Elastic

Release v4.99-54-gcbfbe3d

Paweł T. Jochym

Mar 24, 2018

Contents

New version 5.0 released

The new version is API *incompatible* with the previous versions. It provides a new command line utility as the main user interface to the package - which hoppefully will broaden the user base byond python users.

Elastic is a set of python routines for calculation of elastic properties of crystals (elastic constants, equation of state, sound velocities, etc.). It is a fifth version of the in-house code I have written over several years and is implemented as a extension to the ASE system and a script providing interface to the library not requiring knowledge of python or ASE system. The code was a basis for some of my publications and was described briefly in these papers. The code was available to anyone, presented at our Workshop on ab initio Calculations in Geosciences and used by some of my co-workers but was never properly published with full documentation, project page etc. In 2010, I have decided to re-implement elastic as a module for the ASE system and publish it properly under the GPL as versions 3 and 4.

Later, in 2017, needs of users nudged me into implementing the command-line front-end to the library which is included with version 5.0 of the package.

The version 5.0 also changes API of the library from mix-in class to the set of simple functions providing functionality of the module. The workflow of the package was also changed to prepare data - calculate - post-process style, which is better suited to serious research work.

The source code started live on the launchpad project page and later in 2014 moved to the github repository with corresponding elastic web page and on-line documentation placed at Elastic website (you are probably reading from it already).

The project is free software and I welcome patches, ideas and other feedback.

CHAPTER 1

Physical Principles

Elastic is based on the standard elasticity theory (see *[LL]* for the detailed introduction) and *finite deformation* approach to the calculation of elastic tensor of the crystal. I have described basic physical principles on which the code rests in my habilitation thesis. Here I will include slightly edited second chapter of the thesis introducing the method and some implementation details.

1.1 Elasticity of crystals

The classical, linear theory of elasticity of crystalline materials has been formulated already in the 18th and 19th century by Cauchy, Euler, Poisson, Young and many other great mathematicians and physicists of that time. The standard textbook formulation (e.g. classical book by Landau et al. *[LL]*) can be, in principle, directly used as a basis for numerical determination of the elastic tensor and other mechanical properties of the crystal. Nevertheless, practical implementation of these formulas have some non-obvious aspects, worthy of explicit presentation. The *finite deformation* method developed and used in the mentioned papers *[TiC]*, *[ZrC]* is based on the fundamental relationship between stress and strain of the solid crystalline body with a particular symmetry. This is a simple tensor equation, sometimes called generalised *Hook's law* (in standard tensor notation):

$$\sigma_{\lambda\xi} = C_{\lambda\xi\mu\nu} s_{\mu\nu}$$

This formula simply states that the stress in the crystal $\sigma_{\lambda\xi}$ is a linear function of the strain $s_{\mu\nu}$ incurred by its deformation, and the elasticity tensor $C_{\lambda\xi\mu\nu}$ is just a tensor proportionality coefficient. The Greek indexes run through coordinates x, y, z. The elasticity tensor inherits symmetries of the crystal and has some intrinsic symmetries of its own. Therefore, only a small number of its components are independent. This fact leads to customary representation of this entity in the form of the matrix with components assigned according to Voight's notation. Thus, instead of the rank-4 three dimensional tensor we have 6×6 matrix C_{ij} where the indexes $i, j = 1 \ldots 6$. The stress and strain tensors are represented as six-dimensional vectors. The symmetries of the elastic tensor are directly translated into symmetries of the C_{ij} matrix. The Voight's notation is commonly used in tensor calculus. For this particular case we can write it as an index assignment where each pair of Greek indexes is replaced with a corresponding Latin index (i, j, k, l, m, n): xx=1, yy=2, zz=3, yz=4, xz=5, xy=6.

While this convention makes presentation of elastic constants much easier - since it is just a square table of numbers - it slightly complicates algebraic procedures as we lose the simplicity of the tensor formalism. Every class of crystal implies, through its symmetry, a different number of independent elements in the C_{ij} matrix.

For example, the cubic lattice has just three independent elements in the elastic matrix: C_{11}, C_{12}, C_{44}, and the matrix itself has the following shape:

$$\begin{bmatrix} C_{11} & C_{12} & C_{12} & 0 & 0 & 0 \\ C_{12} & C_{11} & C_{12} & 0 & 0 & 0 \\ C_{12} & C_{12} & C_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & C_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & C_{44} & 0 \\ 0 & 0 & 0 & 0 & 0 & C_{44} \end{bmatrix}$$

Less symmetric crystals have, naturally, a higher number of independent elastic constants and lower symmetry of the C_{ij} matrix (see *[LL]* for full introduction to theory of elasticity).

1.2 Numerical derivation of elastic matrix

Numerical derivation of the C_{ij} matrix may be approached in many different ways. Basically, we can employ the same methods as used effectively in experimental work. From all experimental procedures we can select three classes which are relevant to our discussion:

1. Based on the measured sound velocity, including various methods based on determination of lattice dynamics of the crystal.
2. Based on the strain-energy relation.
3. Based on the measured stress-strain relations for some particular, simple strains.

While the first method is frequently used in laboratory measurements, it is not direct and is not well suited to numerical derivation. For example, you can measure the tangent of all acoustic branches of phonon dispersion curves in several directions to get enough data points to solve the set of equations for most of the independent components of the C_{ij} matrix. The tangent of the acoustic branch is connected with the sound velocity and with components of elastic matrix by a set of equations of the general form:

$$\varrho v_k^2 = L(C_{ij})$$

where $L(C_{ij})$ is a linear combination of independent components of elastic tensor, v_k is a long-wave sound velocity in particular direction, which is equivalent to the slope of the acoustic branch of phonon dispersion curve in this direction, and ϱ is crystal density. Full set of these equations for the cubic crystal is included in *[TiC]*. Unfortunately, it is difficult and non-practical to use this method to obtain more then few of the simplest of components, since the numerical properties of the non-linear formulas involved lead to the error pile-up in the results. It is particularly susceptible to errors in long-wave sound velocities – due to the quadratic function in above equation. Unfortunately, these asymptotic velocities are particularly weakly constrained by most of available computational methods. The same formulas can also be used to obtain elastic matrix from straight-forward sound velocity measurements. The same unfavourable numerical properties lead to high demands on accuracy of the measurements – but in this case these requirements could be quite easily met in experiment since sound velocity can be measured with very high precision.

The second method is not practical for laboratory measurements - it is not easy to accurately measure energy of the deformed crystal. Furthermore, the strain-energy relation is non-linear and we need to extract a derivative of the function – the procedure is quite complex, needs more data points and is prone to errors.

