\sqrt[4]{\Phi}$ minimization is better conditioned than the original, and thus should reach the minimum faster.

Alternatively, a similar effect to the speedup in the minimization obtained with **root4** is possible if use 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 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 garanteed 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 both in the input script and directly by the python version. 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 those directions, by picking:

project_dyn = .true. and **project_structure = .false.**. In this way, I freeze only the dynamical matrix along 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 constrains than mode locking may be activated in the minimization, but their use is limited within the python scripting. You can write your own constraining function that will be applied to the structure gradient or to 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, in this way the overall gradient will have a translation (acoustic sum rule is violated). Be very carefull 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 none 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 on how good is the extracted set of configurations to describe the BO landscape around the current values of 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 of 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. For this reason 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 that is completely random). On the other side, on some very complex system 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 indefinitively 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
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 ensembe?

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.
```

(continues on next page)

(continued from previous page)

```
# 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 as random number generator, you can, at the beginning of the script that generates the ensemble, use the following:

```
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 of initializing with the current local time.

CHAPTER

FIVE

THE API

This chapter contains the documentations for the main methods of the python-sscha code. It can be used both from 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 own purpouses).

The API is divided in Modules.

5.1 The Ensemble Module

This module deals with 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 includes 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.

5.2 The SchaMinimizer Module

This module is the main SSCHA minimizer. It allows to setup 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 namelist are read.

`sscha.SchaMinimizer.`**`ApplyFCPrecond`**(*current_dyn*, *matrix*, *T=0*)

This function perform the precondition on a given matrix, by applying the inverse of the lambda function

current_dyn [3*nat x 3*nat] The current force-constant matrix to compute the Lambda tensor

matrix [3*nat x 3*nat] The matrix on which you want to apply the Lambda tensor.

T [float] The temperature

new_matrix [3*nat x 3*nat] The matrix after the Lambda application

`sscha.SchaMinimizer.`**`ApplyLambdaTensor`**(*current_dyn*, *matrix*, *T=0*)

This function perform the inverse of the precondtioning: it applies the Hessian matrix to the preconditioned gradient to obtain the real one. This is a test function.

current_dyn [3*nat x 3*nat] The current force-constant matrix to compute the Lambda tensor

matrix [3*nat x 3*nat] The matrix on which you want to apply the Lambda tensor.

T [float] The temperature

new_matrix [3*nat x 3*nat] The matrix after the Lambda application

sscha.SchaMinimizer.**GetBestWykoffStep**(*current_dyn*)

This is an alternative way to the preconditioning, in which the best wyckoff step is choosen rescaled on the current dynamical matrix.

NOTE: It works with real space matrices.

$$STEP = \frac{1}{\max \lambda(\Phi)}$$

Where $\lambda(\Phi)$ is the generic eigenvalue of the force constant matrix. This is because Φ is correct Hessian of the free energy respect to the structure in the minimum.

The best step is returned in [Angstrom^2 / Ry].

current_dyn [ndarray 3n_at x 3n_at] The force constant matrix Φ. It should be in Ry/bohr^2.

sscha.SchaMinimizer.**GetStructPrecond**(*current_dyn*)

NOTE: the Phi is in Ry/bohr^2 while the forces are in Ry/A NOTE: the output preconditioner (that must be interfaced with forces) is in A^2/Ry

The preconditioner of the structure minimization is computed directly from the dynamical matrix. It is the fake inverse (projected out the translations).

$$\Phi^{-1}_{\alpha\beta} = \frac{1}{\sqrt{M_\alpha M_\beta}} \sum_\mu \frac{e^\alpha_\mu e^\beta_\mu}{\omega^2_\mu}$$

Where the sum is restricted to the non translational modes.

current_dyn [Phonons()] The current dynamical matrix

preconditioner [ndarray 3nat x 3nat] The inverse of the force constant matrix, it can be used as a preconditioner.

sscha.SchaMinimizer.**PerformRootStep**(*dyn_q*, *grad_q*, *step_size=1*, *root_representation='sqrt'*, *minimization_algorithm='sdes'*)

As for the [Monacelli, Errea, Calandra, Mauri, PRB 2017], the nonlinear change of variable is used to perform the step.

It works as follows:

$$\Phi \to \sqrt{x}\Phi$$
$$\frac{\partial F}{\partial \Phi} \to \frac{\partial F}{\partial \sqrt{x}\Phi}$$
$$\sqrt{x}\Phi^{(n)} \xrightarrow{\frac{\partial F}{\partial \sqrt{x}\Phi}} \sqrt{x}\Phi^{(n+1)}$$
$$\Phi^{(n+1)} = \left(\sqrt{x}\Phi^{(n+1)}\right)^x$$

Where the specific update step is determined by the minimization_algorithm, while the x order of the root representation is determined by the root_representation argument.

dyn_q [ndarray(NQ x 3nat x 3nat)] The dynamical matrix in q space. The Nq are the total number of q.

grad_q [ndarray(NQ x 3nat x 3nat)] The gradient of the dynamical matrix.

step_size [float] The step size for the minimization

root_representation [string] choice between "normal", "sqrt" and "root4". The value of x will be, respectively, 1, 2, 4.

minimization_algorithm [string] The minimization algorithm to be used for the update.

new_dyn_q [ndarray(NQ x 3nat x 3nat)] The updated dynamical matrix in q space

sscha.SchaMinimizer.**get_root_dyn**(*dyn_fc*, *root_representation*)

Get the root dyn matrix

This method computes the root equivalent of the dynamical matrix

5.3 The Relax Module

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

sscha.Relax.**GetStaticBulkModulus**(*structure*, *ase_calculator*, *eps=0.001*)

This method uses finite differences on the cell to compute the static bulk modulus. The cell is strained into several volumes, and the stress tensor is computed in orther to obtain the bulk modulus. Only the symmmetry relevant terms are computed.

structure [CC.Structure.Structure()] The structure on which you want to compute the static bulk modulus

ase_calculator [ase.calculators.calculator.Calculator()] One of the ase calculators to get the stress tensor in several strained cells.

eps [float] The strain module

bk_mod [ndarray (9x9)] The bulk modulus as a 9x9 matrix, expressed in eV / A^3

5.4 The Cluster Module

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

sscha.Cluster.**units = {'A': 63541.72207603944, 'AUT': 0.002375996331368385, 'Ang': 1.0, 'A**

This is an untility script that is able to manage the submission into a cluster of an ensemble

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S