MattPy

MattPy v0.1 User Manual

Updated October 2, 2015

https://github.com/mcaroba/MattPy.git

written by

Miguel A. Caro

Department of Electrical Engineering and Automation; and
COMP Centre of Excellence in Computational Nanoscience
Aalto University, Espoo, Finland
mcaroba@gmail.com, miguel.caro@aalto.fi

MattPy is a collection of **Mat**erial **t**ensor **Py**thon routines designed to help in the manipulation and analysis of material tensors. Currently, it can handle elastic (4-rank) and piezoelectric (3-rank) tensors. Among other functionalities, the user can rotate the tensors or transform from Cartesian to Voigt forms. ***MattPy***'s main capability, and the one for which it was originally designed, is to project and align material tensors of arbitrary symmetry onto a higher (or lower) symmetry tensor. The method was originally proposed by Moakher and Norris [1] in the context of elastic tensor analysis, and I introduced a straightforward angle dependence that allows to optimize the projection [2]. ***MattPy*** is particularly useful in the context of computational materials science involving calculation of alloy properties, where results are typically obtained for supercells where alloying disorder reduces the lattice symmetry from the macroscopic average to the supercell's symmetry (often triclinic). For the scientific details the reader is referred to the original work [1] and my implementation paper [2]. Here I give guidelines for the practical use of the ***MattPy*** library and present sample scripts to perform some of the most usual tasks. For bug reports and (reasonable) feature requests I can be contacted directly on my email.

Contents

1 Getting started

1.1 Downloading and installing *MattPy*

MattPy can be downloaded for free from the official repository which is hosted by GitHub on the URL `https://github.com/mcaroba/MattPy`. The collection of Python functions is contained in the file called `mattpy.py`. In the `doc/` directory the latest version of this documentation is available in PDF form as file `mattpy.pdf`. Other files with (more or less) self-explanatory names might be available from the main directory.
To get started using ***MattPy*** in a "quick and dirty" way, you can simply download `mattpy.py` to a local directory on your machine, and from a Python interactive session or within a Python script simply use the `execfile("/path/to/file/mattpy.py")` command, where `/path/to/file/` is the directory where you downloaded `mattpy.py`. The recommended way to install ***MattPy***, however, is to install it as a Python module so that it can be imported both interactively and from a script without the need to keep track of the source file location. To do this, you can create a directory where you store the `mattpy.py` file and add it to your Python path. In a Linux shell this would be something along these lines:

```
# Create modules directory under your home
mkdir ~/python_modules

# Download mattpy.py
wget https://github.com/mcaroba/MattPy/blob/master/mattpy.py ~/python_modules/.

# Add directory to your Python path
echo "export PYTHONPATH=\${PYTHONPATH}:~/python_modules" >> ~/.bashrc
source ~/.bashrc
```

If you don't use bash as your default shell then you'll need to change to whatever the configuration file is in your case. Now you can import ***MattPy*** as you would do with any other Python module:

```
# Import with mattpy prefix, i.e. function() becomes mattpy.function()
import mattpy

# Import with custom prefix, e.g. function() becomes mp.function()
import mattpy as mp

# Import with mattpy original naming, i.e. function() stays function()
from mattpy import *
```

1.2 Dependencies

MattPy relies on NumPy for all its functionalities and SciPy for some of them. You need to install these packages if you have not already done so.

2 Function usage

The naming convention for most functions tries to be as explicative as possible of the function's purpose. There are two sets of functions: one for elastic tensor manipulation

and another for piezoelectric tensor manipulation. The functions have basically the same name and take the same number of arguments, except for a varying prefix/suffix, either `ela` or `pz`, and the `form` keyword which some functions of the piezo family take as argument. Most functions will print warnings if you're doing something weird, but not always. The default behavior can be changed with the `verbose` keyword which some of the functions accept. `verbose = True` increases the amount of information printed while `verbose = False` switches off all warning information.

