**Lava Wrapper Manual (1.0)**

**Khanh Dang, Jie Chen, Saryu Fensin**

**Los Alamos National Laboratory**

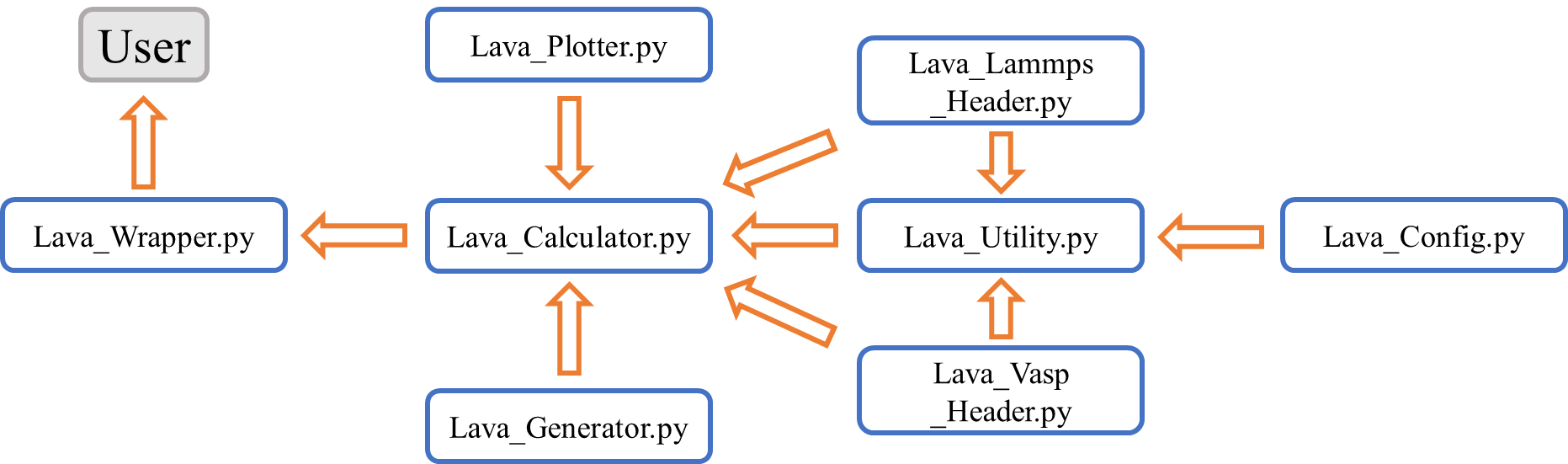
**04/28/2022**

1. **Introduction**
   1. **Overview**

The Lava Wrapper is a general-purpose calculator that provides a python interface to enable one-click calculation of the many common properties with lammps and vasp. The name Lava is derived from the “La” in Lammps and “va” in vasp. It provides a set of classes and functions to generate configurations, run lammps/vasp calculation, retrieve the output, postprocess and plot the results. All the above tasks are hard-coded into the script, without the need to call additional libraries.

* 1. **Features**

Lava Wrapper comprises of eight major python modules**.** The following flow chart gives an overall idea of the workings of Lava Wrapper. **Lava\_Wrapper.py** provides the outmost layer of abstraction, where the users can specify the set of calculations to perform, invoking the corresponding functions in **Lava\_Calculator.py,** which reads in configurations generated by **Lava\_Generator.py**, submits lammps/vasp run, extracts and postprocess the output, invoking utility functions in **Lava\_Utility.py**. In this process, two other modules, **Lava\_Lammps \_Header.py** and **Lava\_Vasp\_Header.py**, are responsible for generating Lammps input script and vasp INCAR file, respectively. Finally, the results are plotted with **Lava\_Plotter.py**. The last module, **Lava\_Config.py** is used for storing global settings.



**Fig. 1** Flow-chart of Lava Wrapper.

The following is a summary of each module’s functionality:

**Lava\_Wrapper.py**: The outmost wrapper where the users specify the type of calculations, and the parameters pertaining to these calculations.

**Lava\_Calculator.py**: The core of Lava Wrapper, responsible for performing the various types of calculations through the following steps: generate the relevant input file, perform calculations, retrieve and postprocess the output, and plot the results.

**Lava\_Generator.py**: Functions for the generation of various crystal structures and defect configurations. The following 8 bulk crystal structures are currently supported: Simple Cubic (SC), Body Centered Cubic (BCC), Face Centered Cubic (FCC), Hexagonal Close Packed (HCP), Diamond Cubic (DC), double HCP, A5\_Beta\_Sn, A15\_Beta\_W. The following defect configurations: vacancy, interstitial, surface, stacking faults, twin faults can also be added to the generated crystal structures.

**Lava\_Utility.py**: Utility functions for file management, data processing, etc.

**Lava\_Lammps\_Header.py**: Functions for generating lammps input script, depending on the type of calculation.

**Lava\_Vasp\_Header.py**: Functions for generating vasp input script, namely, the INCAR file, depending on the type of calculation.

**Lava\_Plotter.py**: Functions for plotting the results.

**Lava\_Config.py**: Specification the lammps executable, the lammps potential file, as well as the element, provided by the user. The first two files are only required for running the wrapper in lammps mode.

* 1. **Types of Calculations**

Currently, Lava Wrapper incorporates the following types of calculations. For some calculations the output a written to “Summary.dat” file that serves as the general output for Lava\_Wrapper, and for other types of calculation where the results are tabulable, then it is written to a separate data file and plotted as well.

|  |  |  |
| --- | --- | --- |
| **Type of calculation** | **Summary**  **.dat?** | **Separate data file + Plot?** |
| Lattice constant and cohesive energy | ✓ | ✓ |
| Cold curve (E\_vs.\_Volume/Density) |  | ✓ |
| Elastic properties (elastic constant, bulk modulus, etc.) | ✓ |  |
| Volume-conserving/non-conserving Bain path (1D and 2D) |  | ✓ |
| Liquid radial distribution function with varying T |  | ✓ |
| Vacancy, interstitial formation energy | ✓ |  |
| Surface energy | ✓ |  |
| Thermal expansion (lattice constant vs temperature) |  | ✓ |
| (Generalized) stacking fault energy (1D and 2D) |  | ✓ |
| Melting point (2-phase method) | ✓ |  |

****

1. **Examples**
   1. **Running the wrapper**

To run Lava Wrapper, load python3 first and then issue the following command on the terminal:

python3 Lava\_Wrapper.py **mode**

wherein the highlighted **mode** can either be **Lammps** or **Vasp**, depending on which mode you wish to run.

Example scripts and output of Lava Wrapper are given below, when invoked for Al using Mishin-Farkas potential [1]. This can be set up by specifying the lammps executable as well as potential file name in **Lava\_Config.py** in the following way:

lammps\_executable = 'lmp\_mpi’

element = "Al"

potential\_file = 'Al99.eam.alloy'

Make sure that the lammps executable and the potential file is in the same folder as the Lava Wrapper scripts.

* 1. **Lattice constants and cohesive energy**

|  |
| --- |
| def get\_cohesive\_energy(phase\_list, NSW=100, IBRION=1, ISIF=3, lat\_rep=[2,2,2], orz=[0,0,1], mode=None): |

The following snippet in **Lava\_Wrapper.py** calculates the lattice constants and cohesive energy of a given list of crystal structures specified in ‘phase\_list’, the 11 crystal structures shown below are currently supported. In addition, the user can provide additional lammps data file named “user\_provided.data” as an additional phase.

|  |
| --- |
| ### Lattice constants and cohesive energy  phase\_list = ["SC", "BCC", "FCC", "HCP", "DC", "DHCP", "A5\_Beta\_Sn", "A5\_Gamma\_Sn", "A15\_Beta\_W","L12","9R"]  Lava\_Calculator.get\_cohesive\_energy(phase\_list) |

The output is written out to **Lammps\_latt.dat**, as shown below. Where phase, alat, blat, clat, atoms\_per\_cell, Ecoh represent the crystal structure, lattice constant a, b/a ratio, c/a ratio, number of atoms per unit cell, and the cohesive energy. This fileis read in to perform other types of calculations which require information regarding lattice constants and cohesive energy.

|  |
| --- |
| phase a b/a c/a atoms\_per\_cell Ecoh  SC 2.7407 1.0000 1.0000 1 -2.9614  BCC 3.2354 1.0000 1.0000 2 -3.2545  FCC 4.0500 1.0000 1.0000 4 -3.3600  HCP 2.8191 1.7321 1.7539 4 -3.3322  DC 6.0465 1.0000 1.0000 8 -2.4691  DHCP 2.8458 1.7321 3.3503 8 -3.3439  A5\_Beta\_Sn 5.3506 1.0000 0.5191 4 -3.0359  A15\_Beta\_W 5.1467 1.0000 1.0000 8 -3.3469 |

All parameters and their specification:

|  |  |
| --- | --- |
| **Parameters** | **Specification** |
| phase\_list | The list of phases to evaluate cold curve on. |
| NSW (vasp) | Maximum number of ionic steps, default = 100 |
| IBRION (vasp) | Tag for selcting minimization algorithm, default = 1 |
| ISIF (vasp) | Tag for principal DOFs allowed to change in relaxation, default = 3 |
| lat\_rep | Unit cell lattice replication along X, Y and Z direction, default = [2,2,2] |
| orz | Z orientation, default = [0,0,1] |
| mode | The mode (Lammps or Vasp) specified by the user when invoking Lava Wrapper |