The third method is well suited for experimental work as well as computational derivation of the elastic matrix. The numerical properties of the formulas – being just a set of linear equations – are well known and provide stable and well-controlled error propagation. Furthermore, while the sound velocity is not directly accessible to computational quantum mechanical methods, the stresses induced by strains on the crystal are almost universally provided by DFT based programs and often do not require any additional computational effort. The comparison of these methods used for computational derivation of the elastic matrix is included in *[TiC]*, *[ZrC]*. The comparison shows that the finite

deformation (stress-strain) method compares favourably to the pure energy-derivative method. The results clearly show that the strain–stress relationship approach described here is much better suited for computational derivation of elastic matrix and provides lower error level than other two methods.

1.3 Crystal symmetry and elastic matrix derivation

As mentioned above, the symmetry of the crystal determines the number and position of independent components of the C_{ij} matrix. Therefore, the stress-strain relation is effectively modified by the symmetry of the case by a simple fact that most, of the coefficients are not independent from one another. We aim to derive the complete set of C_{ij} elements from the set of computational or experimental measurements of strain and stress tensors s^a, σ^a where the upper Latin index a numbers a calculation/experiment setup. In the case described here the "measurement" is a particular computational setup with the crystal deformed in various ways in order to provide enough data points to derive all independent components of the C_{ij} matrix. The set of necessary deformations can be determined by the symmetry of the crystal and contains tetragonal and sheer deformations along some or all axis – as the symmetry of the case dictates. To improve the accuracy of the results the deformations may be of different sizes (typically 0.1-1% in length or 0.1-1 degree in angle).

Having a set of calculation data $\{s^a, \sigma^a\}$, we can rewrite generalised Hook's law to form a set of linear equations (in Voight notation for i, j indexes): $C_{ij}s^a_j = \sigma^a_i$. This set can be further transformed for each symmetry case to the form in which the independent components of the C_{ij} matrix create a vector of unknowns and the symmetry relations and strains s^a_j create a new equation matrix S. $S_{ju}(s^a)C_u = \sigma^a_j$. The $S(s)$ matrix is a linear function of the strain vector s with all symmetry relations taken into account. The index a runs over all data sets we have in the calculation while index u runs over all independent components of the C_{ij} matrix. For the cubic crystal the above equation takes explicit form:

$$\begin{bmatrix} s_1 & s_2+s_3 & 0 \\ s_2 & s_1+s_3 & 0 \\ s_3 & s_1+s_2 & 0 \\ 0 & 0 & 2s_4 \\ 0 & 0 & 2s_5 \\ 0 & 0 & 2s_6 \end{bmatrix}^a \begin{bmatrix} C_{11} \\ C_{12} \\ C_{44} \end{bmatrix} = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \sigma_4 \\ \sigma_5 \\ \sigma_6 \end{bmatrix}^a .$$

Note the a index of S and σ, which creates a set of $n \times 6$ linear equations for 3 unknowns $[C_{11}, C_{12}, C_{44}]$, where n is a number of independent calculations of stresses incurred in crystal by strains. In principle, the above relations could be expressed in the non-symmetry specific form with either a full set of indexes and the symmetry information encoded in the single matrix of constant elements or even in the pure tensor formulation with the four-index elastic tensor C and two-index stress and strain tensors. While this type of formulation is definitely more regular and sometimes easier to manipulate in formal transformations, it is not very useful for numerical calculations or writing computer code – multi-dimensional arrays are difficult to manipulate and are prone to many trivial notation errors. Thus, it is better to split the general formula to crystal classes with different number of C_{ij} components (i.e. length of the C_u vector) and separate shape of the S matrix. This is an approach used by Elastic.

For example, in the orthorhombic crystal the vector of independent C_{ij} components has nine elements and the S matrix is a 9×6 one:

$$\begin{bmatrix} s_1 & 0 & 0 & s_2 & s_3 & 0 & 0 & 0 & 0 \\ 0 & s_2 & 0 & s_1 & 0 & s_3 & 0 & 0 & 0 \\ 0 & 0 & s_3 & 0 & s_1 & s_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2s_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2s_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2s_6 \end{bmatrix}^a \begin{bmatrix} C_{11} \\ C_{22} \\ C_{33} \\ C_{12} \\ C_{13} \\ C_{23} \\ C_{44} \\ C_{55} \\ C_{66} \end{bmatrix} = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \sigma_4 \\ \sigma_5 \\ \sigma_6 \end{bmatrix}^a .$$

The elements of the matrix S have direct relation to the terms of expansion of the elastic free energy as a function of deformation (strain tensor) F(s). For example, the orthorhombic equation can be derived from the free energy formula (see *[LL]* for derivation):

$$F(s) = \frac{1}{2}C_{11}s_1^2 + \frac{1}{2}C_{22}s_2^2 + \frac{1}{2}C_{33}s_3^2 + C_{12}s_1s_2 + C_{13}s_1s_3 + C_{23}s_2s_3 + 2C_{44}s_4^2 + 2C_{55}s_5^2 + 2C_{66}s_6^2$$

The elements of the S matrix are simply coefficients of first derivatives of the F(s) over respective strain components. Alternatively, we can rewrite the S(s) matrix in the compact form as a mixed derivative:

$$S_{iu} = A\frac{\partial^2 F}{\partial s_i \partial C_u},$$

where A is a multiplier taking into account the double counting of the off-diagonal components in the free energy formula (see note at the end of the exercises in *[LL]*). The multiplier $A = 1$ for $i \leq 4$, and $1/2$ otherwise. The above general formula turns out to be quite helpful in less trivial cases of trigonal or hexagonal classes. For instance, the hexagonal elastic free energy (see *[LL]* for rather lengthy formula) leads to the following set of equations:

$$\begin{bmatrix} s_1 & 0 & s_2 & s_3 & 0 \\ s_2 & 0 & s_1 & s_3 & 0 \\ 0 & s_3 & 0 & s_1+s_2 & 0 \\ 0 & 0 & 0 & 0 & 2s_4 \\ 0 & 0 & 0 & 0 & 2s_5 \\ s_6 & 0 & -s_6 & 0 & 0 \end{bmatrix}^a \begin{bmatrix} C_{11} \\ C_{33} \\ C_{12} \\ C_{13} \\ C_{44} \end{bmatrix} = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \sigma_4 \\ \sigma_5 \\ \sigma_6 \end{bmatrix}^a .$$

The set of linear equations, with calculated strains and stresses inserted into the S^a matrix and σ^a vector, could be constructed for any crystal – only the form of the S matrix and the length of the C_u vector will be different for each symmetry.

