
Elastic

Release v4.0.28-2-g1c3d54c

Paweł T. Jochym

May 15, 2016

1	Physical Principles	3
1.1	Elasticity of crystals	3
1.2	Numerical derivation of elastic matrix	4
1.3	Crystal symmetry and elastic matrix derivation	4
2	Implementation	7
2.1	Modules	7
2.1.1	Parallel Calculator Module	7
2.1.2	Elastic Module	8
3	Installation	11
3.1	Testing	11
4	Usage	13
4.1	Simple Parallel Calculation	13
4.2	Birch-Murnaghan Equation of State	14
4.3	Calculation of the elastic tensor	16
5	Indices and tables	19
6	References	21
	Bibliography	23
	Python Module Index	25

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 third version of the in-house code I have written over several years and is implemented as a extension to the [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. Nevertheless the old code is still available to anyone as [Elastic 2](#). I just do not recommend to use it without my help - which I am happy to provide.

In 2010, I have decided to re-implement elastic as a module for the [ASE](#) system and publish it properly under the GPL.

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). You can obtain the [documentation as a PDF file](#) as well.

The project is open and I welcome patches, ideas and other feedback. You can also support the project and motivate me to work on it even more by donating using bitcoin address: 1Geq8khANDueVt1QdCS5GU2oNCtdc1uSMv .

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 $i, j = 1 \dots 6$. 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:

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

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 than 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 straightforward 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): $C_{ij}s_j^a = \sigma_i^a$. 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_j^a create a new equation matrix S . $S_{ju}(s^a)C_u = \sigma_j^a$. 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](#).

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.

2.1 Modules

2.1.1 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 Tritsaridis <gtritsaris@seas.harvard.edu> Not fully tested after merge.

Class description

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 b 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).

```
parcalc.parcalc.work_dir(*args, **kws)
```

Context manager for executing commands in some working directory. Returns to the previous wd when finished.

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

2.1.2 Elastic Module

This module depends on *Parallel Calculator Module* for parallelisation of independent calculations.

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.

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. $C_{16} \rightarrow C_{14}$):

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

The functions outside of the Crystal class define the symmetry of the C_{ij} matrix. The matrix is N columns by 6 rows where the columns correspond to independent elastic constants of the given crystal, while the rows correspond 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](#)

Class description

```
class elastic.elastic.Crystal(symbols=None, positions=None, numbers=None, tags=None,
                             momenta=None, masses=None, magmoms=None,
                             charges=None, scaled_positions=None, cell=None, pbc=None,
                             celldisp=None, constraint=None, calculator=None, info=None)
```

Backward compatibility class. To be removed later.

class elastic.elastic.ElasticCrystal

Mixin extension of standard ASE Atoms class designed to handle specifics of the crystalline materials. This code should, in principle, be folded into the Atoms class in the future. At this moment it is too early to think about it. Additionally there are some aspects of this code which may be difficult to harmonize with the principles of the Atoms class. I am sure it is better, for now to leave this as a separate extension class.

Basically, this class provides set of functions concerned with derivation of elastic properties using “finite deformation approach” (see the documentation for physics background information).

get_BM_EOS (*n=5, lo=0.98, hi=1.02, recalc=False, cleanup=True, mode='full', data=None*)

Calculate Birch-Murnaghan Equation of State for the crystal:

$$P(V) = \frac{B_0}{B'_0} \left[\left(\frac{V}{V_0} \right)^{-B'_0} - 1 \right]$$

using *n* single-point structures generated from the crystal (self) by the scan_volumes method between *lo* and *hi* 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 succeeded. 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 for future reference.

Note: For now you have to set up the calculator to properly optimize the structure without changing the volume at each point. There will be a way to specify basic types of the calculator minimization at the later stage.

get_bulk_modulus (*n=5, lo=0.98, hi=1.02, recalc=False*)

Calculate bulk modulus using the Birch-Murnaghan equation of state data calculated by *get_BM_EOS* routine (see). The returned bulk modulus is a B_0 coefficient of the B-M EOS. The arguments are the same as in BM EOS function.

get_cart_deformed_cell (*axis=0, size=1*)

Return the cell (with atoms) deformed along one of the cartesian directions (0,1,2 = x,y,z ; sheers: 3,4,5 = yz, xz, xy) by size percent.

get_deformed_cell (*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).

get_elastic_tensor (*n=5, d=2, mode='full', systems=None*)

Calculate elastic tensor of the crystal. It is assumed that the crystal is converged and optimized under intended pressure/stress. The geometry and stress at the call point is taken as the reference point. No additional optimization will be run. It is also assumed that the calculator is set to pure IDOF optimization. The size of used finite deformation is passed in *d* parameter as a percentage relative deformation. The *n* parameter defines number of deformed structures used in the calculation.

get_lattice_type ()