* 1. **Cold curve**

|  |
| --- |
| def get\_cold\_curve\_V(phase\_list, V\_start=None, V\_stop=None, npoints=None, NSW=1, IBRION=1, ISIF=2, lat\_rep=[1,1,1], orz=[0,0,1], V\_list=None, mass=None, mode=None, dirname=None, output\_name=None, plot\_axis\_x="volume", plot\_axis\_y="energy"):  def get\_cold\_curve\_R(phase\_list, alat\_expanse=None, npoints=None, NSW=1, IBRION=1, ISIF=2, lat\_rep=[1,1,1], orz=[0,0,1], mass=None, mode=None, plot\_axis\_x="volume", plot\_axis\_y="energy"): |

The following snippet in **Lava\_Wrapper.py** calculates the cold curve of a given list of crystal structures specified with ***phase\_list***. Cold curve is calculated based on volume per atom V (E\_vs.\_V). The grid points range from ***V\_start*** to ***V\_stop***, with ***npoints*** evenly spaced points.

|  |
| --- |
| ### Cold curve based on V  phase\_list = ["SC", "BCC", "FCC", "HCP", "DC"]  Lava\_Calculator.get\_cold\_curve\_V(phase\_list, V\_start=10, V\_stop=30, npoints=30, mass=4.48, mode= mode) |

All parameters and their specification:

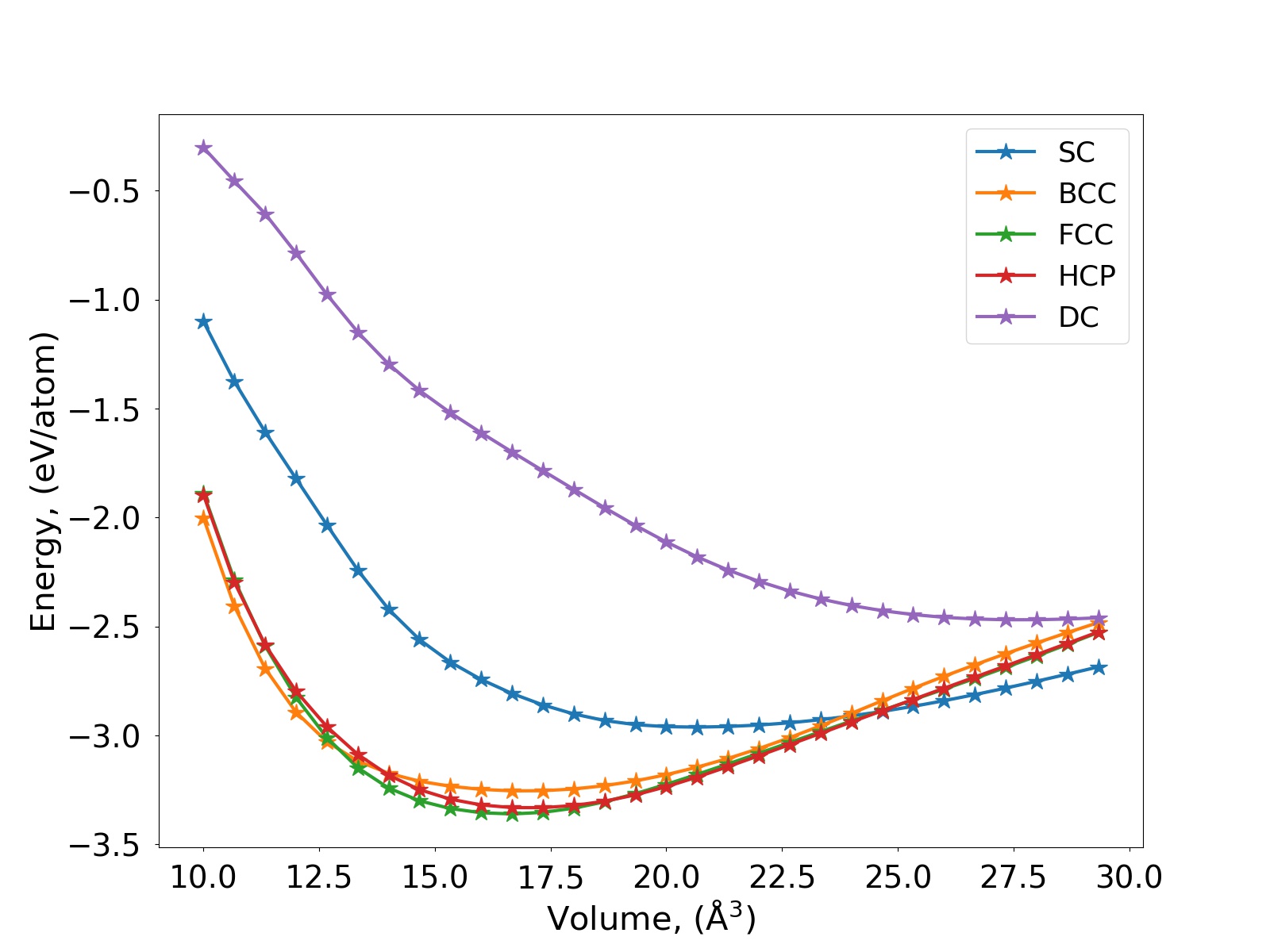
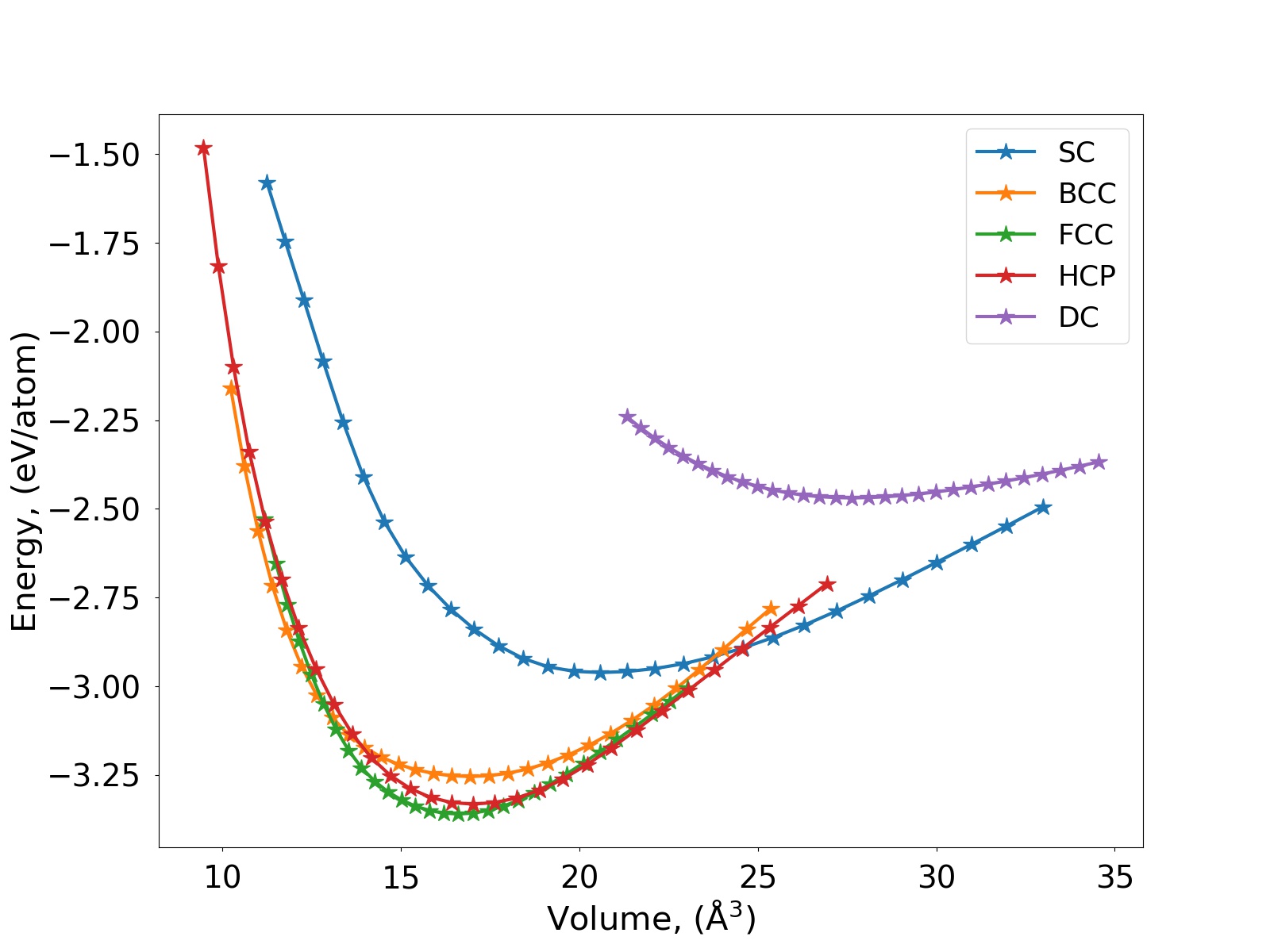
|  |  |
| --- | --- |
| **Parameters** | **Specification** |
| phase\_list | The list of phases to evaluate cold curve on |
| V\_start | Start in the range of volume to be evalauted (inclusive), default = None |
| V\_stop | End in the range of volume to be evalauted (inclusive), default = None |
| npoints | Number of points in the range specified by V\_start and V\_stop |
| V\_list | An explicit list of volume to be evaluated |
| mass | Atom mass (in unit of 10-23 g) for the purpose of calculating density |
| dirname | Path to run the calculations, individually set if not specified |
| output\_name | Name of the output data and jpeg file, individually set if not specified |
| plot\_axis\_x | X axis for the plot, "density" or "volume" or "energy" |
| plot\_axis\_y | Y axis for the plot, "density" or "volume" or "energy" |

Cold curve can also be evaluated based on lattice constant R (E\_vs.\_R). In this case, the grid points range from ***alat*** - ***alat\_expanse*** to ***alat*** + ***alat\_expanse***, where ***alat*** is the lattice constants of the corresponding phase, with ***npoints*** evenly spaced points. As compared to E\_vs.\_V, the advantage of this scheme that the potential valley will always be captured.