The set of equations is usually over-determined. Therefore, it cannot be solved in the strict linear-algebra sense since no exact solution could exist. Nevertheless, this set of equations can be solved in approximate sense – i.e. minimising the length of the residual vector of the solution. Fortunately, a very clever algorithm capable of dealing with just this type of linear equations has been known for a long time. It is called Singular Value Decomposition *[SVD]*. Not only does it provide the approximate solution minimising the residual vector of the equation but also is stable against numerically ill-conditioned equations or equations which provide too little data to determine all components of the solution. The SVD provides also some indication of the quality of the obtained solution in the form of the vector of singular values, which could be used to judge whether the solution is well-determined. It is a well known algorithm and its implementations are available in every self-respecting numerical linear algebra library. The implementation used in the Elastic code is the one included in the Scientific Python library SciPy.

CHAPTER 2

Installation

2.1 Conda

The installation procedure is quite simple if you use, *highly recommended* conda package manager:

```
conda install -c conda-forge elastic
```

The above command installs elastic with all dependencies into your current conda environment. If you want to add my anaconda.org channel into your conda installation you need to run following command:

```
conda config --add channels conda-forge
```

The above method has additional benefit of providing current installation of ASE and spglib libraries.

2.2 PyPi

The package is published simultaneously on conda and pypi. The second recommended way to install elastic is with pip:

```
pip install elastic
```

which should install the package and all its dependencies. Note that the number of dependencies is rather large and some of them are fairly large. Most of them, however, are just standard scientific python packages - almost all are present in standard anaconda install.

2.3 Manual

To install the code *pedestrian way* you need to install following python packages (most, if not all, are available in major linux distributions):

- SciPy and NumPy libraries
- matplotlib (not strictly required, but needed for testing and plotting)
- ASE system
- spglib space group library
- Some ASE calculator (VASP, GPAW, abinit, ...), but be warned that for now the code was developed using VASP only. I will be happy to help you extending it to other calculators. The code can be used without supported ASE calculator using command line interface and external, independent calculation tool.

This is highly system-dependent and I am unable to provide detailed support for this type of install - I use conda install of ASE/elastic myself!

Some legacy installation guides which may help you with manual process could be find at the QE-doc project pages.

Essentially, you need to clone the repository and run:

```
python setup.py install
```

in the main directory. But this is really not recommended way to install. Use it at your own risk and if you know what you are doing.

2.4 Testing

The simplest verification if everything works is just using the `elastic` utility to see the help screen:

```
elastic --help
```

or version of the package:

```
elastic --version
```

Additionally the whole package has a set of unittests based on hypothesis package. The tests are self-contained and do not require any external packages like DFT programs (e.g. VASP). You can run these tests by executing following command in the source directory:

```
python -m unittest discover -s test -b
```

CHAPTER 3

Usage

Starting from ver. 5.0, the command line utility `elastic` is a primary interface to the package and the direct python programming with the library is relegated to non-standard calculations. The basic calculation scheme can be summarized with the following list:

- Prepare the basic structure data in the form of VASP POSCAR file or abinit input file. Support for other programs can be added relatively easily. Contact the author if you need it. The structure should be fully optimized represent what you consider to be ground state of the system.
- run `elastic` on the structure to generate deformed structures probing the properties of the system:

  ```
  elastic -v --cij gen -n 5 -s 2 POSCAR
  ```

  which generates a set of deformed systems named `cij_XXX.POSCAR`, where `XXX` is replaced by numbers with 000 corresponding to undisturbed structure.
- run your DFT program (VASP, abinit, etc.) on all systems. This step depends on your particular system, and you need to handle it yourself. You need to make sure that for each system the internal degrees of freedom are optimized and full stress tensor gets calculated. Example bash script handling this task on my cluster looks like this:

  ```
  #!/bin/bash

  # Command to run vasp in current directory
  RUN_VASP="/opt/bin/run-vasp541"

  for s in $* ; do
      d=${s%%.POSCAR}
      echo -n  $d ": "
      mkdir calc-$d
      (
          cd calc-$d
          ln -s ../INCAR ../KPOINTS ../POTCAR ../vasprun.conf .
          ln -s ../$s POSCAR
          $RUN_VASP
  ```

```
    )
done
```

This produces a set of directories: `calc-cij_XXX` with completed single-point calculations.

- run `elastic` again to post-process the calculations. We do that by feeding it with output from the DFT calculations. Remember to put undisturbed structure at the first position:

```
elastic -v --cij proc calc-cij_000/vasprun.xml calc-cij_*/vasprun.xml
```

You can test this procedure using data provided as a reference in the `tests/data` directory. If you run the script on the provided data you should get following output:

```
elastic -v --cij proc calc-cij_000/vasprun.xml calc-cij_*/vasprun.xml

Cij solution
-----------------------------
 Solution rank:  3
 Square of residuals: 0.00053
 Relative singular values:
 1.0000   0.7071   0.6354

Elastic tensor (GPa):
   C_11     C_12     C_44
-----------------------------
 321.15    95.88   143.44
```

The data provided correspond to cubic MgO crystal. The DFT calculation setup is tuned to provide quick results for testing and *should not* be used as a guide for production calculations. You *need* to determine proper calculation setup for your system.

CHAPTER 4

Library usage

4.1 Simple Parallel Calculation

Once you have everything installed and running you can run your first real calculation. The first step is to import the modules to your program (the examples here use VASP calculator)

```
from ase.spacegroup import crystal
import ase.units as units
import numpy
import matplotlib.pyplot as plt

from parcalc import ClusterVasp, ParCalculate

from elastic import get_pressure, BMEOS, get_strain
from elastic import get_elementary_deformations, scan_volumes
from elastic import get_BM_EOS, get_elastic_tensor
```

next we need to create our example MgO crystal:

```
a = 4.194
cryst = crystal(['Mg', 'O'],
                [(0, 0, 0), (0.5, 0.5, 0.5)],
                spacegroup=225,
                cellpar=[a, a, a, 90, 90, 90])
```