Find the symmetry of the crystal using spglib symmetry finder. Assign to *sg_name* i *sg_nr* members name of the space group and its number extracted from the result. Based on the group number identify also the lattice type (assigned to *sg_type* member) and the Bravais lattice of the crystal (assigned to *bravais* member). The returned value is the lattice type number. The lattice type numbers are (see also Crystal.ls, the numbering starts from 1):

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

get_pressure (*s=None*)

Return *external* isotropic (hydrostatic) pressure in ASE units. If the pressure is positive the system is under external pressure. This is a convenience function.

get_strain (*refcell=None*)

Return 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.

get_vecang_cell (*uc=None*)

Compute A,B,C, alpha,beta,gamma cell params from the unit cell matrix (*uc*) or self. Angles in radians.

scan_pressures (*lo, hi, n=5*)

Scan the pressure axis from *lo* to *hi* (inclusive) using B-M EOS as the volume predictor. Pressure (*lo*, *hi*) in GPa

scan_volumes (*lo, hi, n*)

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).

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:

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

elastic.elastic.monoclinic (*u*)

Monoclinic group, the ordering of constants is:

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

elastic.elastic.orthorombic (*u*)

Equation matrix generation for the orthorombic lattice. The order of constants is as follows:

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

elastic.elastic.regular (*u*)

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

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

elastic.elastic.tetragonal (*u*)

Equation matrix generation for the tetragonal lattice. The order of constants is as follows:

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

elastic.elastic.triclinic (*u*)

Triclinic crystals.

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

The ordering of constants is:

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

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:

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

Installation

The installation procedure is quite simple if you use, *highly recommended* [conda package manager](#)

```
conda install -c jochym 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 jochym
```

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

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
- 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.
- [spglib](#) space group library
- [pyspglib](#) python space group module

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](#).

3.1 Testing

All modules have small testing sets at the end. You can run these test by simply running each module as a python script:

```
python parcalc.py
```

which will run a short series of single-point calculations on the MgO unit cell and plot the resulting equation of state.

The main module testing routine:

```
python elastic.py
```

will run the equation of state and elastic tensor calculations for a set of small crystals - one for each Bravais lattice. This may take some considerable time.

The testing routines will probably not work out of the box in your system. Review the comments at the end of the files to make them work. I will try to make them as setup-agnostic as possible.

Usage

In this section we assume that you have all parts of ASE properly installed and the elastic is installed and working properly. The examples are available in the [example subdirectory](#). The code below use also scipy, numpy and matplotlib functions. The VASP calculator is used in all examples (at least for now).

IPython notebook with [additional example](#) presents calculation using QE-util package

4.1 Simple Parallel Calculation

Once you have everything installed and running you can run your first real calculation. The testing code at the end of the parcalc.py may be used as an example how to do it. The first step is to import the modules to your program (the examples here use VASP calculator):

```
from ase.lattice.spacegroup import crystal
from parcalc import ClusterVasp, ParCalculate
import ase.units as units
import numpy
import matplotlib.pyplot as plt
```

next we need to create the crystal, MgO in this case:

```
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" % (
    cryst.get_isotropic_pressure(cryst.get_stress()))
```

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()

sys=[]
# Iterate over lattice constant in the +/-5% range
for av in numpy.linspace(a*0.95,a*1.05,5):
    sys.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(sys,calc)

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

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

If you set up everything correctly you should obtain plot similar to this:

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):

$$P(V) = \frac{B_0}{B'_0} \left[\left(\frac{V}{V_0} \right)^{-B'_0} - 1 \right]$$

We will start with the same crystal optimized above, but this time we will use a new functionality imported from the elastic module. This module acts as a plug-in for the Atoms class - extending their range of quantities accessible for the user:

```
import elastic
from elastic import BMEOS
```