The following snippet in **Lava\_Wrapper.py** calculates the cold curve based on R:

|  |
| --- |
| ### Cold curve based on R  ## Expanse of lattice (from alat-alat\_expanse to alat+alat\_expanse)  phase\_list = ["SC", "BCC", "FCC", "HCP", "DC"]  Lava\_Calculator.get\_cold\_curve\_R(phase\_list, alat\_expanse=0.5, npoints=30, mass=4.48, mode=mode) |

The tabulated E vs. V and E vs. R output is written to **EOS\_V\_Lammps.dat** and **EOS\_R\_Lammps.dat**, respectively. The corresponding plots: **EOS\_V\_Lammps.jpeg** and **EOS\_R\_Lammps.jpeg** are shown below in Fig. 2.

(a) (b)

**Fig. 2** Cold curve generated by Lava Wrapper with: (a) E vs. V scheme, (b) E vs. R scheme.

* 1. **Elastic constants**

|  |
| --- |
| def get\_elastic\_properties(phase\_list, NSW=100, IBRION=6, ISIF=3, NFREE=2, lat\_rep=[2,2,2], orz=[0,0,1], mode=None, use\_built\_in\_method=False): |

The following snippet in **Lava\_Wrapper.py** calculates the elastic constants of a given list of crystal structures specified with ***phase\_list***. The elastic constants are evaluated by finite displacement method, with a displacement of ±10-4. The parameter ***use\_built\_in*** can be set to True or False. When ***use\_built\_in*** = True, the Lava Wrapper generates all the deformed configurations and calculates the elastic constants, whereas when ***use\_built\_in*** = False, fix/deform is used in lammps to apply deformation, and a combination of IBRION, ISIF, NFREE tag (see default values in function definition) is specified in vasp allowing for calling the built-in subroutine to evaluate elastic constants. Besides elastic constants, other elastic properties such as bulk modulus, shear modulus and Poisson’s ratio are also calculated.

|  |
| --- |
| ### Elastic properties  phase\_list = ["FCC", "HCP"]  Lava\_Calculator.get\_elastic\_properties(phase\_list, mode=mode, use\_built\_in=False) |

The output in **Summary.dat** is shown below:

|  |
| --- |
| FCC:  C11 = 113.792 GPa  C12 = 61.550 GPa  C44 = 31.595 GPa  Bulk modulus = 78.964 GPa  Shear modulus = 26.121 GPa  Poisson Ratio = 0.351  HCP:  C11 = 116.117 GPa  C12 = 55.404 GPa  C13 = 45.209 GPa  C33 = 69.833 GPa  C44 = 11.185 GPa  Bulk modulus = 75.641 GPa  Shear modulus = 30.357 GPa  Poisson Ratio = 0.323 |

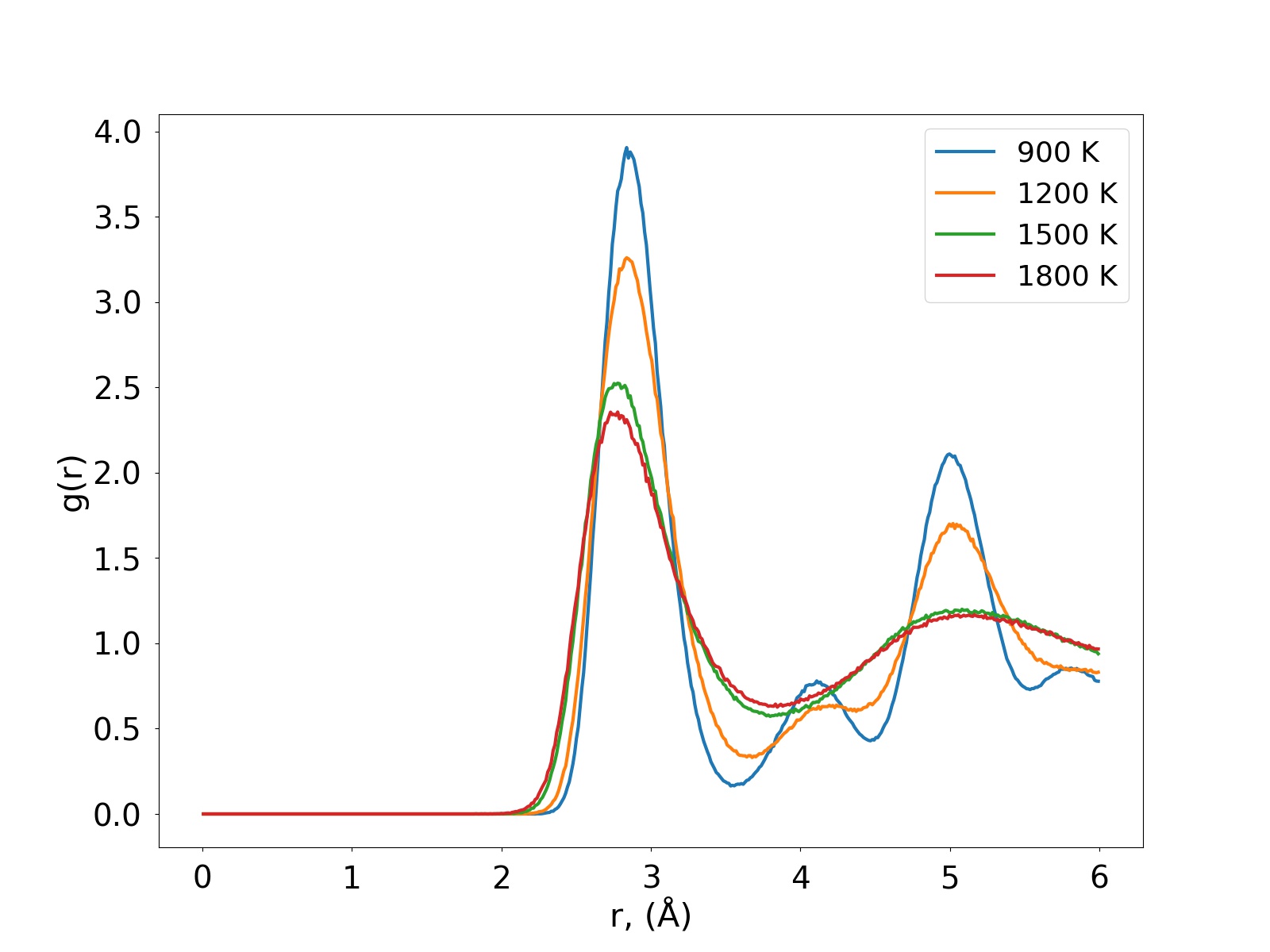
* 1. **Radial distribution function (RDF): Lammps only**

|  |
| --- |
| def get\_liquid\_rdf(T\_dict, lat\_rep=[40,40,40], orz=[0,0,1], mode=None): |

The following snippet in **Lava\_Wrapper.py** calculates the RDF specified with ***T\_dict***, which is a dictionary with the keys corresponding to a phase and values corresponding to a list of temperature for that phase. This calculation is performed in Lammps mode only.

|  |
| --- |
| ### Radial distribution function for liquid at different T (Lammps only)  T\_dict = {"FCC": range(300,1500,300)}  Lava\_Calculator.get\_liquid\_rdf(T\_dict, mode=mode) |

The output is written to **Liquid\_RDF\_FCC\_Lammps.dat**.The corresponding plot **Liquid \_RDF\_FCC\_Lammps.jpeg** is shown below in Fig. 3.