2.1 Defining a tensor

A tensor can be defined either in Voigt form or in Cartesian form as a Python (nested) list. The following elastic and piezoelectric tensors

$$C = \begin{pmatrix} C_{11} & C_{12} & C_{13} & C_{14} & C_{15} & C_{16} \\ C_{12} & C_{22} & C_{23} & C_{24} & C_{25} & C_{26} \\ C_{13} & C_{23} & C_{33} & C_{34} & C_{35} & C_{36} \\ C_{14} & C_{24} & C_{34} & C_{44} & C_{45} & C_{46} \\ C_{15} & C_{25} & C_{35} & C_{45} & C_{55} & C_{56} \\ C_{16} & C_{26} & C_{36} & C_{46} & C_{56} & C_{66} \end{pmatrix} \qquad e = \begin{pmatrix} e_{11} & e_{12} & e_{13} & e_{14} & e_{15} & e_{16} \\ e_{21} & e_{22} & e_{23} & e_{24} & e_{25} & e_{26} \\ e_{31} & e_{32} & e_{33} & e_{34} & e_{35} & e_{36} \end{pmatrix}$$

can be defined in the following ways:

```
# Elastic tensor in Voigt form:
c_voigt = [
  [C11, C12, C13, C14, C15, C16], [C12, C22, C23, C24, C25, C26],
  [C13, C23, C33, C34, C35, C36], [C14, C24, C34, C44, C45, C46],
  [C15, C25, C35, C45, C55, C56], [C16, C26, C36, C46, C56, C66]
  ]

# Elastic tensor in Cartesian form (I'm not gonna fill the gaps in
# this one, but you get the idea):
c_cart = [
  [[C1111, C1112, C1113], [], []], [[], [], []], [[], [], []]],
  [[[], [], []], [[], [], []], [[], [], []]],
  [[[], [], []], [[], [], []], [[], [], []]]
  ]

# Piezoelectric tensor in Voigt form:
e_voigt = [
  [e11, e12, e13, e14, e15, e16], [e12, e22, e23, e24, e25, e26],
  [e13, e23, e33, e34, e35, e36]
  ]

# Piezoelectric tensor in Cartesian form:
e_cart = [
  [[e111, e112, e113], [e121, e122, e123], [e131, e132, e133]],
  [[e211, e212, e213], [e221, e222, e223], [e231, e232, e233]],
  [[e311, e312, e313], [e321, e322, e323], [e331, e332, e333]]
  ]
```

If your piezoelectric tensor is in the d form rather than the e form, all the definitions are done in the same way, but you need to tell ***MattPy*** about it via the `form` keyword. Also remember that there is a factor of 2 going from d_{ijk} to $d_{i\mu}$ for some components [3]. Usually giving the tensor in Voigt form is easier on your metal health and one does not need to worry (so much) about symmetrization. For this reason most of the functions take tensors

in Voigt form as argument, although a tensor in Cartesian form can be transformed to Voigt (and *vice versa*) with the appropriate functions provided by ***MattPy***.

2.2 Available symmetries

All the point group symmetries and crystal classes have projectors associated. Note that elastic tensors belonging to the same crystal class, except for trigonal, have the same form independently of the point group. In the case of piezoelectricity the projection is zero for the centrosymmetric point groups and varies between non-centrosymmetric point groups even when they belong to the same crystal class. It is always safer to choose a point group rather than a crystal class. If a crystal class is chosen, a default point group belonging to that class will be assigned, so exercise caution when specifying the symmetries. The nomenclature is the following (the abbreviations mean exactly what you think they do):

```
# Available classes, non centrosymmetric point groups and
# centrosymmetric point groups
 classes = ["iso", "cub", "hex", "tig", "tet", "ort", "mon", "tic"]
 ncspointgroups = ["23", "432", "-43m", "6", "-6",
                   "622", "6mm", "-62m", "3", "32", "3m",
                   "4", "-4", "422", "4mm", "-42m",
                   "2", "222", "m", "-2", "mm2", "1"]
 cspointgroups = ["m-3", "m-3m", "6/m",
                  "6/mmm", "-3",
                  "-3m", "4/m", "4/mmm",
                  "2/m", "mmm", "-1"]
 pointgroups = ncspointgroups + cspointgroups

# Default point groups when only the crystal class is specified
 defaultpg = {"cub": "-43m", "hex": "6mm", "tig": "3m", "tet": "4mm",
              "ort" :"222", "mon": "2", "tric": "1"}
```

If you don't specify any symmetry at all, or your symmetry is not on the list, you'll get a warning and the target projection will default to point group $\bar{4}3m$ (zinc blende).