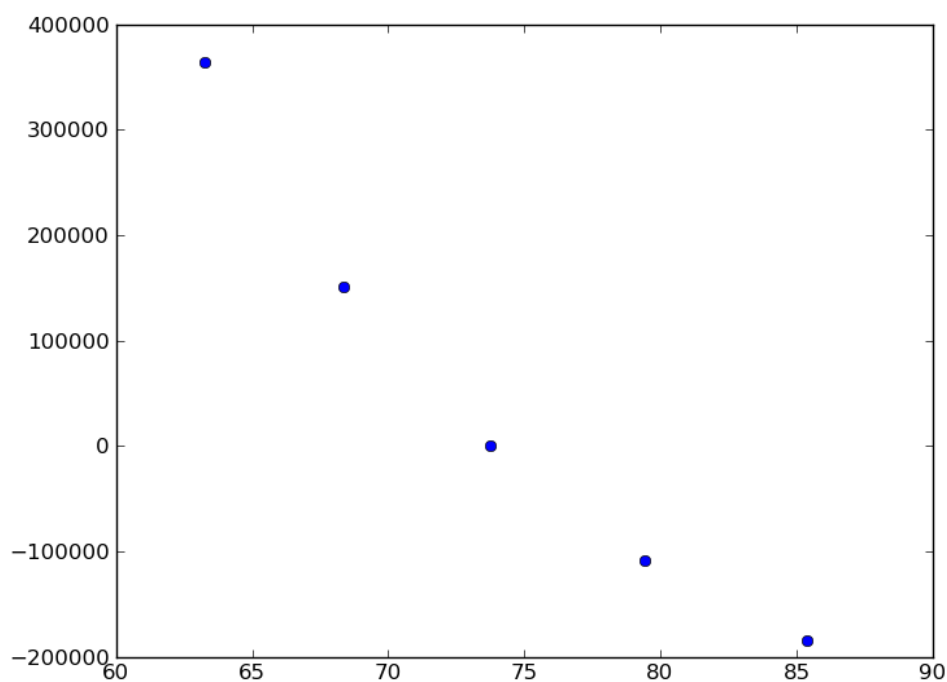


Fig. 4.1: The pressure dependence on volume in MgO crystal (example1.py).

```
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])
```

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
fit=cryst.get_BM_EOS(n=5,lo=0.97,hi=1.03)

# 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: [BMEOS(xv,p[0],p[1],p[2]) for xv in x]

# Ranges - the ordering in pv is not guaranteed 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],'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),'-')
plt.draw()

```

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

$$V_0 = 73.75 \text{ \AA}^3 \quad B_0 = 170 \text{ GPa} \quad B'_0 = 4.3 \quad a_0 = 4.1936 \text{ \AA}$$

and the following (or similar) plot:

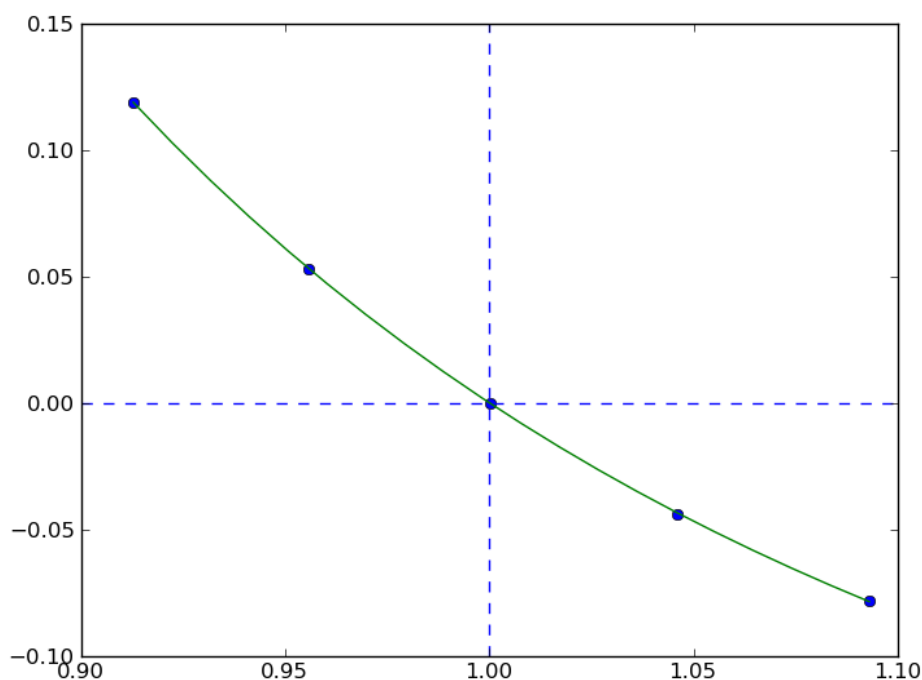


Fig. 4.2: The pressure dependence on volume in MgO crystal (example2.py).

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 [Simple Parallel Calculation](#) 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)

# Elastic tensor by internal routine
Cij, Bij=cryst.get_elastic_tensor(n=5,d=0.33)
print "Cij (GPa):", Cij/units.GPa
```

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 3rd order polynomial to the stress-strain data. The linear component of the fit is the element of the elastic tensor:

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

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

# Plot strain-stress relation
ss=[]
for p in r:
    ss.append([p.get_strain(cryst),p.get_stress()])
ss=array(ss)
lo=min(ss[:,0,0])
hi=max(ss[:,0,0])
mi=(lo+hi)/2
wi=(hi-lo)/2
xa=linspace(mi-1.1*wi,mi+1.1*wi, 50)
plt.plot(ss[:,0,0],ss[:,1,0],'k.')
plt.plot(ss[:,0,0],ss[:,1,1],'r.')