**Fig. 3** RDF plot generated by Lava Wrapper.

* 1. **Vacancy, interstitial formation energy**

|  |
| --- |
| def get\_vacancy\_interstitial\_energy(phase\_list, NSW=100, IBRION=1, ISIF=2, lat\_rep=None, orz=[0,0,1], lat\_dim=[12,12,12], mode=None): |

The following snippet in **Lava\_Wrapper.py** calculates the vacancy and interstitial formation energy of given list of crystal structures specified with ***phase\_list*** at 0 K. A ***lat\_rep*** can be specified to replicate the unit cell, or ***lat\_dim*** can be specified such that the unit cell is replicated to ***lat\_dim***.

|  |
| --- |
| ### Vacancy/interstitial formation energy  phase\_list = ["FCC"]  Lava\_Calculator.get\_vacancy\_interstitial\_energy(phase\_list, lat\_dim=[12,12,12], mode=mode) |

The output in **Summary.dat** is shown below:

|  |
| --- |
| FCC:  Vacancy formation energy: 0.681 eV  Interstitial formation energy: 3.096 eV |

* 1. **Surface energy**

|  |
| --- |
| def get\_surface\_energy(surface\_dict, NSW=100, IBRION=1, ISIF=2, lat\_rep=None, lat\_dim=[3,3,20], vac=10, mode=None): |

The following snippet in **Lava\_Wrapper.py** calculates surface energy at 0 K specified with ***surface\_list***, which is a dictionary with the keys corresponding to a phase and values corresponding to a list of surfaces for that phase. Specifications of ***lat\_rep*** and ***lat\_dim*** is similar to that in section 3.6.

|  |
| --- |
| ### Surface energy  surface\_dict = {"FCC": [(0,0,1), (1,1,0), (1,1,1)],  "HCP": [(0,0,0,1), (1,1,-2,0), (1,0,-1,0)]}  Lava\_Calculator.get\_surface\_energy(surface\_dict, mode=mode) |

The output in **Summary.dat** is shown below:

|  |
| --- |
| (001) Surface energy: 943.580 mJ/m2  (110) Surface energy: 1006.016 mJ/m2  (111) Surface energy: 870.511 mJ/m2  (0001) Surface energy: 821.416 mJ/m2  (11-20) Surface energy: 982.039 mJ/m2  (10-10) Surface energy: 1142.438 mJ/m2 |

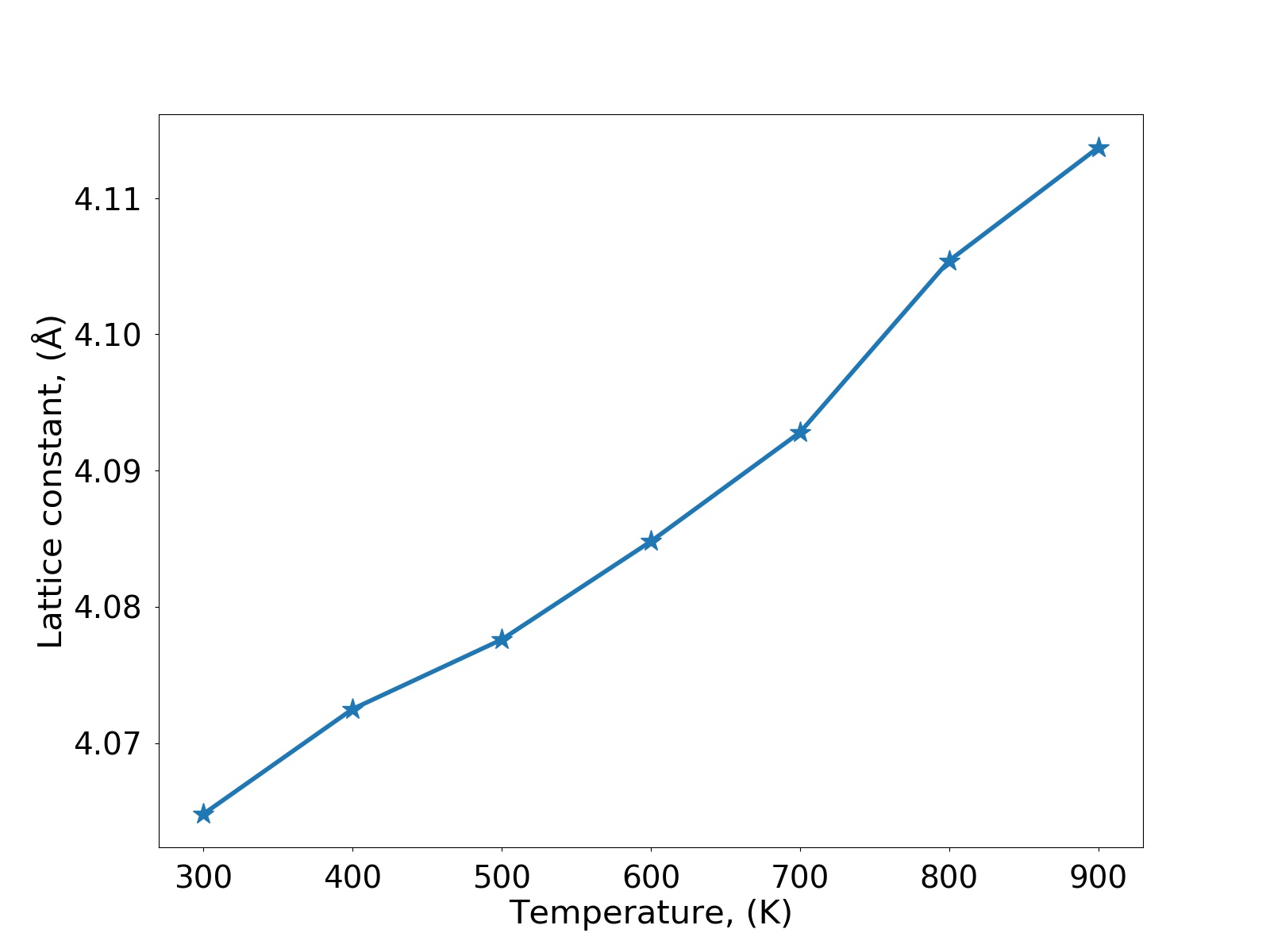
* 1. **Thermal expansion: Lammps only**

|  |
| --- |
| def get\_thermal\_expansion(T\_dict, lat\_rep=[10,10,10], orz=[0,0,1], mode=None): |

The following snippet in **Lava\_Wrapper.py** calculates temperature-dependent lattice constants. ***T\_dict*** is a dictionary with the keys corresponding to a phase and values corresponding to a list of surfaces for that phase. This calculation is performed in Lammps mode only.

|  |
| --- |
| ### Thermal expansion (Lammps only)  T\_dict = {"FCC": range(400,1000,200)}  Lava\_Calculator.get\_thermal\_expansion(T\_dict, mode=mode) |

The output is written to **Thermal\_expansion\_FCC\_Lammps.dat**.The corresponding plot **Thermal\_expansion \_FCC\_Lammps.jpeg** is shown below in Fig. 4.