We need a calculator for our job, here we use VASP and ClusterVasp defined in the parcalc module. You can probably replace this calculator by any other ASE calculator but this was not tested yet. Thus let us define the calculator:

```
# Create the calculator running on one, eight-core node.
# This is specific to the setup on my cluster.
# You have to adapt this part to your environment
calc = ClusterVasp(nodes=1, ppn=8)

# Assign the calculator to the crystal
```

```
cryst.set_calculator(calc)

# Set the calculation parameters
calc.set(prec = 'Accurate', xc = 'PBE', lreal = False,
            nsw=30, ediff=1e-8, ibrion=2, kpts=[3,3,3])

# Set the calculation mode first.
# Full structure optimization in this case.
# Not all calculators have this type of internal minimizer!
calc.set(isif=3)
```

Finally, run our first calculation. Obtain relaxed structure and residual pressure after optimization:

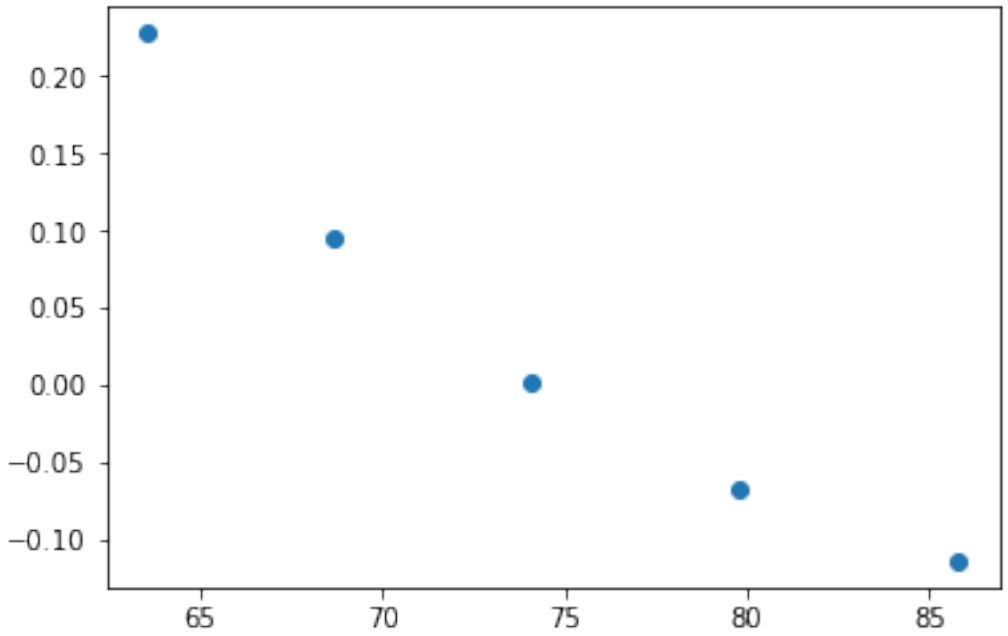
```
print("Residual pressure: %.3f bar" % (
        get_pressure(cryst.get_stress())))
```

```
Residual pressure: 0.000 bar
```

If this returns proper pressure (close to zero) we can use the obtained structure for further calculations. For example we can scan the volume axis to obtain points for equation of state fitting. This will demonstrate the ability to run several calculations in parallel - if you have a cluster of machines at your disposal this will speed up the calculation considerably.

```
# Lets extract optimized lattice constant.
# MgO is cubic so a is a first diagonal element of lattice matrix
a=cryst.get_cell()[0,0]

# Clean up the directory
calc.clean()

systems=[]
# Iterate over lattice constant in the +/-5% range
for av in numpy.linspace(a*0.95,a*1.05,5):
    systems.append(crystal(['Mg', 'O'], [(0, 0, 0), (0.5, 0.5, 0.5)],
                    spacegroup=225, cellpar=[av, av, av, 90, 90, 90]))

# Define the template calculator for this run
# We can use the calc from above. It is only used as a template.
# Just change the params to fix the cell volume
calc.set(isif=2)

# Run the calculation for all systems in sys in parallel
# The result will be returned as list of systems res
res=ParCalculate(systems,calc)

# Collect the results
v=[]
p=[]
for s in res :
    v.append(s.get_volume())
    p.append(get_pressure(s.get_stress()))

# Plot the result (you need matplotlib for this
plt.plot(v,p,'o')
plt.show()
```

```
Workers started: 5
```



4.2 Birch-Murnaghan Equation of State

Let us now use the tools provided by the modules to calculate equation of state for the crystal and verify it by plotting the data points against fitted EOS curve. The EOS used by the module is a well established Birch-Murnaghan formula (P - pressure, V - volume, B - parameters):

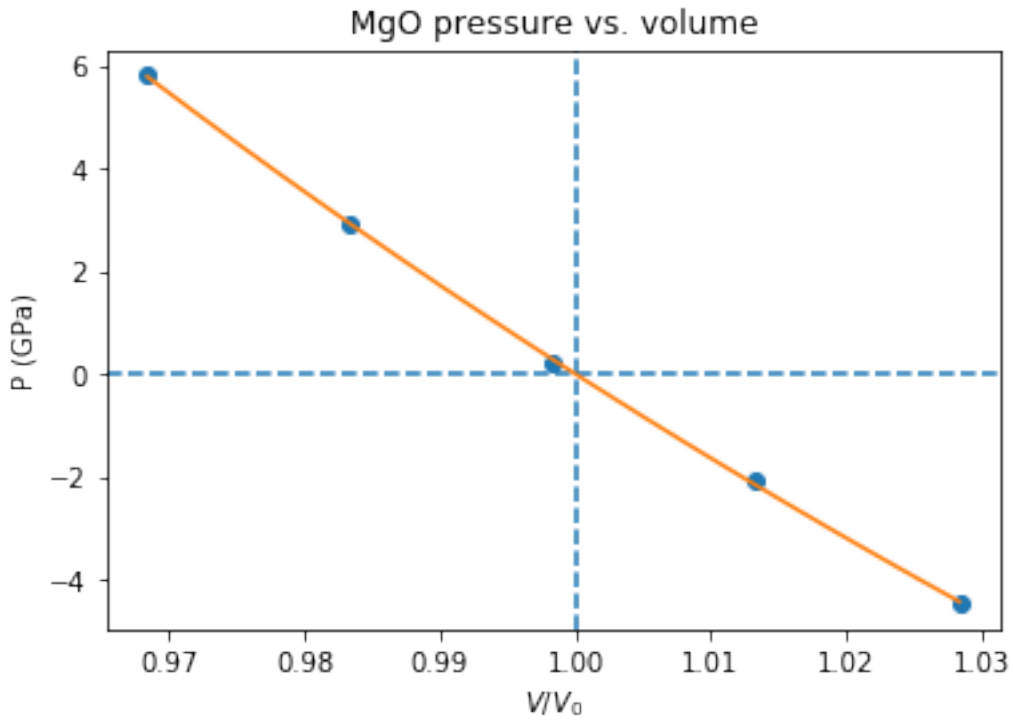
$$P(V) = \frac{B_0}{B_0'}\left[\left(\frac{V}{V_0}\right)^{-B_0'} - 1\right]$$