2.3 Elastic tensor functions

`vectorize_ela_voigt(c_voigt, verbose = True)`

Transforms an elastic tensor `c_voigt` in Voigt form to its norm-conserving vector form.

`tensorize_ela_voigt(vector_c_voigt)`

Transforms an elastic tensor `vector_c_voigt` in vector form to its Voigt tensor form. The vector is assumed to be norm preserving.

`ela_voigt_to_cartesian(c_voigt)`

Transforms an elastic tensor `c_voigt` in Voigt form to its Cartesian tensor form.

`ela_cartesian_to_voigt(c_cart)`

Transforms an elastic tensor `c_cart` in Cartesian form to its Voigt tensor form.

`rotate_ela(c_cart, rot_angles)`

Rotates an elastic tensor `c_cart` in Cartesian form by a set of angles `rot_angles = [tx, ty, tz]` given as a list with 3 elements. These elements are the rotation angles (in radians) around the x, y and z axes. The rotation is performed in this order: first rotate around x by `tx` radians, then y by `ty` radians, then z by `tz` radians.

`project_ela(vector_c_voigt, sym = None, verbose = True)`

Projects an elastic tensor in vector form onto another tensor of symmetry `sym` chosen from the list of available symmetries.

`res_ela(t, c_voigt, sym = None, verbose = False)`

Creates a residual given as the Euclidean distance between tensor `c_voigt` in Voigt form, rotated by `t = [tx, ty, yz]`, and its projection onto symmetry `sym`. This function is called by `ela_dist()` to perform an alignment optimization of the original and projected tensors.

`ela_dist(c_voigt, symlist = ["iso", "cub", "hex", "3", "32", "4", "4mm", "ort", "mon"], rotate = False, xtol = 1e-8, verbose = True, printmin = False)`

Performs a calculation of the Euclidean distance between tensor `c_voigt` in Voigt form and its projection onto each of the symmetries given in the `symlist` list. By default the tensor alignment is not optimized, use `rotate = True` to switch on rotation optimization (requires SciPy). `printmin = True` prints information from the minimization routine (e.g. number of minimization steps). `xtol` is the error tolerance below which the minimization is assumed converged and will stop. This function prints the results with nice text format but also returns a list of symmetries and distances (and angles if `rotate = True`) that might come in handy for further processing.

3 Examples

3.1 Elastic tensor: Euclidean distance calculation and symmetry projection

We are going to check Euclidean distances for different symmetries, with and without rotations, for an elastic tensor. We use as input the triclinic tensor from Ref. [4]:

```
# Tensor definition
c_voigt=[[436,161,160,12,11,25],[161,453,160,4,15,1],
         [160,160,428,13,3,8],[12,4,13,188,12,9],
         [11,15,3,12,186,9],[25,1,8,9,9,189]
        ]

# Call the routine without rotations
dist = ela_dist(c_voigt)

# And now with rotations
dist_rot = ela_dist(c_voigt, rotate = True)
```