**Fig. 4**  Thermal expansion plot generated by Lava Wrapper.

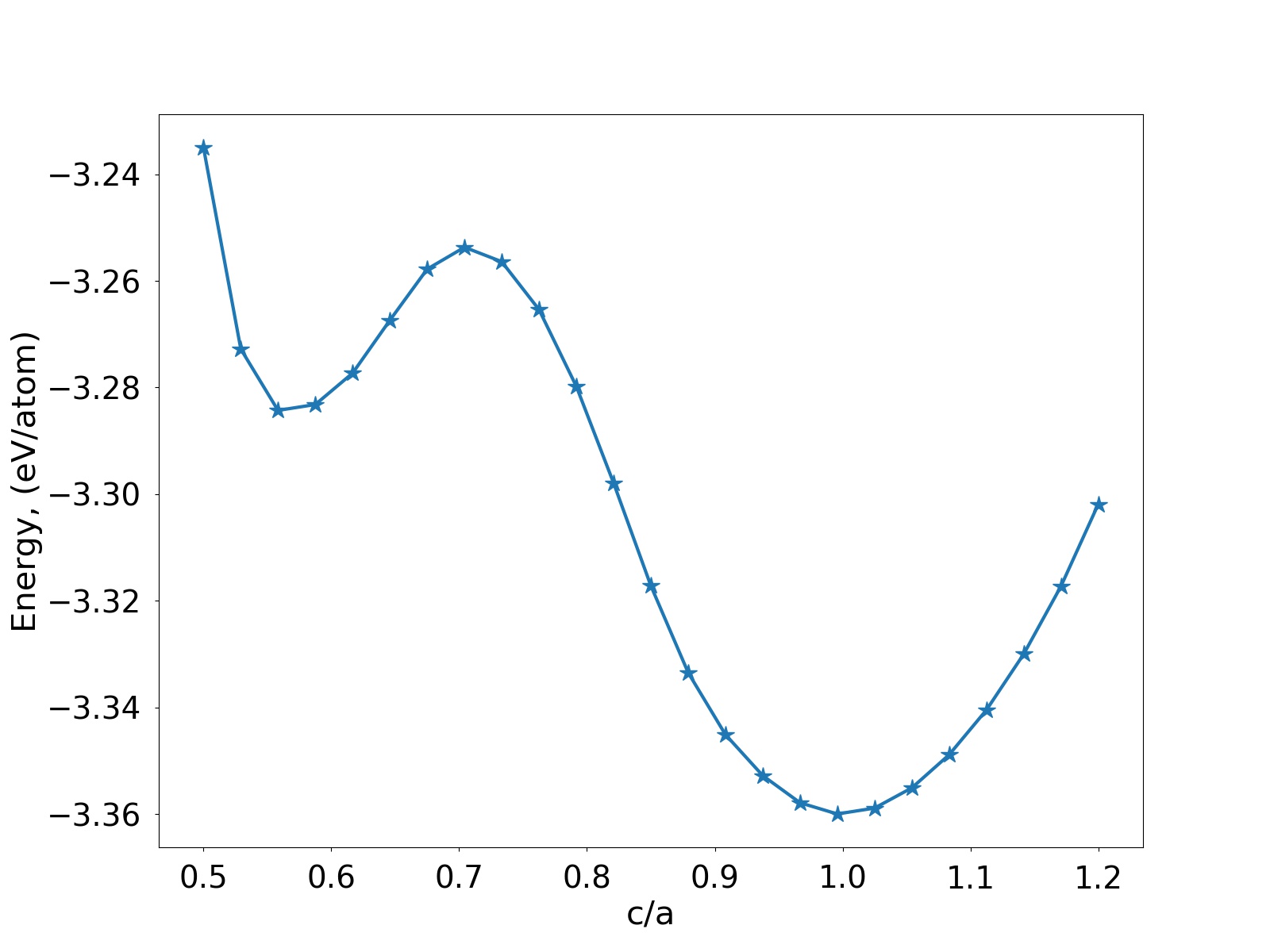
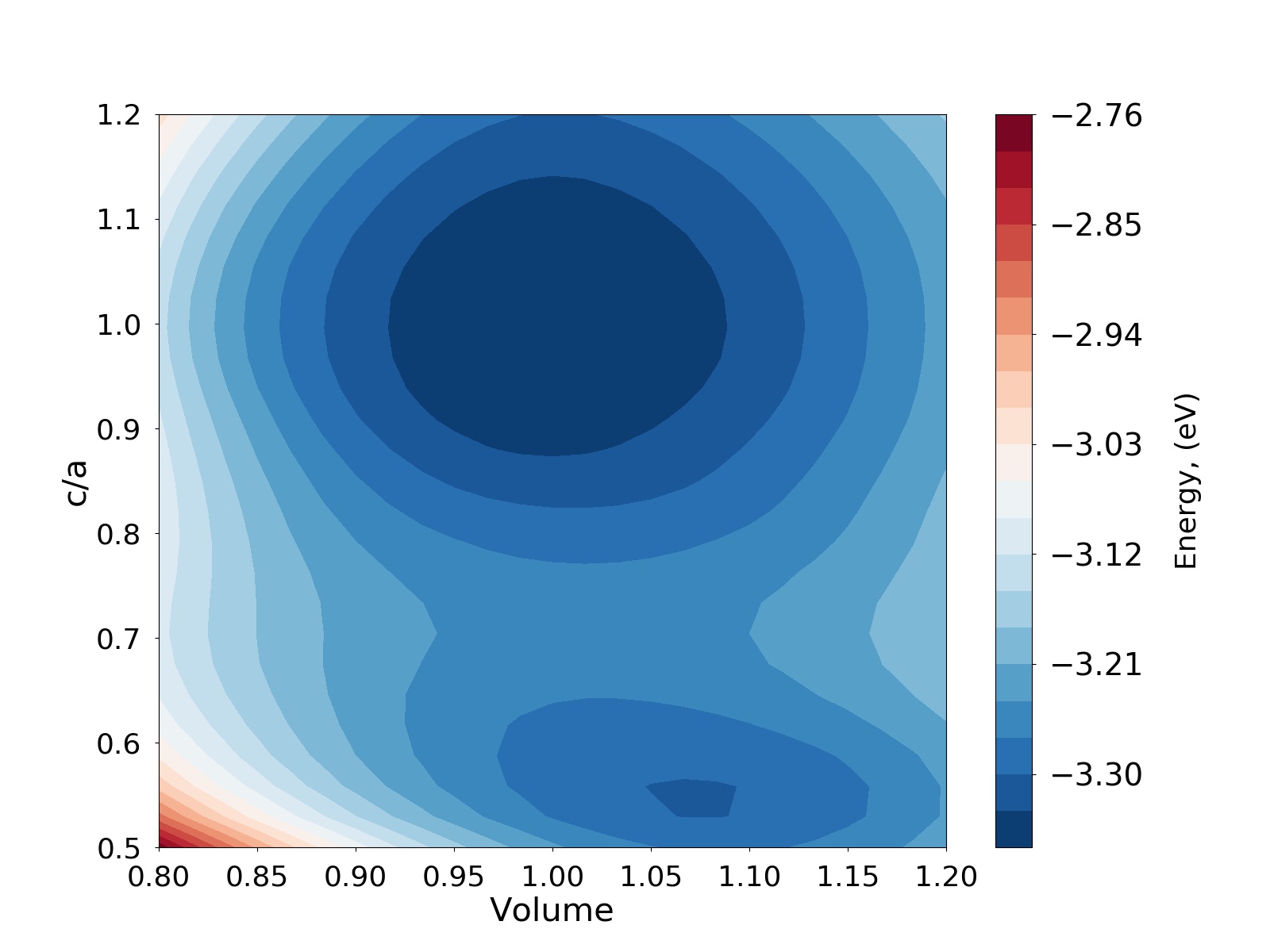
* 1. **Bain path (1D and 2D)**

|  |
| --- |
| def get\_Bain\_path(clat\_start, clat\_end, npoints, NSW=1, IBRION=1, ISIF=2, lat\_rep=[1,1,1], orz=[0,0,1], mode=None):  def get\_Bain\_path\_2D(clat\_start, clat\_end, npoints\_c, Vol\_start, Vol\_end, npoints\_V, NSW=1, IBRION =1, ISIF=2, lat\_rep=[1,1,1], orz=[0,0,1], mode=None): |

The following snippet in **Lava\_Wrapper.py** calculates volume-conserving 1-D Bain path and volume-non-conserving 2-D Bain path, where the energy of a FCC lattice is calculated for varying c/a ratio at constant volume (volume-conserving 1-D Bain path), or at varying c/a ratio and varying volume (volume-non-conserving 2-D Bain path).

|  |
| --- |
| ### Volume-conserving Bain path  npoints = 30  Lava\_Calculator.get\_Bain\_path(0.5, 1.2, npoints, mode=mode)    ### Volume-conserving Bain path (2D)  npoints = 15  Lava\_Calculator.get\_Bain\_path\_2D(0.5, 1.2, npoints, 0.8, 1.2, npoints, mode=mode) |

The output is written to **Bain\_Lammps.dat**, and **2D\_Bain\_Lammps.dat**.The corresponding plots: **Bain\_Lammps.jpeg** and **2D\_Bain\_Lammps.jpeg** are shown below in Fig. 5.

(a) (b)

**Fig. 5** (a) 1-D volume-conserving Bain path plot, (b) 2-D Bain path plot generated by Lava Wrapper.

* 1. **Stacking fault energy (1D and 2D)**

|  |
| --- |
| def get\_stacking\_fault(phase, sf\_mesh, NSW=100, IBRION=1, ISIF=2, mode=None): |

The following snippet in **Lava\_Wrapper.py** calculates to get the energetics of the slip vs twin path for FCC phase. Currently, four Gamma surfaces system are supported:

FCC -> X: [11-2], Y: [1-10], Z:[111]

BCC -> X: [111], Y: [-110], Z: [11-2]