Now we repeat the setup and optimization procedure from the example 1 above but using a new Crystal class (see above we skip this part for brevity). Then comes a new part (IDOF - Internal Degrees of Freedom):

```
# Switch to cell shape+IDOF optimizer
calc.set(isif=4)

# Calculate few volumes and fit B-M EOS to the result
# Use +/-3% volume deformation and 5 data points
deform=scan_volumes(cryst, n=5,lo=0.97,hi=1.03)

# Run the calculations - here with Cluster VASP
res=ParCalculate(deform,calc)

# Post-process the results
fit=get_BM_EOS(cryst, systems=res)

# Get the P(V) data points just calculated
pv=numpy.array(cryst.pv)

# Sort data on the first column (V)
```

```
pv=pv[pv[:, 0].argsort()]

# Print just fitted parameters
print("V0=%.3f A^3 ; B0=%.2f GPa ; B0'=%.3f ; a0=%.5f A" % (
        fit[0], fit[1]/units.GPa, fit[2], pow(fit[0],1./3)))

v0=fit[0]

# B-M EOS for plotting
fitfunc = lambda p, x: numpy.array([BMEOS(xv,p[0],p[1],p[2]) for xv in x])

# Ranges - the ordering in pv is not guarateed at all!
# In fact it may be purely random.
x=numpy.array([min(pv[:,0]),max(pv[:,0])])
y=numpy.array([min(pv[:,1]),max(pv[:,1])])


# Plot the P(V) curves and points for the crystal
# Plot the points
plt.plot(pv[:,0]/v0,pv[:,1]/units.GPa,'o')

# Mark the center P=0 V=V0
plt.axvline(1,ls='--')
plt.axhline(0,ls='--')

# Plot the fitted B-M EOS through the points
xa=numpy.linspace(x[0],x[-1],20)
plt.plot(xa/v0,fitfunc(fit,xa)/units.GPa,'-')
plt.title('MgO pressure vs. volume')
plt.xlabel('$V/V_0$')
plt.ylabel('P (GPa)')
plt.show()
```

```
Workers started: 5
V0=74.233 A^3 ; B0=168.19 GPa ; B0'=4.270 ; a0=4.20275 A
```



If you set up everything correctly you should obtain fitted parameters printed out in the output close to:

$$V_0 = 73.75 \text{ A}^3 \quad B_0 = 170 \text{ GPa} \quad B_0' = 4.3 \quad a_0 = 4.1936 \text{ A}$$

4.3 Calculation of the elastic tensor

Finally let us calculate an elastic tensor for the same simple cubic crystal - magnesium oxide (MgO). For this we need to create the crystal and optimize its structure (see :ref:`parcalc` above). Once we have an optimized structure we can switch the calculator to internal degrees of freedom optimization (IDOF) and calculate the elastic tensor:

```
# Switch to IDOF optimizer
calc.set(isif=2)

# Create elementary deformations
systems = get_elementary_deformations(cryst, n=5, d=0.33)

# Run the stress calculations on deformed cells
res = ParCalculate(systems, calc)

# Elastic tensor by internal routine
Cij, Bij = get_elastic_tensor(cryst, systems=res)
print("Cij (GPa):", Cij/units.GPa)
```

```
Workers started: 10
Cij (GPa): [ 338.46921273  103.64272667  152.2150523 ]
```

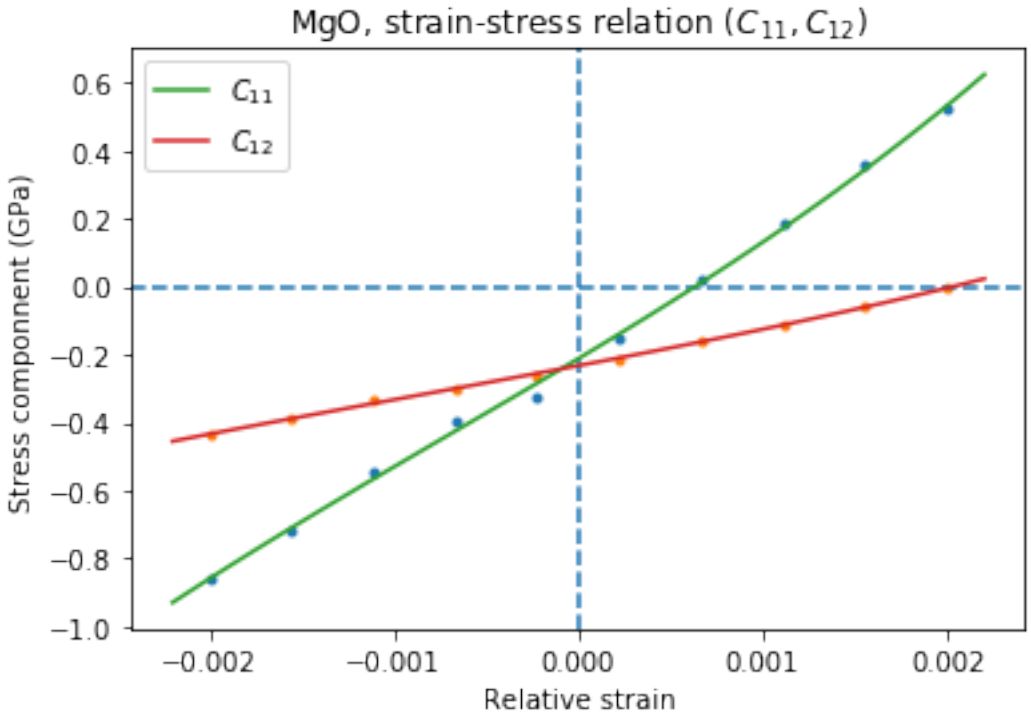
To make sure we are getting the correct answer let us make the calculation for C_{11}, C_{12} by hand. We will deform the cell along a (x) axis by +/-0.2% and fit the 3^{rd} order polynomial to the stress-strain data. The linear component of the fit is the element of the elastic tensor:

```
from elastic.elastic import get_cart_deformed_cell

# Create 10 deformation points on the a axis
systems = []
for d in numpy.linspace(-0.2,0.2,10):
    systems.append(get_cart_deformed_cell(cryst, axis=0, size=d))

# Calculate the systems and collect the stress tensor for each system
r = ParCalculate(systems, cryst.calc)
ss=[]
for s in r:
    ss.append([get_strain(s, cryst), s.get_stress()])

ss=numpy.array(ss)
lo=min(ss[:,0,0])
hi=max(ss[:,0,0])
mi=(lo+hi)/2
wi=(hi-lo)/2
xa=numpy.linspace(mi-1.1*wi,mi+1.1*wi, 50)
```

```
Workers started: 10
```

```
# Make a plot
plt.plot(ss[:,0,0],ss[:,1,0]/units.GPa,'.')
plt.plot(ss[:,0,0],ss[:,1,1]/units.GPa,'.')

plt.axvline(0,ls='--')
plt.axhline(0,ls='--')

# Now fit the polynomials to the data to get elastic constants
# C11 component
f=numpy.polyfit(ss[:,0,0],ss[:,1,0],3)
c11=f[-2]/units.GPa

# Plot the fitted function
plt.plot(xa,numpy.polyval(f,xa)/units.GPa,'-', label='$C_{11}$')

# C12 component
f=numpy.polyfit(ss[:,0,0],ss[:,1,1],3)
c12=f[-2]/units.GPa

# Plot the fitted function
plt.plot(xa,numpy.polyval(f,xa)/units.GPa,'-', label='$C_{12}$')
plt.xlabel('Relative strain')
plt.ylabel('Stress componnent (GPa)')
plt.title('MgO, strain-stress relation ($C_{11}, C_{12}$)')
plt.legend(loc='best')
# Here are the results. They should agree with the results
# of the internal routine.
print('C11 = %.3f GPa, C12 = %.3f GPa => K= %.3f GPa' % (
        c11, c12, (c11+2*c12)/3))

plt.show()
```

```
C11 = 325.005 GPa, C12 = 102.441 GPa => K= 176.629 GPa
```