This returns the following output:

```
************************* R E S U L T S *************************
Results without rotation optimization

Symmetry     Euclidean distance
--------     ------------------
     iso              139.65 GPa
     cub               91.05 GPa
     hex              112.08 GPa
       3              106.78 GPa
      32              109.02 GPa
       4               83.33 GPa
     4mm               89.98 GPa
     ort               89.13 GPa
     mon               76.66 GPa
************************* R E S U L T S *************************


************************* R E S U L T S *************************
Results with rotation optimization

Symmetry     Euclidean distance     Angles tx,     ty,     tz
--------     ------------------     ------------------------------
     iso              139.65 GPa           n/a     n/a     n/a  deg.
     cub               83.66 GPa         -1.89   -1.83    6.37  deg.
     hex              110.23 GPa         -2.14   -4.22     n/a  deg.
       3              102.16 GPa         -3.27   -6.94     n/a  deg.
      32              102.16 GPa         -3.27   -6.94     n/a  deg.
       4               82.32 GPa         -2.52   -1.15    2.14  deg.
     4mm               82.32 GPa         -2.52   -1.15    6.31  deg.
     ort               78.00 GPa         -3.05   -1.71    7.87  deg.
     mon               62.62 GPa         -2.85    0.76    7.62  deg.
************************* R E S U L T S *************************
```

This allows to identify the cubic symmetry as a good candidate to carry out the projections scheme. Of course, one should note that as the symmetry is further reduced the Euclidean distance also goes down. In particular, the cubic crystal class is a special case of the tetragonal, orthorhombic and monoclinic crystal classes (and obviously triclinic as well). As a consequence, the `tet` (or anything of the form `4...`, `ort` and `mon` elastic projections will always yield a lower Euclidean distance than `cub`.
If now one wants to get the elastic constant values corresponding to the, say, `cub` symmetric, the following script can be used:

```
# With no rotation
v = vectorize_ela_voigt(c_voigt)
pv = project_ela(v, sym = "cub")
pt = tensorize_ela_voigt(pv)
print "Results without rotation optimization"
print "C11 = %6.2f GPa; C12 = %6.2f GPa; C44 = %6.2f GPa \n\n" % \
(pt[0][0], pt[0][1], pt[3][3])

# With rotation (we get angles from previous optimization)
# First rotate the tensor
f = np.pi/180. # Angles need to be in radians
angles = [dist_rot[1][2]*f, dist_rot[1][3]*f, dist_rot[1][4]*f]
ct = ela_voigt_to_cartesian(c_voigt)
ct_rot = rotate_ela(ct, rot_angles = angles)
```

```
t_rot = ela_cartesian_to_voigt(ct_rot)
# Now the same as before
v_rot = vectorize_ela_voigt(t_rot)
pv_rot = project_ela(v_rot, sym = "cub")
pt_rot = tensorize_ela_voigt(pv_rot)
print "Results with rotation optimization"
print "C11 = %6.2f GPa; C12 = %6.2f GPa; C44 = %6.2f GPa" % \
(pt_rot[0][0], pt_rot[0][1], pt_rot[3][3])
```

The output is:

```
Results without rotation optimization
C11 = 439.00 GPa; C12 = 160.33 GPa; C44 = 187.67 GPa

Results with rotation optimization
C11 = 436.84 GPa; C12 = 161.42 GPa; C44 = 188.75 GPa
```

As can be seen, a current difficulty is that rotations can only be performed on

4 Citation information

Bla

5 License and copyrights

Bla

[1] M. Moakher and A. N Norris, "The closest elastic tensor of arbitrary symmetry to an elasticity tensor of lower symmetry", *J. Elasticity* **85**, 215 (2006).

[2] M. A. Caro, "Extended scheme for the projection of material tensors of arbitrary symmetry onto a higher symmetry tensor", *arXiv:1408.1219* , (2014).

[3] H. Grimmer, "The piezoelectric effect of second order in stress or strain: its form for crystals and quasicrystals of any symmetry", *Acta Crystallogr. Sect. A: Foundations of Crystallography* **63**, 441 (2007).

[4] F. Tasnádi, M. Odén, and I. A. Abrikosov, "*Ab initio* elastic tensor of cubic $Ti_{0.5}Al_{0.5}N$ alloys: Dependence of elastic constants on size and shape of the supercell model and their convergence", *Phys. Rev. B* **85**, 144112 (2012).