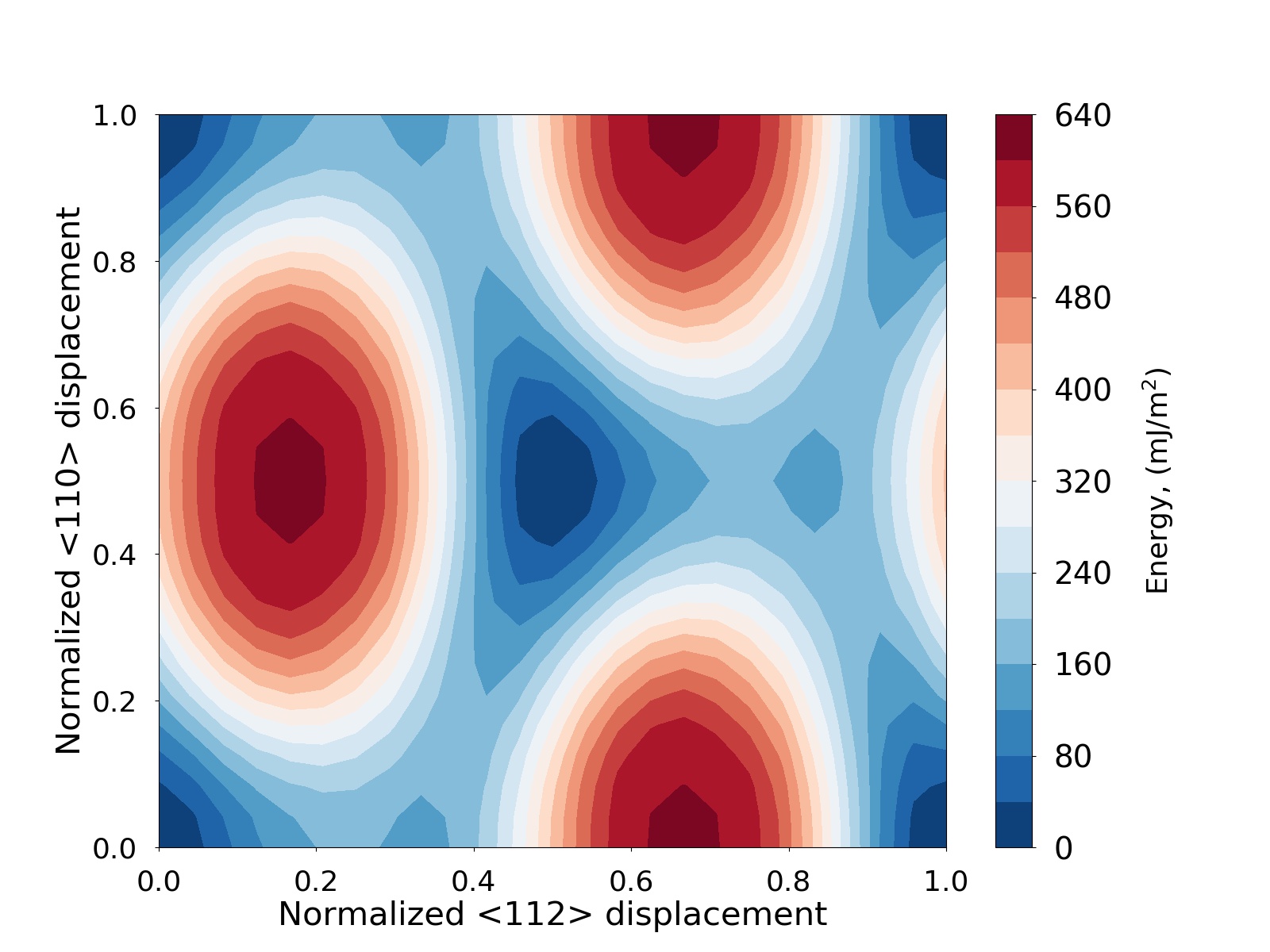
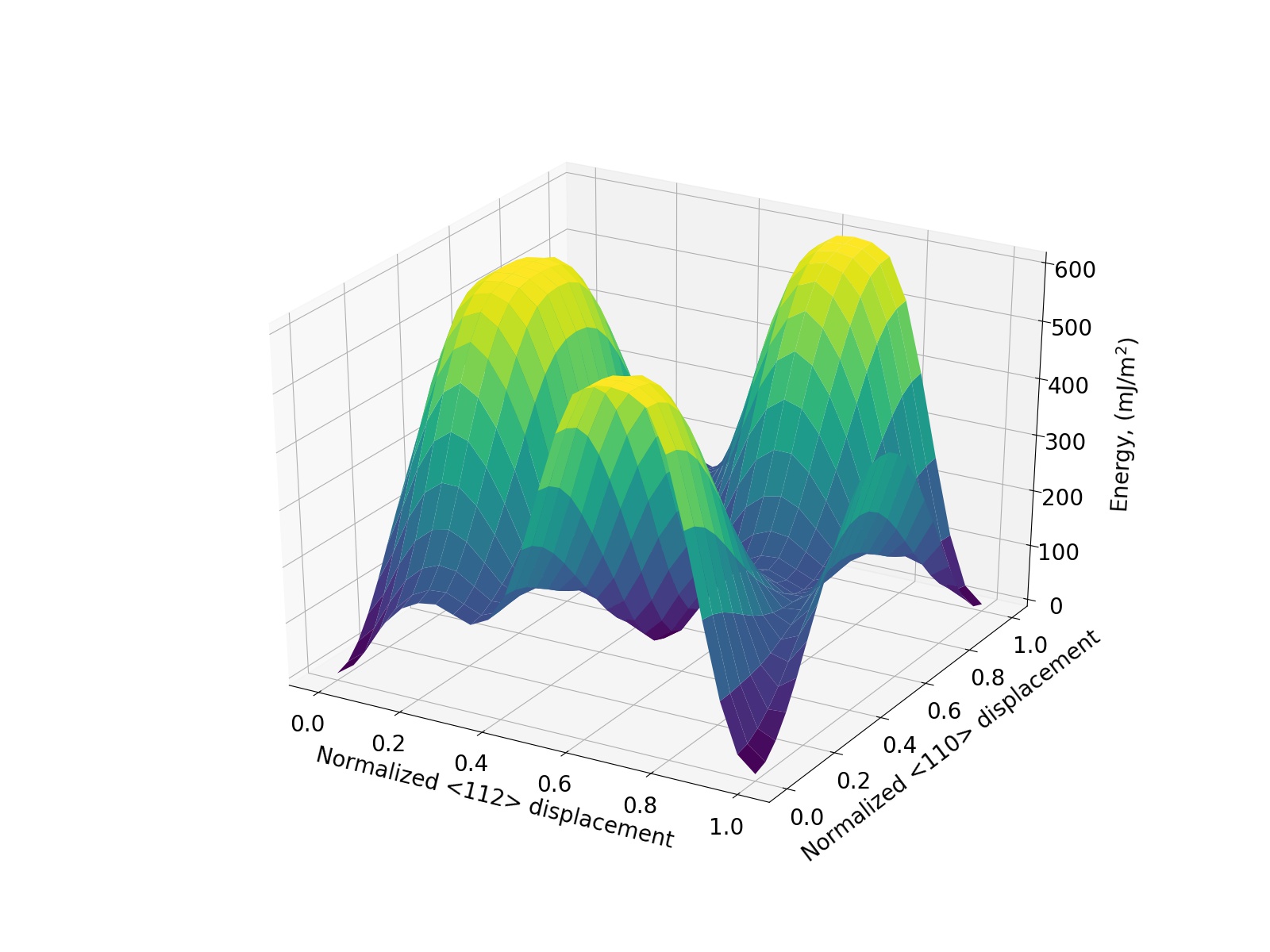
HCP -> X: [11-20], Y: [10-10], Z: [0001].

A5\_Beta\_Sn -> X: [001], Y: [010], Z: [100]

The corresponding slip systems are selected when ***phase*** is specified. Here, ***sf\_mesh*** specifies the 2-D mesh used to sample theGamma surface.

|  |
| --- |
| ### Stacking fault energy: 2-D grid, very expensive for VASP  phase = "FCC"  sf\_mesh = [18,18]  Lava\_Calculator.get\_stacking\_fault(phase, sf\_mesh, mode=mode) |

The output is written to **2D\_Gamma\_Lammps.dat**, and **3D\_Gamma\_Lammps.dat**.The corresponding plots: **2D\_Gamma \_Lammps.jpeg** and **3D\_Gamma\_Lammps.jpeg** are shown below in Fig. 6.

(a)  (b) 

**Fig. 6** (a) 2-D Gamma surface, (b) 3-D Gamma surface plot generated by Lava Wrapper.

Moreover, a generalized stacking fault energy curve (GSFE) can also be calculated.

|  |
| --- |
| def plot\_GSFE\_v2(blat,clat,phase,direction,input\_X, input\_Y, input\_Z, output\_file): |

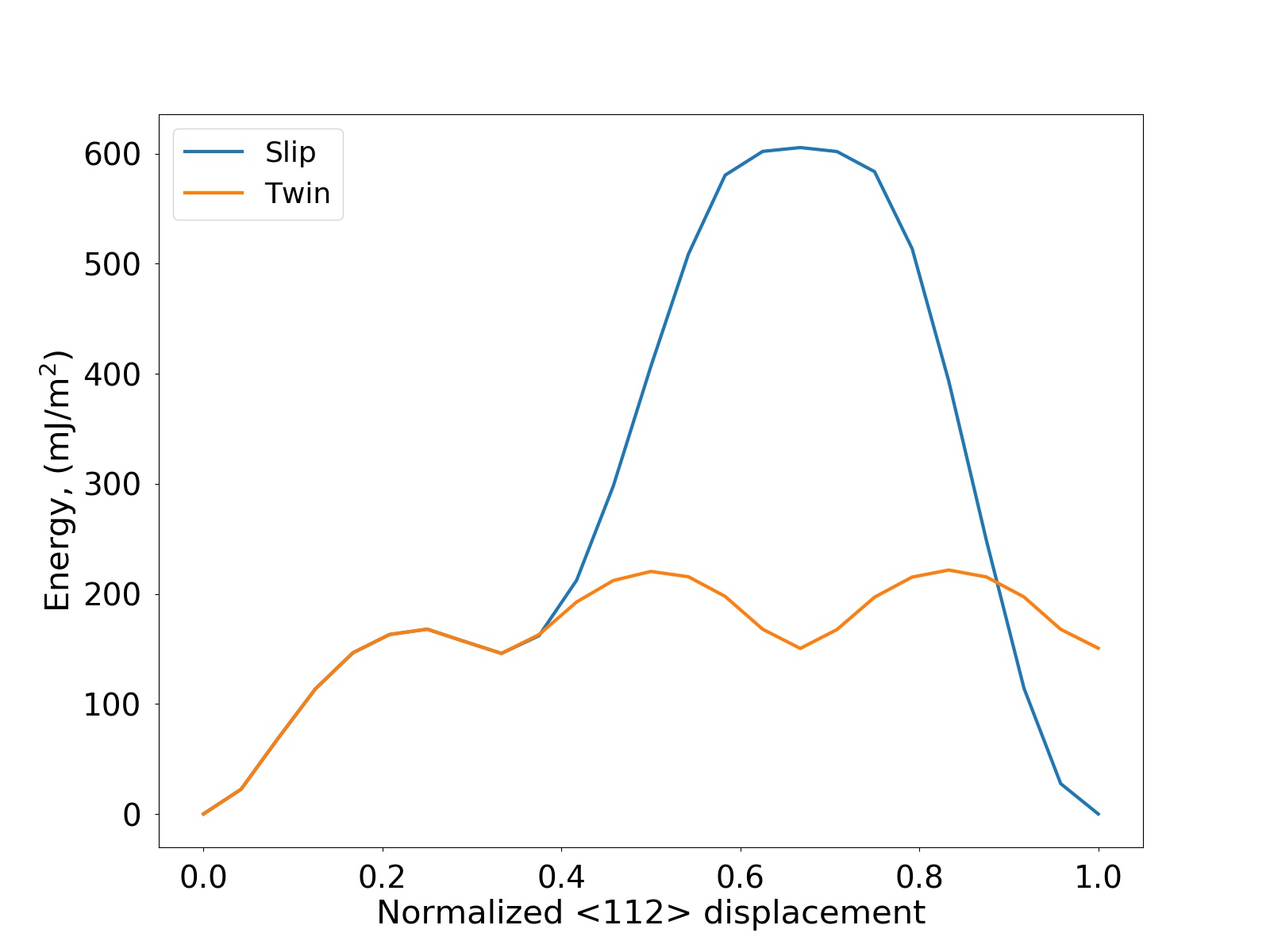
From the 2-D, 3-D gamma map, the GSFE along any direction within the surface can be obtained. This is available for all 4 gamma surfaces.