If you set up everything correctly you should obtain fitted parameters printed out in the output close to:

```
Cij (GPa): [ 340   100   180]
```

With the following result of fitting:

```
C11 = 325 GPa, C12 = 100 GPa => K= 180 GPa
```

The actual numbers depend on the details of the calculations setup but should be fairly close to the above results.

CHAPTER 5

Implementation

Elastic is implemented as an extension module to ASE system

The Elastic package provides, basically, one main python module and one auxiliary module (*Parallel Calculator Module*) which can be useful outside of the scope of the main code. The *Parallel Calculator Module* is not distributed separately but can be just placed by itself somewhere in your python path and used with any part of the ASE. I hope it will be incorporated in the main project sometime in the future.

5.1 Modules

5.1.1 Parallel Calculator Module

Parallel calculator module is an extension of the standard ASE calculator working in the parallel cluster environment. It is very useful in all situations where you need to run several, independent calculations and you have a large cluster of machines at your disposal (probably with some queuing system).

This implementation uses VASP but the code can be easily adapted for use with other ASE calculators with minor changes. The final goal is to provide a universal module for parallel calculator execution in the cluster environment.

The SIESTA code by Georgios Tritsaris <gtritsaris@seas.harvard.edu> Not fully tested after merge.

class parcalc.parcalc.**ClusterAims**(*nodes=1*, *ppn=8*, ***kwargs*)
Encapsulating Aims calculator for the cluster environment.

class parcalc.parcalc.**ClusterSiesta**(*nodes=1*, *ppn=8*, ***kwargs*)
Siesta calculator. Not fully tested by me - so this should be considered beta quality. Nevertheless it is based on working implementation

class parcalc.parcalc.**ClusterVasp**(*nodes=1*, *ppn=8*, *block=True*, *ncl=False*, ***kwargs*)
Adaptation of VASP calculator to the cluster environment where you often have to make some preparations before job submission. You can easily adapt this class to your particular environment. It is also easy to use this as a template for other type of calculator.

calc_finished()
Check if the lockfile is in the calculation directory. It is removed by the script at the end regardless of

the success of the calculation. This is totally tied to implementation and you need to implement your own scheme!

calculate(*atoms*)
Blocking/Non-blocking calculate method

If we are in blocking mode we just run, wait for the job to end and read in the results. Easy ...

The non-blocking mode is a little tricky. We need to start the job and guard against it reading back possible old data from the directory - the queuing system may not even started the job when we get control back from the starting script. Thus anything we read after invocation is potentially garbage - even if it is a converged calculation data.

We handle it by custom run function above which raises an exception after submitting the job. This skips post-run processing in the calculator, preserves the state of the data and signals here that we need to wait for results.

prepare_calc_dir()
Prepare the calculation directory for VASP execution. This needs to be re-implemented for each local setup. The following code reflects just my particular setup.

run()
Blocking/Non-blocing run method. In blocking mode it just runs parent run method. In non-blocking mode it raises the __NonBlockingRunException to bail out of the processing of standard calculate method (or any other method in fact) and signal that the data is not ready to be collected.

parcalc.parcalc.**ParCalculate**(*systems*, *calc*, *cleanup=True*, *block=True*, *prefix='Calc_'*)
Run calculators in parallel for all systems. Calculators are executed in isolated processes and directories. The resulting objects are returned in the list (one per input system).

class parcalc.parcalc.**RemoteCalculator**(*restart=None*, *ignore_bad_restart_file=False*, *label=None*, *atoms=None*, *calc=None*, *block=False*, ***kwargs*)
Remote calculator based on ASE calculator class. This class is only involved with the machanics of remotly executing the software and transporting the data. The calculation is delegated to the actual calculator class.

classmethod ParallelCalculate(*syslst, properties=['energy'], system_changes=['positions', 'numbers', 'cell', 'pbc', 'initial_charges', 'initial_magmoms']*)
Run a series of calculations in parallel using (implicitely) some remote machine/cluster. The function returns the list of systems ready for the extraction of calculated properties.

read_results()
Read energy, forces, ... from output file(s).

run_calculation(*atoms=None, properties=['energy'], system_changes=['positions', 'numbers', 'cell', 'pbc', 'initial_charges', 'initial_magmoms']*)
Internal calculation executor. We cannot use FileIOCalculator directly since we need to support remote execution.