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),'b-')

# 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),'g-')

# 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()
```

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

C_{ij} (GPa): [319.1067 88.8528 139.35852632]

With the following result of fitting:

$$C_{11} = 317.958 \text{ GPa}, C_{12} = 68.878 \text{ GPa} \Rightarrow K = 151.905 \text{ GPa}$$

and the following (or similar) plot:

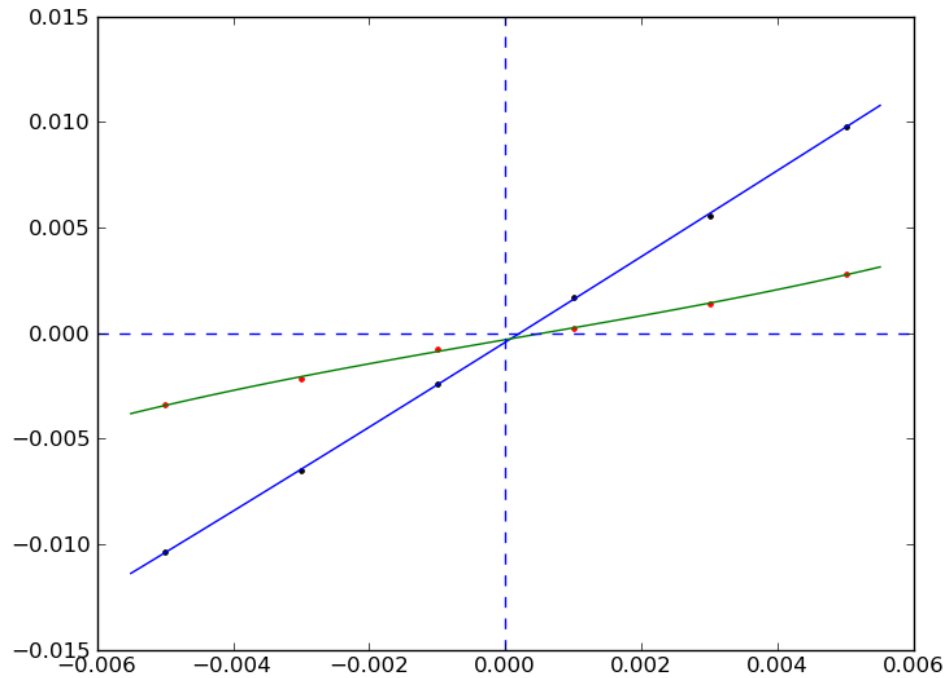


Fig. 4.3: The pressure dependence on volume in MgO crystal (example3.py).

Indices and tables

- `genindex`
- `modindex`
- `search`

References

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

- search

- [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. Indust., Appl. Math.: Ser. B **2**, (1964) pp. 205-224 ; doi:10.1137/0702016 ; Wikipedia article on SVD

e

`elastic.elastic`, 8

p

`parcalc.parcalc`, 7

C

calc_finished() (parcalc.parcalf.ClusterVasp method), 7
calculate() (parcalc.parcalf.ClusterVasp method), 7
ClusterSiesta (class in parcalc.parcalf), 7
ClusterVasp (class in parcalc.parcalf), 7
Crystal (class in elastic.elastic), 8

E

elastic.elastic (module), 8
ElasticCrystal (class in elastic.elastic), 8

G

get_BM_EOS() (elastic.elastic.ElasticCrystal method), 9
get_bulk_modulus() (elastic.elastic.ElasticCrystal method), 9
get_cart_deformed_cell() (elastic.elastic.ElasticCrystal method), 9
get_deformed_cell() (elastic.elastic.ElasticCrystal method), 9
get_elastic_tensor() (elastic.elastic.ElasticCrystal method), 9
get_lattice_type() (elastic.elastic.ElasticCrystal method), 9
get_pressure() (elastic.elastic.ElasticCrystal method), 9
get_strain() (elastic.elastic.ElasticCrystal method), 9
get_vecang_cell() (elastic.elastic.ElasticCrystal method), 9

H

hexagonal() (in module elastic.elastic), 10

M

monoclinic() (in module elastic.elastic), 10

O

orthorombic() (in module elastic.elastic), 10

P

parcalc.parcalf (module), 7
ParCalculate() (in module parcalc.parcalf), 8
prepare_calc_dir() (parcalc.parcalf.ClusterVasp method), 8

R

regular() (in module elastic.elastic), 10
run() (parcalc.parcalf.ClusterVasp method), 8

S

scan_pressures() (elastic.elastic.ElasticCrystal method), 10
scan_volumes() (elastic.elastic.ElasticCrystal method), 10

T

tetragonal() (in module elastic.elastic), 10
triclinic() (in module elastic.elastic), 10
trigonal() (in module elastic.elastic), 10

W

work_dir() (in module parcalc.parcalf), 8