|  |
| --- |
| def get\_general\_stacking\_fault(phase, sf\_mesh, NSW=100, IBRION=1, ISIF=2, mode=None): |

For FCC and BCC structures, a full slip path, as well as a twinning path achieved through successive slip of neighboring layers along [11-2] (FCC) and [111] (BCC) direction is calculated, with the following snippet in **Lava\_Wrapper.py**. ***sf\_mesh*** is the size of the mesh, wherein ***sf\_mesh[0]*** needs to a multiplier of 3, and ***sf\_mesh[1]*** needs to be 1. GSFE can be calculated for FCC or BCC phase.

|  |
| --- |
| ### General stacking fault curve (full slip vs twin)  phase = "FCC"  sf\_mesh = [24,1]  Lava\_Calculator.get\_general\_stacking\_fault(phase, sf\_mesh, mode=mode) |

The output is written to **GSFE\_Lammps.dat**, with the corresponding plot: **GSFE\_ Lammps.jpeg** as shown below in Fig. 7.



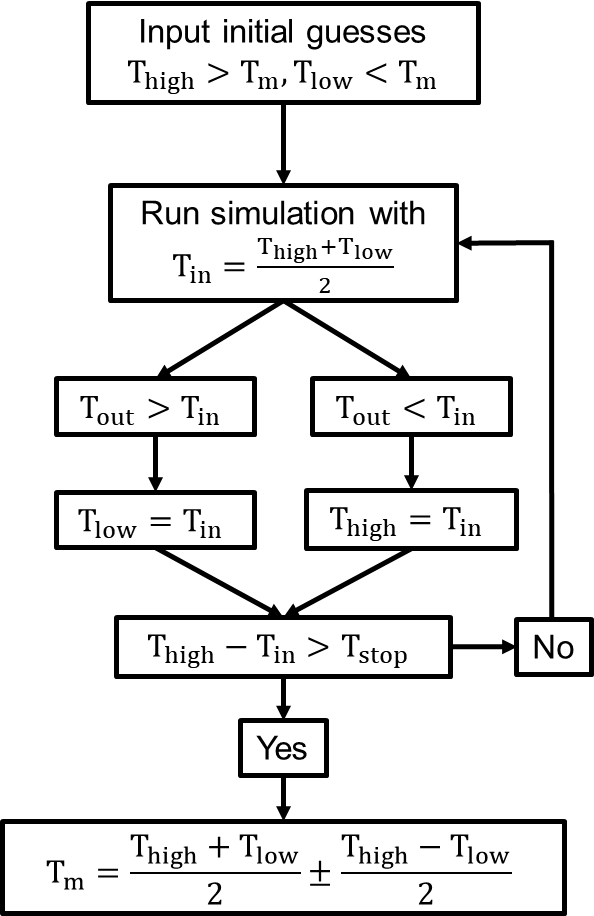
**Fig. 7** GSFE plot generated by Lava Wrapper.

* 1. **Melting point: Lammps only**

The following snippet in **Lava\_Wrapper.py** determines the melting point for each phase in the ***phase\_list*** corresponding to a phase with initial guess between [***Tleft,Tright***] and desired ***T\_error***. Specifications of ***lat\_rep*** is similar to that in section 3.6.

|  |
| --- |
| def get\_melting\_point(phase\_list, Tleft, Tright T\_error, lat\_rep=[10,10,50], orz=[0,0,1], mode=None): |

The idea is to first construct a solid-liquid interface at the center of the simulation cell along the Z direction at the estimated melting point ***T\_est=(T\_left+T\_right)/2***, and equilibrate the system via an isenthalpic Nosé-Hoover style barostat (NPH). By default, LAVA creates this with a fully periodic simulation cell measuring 10x10x50 unit cells with the Z direction aligned along the 50 unit cell axis. On one side of the halfway mark, the cell is initially heated to the input temperature, ***T\_est***, and the other half is superheated to ***1.2T\_est+300*** to form a liquid structure. Both domains then cooled to ***T\_est***. Under isenthalpic conditions, ***T\_est>T\_m*** means the solid is unstable and will melt, lowering the overall temperature as sensible heat becomes latent heat. Conversely, ***T\_est<T\_m*** will cause the liquid to solidify, raising the temperature as latent heat is released. In either scenario, the isenthalpic Nosé-Hoover style barostat brings the overall temperature closer to Tm. Therefore, the sign of the difference between the input temperature and output temperature will be positive below ***T\_m*** and negative above ***T\_m***. LAVA leverages this by using the bisection method to converge to ***T\_m***. The simulation is terminated when the difference between the lower and upper bounds is less than ***T\_error***. The final melting point is given as ***T\_m=T±T\_error/2***.



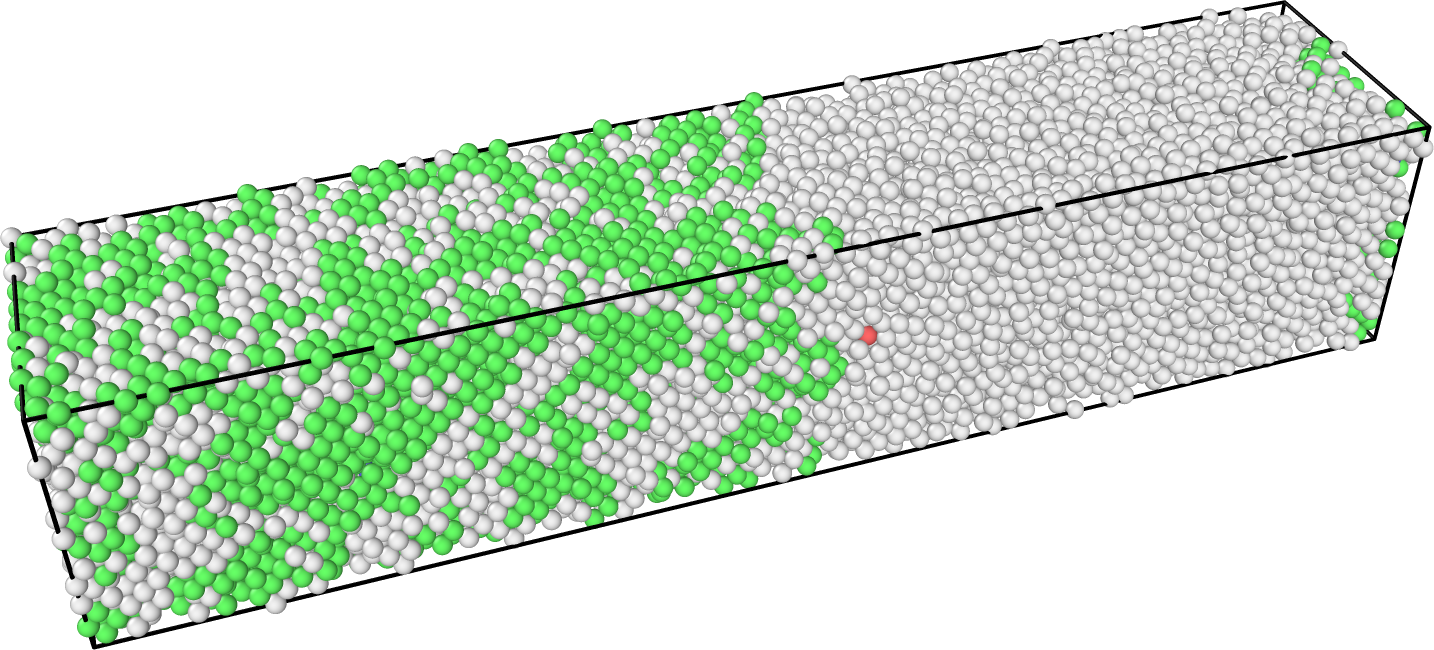
**Fig. 7** Bisection algorithm to determine the melting temperature using LAMMPS for LAVA.