This calculator is different from others. It prepares the directory, launches the remote process and raises the exception to signal that we need to come back for results when the job is finished.

write_input(*atoms=None, properties=['energy'], system_changes=['positions', 'numbers', 'cell', 'pbc', 'initial_charges', 'initial_magmoms']*)
Write input file(s).

parcalc.parcalc.**work_dir**(**args*, ***kwds*)
Context menager for executing commands in some working directory. Returns to the previous wd when finished.

Usage: >>> with work_dir(path): ... subprocess.call('git status')

5.1.2 Elastic Module

Elastic is a module for calculation of C_{ij} components of elastic tensor from the strain-stress relation.

The strain components here are ordered in standard way which is different to ordering in previous versions of the code (up to 4.0). The ordering is: $u_{xx}, u_{yy}, u_{zz}, u_{yz}, u_{xz}, u_{xy}$.

The general ordering of C_{ij} components is (except for triclinic symmetry and taking into account customary names of constants - e.g. $C_{16} \to C_{14}$):

$$C_{11}, C_{22}, C_{33}, C_{12}, C_{13}, C_{23}, C_{44}, C_{55}, C_{66}, C_{16}, C_{26}, C_{36}, C_{45}$$

The functions with the name of bravais lattices define the symmetry of the C_{ij} matrix. The matrix is N columns by 6 rows where the columns corespond to independent elastic constants of the given crystal, while the rows corespond to the canonical deformations of a crystal. The elements are the second partial derivatives of the free energy formula for the crystal written down as a quadratic form of the deformations with respect to elastic constant and deformation.

Note: The elements for deformations u_{xy}, u_{xz}, u_{yz} have to be divided by 2 to properly match the usual definition of elastic constants.

See: *[LL]* L.D. Landau, E.M. Lifszyc, "Theory of elasticity"

There is some usefull summary also at: ScienceWorld

`elastic.elastic.`**`get_BM_EOS`**(*cryst*, *systems*)

Calculate Birch-Murnaghan Equation of State for the crystal.

The B-M equation of state is defined by:

$$P(V) = \frac{B_0}{B_0'}\left[\left(\frac{V}{V_0}\right)^{-B_0'} - 1\right]$$

It's coefficients are estimated using n single-point structures ganerated from the crystal (cryst) by the scan_volumes function between two relative volumes. The BM EOS is fitted to the computed points by least squares method. The returned value is a list of fitted parameters: V_0, B_0, B_0' if the fit succeded. If the fitting fails the `RuntimeError('Calculation failed')` is raised. The data from the calculation and fit is stored in the bm_eos and pv members of cryst for future reference. You have to provide properly optimized structures in cryst and systems list.

Parameters

- **`cryst`** – Atoms object, basic structure
- **`systems`** – A list of calculated structures

Returns tuple of EOS parameters V_0, B_0, B_0'.

`elastic.elastic.`**`get_bulk_modulus`**(*cryst*)

Calculate bulk modulus using the Birch-Murnaghan equation of state.

The EOS must be previously calculated by get_BM_EOS routine. The returned bulk modulus is a B_0 coefficient of the B-M EOS. The units of the result are defined by ASE. To get the result in any particular units (e.g. GPa) you need to divide it by ase.units.<unit name>:

```
get_bulk_modulus(cryst)/ase.units.GPa
```

Parameters **`cryst`** – ASE Atoms object

Returns float, bulk modulus B_0 in ASE units.

elastic.elastic.**get_cart_deformed_cell**(*base_cryst*, *axis=0*, *size=1*)
Return the cell deformed along one of the cartesian directions

Creates new deformed structure. The deformation is based on the base structure and is performed along single axis. The axis is specified as follows: 0,1,2 = x,y,z ; sheers: 3,4,5 = yz, xz, xy. The size of the deformation is in percent and degrees, respectively.

Parameters

- **base_cryst** – structure to be deformed
- **axis** – direction of deformation
- **size** – size of the deformation

Returns new, deformed structure

elastic.elastic.**get_cij_order**(*cryst*)
Give order of of elastic constants for the structure

Parameters **cryst** – ASE Atoms object

Returns Order of elastic constants as a tuple of strings: C_ij

elastic.elastic.**get_deformed_cell**(*base_cryst*, *axis=0*, *size=1*)
Return the cell (with atoms) deformed along one cell parameter (0,1,2 = a,b,c ; 3,4,5 = alpha,beta,gamma) by size percent or size degrees (axis/angles).

elastic.elastic.**get_elastic_tensor**(*cryst*, *systems*)
Calculate elastic tensor of the crystal.

The elastic tensor is calculated from the stress-strain relation and derived by fitting this relation to the set of linear equations build from the symmetry of the crystal and strains and stresses of the set of elementary deformations of the unit cell.

It is assumed that the crystal is converged and optimized under intended pressure/stress. The geometry and stress on the cryst is taken as the reference point. No additional optimization will be run. Structures in cryst and systems list must have calculated stresses. The function returns tuple of C_{ij} elastic tensor, raw Birch coefficients B_{ij} and fitting results: residuals, solution rank, singular values returned by numpy.linalg.lstsq.

Parameters

- **cryst** – Atoms object, basic structure
- **systems** – list of Atoms object with calculated deformed structures

Returns

tuple(C_{ij} float vector, tuple(B_{ij} float vector,

residuals, solution rank, singular values)

elastic.elastic.**get_elementary_deformations**(*cryst*, *n=5*, *d=2*)
Generate elementary deformations for elastic tensor calculation.

The deformations are created based on the symmetry of the crystal and are limited to the non-equivalet axes of the crystal.

Parameters

- **cryst** – Atoms object, basic structure
- **n** – integer, number of deformations per non-equivalent axis
- **d** – float, size of the maximum deformation in percent and degrees

Returns list of deformed structures

elastic.elastic.**get_lattice_type**(*cryst*)
Find the symmetry of the crystal using spglib symmetry finder.

Derive name of the space group and its number extracted from the result. Based on the group number identify also the lattice type and the Bravais lattice of the crystal. The lattice type numbers are (the numbering starts from 1):

Triclinic (1), Monoclinic (2), Orthorombic (3), Tetragonal (4), Trigonal (5), Hexagonal (6), Cubic (7)

Parameters **cryst** – ASE Atoms object

Returns tuple (lattice type number (1-7), lattice name, space group name, space group number)

elastic.elastic.**get_pressure**(*s*)
Return *external* isotropic (hydrostatic) pressure in ASE units.

If the pressure is positive the system is under external pressure. This is a convenience function to convert output of get_stress function into external pressure.

Parameters **cryst** – stress tensor in Voight (vector) notation as returned by the get_stress() method.

Returns float, external hydrostatic pressure in ASE units.

elastic.elastic.**get_strain**(*cryst*, *refcell=None*)
Calculate strain tensor in the Voight notation

Computes the strain tensor in the Voight notation as a conventional 6-vector. The calculation is done with respect to the crystal geometry passed in refcell parameter.

Parameters

- **cryst** – deformed structure
- **refcell** – reference, undeformed structure

Returns 6-vector of strain tensor in the Voight notation

elastic.elastic.**get_vecang_cell**(*cryst*, *uc=None*)
Compute A,B,C, alpha,beta,gamma cell params from the unit cell matrix (uc) or cryst. Angles in radians.

elastic.elastic.**hexagonal**(*u*)
The matrix is constructed based on the approach from L&L using auxiliary coordinates: $\xi = x + iy$, $\eta = x - iy$. The components are calculated from free energy using formula introduced in *Crystal symmetry and elastic matrix derivation* with appropriate coordinate changes. The order of constants is as follows:

$$C_{11}, C_{33}, C_{12}, C_{13}, C_{44}$$

Parameters **u** – vector of deformations: [$u_{xx}, u_{yy}, u_{zz}, u_{yz}, u_{xz}, u_{xy}$]

Returns Symmetry defined stress-strain equation matrix

elastic.elastic.**monoclinic**(*u*)
Monoclinic group,

The ordering of constants is:

$$C_{11}, C_{22}, C_{33}, C_{12}, C_{13}, C_{23}, C_{44}, C_{55}, C_{66}, C_{16}, C_{26}, C_{36}, C_{45}$$

Parameters **u** – vector of deformations: [$u_{xx}, u_{yy}, u_{zz}, u_{yz}, u_{xz}, u_{xy}$]

Returns Symmetry defined stress-strain equation matrix

elastic.elastic.**orthorombic**(*u*)
Equation matrix generation for the orthorombic lattice. The order of constants is as follows:

$$C_{11}, C_{22}, C_{33}, C_{12}, C_{13}, C_{23}, C_{44}, C_{55}, C_{66}$$

Parameters **u** – vector of deformations: [$u_{xx}, u_{yy}, u_{zz}, u_{yz}, u_{xz}, u_{xy}$]

Returns Symmetry defined stress-strain equation matrix

elastic.elastic.**regular**(*u*)
Equation matrix generation for the regular (cubic) lattice. The order of constants is as follows:

$$C_{11}, C_{12}, C_{44}$$

Parameters **u** – vector of deformations: [$u_{xx}, u_{yy}, u_{zz}, u_{yz}, u_{xz}, u_{xy}$]

Returns Symmetry defined stress-strain equation matrix

elastic.elastic.**scan_pressures**(*cryst*, *lo*, *hi*, *n=5*, *eos=None*)
Scan the pressure axis from lo to hi (inclusive) using B-M EOS as the volume predictor. Pressure (lo, hi) in GPa

elastic.elastic.**scan_volumes**(*cryst*, *lo=0.98*, *hi=1.02*, *n=5*, *scale_volumes=True*)
Provide set of crystals along volume axis from lo to hi (inclusive). No volume cell optimization is performed. Bounds are specified as fractions (1.10 = 10% increase). If scale_volumes==False the scalling is applied to lattice vectors instead of volumes.

Parameters

- **lo** – lower bound of the V/V_0 in the scan
- **hi** – upper bound of the V/V_0 in the scan
- **n** – number of volume sample points
- **scale_volumes** – If True scale the unit cell volume or, if False, scale the length of lattice axes.

Returns a list of deformed systems

elastic.elastic.**tetragonal**(*u*)
Equation matrix generation for the tetragonal lattice. The order of constants is as follows:

$$C_{11}, C_{33}, C_{12}, C_{13}, C_{44}, C_{14}$$

Parameters **u** – vector of deformations: [$u_{xx}, u_{yy}, u_{zz}, u_{yz}, u_{xz}, u_{xy}$]

Returns Symmetry defined stress-strain equation matrix

elastic.elastic.**triclinic**(*u*)
Triclinic crystals.

Note: This was never tested on the real case. Beware!

The ordering of constants is:

$$C_{11}, C_{22}, C_{33}, C_{12}, C_{13}, C_{23}, C_{44}, C_{55}, C_{66}, C_{16}, C_{26}, C_{36}, C_{46}, C_{56}, C_{14}, C_{15}, C_{25}, C_{45}$$

Parameters **u** – vector of deformations: [$u_{xx}, u_{yy}, u_{zz}, u_{yz}, u_{xz}, u_{xy}$]

Returns Symmetry defined stress-strain equation matrix

elastic.elastic.**trigonal**(*u*)

The matrix is constructed based on the approach from L&L using auxiliary coordinates: $\xi = x + iy$, $\eta = x - iy$. The components are calculated from free energy using formula introduced in *Crystal symmetry and elastic matrix derivation* with appropriate coordinate changes. The order of constants is as follows:

$$C_{11}, C_{33}, C_{12}, C_{13}, C_{44}, C_{14}$$

Parameters **u** – vector of deformations: [$u_{xx}, u_{yy}, u_{zz}, u_{yz}, u_{xz}, u_{xy}$]

Returns Symmetry defined stress-strain equation matrix

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

CHAPTER 7

References

The Elastic package should be cited using one or both of the following papers (TiC, ZrC) and its own reference.:

- search

Bibliography

[Elastic] P.T. Jochym, *Module for calculating elastic tensor of crystals*, software, https://github.com/jochym/Elastic/, doi:10.5281/zenodo.18759.

[TiC] P.T. Jochym, K. Parlinski and M. Sternik, *TiC lattice dynamics from ab initio calculations*, European Physical Journal B; **10**, 1 (1999) 9-13 ; doi:10.1007/s100510050823

[ZrC] P.T. Jochym and K. Parlinski, *Ab initio lattice dynamics and elastic constants of ZrC*, European Physical Journal B; **15**, 2 (2000) 265-268 ; doi:10.1007/s100510051124

[LL] L.D. Landau, E.M. Lifszyc, Theory of elasticity, Elsevier (1986) ; ISBN: 075062633X, 9780750626330

[SVD] G. Golub and W. Kahan, *Calculating the Singular Values and Pseudo-Inverse of a Matrix*, J. Soc. Indus,.Appl. Math.: Ser. B **2**, (1964) pp. 205-224 ; doi:10.1137/0702016 ; Wikipedia article on SVD

Python Module Index

e

p

Index