|  |
| --- |
| ### Melting point using 2 phase method (Lammps only)  ## get\_melting\_point(T, T\_error): T is trial melting point, T\_error is convergence criteria  phase\_list = ["FCC"]  Lava\_Calculator.get\_melting\_point(phase\_list, 400,1000, 10, mode=mode) |

Thus, if we start from a trial melting point between (***T\_left***, ***T\_right)*** of (400,1000) K, and set ***T\_error*** to be 10 K, then we obtain the following output in **Summary.dat**. Five runs are required to achieve convergence to the melting point of 878.1 ± 10.0 K obtained by the Lava Wrapper. The bisection provides a consistent speed to convergence compared to other searching algorithm. The number of iterations can be predetermined based on the initial range and the convergence tolerance ***T\_error***.

|  |
| --- |
| Left T: 700.0 K, Right T: 1000.0 K, delta\_T: 150.0 K  Left T: 850.0 K, Right T: 1000.0 K, delta\_T: 75.0 K  Left T: 850.0 K, Right T: 925.0 K, delta\_T: 37.5 K  Left T: 850.0 K, Right T: 887.5 K, delta\_T: 18.8 K  Left T: 868.8 K, Right T: 887.5 K, delta\_T: 9.4 K  delta\_T = 9.4 < 10.0 (T\_error), convergence achieved.  Melting point: 878.1 +- 10.0 K |

The final melt in the last run is shown below in Fig. 8, where a well-defined solid-liquid interface at the center along Z direction can be seen. The dimension of the system, with a replication of 10 x 10 x 50, is ~ 4 x 4 x 20 nm, with 20k atoms.



**Fig. 8** The final melt at 1039.2 K obtained with 2-phase method in Lava Wrapper. Atoms are colored with Common Neighbor Analysis (CNA).

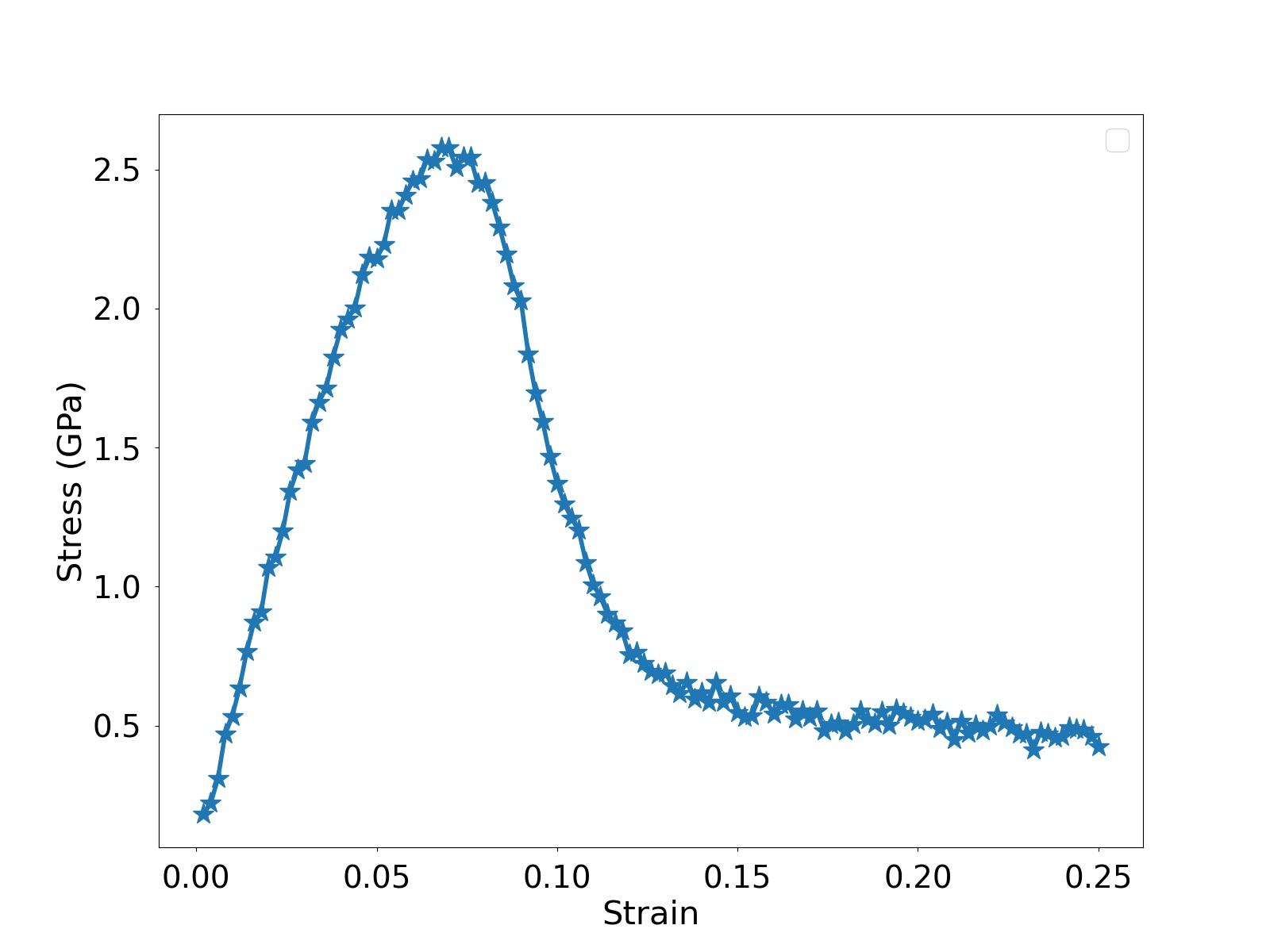
* 1. **Uniaxial deformation: Lammps only**

|  |
| --- |
| def uniaxial\_deform(phase\_list, temp, deform\_mode , rate, final\_strain,lat\_rep, orz=[0,0,1], mode=None): |

The following snippet in **Lava\_Wrapper.py** applies uniaxial deformation along desired directions (specified in the ***deform\_mode***) at a given temperature (***temp***), strain rate (***rate***), and ***final\_strain***. The values for the deform\_mode are defined such as: 1🡪x, 2🡪y, 3🡪z, 4🡪yz, 5🡪xz, 6🡪 yz directions. The uniaxial deformation along desired direction is performed via the fix/deform command.

|  |
| --- |
| # Uniaxial deformation (Lammps\_serial only)  # Perform uniaxial deformation along desired direction with erate/trate via a fix/deform command  # deform\_mode indicates which deformation mode is performed  # 1=x 2=y,3=z, 4=yz, 5=xz, 6=xy  phase\_list = ["A5\_Beta\_Sn","A5\_Gamma\_Sn"]  temp = 300  lat\_rep=[15,15,15]  deform\_mode = [1,2,3,4,5,6]  rate = 0.01  final\_strain = 0.25  Lava\_Calculator.uniaxial\_deform(phase\_list,temp,deform\_mode,rate, final\_strain, lat\_rep,mode=mode) |

The output is written to **Stress\_strain\_\*deform\_mode\*\_\*phase\*.txt**.The corresponding plots: **Stress\_strain\_1\_A5\_Beta\_Sn.jpeg** are shown below in Fig. 9.



**Fig. 9** Stress-strain curve beta phase (x) final melt at 300 K obtained with 2-phase method in Lava Wrapper. Atoms are colored with Common Neighbor Analysis (CNA).

**Appendix: Lava\_Generator.py as a stand-alone configuration generator**

**Lava\_Generator.py** itself can be used as a stand-alone python code to generate configurations, and thus can be useful outside of Lava Wrapper as well.

The following call generates FCC lattice with a lattice constant of 4.05, and c/a ratio of 1.00, and Z orientation of [111], replicated by 2 x 2 x 2 times along X, Y, Z direction:

python Generator.py latt FCC 4.05 1.00 orz 1 1 1 rep 2 2 2

Adding a “vac 10” to the above call will generate a (111) surface with a 10 Å vacuum (half on each side):

python Generator.py latt FCC 4.05 1.00 orz 1 1 1 rep 2 2 2 vac 10

The following call adds 1 vacancy to the input data file **Temp.data**:

python Generator.py Temp.data vacancy 1

Substitute vacancy with interstitial will add 1 interstitial atom to the input data file **Temp.data**:

python Generator.py Temp.data interstitial 1

The following call generates a 24x24 stacking fault mesh structures, based on a lattice constant of 4.05.

python Generator.py Stacking\_Fault 4.05 24 24

**Reference**

1. Mishin, Y., et al., *Interatomic potentials for monoatomic metals from experimental data and ab initio calculations.* Physical Review B, 1999. **59**(5): p. 3393-3407.